Laboratoire de Microélectronique
Faculté de Sciences Appliquées

Universite Catholique
de Louvain

A Study of the Application of Binary Decision Diagrams in Multilevel Logic Synthesys

Une Etude de l'Application des Diagrammes de Décision Binaire à la Synthèse de Circuits Digitaux en Logique Multi-Couches

Promoteur: C. Trullemans

Jury: E. Sanchez
P. de Marneffe

Ricardo Pezzuol Jacobi

Thèse présentée en vue de l'obtention du grade de Docteur en Sciences Appliquées

September 3, 1993

Table of Contents

1. Introduction 1
1.1 Design Process 1
1.2 Logic Synthesis 4
1.3 Two-level Logic Synthesis 7
1.4 Multi-level Logic Synthesis 8
1.5 Comments 11

2. Preliminaries 16
2.1 Definitions 16
2.2 Comments 21

3. Modified Binary Decision Diagrams 23
3.1 Reduced Ordered Binary Decision Diagrams 24
3.2 Modified Binary Decision Diagrams 28
3.2.1 Some Properties of MBDs 29
3.2.2 Logic Verification with MBDs 30
3.2.3 Logic Operations with MBDs 31
3.3 Other Types of ROBDDs 32
3.4 Comments 34

4. Input Variable Ordering Problem 35
4.1 Incremental Manipulation of MBDs 37
4.2 Variable Ordering: a Greedy Approach 43
4.3 Variable Ordering: a Stochastic Approach 45
4.4 Variable Ordering: the Exact Solution 49
4.5 Experimental Results 50
4.6 Comments 52

5. Minimization of Incompletely Specified Functions 54
5.1 Two-level Minimization with MBDs 56
5.1.1 The MBD Cover 61
5.1.2 The Method 61
5.1.3 Primality 62
5.1.4 Irredundance 65
5.1.5 Don't Care Minimization 66
5.1.6 Implementation 67
5.1.7 Experimental Results 69
5.2 Minimizing the MBD Size Using the Don't Care Set 70
5.2.1 The Subgraph Matching Approach 71
5.2.2 Subgraph Selection 74
5.2.3 Algorithms 77
5.2.4 Experimental Results 78
5.3 Comments 80

6. FPGA Synthesis 81
6.1 ACTEL Device Architecture 85
6.2 Subgraph Resubstitution 86
6.2.1 Resubstitution Algorithms 89
6.3 FPGA Mapping 94
6.4 Experimental Results 97
6.5 Comments 99

7. Multi-level Synthesis 102
7.1 Logic Decomposition 103
7.1.1 Functional Decomposition 104
7.1.2 Structural Decomposition 106
7.1.3 MBD Direct Decomposition 108
7.1.4 Experimental Results - Direct Decomposition 112
7.1.5 Path Oriented Decomposition 115
7.1.6 Experimental Results - Path Oriented Decomposition 122
7.1.7 Boolean Division 123
7.1.7.1 Boolean Divisors 126
7.1.7.2 Factorization Examples 128
7.1.8 Comments 131
7.2 Multi-level Minimization 133
7.2.1 Implicit Don't Cares 136
7.2.2 Node Minimization 137

7.2.3 Experimental Results 141
7.2.4 Comments 142
7.3 Technology Mapping 143
7.3.1 Technologic Features 146
7.3.2 The Matching Problem 147
7.3.3 Matching with MBDs 150
7.3.4 MBD Classification 151
7.3.5 MBD Matching 155
7.3.6 Matching Algorithms 158
7.3.7 Network Covering 159
7.3.8 Phase Optimization 163
7.3.9 Experimental Results 164
7.3.10 Comments 167

8. Conclusions and Future Work 168

9. References 174

Apendix: The LISP Prototype 185

Chapter 1

Introduction

A general view of the automated synthesis of digital circuits is presented. Some aspects of the logic synthesis are introduced. The proposed work is outlined.

In a few decades computers and informatics have deeply changed our daily life. The basis of this revolution is the simultaneous progress attained in several domains of science that are globally referred as microelectronics. Semiconductor technology, for instance, has been continuously reducing the minimum device's feature size. As a result, device density has been increasing at an $O(n^2)$ rate, which allows for a corresponding augmentation of circuit complexity. One of the challenges of digital design is to deal with such complexity and to explore the ever-increasing functionality provided by VLSI systems. New design methodologies have evolved, with intensive use of automatic synthesis tools. This work addresses one of the design automation fields, the logic synthesis.

1.1 Design Process

The design complexity of VLSI circuits is dictated by several interrelated factors. Minimum device's feature size, maximum chip area, maximum dissipated power and the maximum number of I/O pads impose physical limits to the circuit's capability. The design complexity increases as long as designers try to explore these technological limits in order to integrate large digital systems into a single chip.

Minimum feature size is by far the most important technological parameter. The number of devices per chip is as an $O(n^2)$ function of the feature size. Moreover, device's switching time varies as an $O(n)$ function of its dimension. Thus, decreasing the minimum feature size is responsible for an $O(n^3)$ increasing in the computing capability of integrated systems. Its

reduction from 50 microns in 1960 to 0.8 μ in 1990 gives an idea of the evolution of the device density and computing capability in last decades.

The amount of information involved in a VLSI project have grown at similar rate. Roughly speaking, a VLSI design consists in translating an abstract system specification into an intricate layout description to be sent to a foundry. When circuits were built with a small set of logic gates the designer's attention was concentrated in the layout details. As circuits became larger and larger, the designer started to build abstractions that hidden the mass of layout details and simplify the problem. The process continued to evolve and a set of abstraction levels was then developed that establish a smoother path from the initial description up to the masks. Each level presents a particular view of the design in terms of primitive components proper to that abstraction level.

Although the widespread use of abstraction levels, there is no consensus about a design and synthesis representation model. Little work was done in this area. Two similar approaches are presented in [Wal85] and [Gaj88] and have gained some acceptation. They proposed to divide the design representation into three domains: behavioral, physical and structural. The domains are further subdivided according to a set of general abstraction levels. Following the naming convention of [Gaj88], they are:

- system level
- algorithm level
- microarchitectural level
- logic level
- circuit level

The domains and their subdivisions can be graphically visualized in a tri-partite diagram called Y-chart (see figure 1.1). Each domain correspond to an axle in the diagram. Each abstraction level is drawn as a circle that intersects every axle. The intersection points define the respective domain's abstraction levels. One design process can be described in this diagram by indicating its transitions from the circuit specification up to the final layout over the diagram. A single transition is denoted by an arrow that connects two abstraction levels[1].

---

[1]It is interesting to note that the Y-chart could be further improved by adding a fourth axle to represent the time domain. In this case, the following correspondence could be depicted:

• system level	- inter-chip synchronization
• algorithm level	- scheduling of the operations
• microarchitectural level	- process synchronization
• logic level	- gate delays
• circuit level	- detailed devices timing

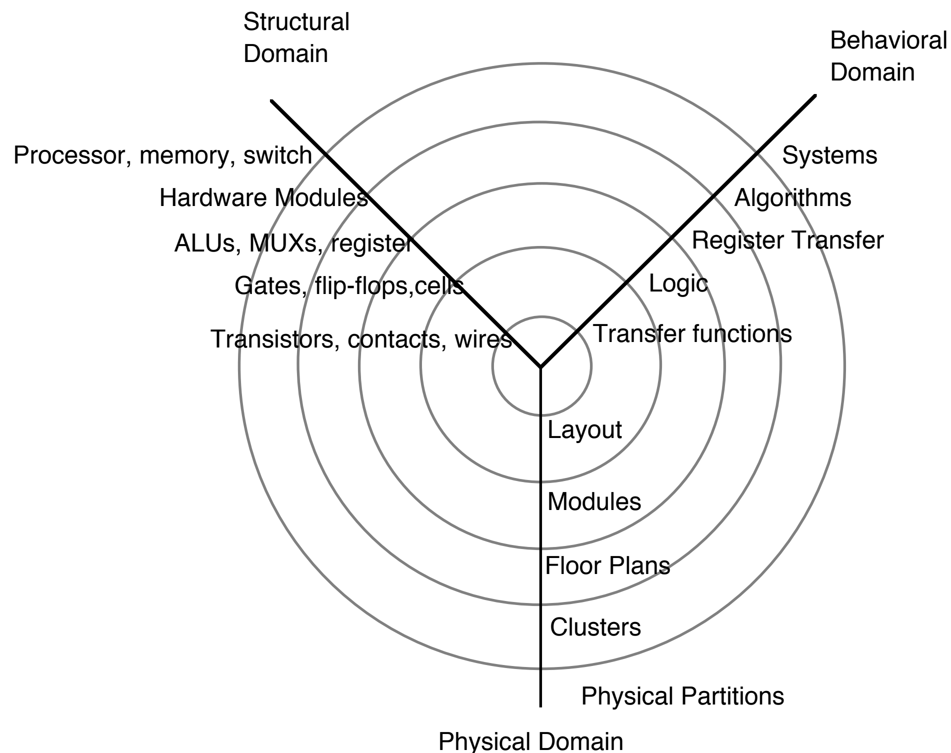


Figure 1.1. Y chart.

The Y-chart can be interesting to present a detailed description of a design or synthesis flow. A more resumed way to represent it is with data flow graphs. Indeed, this is the most used description method found in the literature. The synthesis process is presented linearly, as a set of transitions between general abstraction levels. The levels can vary from one author to another, but in general they are close to the division presented above. In this work we will use mainly the data flow graphs when discussing the synthesis process. A typical synthesis flow is shown in figure 1.2.

The input description is an algorithm written in a high level hardware description language. We have currently several ones, as Verilog, Silage, HardwareC and VHDL. The later deserves special attention as it was conceived to be the standard in this area. The behavioral synthesis consists in extracting from this initial description control and data flow graphs that are manipulated by scheduling and allocation algorithms to optimize the circuit architecture in terms of timing and size at a high abstraction level. The architectural synthesis takes these intermediate data and transform them into a more detailed register transfer level circuit representation. Operators are assigned to specific library modules and variables are associated to memory elements. Next step is the logic synthesis which provides detailed netlists for the combinational and sequential blocks. The final step is the layout generation, that yields the mask description to be sent to the foundry.

The automation of these tasks was subject of intense research in last decade. Systems that fully automate this process were first called *Silicon Compilers* [Cor88][Kow85][Jac89a][Rab88][Gin89]. This is a very hard problem, and feasible solutions can be found only under a set of severe restrictions (typically the definition of a target technology for a specific class of circuits).

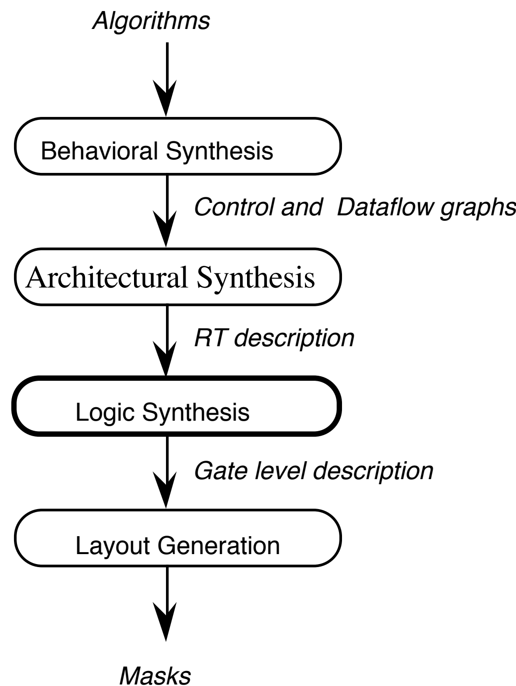


Figure 1.2. Synthesis flow.

The logic synthesis area - the subject of this work - can still be subdivided into two main domains: sequential and combinational logic synthesis. Sequential synthesis deals with the design of finite state machines (FSM), which combines memory elements with pure combinational logic blocks. Combinational blocks are built with memoryless devices that are interconnected in a cycle free way. In the literature, the expression *logic synthesis* is often associated to pure combinational blocks. In the other case, *sequential logic* or *finite state machine* terms are explicitly mentioned. We adopt this convention through this work, which concerns exclusively pure combinational logic.

1.2 Logic Synthesis

The logic synthesis is the task of generating an *acceptable* circuit design starting from a logic description of its functionality. An *acceptable* circuit must meet some design constraints, like surface on silicon, delay and testability. These requirements, derived from the architectural level or established by the designer, are incorporated to the synthesis process by the use of *cost functions*.

The logic circuit is modeled by Boolean functions. They provide a behavioral description that is the starting point for the circuit synthesis. The synthesis process adds structural information to the initial behavioral specification, producing a net-list in the target technology. This process can be represented in the Y-chart as depict figure 1.3. It is represented by a simple transition from the behavioral axis to the structural one. If the net-list is composed by logic gates the transition is done at the same abstraction level, as shown by arrow 1. If the net-list is composed by transistors, then the synthesis is represented by arrow 2 - a transition between two abstraction levels.

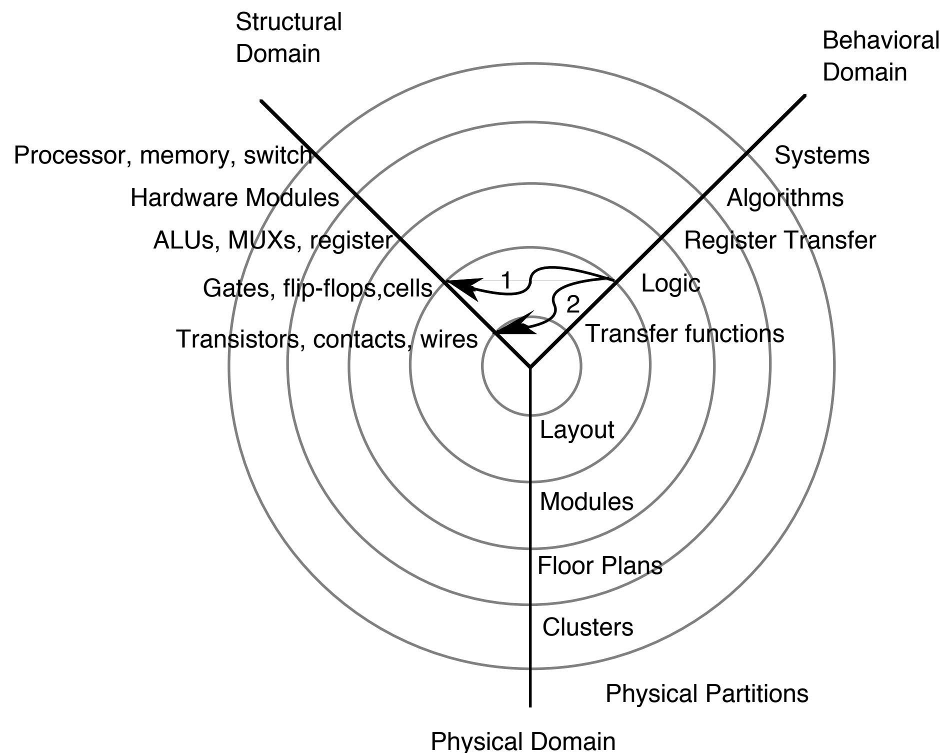


Figure 1.3. Y chart representation of the logic synthesis process.

In general, logic synthesis can be divided in two areas: two-level and multi-level synthesis. The term *two-level* indicates that any input signal in the circuit traverses at most two logic gates up to arrive to the output (supposing that the complement of the signals are directly available). The circuit is described by two-level expressions which are implemented in a regular and compact structure called Programmable Logic Array (PLA). The development of efficient methods for the minimization of two-level expressions and the existence of a economical circuit to implement them made the two-level synthesis very popular in the research community and also in industrial applications. It is still widely used for the design of logic circuits and it is offered almost by any commercial logic synthesis system.

Multi-level or *multi-stage* logic is a circuit were the signals can cross an arbitrary number of gates before reaching the outputs. The circuit, in this case, is described by a set of

interconnected subfunctions. Multi-level synthesis techniques have been studied for a long time. Its first application was probably for the design of switching circuits, but it was lagged behind two-level synthesis with the advent of PLAs. However, as integrated systems grow in complexity and new technologies were developed, multi-level design became an interesting alternative to implement large circuits that are difficult to implement with a single PLA. In the last decade we have seen a strong development of multi-level optimization techniques and multi-level circuits start to compete in terms of area and delay with PLA implementations and even with hand-crafted designs.

One of the reasons of the progress attained by multi-level design is the development of new design styles associated with new powerful layout synthesis tools. In fact, the logic synthesis methods are strongly tied to the design style the circuit will be implemented. This relationship can be exemplified by historical facts:

- the switching theory known a strong development [Sha49] [Lee59] when the relays and vacuum tubes were used as primitive devices to design logic circuits;
- the introduction of small scale logic gates stimulates the development of NAND/NOR [Dav69] [Die69] [Iba71]based synthesis methods;
- the development of regular structures like PLAs fostered the research in two-level logic minimization tools [Hon74] [Bra84][Dag86][Gur89];
- more recently, the emergence of complex cells based design encouraged the development of multi-level logic synthesis methods [Bra87][Bar86][Ykm89][Abo90][Mat88] [Geu85] [Bos87].

The sum-of-products form and its associated PLA target implementation is perhaps the most significant example of this interrelationship. The PLA structure is shown in figure 1.4.

It is composed by two interconnected matrices, one performing the logic AND operation of the input variables and the other executing the logic OR operations of the AND plane outputs. The columns of the AND plane are associated to the input literals. The rows represent the product terms. A dot indicates the electrical connection among the orthogonal wires. The columns of the OR plane stand for the function outputs. A multiple output function is implemented by specifying the connections in the AND/OR planes such that the first one contains all the cubes used by the functions and the second one describes which cubes are used by each output function. The minimization of the number of products in the sum-of-products representation implies the minimization of the number of rows in the PLA, which reduce its size. Thus, there is a direct association between the data structure and its implementation.

Next sections make a brief overview of two and multi-level synthesis to better situate the scope of this work.

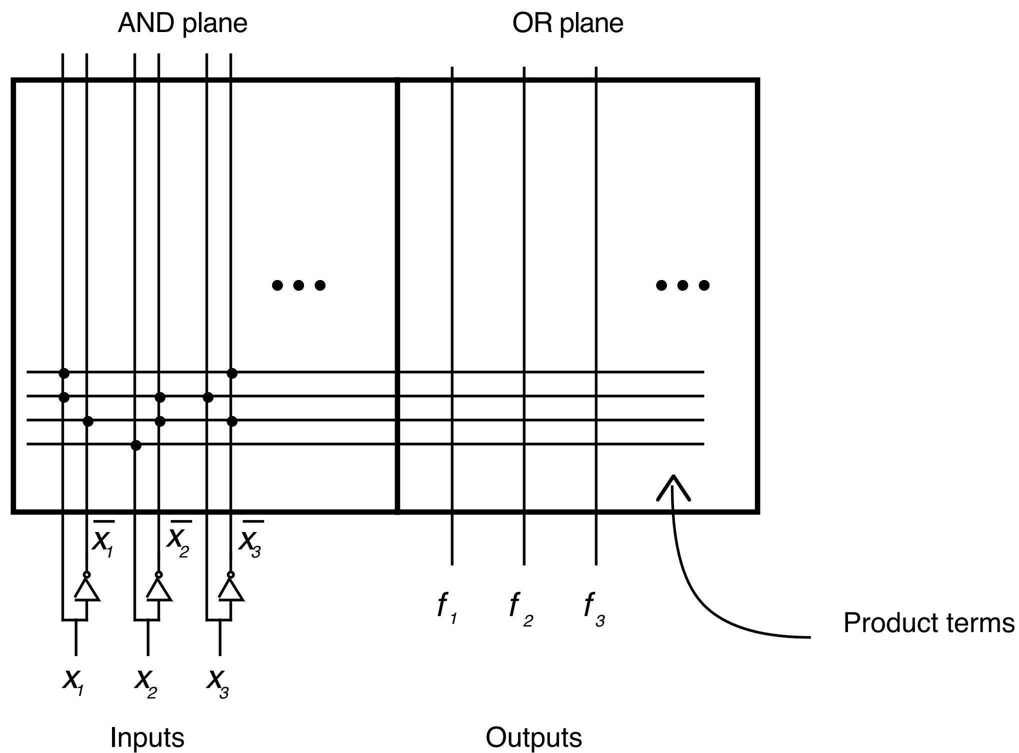


Figure 1.4. The PLA structure.

1.3 Two-level logic synthesis

Two-level minimization appears first as a general logic minimization tool to deal with truth functions. Quine [Qui52] laid down the basic theory that was adapted by McCluskey [McC56] for switching circuits applications. The resulting method, called Quine-McCluskey, basically consists in computing all prime implicants of the function followed by the selection of an optimal subset of them that covers all minterms. The main disadvantages of the method is that it uses minterms as starting point for minimization and requires computing all prime implicants. The number of minterms, for instance, grows exponentially with the number of input variables. Thus, the method is applicable only to small problems. Any way, it is a milestone in logic minimization and its principles can be found even in recent publications [Dag86].

The next important contribution in this area was MINI [Hon74], a multi-valued logic minimizer that introduced several significant improvements. The use of cubes instead of minterms as starting description, the reduction/expansion iterative process and many other heuristics that are the foundations of the next generation of logic minimizers can be found in MINI.

MINI concepts were the basis of ESPRESSO [Bra84], the most popular two-level minimizer up to date. ESPRESSO improved the heuristics proposed in MINI and developed new efficient algorithms that speed up significantly the logic minimization process. It is able to find optimal

or near-optimal solutions for incompletely specified Boolean functions with hundreds of inputs and outputs in a reasonable computing time. ESPRESSO has had several versions. The first one was in APL. This first version was quickly replaced by other written in C language. Next versions were ESPRESSO-MV (multi-valued) and ESPRESSO-EXACT [Rud86][Rud87]. The multi-valued program adopt the strategy of MINI, with new algorithms. ESPRESSO-EXACT computes all prime implicants, which allows it to find the best solution with respect to product count. It should be noted that the best solution for non trivial cases was first proposed in MacBoole [Dag86], which extended the techniques of Quine-McCluskey using efficient data structures and algorithms. However, these techniques are still restricted by the total number of prime implicants of the function, which is upper bounded by $3^n/n$, where n is the number of inputs.

Although well known, two-level logic minimization is still a research field. The use of two-level techniques in multi-level minimization lead to the development of new techniques to overcome some restrictions introduced by the multi-level context [Mal88]. More recently, a new method for the implicit computation of prime and essential prime implicants was presented by Coudert and Madre [Cou92] that allows the treatment of functions with huge sets of implicants. It was already extended to deal with multiple output functions with symbolic inputs [Lin92] and its use in two-level minimization is currently under research [Cou93].

1.3 Multi-level logic synthesis

Two-level logic and PLAs provide good solutions to a wide class of problems in logic design. However, it suffer from some limitations that stimulated the development of multi-level synthesis techniques.

The improvement of the quality of the circuits in terms of area and delay is one of such motivations. The long connections crossing active regions in large PLAs can lead to significant or unacceptable critical path delays. A set of interconnected active gates can produce faster circuits. Area reduction, however, is harder to obtain. It depends heavily on the layout generation tools. Standard Cells synthesis, for instance, can eventually lead to larger circuits than those designed with PLAs. This reflect some limitations of current multi-level techniques. But multi-level circuits are more flexible both in the topological and in the functional sense. They can be more easily adapted to different layout situations. They can also be used in data flow design, while PLAs are restricted to control logic.

Multi-level design can produce better solutions than PLAs essentially due to the degrees of freedom introduced by the increased potential for re-using subfunctions. This implies a larger solution space and, consequently, higher design complexity.

Several approaches have already been developed to tackle this problem. A systematic method that produces optimal solutions was proposed in [Law64]. It is not applicable for practical circuits, but it is interesting from theoretical viewpoint. *Rule-based* systems like LSS [Dar81] appears in the late '70's and worked directly on the circuit netlist, trying to improve the solution by performing local transformations. Each rule describes a subcircuit configuration and proposes an equivalent one that is optimized in some aspect. The optimization process consists in identifying and replacing subcircuits in the netlist. It has the advantage of taking into account technological features, but it is limited by its local nature and its lack of flexibility to support different technologies. The *algorithmic* approach starts its development at the beginning of the '80's. It is based on algorithmic transformations and is divided into two phases: a technology independent optimization followed by a technology mapping step. In the technology independent phase the minimization is performed on a general Boolean function representation of the circuit. Its global nature leads to significant reduction of the circuit logic complexity. Then the set of optimized subfunctions is mapped into gates of the chosen design style. In this step the particular features of each technology are considered.

Both approaches have been successful, but the algorithmic method usually produces better results due to the global nature of its minimization process and it is more efficient in terms of computing resources (time and storage). Several synthesis systems based on this technique have emerged [Bra87][Bos87][Abo90][Mat88][Mat89]. Some rule-based systems [Dar84] [Bar88] incorporated it in earlier phases of the synthesis, creating a mixed approach where the technology independent minimization is performed by algorithmic tools and the technology mapping is done using the rule-based approach.

The general multi-level synthesis flow is depicted in figure 1.5. The initial Boolean function is described by a sum-of-products form. It is minimized with respect to the number of cubes and literals. Then it is decomposed into a set of subfunctions, producing a Boolean network description. These subfunctions can still be simplified by computing redundancies that arise from the multi-level nature of the Boolean network. They are also named *internal don't cares* in the literature [Bar86] [Bra89]. These phases characterize the *technology independent* minimization. The next step in the mapping of the subfunctions into gates of the target technology, also called *technology dependent synthesis* or *technology mapping*.

The two phase synthesis approach works well for libraries of regular size, with no *exotic* gates, as those usually provided by standard cells and gate array design styles. Those gates can be easily decomposed in terms of the generic Boolean operators AND, OR and NOT, with are the basis of the generic Boolean minimization techniques. But it is not efficient to deal with implementation styles that are not adequately represented by a set of gates, like FPGAs. In this case, the programmable nature of FPGAs makes them difficult or even impractical to be represented by a library of cells, even applying Boolean function classification [Gol59]

techniques to reduce its size. New approaches that explore the very nature of these devices must be developed to overcome the limitations of the multi-level synthesis methods.

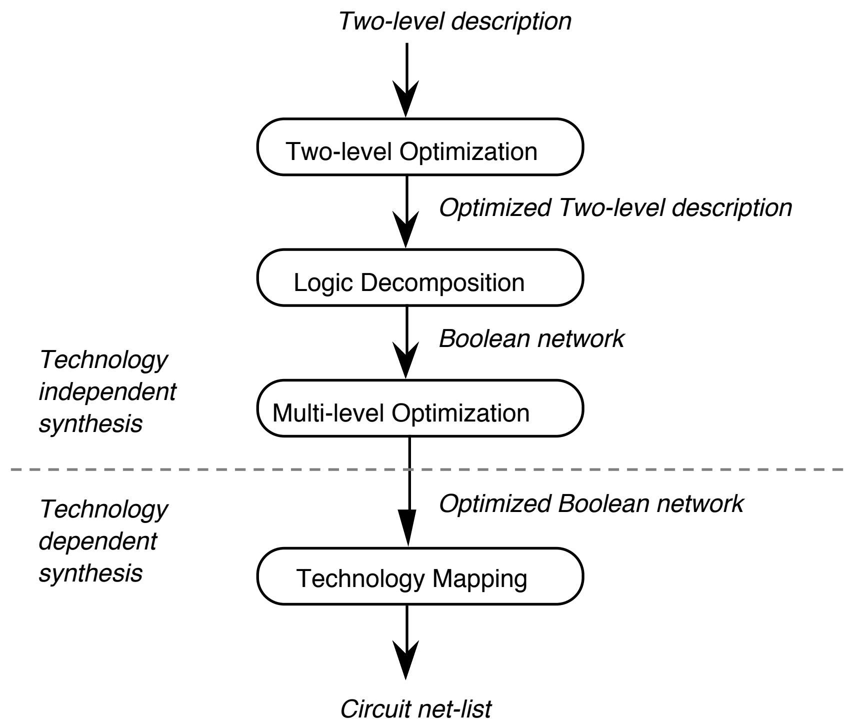


Figure 1.5. Multi-level synthesis flow

Several kinds of FPGAs emerged in last years. Two of the most important ones are the Look Up Tables from Xilinx [Xil92] and the multiplexor based cells from ACTELs [ElG89]. In both cases, all elementary cells have the same cost. The global circuit cost can then be estimated by the number of cells it is built with. Xilinx cells - also called Configurable Logic Blocks (CLBs) - implements any Boolean function with a small number of inputs (typically: 4 or 5 inputs). The synthesis problem reduces to the decomposition of a Boolean function into an as small as possible set of CLBs [Kar91a][Fil90][Wan92][Woo91][Bab92]. It does not matter the complexity of the individual subfunctions to be implemented by the CLBs. The point here is the relationship among subfunctions and its contribution in reducing the total number of blocks required to implement the circuit.

ACTEL multiplexor cells have already been modeled as a set of gates in a library based synthesis style [Mai88]. However, ACTEL cells can be more adequately treated if we adopt a Boolean function representation that reflects more closely its implementation cost in terms of two-input multiplexors. Binary Decision Diagrams (BDDs) [Lee59] [Ake78] [Bry86], in this case, seems a good solution due the direct correspondence that exists between a BDD node and a multiplexor cell. Indeed, some synthesis systems started recently to use BDDs in the synthesis with ACTEL cells [Kar91][Ben92][Mur92][Erc91] obtaining better results than those based on traditional multi-level synthesis techniques.

Xilinx and ACTEL are examples where the introduction of technology related factors in early phases of the synthesis and a good choice for the Boolean function representation improved the results. Indeed, there is a current trend in considering implementation factors during the synthesis stages. The influence of the connections in the final circuit layout surface have received more attention lately. In [Abo90] the complexity of the connections is modeled as a topological ordering of the variables in the circuit. In [Ped91] the position of the gates in the final layout is estimated by a simulation of the placement and routing during the technology mapping step. [Hwa89] tries to reduce the interconnection complexity by reducing the number of connections during the logic decomposition phase. The relative small improvements produced by these methods is a symptom of the difficulties to grasp the inter-relationship between the logical and the physical aspects of the design. The success in the FPGAs case is better explained by the nature of the new features introduced by the technology, which are closer to the logical aspects of the design than by the progress in modeling physical properties of the technology.

1.5 Comments

The diversity of possible implementations of logic circuits makes the research on new data structures and algorithms an essential step in the quest of optimal or near-optimal solutions. Growing research efforts in this direction gave rise to new interesting logic representations like Reduced Ordered Binary Decision Diagrams (ROBDDs) [Bry86] and If-Then-Else DAGs [Kar89]. Other non-usual logic descriptions as the Reed-Muller expressions has gained increasing attention recently due to its advantages in some particular applications [Sau92]. The point here is that each kind of logic representation can highlight certain functional properties that can be useful for the synthesis. In general, the adequacy of a data structure for a given application can be measured in terms of the speed of the logic operations that manipulate it and the amount of memory it requires. Linear functions, for example, are easily represented with Reed-Muller expressions while its sum-of-products space complexity grows exponentially with the number of inputs. Another advantages are those that favor some target technology, as sum-of-products and their PLA implementation. Some circuit properties, however, are not always easily identified and require an extensive exploratory work to be perceived. Indeed, every logic function representation may have useful properties for synthesis purposes and a lot of work must be done to discover them.

The present work is motivated by the research of new synthesis methods based on the use of ROBDDs to represent logic functions. The idea is not really new. Lee [Lee59] and Akers [Ake78] have already proposed the use of Binary Decision Diagrams (BDDs) in combinational logic synthesis. However, only after the modifications proposed by Bryant [Bry86] they gained more attention. Bryant's contribution was the introduction of a variable ordering constraint in the computation of the BDDs. The resulting graph is called Reduced Ordered

Binary Decision Diagram. The main advantage of ROBDDs is that they are canonical representations of Boolean functions and they can be very compact.

Initially, ROBDDs were considered just as an efficient representation of a truth table. In fact, it holds much more information. In this work, we develop an exploratory study of the application of Modified Binary Decision Diagrams (MBDs) in several phases of the logic synthesis problem. MBDs are multi-rooted ROBDDs where a third terminal value X is added to represent the don't care set. The main research focus is in multi-level logic synthesis. Two-level minimization techniques for MBDs are also addressed, mainly for its use in the multi-level context. As there is a wide range of possible design styles for multi-level circuits, we opted to tackle only two of them: the synthesis on multiplexor based FPGAs (ACTEL cells) and on library based designs, like standard cells. The use of multiplexor type FPGAs is a natural outcome of the choice of MBDs to represent the Boolean functions. Moreover, the flexibility of application and the fast turn-around time of FPGAs make them a major technology trend for the next years. Library based design, on the other hand, covers a large range of applications. Indeed, any design style that relies on a small to medium set of gates can be treated by these techniques. Among them, standard cells and gate arrays are two of the most important examples.

The strategy adopted in this work was to build a complete logic synthesis system. Its architecture is shown in figure 1.6.

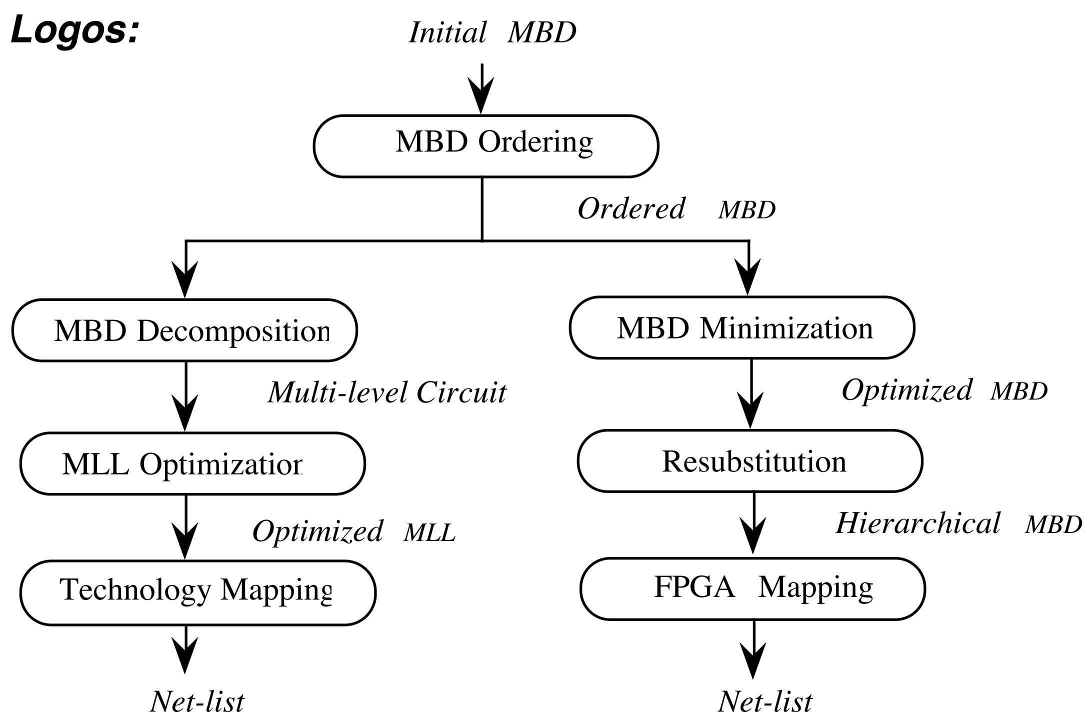


Figure 1.6. LOGic Optimization System architecture.

The starting point is a MBD of the function to be synthesized. Left branch shows the library based synthesis flow and the right one the FPGA synthesis flow. The first step, common to both synthesis, is to find a good input ordering for the MBD. The size of the MBD depends on its variable ordering, and a *good ordering* is an ordering that reduces the MBD size. In the library based synthesis the next step is the logic decomposition of the MBD into a set of subfunctions, which prepares its implementation in terms of the cells of the target library. The multi-level circuit produced by the decomposition is then minimized (if possible) by computing the don't care set of each subfunction and simplifying them using two-level minimization techniques. The simplified multi-level network is then mapped into cells of the target library, producing the final circuit net-list. This synthesis path is open, i.e., the user can start with a MBD or he may provide its own multi-level circuit and apply only the minimization and/or technology mapping steps. The FPGA synthesis is composed also by three main phases. The initial MBD is simplified with respect to its don't care set, if it exists. The idea is to further reduce the MBD size, which is a good estimation of the final circuit cost in terms of multiplexor cells. Then, a subgraph resubstitution phase is performed, where isomorphic subgraphs are identified and extracted in order to produce a hierarchical MBD. The resubstitution goal is to reduce the total MBD size, which is given by the size of the main graph plus the size of the extracted subgraphs. The final step is the mapping of the MBD into FPGA cells, that follows a completely different approach than that used in the library based synthesis. The result is a netlist of the circuit in terms of ACTEL cells.

The structure of this text follows the flow presented above. Each set of inter-related task is presented in one chapter. Each chapter has the following structure:

- chapter abstract, presented in italics below the chapter title
- brief introduction to the subject
- description of the method developed or implemented
- experimental results
- comments

As there is an intense research in multi-level synthesis today, we compare (when possible) the results obtained in each topic with the results provided by the literature. In the *comments* section the positive or negative aspects of the methods are analyzed. In the sequel we summarize the contents of each chapter.

Chapter 2 introduces some basic definitions used through this work.

In chapter 3 the data structure we have adopted is defined. It is an extension of ROBDDs to allow the representation of incompletely specified multiple-output Boolean functions and is called Modified Binary Decision Diagrams (MBDs). Its definition is developed step by step,

showing first the derivation of ROBDDs from Binary Decision Trees. Some properties of MBDs are presented as well as the extension of the elementary logic operations AND and OR to deal with the incompletely specified case.

Chapter 4 deals with the input variable ordering problem, which is a way to produce compacts MBDs by considering just the ordering the decision variables appear in the graph. A method for the reordering of the variables after the MBD is created is presented. It relies on an incremental technique called *swap*, that exchanges to adjacent variable in the MBD. We present three reordering methods based on *swap*. First, a greedy approach that produces fast results with an average reduction of 20% of the initial MBD size. Second, a stochastic method is shown that produces better results (30% reduction) at the expense of more computing time. Finally, an exact method is presented that can compute the best solution for relatively small cases.

Chapter 5 regroups the methods for the minimization of incompletely specified functions represented by MBDs. Two methods were developed, one that applies two-level simplification techniques in order to derive a sum-of-products from MBDs and another one that uses graph matching algorithms in order to reduce the MBD's size, used for FPGAs synthesis. The two-level based approach is used to simplify single-output incompletely specified subfunctions of the multi-level circuit. It is used in the multi-level minimization phase as well as in the technology mapping phase. In this case, it provides a sum-of-products expression that is useful to split complex subfunctions into a AND/OR tree. The subgraph matching method can be applied to both single and multiple output functions. It finds equivalent nodes in the MBD by matching the X terminal with MBD subgraphs. The result is a completely specified MBD that is smaller than the original one. We show that this approach produces better results than the application of the traditional two-level minimization for the reduction of the MBD size.

Chapter 6 describes the synthesis into multiplexor type FPGAs. The ACTEL cell architecture as well as the chip architecture are introduced. The minimization of the MBD with respect to the X terminal was already presented in chapter 5. The subgraph resubstitution technique and the MBD mapping are presented. Subgraph resubstitution consists in finding a particular kind of isomorphic subgraphs that can be replaced by a single node in the MBD. This means that they must start with a single root and have only two descendants. The resubstitution may further reduce the total number of nodes of the MBD. The mapping into FPGAs is done using a graph covering approach based on dynamic programming. Each subtree is optimally mapped and some pos-processing steps are presented to improve the solution.

Chapter 7 touch the library based multi-level synthesis. Three decomposition techniques are presented. The first one extract directly subMBDs of the MBD and replaces it by new variables. These are subgraphs that start at some MBD node and end at the terminals. Applying iteratively this kind of extraction transforms a multiple output MBD into a multi-level circuit where each

subfunction is described by a MBD. The second decomposition method is path oriented. At each step a node in the MBD is selected and all the paths that cross it are extracted and represented by separate MBD. This produces a logic division in the form $f = q \cdot d \vee r$. The third decomposition method is based on the Boolean methods. Next the satisfiability and observability don't cares are introduced. We use some known algorithms to compute them in order to simplify the node functions with the don't care based minimization presented in chapter 5. The last section describes technology mapping process that applies Boolean techniques to match subfunctions to gates of the target library. A fast symmetry detection method is presented that is used to classify Boolean functions in the matching phase. A flexible network covering algorithm allows the mapping of single as well as complex subfunctions. The mapping finish with a global phase minimization step that tries to reduce the number of inverters of the circuit by choosing the polarity of the each mapped subfunction (direct or complemented).

Chapter 8 discuss the results obtained and analyze some ideas of future research.

Chapter 2

Preliminaries

In this chapter we present the definitions of some basic concepts that are frequently used through this work.

Digital combinational systems can be seen as black boxes that receive a set of input signals and after some delay provide a set of output signals, established as a function of those inputs. To analyze the input-output transformation performed by the digital system it is necessary, first, to establish a mathematical structure that could model its behavior. Digital signals may assume only two meaningful distinct values (say, 0 volts and 5 volts). Their binary nature leads to a natural association with logical values *true* and *false* from formal logic. One of the first scientists that perceived this analogy was the American mathematician Claude Shannon in the late '30's [Sha38]. The structure he adopted was the Boolean algebra, conceived by the English mathematician George Boole around 1850 to model human reasoning[1].

2.1 Definitions

Definition 2.1. A *Boolean algebra* $\langle \mathbf{B}, \cdot, \vee, \bar{}\,, \mathbf{0}, \mathbf{1} \rangle$ is a mathematical structure composed by a set **B** which contains at least two distinct elements, **0** and **1**, on which are defined three operations:

- the *Boolean product* "·" (or *conjunction*);
- the *Boolean sum* " ∨ " (or *disjunction*);
- the unary operation called *complementation*, denoted by an upper bar;

[1]His work is reported in two books: *The Mathematical Analysis of Logic*, Cambridge, 1847, and *The Laws of Thought*, Cambridge, 1854.

A more precise definition of Boolean algebra should include the more general concepts of *algebraic structures*, *lattices* and *distributive lattices* and is out of the scope of this work. A good introduction to this subject can be found in [Ger82]. A digital systems oriented view is given in [Har65] and [Dav91a]. [Rut65] presents a good introduction to the lattice theory.

The set **B** denotes any set of elements that forms a Boolean algebra. The simplest one is **B** = {0,1}. A n-tuple of Boolean variables $\mathbf{B}^n = \{0,1\}^n$ forms also a Boolean algebra if the operations on this set are a component wise extension of the operations on **B**.

Definition 2.2. A *Boolean function f* is a mapping f: $\mathbf{B}^n \rightarrow \mathbf{B}$. Its domain (also denominated *Boolean space*) is the set of 2^n binary vectors {(0,0,...,0), (0,0,...,1), ..., (1,1,...,1)}. An *input vector* is represented by $\mathbf{x}_i = (b_1, b_2, ..., b_n)$, where $i = \sum_{j=1}^{n} 2^{j-1}.b_j$, $b_j \in \{0,1\}$.

Thus, a Boolean function is a set of ordered pairs in which the first element is an binary input vector and the second element is the constant **0** or **1**. If the domain **D** of a Boolean function f is $\mathbf{B}^n$ then f is said to be a *completely specified function*. If $\mathbf{D} \subset \mathbf{B}^n$ then f is an *incompletely specified function*, and is represented as $\tilde{f}$. For the set of points in DC = $\mathbf{B}^n \setminus \mathbf{D}$, the value of the function is undefined. DC is called the *don't care* set of f. The undefined values of the function are represented by the symbol '-'.

Proposition 2.1. There are 2^{2^n} completely specified Boolean functions of n variables.

Proof. The truth vector of a function with n inputs contains 2^n elements. For completely specified functions, to each element must be assigned a binary value. Thus, there are $2^{\text{size(truth vector)}} = 2^{2^n}$ different functions of n variables. ◊

Another way to denote an incompletely specified function $\tilde{f}$ is through its *characteristic sets* $\tilde{f}^{-1}(b) = \{\mathbf{x} \mid f(\mathbf{x}) = b, b \in \{0,1,-\}\}$. The characteristic set of $\tilde{f}^{-1}(0), \tilde{f}^{-1}(1)$ and $\tilde{f}^{-1}(-)$ are called, respectively, the OFF-set, the ON-set and the DC-set of $\tilde{f}$ and are represented by f_{off}, f_{on} and f_{dc}. Therefore, an incompletely specified function can be represented by a triplet $\tilde{f} = (f_{on}, f_{off}, f_{dc})$.

A *multiple-output Boolean function* is a mapping in the form:

$$f: \mathbf{B}^n \rightarrow \mathbf{B}^m, \text{ with } n \text{ inputs and } m \text{ outputs.}$$

In any Boolean algebra **B** the set **B** x **B** is a *partially ordered* set in which a binary relation $\mathbf{x} \leq \mathbf{y}$ is defined. The binary relation ≤ is *partial order relation* (that is, reflexive, anti-symmetric and transitive). Intuitively, if $\mathbf{x} \leq \mathbf{y}$ then for each **1** in vector **x** there must be a **1** in vector **y** at the same position. For example, (1,0,1,0) ≤ (1,0,1,1), but (1,0,1,0) does not relates to (1,1,0,0). Since Boolean functions can be represented by truth vectors, the set of all

functions of n variables forms a partially ordered set under the relation $\leq$. Using this relation, an incompletely specified function $\tilde{f}$ may be expressed by an interval $[f_{min}, f_{max}]$, and any function f that satisfies $f_{min} \leq f \leq f_{max}$ is *compatible* with $\tilde{f}$. Note that $f_{min} = f_{on}$, and $f_{max} = f_{on} \vee f_{dc}$.

The algebraic manipulation of Boolean functions relies on the use of variables to represent elements in **B**. A *Boolean variable* is a single coordinate in the Boolean space, and is represented by a lowercase character with an integer index: x_1, for instance. A *literal* is a variable x_i or its complement $\bar{x}_i$. $\mathbf{X} = \{x_1, x_2, ..., x_n\}$ is the set of input variables of a Boolean function. The representation of Boolean functions by means of variables, constants and operators introduces the concept of Boolean expression.

Definition 2.3. A *Boolean expression* is defined recursively as follows:

- any element $e \in \mathbf{B}$ is a Boolean expression;
- any variable $x_i \in \mathbf{X}$ is a Boolean expression;
- if F, G and H are Boolean expressions, then $(F \vee G)$, $(F \cdot G)$ and $\overline{H}$ are also Boolean expressions;
- there are no others Boolean expressions than those resulting from the application of the rules above.

The precedence of the Boolean operators is NOT, AND and OR, in this order. To simplify the notation, the parenthesis are dropped out where the meaning of the expression can be deduced from the precedence relation of the operators. The product operator '·' can be also eliminated and replaced by the juxtaposition of the literals, i.e., $x \cdot y = xy$. The Boolean function associated to a Boolean expression can be retrieved by evaluating the expression for each input vector.

The *support* of a Boolean function is the set of variables the function effectively depends on. For instance, if $\mathbf{X} = \{x_1,x_2,x_3\}$ and $f = x_1 \cdot x_2$, then support(f) = $\{x_1,x_2\}$.

Definition 2.4. The function $x^{(e)}$ is called *Boolean exponenciation*, and is defined by:

$x^{(e)} = 0$, if $x = \bar{e}$ and
$x^{(e)} = 1$, if $x = e$

It is component-wise extended to vectors: $\mathbf{x}^{(\mathbf{e})} = (x_1^{(e_1)}, x_2^{(e_2)}, ..., x_n^{(e_n)})$.

Let $\mathbf{v} = (v_1, v_2, ..., v_n)$, be a Boolean vector.

Definition 2.5. A *cube* (or *product term*) c is a function defined by a cluster of adjacent vertices in the Boolean space that can be expressed by a conjunction of literals.

$$c = \bigwedge_{i \in D} x_i^{v_i}$$

where $D \subset \{1,2,...,n\}$, $x_i \in \mathbf{X}$ and $v_i \in \mathbf{v}$.

Therefore, $l(x_i, 1) = x_i$ and $l(x_i, 0) = \bar{x}_i$. D is a subset of indices that identifies the support of c. For example, $c = x_1 \bar{x}_2 x_3$ is a Boolean expression that denotes a cube function with $support(c) = \{x_1, x_2, x_3\} \subset \mathbf{X}$ and $\mathbf{v} = (1,0,1, \ldots)$, the *polarity vector* of c. Since a Boolean expression always denotes a Boolean function, we will sometimes let the former term implicit and use explicitly the later one. Therefore, c is called a *cube* function or simply a *cube*. For all the vertices in the cube the function evaluates to **1**. For this reason it is also called an *implicant* of the function f, which can be denoted by:

$$c \leq f \text{ or } c \subseteq f.$$

The expression at left invokes the truth vector representation of the functions over the domain $\mathbf{B}^n$, while the right hand expression is more related to the description of a function as a set of ordered pairs.

The number of vertices contained by a cube depends on the number of literals that appear in the product and on the total number of variables of the function, $n = |\mathbf{X}|$. If we represent the support of a cube c by a subset $\mathbf{X}_c \subseteq \mathbf{X}$, then the number of vertices it contains is equal to $2^{|\mathbf{X}| - |\mathbf{X}c|}$. When the support of a cube is $\mathbf{X}$ then it denotes a single point in the Boolean space and it is called a *minterm*. The dual case is a sum of n literals (*anti-cube*) that represents a single vertex in the Boolean space where the function evaluates to **0**. It is called a *maxterm*.

One particular type of Boolean expression that is very important in logic synthesis is the *factored form*. Its definition is very similar to the Boolean expression's one.

Definition 2.6. A *factored form* is defined by a recurrent relation:

- any element $e \in \mathbf{B}$ is a factored form ;
- any literal x_i or $\bar{x}_i$ is a factored form;
- if F and G are factored forms, then $(F \vee G)$ and $(F \cdot G)$ are also factored forms;
- there are no others factored forms than those resulting from the application of the rules above.

A factored form is a Boolean expression where the complement operator has its application restricted to the input variables. Thus, $f = \bar{x}_1(x_2 + x_3)$ is a factored form but its complement, $\bar{f} = x_1 + \overline{x_2 x_3}$ is not. Any Boolean expression can be transformed into an equivalent factored form by applying the *De Morgan's* rules.

Theorem 2.1. (De Morgan's law) In a Boolean algebra,

$$\overline{x \cdot y} = \bar{x} \vee \bar{y} \text{ and } \overline{x \vee y} = \bar{x} \cdot \bar{y}$$

Proof. To prove the assertions above we must just to show that $(\bar{x} \cdot \bar{y})(x \vee y) = 0$ and $x \cdot y \vee (\bar{x} \vee \bar{y}) = 1$.

$(\bar{x} \cdot \bar{y})(x \vee y) = \bar{x} \cdot \bar{y} \cdot x \vee \bar{x} \cdot \bar{y} \cdot y = \bar{x} \cdot x \cdot \bar{y} \vee \mathbf{0} = \mathbf{0}$;
$x \cdot y \vee (\bar{x} \vee \bar{y}) = (x \cdot y \vee \bar{x}) \vee \bar{y} = (x \vee \bar{x}) \cdot (y \vee \bar{x}) \vee \bar{y} = \mathbf{1} \cdot (y \vee \bar{x}) \vee \bar{y} = y \vee \bar{x} \vee \bar{y} =$
$= y \vee \bar{y} \vee \bar{x} = \mathbf{1}$. ◊

Therefore, by successive application of the De Morgan's law we can transform a generic Boolean expression in a factored form. One type of factored form, the *normal forms*, play an important role in the synthesis of two-stage logic.

Definition 2.7. A *disjunctive normal form* (also called *sum-of-products or cover*) is a sum of products of literals (or cubes). A *conjunctive normal form* (also called *product-of-sums*) is a product of sums of literals (or anti-cubes).

Sum-of-products (SOPs) and products-of-sums (POSs) are also known as *two-level expressions*, *two-stage logic* or *two-level forms* because they can be implemented by two layers of logic gates.

Two-level expressions possess some interesting properties. A *sum-of-minterms* (*product of maxterms*) is a *canonical representation* of a Boolean function. A canonical form define a unique Boolean function, i.e., there can not be two different representations of the same function. This is very useful in some logic fields as Boolean verification, where the equivalence between two logic functions can be checked by comparing their logic representations. Unfortunately, the complexity of *sum-of-minterms* is $O(2^n)$ and it is of practical use only for very small functions. As we will see later, there are another types of representations, like BDDs, that also hold the canonical property and are applicable to practical problems.

Another important feature of logic representation forms is the existence of evaluation criteria that estimate their implementation cost. A good example are two-level expressions and their PLA implementations (see chapter 1, figure 1.4). Finding the smallest cover of a function is a NP-complete problem. An alternative is to find *good* or *acceptable* solutions using heuristic techniques. For two-level expressions, *prime* and *irredundant* covers form a set of acceptable solutions.

Definition 2.8. A *prime and irredundant cover* of a Boolean function *f* is a sum of *prime* cubes where no cube is covered by a proper subset of the remaining cubes of the cover. A *prime*

cube c_i is a cube no properly contained in another cube c_j of the function, i.e., $\nexists\ c_j \le f \mid c_i < c_j$.

If a cube is prime, no literal can be replaced **1** without making the cube intersect the function OFF-set. O prime implicant is called *essential* if it covers a minterm that is not covered by any other implicant of the function.

Multi-level logic expressions are an extension of the two-level concept where there is a non limited number of logic levels between the inputs and the outputs. The derivation of a multi-level expression from a Boolean function can be accomplished by means of *decomposition* techniques. A method that is the basis of several decomposition techniques is the *Shannon expansion* [Sha49]. The Shannon expansion of a function f with respect to a variable x_i is given by:

$$f(\mathbf{x}) = \bar{x}_i \cdot f(x_1, .., x_i{=}0,\ldots,x_n) \vee x_i \cdot f(x_1, .., x_i{=}1,\ldots,x_n)$$

The function $f(x_1, .., x_i{=}1(0),\ldots,x_n)$ is the *cofactor* or *residual function* of f with respect to x_i ($\bar{x}_i$) f_{x_i} ($f_{\bar{x}_i}$) . It is obtained by replacing all instances of literal x_i ($\bar{x}_i$) by 1(0) in the Boolean expression that denotes the function.

The mathematical model of multi-level expressions is the *Boolean Network* (BN). It is a direct acyclic graph (DAG) where to each node n_i is associated a subfunction f_i and a variable $y_i = f_i(\mathbf{X},\mathbf{Y},\mathbf{Z})$. $\mathbf{Y} = \{y_i\}$ is the set of intermediate variables and $\mathbf{Z} = \{z_i\}$ is the set of output variables of BN. Each edge e_i in BN is defined by an ordered pair (n_s, n_e). The first element is the start node and the second is the end node. The start nodes of the set of edges that have n_i as end node form the *fanin* nodes of n_i . The literals associated to the fanin nodes of n_i are the inputs of f_i. The end nodes of the set of edges that have n_i as start node are the *fanout* nodes of n_i . A set of nodes connected by edges is a *path* in BN. The number of edges in a path is the *length* of the path. Thus, the fanout nodes of n_i define the set of paths with length = 1 that start at n_i and the fanin nodes of n_i define the set of paths with length = 1 that end at n_i . The set of nodes that can be reached from n_i are the *transitive fanout* of n_i and the set of nodes that are in any path that ends in n_i form its *transitive fanin*.

2.2 Comments

The algorithms in this work will be presented in a Pascal-like pseudo-code. The commands of the language are printed in bold type characters, like "**begin**". Comments delimiters are either curly brackets "{}" or "/* */". The fields of variables of type *record* (*structure* in C) are accessed by the point '.' operator. Thus, *a.first* means the data called first inside the record a.

Only the algorithms that were effectively implemented are presented in pseudo-code, which is a summary of the programmed code. Description of algorithms from the literature with be present in textual way, for easy of reading.

Chapter 3

Modified Binary Decision Diagrams

This chapter describes the data structure that is the kernel of our the synthesis techniques. It is an extension of the Reduced Ordered Binary Decision Diagrams (ROBDDs), which we call Modified Binary Decision Diagrams (MBDs). Some useful properties of MBDs for logic synthesis are outlined.

Boolean functions play an important role in several areas of computer science and digital system design. It can be used to model or simulate both physical and abstract structures like digital circuits and logic reasoning. It has a large application in the automatic design and test of logic circuits. Consequently, the representation and the manipulation of Boolean functions are problems of major concern. In general, even simple tasks as testing if a Boolean expression is a tautology ($f = \mathbf{1}$) or checking if two Boolean expressions denote the same function require solutions to NP or co-NP complete problems. In the worst case, any known logic representation can require solutions of exponential cost for such problems. For most practical applications, however, a good choice for the logic representation and algorithms can avoid this exponential complexity.

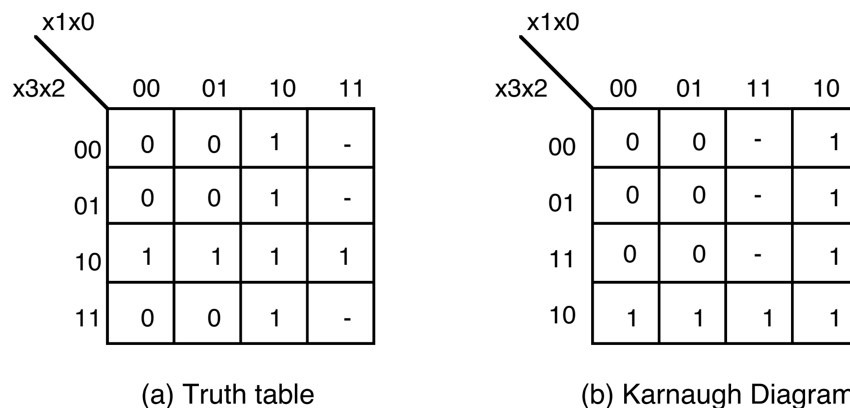
Several methods for the representation of Boolean functions have been developed. Truth tables, Karnaugh maps, canonical sum-of-products, prime and irredundant sum-of-products, Reed-Muller expressions, factored forms, Binary Decision Diagrams (BDDs) [Lee59][Ake78] and, more recently, Reduced Ordered Binary Decision Diagrams (ROBDDs) [Bry86] and If-Then-Else (ITE) Dags [Kar88][Kar89] are the most relevant examples among them. Truth tables, Karnaugh maps and canonical sum-of-products always produce exponential size representations, i.e., their size is proportional to 2^n, where n is the number of input variables. This is unacceptable for practical problems. Prime and irredundant sum-of-products and factored forms are two of the most popular logic representations that provide interesting

solutions for a variety of practical functions. Although their widespread use they suffer from serious drawbacks. First, some common functions, like odd and even parity ones, lead to exponential size representations. Second, certain simple operations as logic complementation can have exponential cost. Third, none of these representations are *canonical forms*, i.e., two different Boolean expressions can denote the same function. Thus, equivalence checking can be a time consuming task.

ROBDDs provide interesting solutions to most of these problems. It is a canonical form whose size keeps between reasonable limits for a large set of practical functions. These features allow the development of fast and efficient algorithms to perform Boolean manipulations.

3.1 Reduced Ordered Binary Decision Diagrams

We will introduce ROBDDs by following the evolution of logic functions representation. Truth tables have been certainly the first method developed to denote Boolean functions. From the computational viewpoint it can be described by a vector with 2^n elements or by a square matrix with $2^{n/2}$ entries. Each element of these structures is associated with a vertex in the Boolean space. The entries are represented by binary codes and each binary digit stands for an input variable of the function. A *Karnaugh* diagram is a particular case of the truth table where the row and columns' entries are *Gray* coded, i.e., two successive codes differ only in one bit. Figure 3.1 gives an example of the same function represented by a truth table and a Karnaugh diagram.



(a) Truth table

(b) Karnaugh Diagram

Figure 3.1. The truth table and the Karnaugh diagram representations

Karnaugh diagrams provide an useful notation for small functions due to the easy of visualizing cubes in it. We shall adopt them when discussing Boolean functions properties but using a different notation. Zero values are not indicated, and the sub-regions where the variables have value **1** are indicated as depicted in figure 3.2.

Another way to describe a truth table is by means of graph based structures, like trees and DAGs.

Definition 3.1: A *binary decision tree* (BDT) is a direct, acyclic and connected graph, where each node has at most two *successor* and only one *ancestor*. There are two types of nodes: *non-terminal* (or internal) and *terminal* (or leaf) ones. Terminal nodes have no successors and represent Boolean constant functions. Each non-terminal node n_i is associated with a decision variable x_i . Let $low(n_i)$ and $high(n_i)$ be the n_i's successors. The Boolean function denoted by n_i, $f^{n_i}(X)$, can be described by the following Boolean expression:

$$f^{n_i}(X) = \bar{x}_i \cdot f^{low(n_i)}(X) \lor x_i \cdot f^{high(n_i)}(X)$$

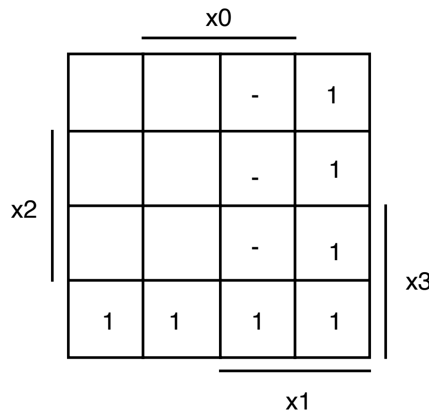


Figure 3.2. Karnaugh diagram notation

To get the value of the function at one vertex we must traverse the tree from the root up to a terminal. The path is defined by the values of the input variables. If the variable associated with one node evaluates to **1** then the high branch is added to the path. If the variable evaluates to **0**, then the low branch is taken. The cost of a BDT is, of course, proportional to 2^n.

A significant improvement in the Boolean function representation was proposed by Akers [Ake78]. He suggested the application of some simplification rules to reduce the complexity of a BDT, transforming it into a direct acyclic diagram.

Definition 3.2: Two nodes are *equivalents* in a BDT either if they are the same terminal or if they are associated with the same input variable and both their *low* successors are equivalents and their *high* successors are equivalents.

Definition 3.3: A node n is *redundant* in a BDT if low(n) is equivalent to high(n).

Definition 3.4: A *Binary Decision Diagram* (BDD) is a direct acyclic graph derived from a BDT where *redundant* nodes are deleted and *equivalent* nodes are merged.

The main feature of a BDD is that its size complexity is much smaller than BDTs for a large class of functions. The application of simplification rules saves a lot of storage space. For instance, while a BDT has 2^n terminal nodes the BDD needs only two leaves to represent the logic values **1** and **0**. An illustration of the derivation of a BDD from a BDT is presented in figure 3.3. A "d" indicates nodes that will be deleted and a "m" indicates two nodes that will be merged. The decision variables are shown inside the nodes they control.

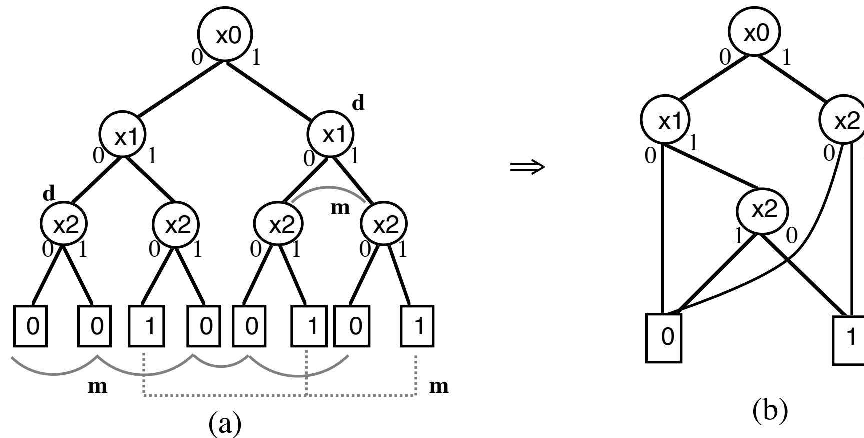


Figure 3.3. (a) binary decision tree (b) binary decision diagram

BDDs were shown to be an efficient structure for the analysis, simulation and test of digital circuits. Akers [Ake78][Ake79] gives several examples of such applications for well known combinational and sequential devices. He states, without proof, that even thought in the worst case the size of a BDD can be exponential with respect to the number of inputs ([Lee59]), for almost all common digital devices this number grows *linearly* with *n*. Unfortunately, the examples and the methods proposed are not oriented to computer implementations and are more adequate to pencil and paper investigation, which restricts its application to small problems. Another problem is the complexity of logic operations performed with BDDs. How to combine two BDDs with an AND operator? A possible solution is to connect the two graphs in series with respect to the **1** terminal, but it will be a waste of data space because there will be no sharing of subfunctions. Logic equivalence and tautology checking are examples of other logic operations frequently performed in digital analysis and synthesis that can be quite complex to implement with BDDs.

These drawbacks were overcame by the introduction of an ordering restriction for the input variables [Bry86], which lead to the ROBDD structure.

Definition 3.5: A *Reduced Ordered Binary Decision Diagram* is a BDD in which the input variables are ordered in such a way that in every ROBDD path each variable appears only once and in the same relative order.

Every input variable is associated with an integer index which indicates its position in the ordering. In the same way, each node *n* in the ROBDD has an attribute *index(n)* that indicates the variable it is associated with. *Index(n)* grows from the root to the leaves of the graph. Thus, in each ROBDD path the indices of the nodes must appear in crescent order. As variables appear only once in a path, the function associated with a node n with index(n) = i, $f^{n_i}(X)$, can be described by the Shannon expansion:

$$f^{n_i} = \bar{x}_i \cdot f^{n_i}_{\bar{x}_i}(X) + x_i \cdot f^{n_i}_{x_i}(X)$$

It is curious to note that ROBDDs had already been obtained before Bryant. One example is the work of Thayse [Tha82][Tha84] with his *P-functions*. In this work, the author proposes some methods for the synthesis of binay decision programs. One of them, the *simple* BDD synthesis, effectively leads to the construction of a ROBDD. Bryant, however, was the first to perceive the interest of ROBDDs as a logic representation form.

One of the main properties of a ROBDD is that it is a *canonical* representation. This means that if two Boolean functions are equivalent their ROBDDs are isomorphic. Logic equivalence, thus, is reduced to checking if two DAGs are isomorphic, which is $O(s)$, where s is the size of the graph. Another useful outcome of the ROBDD canonicity is that tautology checking becomes trivial: the graph must be the **1** terminal. Complementing a ROBDD consists in exchanging the values of its terminals, which is also $O(s)$. In general, operations involving two ROBDDs are $O(s_0.s_1)$, i.e., proportional to the product of their sizes. Figure 3.4 shows the graphs for some well known logic functions.

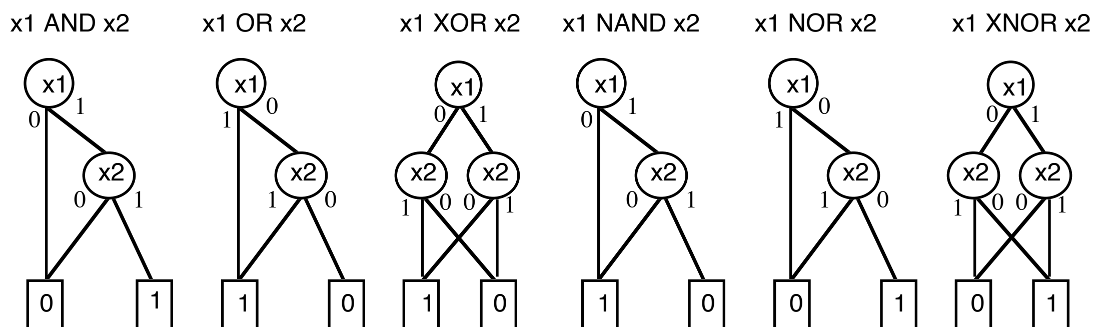


Figure 3.4. ROBDDs of some simple functions.

The algorithms for the construction and manipulation of ROBDDs are reported in [Bry86]. They rely on two main functions, *apply* and *reduce*. *Apply* combines two graphs using a binary operator and then *reduce* is called to put the ROBDD in its canonical form by deleting all redundant nodes. More recently, in [Bra90] a new method was developed that was shown to be more efficient than the *apply/reduce* technique. The logic operations are based on the *ite* (if-

then-else) operator that combines three ROBDDs without need of the *reduce* operation. It is defined as follows:

$$ite(f,g,h) = f \cdot g \ \vee \ \bar{f} \cdot h$$

It is the same operation performed by a ROBDD node, the basic building block of the diagram. The difference is that *ite* takes three functions as parameters, each one described by one ROBDD while a ROBDD node implements the same function but with the *f* parameter replaced by a single variable.

The node operation can be represented by a triple $(x_i,\ f_{x_i},\ f_{\bar{x}_i})$, where x_i is the variable associated with node n_i. Let $z = ite(f,g,h)$. The ROBDD of z is built using the following recurrent formulae:

$$z = (v, ite(\ f_v, g_v, h_v), ite(\ f_{\bar{v}}, g_{\bar{v}}, h_{\bar{v}})\ )$$

A more detailed description of the method can be found in [Bra90], including the main data structures and computing techniques to improve the algorithm's performance. In this work we have first developed a prototype system in Common Lisp that uses the *apply*/*reduce* method. A second version of the system was re-coded in C++ and a mixed approach was adopted, using *ite* as well as *reduce* in some cases. The rationale for these decisions and others are presented in the last section of this chapter.

3.2 Modified Binary Decision Diagrams

Consider the problem of dealing with multiple output incompletely specified functions. One can handle multiple outputs by assigning a ROBDD to each output function. This is not an efficient solution because there is no sharing between subfunctions from distinct outputs. For example, a function with m outputs would have $2 \cdot m$ terminal nodes. In the same way, a incompletely specified function could be described by two distinct ROBDDs, one for f_{on} and another for f_{dc}, with the same limitation. Thus, in the general case a m output function will require $2 \cdot m$ ROBDDs.

To simplify the representation complexity we have made the choice of representing all these data in a same graph, the Modified Binary Decision Diagram (MBD).

Definition 3.6: A *Modified Binary Decision Diagram* is a ROBDD where multiple roots are allowed to appear to represent multiple-output functions and a third terminal value X can be also added to the graph to denote the don't care set.

Thus, in a MBD there is an explicit representation of $(f_{on}, f_{off}, f_{dc})$ of multiple output functions. An advantage of this representation is that it is always more economical to merge MBDs into a

single diagram than to keep them separated, due to the subgraph sharing. An example of the MBD of a multiple output incompletely specified function is shown in figure 3.5.[1]

Note that the added features are optional and the ROBDD is a particular case of the MBD. By consequence, when we refer to a MBD herein it is not implied that it denotes an incompletely specified and/or multiple output function. This shall be clear from the context, although sometimes the term ROBDD will be used instead to stress that it is the case of a single output completely specified function.

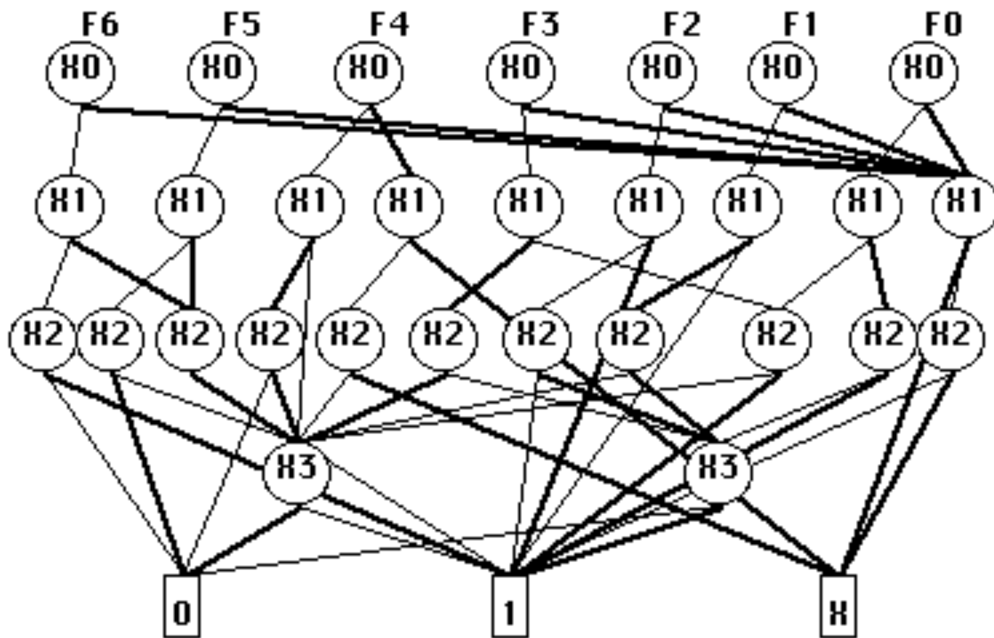


Figure 3.5. Example of a MBD.

3.2.1 Some Properties of MBDs

The MBD can be thought as a compact graph representation of a truth table. Besides being more compact, it presents several interesting properties derived from its graphic nature. The properties presented are well known and may be found in some previous works on BDDs and BDTs as [Ake78] and [Cer78]. For instance, each path in a MBD denotes an implicant of the function. The value of the function at the vertices contained in the implicant is given by the terminal node connected to the path. This is illustrated in figure 3.6, where a path in the MBD and its associated implicant in the Karnaugh diagram are indicated.

[1]The figure 3.5 was generated automatically by a program and does not follow the notation adopted in this work. The *one* (or *then*) branches are drawn with thicker lines, output functions are represented by literal ‘F’

More interesting from the minimization viewpoint is the relationship between MBD paths and a cover for the function.

Theorem 3.1: The set of all paths that connect the root of the MBD to the **1** (**0** or **X**) terminal, called ON (OFF or DC) paths, defines a disjoint and irredundant cover for the ON (OFF or DC) set of the function.

Proof: First, let us assume that the MBD is generated from a BDT representation of the function. Then each ON path in the BDT defines a minterm of the function. The set of all ON paths in the BDT forms a cover for the ON-set of the function. As the reduction operation performed on the BDT removes only redundant information and does not eliminate any minterm, the set of ON paths to the **1** terminal in the MBD is still a cover for the ON set of the function. They are disjoint because any pair of paths must differ at least in one variable, that appears in opposite phases in each path. It is a irredundant cover because all implicants are essential and, thus, no implicant can be covered by a combination of the other ones. ◊

Disjoint covers are generally a good starting point for two-level minimization [Hon74]. Moreover, a disjoint cover can be directly converted into a Reed-Muller expression, which is useful for the synthesis of linear functions.

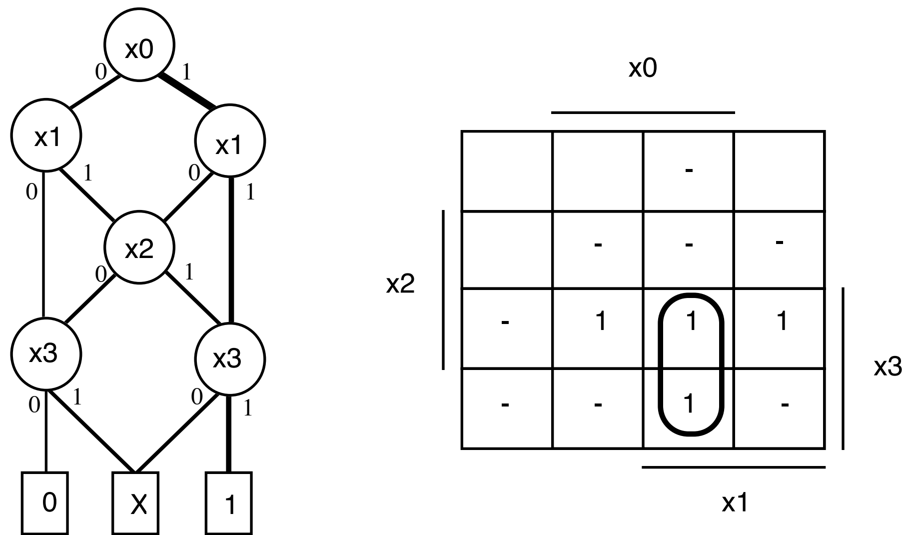


Figure 3.6. Relationship between MBD paths and implicants of the function.

As in BDTs, the simulation of the function for a given input vector (also called satisfiability) can be easily executed by traversing the graph and selecting the low or high branch at each node according to the value of its associated variable (0 or 1, respectively) up to reaches a terminal. The total number of minterms for the ON (OFF or DC) set can be computed in a single traversal of the diagram, by keeping track of the number of variables in each path connected to the **1** (**0** or **X**) terminal. If a path have k variables then the number of minterms it contains is

2^{n-k}, where n is the total number of variables. The set cardinality is given by the sum of the minterms in all paths. For further properties of ROBDDs the reader can refer to [Bry86], [Min90] and [Lia92] among others.

3.2.2 Logic Verification with MBDs

The inclusion of the don't care set in the MBD saves not only space but also computing time for some logic operations. We show here how logic verification is simplified by the presence of the **X** terminal [Cal91].

In order to verify if two single output incompletely specified functions f and g are equivalent, it is necessary to prove that $f_{on} \cap g_{off} = \varnothing$ and $f_{off} \cap g_{on} = \varnothing$. Using ROBDDs, this is done by building four graphs, one for each on-set and one for each dc-set. Afterwards, the *apply* [Bry86] procedure is used to verify the following equality:

$$(g_{on} \rightarrow (f_{on} \vee f_{dc})) \cdot (f_{on} \rightarrow (g_{on} \vee g_{dc})) = \mathbf{1}$$

If the expression is found to be a tautology, f and g are equivalent. This is the approach used by Malik et al. [Mal88]. On the other hand, using MBDs only two graphs must be built, one for f and another for g. The equivalence is performed in a one step by extending the MBD isomorphism checking to deal with **X** terminals. In is worthy to extend definition 3.2 to the case of MBDs.

Definition 3.7: two nodes in a MBD are X-equivalents if either:

- they are the same terminal;
- one of the nodes is the **X** terminal.
- they are associated with the same input variable and both their *low* successors are X-equivalents and their *high* successors are X-equivalents.

Some experimental results comparing the two approaches with respect to total number of nodes and execution time for logic verification can be found in [Cal91].

3.2.3 Logic Operations with MBDs

All logic functions with two variables can be implemented with the *ite* operator [Bra90], by an adequate choice of its function parameters. For example, consider the case of AND, OR and NOT functions:

ite(f, g, h) = $f \cdot g \vee \bar{f} \cdot h$
AND(f, g) = ite(f, g, **0**)
OR(f, g) = ite(f, **1**, g)

NOT(f) = ite(f, **0**, **1**)

The NOT operation, however, is more efficiently realized by just inverting the value of the terminals.

In the case of MBDs, the algorithms that manipulate ROBDDs should be adapted to deal with the **X** terminal. For constant Boolean values the results of the application of logic operators AND, OR, NOT are shown in figure 3.7.

The logic operation of a MBD with the **X** terminal results in a new MBD with the terminal's values changed according the tables above, as illustrated in figure 3.8. This means that, when recursively traversing two MBDs to perform a logic operation, if a **X** terminal is found, recursion proceeds in the other MBD up to the leaves. In this case, an explicit reduction of the MBD may be required if the **X** terminal is duplicated.

AND	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

OR	0	1	X
0	0	1	X
1	1	1	1
X	X	1	X

NOT	
0	1
1	0
X	X

Figure 3.7. Logic operations with don't cares

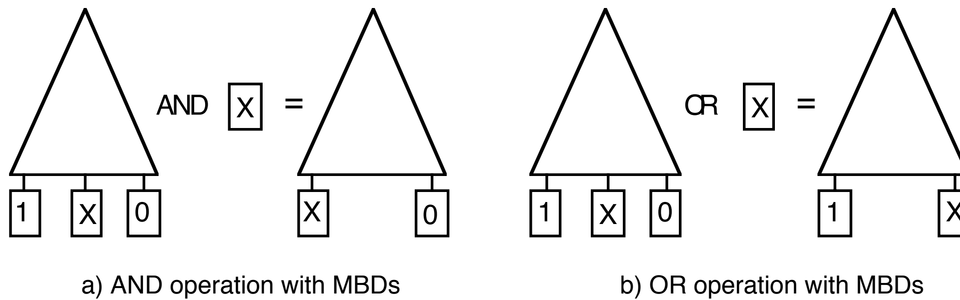


Figure 3.8. Logic operations with MBDs.

3.3 Other Types of ROBDDs

MBDs implement two kind of extensions to the original ROBDD from [Bry86]: the don't care terminal and the multiple roots. Several other alternatives can be found in the literature. Akers [Ake78] had introduced the concept of *complement edges*. It follows from the fact that one can very easily find the complement of a BDD by just complementing the terminal's values. The rest of the BDD remains unchanged. Thus, instead of representing two subfunctions f and $\bar{f}$ with two different subgraphs, we can use a single graph for f and indicate that some node uses $\bar{f}$ by adding a tag in the edge that connects them. The tag can be seen as an embedded NOT operation applied to the function denoted by the node. This extension was implemented in

some ROBDD packages like [Bra90], and it was reported that it can in average reduce the size of the graph by 7%. However, there is an overhead of introducing a tag on each edge and also an additional processing step to keep the graph canonical with respect to the tags, which is done by reorganizing them in such a way that the *high* (or *then*) branch of a node is always untagged. In [Bra90] the tag is stored in the lowest bit of the node address with no increase in memory space, but in machines that have not this capability it must be explicitly represented, which increases the space needed to store the graph.

In [Min90] Minato presents a set of extensions to ROBDDs that result in a new data structure named Shared Binary Decision Diagrams (SBDDs). Don't cares can be included in the graph either as a third terminal value or as an especial input variable D. If D appears as the top variable, we have a representation of type f_{min} (D = 0), f_{max} (D = 1). If D appears as the bottom variable, it is equivalent to a third terminal value. The term *shared* indicates, of course, a multiple output graph. A more elaborate feature is the *Variable Shifter*. The index of a node is not unique, but relative the father's index plus an increment associated with the edge. This can reduce the size of the graph in some cases, but no comments are made on the complexity of the algorithms that manipulate these kind of graphs. SBDDs use also the *complement edge* attribute and proposes another variation called *Input Inverter*. In this case, the edge tag indicates that the low and high pointers of a node are exchanged, which is equivalent to complementing the variable. A further enhancement is the combination of the graph with truth tables. A function with 4 variables can be represented by a binary vector of 16 bits. This corresponds to a single word in a computer and is more economical than using set of nodes. Of course, there is no sharing of subfunctions with less that 4 variables.

Another interesting improvement is the so called *strong canonical form*, proposed by Karplus [Kar89]. In this case two identical functions must have the same address in the memory. The idea is that when a new ROBDD is needed the system checks if there is an equivalent one already computed and, in this case, the later is returned. The main advantages of this approach are the reduction of the graph size and the transformation of the Boolean verification problem in a simple comparison of two pointers.

In [Bra89a] a reference is made to multi-valued ROBDDs. This means that a node can have more than two successors, one for each value of the correspondent multi-valued variable. No further details are provided, but this kind of representation could be directly used in multi-valued logic, expectedly with an advantage equivalent to that provided by ROBDDs for Boolean functions.

Finally, a generalization of ROBDDs, called If-Then-Else DAGs (ITEs), was proposed by [Kar89]. He replaces the *decision variable* of the ROBDD nodes by an arbitrary Boolean

expression that is itself described by another ITE. It presents some advantages with respect to ROBDDs in certain cases and may be another interesting subject for future research.

3.4 Comments

In this chapter we introduced the data structure used through this work, the MBDs. It extends the original ROBDD structure by allowing multiple roots and the representation of the don't care set in the graph by means of a third terminal value **X**. MBDs improve the performance and compactness attained with the ROBDD representation for the treatment of multiple outputs incompletely specified functions. An example was given for the logic verification problem. We have shown also how to extend some elementary logic operations to deal with the **X** terminal case.

MBDs implement two out of several possible ROBDD generalizations. Among those not supported by MBDs presented in this chapter, the complement edges and the strong canonical form are surely the most important ones. We present now some reasons for not using them.

Most systems use ROBDDs basically for logic verification purposes. In general they perform only elementary logic manipulations as function composition and the logic operations AND, OR, NOT, XOR, etc. In these cases, complemented edges and, specially, strong canonical forms can be interesting to improve the performance of the algorithms and to reduce the storage requirements. In our case, however, these attributes will be counterproductive due to the nature of the graph manipulations performed by our system. The strong canonical form can be seen as a large multiple output MBD that holds all possible functions in the system. If we swap two variables in a small function, we must in fact update all the nodes in this large MBD, because we can not have functions with different orderings. In the technology mapping phase this will be particularly ackward, because each subfunction in the Boolean network is represented by a local MBD that has its own ordering. The classification of these subfunctions generated MBDs with different orderings, which will be not possible with the strong canonical representation. For the case of complement edges, they require an additional external bit along with the node address, which increases the storage requirements and introduces alignment problems (a pointer + a bit). Worse yet, the algorithms for two-level minimization can not be applied to complement edges MBDs because they generate intermediate sum-of-products whose complement will be too complex to compute.

Chapter 4

Input Variable Ordering Problem

In this chapter the problem of finding good input variable orderings for MBDs, i.e., orderings that leads to smaller graphs, is discussed. A new technique called incremental manipulation is described which produces new orderings by exchanging adjacent variables in the MBD. Three methods for the search of new orderings are presented: a greedy reduction, a stochastic reduction and the exact solution, all of them based on the incremental technique.

The MBD's size is very sensitive to the ordering of the decision variables, at least for most practical functions. It should be emphasized that this is valid for *practical functions*, which is an empirical outcome of the research in this area. Indeed, it was proven in [Lia92] that, from the theoretical point of view, ROBDDs for general Boolean functions *are not* sensitive to the variable ordering. The general case considers Boolean functions up to an infinite number of inputs and has little correspondence to practical applications.

The relationship between variable ordering and the graph size is exemplified in figure 4.1 for the function $x_1x_2 \vee x_3x_4 \vee x_5x_6$. In this case, the natural ordering $<x_1,x_2,x_3,x_4,x_5,x_6>$ gives the smaller graph, which is shown at left. The worst ordering is shown in the diagram at right and consists in taking one variable of each cube at time, i.e., $<x_1,x_3,x_5,x_2,x_4,x_6>$. The problem can be interpreted by considering the MBD as a functional processor that tries to find the result of an expression by examining the values of its successive decision variables. In the former case, the processor needs only to keep track of the value of the preceding cube. If the cube does not evaluates to **1** then we examine the next cube. In the later case, the processor must store the first three arguments before starting to evaluate the possible function result. This ordering problem is valid for any sum of products where the cubes have disjoint supports.

The size of a MBD (ROBDD) is a very important factor because it not only dictates the amount of memory required to store the logic functions but also directly affects the speed of the logic manipulation algorithms. In some cases, for complex functions a bad variable ordering can lead to a huge diagram and the machine may run out of memory. Nonetheless, the same function with a good variable ordering could be represented by a feasible diagram. For those reasons the research on the variable ordering has received a lot of attention recently. The exact solution requires the generation of all possible orderings and the selection of one of the orderings that results in the smaller diagram. The complexity of the exact solution is $O(n!2^n)$, where $n!$ represents all possible permutations of the variables and 2^n is due to the time to build the graph for each ordering. This, of course, is impractical even for medium size problems ($|\mathbf{X}| \leq 10$). A better algorithm to find the exact solution was proposed in [Fri87], that exploit some ROBDD's properties to prune the space search and attain a cost of $O(n^2 3^n)$. This is also too time consuming for practical problems, but it defines a upper bound for the cost of the best solution. The complexity of the problem stimulates the search of heuristic solutions that could produce good orderings, i.e., that avoid the exponential explosion of diagram's size.

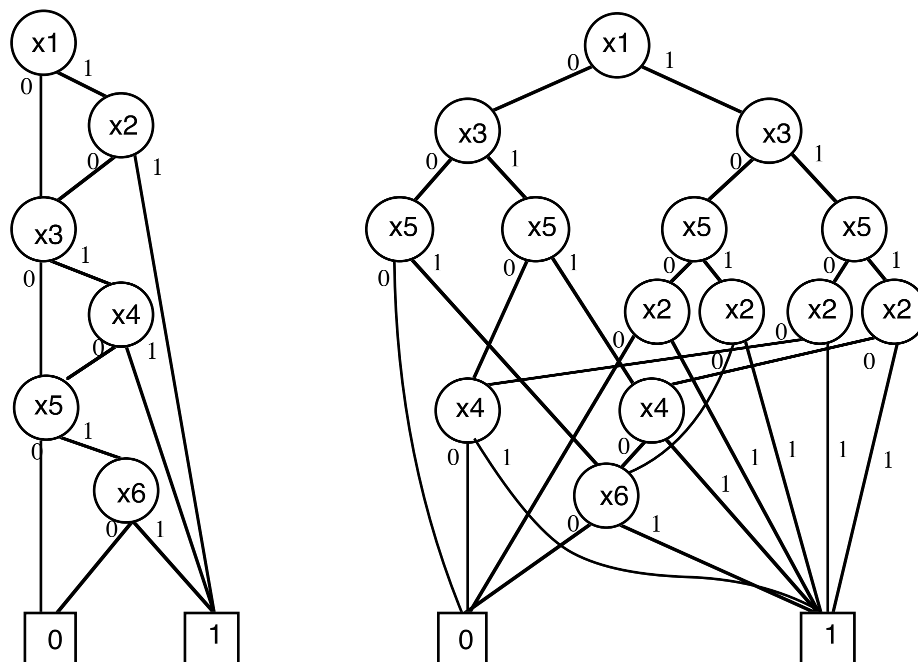


Figure 4.1. Ordering sensitivity for MBDs

The first methods published address the problem of finding a good ordering from a net-list description of a logic circuit. The idea is to traverse the circuit using a deep first algorithm oriented by some heuristic and keep track of the primary inputs found in the traversal. The heuristics should take into account fanin/fanout constraints and fanout reconvergence. Several works based on this method have produced similar results [Mal88], [Fuj88], [Cal92], [Jeo92].

Another approach consists deriving new orderings after the diagram is built. In this case, for each new ordering generated the diagram must be updated accordingly to reflect the change in the order of the decision variables. The method presented here belongs to this class of solutions. The initial MBD is built using a net-list analysis to find a good initial ordering [Cal92]. The generation of new orderings relies on an incremental manipulation technique (see [Jac91]) that exchanges two variables in the variable ordering and update the diagram to keep the representation coherent. It is interesting to note that two similar methods were developed almost simultaneously by different authors and were presented in [Fuj91] and [Ish91].

In the next sections we present the incremental manipulation technique and the heuristics developed to reduce the size of MBDs based on it. An exact solution that exploit the incremental nature of the algorithm is presented in the last section.

4.1 Incremental Manipulation of MBDs

The basic idea is to swap two adjacent variables in the MBD to produce a new ordering that leads to a different diagram (figure 4.2). The variables in the ordering vector are exchanged and the problem now is to update the diagram to reflect this change. The term *incremental* is used here to emphasize that the operation is of local nature. In fact, only the nodes associated to the variables being swapped are manipulated, the rest of the graph remains unchanged. The importance of this fact is that we can generate new MBDs (for the same function) without need of rebuilding the whole graph.

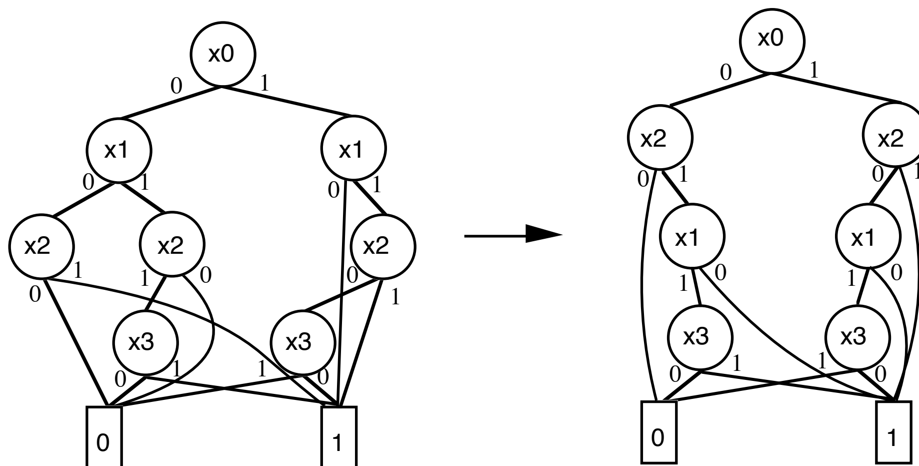


Figure 4.2. Example of the swap of variables x_1 and x_2.

Let us first define what is intended by *adjacent variables* in a MBD. Two adjacent variables in the MBD ordering are not necessarily adjacent in the diagram because the MBD may not depend on one or both of them. Thus, there is no meaning in selecting two variables from the ordering for swapping if they do not appear in the MBD. To circumvent this notation problem, we define the *effective MBD ordering* as the ordering of the variables that the function depends on.

Definition 4.1. The *effective ordering* of a MBD M is the ordering in which the variables $\{x_i \mid x_i \in support(M)\}$ appears in the graph. It is defined as $efO(M) = \langle \xi_1, \xi_2, \ldots, \xi_p \rangle$, where $p \leq |\mathbf{X}|$, $\xi_i = x_{\pi(j)}$, and $\pi(j)$ is a permutation of the indices of the input variables $\{x_j \mid x_j \in \mathbf{X}\}$

Definition 4.2. Two variables $x_i, x_j \in \mathbf{X}$ are *adjacent* in the effective MBD variable ordering if $\pi(i) = u$, $\pi(j) = v$, and $u = v \pm 1$.

Herein, all references to the variable ordering will address the effective one, and the term *effective* will be dropped out for short.

The exchange of adjacent variables is implemented in a two step process. First, the nodes at the adjacent levels are *swapped*, i. e., they are re-organized in order to reflect the change in the ordering. This may introduce redundancies that are eliminated in a second step, in which a local reduction is performed to put the diagram back into its canonical form. The locality of the *swap* operation is stated by the following theorems.

Theorem 4.1. Let f be a Boolean function of the input variables $\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$. Let M_1 be a MBD of f, constructed using the input variables ordering $O1 = \langle \xi_1, \ldots \xi_i, \xi_{i+1}, \ldots, \xi_p \rangle$, where ξ_i is given by: $\xi_i = x_{\pi(j)}$. Let $O_2 = \langle \xi_1, \ldots \xi_{i+1}, \xi_i, \ldots \xi_p \rangle$ be another ordering obtained from $O1$ by exchanging two adjacent variables $\{\xi_i, \xi_{i+1}\}$. The MBD M_2 of f, corresponding to this new ordering, differs from M_1 only at the MBD levels corresponding to indices i and $i+1$.

Proof. Consider a partition of the nodes of M_1 into three blocks, related to three disjoint subsets of levels (figure 4.3):

$\mathbf{X}_0 = \{n_j \mid n_j \in \text{level}_j(M_1), 1 \leq j < i\}$,
$\mathbf{S} = \{n_j \mid n_j \in \text{level}_j(M_1), i \leq j \leq i+1\}$ and
$\mathbf{X}_1 = \{n_j \mid n_j \in \text{level}_j(M_1), i+1 < j \leq n\}$.

To prove that M_2 differs from M_1 only at $\mathbf{S}$, we must show that $\mathbf{X}_0$ and $\mathbf{X}_1$ remain unchanged. Consider first the block $\mathbf{X}_0$. It defines a set of paths starting at the root and ending at level $i - 1$. Each path can be associated with a partial cube c_k, composed by the conjunction of the literals related to the nodes visited in the path. The set of subfunctions f_j defined by cofactoring f with respect to each c_k corresponds to a set of subMBDs with root either in $\mathbf{S}$ or in $\mathbf{X}_1$. Since function evaluation is independent of variable ordering, $\{f_j\}$ cannot be changed by any transformation that preserves the equivalence between M_1 and f. Thus, all the paths in $\mathbf{X}_0$ must remain associated to the same subfunctions in M_2, and $\mathbf{X}_0$ is not altered. A similar reasoning can be applied to the set of subfunctions g_j of M_1, obtained by cofactoring f with respect to the partial cubes defined by levels $\mathbf{X}_0$ and $\mathbf{S}$. Since the

cofactor operation is commutative, i.e. $(f_x)_y=(f_y)_x$, for any x, y, $\{g_j\}$ is unaffected by the swapping of variables ξ_i, ξ_{i+1}. So, block $\mathbf{X}_1$ must remain unchanged after the swapping of ξ_i and ξ_{i+1}. ◊

Theorem 4.2: Let M_2 be the MBD obtained by the application of the *swap* function on indices *i* and *i+1* of a reduced MBD M_1. Only the nodes *at level i+1* can be eliminated by the application of the local reduction operation *after* swapping.

Proof: Suppose we have $x_u = \xi_i$ and $x_v = \xi_{i+1}$. After swapping, a node corresponding to variable x_v can appear at level *i* if and only if it already existed at level *i+1 before* the swapping. It is associated to subfunctions in the MBD that depend on x_v, since the canonical form is prime and irredundant. Eliminating a node at level *i* means either making some subfunction independent of x_v or replacing the subdiagram whose root is the eliminated node by an equivalent one. The first case is clearly impossible because it implies modifying the Boolean function *f*, represented by the MBD. The second case imply the creation of two equivalent subfunctions at level *i* by the swapping procedure, but this cannot occur due to the independence of function evaluation with respect to variable ordering. So, for two subfunctions at level *i* to be equivalent after the swapping, they should be equivalent before it. This cannot happen because the initial MBD is supposed to be reduced. ◊

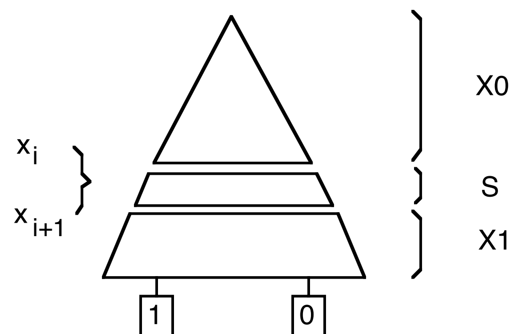


Figure 4.3. Swapping variables and MBD partition

The swapping of the variables is implemented by two functions: *swap* and *inc_reduce*. *Swap* is based on the independence of the cofactor operation with respect to the order of the cofactored variables. The MBD can be built by successively cofactoring the function up to reach a constant value. Consider a function $f(\boldsymbol{X})$. We can build the MBD by successively calculating $f_{x_1}, f_{x_1 x_2}$, etc. Based on the commutative property of the cofactor operation, which states that $f_{x_1 x_2}=f_{x_2 x_1}$, we can verify the equivalence between subgraphs in figure 4.4.

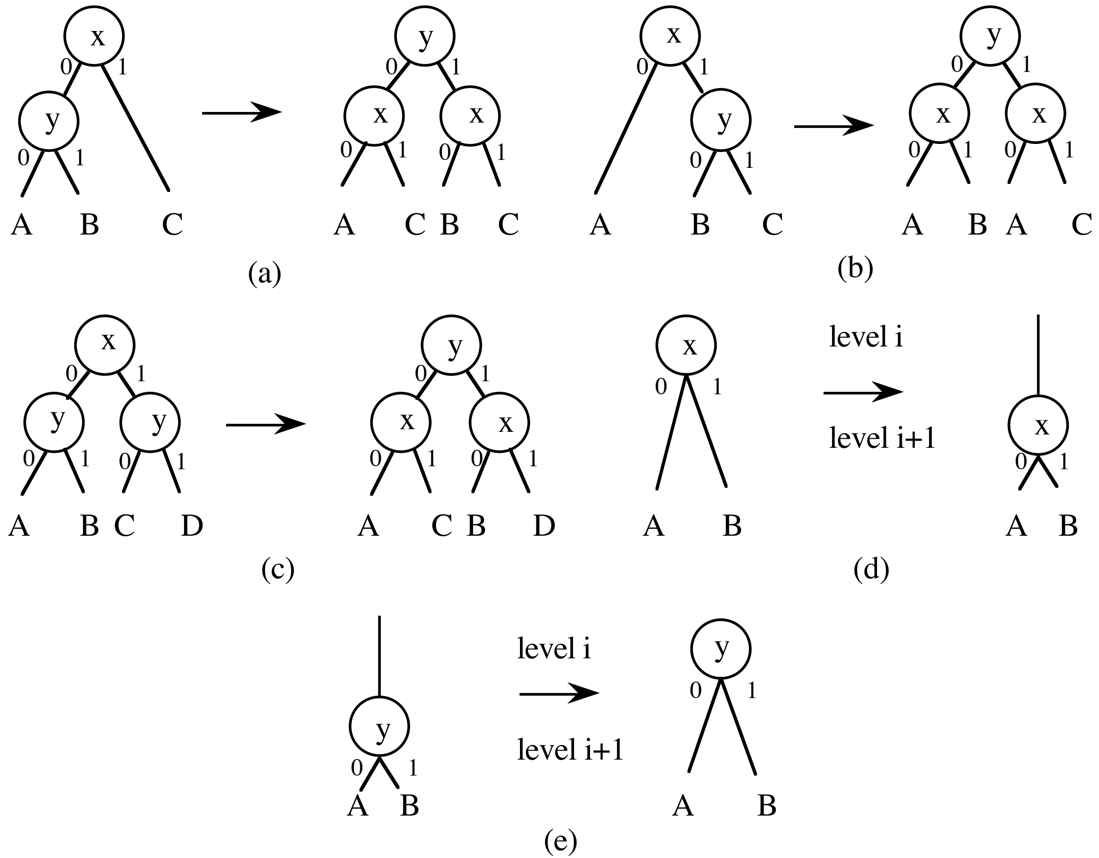


Figure 4.4: swapping configurations

The subgraphs represent possible configurations of nodes in the swapping levels *i*, *i+1*. Letters A, B, C and D denote some subfunctions (subdiagrams) in the MBD connected to the swapping nodes. *Swap* function performs a kind of *expansion* of the subgraphs, in the sense that it produces a complete subtree of three nodes (cases a, b and c in figure 4.4). The expansion phase may introduce new nodes (cases a and b) and in this case the size of the diagram is increased, but some of them can become redundant and must be deleted, eventually reducing the size of the MBD. A node become redundant either if (1) its two sons points to the same subfunction or if (2) an equivalent node is created after the subgraph's expansion. Figure 4.5 illustrates both cases. It was shown in [Lia92] that the contribution of factor (1) to the diagram's reduction is much smaller than the contribution of (2).

The deletion of the redundant nodes is done by the function *inc_reduce*. The algorithm is based on the *reduce* function by [Bry86], adapted to deal only with the set of *i_brothers* nodes with index i+1. A first step is to delete those nodes that have their low sons equivalent to their high sons (figure 4.5 (a)). Next step is to delete equivalent nodes at level i+1. For each of such nodes a key in the form *k(n)* = (node, low(node), high(node)) is built and a pair *(n, k(n))* is stored in an associative list *keylist*. If a node n_z has its key $k(n_z)$ already computed then it is discarded and its fathers are redirected to its equivalent node retrieved from *keylist*. Pseudo-codes for the *swap* and *inc_reduce* algorithms are shown in figures 4.6 and 4.7.

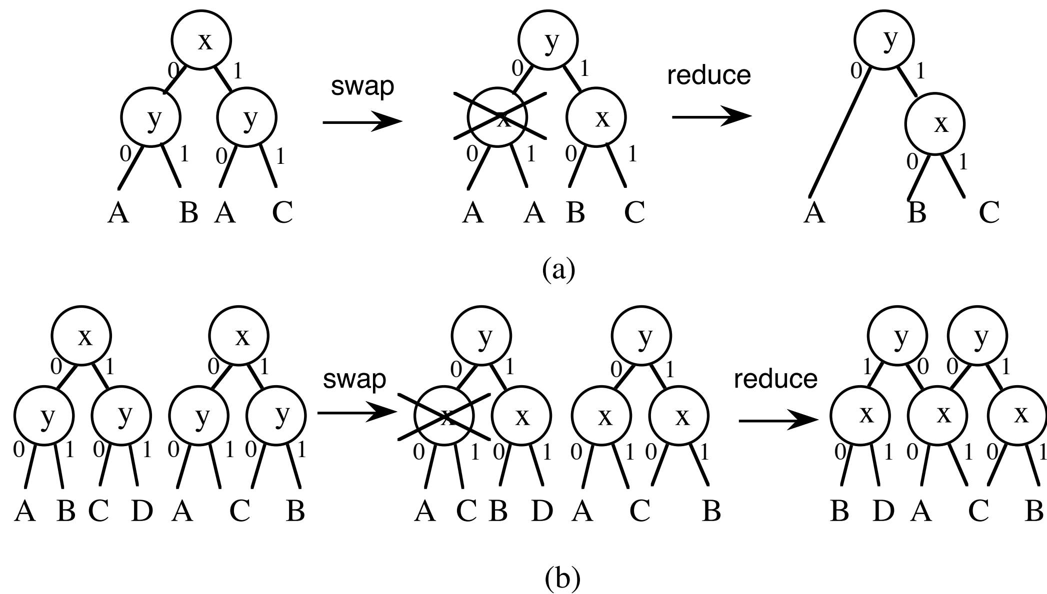


Figure 4.5. Redundancies arose after swapping variables.

Function *swap* uses a hash table to store all fathers of each node, the *Fathers_Table*, and an array of *i_brothers*; the *Brothers_Array*. The variable ordering is stored in MBD_order and the current size of the diagram in *MBD_size*. These data are computed at the beginning of the process and incrementally updated along the swapping process. The first step is to duplicate the nodes in the second swapping level (***foreach*** *node n* ***in*** *second*) in such a way that each node will have only one father. If a node from this level is the root of one function of the MBD and has a non empty set of fathers, then a special copy of it that can not be swapped is made for the output function. This prevents the modification of the function behavior that would be produced by the swapping of the root node. Next step in the algorithm is the identification of the cases shown in figure 4.4 (***foreach*** *n in first*) and the application of the correspondent transformation.

The function *inc_reduce* scans all nodes in level i+1. Each node is checked for redundancy, as stated in *definition 3.7*. If a node is redundant, then a pair with the redundant node as the first element and the node that will replace it in the diagram as the second element if created and inserted in the list *redundant_list*. After all nodes in level i+1 are processed, the *redundant_list* is used to delete the redundant nodes. Redundant nodes are found by checking either if their *low* and *high* pointers are equal or if there is a brother node with the same *low* and *high* sons. The later case is detected with the help of the list *Keys* that holds pairs in the form ((*low(n)*, *high(n)*), *n*). For each node *m* in level i+1 a pair (*low(m)*, *high(m)*) is formed. If there is an equivalent pair in *Keys*, then *m* is redundant and it will be replaced by its equivalent node retrieved from *Keys*. The elimination of the redundant nodes requires the update of the MBD diagram and also the update of the auxiliary data structures Fathers_Table and Brothers_Array.

```
function swap (i0, i1: index)
begin
  var    first,      /*list of nodes with index = i0*/
         second,     /*list of nodes with index = i1*/
         fathers,    /*list of fathers of a node*/
         gfathers:   /*list of grandfathers of a node*/
       list;

  /*take all nodes with index i0 and index i1 */
  first := get_brothers (Brothers_Array, i0);
  second := get_brothers (Brothers_Array, i1);
  foreach node n in second begin
    fathers := get_fathers (n, Fathers_Table);
    gfathers := {n in fathers | index (n) < i0}
    fathers := fathers - gfathers;
    if (gfathers not null  or  is_output (n))
     then   let the original node to the gfathers or to the
            output function
     else      /*let the original node to the first father*/
      fathers := remove_first_element(fathers);
   foreach f in fathers begin
     duplicate_node (n);
     update MBD, Fathers_Table, Brothers_Array and MBD_size;
   end;
  end;
  /* now perform transformations */
  foreach n in first begin
     il := index(low(n));
     ih := index(high(n));
     if (il = ih = i1) then begin /*low and high sons at i1*/
          exchange high(low(n)), low(high(n));
          update Fathers_Table;
     end;
     else if (il = i1) then begin   /*no high son at i1*/
          create a new node and place it at high(n);
          update Fathers_Table, Brothers_Array and MBD_size;
     end;
     else if (ih = i1) then begin /*n has no low son at i1*/
          create a new node and place it at low(n);
          updateFathers_Table, Brothers_Array and MBD_size;
     end;
     else        /*no sons at i1*/
       change n from level, update Brothers_Array;
  end;
  update variable order in MBD_order;
end;
```

Figure 4.6. The pseudo-code for *swap* function.

4.2 Variable Ordering: a Greedy Approach

The variable ordering can be modeled as a permutation problem.

Definition 4.3. Let $\mathbf{X} = \{x_1, x_2, ..., x_n\}$ be the set of input variables of a MBD M and $\Xi = \{\xi_1, \xi_2 ..., \xi_n\}$ be the set of locations of the x's in the variable ordering O of M. An ordering O is a bijective function π: $\mathbf{X} \rightarrow \Xi$, which is a permutation of the input variables. The set of all possible orderings forms the *solution space* of the variable ordering problem.

A *solution* of the variable ordering problem is a permutation π^i that, when used to build M, result in a MBD with the minimum number of nodes.

```
function  inc_reduce (i: index)
begin
var redundant_list,       /*nodes to be eliminated*/
    reduced_list,         /*nodes that will stay*/
    Keys,                 /*list of node keys*/
    node_lis:             /*nodes at level i*/
  list;

  node_lis := get_nodes (Brothers_Array,i);
  for each n in node_list begin
    if (low(n) = high(n))
     then redundant_list := append (<n,low(n)>, redundant_lis);
     else
     if (find (<low(n),high(n)>, Keys) /*isomorphic subgraphs*/
      then begin
       new_node := get isomorphic subgraph root from Keys;
       redundant_lis := append (<n, new_node>, redundant_lis);
      end;
     else Keys := append (<<low(n),high(n)>,n>, Keys);
  end;
  reduced_list := node_list - redundant_list;
  update Brothers_Array, MBD_size;
  foreach pair <n,newn> in redundant_list
      foreach father in get_parents(n, Fathers_Table)
       begin
         replace n by newn in father;
         update Father_Table;
       end;
  regenerate node identifiers in MBD;
end;
```

Figure 4.7. Pseudo-code of function *inc_reduce*.

A first remark is that the solution is not unique. In fact, there may have several distinct orderings that produce a diagram with the minimum number of nodes. Regarding the problem as a generic combinatorial problem, the only way to obtain an exact solution is to scan all the solution space, which is too costly for practical functions. An alternative way to find acceptable solutions within reasonable costs is to navigate the space solution following an heuristic strategy.

Definition 4.4. A point in the solution space is called a *state*. **X** is the set of *movable elements*. A new state π: **X** -> Ξ, is generated by *moving* some elements in the current state. A *move* can be *simple* or *composed*. In any case, a move must generate an unique new state. The gain of a move *m* is defined as the difference between the cost of the current state π^i and the next state π^{i+1}: gain(*m*) = cost(π^i) - cost(π^{i+1}).

A simple *move* in the ordering space is obtained by swapping two adjacent variables. Thus, for a MBD with *n* variables there are *n-1 simple moves* that can be made at the current state. The figure 4.8 illustrates the simple moves for a 4 variables ordering. A *composed move* is obtained by combining a set of simple moves.

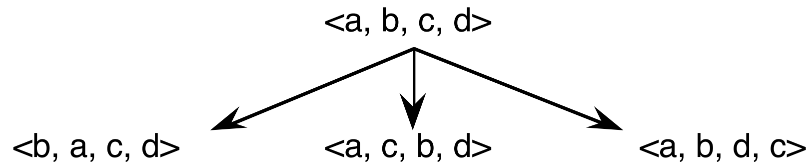


Figure 4.8. Set of simple moves for a 4 variable ordering.

Definition 4.5. A *greedy* heuristic to find a solution to the variable ordering problem selects among the set of possible next states the state that produces the highest gain.

We have implemented two greedy heuristics to find a good ordering for MBDs. The first one is called *swap_all* and the second one is *swap_down*. *Swap_all* evaluates all simple moves and selects the move that produces the highest gain. The process is repeated until no further gain is obtained. A pseudo-code for the functions *swap_all* and *swap_down* are presented in figures 4.9 and 4.10.

```
function swap_all (mbd): MBD_TYPE
var mbd: MBD_TYPE;
begin
var i,cost, oldcost: integer;

  oldcost := MBD_size(mbd);
  /*returns the move with the minimum cost and its index*/
  cost := eval_swaps(mbd,i);
  while (cost < oldcost) do
     begin
        swap(mbd,i);              /*swap and reduce index i,i+1*/
        oldcost := MBD_size(mbd); /*store new size*/
        cost := eval_swaps(mbd,i);
      end;
  return(mbd);
end;
```

Figure 4.9. Pseudo-code of function *swap_all*..

Swap_down differs from *swap_all* by allowing composite moves to occur at each iteration. The term *down* is used to express that the process start at the root of the MBD (which corresponds to the leftmost variable in the ordering) and continues *down to* the terminals[1]. In a composite move the algorithm visits all variables consecutively in the current ordering, starting

[1] In fact, there is no *up* or *down* regions defined in the MBDs. This concept comes from the tendency of drawing the diagrams with the roots at the top and the terminals at the bottom.

by the leftmost one. Each variable is swapped as long as it does not increase the MBD size (negative gain). Thus, if variable with index 1 does not produces a negative gain when swapped, it is exchanged with variable with index 2. Next, it is tested against variable with index 3. If it produces a negative gain when swapped, it stays at index 2 and the process continues from index 3 up to index |**X**| - 1. In each composite move an arbitrary number of swaps can be executed.

```
function swap_down (mbd) : MBD_TYPE
var mbd: MBD_TYPE;
begin
var cost, oldcost: integer;

  cost := oldcost := MBD_size(mbd);
  while (true) do   {endless loop}
  begin
     for i := 1 to (depth (mbd) - 1) /*don't swap terminals!*/
     /*eval_swap returns the number of nodes eliminated*/
       if (eval_swap(mbd,i) >= 0)
            then begin
            swap(mbd,i);
            cost := MBD_size(mbd);
            end
     if (cost = oldcost)
       then return(mbd);
       else oldcost := cost;
  end
end
```

Figure 4.10. A pseudo-code of function *swap_down*..

The algorithm finishes when a composite move does not produce a positive gain. Figure 4.11 gives an example of the behavior of the algorithms in the same situation. An hypothetical ordering $\langle a,b,c,d,e \rangle$ is shown and the gain of each possible swap is indicated. Note that in (b) while the pair (c,d) had a swap gain of -1, the new pair introduced (a,d) has a gain of 1. The creation of new pairs that may lead to further gains is the main potential advantage of *swap_down*.

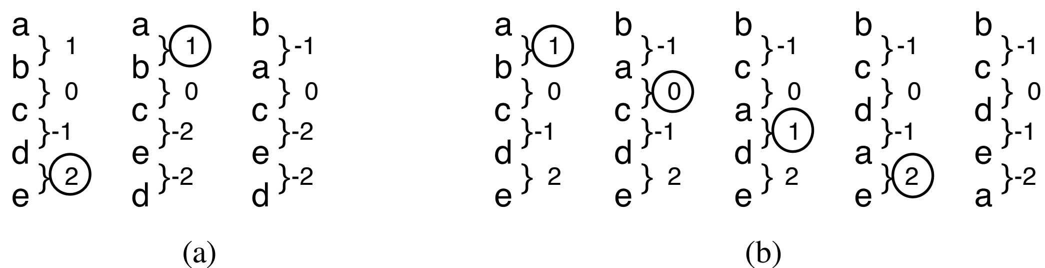


Figure 4.11. (a) swap_all and (b) swap_down examples.

4.3 Variable Ordering: a Stochastic Approach

One restriction of the greedy method is that it can get stuck at a local minimum that may be far away from the best solution. The result obtained is too dependent on the initial state in the solution space. As the exhaustive search is impractical, we must find an alternative way to explore the solution space without expending too much computing resources.

The variable ordering problem can be modeled as a permutation problem (definition 4.3), which belongs to a class of pure combinatorial problems. Therefore, we can apply combinatorial optimization methods in the search of a *good* solution. Note that this is feasible mainly due to the availability of the incremental manipulation techniques, otherwise the generation of new permutations would be too costly.

Combinatorial optimization is a vast research field. Its application in the CAD area has been growing in last years, stimulated by the development of new algorithms. Simulated annealing, for example, is one of such algorithms that was successfully applied in different synthesis domains like placement and routing of physical cells as well as in logic optimization. It is based on a random navigation of the space solution, controlled by a parameter called *temperature* (T) which regulates the amount of negative gain can be accepted when moving from one state to another. The initial temperature T_0 is high, which allows large jumps in the solution space. As in a physical cooling process, T starts to get smaller and smaller, and the search converges to a more precise region. Of course, the algorithm keeps track of the best solution found in the *walk* and returns it at the end. The method, however, suffer from some limitations. The time expended to find the solution is usually large. The algorithm is very sensitive to the cooling strategy, which is usually hard to define.

Recently, new set of algorithms based on *biological evolution* have evolved. [Kin89] presents an algorithm that applies the ideas of genetic evolution and mutation in the combinatorial optimization domain. The process can be roughly divided into two main phases that are supposed to simulate the biological evolution of living beings. First, the evolution phase, where the system tries to progress as far as possible in the search of a optimum solution. Second, when the evolution is blocked then the system mutates, and try to continues its evolution. Saab and Rao [Saa91] have adapted these ideas and formulated a new combinatorial optimization approach, called *Stochastic Evolution* (SE).

The basic difference between SA and SE is that in the latter the process initially allows only positive gains, running quickly up to the nearest local minimum. Then it tries to uphill climb by increasing the amount of negative gain that can be accepted. The *moves* in SE are not random as in SA, but heuristically oriented to improve the solution. According to the experiments reported in [Saa91], this strategy produces similar or better quality results than SA

in a smaller computing time. For those reasons we have decided to apply SE techniques to the variable ordering problem.

The process consists in a controlled traversal of the state solution where new states are generated by *single* or *composed moves* (definition 4.4). Each move *m* is accepted if its gain *gain(m)* is greater than an integer *r* defined as a random value in the interval [-*p*,0]. *P* is the control value that allows negative gains in order to uphill climb. It is initially set to zero. Thus, only positive gains are accepted at the beginning. The number of iterations is controlled by *R*, which is an estimation of the time needed to improve the current solution. Each time a better solution is found, the counter *c* is decremented by *R*, providing more steps to the SE algorithm to improve the solution. The SE algorithm can be summarized by the following steps:

SE(S: state):
1) set initial values: c <- 0, p <- p_0
2) store the current cost C_{cur} <- cost(s), the initial best value S_{best} <- s.
3) iteration: while c < R
   3.1 store old cost C_{old} = cost(S)
   3.2 generate a new state S = perturb(S, p)
   3.3 compute current cost C_{cur} = cost(S)
   3.4 update negative gain control p = update(p,C_{cur},C_{old})
   3.5 if improved, C_{cur} < cost(S_{best}) then store best solution, S_{best}=S, and update counter, c <- c - R
   3.6 if not improved, increment counter c
4) return S_{best}

Function *perturb(S, p)* computes the new state taking into account the current control gain *p*. Function *update* computes the new value for control gain *p*, which should be incremented each time the solution is not improved. These functions must be adapted for the variable ordering case. Thus, a single move in the ordering state space is given by swapping two adjacent variables. A composed move is obtained by a sequence of single moves. The SE algorithm for the variable ordering problem is called MBD_SE and is parameterized in order to accept different cost functions. The search in the ordering space is based on the SE algorithm and the cost function to be optimized is passed as a parameter to MBD_SE. The cost function can be any function that returns an integer value that reflect some property of the MBD. A condition that must be met is that positive gains correspont to an improvement of the solution. MBD_SE algorithm is shown in figure 4.12. The *perturb* function is presented in figure 4.13.

Perturb performs a composed move. Each simple move is accepted if the gain is positive or if it is smaller than a random negative value between 0 and -*p*. Variables can thus be arbitrarily displaced in the ordering. One alternative way to implement *perturb* is to execute a simple

move each time it is called, but in this case the counter *c* is too much decremented. For this reason a set of single swaps is used in order to produce higher gains. By the way, this may also produce higher negative gains to escape from a local minimum.

```
function MBD_SE (mbd, costf) : MBD_TYPE
var costf:    function;    /* cost function */
    mbd:      MBD_TYPE;    /* initial MBD */

begin
var p:              integer;          /*control gain parameter*/
    c:              integer;          /*counter*/
  cost, oldcost :   integer;          /*current and old cost*/
  best:             MBD_TYPE;         /*best mbd*/

  c := 0;
  p := 0;
  best := mbd;
  while (c < R) begin
    oldcost := costf(mbd);
    mbd := perturb(mbd, p, costf);
    cost := costf(mbd);
    p := update(p, cost, oldcost);
    if (cost < costf(best)) then begin
      best := mbd;
      c := c - R;
    end;
    else c = c+1;
  end;
  return(best);
end;
```

Figure 4.12. Pseudo-code of *MBD_SE* algorithm.

```
function perturb (mbd, p, costf): MBD_TYPE;
var mbd:   MBD_TYPE;      /* current state */
    p:     integer;       /* control parameter */
    costf: function;      /* the cost function */
begin
  var  gain:      integer;
       oldcost:   integer;

  if (Last_Best > R)      /* if no improvement after R steps */
    then  begin
      mbd = random_order(mbd);  /* generate a random ordering */
      return(mbd);
    end;
  oldcost = costf(mbd);             /* store initial cost */
  for i = 0 to n-2 begin            /* for all indices */
    mbd = swap(mbd, i);             /* make a move */
    gain = oldcost - costf(mbd);    /* compute the gain */
    if (gain > random(-p))          /* random accept */
      then oldcost = costf(mbd);    /* update initial cost */
      else mbd = swap(mbd,i);       /* no:restore initial state*/
  end;
  return  (mbd);
end;
```

Figure 4.13. Pseudo-code of the *perturb* algorithm.

A potential problem with the SE algorithm which was detected in our experiments is the lack of large uphill climb steps. The solution can *cycle* around a local minimum if the *valley* is too deep (see figure 4.14). There is no mechanism to prevent the process to go back to the same local minimum after a few steps. To avoid this problem a random state is generated if the best solution is not improved after *R* steps. The number *R* is chosen because it is the expected number of iterations to improve the best solution.

The *update* function should also adapted to the ordering problem. We have verified that *perturb* generates states that oscillate around a local minimum if we adopt the approach of [Saa91] where the control parameter is incremented only if two successive costs are equal and otherwise reinitialized. To circumvent this problem we choose an average value of the last four costs as a reference. If the difference between the current cost and the average value is smaller than a threshold value *t*, *p* is incremented, otherwise it is reinitialized to its default value. The threshold t is selected to take into account the small oscillations around a local minimum and is usually set to 2 or 3. The *update* function is sketched in figure 4.15.

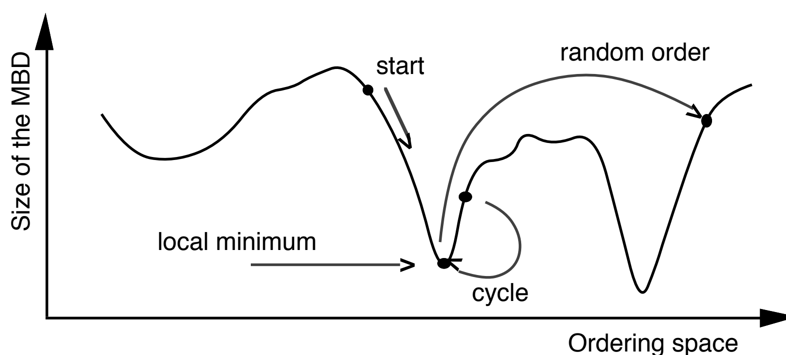


Figure 4.14. Cycles in the SE algorithm.

4.4 Variable Ordering: the Exact Solution

There are only two methods to find the exact solution for the MBD ordering problem up to date. The exhaustive search, which performs all *n!* permutations rebuilding the MBD at each new ordering with cost of $O(n!2^n)$ and the method proposed by [Fri87] that restricts the search to a subset of the combinations based on the partial orderings, leading to a time-complexity of $O(n^2 3^n)$. In order to get a reference to compare the results of the SE technique, we developed a version of the exhaustive search algorithm that uses the incremental techniques to produce new orderings. The advantage is that the MBD must be build only once and the resulting algorithm has complexity of $O(n!)$ only. To have an idea of the differences with respect to the other cost functions we show in table 4.1 the respective costs for some small values of *n*.

The algorithm is based on a method proposed in [Knu69]. For each permutation of $\{x_1, x_2, ..., x_{n-1}\}$ variables, it generates n others by inserting the variable x_n in all possible places:

$$\{x_n\ x_1\ x_2\ ...\}\ ,\ \{x_1\ x_n\ x_2\ ...\},\ ...\ \{x_1, x_2,\ ...\ , x_{n-1}, x_n\}$$

This procedure can be easily implemented using the *swap* technique. Each swapping generates a new ordering not already produced. Thus, the number of steps is exactly $n!$.

```
global    Queue: array of integer;    /* last four costs */
          Threshold: integer;
          Pse: integer;               /* default value */
function update (p, cost, oldcost) : integer
var p :        integer;          /* control parameter */
     cost:     integer;          /* current cost */
     oldcost:  integer;          /* last cost */

begin
  var av: integer;

  push(oldcost, Queue);      /* put last cost in the queue */
  a=(Queue[0]+Queue[1]+Queue[2]+Queue[3])/4; /*compute average*/
  if (abs(cost - av) < Threshold)  /*  if out of threshold */
  then return(p + 1)              /*  increase p */
  else return (Pse);             /* else return default value */
end;
```

Figure 4.15. Pseudo-code of *update* function.

n	$n^2 3^n$	$n!2^n$	**n!**
6	26.244	46.080	720
8	419.904	~ 10.000.000	40.320
10	~5.900.000	~ 3.7 x 10^9	3.628.800
11	~21.500.000	~ 2 x 10^{12}	~40.000.000

Table 4.1. Cost functions for the exact solution

4.5 Experimental Results

We have tested a prototype of the algorithms implemented in Common LISP over a set of circuits from the MCNC benchmark set and from the Berkeley PLA test set. The results obtained are summarized in table 4.2. The *Initial* column shows the MBD size built with the variable ordering specified in the file. The *Swap* column shows the application of the greedy methods. First the function *swap_all* is applied and then the function *swap_down* is executed. This have produced the best results with respect to the greedy methods. *Swap%* gives the

percentage of gain of the greedy methods. Column *MBD_SE* presents the results for the stochastic approach applied to the initial MBDs. *SE%* shows the gains produced with the stochastic approach. Column *Exact* presents the exact solution for the cases where it could be computed, here restricted to $|\mathbf{X}| \leq 10$. *Nvar* column informs the number of input variables of each circuit. Last columns *AR* and *ARSA* refers to results published in [Ish91]. *AR* is a greedy approach that swaps arbitrarily chosen variable pairs and accepts the swap only if it reduces the size. *ARSA* is based on simulated annealing and provides, thus, a hill climb alternative to better explore the ordering space. A '-' indicates that there is no reported result.

Circuit	Initial	Swap	Swap%	MBD_SE	SE %	Exact	nvar	AR	ARSA
5xp1	90	87	3	**70**	22	70	7	**70**	**70**
Alu3	145	99	32	**65**	55	65	10	**65**	**65**
Bw	120	112	7	**107**	11	107	5	**107**	**107**
Dekoder	24	23	4	**21**	13	21	4	-	-
Dk27	33	**28**	15	**28**	15	28	9	30	**28**
Duke2	978	449	54	404	58	-	22	-	-
F51m	72	**69**	4	**69**	4	69	8	71	**69**
O2	20	**20**	0	**20**	0	20	4	-	-
M1	49	42	14	**38**	22	38	8	**38**	**38**
M2	142	113	27	87	39	-	25	-	-
P1	347	243	30	**193**	44	193	8	-	-
P82	72	67	7	**61**	15	61	5	-	-
Risc	111	72	35	**70**	36	70	8	**70**	**70**
Sqn	81	70	14	**55**	32	55	7	**55**	60
Z4	66	**28**	57	**28**	57	28	7	35	33

Table 4.2. Experimental results for swapping based techniques.

If we do not consider circuit O2, which is a *pathological* case for the reduction[2], we have an average reduction of the initial MBD of 20% for the greedy approach and of 30% for the stochastic approach. Compared with the greedy method, MBD_SE produces an average improvement of 10% over the *swap* result which represents a relative gain of 50%. MBD_SE found the exact solution it in all cases where we could compute it. For increasing values of *n* the space solution grows in *n!* and it is hardest to find the exact solution in a reasonable computing time. In theory, the best solution should be found if enough computing time is provided. In practical cases we can expect that, in average, the solution is close to the optimum. Of course, the time required by the stochastic algorithm is sometimes much larger than that require by the greedy method. In the worst case (Duke2) MBD_SE run for several hours, while the greedy algorithms required a few minutes. We have thus a compromise: if one needs a reasonable solution in a shorter time the greedy approach can be used. If a better result is more important (as in the case of FPGA synthesis, for example), then the stochastic

[2]In fact, there is a class of functions whose MBD size is insensitive to the variable ordering. It is the class of symmetric functions. In this case, for a given function any ordering will produce a MBD with the same size.

approach should be chosen. Last but not least, for small functions with |**X**| ≤ 10, the exact solution can be computed in reasonable time (some hours in LISP on a Macintosh, surely less than a hour in a workstation in C++).

Comparing our results to those from [Ish91], we may roughly say that they have implemented a better greedy approach, while our stochastic evolution method have obtained better results than their simulated annealing one. One precision must be done, however. The execution times provided in [Ish91] indicates that AR takes in average more than a half of the ARSA average execution time. In our case, the ratio between MBD_SE and Swap times ranges from 5 to 10 times. As the stochastic techniques should take comparable time, we estimate that Swap should be much faster than AR in comparable environments.

4.6 Comments

The size of the MBD is a very important parameter. It is directly related to the storage required to manipulate Boolean functions and to the performance of its logic manipulation algorithms. For the synthesis into FPGA devices, the diagram's size is the main criterion in the estimation the final circuit cost for the methods based on the graph covering approach.

The main factor in the determination of the MBD size is its variable ordering. In this chapter we have introduced a *incremental manipulation* technique for MBDs that allows the fast generation of new variable orderings by swapping *adjacent* variables in the diagram. As we will see in a later chapter, this technique can be useful not only for the production of new orderings but also in the determination of some Boolean properties of functions represented by MBDs.

We have developed three incremental methods for the reduction of the MBD size. One method uses a greedy approach and is the fastest one, yielding an average reduction of 20% over the initial MBD size. The stochastic approach overcomes one limitation of the greedy algorithms - to get stuck at local minimum - at the expenses of more computing time, obtaining average reductions of 30% with respect to the initial MBD size. Finally, we have implemented the exact solution by exhaustive search. All input variable permutations are generated by swapping adjacent variables in the ordering and the best result is retrieved. Its main feature is that it is the fastest method to find the exact solution for small to medium functions (|**X**| ≤ 10, see table 4.1). The reason why it achieves better performance than the traditional exhaustive search is that the use of incremental generation of new orderings avoids the cost of rebuilding the diagram at each step. For functions with more than 10 variables the approach proposed by [Fri87] provides better results, but any way they are already too costly to be of practical use.

The greedy method can be used in cases where no strong MBD compaction is required. For example, if one need to perform a few logic operations with the MBD, then the overhead introduced by a larger diagram in the manipulation algorithms is probably less important than

the time needed to find a smaller diagram with the stochastic approach. On the other hand, for the FPGA synthesis, where the MBD size reflects the final implementation cost, a strong compaction is recommended even if it can take hours to be completed. The exact approach can be used efficiently in very small functions, from 4 to 6 variables, as those that occur in nodes of a Boolean network, for instance.

We found two other similar works in this domain in the literature, both coming from Japan. [Fuj91] had developed a similar technique for swapping variables. However, the heuristics proposed to reduce the BDD size based on their swapping technique did not seem very interesting. They provided a few results on some benchmarks that were not treated here.

[Ish91], on the other hand, presents a set of interesting heuristics based on the swapping technique of [Fuj91] over a large range of benchmarks. In particular, they have obtained an important constant time improvement on the exact method from [Fri87] by using BDDs to compute the intermediate functions. They developed a more exhaustive greedy approach, that swaps arbitrarily pairs of variables while this reduces the BDD size. If a pair of non-adjacent variables is selected for swapping, this is done by displacing the variables through a series of adjacent swaps. This produces better results that our greedy approach, but it takes more time.

Their simulated annealing approach have produced worse results than our stochastic evolution in two examples. However, this is not really concluding, since they are both stochastic techniques. To have more confident comparison, we should run both algorithms several times over the same benchmarks and compare the average results.

In short, the important point here is that the swapping technique allow us to develop several reordering approaches that are useful not only for the reduction of the MBD size, but also in other domains of logic synthesis, as it will be seen in next chapters.

Chapter 5

Minimization of Incompletely Specified Functions

This chapter describes two methods for the simplification of incompletely specified functions described by MBDs. One is based on the two-level minimization techniques and generates a prime and irredundant cover for the function. The other one is based on a graph matching approach and aims at the reduction of the number of nodes of the diagram.

The minimization of incompletely specified functions plays an important role in logic synthesis. The existence of a set of points where the function value is not defined introduces additional degrees of freedom in the design that can be explored to reduce the cost of its physical implementation. A incompletely specified function defines a family of compatible completely specified functions. The logic minimization problem is how to select among those functions one that minimizes the circuit realization in the desired technology. The solution is, of course, strongly related to the design style the logic function shall be implemented.

Let $\tilde{f} = (f_{on}, f_{off}, f_{dc})$ be an incompletely specified function $\tilde{f}: \mathbf{D} \rightarrow \{0, 1\}$, $\mathbf{D} \subset \mathbf{X}$. $\tilde{f}$ defines a family of functions comprised between two extremes: $f_{min} \leq f \leq f_{max}$, where f_{min} and f_{max} are the completely specified functions obtained by setting the values associated to the vertices in the don't care set f_{dc} to 0 and to 1, respectively. Thus, $f_{min} = (f_{on}, f_{off} \vee f_{dc}, \emptyset)$ and $f_{max} = (f_{on} \vee f_{dc}, f_{off}, \emptyset)$. Consider also that *cost(f)* is the cost function we want to minimize.

Definition 5.1. The *logic minimization* of an incompletely specified function $\tilde{f}$ consists in the choice of a compatible function $f, f_{min} \leq f \leq f_{max}$, such that *cost(f)* is minimum.

Although we usually refer to the *minimization of a Boolean function*, in fact what is minimized is its logic representation. Indeed, the solution to the minimization problem is a compatible function whose logic representation leads to a minimal realization in a given technology. The expressions are related to technological features by means of cost functions.

There are several ways to describe logic functions. Sum-of-products, Boolean networks, factored forms, Read-Muller expressions, MBDs, Functional Decision Diagrams [Keb92] and ITE DAGs are some examples. Our interest here concentrates on the minimization of sum-of-products, Boolean networks and MBDs, due to their application in multi-level synthesis.

The exact minimization of logic functions involves the solution of NP-complete problems and, thus, can not be envisaged for all but small problems. The alternative in this case is to turn our attention to heuristic algorithms that can produce *good* or *acceptable* solutions. To define more precisely what is a good solution, let us first generalize the concepts of primality and irredundance.

Definition 5.2. Let S be a set of elements s_i, where each element s_i is itself a set. An element s_k is *prime* if there is no other element $s_j \in S$ such that $s_k \subseteq s_j$. S is irredundant if there is no element s_k such that $s_k \subseteq S \setminus \{s_k\}$.

A prime and irredundant form is not necessarily an optimum solution, but it presents some interesting properties:

- the fact that each element is prime means that the correspondent digital devices implement no useless logic.
- if the expression is irredundant, the resulting circuit contains no redundant gates.
- a prime and irredundant circuit is 100% testable for single stuck-at faults.

For those reasons, primality and irredundance are used to guide the heuristic minimization algorithms in the search of acceptable solutions. Therefore, the logic minimization problem can be restated as follows:

- given a function $\tilde{f}$, find a compatible function f, $f_{min} \leq f \leq f_{max}$, such that the logic expression of f is prime and irredundant.

The concepts of prime and irredundant cover come from the two-level minimization context. Using the set notation, the ON-set of the function f is defined as $f_{on} = \{ \boldsymbol{x}_i \in \mathbf{B}^n \mid f(\boldsymbol{x}_i) = 1\}$. A cube is defined as the set of points in the Boolean space that is covered by the cube function. If C is a cover of f, then a cube $c_i \in C$ is prime if it can not be expanded without intersecting the OFF-set of the function f. This is equivalent to say that there is no cube c_j belonging to the

set P of all implicants of f such that $c_i \subseteq c_j$. Finally, C is irredundant if there is no cube c_i that is covered by $C \setminus \{ci\}$.

A node n_i in a MBD is prime if its low and high sons are not equivalent. In other words, n_i is prime is there is no $n_j \in \{n_k \mid n_k$ is descendent of $n_i\}$ such that the f^{n_i} is covered by f^{n_j}. A MBD M is irredundant if there are no nodes n_i and n_j such that n_i is equivalent to n_j. A MBD without the **X** terminal is always prime and irredundant. An incompletely specified MBD, i.e., a MBD with the don't care terminal, is prime but not irredundant because the **X** terminal introduces node equivalencies.

The primality and irredundance concepts are extended to multi-level expressions (or Boolean networks) by considering that each node function is represented by a two-level expression. Then, if the covers of the nodes are prime and irredundant with respect to the function denoted by the Boolean network, the network itself is prime and irredundant. It must be emphasized that a cube of a node cover is prime with respect to the global function, and *not* to the node function. This means that we may expand the cube even if this alters the node function, provided that the network function is not changed. These concepts were used in the multi-level minimizer BOLD [Bos87].

In this chapter we discuss the minimization MBDs and two-level expressions. The problem of minimizing multi-level expressions will be tackled in a later chapter, but it is essentially based on the application of the techniques presented here to minimize the node functions of the Boolean network. Next section presents a method that produces prime and irredundant two-level forms [Jac92] directly from the function diagram. The following one aims at the reduction of MBD size [Jac93], and is targeted to FPGA synthesis.

5.1 Two-level Minimization with MBDs

One of the first logic minimization methods was based on the use of the Karnaugh diagram. With a little practice, the user can find optimal solutions for functions up to 6 variables. Beyond this limit the task becomes very cumbersome. The diagram provides a graphical notation for the following Boolean property:

$$x_1 \cdot x_2 \vee \bar{x}_1 \cdot x_2 = (x_1 \vee \bar{x}_1) \cdot x_2 = x_2$$

This property says that two cubes that differ only by one variable that appears in opposite phases in each one may be merged into a single cube that does not depend on that variable. By applying iteratively this property we can generate the prime implicants of the function. Lets show some examples.

• Figure 5.1 illustrates the process of generating a prime implicant of a function. The starting point is a single minterm $x_1 \cdot x_2 \cdot x_3 \cdot x_4$. It is expanded first in the x_2 direction (***1***). Next it is expanded in the x_4 direction (***2***), producing the prime implicant $x_1 \cdot x_3$. The operations performed are the following ones:

(***1***) $x_1 \cdot x_2 \cdot x_3 \cdot x_4 \vee x_1 \cdot x_2' \cdot x_3 \cdot x_4 = x_1 \cdot x_3 \cdot x_4$
(***2***) $x_1 \cdot x_3 \cdot x_4 \vee x_1 \cdot x_3 \cdot x_4' = x_1 \cdot x_3$

• Figure 5.2 shows an example of three different covers of the same function.

(a) is not prime but irredundant. The cubes marked with an asterisk can be merged.
(b) is prime but redundant. The minterms contained on the cube marked with an asterisk are already covered by other cubes.
(c) is prime and irredundant.

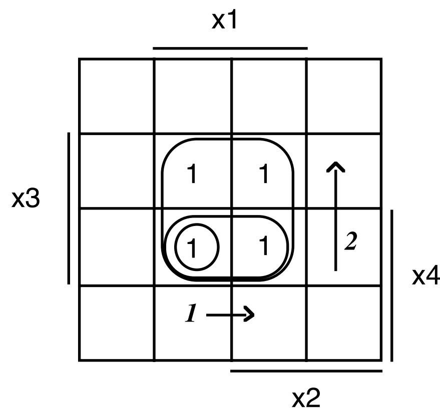


Figure 5.1. Cube expansion example.

The first systematic method for two-level minimization was developed by Quine [Qui52]. It consists basically in two phases:

• compute all prime implicants of the function
• find a minimum prime cover

The main handicap on this approach is the necessity of computing all prime implicants. It can be shown that a function with *n* inputs may have up to $3^n/n$ primes. Moreover, the extraction of a minimum cover from the set of primes is implemented with a branch and bound technique that is known to belong to the class of NP-complete problems. These limitations restrict the application of the method to very simple problems.

The advent of the PLAs in the 70's stimulated the development of new heuristic algorithms for the minimization of more complex two-level functions. They start by relaxing some optimization criteria such as either computing a subset of all primes or extracting a near

minimum cover. Further improvements in terms of performance were achieved by working directly with the implicants of the function, instead of computing all minterms.

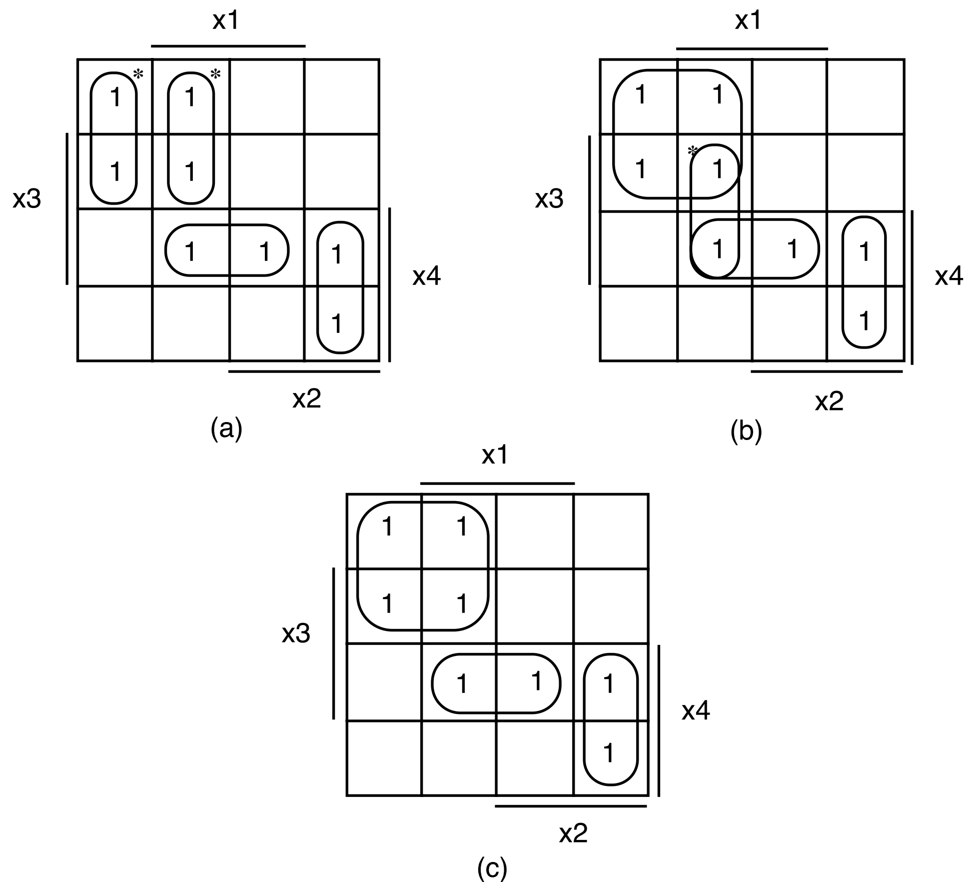


Figure 5.2. Examples of primality and irredundance.

MINI [Hon74] introduced a set of new heuristics that produced remarkable results. It is a multi-valued minimizer, that is, the inputs may have more than two binary values. The motivation for the extension of the minimization to multi-valued functions came from the use of decoders at the PLA inputs. A two-bit decoder receives two inputs, say x_1 and x_2 and produces four outputs: $x_1 \cdot x_2$, $\bar{x}_1 \cdot x_2$, $x_1 \cdot \bar{x}_2$ and $\bar{x}_1 \cdot \bar{x}_2$. The inverse of these are fed into the input plane of the PLA. These four functions provide more information than the non-decoded inputs x_1, x_2, $\bar{x}_1$ and $\bar{x}_2$, and may reduce the PLA array of 10-20%. A decoded input can be represented by a single multi-valued variable. Moreover, MINI approach to handle the multiple-output minimization was to represent the outputs as a single multi-valued variable, and treat it as any other input in the minimization process. Another major contribution of MINI was the expand-reduce iteration. In a first step, an implicant is maximally expanded in its input and output parts, and the other implicants covered by it are removed. The problem here is that the prime cover found depends on the ordering in which the variables are taken in the expansion. A simple example is given in figure 5.3 (a) and (b), with a function of five variables. Both

covers are prime and irredundant, but (a) is composed by five cubes while (b) is formed by only four cubes.

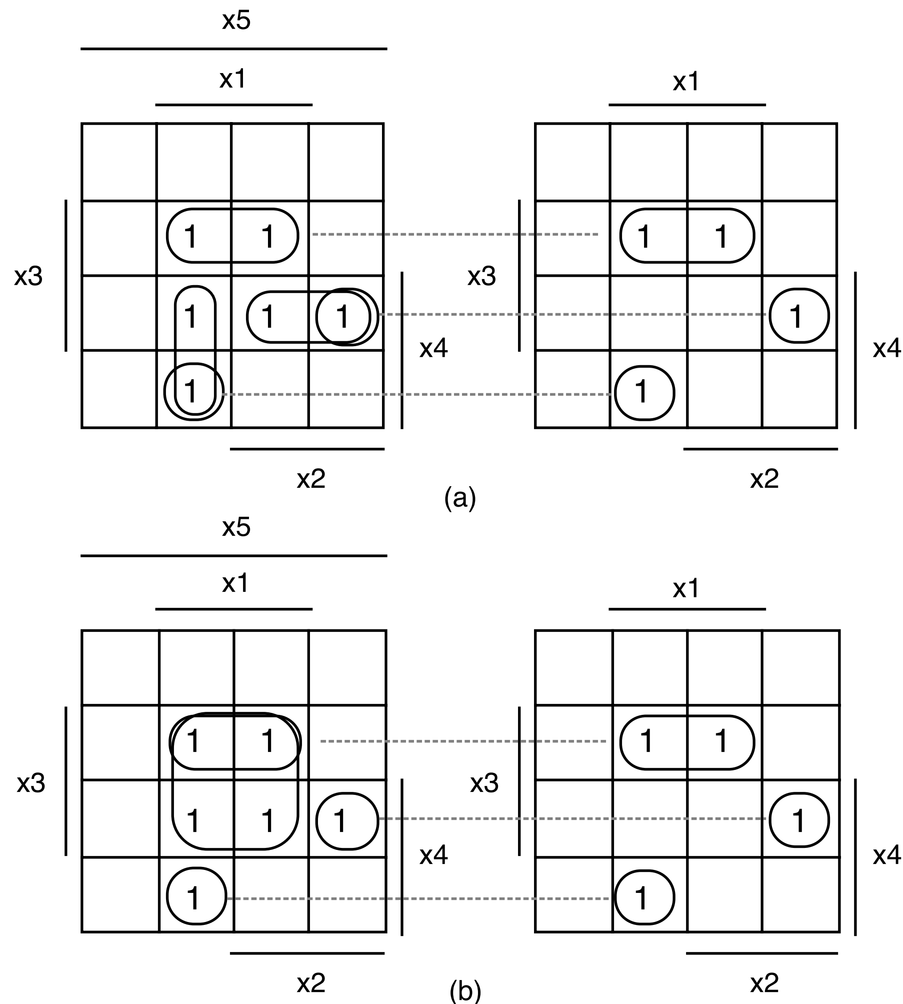


Figure 5.3. Two prime and irredundant covers.

The problem in cover (a) is that cube $x_1 \cdot x_3$ was missed due to the ordering the variables were selected in the expansion. To minimize the effect of the expansion ordering, MINI reduces the expanded cover such that each cube contains only essential minterms. Then the expansion-reduction process iterates until no improvement.

Several minimizers followed the approach pioneered by MINI. The most successful one was ESPRESSO [Bra84][Rud86]. Based on the expansion-reduction cycle, ESPRESSO introduced new algorithms that improved the results of MINI both in execution time as well as in the cover size. The main operations carried out by ESPRESSO are summarized bellow.

1) Compute the OFF-set of the function.
2) Expand implicants into primes and remove covered implicants.
3) Extract essential primes and put them in the don’t care set.

4) Find a minimal irredundant cover
5) Reduce each implicant until it is composed by only essential minterms.
6) Iterate 2, 3 and 5 until no improvement.
7) Lastgasp - try 5, 2 and 4 one last time using a different heuristic. If successful, continue the iteration.
8) Makesparce - include the essential primes back to the cover and make the PLA structure as sparse as possible.

ESPRESSO needs the OFF-set of the function to check if the prime implicants can be expanded with respect to some variable. If the expanded cube intersects the OFF-set, then the expansion fails for that variable. Since the essential primes must be present in any solution, they are removed from the cover in the beginning of the process and included in the don't care set, in order to allow the other implicants to be expanded over them. The expansion-reduction process reshapes the cubes exploring several different covers. The lastgasp is a final try to improve the result by making sure that no new prime can be added to the cover in such a way that two primes are removed. This algorithm has produced good results over a large set of PLA examples. A new version called ESPRESSO-MV was presented in [Rud86]. It extends the original algorithms to tackle the multi-valued minimization problem. The new heuristics produced better results than the original ESPRESSO even for binary valued inputs. Another methods that use different approaches were presented in the literature, as [Gur89] and McBoole [Dag86]. This later presents an exact procedure that can be applied to functions with up to 20 inputs. ESPRESSO also had its exact version [Rud87], that computes all primes and use an exact branch and bound technique to find a minimum cover. Both EXPRESSO-EXACT and McBoole has achieved comparable results, treating functions of reasonable complexity.

In this work we are interested in the application of two-level techniques for the multi-level synthesis problem. Its main application is the minimization of the node functions which is employed either in the multi-level minimization phase or in the node decomposition for technology mapping. This restrict our attention to single output functions. Another particularity of the multi-level context is that node functions may generate large OFF-sets. This can lead to inefficient performance for minimizers that need the OFF-set to expand its primes, which is the case of ESPRESSO. A new version of ESPRESSO was developed specially to tackle this problem [Mal88]. They propose to use what they called *reduced OFF-set*. The idea is that not all cubes of the OFF-set are needed for the expansion of the implicants. Only the OFF-set cubes that are adjacent to the implicants are used and the others may be discarded.

We have developed a method that generates prime and irredundant sum-of-products from single output incompletely specified MBDs. It works directly on the diagram and use some properties of the MBDs to simplify the minimization process. The cover is produced in a single bottom-up traversing of the graph. The ordering in which cubes are expanded is defined by the

ordering of the variables in the MBD. It should be noted that a cube with k variables may be expanded in $k!$ different ways. Restricting the directions of the cube expansion speeds up the process, at the price of eventually missing a better expansion. A useful property of MBDs is that the ON, DC and OFF-sets are all represented in the same graph. Thus, there is no need of computing a separate OFF-set for cube expansion, we just use the OFF-set already present in the MBD. As it is a compact representation, the large OFF-sets of the node functions do not introduce significant overhead.

5.1.1 The MBD Cover

Dealing at the same time with MBDs and sum-of-products introduces a compatibility problem. They are different types of function representations and a conversion is needed from one type to the other in order to manipulate them together. To simplify this task we have adopted a mixed representation called the *MBD cover*. Each cube is represented by a small diagram, the *MBD cube*. An example of this kind of representation is shown in figure 5.4.

The advantage of using this mixed representation is that we can apply all logic operations defined over MBDs to the MBD cubes as well. In the following subsections we describe the method in detail.

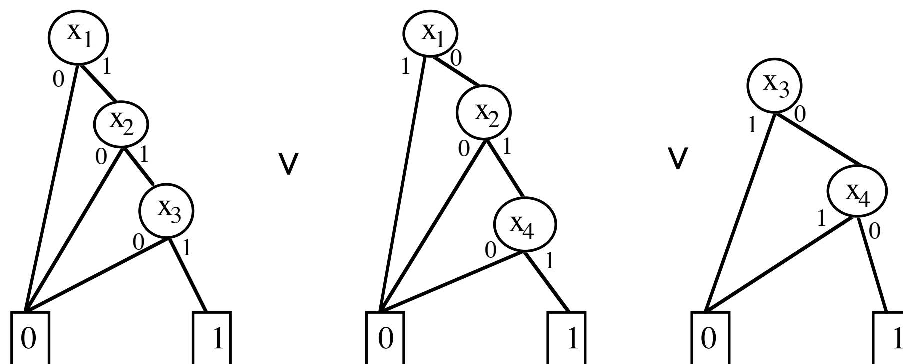


Figure 5.4. A MBD cover for the function $f = x_1x_2x_3 \vee \bar{x}_1x_2x_4 \vee \bar{x}_3\,\bar{x}_4$.

5.1.2 The Method

Let f be a Boolean function and M its MBD representation. Each node n inside M is the root of a of M and is associated to a subfunction f^n of f. The basic idea behind this method is to find a prime and irredundant cover *for each node n* of M, starting by the terminal we want to compute the cover. Then, at the end of the process, the root of M will contain a prime and irredundant cover of f.

To find a prime and irredundant cover of f^n, we assume that the low and high sons of n are already described by prime and irredundant covers. The variable associated to node n, v_n,

divides the Boolean space of f^n into two disjoint subspaces: one connected to v_n and the other connected to its complement, $\bar{v}_n$. This fact is illustrated in figure 5.5. The cover of the function $f^{low(n)}$ is represented by $S_l = \{c_{1l}, c_{2l}, \ldots, c_{pl}\}$ and the cover of the function $f^{high(n)}$ is represented by $S_h = \{c_{1h}, c_{2h}, \ldots, c_{qh}\}$. This corresponds to the Shannon expansion [Sha49]:

$$(1)\ f^n = v_n f^{high(n)} \vee \bar{v}_n f^{low(n)} = v_n(c_{1h} \vee c_{2h} \vee \ldots \vee c_{qh}) \vee \bar{v}_n(c_{1l} \vee c_{2l} \vee \ldots \vee c_{pl}$$

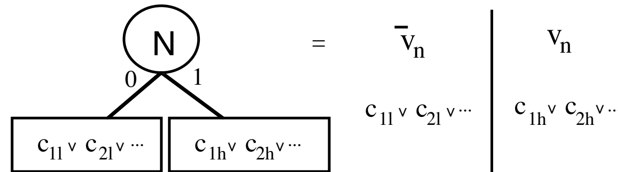


Figure 5.5. Boolean subspaces of N.

The cover of f^n is computed by merging the covers of $f^{high(n)}$ and $f^{low(n)}$ as in (1). The cubes are made prime with respect to node variable v_n and the cover is checked for irredundance. The global process is exemplified in figure 5.6.

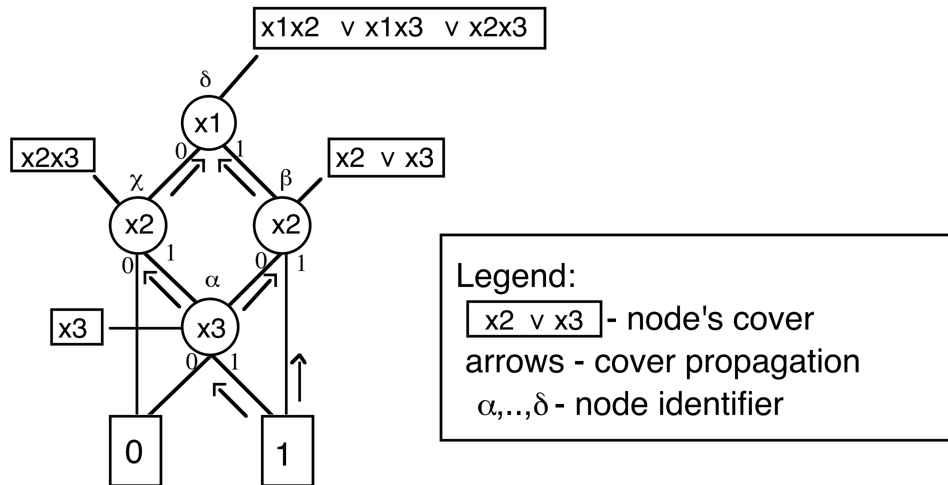


Figure 5.6. Generation of a cover for a MBD.

The diagram of figure 5.6 represents the majority function. To find its correspondent sum-of-products form we start by the terminal **1**. The next step is to compute the cover of node α. Then, the covers of β and χ can be computed. Finally, the root cover is found. Note that this order must be respected to allow the application of (1).

5.1.3 Primality

Verifying if a cover is prime can be done by taking each cube and testing it for cube primality. ESPRESSO [Bra84] *expand* procedure does this by expanding each cube - from a list of cubes

ordered by size - and eliminating any cubes of the rest of the cover that are covered by the expanded one. In the MBD context, as the covers of $f^{low(n)}$ and $f^{high(n)}$ are assumed to be prime, the primality of the cover of f^n can be guaranteed by dropping the variable v_n ($\bar{v}_n$) from the cubes $c_i \in f^{high(n)}$ ($f^{low(n)}$) cover that can be made independent of v_n ($\bar{v}_n$) . Thus, a prime cover for f^n can be found as stated in the following theorem:

Theorem 5.1. Let n be a node of a MBD. Suppose that the low and the high sons of n have their associated functions represented by prime covers. Then, a *prime cover* that describes the function associated to n is obtained by dropping the variable v_n ($\bar{v}_n$) from the cubes $c_i \in f^{high(n)}$ ($f^{low(n)}$) that can have their ON set expanded over the $f^{low(n)}$($f^{high(n)}$) and merging with single cube containment the covers of $f^{high(n)}$ and $f^{low(n)}$ (sons' covers).

Proof. If the sons' covers are already prime, the only variable that can be dropped is v_n. The only exception is if a cube $c_i \in f^{high(n)}$ ($f^{low(n)}$) cover is contained by a cube $c_j \in f^{low(n)}$ ($f^{high(n)}$), and both cubes are independent of v_n. This is solved by testing the cubes for single cube containment when merging the two covers. ◊

The first step in obtaining a prime cover for the function associated to a node n is to look for the cubes that can be made independent of variable v_n. Let c_h be a cube that belongs to the high son node's cover, and c_l be a cube that belongs to the low son node's cover. The expansion of the ON set of a cube from the high (low) son's cover over the low (high) son's cover can be verified by testing the following cases:

1) if there are two cubes c_h and c_l that are identical, i.e., $c_h = c_l$, then these cubes can be merged and v_n and $\bar{v}_n$ are eliminated.

$$v_n c_h \vee \bar{v}_n c_l = c_h(v_n \vee \bar{v}_n) = c_h = c_l$$

2) if a cube c_h (c_l) is contained in some cube c_l (c_h), then v_n ($\bar{v}_n$) can be dropped from c_h (c_l).

$$c_h \leq c_l: \; v_n c_h \vee \bar{v}_n cl = v_n c_h \vee \bar{v}_n (c_h \vee c_l) = v_n c_h \vee \bar{v}_n c_h \vee \bar{v}_n c_l = c_h \vee \bar{v}_n c_l$$

Simplification occurs due to the Boolean property: $x \leq y \Rightarrow x \vee y = y$. A graphical representation of the expansion of c_l over c_h in a Karnaugh like diagram is shown in figure 5.7. The dashed lines shows the effect of dropping v_n and $\bar{v}_n$.

3) if a cube c_h (c_l) is contained by the low (high) son's cover, then v_n ($\bar{v}_n$) can be dropped from c_h (c_l).

$$c_h \le f^{low(n)}:\ v_n c_h \vee \bar{v}_n f^{low(n)} = v_n c_h \vee \bar{v}_n (c_h \vee f^{low(n)}) =$$
$$= v_n c_h \vee \bar{v}_n c_h \vee \bar{v}_n f^{low(n)} = c_h \vee \bar{v}_n f^{low(n)}$$

An example of a Karnaugh like diagram of property (3) is shown in figure 5.8.

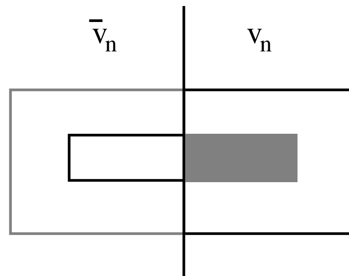


Figure 5.7. Cube expansion over another cube in an adjacent Boolean space.

The first two tests above are easy to implement when representing two-level functions by theirs covers. Cube identity and cube containment involves the comparison of only two cubes at a time. The third case concerns function containment and is costlier. In ESPRESSO a tautology checking algorithm is used to verify if a cube is contained by a cover. It should be noted that the three cases above are in growing generality. Testing the third one automatically implies testing the first two ones.

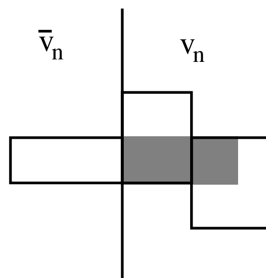


Figure 5.8. Cube expansion over a set of cubes in an adjacent Boolean space.

With the mixed sum-of-bdd-cubes representation, it is quite easy to implement the third option. The *imply* operation [Bry86] can be used to verify if a cube c is contained by a function f: $c \rightarrow f = \bar{c} \vee f = \mathbf{1}$. Figure 5.9 illustrates the idea. However, with MBDs there is a simpler way to verify this implication using the *restrict* operation. Restricting the MBD to the cube is equivalent to cofactor the function with respect to the literals that appear in the cube. Checking the resulting function provides the desired information. In the implementation section more details of the operation will be provided.

5.1.4 Irredundance

The first cover of node n, obtained according to the previous section, is already prime. To be irredundant, there must be no cube c_i in f^n such that c_i is contained by $f = f^n \setminus \{c_i\}$. Let us first analyze how redundant cubes can be created when merging the prime and irredundant covers of each son of *n*.

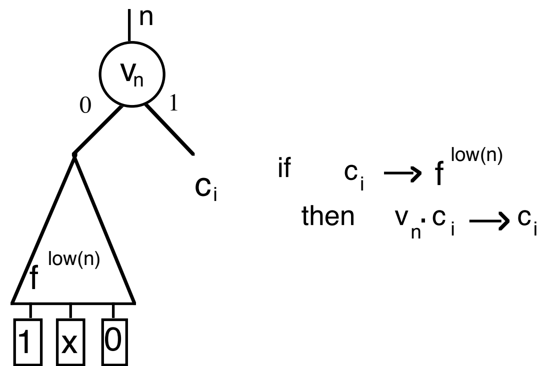


Figure 5.9. Test for cube primality.

When dropping the node variable v_n from a cube, it expands its ON set over the complemented side of the variable in the Boolean space, as illustrated in figure 5.7 and 5.8. In this case, some cubes in the complemented side can be totally or partially covered by the expanded ones. By consequence, some irredundant cubes can become redundant. Making the cover of *n* irredundant consists in identifying and removing those cubes that become redundant due to this process.

Some heuristics can be applied to simplify the identification of the redundant cubes. We do not need to look for redundant cubes inside one son's cover. Only those cubes who intersect expanded cubes from the brother's cover can be made redundant.

We can divide the cubes from each son’s cover into two categories: the expanded and the non expanded ones. We have then four groups:

- low expanded cubes
- low non-expanded cubes
- high expanded cubes
- high non-expanded cubes

Candidate cubes for elimination are those that intersect expanded ones:

- low cubes that intersect high expanded ones.
- high cubes that intersect low expanded ones.

We do not need to test low and high non expanded cubes between themselves, as they do not intersect.

The solution adopted here was to keep a list of intersection cubes for each cube. The whole process consists in a bottom up traversing of the MBD, starting at the terminal from which we want to obtain the cover. The cube's intersection list is updated as it climbs up the MBD. Cube redundancy is detected by testing the cube against its intersection list. More details of this procedure are presented in section 5.1.7.

5.1.5 Don't Care Minimization

When minimizing a incompletely specified MBD, the don't care set must be taken into account both in the test for cube primality and in the verification of cube redundancy.

The checking for cube primality is easily extended to the don't care case using the *restrict* operation. When the function is restricted to the cube we generate a new subfunction, which is independent of the variables in the support of the cube. The new subfunction is defined over the set of vertices contained in the cube. Figure 5.10 presents an schematic view of this operation by representing the cube as a window over a Karnaugh like diagram. If we do not consider the DC set, the cube imply the function when the function is a tautology over the cube domain, i.e., $\forall \mathbf{x} \in$ support(cube), f($\mathbf{x}$) = **1**. Taking the DC set into account then the cube imply the function if $\forall \mathbf{x} \in$ support(cube), f($\mathbf{x}$) $\in$ {**1**, **-**}, i.e., if the function is not **0** over any vertex of the cube domain.

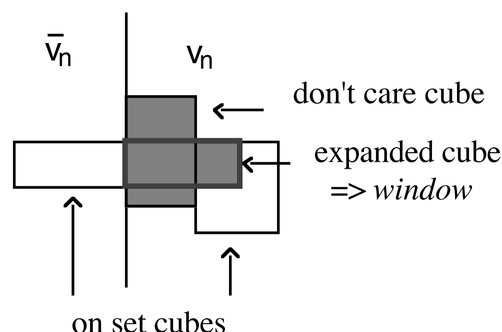


Figure 5.10. Cube expansion over the don't care and on sets.

The cube redundancy check is performed by keeping for each cube *c* in the subfunctions' covers a list of the cubes that intersect it. Any time the list is modified, *c* is compared against the cubes in the list to see if it is covered by then. The DC set can be present both in *c* and in the cubes of the intersection list. Indeed, a cube may be expanded over don't care points and these becomes part of the cube. If the ON set points of *c* are covered by the cubes in the intersection list, then the DC points can be switched to **0** and the cube is deleted.

Applying the extensions above allows the generation of prime and irredundant covers for incompletely specified functions.

5.1.6 Implementation

To compute the cover of a node function we have supposed that both sons have their associated covers already generated. To achieve this, we start processing the MBD at the target set's terminal and process all nodes in all paths that connect the root to this terminal. We create first a list of processing nodes, which is initialized with the **1** terminal. The method is general in the sense that, if we initialize the list with the **0** terminal, we get the cover of the OFF set. To process a node means to take all of its fathers, make a copy of its cover for each father and merge it with the fathers covers. These fathers are placed in the processing list and the node just processed is removed from it. The nodes inside the list are ordered by level, so the root is always the last node to be processed.

Node variable dropping (v_n) can be tested by verifying if each arriving cube from the sons covers implies the brother's function, i.e., if ($c_{il} \rightarrow f^{high(n)}$) then the variable v_n can be dropped from cube c_{il}. The cubes are stored into two lists: *exp* and *nexp*, corresponding to cubes that were expanded over the node variable and cubes that were not expanded, respectively. The "cube imply MBD" operation is implemented in the following way. Let $c = l_0 l_1 \ldots l_c$ be a cube, where l_i is a literal of c. If c is an implicant of a function f, then cofactoring f with respect to all $l_i \in c$ should produce a tautology.

As it was explained in a previous section, cube implication may be reduced to MBD restriction, which is a faster operation. The time needed to restrict a MBD to a set of variable values $\{l_i\}$ is constant [Bry86] and independent of the cardinality of $\{l_i\}$. It is basically the cost of traversing the whole MBD.

The redundancy verification is made by keeping all cube intersections of a cube in an intersection list. Cube intersection is verified when merging the sons covers. After merging the covers, each cube is tested against its intersection list. Each intersecting cube is subtracted from it. If the cube is reduced to the 0 terminal or is contained in the DC set of the node, it is redundant. Note that when subtracting two cubes, a sum of products form can arise. MBDs are nice to handle this problem because the MBD diagram can hold either a cube or a sum of products.

The function that does the main job is MBD_SOP. A sketch of the algorithm is shown in figure 5.11. The variable associated to a node n is referred as *n.var*. The function takes as parameters a list of *node_st*'s called *node_list* and a hash table *father_table*(k,d), where the keys are MBD nodes and the associated data is a list of the key node's fathers in the MBD. The *node_st* is an

auxiliary structure that holds a MBD node, the cover of the node function (herein called node's cover, for short) and the lists of expanded and non-expanded cubes of the node's sons. The list *node_list* is used to execute a breath first traversing of the MBD, starting by the terminal we want to compute the cover. The list is initialized with the desired terminal node. Each time MBD_SOP is called it processes the first node of *node_list* (the current node) and remove it from the list, discarding all data associated to it (the cover associated to the node, etc.). Then the fathers of the current node which are not already present in *node_list* are included in it, ordered by their variable index. The function is executed iteratively up to the root node is processed, which indicates the end of the algorithm.

```
function mbd_sop (node_list, father_table): list
var node: node_st;              /* current processing node */
     fl: MBD_node;              /* list of fathers of the node */
     fst: node_st;              /* father's node structure */
begin
    n := node_list[1];
    node_list := delete(node_list, 1);
    make_irredundant(n);
    fl := get_fathers(n, father_table);
    if (fl = nil)                        /* the empty list indicates the root */
    then return(n);
    foreach f in fl begin
        cover := make a copy of n.cover;
        if (f not in node_list)          /* first time f is acessed */
        then  begin
         fst := create a node_st for f;
         foreach cube in cover begin
            if (cube imply brother function)
            then put cube in fst.exp
            else fst.nexp:= fst.nexp+(n.var · cube)
            node_list := adjoin(fst, node_list);
            end; {foreach}
        end; {then}
        else begin                       /* f has one son already prosseded */
         fst := get fst from node_list;
         foreach cube in cover begin
            if (cube do not imply brother function)
            then cube:= add n.var to cube;
            if (cube not covered by fst.exp)
            then put cube in fst.cover;
            end;
        end; {else}
    end; {foreach}
end; {MBD_SOP}
```

Figure 5.11. MBD_SOP algorithm.

The current node *n* must have its cover already computed by a previous call to MBD_SOP. The first step is to make the cover of *n* irredundant, which is done by function *make_irredundant*(*n*). It scans all cubes in the *n*'s cover and check if each cube is not covered by the set of cubes that intersects it. After the current node is processed, then the fathers of the node will be handled. For each father *f* of *n* one out of two cases may arrive. Let *b* be the

node connected to the other branch of *f*, i.e., a *brother* of *n*. In the first case, *b* has not already been processed. Then, the cover of *n* will form an initial cover of *f*. The cubes that come from *n*'s cover will be tested against the function associated to *b* to check for cube primality. Those that have the variable *f.var* dropped are stored in a list of expanded cubes, *f.exp*. The cubes that are not expanded are stored in the list *f.nexp*. The second case is when *b* has already been processed. In this case, *f* has already an initial cover obtained from the processing of *b* and the cover of *n* is combined with the initial *f*'s cover. If a cube from *n*'s cover is expanded with respect to *f.var* then it is compared with the lists *f.exp* and *f.nexp* to find cube intersections. Cubes from *n*'s cover that are not expanded are checked against those from *f.exp* to find possible cube intersections.

5.1.7 Experimental Results

We have implemented these algorithms in Common Lisp and experimented MBD_SOP with the PLA test set from Berkeley. The results were compared with ESPRESSO. For each PLA we have selected the output that produces the greater MBD and applied to it the minimization algorithms. The results are shown in table 5.1. We see that MBD_SOP produces the same results as ESPRESSO most of the time. This is rather surprising, because MBD_SOP does not iterate over the final solution trying to jump out of a local minimum. Neither can it identify local minimum during the MBD processing. We estimate that the fact that the MBD is itself a prime and irredundant representation of a function could have led to these good initial results.

We have tried to analyze the effect of the MBD size on the performance of MBD_SOP. The MBDs were processed twice. First MBD_SOP was applied to the MBDs built with the original variable ordering. Then they were reduced with the incremental manipulation techniques presented in the previous chapter. The last columns of table 5.1 shows the results. We present the ratio of the size of the MBDs before and after the reduction and the ratio of the processing time to generate the two-level covers as well as the results in terms of number of cubes and literals. We can see that generally a smaller MBD for the same function requires less computing time. This is striking for the DUKE example, were the initial MBD was reduced by a factor of 0.24, and the computing time was reduced to 0.142 of the initial case. But DUKE is rather a pathological case where the initial variable ordering is too bad. A counter example is circuit 5XP1, where the diagram compaction lead to an increase in computing time. The problem is that the variable ordering affects the way cubes are expanded and merged and thus may alter the result obtained, eventually leading to a degradation in the algorithms performance.

The MBD could be considered as a look ahead structure to predict the binate variables selection in the unate recursive paradigm [Bra84]. The size of a MBD is not, nevertheless, a good estimation for computing complexity. If we analyze some circuits with similar MBD size as 5XP1, F51M, RD73 and M2, after reduction, we observe that they have covers that are far

different. The processing time in this case is proportional to the cover size rather than to the MBD size.

	ESPRESSO		BDD-SOP non-reduced			BDD-SOP reduced			Gain %	
Circuit	**Cubes**	**Lit.**	**Size**	**Cubes**	**Lit.**	**Size**	**Cubes**	**Lit.**	**Size**	**Time**
5XP1	18	82	25	19	89	19	19	89	0.76	1.167
ALU3	22	124	52	22	124	31	22	124	0.596	0.4
BW	6	21	15	6	21	12	6	21	0.8	1.044
DEKODER	4	6	8	4	6	7	4	6	0.875	0.754
DK27	2	12	15	2	12	9	2	12	0.6	0.269
DUKE	15	160	180	15	160	44	15	160	0.24	0.142
F51M	23	110	41	23	110	23	23	110	0.56	0.845
M1	5	21	15	5	21	13	5	21	0.867	0.771
M2	2	19	19	2	19	19	2	19	1	1
O2	3	9	8	3	9	7	3	9	0.875	1.1
P1	13	54	63	14	60	32	13	54	0.51	0.31
P82	5	18	14	5	18	13	5	18	0.92	0.778
RD73	42	252	22	48	288	22	48	288	1	1
RD83	10	40	14	14	56	14	14	56	1	1
RISC	3	17	12	3	17	12	3	17	1	1
SQN	17	85	38	17	85	31	17	85	0.826	1.03
X9DN	20	368	61	20	368	57	20	368	0.934	0.871
Z4	28	136	33	28	136	13	28	136	0.712	0.712

Table 5.1. Comparison with ESPRESSO.

5.2 Minimizing the MBD Size Using the Don't Care Set

More recently, the development of the application of programmable devices like FPGAs raised up the interest on new methods to synthesis and optimization programmable networks. The particular features of FPGAs open new fields of research targeted to specific technologies. Among them, the synthesis on selector based FPGAs using MBDs is one of the subjects of this work. The proposed method, described in a later chapter, relies on a graph covering approach that maps the MBD diagram directly to a network of multiplexor cells. In this case, the MBD size is a good estimation of the cost of the mapped network. It is well known that the size of the MBD diagram is closely related to the ordering of the input variables (see chapter 4). There is, however, a correlated problem that, up to the knowledge of the author, was never treated in the literature:

- given a incompletely specified function $\tilde{f}$, select a completely specified function $f_{min} \le f \le f_{max}$, that has the *minimum MBD size*.

The problem is how to use the don't care information to attain this goal. One way is to use two-level minimization techniques: first compute a prime and irredundant cover for the function and then use this cover to build the MBD of the now completely specified function.

But in fact the two-level logic's cost functions - number of cubes and literal count - do not directly relate to the MBD size. A smaller cover does not necessarily lead to a smaller MBD. This may be shown on the example of figure 5.12. The cover of function (a) is greater than the cover of function (b), but the MBD of (b) is greater than the MBD of (a). In consequence, the use of efficient two-level minimization techniques may be inefficient to tackle the MBD size's minimization problem.

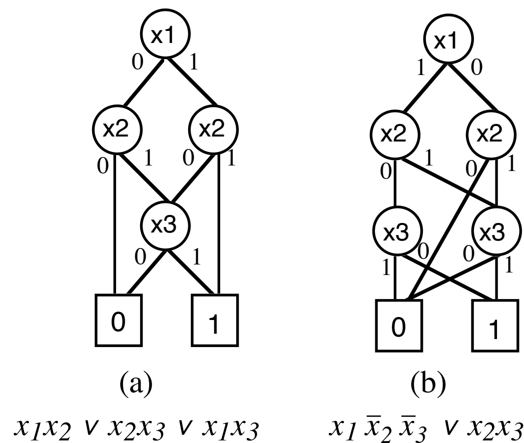


Figure 5.12. Sum-of-products and MBD representation of two functions.

The solution proposed here relies on the interrelation between graphs and logic properties in MBDs. The don't cares in the function's domain are used to locate redundant nodes in the MBD diagram that can then be removed, reducing the diagram's size.

5.2.1 The Subgraph Matching Approach

The method proposed here works directly on the MBD graph, using the **X** terminal in such a way that the number of nodes is reduced. The basic idea is to assign values to the don't care set and then to reduce the MBD by deleting redundant nodes that can result from the don't care assignment. A node is redundant in the MBD if either there is another node in the MBD that is equivalent to it or its low and high sons are equivalents. Remember that two nodes are equivalent if:

- they are the same leaf, or
- they have the same index and value, and their respective low and high sons are equivalent too.

The following property of MBDs is the basis of our minimization method.

Proposition 5.1. Let M be an MBD that represents an incompletely specified Boolean function *f* of *p* variables. Let *n* be a node in M that has the don't care terminal **X** as one of its sons.

Let i be the index of n. The **X** son of n can be replaced by any subfunction which depends on any subset of variables in $\Xi = \{x_j \mid i < j \leq p\}$.

Proof. The ordered nature of the MBD indicates that the subfunctions connected to the low and high branches of n depend only on variables with indices $i < j \leq p$. If one of its sons is **X** then it stands for a subfunction with $2^{(p-i)}$ minterms in the domain of f_{dc}. Its clear that **X** can be replaced by any subfunction in this domain by assigning appropriate values to each of its minterms.

Figure 5.13 puts in evidence this point. The low son of x_1 can be replaced by any subfunction depending on any subset of $\mathbf{X}_1 = \{x_2, x_3, x_4\}$. Note that, in an MBD, this is equivalent to replace the **X** terminal by any possible submbd depending on those variables.

The MBD minimization problem can then be stated as follows:

- find an adequate subfunction to replace the **X** terminal
- reduce the resulting MBD by deleting redundant nodes

The reduction of a completely specified MBD is identical to the reduction of ROBDDs, and is implemented as described in [Bry86]. Hence, the main problem in MBD minimization is the choice of a subfunction to replace the **X** terminal. If its father is a node n with index i, then there are $2^{2^{(p-i)}}$ candidate subfunctions to replace it. The exhaustive search is, of course, impractical. The problem is simplified by noting that the only subfunctions that can introduce redundancies in a MBD are those that already appears in it.

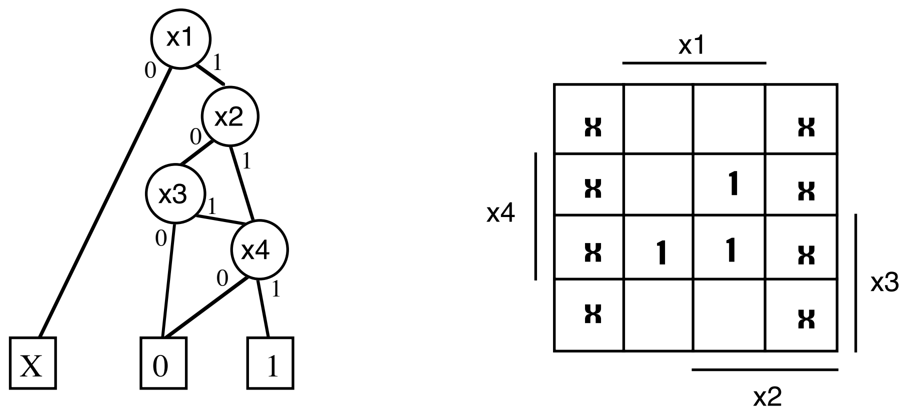


Figure 5.13. A MBD and its correspondent Karnaugh diagram.

Proposition 5.2. Let M be a MBD that denotes an incompletely specified Boolean function f of p variables. Let n, with index i, be a **X** leaf's father. Then, the set of submbds depending on $\mathbf{X}_i = \{x_j \mid i < j \leq p\}$ that can generate redundant nodes in M when replacing the **X** leaf is contained in the set $S = \{n_{[j]} \in M \mid i < j \leq p\}$, where $n_{[j]}$ means a MBD node with index j.

Proof. Redundancy in a MBD is based on node equivalence. In the general case, two nodes are equivalent if they have the same indices and values and if their respective low and high sons are equivalent too. Thus, a node *n* that has the **X** terminal as son can be make equivalent to other node *m* in M if and only if **X** is replaced by a son of *m* (low or high). ◊

The space solution for subfunctions is then reduced to the subset of submbds that start with nodes with indices greater than *i*. Put another way, the MBD minimization can be seen as a matching problem: two nodes are equivalent if it is possible to find a matching for the **X** terminal that make them isomorphic subgraphs.

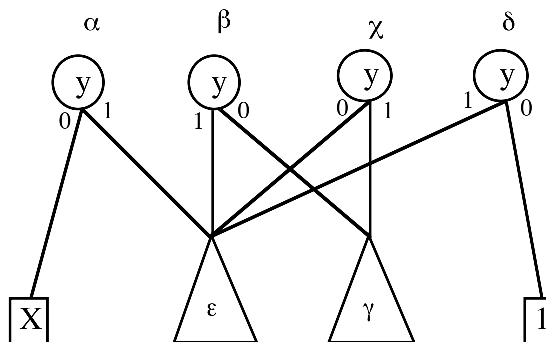


Figure 5.14 . Subbdds equivalence.

Figure 5.14 shows an example. Nodes α, β, χ and δ are submbds of an MBD, all of them with same index and value. Either node β or node δ can be made equivalent to node α, depending on the matching chosen to the low(α) = **X**. If low(α) = **X** is replaced by low(α) = **1**, then nodes α and δ become equivalent, and node α can be deleted, for instance. In this case, node α's fathers will be redirected to node δ. In trivial cases, **X** can be either replaced by its brother - in this case the father is redundant and can be replaced by the **X**'s brother - or simply by the **0** or **1** constant. The objective here is to find a matching to **X** that maximizes the reduction of the MBD. Consider still figure 5.14. If node α is made equivalent to node β or δ, or if low(α) = **X** is replaced by low(α) = ε, then node α will became redundant and will disappear when the MBD is reduced. It is not however, the only node that can be eliminated. The redirection of the branches from the fathers of node α to the node that replaces it may produce new redundant nodes among node α's fathers and the reduction proceeds.

Figure 5.15 gives another example that illustrates this point. In this graph, when replacing low(β) = **X** by low(β) = δ in MBD (a), we have low(β) = high(β) and node β becomes redundant. Replacing β by high(β) we get the MBD (b). However, if low(β) = **X** is replaced by the low(β) = ε, we have that low(β) = low(χ) and high(β) = high(χ). In this case, β and χ are equivalents. Thus, low(α) ≡ high(α) and node α is redundant. Replacing α by high(α) we get the MBD (d). We say that the redundancy has propagated through node β up to node α.

In general, it is hard to predict the reduction achieved when replacing **X** by another subgraph without reducing the entire MBD. It should also be noted that a **X** leaf can have several fathers. In this case, a **X** leaf is associated to each father and can be matched independently of the others. Hence, to find the best reduction we must evaluate all possible combinations of **X** matching.

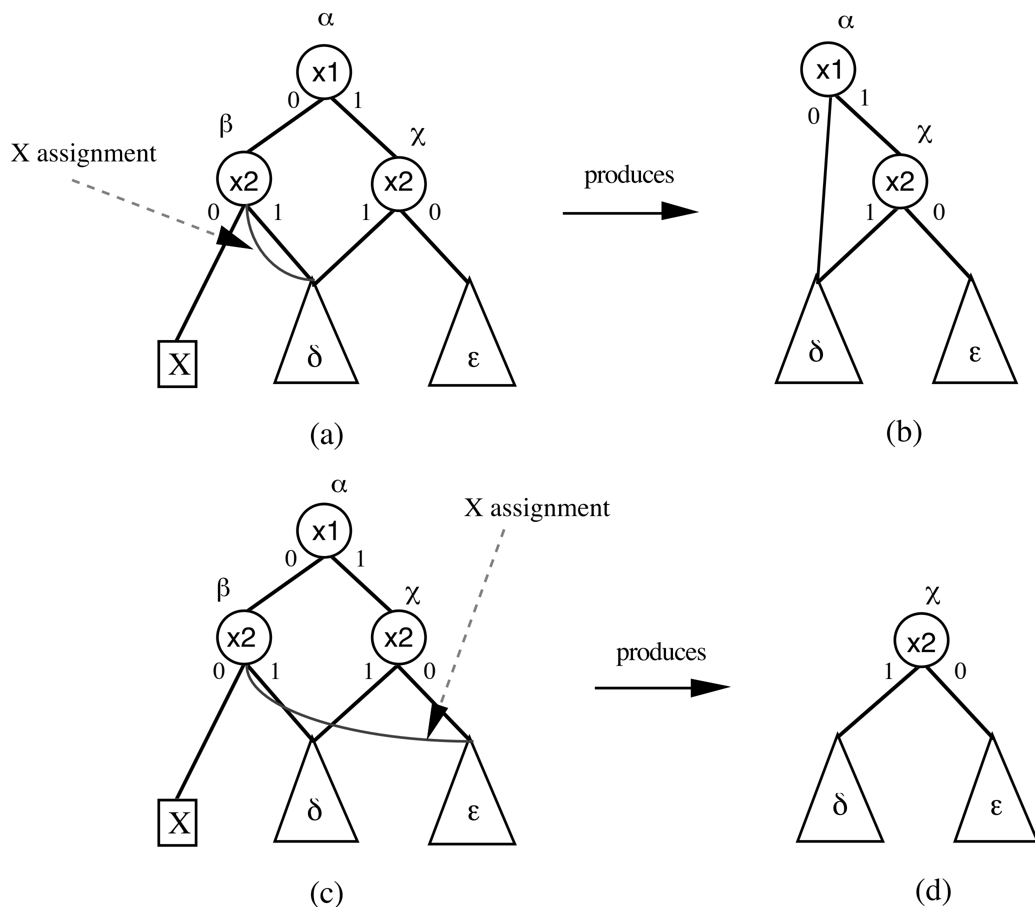


Figure 5.15. Alternative assignments.

5.2.2 Subgraph Selection

The approach presented here consists in finding all the subgraphs that can be matched to the **X** leaf and then selecting the matches that produce the smallest MBD. The set of candidate submbds to match the **X** leaf is defined in proposition 5.2. However, for a given **X** father n, the number of candidates may be significantly further reduced. Indeed, as we look for the generation of equivalent nodes in the MBD, only the *i-brothers* of n (nodes at level l_i) must be compared against n. Nodes with different indices can not be equivalent.

The subgraph (submbd) selection can be divided into the following steps:

- find all nodes that have the **X** leaf as one son

- for each of these nodes find the set of equivalent nodes in the MBD
- select the set of matchings that produces the smallest MBD

Finding all **X**'s fathers is trivial. One must traverse the MBD and look at the node's sons. To build the set of equivalent nodes for a **X**'s father *f* we must test it against each of its brothers. Subgraph equivalence is checked by means of a graph isomorphism verification algorithm that includes the don't care processing. Selecting the set of subgraph matchings that produces the smallest MBD is not a trivial task. The subgraph interactions must be taken into account when evaluating the possible matches.

Let L_j, $1 \le j \le p$, be the set of nodes at level *j*, where *p* is the number of input variables. Let $\{n_i^x\}$ be the set nodes n_i in L_j that are **X**'s fathers, and $E_i = \{m_k \mid m_k$ is equivalent to $n_i^x\}$ be the subset of i-brothers(*j*) that match n_i^x. For each m_k there is *a set of* **X** matchings that makes m_k equivalent to n_i^x. This set is composed by X leaves directly connected to n_i^x and also by all **X** leaves that can be reached from nodes n_i^x and m_k. Remember that the heuristic adopted deal only with the **X**'s fathers and do not consider the others **X**'s ancestors. The point here is that, if n_i^x is matched against a i-brother m_k, some **X** leaves that can be reached from n_i^x and from m_k can be matched too. This means that, if there is a **X**'s father f which can be reached from n_i^x or m_k, the matching of n_i^x with m_k may imposes a match to f, and in this case we must forget all other possible matches of f with its brothers. This is illustrated in figure 5.16.

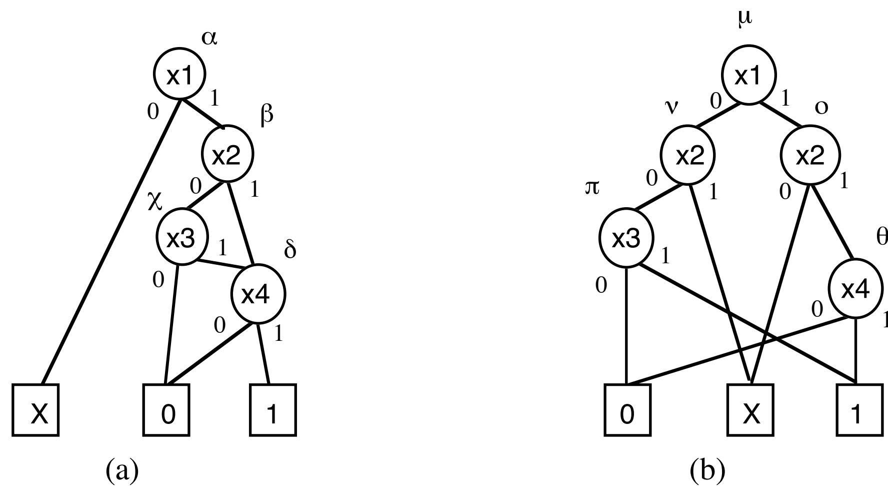


Figure 5.16. X leaf matchings.

Subdiagrams (a) and (b) belongs to the same MBD. To match submbds (a) and (b) we must set low(α) = ν and low(ο) = χ. This is illustrated in figure 5.17. Subdiagrams χ and ν appear duplicated for easy of visualization. The subdiagram that results from the matching is shown in (c). Thus, if α and μ are matched, node ο can not be further matched with another node at

level 2. However, if α and μ are not matched then node ο can be treated independent of graph (a).

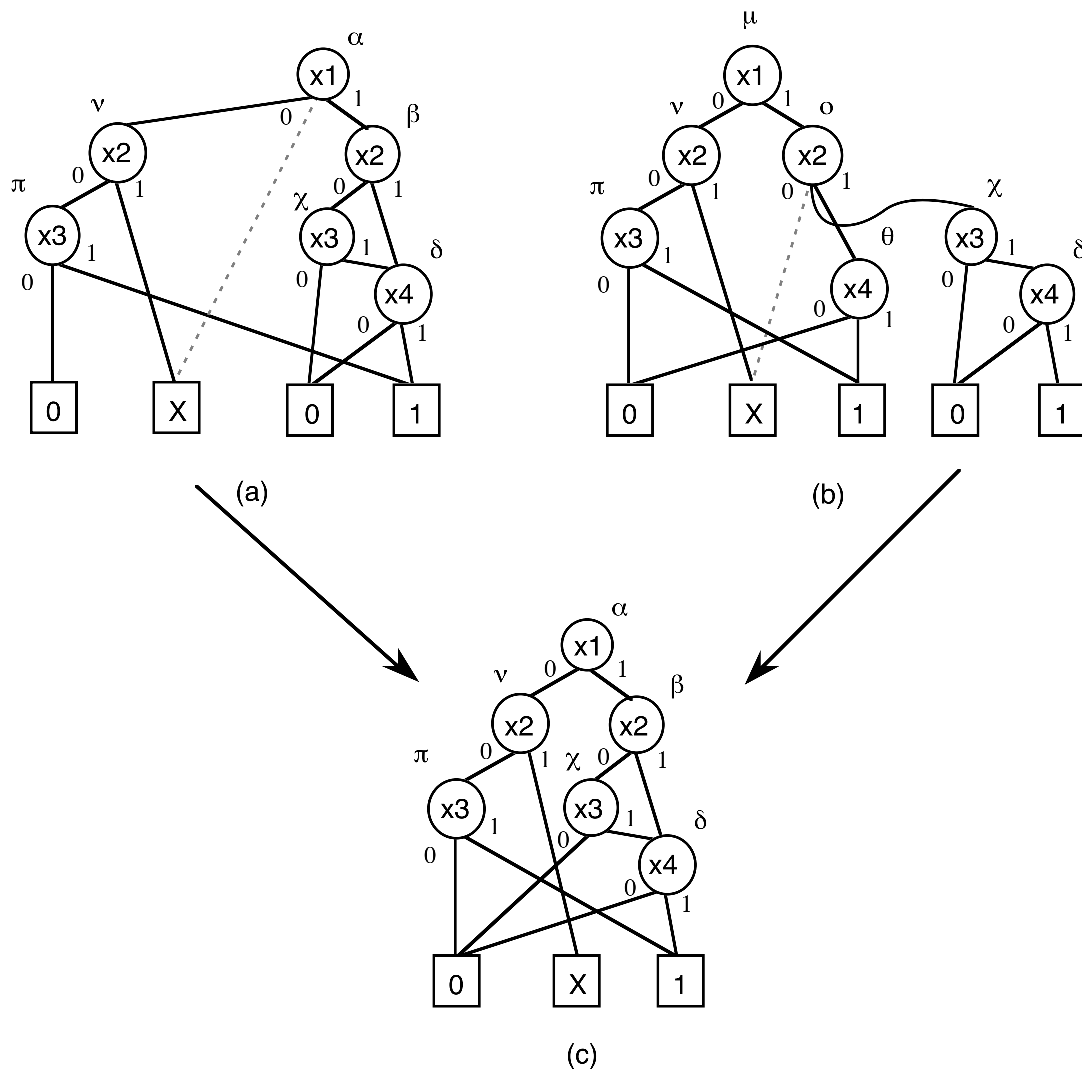


Figure 5.17. MBD matching.

Definition 5.3. A *bound set of assignments* of two subgraphs s_1 and s_2 is the set of assignments of the X leaves needed to make s_1 equivalent to s_2.

In the previous example, the bound set of assignments of subgraphs (a) and (b) is {(low(α), ν), (low(ο), χ)}. The term *bound* enforces the fact that these assignments must be made simultaneously to guarantee the equivalence. Note that high(ν) do not need to be matched to make (a) and (b) equivalent, because it was introduced in subdiagram (a) by a previous assignment (low(α) = ν) that made (a) equivalent to (b). To say in other words, node ν can be independently matched with any other i-brother of the MBD that contains (a) and (b), because it does not belongs to the bound set of assignments that makes (a) equivalent to (b).

To find the set of **X** matches that produces the smallest MBD we compute all possible combinations of **X**'s fathers assignments. For each **X**'s father we build a list of equivalent nodes. For each equivalent node we build the bound set of assignments. We start the matching from the root *down to* the terminals of the MBD. For each match we remove from the **X** fathers set all fathers that are involved in the bound set of assignments of the match. This restrict the search by reducing the matching possibilities of the affected **X**'s fathers. When the **X**'s father set is empty, we reduce the MBD and evaluate its size. The set of assignments that results in the smallest MBD is returned as the desired result.

5.2.3. Algorithms

Two auxiliary data structures are used: a fathers-table, which is a hash table with the node pointers as keys and the list of their father nodes as data and a brothers-array, which is an array of lists of i-brothers. The main function is MBD_DC. It takes an MBD with a **X** terminal and reduces it by applying the subgraph matching technique. A sketch of the algorithm is shown in figure 5.18. The main functions are *make_eqvlis* and *get_best_assgn*. The first one finds all equivalent nodes for the **X**'s fathers and builds the bound set of assignments for each match. The second one evaluates the set of assignments and chooses those that result in the smaller MBD. Finally, the selected set of assignments is applied to the MBD. Then it is reduced and returned as the result of the function.

```
function MBD-DC (mbd) : MBD_TYPE
var mbd: MBD_node;
begin
  build up Brothers_Array and Fathers_Table;
  X_node = find_node_in_mbd(mbd, value, X);
  X_Fathers = get_fathers(X_node);
  /* for each X_Father make the list of equivalent nodes
     and the associated binded assignment sets */
  eqvlis = make_eqvlist(Brothers_Array, X_Fathers);
  best_assgn = get_best_assgn(mbd,eqvlis,X_Fathers,null);
  mbd = reduce(apply_assgn(mbd, best_assgn));
  return(mbd);
}
```

Figure 5.18. MBD_DC algorithm.

Make_eqvlis simply takes each **X**'s father and finds its list of equivalent nodes by successive calls of *match_MBD*. *Match_MBD* works basically as the unification function of PROLOG. If the two MBDs are identical, it returns TRUE. If they can be made equivalent by matching **X** leaves then the function returns the list of matched nodes. If the two MBDs are not equivalent, then the function returns FALSE.

The result of *make_eqvlis* is a structure that contains for each **X**'s father, the list of equivalent nodes. Each equivalent node is in fact a structure composed by the following fields:

- EQV: the equivalent node.
- ASSGN: the list of **X**'s fathers that must to be matched.

The function *get_best_assgn*, shown in figure 5.19, recursively processes the list of **X**'s fathers for every set of assignments. When the list is empty there is a complete set of **X** assignments done and the MBD is reduced to evaluate its size. The best result is stored and returned as a bound set of assignments.

```
function get_best_assgn (mbd, eqv_lis, X-fathers, assgn)
var x_node:     MBD_vertex;
    eq_nlis:    list;
    eq_node:    MBD_node;
begin
   if (empty(X_fathers)) then begin
      evaluate_assgn(assgn);
      size = count_nodes(redude(mbd));
      if (size < Best_Size)
      then begin
         Best_Size = size;
         Best_Assgn = assgn;
      end;
   return();
   end;
   x_node = first(X_fathers);
   X_fathers = rest(X_fathers);
   /* get all i-brothers that are equivalent to x_node */
   eq_nlis = get_eqv_nlis(x_node, eqv_lis);
   for each eq_node in eq_nlis begin
    x_f = remove matched x_fathers from X_fathers;
    /* add to the current set of assignments the set obtained from eq_node */
    assgn_aux = append(assgn, get_assgn(eq_node));
    /* proceed to match the remainding x_fathers */
   get_best_assgn(mbd, eqv_lis, x_f, assgn_aux);
   end;
   return(Best_Assign);
end;
```

Figure 5.19. Get_best_assgn procedure.

5.2.4. Experimental Results

We have implemented a prototype version of the algorithms in common LISP in our system. To simplify the generation of the functions to test the algorithm we have selected them among the outputs of incompletely specified PLAs from a PLA benchmark set. The idea is to *simulate* a multi-level circuit by representing each circuit subfunction by a PLA output. There are some advantages with this approach. First, we have a rich variety of subfunctions, possibly far different each other. Second, they are *known* subfunctions, not a set of anonymous ones produced by some decomposition method.

The subfunctions are optimized with respect to their don't care sets and then they are mapped into ACTEL cells. The cost function to be minimized is the size of the MBDs that represent the subfunctions, which is a good estimation of the final circuit cost in terms of multiplexor cells.

We have compared MBD_DC against ESPRESSO [Bra84] for the minimization of the MBD size. When using ESPRESSO, we start with a sum-of-products representation of an incompletely specified subfunction that is minimized to generate a prime and irredundant cover. Then its MBD representation is built. When using MBD_DC, the don't care set of a subfunction is directly included in its MBD representation. Then the subfunction is optimized with MBD_DC. In both cases the variable ordering is the same. After being minimized, the MBDs are mapped onto ACTEL cells. The results are shown in table 5.2.

example	initial MBD	espresso	MBD_DC	% gain
apla0	35	29(15)	17(7)	41(53)
apla1	34	28(14)	16(7)	43(50)
apla2	30	15(7)	12(7)	20(0)
bcd0	8	5(1)	5(1)	0(0)
bcd1	10	9(3)	7(2)	22(33)
bcd2	11	10(4)	8(3)	20(25)
bench0	24	6(1)	5(1)	17(0)
bench1	29	7(3)	7(3)	0(0)
bench2	35	18(8)	15(6)	15(25)
dk170	38	21(8)	10(4)	52(50)
dk171	35	21(7)	10(3)	52(57)
dk172	32	18(7)	9(4)	50(43)
dk270	25	8(2)	7(2)	13(0)
dk271	25	8(2)	7(3)	13(-33)
dk272	32	15(5)	7(2)	53(60)
Total	428	218(87)	142(55)	35(37)

Table 5.2. Comparison between Espresso and MBD_DC

The *initial MBD* is the subfunction's MBD with don't cares included. It is the starting point for the MBD_DC algorithm. The *espresso* and *MBD_DC* columns show the size of the minimized MBDs and, in parenthesis, the number of ACTEL cells after technology mapping. We have used the same mapper in both cases. The column *gain* compares the results of ESPRESSO and MBD_DC. For the MBD size, MBD_DC wins in all but two cases, where ESPRESSO obtained equivalent results. For the number of ACTEL cells, MBD_DC lost in one case, even though the MBD it has generated is smaller. This occurs because the mapping is affected not only by the MBD size but also by its specific topology. Considering this set of subfunctions as a single multi-level circuit, the minimization of MBDs with MBD_DC has saved 35% in terms of MBD size and 37% in terms of number of ACTEL cells with respect to the minimization with ESPRESSO.

5.3 Comments

In this chapter two minimization methods for incompletely specified functions represented by MBDs were presented.

In the case of the two-level minimization, it was introduced a mixed representation of functions, the *sum-of-mbd-cubes* that provides a uniform way to deal with both MBDs and two-level covers. The algorithm is flexible in the sense that it can generate a cover for either the ON or the OFF sets by just selecting the starting terminal and exchanging the terminal values. The minimization process is exactly the same for both cases. An additional interesting feature is that in this approach we do not need to compute explicitly the OFF set of the function to perform the minimization. In multi-level logic optimization node functions may have large don't care sets, which can produce huge OFF sets. This is a serious drawback for minimizers such as ESPRESSO that computes the OFF set in the simplification process. It should be noted that recent versions of ESPRESSO where modified to tackle this problem [Mal89].

The cost of the generation of the sum-of-products is difficult to estimate. It is more related to the cover size than to the MBD size. The specific topology of the diagram is also important because it dictates the way cubes are merged when computing the covers of the MBD nodes. The quality of the results was similar to those obtained by ESPRESSO, which is rather surprisingly due to the sophistication of its algorithms.

Next, a new method was presented for the minimization of incompletely specified MBDs targeted to the reduction of its size, which is an important factor in the case of selector-based FPGA synthesis. The method is based on an algorithm that uses the **X** terminal to match subgraphs in the MBD in order to maximize the number of redundant nodes that, when deleted, will reduce the diagram size. The algorithm is quite complex and the solution proposed is exhaustive in the sense that it tries all possible matchings. It can, thus, be time consuming for large examples. As in the two-level case, the time complexity is not directly related to the MBD size, but depends on the number of nodes connected to the **X** terminal and on their topology. The matching itself is not the bottleneck of the algorithm, but the selection of the best set of matchings can be quite costly if the number of bound set of assignments is large.

The benchmarks results have confirmed that the MBD size minimization with subgraph matching produce better results than the application of two-level techniques. This is a rather natural outcome of the lack correspondence between the two-level cost function and the size of the MBDs.

Chapter 6

FPGA Synthesis

This chapter presents a method for the synthesis of selector based FPGAs circuits. The initial MBD is reduced using the techniques described in chapters 4 and 5. Then a subgraph resubstitution phase takes place that tries to replace isomorphic subgraphs by new variables. The multiplexor cells are represented by small MBDs. The MBD diagram is then covered with these small MBDs and a netlist of multiplexor cells is obtained.

It is well known that the design of a digital system is guided by a set of competing targets. The choice of the design style to implement the circuit is driven not only by performance aspects but also by economical purposes. Fast turn around time (the time to design and implement a circuit in a given technology), for instance, is very important factor in the highly concurrent market of digital electronics. Another key factor is the amount of devices produced. Circuits produced in large scale must be strongly optimized to reduce fabrication costs. On the other hand, circuits fabricated in small scale should privileged the reduction of the development costs.

The set of design styles available to the designer today reflects this diversity of constraints. Gate arrays, standard cells and full custom technologies provide different trade off between device density and turn around time. Gate arrays allow for faster but less complex designs than full custom, for instance. Another important category is the Programmable Logic Devices (PLDs). A PLD is a logical device that can be programmed to implement a variety of different logic functions. Its main features are:

- fast turn around time (from some minutes to a few hours)
- low cost computational resources (most designs are made in personal computers)

• fast and simple correction of design errors
• flexibility of application

The first PLDs were introduced in the beginning of the 70's, the Programmable Read-Only Memory (PROM). In 1975 Signetics introduced the Programmable Logic Array (PLA), which had (and still has) a large acceptation in the market. The Programmable Array Logic (PAL) developed by MMI appeared in 1978. Its is composed by a programmable AND matrix connected to a fixed OR matrix. In 1985 Lattice proposes the Generic Array Logic (GAL), that is an extension of the PAL but with the additional feature of being electrically erasable. Since then new devices had been introduced with crescent complexity, including features as internal registers, feedback loops, output polarity and so forth.

A major restriction of these devices was the low density of integration. However, the technological evolution associated to new pre-diffused architecture is changing this panorama. Two of the main major advances in the PLD area were the introduction of the Logic Cell Array (LCA) by Xilinx [Xil92] and the multiplexor based cells by ACTEL [ElG89], which were afterwards denoted Field Programmable Gate Arrays (FPGAs). The FPGAs can be viewed both as an evolution of PALs, where size is increased by an order of magnitude as well as a refinement of mask programmed gate arrays, where reprogramming time and cost are drastically reduced. These features lead to a increasing acceptation and use of those devices by the electronic industry.

Xilinx architecture consists in an array of Configurable Logic Blocks (CLBs) that can be connected by a network of programmable interconnections. The circuit interface is controlled by Input/Outputs interface Blocks (IOBs). Each CLB is a rather complex block if compared with the previous PLDs. In general it is composed by two logic blocks that can implement any function up to 4 variables (series 3000/4000) plus a set of multiplexors, two D flip-flops and some feedback loops. One of the main features of Xilinx devices is its complete reconfigurability. Both the interconnections and the logic functions implemented by the CLBs can be reprogrammed. Interconnections are implemented with FETs controlled by SRAM cells. Thus, the same device can be reused to realize a completely different logic function.

ACTEL cells, on the other hand, provide a completely different approach. Its structure is similar to the mask programmable gate arrays, formed by strips of logic cells separated by interconnection channels. Each logic block provides a tree of three 2 to 1 multiplexors. The interconnections are programmed by means of fuses (or anti-fuses) that connects vertical and horizontal segments. A fundamental difference with respect to Xilinx devices is that the fuses are not reconfigurable. The lack of flexibility of this approach is compensated by a greater device density due to the smaller size of the fuses.

Each type of FPGA device has its advantages and disadvantages [Hil92]. The multiplexor based nature of the ACTEL devices had motivated us to develop synthesis techniques based on MBDs due to the direct correspondence that exists between MBD nodes and multiplexor cells.

The first mapping systems that address the ACTEL technology used the standard cells based approach [Mai88], [Bra87]. This requires the creation of a library with the set of all possible gates that can be simulated by the ACTEL cell. Each cell implements all functions with two variables, 243 functions of 3 variables [Kar91], and so forth. It results in a large and complex library.

The problem can be simplified by choosing logic representations that are closer to the target device. Amap [Kar91], for instance, is based on If-Then-Else DAGs (ITE), the data structure created by Karplus [Kar88] as an extension of BDDs. The main difference between a ITE and a BDD is that in the former the decision function associated to a node is not a single variable, but an arbitrary Boolean function. If we restrict the decision function to a single variable, then the ITE becomes equivalent to a BDD. Amap constructs an ITE for each function and covers it with a greedy approach. The algorithm exploits the similarity between multiplexors and ITEs to achieve a very fast mapping. Proserpine [Erc91] and ASYL-FPGA [Ben92] are BDD based FPGA mappers. They differ in the way BDDs are used to cover the functions. In Proserpine the circuit is represented by a Boolean network, where each node function is described by a BDD. The matching algorithm checks if the node function can be covered by ACTEL cells by looking for graph isomorphism between the BDD of the node and a BDD representation of the multiplexor cell. In the search of a solution several orderings for the node BDD are tried, up to find a good match. In ASYL-FPGA the Boolean functions are represented by a single multi-rooted BDD, and the mapping consists in covering it with the BDDs of the multiplexor cells.

Another mapper is Mis-PGA [Mur92], which combines different alternative techniques in the search of the best solution. It uses in fact three mapping methods. The initial circuit is represented by a Boolean network. In the first step the node functions are mapped from its sum-of-products representation. If a node can not be mapped into a single cell then its function is re-expressed as a ITE DAG. The ITE is then implemented using a graph covering approach that matches it with a ITE representation of the cells. A final improvement is tried by collapsing the circuit into a single multi-rooted BDD and then by mapping it directly with the BDD of the cells. The best result is returned.

Our approach is similar to that of [Mur92] and [Ben92]. In this case the cost of the final circuit is proportional to the size of the MBD. The idea is to apply the techniques shown in chapters 4 and 5 to reduce the size of the MBD, which should heuristically minimize the final circuit, and then to map the reduced MBD the multiplexors using a graph covering approach.

The global process is depicted in figure 6.1. The starting description is the MBD of the function to be synthesized. The graph covering technique associates nodes in the diagram with multiplexors in the cells. The relationship among nodes and multiplexors is not one to one because each ACTEL cell contains three multiplexors and only one of them, the root, is externally accessible. The consequence is that some multiplexors of the ACTEL devices will not be associated to MBD nodes. Anyway, the size of the MBD is still the most important parameter in the estimation of the implementation cost in this kind of synthesis.

The first step in the synthesis is to reduce the size of the MBD using the incremental manipulation techniques presented in chapter 4. This first step is based on the sensitivity of the MBD size with respect to its input variable ordering.

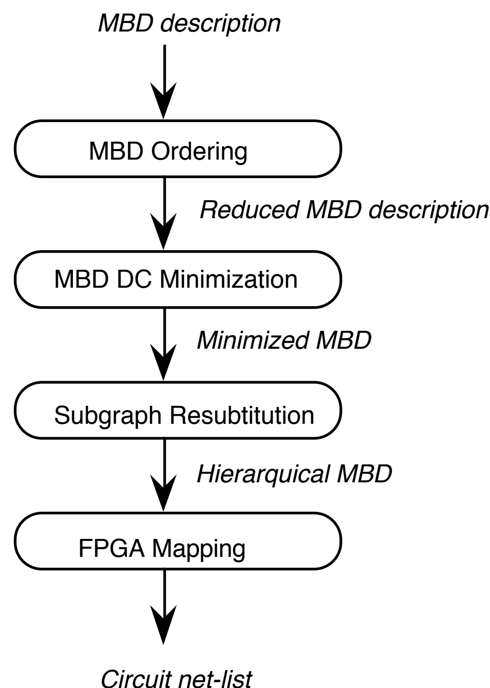


Figure 6.1. FPGA synthesis process.

Next step tackle the case of incompletely specified functions, which are minimized using the subgraph matching approach presented in chapter 5. This method takes the number of nodes of the diagram as cost function and tries to minimize it by replacing the X terminal by other subgraphs in the MBD. The resulting diagram is submitted to a resubstitution process that identifies isomorphic subgraphs in the MBD that can be replaced by a single node associated to a new variable. This produces a hierarchical MBD, where some input variables can represent subfunctions described by subMBDs. The final step is the mapping into ACTEL cells, which is performed using a graph covering approach. The rest of the chapter gives a more detailed view of these processes.

In the next section, the architecture of the ACTEL cells is described. The subgraph resubstitution method is presented in the sequel. Finally, the mapping into ACTEL devices is outlined.

6.1 ACTEL Devices Architecture

The ACTEL devices combines some of the flexibility of mask programmable gate arrays and the convenience of field programmable devices. The basic logic block is based on multiplexor devices that can implement a large set of Boolean functions through an adequate assignment of values to the logic block inputs. The channeled structure eases the task of CAD routing tools, while the electrically programmable architecture improves its flexibility of application.

The architecture of the chip is sketched in figure 6.2. It consists in a set of rows of logic blocks separated by routing channels. The interconnections are realized by programming the fuses that connects predifused horizontal and vertical wire segments. The fuse is a special structure that present a high impedance in its original state. It can be turned on by the application of a high voltage across it. The fuses have a relatively low resistance in the conducting state (≈ 500 Ω) and are relatively small.

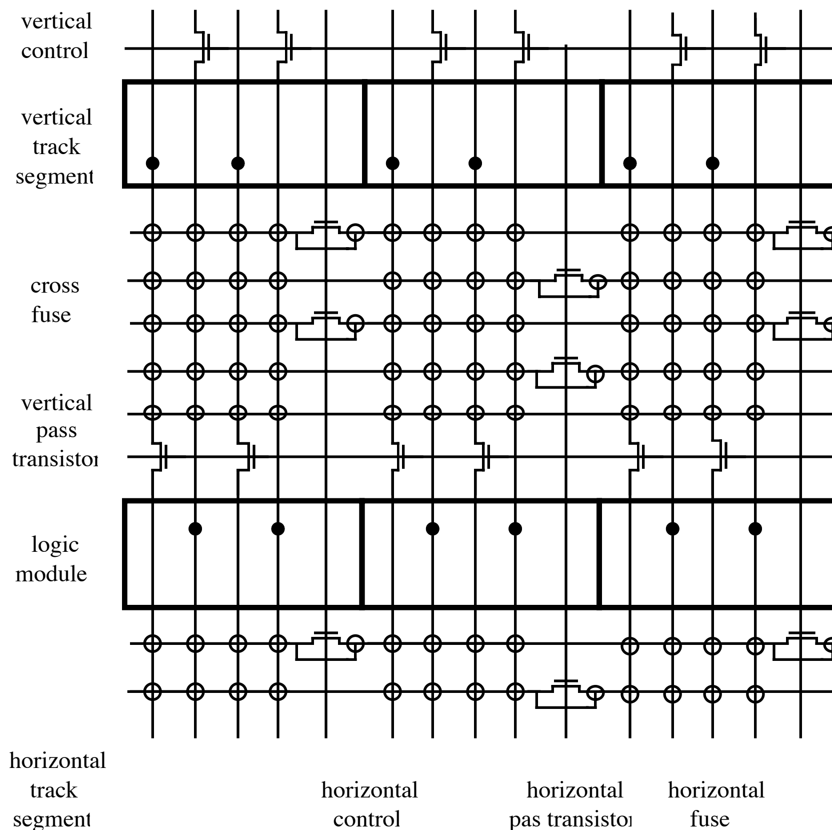


Figure 6.2. ACTEL chip architecture.

There are two kind of fuses. Cross fuses controls the interconnection of vertical and horizontal segments. Horizontal fuses control the interconnection of horizontal segments. Programming the fuses is simplified by an efficient addressing scheme that uses the wiring themselves, pass transistors connecting adjacent segments and control logic at the periphery of the array.

The logic block architecture is shown in figure 6.3. It consists on a tree of two to one multiplexors. The inputs of the first stage multiplexors as well as the control signals of all of them can be programmed by the user. The second stage multiplexor's control signal is driven by an OR gate, which enhances the flexibility of the logic module. The user can program the logic function implemented by the block by selecting appropriate values for those inputs. The logic block can realize any function with two inputs, most of the functions with three inputs, and so forth, up to a single function with eight inputs.

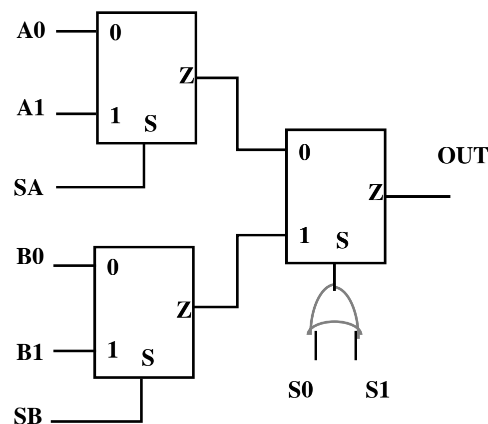


Figure 6.3. Logic block architecture.

The ACTEL chip present some limitations specially in term of routability. The fuse resistance together with the parasitic capacitance's of the segments form a RC tree. To avoid an unacceptable growth of the circuit delay the number of fuses in a path that drives any input is restricted to three. Another restriction is that the inputs of a logic block are accessible either from the channel above or below but not from both of them. This handicap was minimized by using a flexible cell, where most logic function can be implemented in different ways, using distinct inputs. Selecting which implementation meets the routing constraints is the task of the router.

6.2 Subgraph Resubstitution

The subgraph resubstitution step can be seen as a kind of logic decomposition. The idea is to find some subgraph configurations that may be replaced by a single node in the MBD. The subgraphs extracted form new subfunctions, associated to intermediate variables that are introduced into the original MBD. This process can be quite complex. We shall introduce it through a set of examples for the sake of simplicity.

A simple example is given in figure 6.4. The two subgraphs in 6.4(a) belongs to the same MBD M. They have the same structure, the edges have the same labels (0 and 1) and they depend on the same variables. Moreover, each subgraph has one root and only two descendants. If we make a copy of one of these subgraphs and assign Boolean constant values **0** and **1** to its descendants we obtain a subMBD that denotes a subfunction of M (figure 6.4(b)). We then choose a intermediate variable to stand for this function, say y_1, and replace the subgraphs on M by a single node depending on y_1, as depict figure 6.4(c). We obtain a new MBD M', where the subgraphs of figure 6.4(a) were replaced by subgraphs of figure 6.4(c). The size of M' is two less the size of M, because we replaced two subgraphs of size two by two subgraphs of size one. However, a new subfunction was introduced, and the global size of M' is size(M') + size(MBD of y_1). If we discard the terminals, then the global size of M' is the same of M and there was no gain with this resubstitution.

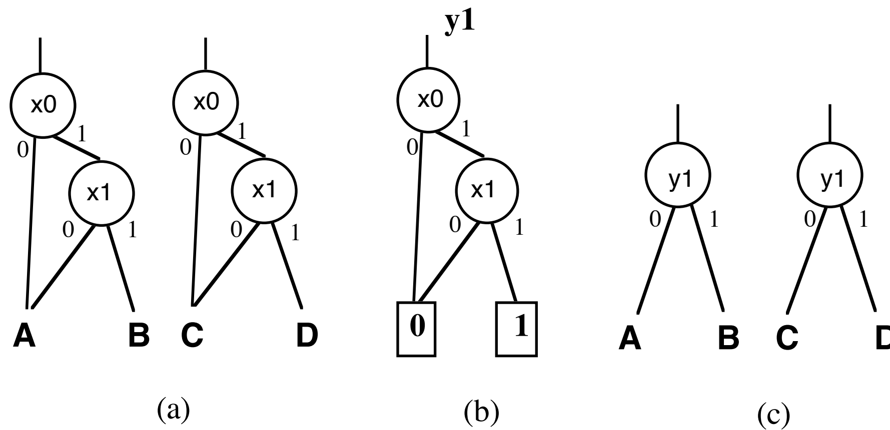


Figure 6.4. A simple subgraph resubstitution.

This simple example introduces some of the main problems with subgraph resubstitution. The first one is the identification of isomorphic subgraphs, and the second one is the evaluation of the gain obtained. The isomorphism detection algorithm must consider the subgraph structure, the labels of the edges and also the variables associated to each node. Therefore, if the node connected to B in figure 6.4(a) was associated to a variable different than x_1, then these subgraphs will not be isomorphic. On the other hand, the gain of must be evaluated before executing the resubstitution. Note that there can be several possible resubstitutions inside a MBD and some of them may be incompatible. The gain evaluation must consider only compatible resubstitutions.

The subgraph isomorphism algorithm is similar to the MBD equivalence one, with some additional constraints derived from possible interactions between the subgraphs. Let us make an initial analysis of the problem in order to get a felling of the type of difficulties that must be tackled. Equivalent or isomorphic subgraphs may appear in any region of the MBD. They must have a single root, otherwise they could not be replaced by a single node. Another restriction

is that any set of equivalent subgraphs must start at the same MBD level. This is exemplified in figure 6.5, where two sets of isomorphic subgraphs are shown. The first set S_1 is composed by subgraphs whose roots are associated to variable x_0. The second set S_2 is that whose subgraph's roots are associated to variable x_1. As S_1 and S_2 have a non empty intersection, they are *incompatible subgraphs*. This means that we can resubstitute one of them, but not both. Note that we can have more than a set of equivalent subgraphs that start with roots with same index. This is shown in figure 6.6. In this case, the subgraphs are compatible, even though they can share some nodes. These nodes do not preclude the resubstitution, but will affect the gain evaluation. In the follow we define more precisely the terms used informally up to here.

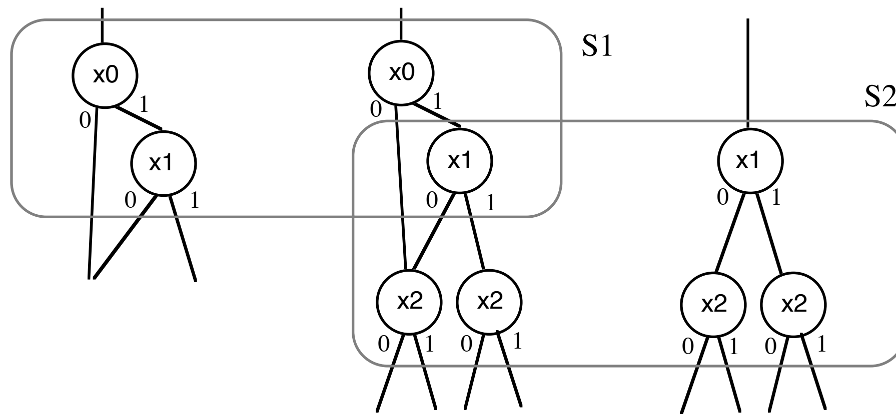


Figure 6.5. Sets of isomorphic subgraphs.

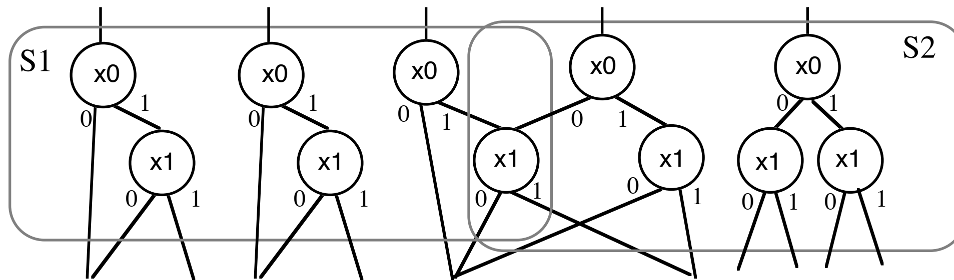


Figure 6.6. Two sets with the same root.

Definition 6.1. Two subgraphs s_1 and s_2 of a MBD M are *single-output-equivalent* (*s-out-eqv*, for short) if the following conditions are verified.

- s_1 and s_2 have a single root node
- all the edges that leave s_1 (s_2) are connected to only two distinct nodes of M
- the MBDs associated to s_1 and s_2 are logic equivalent under as adequate choice of its descendant nodes.

A set of s-out-eqv subgraphs $S = \{s_i\}$ of a MBD M is referred as a s-out-eqv class.

Definition 6.2. The *characteristic MBD* of a set of s-out-eqv subgraphs $\{s_i\}$ of a MBD M is the MBD obtained by making a copy of one subgraph and assigning a terminal node to each of its two descendants. It represents the subfunction that is extracted from the M if the set of subgraphs $\{s_i\}$ is resubstituted by a new node.

For instance, the MBD of figure 6.5(b) is the characteristic MBD of the subgraphs shown in figure 6.5(a).

Definition 6.3. The *resubstitution gain (r_gain)* of a set of s-out-eqv subgraphs $S = \{s_i\}$ of a MBD M is the difference between the size of M and the global size of $\mathbb{M}$, the hierarchical MBD obtained after the resubstitution. Its value is determined by the following formula:

$$r_gain(S) = |S| * (size(M_S) - 1) - size(M_S)$$

where M_S is the characteristic MBD of S. Size(M_S) is the size of M_S without terminals.

Thus, *r_gain*(S) gives the amount of nodes saved if we resubstitute each s-out-eqv subgraph s_i in S by a single node. The formula simple states that each subgraph s_i contributes with a gain of size(s_i) (the size of its characteristic MBD less the terminals) minus one (the new node introduced at its place). To the contribution of all subgraphs we must still subtract the number of nodes of M_S, the intermediate function that was created.

6.2.1 Resubtitution Algorithms

The process of finding, evaluating and resubtituting subgraphs in a MBD is called `mbd_sg_rsb`, summarized in figure 6.7. It is a greedy approach, were the algorithm iterates over the MBD until no more gain is obtained. At each loop it scans the diagram computing the *r_gain* associated to each MBD level l_i. The *r_gain* of l_i is the gain obtained if we resubstitute the *s-out-eqv* subgraphs that have their roots in l_i, if any. If the algorithm finds a set of positive gains in a single loop, then it selects and executes the resubstitution corresponding to the highest gain and continues the iteration. If no positive gain is found in a single loop, it stops and return the new MBD generated.

Figure 6.7 shows only the logic flow of the process. In fact, to compute the resubstitution gain of the *s-out-eqv* subgraphs of MBD level *i* we must first to find them. This is the more complex task. To avoid repeating this process later (when doing the resubstitution) we keep the *s-out-eqv* subgraphs previously computed in a auxiliar data structure. Thus, the function `subgr_rsb` does not recompute those subgraphs, but just replace then by a new node in the MBD.

The identification of the *s-out-eqv* subgraphs works at each MBD level at time. The nodes of the working level are considered as the roots of possible *s-out-eqv* subgraphs. Then these

subgraphs must be compared among themselves to identify the equivalent ones. Note that the depth of the subgraph is not known a priory.

```
function mbd_sg_rsb (mbd): MBD_type
var ga: array of int;     {gain array}
    imax: int;    {max gain index}
begin
  repeat
    forall_levels  i in mbd
      ga[i] = subgr_gain(mbd,i);
    imax = max_gain(ga);
    if (ga[imax] > 0)
      subgr_rsb(mbd,imax);
  until (ga[imax] <= 0);
end; {MBD_SOP}
```

Figure 6.7. MBD_SG_RSB algorithm.

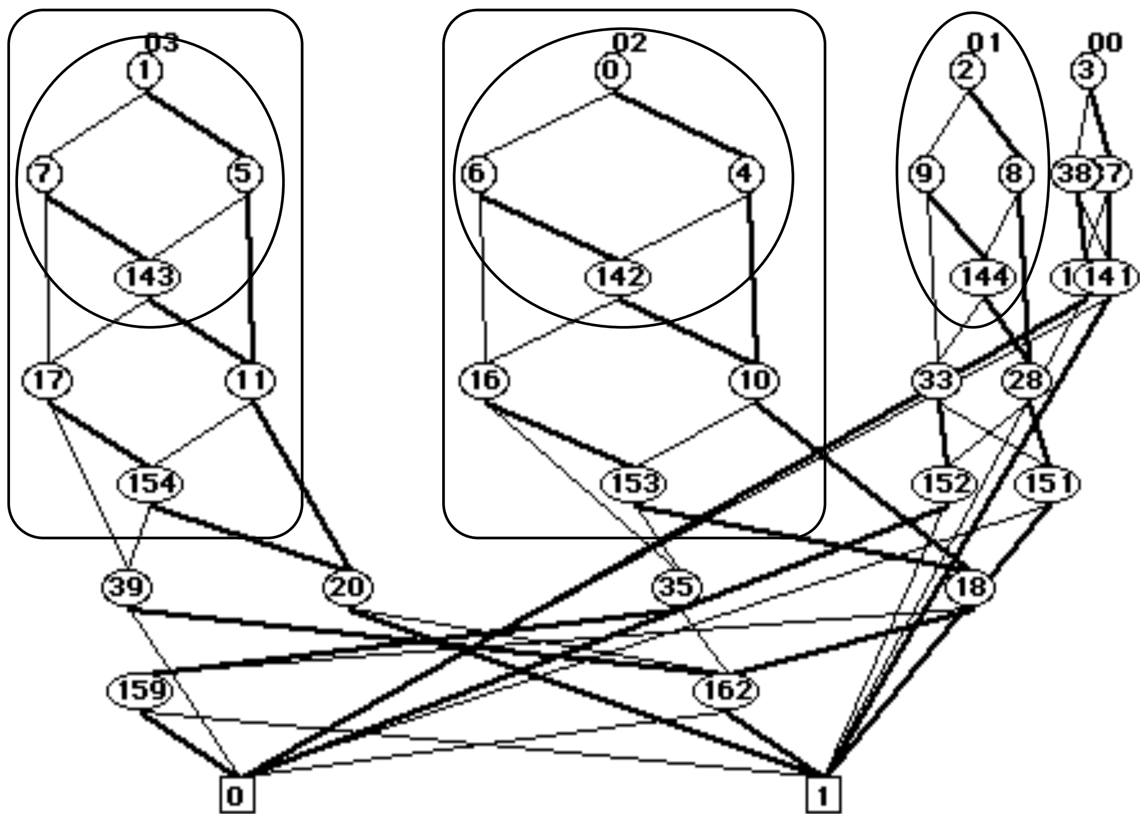


Figure 6.8. Example of equivalent subgraphs.

Comparing subgraphs two by two will result in an unacceptable cost. We should consider not only n/2*(n-1) comparisons, where n is the number of subgraph roots in the level being processed, but we must take into account that the same root can denotes more than one subgraph and may belong to different *s-out-eqv* classes. Figure 6.8 exemplifies this point. Roots **O3** and **O2** denote two subgraphs each one, one defined by the set of nodes inside the rectangular region and the other, smaller, defined by the nodes inside the circular region. The

subgraphs in the rectangular region are deeper than those in the circular regions, but smaller in number (two against three). We must evaluate all possible resubstitutions to get the better result.

We describe know the algorithm for the identification of candidate subgraphs for resubstitution. A *candidate subgraph* (or simply *candidate*, for short) has a single root and only two descendants. Following this definition, any node in a MBD is a candidate, because all of then are connected to the two terminals of the MBD. This definition of candidate is too general, and we must restrict ourselves to a simpler type of subgraph. We consider only subgraphs formed by a set of predefined patterns, where each pattern is itself a candidate. The set of basic patterns is shown in figure 6.9.

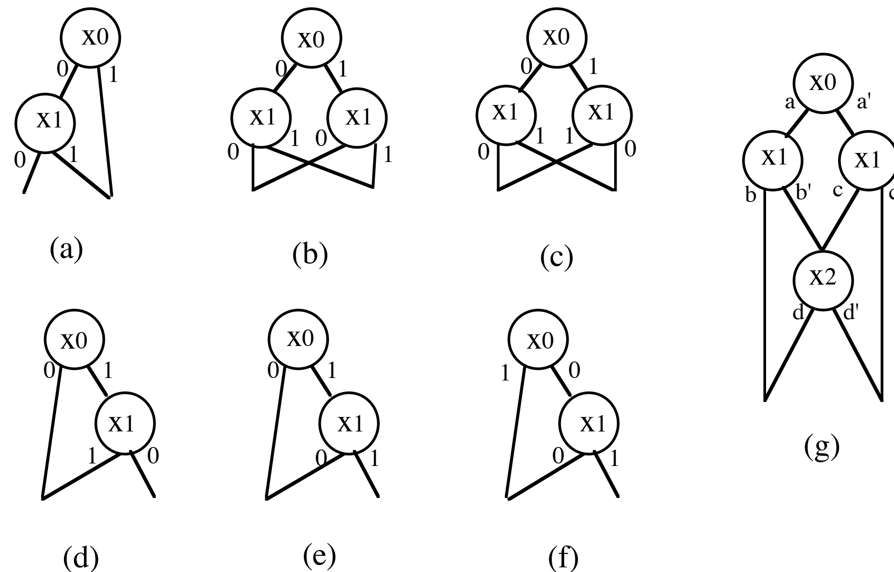


Figure 6.9. Candidate patterns for resubstitution.

Patterns from (a) through (f) show explicitly the subgraphs as they appear in a MBD. Subgraph (g) denotes a set of patterns in a compact way, in order to avoid drawing all possible combinations obtained by replacing labels {*a, b, c, d*} by 1's and 0's.

The identification of a candidate using patterns is a stepwise process. Each root node is checked to see if it is the root of one candidate pattern. The root nodes associated to the same candidate pattern are grouped together. For example, in figure 6.8 roots **O1**, **O2** and **O3** are associated to the candidate pattern MAJ (figure 6.9(g)). They form a s-out-eqv class which is a first alternative for subgraph resubstitution. In this example, there is no other s-out-eqv classes in the same MBD level, but in general there can be more than a single one. Next step is to reapply the candidate recognition algorithm to the subgraphs rooted at {**O1**, **O2, O3**} in order to determine its depth. To do that, the root nodes are changed to the set of nodes {144, 142, 143}, respectively. Then, these new roots are checked against the candidate patterns. Nodes

{144, 142} match the MAJ pattern, but 143 not. The subgraphs that start at {**O1**, **O2**} and finish at nodes {39, 20} and {35, 18} form a second alternative for resubstitution. Each of these alternatives are evaluated and the best one is selected. Their respective gains are:

r_gain({**O1**, **O2, O3**}) = 3*(4 - 1) - 4 = 5
r_gain({**O1**, **O2**}) = 2*(7 -1) - 7 = 5

Therefore, in this example both alternatives provide the same gain. Resubstituting subgraphs {**O1**, **O2**} results in the MBD shown in figure 6.10.

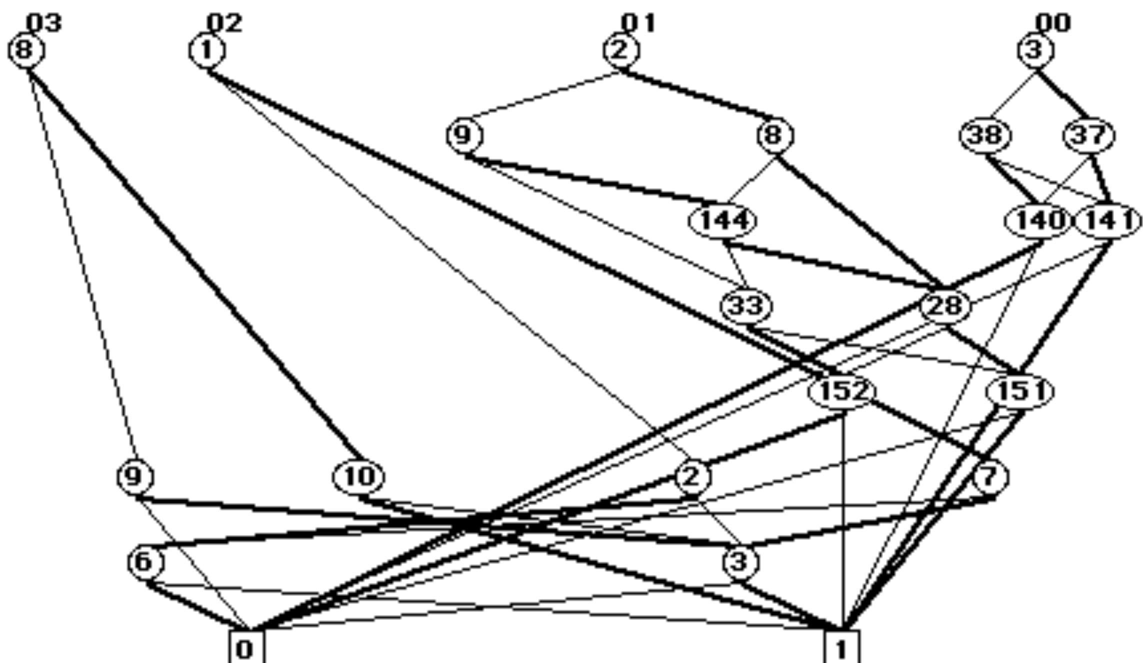


Figure 6.10. MBD after resubstitution.

A sketch of the algorithm for finding candidates is presented in figure 6.11. First the two main structures used are described. SubgrType holds the data associated to a subgraph being identified. PatternGroup store a set of s-out-eqv subgraphs, the type of the current pattern and the characteristic MBD of the subgraphs. The function receives a PatternGroup as parameter. The first step when processing one MBD level is to create a PatternGroup where each node in that level is transformed into a SubgrType. In this case, the root and the current nodes are the same, and t0 and t1 are the low and high sons. The list nl is used to keep track of the nodes present in the subgraph. Its purpose, among others, is to help computing the r_gain associated to the subgraph. In fact, the r_gain presented in definition 6.3 is valid only for a special case where the nodes of the subgraph have no fathers outside the subgraph. Otherwise we must subtract from the subgraph size all nodes with *external* fathers as well as all their descendants. Eliminating nodes used outside the subgraph would corrupt the function represented by the

MBD. The list nl helps in locating the node's fathers and in identifying nodes that must stay in the diagram. The algorithm summarizes the following steps:

1. splitting a set of subgraphs passed as parameter into a set of PatternGroups;
2. recursive application of `subgr_resub` to every new PatternGroup created;
3. the evaluation of the cost of the subgraph resubstitutions.

```
record SubgrType:
  root:     MBD_Node;   /* the root of the subgrah */
  cur:      MBD_Node;   /* current root of a new pattern */
  t0, t1:   MBD_Node;   /* the current subgraph terminals */
  nl: list of MBD_Node;/*the list of nodes inside the subgraph*/
end  record;

record PatternGroup:
  type:  PatternType    /* type of the pattern (figure 6.9)*/
  mbd:   MBD_type;      /* characteristic MBD - s-out-eqv class*/
  subgr_list: list of SubgrType; /* the s-out-eqv class */
end  record;

function subgr_resub (PatternGroup pg) : PatternGroupCost
var pattern_list:   list of PatterhGroup;
    subgr_list:     list of SubgrType;
begin
  subgr_list = make a copy of pg.subgr_list;
  foreach subgraph sb in subgr_list begin
    n = sb.cur; /* get the current node of sbgraph */
    if (n is the root of a pattern) then begin
      if pattern ∈ pattern_list /* pattern already exists */
        then insert sb in pattern.subgr_list;
        else begin
            create a new pattern p;
            put sb in p.subgr_list;
            include p in pattern_list;
        end;
      update sb; /* sb.cur, sb.t0, sb.t1 and sb.nl */
    end;
   end {foreach};
  if (pattern_list is empty) then
    return (r_gain(pg), list(pg));
  else  begin
    foreach PatternGroup p in pattern_list begin
      (cost, pgl) = subgr_resub(p);
      accum_cost += cost;
      patt_group_list = patt_group_list ∪ pgl;
    end;
    if (accum_cost > r_gain(pg))
      return (accum_cost, patt_group_list);
    else
      return (r_gain(pg), list(pg));
  end;
end;
```

Figure 6.11. Subgraph_resubstitution algorithm.

Steps 2 and 3 are meaningful only if step 1 is successful. If the current nodes of the subgraphs passed as parameter to the function are not associated to candidate patterns, then the function returns the cost of its parameter. If one or more new PatternGroups is found, then

subgr_resub is applied to each one of them. The r_gain of those subgraphs is computed, added and compared with the r_gain of the subgraphs passes as parameter. The best result is returned in the form of a list with the cost as first element and a list of PatternGroups as the second element.

After a set of PatternGroups is selected, the resubtitution takes place. See figure 6.4 to follow this process. For each PatterGroup PG a new variable is created (6.4(a) is a possible PG). This variable is associated to a new subfunction, represented by the characteristic MBD of PG (6.4(b)). For each subgraph in the subgr_list of PG a new MBD node is created. The ancestors nodes of the subgraph root are redirected to the new node. The new node's low son is assigned to the t_0 subgraph's terminal and its high son is assigned to t_1 terminal (6.4(c)).

6.3 FPGA Mapping

Once the MBD is reordered, minimized with respect to the don't care set and decomposed by subgraph resubstitution, it is ready for the mapping. The mapping is performed directly over the MBD diagram. The ACTEL device is represented itself by a MBD. As the control variables of the ACTEL_MBD nodes are accessible to the user, it is possible to generate a family of MBDs from the ACTEL_MBD by setting those control variables to adequate values. Figure 6.12 shows the ACTEL_MBD and its derived subgraphs.

The approach proposed here is to cover the MBD using the derived patterns obtained from the ACTEL_MBD. Graph covering is known to be a complex task, belonging to the class of NP-complete problems. For such intractable problems we must turn out to heuristic methods that may produce acceptable results. The covering algorithm adopted here is based on the dynamic programming approach [Aho83]. This technique was been successfully used in the technology mapping area [Keu87][Det87]. We describe herein its application for the MBD covering case.

The MBD diagram is logically decomposed into a forest of trees. We use the term *logically* because the subfunctions are not extracted and replaced by new variables. What happens is that MBD nodes with more than one father are not merged during the covering process. Instead, each of these nodes is treated as the root of a subMBD, and the dynamic covering is applied independently to each subMBD. The dynamic covering algorithm finds the optimal solution for each MBD tree, by computing the cost of all possible subtrees. This idea is illustrated in figure 6.13. First, subtrees B and C are optimally covered. Then the other subtrees can be optimally covered using the results obtained to B and C.

To see how this work on a MBD, let us take another example. Figure 6.14 shows a familiar MBD example in (a) and a set of possible matchings of the root node (0).

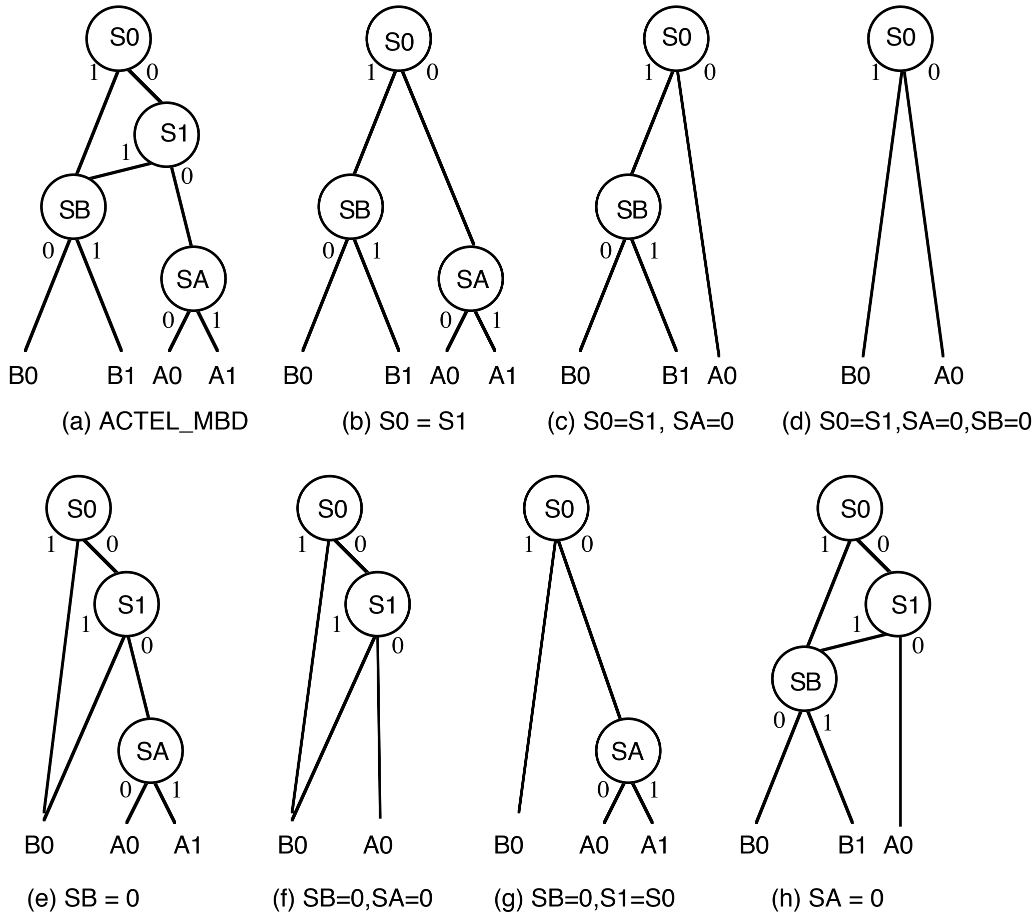


Figure 6.12. ACTEL_MBD and derived subgraphs.

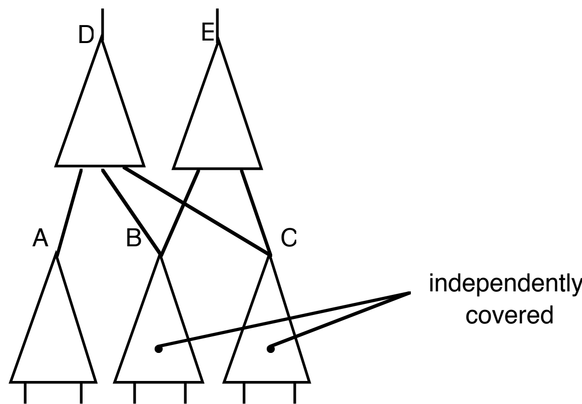


Figure 6.13. Dynamic covering (tree tiling).

Thus, from the eight subgraphs derived of the ACTEL_MBD (figure 6.12), four subgraphs can be matched against the root node 0. Note that node 3 can not be used in the matching because it has more than one father. In the dynamic covering, the four possible matches of node 0 must be evaluated. As the ACTEL device is the same for all matches, the cost of each match is 1. Therefore, the cost of the mapping in terms of area is approximated by the total number of devices used. The goal here is to find a mapping with the minimum number of devices.

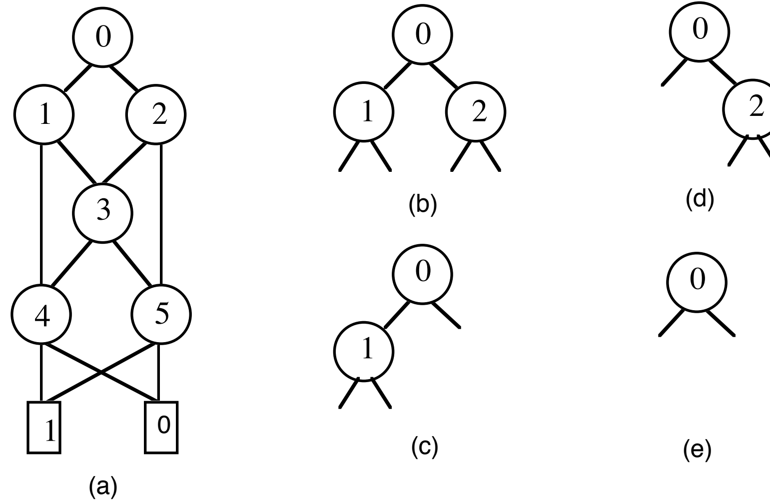


Figure 6.14. A MBD and the possible matchings of its root node.

Following the dynamic covering approach the cost of the root mapping is one (the root match) plus the sum of the costs of the input subgraphs mappings.

$$\text{cost(node_match)} = 1 + \sum_{i \in match_inputs} cost(i)$$

In our example, the cost of the match (b) is one plus the cost of the mapping of nodes {3,4,5}. For match (c) the cost is one plus the mapping of nodes {2,3,4}, and so on. The root node match will be determined only after the all subgraphs bellow it were mapped. The cost evaluation of the mapping of our example is depicted in figure 6.15.

Nodes {3,4,5} must be matched independently because they have more than one father. It remains four possible matches for nodes {0,1,2}, as shown in figure 6.14. The four corresponding costs are shown at the root node 0. The values in parenthesis are the costs as calculated by the algorithm, while the real cost is shown outside. The difference between the values is that some costs are computed twice or more, due to reconvergent paths in the MBD (nodes 5-2 and 5-3-2 are reconvergent). However, this difference propagates upward in the MBD and provide a relative estimation of the costs. The effective cost can be found easily by a single traversing of the mapped MBD.

The result of the mapping is shown in figure 6.16 (a). There is a potential problem with the dynamic covering method that is put in evidence in the ACTEL case. The constraint that a node with a fanout greater than one must be mapped separately is too restringent here. The problem is that an ACTEL device associated to a single MBD node has a high absorption capability, i.e., it can absorb up to three other nodes, and this capability is sometimes lost in the dynamic covering. This drawback is tackle by a post-processing step where the devices are selectively collapsed.

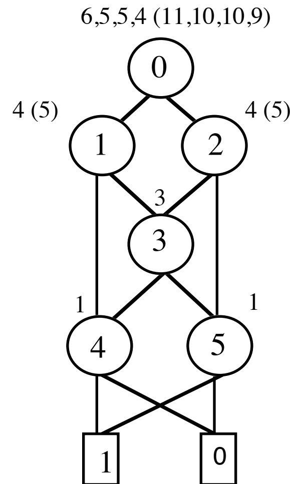


Figure 6.15. Cost evaluation in a MBD.

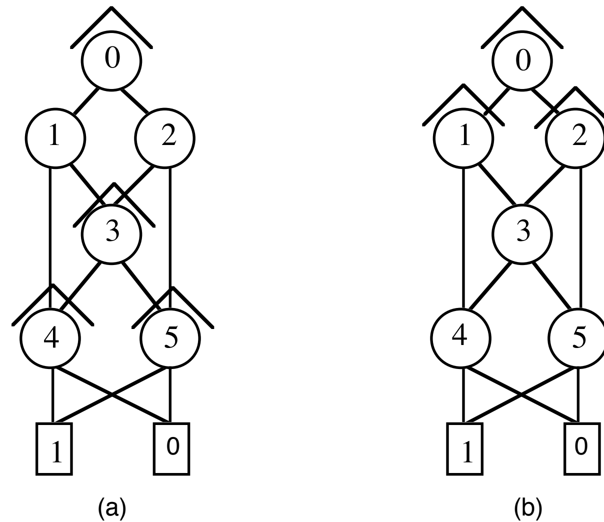


Figure 6.16. MBD mappings.

Two collapsing rules are used to reduce the ACTEL devices count:

1. if a single node is associated to a ACTEL cell then all its fathers are checked to verify is they can absorb it. In this case, the node is unmapped and included in its father's cells.
2. if all but one fathers of a single mapped node can absorb it, then it is unmapped and the father that could not absorb it is mapped into a new ACTEL cell containing the father and the node.

Rule 2 above is a generalization of a similar rule presented in [Ben92]. The result of these rules can be see in figure 6.16. In (a) node 4 (5) can be absorbed by node 3, but not by node 1 (2). Applying rule 2 node 4 (5) is unmapped and node 1 (2) becomes the root of an ACTEL device. Know nodes 1 and 2 can absorb node 3, and the final result is shown in (b), a mapping that saves one device.

6.4 Experimental Results

We use our LISP prototype to evaluate the method on a set of standard benchmarks for logic synthesis. The circuits come from the MCNC benchmark set and [Geu86]. The circuits are described in the MLL or in the PLA format (see annexes). Each file is read and coverted into a MBD. Then we apply the following operations:

- MBD reordering, with the greedy and the stochastic approaches
- subgraph resubstitution
- mapping

Table 6.1 draws the results. We compare our technique with Amap [Kar91], ASYL-PGA [Ben92], Mis-PGA [Mur92], Proserpine [Erc91] and to MisII [Bra87]. The later uses a general mapping algorithm, while the others are targeted to multiplexor based mapping. Results from Amap, Mis-PGA and MisII were taken from [Mur92]. The results from Proserpine were taken from [Erc91] and those from ASYL-PGA were obtained by running it on a SPARC station.

Circ	Logos	MisII	MisPGA	AsylPGA	Proser	Amap
5xp1	41	51	**35**	37	53	42
alu2	**34**	193	175	39	-	188
bw	61	81	**54**	61	67	83
duke2	217	176	**158**	266	177	175
f51m	38	52	39	**35**	63	56
misex1	21	22	**16**	21	25	25
misex2	52	46	**38**	46	45	47
rd53	**10**	-	-	**10**	-	-
rd73	**20**	32	25	**20**	-	32
rd84	**30**	62	36	**30**	70	62
sao2	**47**	52	49	48	-	56
vg2	243	47	**30**	43	46	44
z4	**9**	20	14	15	-	20

Table 6.1. Benchmarks for mapping with ACTEL cell.

The results from *Logos* correspond to the best ones produced by the combination of the heuristics described above. As expected, most of them comes from the application of the stochastic reordering, which produces smaller MBDs than the greedy approach.

We can see from the table that *Logos* produces in average circuits with costs equivalent to those produced by the state-of-art technology mappers. In some cases we indeed obtain the best result. Z4 is an example were the stochastic ordering finds the best ordering, which have a topology that is suitable for the graph covering approach with ACTEL cells. A counter example is VG2. This is a particular case where the multi-level representation is simple, but its MBD

representation is large (≈ 1000 nodes). The difference with respect to the other mappers is that their results refers to the multi-level circuit mapping. Both ASYL and MisPGA map both the decomposed and the flattened MBD representations and report the best result. The ordering heuristics in this case where not able to reduce the MBD such that the direct mapping produces results equivalent to the mapping of the multi-level representation, probably because such ordering does not exists (the minimum size MBD is still too large).

There is a trade off between a fast reordering (greedy) that produces in average less optimal outcomes and the more time consuming reordering (stochastic) that usually leads to better mappings. In synthesis, the quality of the result is more important than design time, if this one keeps among reasonable limits. A good solution, in this case, is to try both ordering methods and take the best solution. Although the time for stochastic reordering can be of one to two orders of magnitude greater than the greedy approach, this is feasible for most practical circuits.

In these examples, the synthesis time goes from a few minutes to some hours in the LISP prototype running on a personal computer. It is expected that an efficient implementation in a workstation could change minutes and hours in seconds and minutes, respectively.

This is valid for medium and even large circuits. On the other hand, *industrial strength examples* [1] as those from ISCAS benchmarks are sometimes so complex that we are not able even to build their MBD representation. The workstations run out of memory because the diagram required is too large. For huge MBDs we certainly will not be able to apply stochastic re-ordering techniques, but the greedy one may be surely envisaged.

6.5 Comments

In this chapter we have developed algorithms to synthesize multiplexor based circuits from the MBD description of the functions. The basic assumption at the beginning of this task was that the similarity between the logic representation primitives and the technological resources should lead to an improvement in the results obtained with respect to *standard* or *general* synthesis methods. The empirical results obtained seems to confirm this hypothesis. In fact, if we analyze the results of the benchmarks in terms of standard mapper x specialized mappers (those based on multiplexor like structures as ITEs and BDDs) we can see that the standard mapper (MisII) never produced the best result, but frequently the worst one. The same remarks are valid for the time taken for the synthesis.

The approach adopted here is a natural outcome of three main factors:

---

[1] The term *industrial strength* is used in the literature to refers to huge circuits, with very high time

- MBDs are suitable for the synthesis in ACTEL cells
- the main cost function in multiplexor-based synthesis is the MBD size
- the availability of good reordering heuristics for the reduction of the MBD size

Thus, the proposed method relies on a good initial reduction of the MBD size followed by the mapping to the ACTEL cells. An intermediate step, always pursuing the goal of reducing the MBD size, is the subgraph resubstitution that produces an hierarchical structure. In this case, some of the MBD inputs represent in fact subfunctions which are themselves described by their own MBDs.

Among the heuristics introduced in the FPGA synthesis, the reordering algorithms presented in chapter 4 are surely the main responsible of the good results obtained. The mapping itself is quite similar to the other approaches based on graph covering. The subgraph resubstitution, although conceptually interesting, is useful only for particular cases. In most examples the subgraph resubstitution either does not apply or does not produce any gain.

The minimization of the MBDs with respect to the don't care set is another factor with strong influence in the FPGA synthesis. In this case it can be used after the reordering heuristics in order to produce a further reduction of the MBD size. The reordering techniques can then be reapplied after the don't care minimization as a final step in the reduction process before the mapping. Some examples of the application of the don't care based MBD minimization were presented in the previous chapter.

With respect to the performance of the algorithms, the subgraph resubstitution is quite fast, because it deals with a simplified version of the subgraph isomorphism problem. In a single top-down step all the *s-out-eqv* classes for one MBD level are computed. Looking for the *s-out-eqv* classes of all levels requires n - 2 steps, where n is the number of MBD levels. At each step the algorithm can potentially go from the processing level up to level n - 2. If we let $k = n - 2$, this stands for a complexity in the order of $O(k(k-1)/2) = O(k^2)$. But in almost all cases there are either no *s-out-eqv* subgraphs or only swallow ones, which drastically reduces the average algorithm cost.

The FPGA mapping cost is dominated by the dynamic programming graph covering technique. Again in this case the cost is reduced due to the small size of the trees. It is a consequence of the MBD reordering that increases the sharing of subfunctions. This leads to a reduction in the processing time but also means that the optimality of the dynamic programming is somewhat lost. The pos-processing steps compensate this improving the quality of the result.

Chapter 7

Multi-level Logic Synthesis

This chapter presents an application of the MBDs in the synthesis of library based multi-level circuits. After compacted with respect to its input variable ordering, the MBD is decomposed into a Boolean network. The network is then minimized with respect to its internal and external don't care set. Then, the resulting network is mapped on the target technology.

In this chapter we discuss the application of MBDs in the synthesis of library based designs. An introductory analysis of the multi-level synthesis problems was done in section 1.3. The approach adopted here is to follow the traditional two-phases method that split the synthesis in two tasks: a technology independent logic optimization followed by a technology mapping step. The synthesis flow is presented in figure 7.1.

The starting Boolean function description is a MBD. The first step is the variable ordering that tries to reduce the MBD size using the techniques presented in chapter 4. The reduction of the MBD size leads to savings in memory and computing time. Another aspect to consider is that a reduced MBD means a diagram where the sharing of subfunctions is increased. This statement is based on the following reasoning. There are two ways a node *n* can be eliminated on a MBD. First, if its low and high sons are in fact the same node. Second, if there is another node *m* that have the same low and high sons. In this case, *n* and *m* are equivalents and *n* can be deleted. The fathers of *n* are then redirected to *m*. This implies that the subfunction represented by *m* is shared by more subMBDs. It was proven in [Lia92] that node deletion due to equivalent nodes is responsible by more than 90% of the reduction of a MBD. Thus, a MBD that is reduced with respect to its variable ordering has a greater amount of sharing among its subfunctions, and this feature can be interesting for some decomposition techniques that aims to split the MBD into a reduced number of subfunctions.

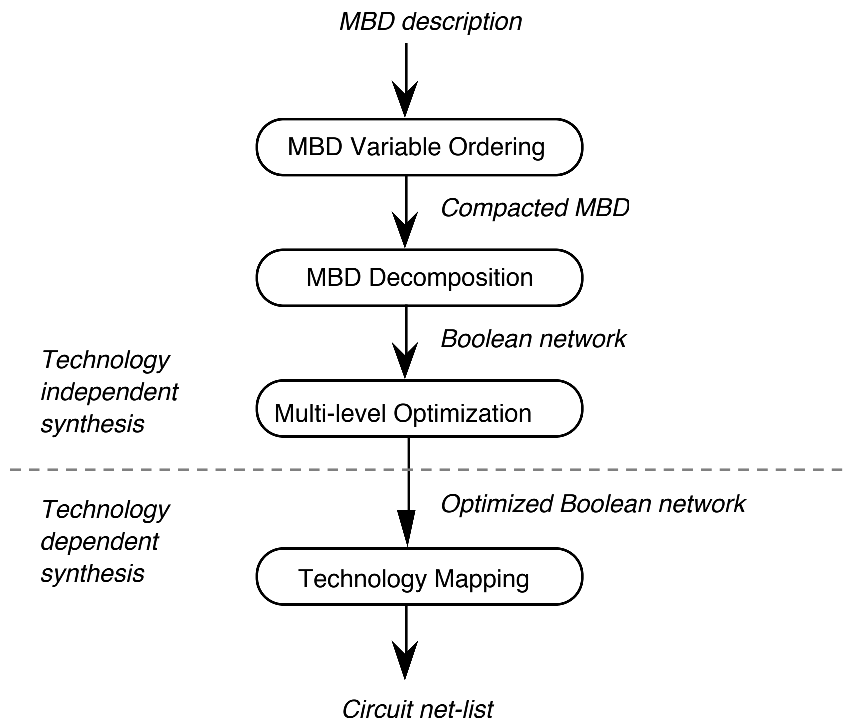


Figure 7.1. Multi-level synthesis with MBDs.

After the variable ordering, the MBD is decomposed into a set of subfunctions represented by a Boolean network. New methods were developed in order to explore some MBD features in the logic decomposition. One of those methods induces a relative ordering on the variables that are introduced in the network, in order to heuristically optimize the interconnection complexity. The technology independent phase ends with a multi-level minimization step. Subsets of the node functions' don't care sets are computed and the techniques presented in chapter 5 are used to simplify them. The technology mapping is performed using Boolean methods, and relies on a fast algorithm to detect symmetric variables in MBDs that is used in the classification of the Boolean functions. Next sections explain these topics with more detail.

7.1 Logic Decomposition

Logic decomposition of digital circuits has been studied after some decades. The basic idea relies on a divide and conquer strategy where a function f is re-expressed in terms of simpler functions, which are easier to implement. In general, a Boolean function $f(X)$ is decomposable if we can find function $g(X)$ such that:

$$f(X) = f'(g(X),X)$$

A fundamental problem associated to any decomposition method is the evaluation of the complexity of the subfunctions generated. A decomposition is meaningful only if the subfunctions obtained are simpler than the original function. This allows the recursive

decomposition of the subfunctions until no simpler function is found or until some technological constraint is attained. The estimation of the complexity of subfunctions is a difficult problem. Ideally, it should be expressed as technological costs such as area and delay of the implemented circuit. In practice, however, the physical properties are abstracted and the cost of a function is defined either in terms of Boolean properties or in terms of primitives of a given logic representation. The decomposition methods are closely related to the cost functions they use.

Definition 7.1. A logic decomposition that is based on Boolean properties is called *functional* decomposition. A logic decomposition based on the properties of a given logic representation is called *structural* decomposition.

The next two sub-sections introduce and discuss some aspects of functional and structural decomposition. The rest of the section presents some decomposition methods developed based on MBDs, which can be used in both approaches. Functional decomposition will benefit from their compactness and from the speed of the logic operations that manipulate them. Structural decomposition can also be envisaged with MBDs, by exploring some of its topological features. It provides a fast method to broke a Boolean function into a set of subfunctions.

7.1.1 Functional Decomposition

Functional decomposition is more general, because it deals with any Boolean function with no regard to the particular logic representation used. On the other hand, the advantage of structural decomposition is that their logic expressions may be related to some implementation technology and the cost functions in this case are closer to the physical design. For this reason, structural decomposition has received more attention in last decades. The advent of several types of FPGAs with a variety of complex logic cells is changing this panorama.

In functional decomposition, the Boolean property most used as cost function is the number of variables in the support of a Boolean function. The decomposition goal in this case could be to split a n variable function into a set of k variable subfunctions, with $k < n$. Other properties that may be useful as cost functions are the size of the ON-set (OFF or DC sets too), unateness, number of symmetry classes, and so forth.

A simple way to ensure that a decomposition will produce subfunctions depending on less variables is to restrict our attention to decompositions where the subfunctions have disjoint supports.

Definition 7.2. A function f is *disjoint decomposable* if we can find a partition $(\mathbf{X}_1,\mathbf{X}_2)$ of $\mathbf{X}$, with $|\mathbf{X}_1| = k$ and $|\mathbf{X}_2| = n\text{-}k$, and a function $g(\mathbf{X}_1)$ such that:

$$f(\mathbf{X}) = f'(g(\mathbf{X}_1), \mathbf{X}_2)$$

One of the first disjoint decomposition methods was proposed by Shannon [Sha49] and is usually referred as Shannon expansion. Let $(\mathbf{X}_1, \mathbf{X}_2)$ be a partition of the set of input variables $\mathbf{X}$ and $\mathbf{e}_i$ be a binary vector with length $s = |\mathbf{X}_1|$ which is the binary representation of the integer i. Then:

$$f(\mathbf{X}) = \bigvee_{i=0}^{2^s - 1} \mathbf{X}_1^{\mathbf{e}_i} \cdot f_i(\mathbf{X}_2)$$

In other words, $f_i(\mathbf{X}_2)$ is the function obtained by evaluating f with the variables $x_k \in \mathbf{X}_1$ set to the corresponding values $e_k \in \mathbf{e}_i$. If we make $\mathbf{X}_1 = \mathbf{X}$, then the expansion produces the canonical sum of minterms form. The Shannon expansion is a landmark in the switching theory. Its main contribution is to provide a systematic way to evaluate and manipulate Boolean expressions. Indeed, this recursive algorithm, with some variations like the *unate recursive paradigm* [Bra84], is up today the most used method for evaluating Boolean expressions.

Another significant contribution in functional decomposition was the work of Ashenhurst [Ash57] which was further developed by Curtis [Cur62]. They proposed several decomposition schemes both for the disjoint and for the non-disjoint cases. In the sequel we enumerate some of the more interesting cases.

- Simple disjoint decomposition:

$$f(\mathbf{X}) = f'(g(\mathbf{X}_1), \mathbf{X}_2)$$

- Multiple disjoint decomposition:

$$f(\mathbf{X}) = f'(g(\mathbf{X}_1), h(\mathbf{X}_2), \ldots, p(\mathbf{X}_k), \mathbf{X}_{k+1})$$

- Iterative disjoint decompositions:

$$f(\mathbf{X}) = f'(g(h(\mathbf{X}_1), \mathbf{X}_2), \mathbf{X}_3, \mathbf{X}_4)$$

- Complex disjoint decomposition:

$$f(\mathbf{X}) = f'(g(h(\mathbf{X}_1), \mathbf{X}_2), p(\mathbf{X}_3), \mathbf{X}_4)$$

Curtis [Cur62] provides a detailed analysis of these among other decomposition techniques in terms of decomposition charts, which are truth table like representations. These techniques are useful for technologies where the cost function may be expressed by the number of input variables of a function, which is the case of Look-up Table FPGAs from Xilinx [Xil92]. An

example of the application of these techniques can be found in [Wan92], which addresses the Xilinx target.

Another type of functional decomposition applies transformations to the input variables in order to derive simpler functions. These techniques are known as spectral methods or spectral transforms, as the Reed-Muller transform and the Rademacher-Walsh transform [Dav91a]. In [Dav86], a spectral method is developed using Binary Decision Trees. The leaves of the tree are exchanged in order to generate a unate function, which is easier to synthesize. The complexity of the transformations, however, is not easy to predict. Another interesting approach is presented in [Ola90], where the set of input vectors is reduced by successive mappings. The idea is that two input vectors $\mathbf{x}_j, \mathbf{x}_k \in \mathbf{X}$ may be mapped into a single point $\mathbf{x}_k$ of a new Boolean space $\mathbf{X}^1$, thus reducing the size of the Boolean domain. The transformations are applied up to transform the initial function into a goal function, which usually corresponds to a simple gate like a NAND, NOR or XOR. [Iba71] proposes a method that looks for the generation of negative unate functions, which exploits the fact that negative gates are less costly in most technologies, as CMOS and NMOS. The technique used is to introduce new intermediate variables that modify the Boolean space in such a way that any pair of points $(\mathbf{x}_i, \mathbf{x}_j)$, with $\mathbf{x}_i < \mathbf{x}_j, f(\mathbf{x}_i) = 0$ and $f(\mathbf{x}_j) = 1$ on the original domain are mapped to unrelated points in the new Boolean space. Hence, all positive unate variables are transformed into negative ones. The main problem with those methods is their exponential complexity on the number of variables of the function, which restrict their application to small problems.

7.1.2 Structural Decomposition

Most structural decomposition approaches rely on the algebraic representation of Boolean functions. In this case, the function is expressed as a polynomial, and the decomposition is implemented using algebraic manipulations. For example, the function:

$$f(\mathbf{X}) = x_1 \cdot x_2 \cdot x_3 \vee x_6 \cdot x_2 \cdot x_3 \vee x_7$$

can be expressed as:

$$f(\mathbf{X}) = (x_1 \vee x_6) \cdot x_2 \cdot x_3 \vee x_7$$

or:

$$f(\mathbf{X}, \mathbf{Y}) = f_1(\mathbf{X}) \cdot f_2(\mathbf{X}) \vee x_7 = y1 \cdot y2 \vee x_7$$
$$f_1(\mathbf{X}) = x_1 \vee x_6 = y1$$
$$f_2(\mathbf{X}) = x_2 \cdot x_3 = y2$$

where f_1 and f_2 are subfunctions of f and $y1$ and $y2$ are intermediate variables introduced to represent them in the system of equations obtained.

The algebraic decomposition works much as our common paper and pencil method of putting in evidence common terms that are factored out of the expression. The starting description is a sum-of-products form and the resulting representation is a system of equations or *Boolean network*. In the example above, the first step was to find the common sub-expressions $x_2 \cdot x_3$ that was factored out of the terms it appears on. Then, the factored expressions were extracted out and replaced by new functions.

It is worthy to note the difference that exists between *factorization* and *decomposition*, which are sometimes erroneously used as synonymous in the literature. The factorization is the process of rewriting an expression putting common sub-expressions in evidence in a parenthesized form. The decomposition is the identification and extraction of the sub-expressions from the original form. Hence, the decomposition introduces new intermediate variables and also new subfunctions, creating a system of Boolean equations.

One of the first methods successfully applied to large problems was the optimum NAND-gate synthesis of Dietmeyer and Su [Die69]. They start with a sum-of-products form and successively factors out common cubes which have the highest *figure of merit,* which is a cost function that estimates the decomposition gain as the number of literals eliminated by the factorization. Some restrictions of this method is that it works only for single-output functions and it fails in identifying more complex common sub-expressions. A method that overcomes this restriction was developed by Brayton [Bra82][Bra87a][Bra89]. The decomposition is based on *logic division* , which is a rather improper use of the corresponding mathematical term, since there is no additive or multiplicative inverses in Boolean algebra.

Definition 7.3. Let f and d be two Boolean functions. The *logic division* of f by d consists in finding two functions q and r such that:

$$f = q \cdot d \vee r$$

If q and d have disjoint support, then the process is called *algebraic division*, otherwise it is called *Boolean division*. To select good divisors for the decomposition, Brayton proposes the use of the kernels of an expression.

Let **D**(f) be the set of expressions $\{ f/c \mid c$ is a cube $\}$, which are called the *primary divisors* of f. The symbol '/' refers to the logic division of f by a cube c, which produces a subfunction sf of f such that $sf \leq c$. An expression is said to be *cube free* if there is no common literal that can be factored out.

Definition 7.4. The *kernels* of a function f are the elements of the set $\mathbf{K}(f) = \{ k \in \mathbf{D}(f) \mid k$ is cube-free$\}$.

The *level* of a kernel identifies the depth of its parenthesized expressions. Hence, a kernel of level 0 contains no kernel except itself and a kernels of level *n* contains at least one kernel of level *n-1* and no kernel of level *n* except itself. The cube that originates a kernel *k* is called the co-kernel of *k*. A kernel may have more than one co-kernel.

For example, in the expression:

$$f = (x_1 \vee x_2) \cdot x_3 \vee x_4 \cdot x_5$$

we find the following kernels and associated co-kernels:

kernel	co-kernel
$x_1 \vee x_2$	x_3
$(x_1 \vee x_2) \cdot x_3 \vee x_4 \cdot x_5$	**1**

Note that the whole expression is also a kernel with respect to the cube **1**. Kernel based decomposition became very popular and was employed by several synthesis systems, both in the academic [Abo90][Ber88][Bra87][Bra87a][Mat88] as well as in the industrial context (Synopsys, Compass, Fujitsu and many others).

Kernels are suitable both for algebraic and for Boolean division. Algebraic decomposition is usually faster. A common heuristic is to take any level 0 kernel as starting point, which was shown to produce minor disadvantages with respect of using higher level kernels [Bra87]. For Boolean division, however, higher level kernels are required, which increases the computing time. This lead to the development of new heuristics to overcome the speed limitations of the Boolean division [Mal89][Dev88][Dev88a]. [McG89] presents a formal analysis of the factorization problem, applying the concept of primality to factored forms in order to derive some criterion to evaluate optimal solutions. In particular, it is proved that there is a unique prime factorization for any logic expression. However, no experimental results are presented for the sake of comparison with heuristic approaches.

7.1.3 MBD Direct Decomposition

In this subsection we develop a structural decomposition method for MBDs. It is structural because it relies entirely on the underlying MBD topology to select and extract subfunctions.

It is a common accepted idea that MBDs are not associated to any structural information. They are considered simply as a graph representation of a truth table. However, is not difficult to extract structural information from the MBD. We have seen that each node in diagram is the

root of a subdiagram and, thus, stands for a subfunction of the function represented by the MBD. The *direct decomposition*, proposed here, consists in extracting these subfunctions from the MBD up to find a *convenient* Boolean network representation. By convenient we mean a Boolean network that is adequate to the target technology. For instance, in the case of a Xilinx target, a good criteria is to extract subfunctions that dependents on a limited number of inputs (4 or 5). For the usual standard cells or gate array targets, the decomposition must produce simple subfunctions, that is, subfunctions with complexity near the library gate's complexity.

The *extraction* of a MBD's subfunction is done in the following way. First, we must find a subfunction that meet the criteria established to find a convenient Boolean network. When such subfunction is found, we have located a node r in the MBD which is the root of the subfunction. Then a new intermediate variable y_i is created that will represented in the subfunction in the Boolean network that is being built. A new MBD node r_i depending on this variable is introduced in the MBD replacing the subfunction's root node r. All the fathers of r point now to r_i. Note that r_i represents the whole subfunction, and its low and high sons are connected to terminals $\{\mathbf{0}, \mathbf{1}\}$. Thus, the nodes and edges that were used only by the extracted subfunction can now be eliminated from the MBD. This is done by traversing the subfunction and deleting any node that have no father outside it.

Definition 7.5. The *direct decomposition* of a MBD consists in the selection and extraction of subfunctions associated to the MBD subgraphs in order to derive a Boolean network representation of the function. The subfunctions are associated to new intermediate variables that are reintroduced in the original MBD replacing the subgraphs they represent.

Definition 7.6. The *transitive closure* of a MBD node n is defined as the set of nodes of the MBD that can be reached only through n.

We propose three criteria to select a subgraph to extracted from the MBD:

1. Select a node that has at least *father_threshold* fathers.
2. Select a node that its subfunction depends on at least *variable_threshold* inputs.
3. Select a node whose subMBD has a size equal or greater than *node_threshold*.

Those threshold values should be specified by the designer. The first criterion is related to the amount of sharing of a subfunction in the MBD. The number of fathers of a MBD node n indicates the number of subfunctions that use or depends on the subfunction associated to n. This gives a direct measure of the fanout of the node n_n representing n's subfunction in the Boolean network. Each n's father in the MBD corresponds to a node n_f in the network that will have n_n in its fanin. Thus, the fanin/fanout constraints may be analyzed directly in the MBD graph. The second criterion is an approximation of the function complexity by the number of variables it depends on. It directly related to the complexity of the gates in the library, which

frequently is estimated by its fanin capability. The third criterion provides an alternative where the size of the subMBD is used as cost function.

The algorithm that does the job is depicted in figure 7.2.

```
function MBD_dir_dcmp (mbd): MBD_TYPE
var mbd: MBD_TYPE;
begin
  var n,m:          MBD_NODE;
      smbd:         MBD_NODE;
      v:            VARIABLE_TYPE;
      brother_ar:   array of MBD_NODE;

  init_brother_array(brother_ar, mbd);
  for i = legth(brother_ar) down to 1 begin
    n := select_subfunction(brother_ar[i], CriteriaList);
    if (n != NULL) then begin
      smbd := copy_mbd(n);
      v := create new variable;
      v.mbd = smbd;
      m = new_node(v, low(m)=0, high(m)=1);
      forall node ∈ transitive_closure(n)
        delete(node);
      replace(n,m,mbd);  /* replace n by m in the mbd */
    end;
  end;
end;
```

Figure 7.2. Direct decomposition algorithm.

The parameter *mbd* holds the MBD to be decomposed. *Brother_ar* is an array of lists of nodes. Each list contains a set of nodes of the MBD with same index. The decomposition is performed in a single bottom up step. Each MBD level is processed in turn, starting from the last level before the terminals. A node *n* is selected according to the criteria explained above. *CriteriaList* is a global parameter that store a list of node selector functions. Each node selector (a LISP or a C++ function, for instance) is applied to *n* and if one of them is satisfied, then the node is selected for extraction. For instance, if we want to apply just the first criterion, i.e., the amount of sharing, then we set *CriteriaList = (sharing_degree(n))*. *Sharing_degree(n)* is a function that checks if the amount of fathers of *n* exceeds the *fathers_threshold*. When *n* satisfies one decomposition criterion, then a copy of the subMBD that starts at *n* is made and associated to a new intermediate variable *v*. The nodes in the transitive closure of *n* are eliminated, thus reducing the MBD. Note that we can not eliminate a node contained in the subMBD rooted at *n* if it can be reached from a node outside this subMBD. This is illustrated in figure 7.3(b)-(c), where nodes associated to variable x_4 must stay in the MBD after the extraction of the subMBD rooted at the node associated to variable x_3. Then *n* is replaced by a new node depending on the new intermediate variable created. This node is connected directly to the terminals **0** and **1**. External don't cares are not considered in this step. They are left to be used in the next phase of the multi-level synthesis, the multi-level minimization.

The algorithm is illustrated at figure 7.3. In this example *father_threshold* is 2, *variable_threshold* is 2 and *node_threshold* is 3. In 7.3(a) the initial MBD is shown together with its Boolean network representation. In this case, the network is represented by a single node that is associated to the global function. By applying the criterion 1, we select node controlled by x_3 as the root of a subMBD to be extracted, because it has two fathers.(figure 7.3(b)). A new intermediate variable y_1 is associated to the extracted subfunction. The new extracted subfunction is represented by a new node in the network that depends on the subset of inputs { x_3 , x_4 }. With the extraction of subfunction y_1, the main function f becomes independent of variable x_3 , which does not appear in its MBD representation (7.3(c)). In the resulting MBD the criterion 1 no more applies. Thus, criterion 2 is applied and nodes associated to variable x_2 are selected. Both depend on three variables. Their subMBDs are extracted and the new subfunctions are associated to intermediate variables y_2 and y_3. Both subfunctions depends on the same subset of variables, { x_2 , x_4, y_1 }. Note that y_1 appears on both subfunctions. This is shown in the network by the two connections that go out of node y_1.

Finally, in 7.3(d) the final multi-level circuit is presented. No criterion applies to the remaining MBD that is associated to the output node of the network. The final network contains four nodes, with three intermediate variables. The nodes are ordered from left to right according to the their logic level. Node y_1 belongs to the first logic level; y_2 and y_3 belong to the second logic level, because they have y_1 in their fanin and f belongs to the third logic level, because it comes after y_2 and y_3.

One interesting feature of the direct decomposition is that it induces a partial ordering among the inputs. Variables with smaller indices tend to be nearer to the output's logic level. The root variable, for instance, usually traverses only one logic level to arrive to the output because it is the last one to be extracted. In figure 7.3, variables { x_3 , x_4 } traverse three logic levels, variable x_2 traverses two logic levels and variable x_1, the root, traverses only one logic level. The order relation, however, depends on the particular MBD topology and can not be uniquely determined by the variable index. The ordering in the way variables enter the logic stages is in fact a useful property. First, this lead to a heuristic reduction on the complexity of the interconnections [Abo90]. Second, this information can be used to optimize the critical path delay. The delay of a signal s is the time it takes to go from one input up to the circuit outputs. Variables near the root traverse less logic stages, and thus introduce smaller delays to logic signals. If the inputs of a circuit are associated to different delays, a delay oriented decomposition can reorder the MBD in such a way that inputs submitted to more severe delay constraints should be assigned smaller indices in the variable ordering.

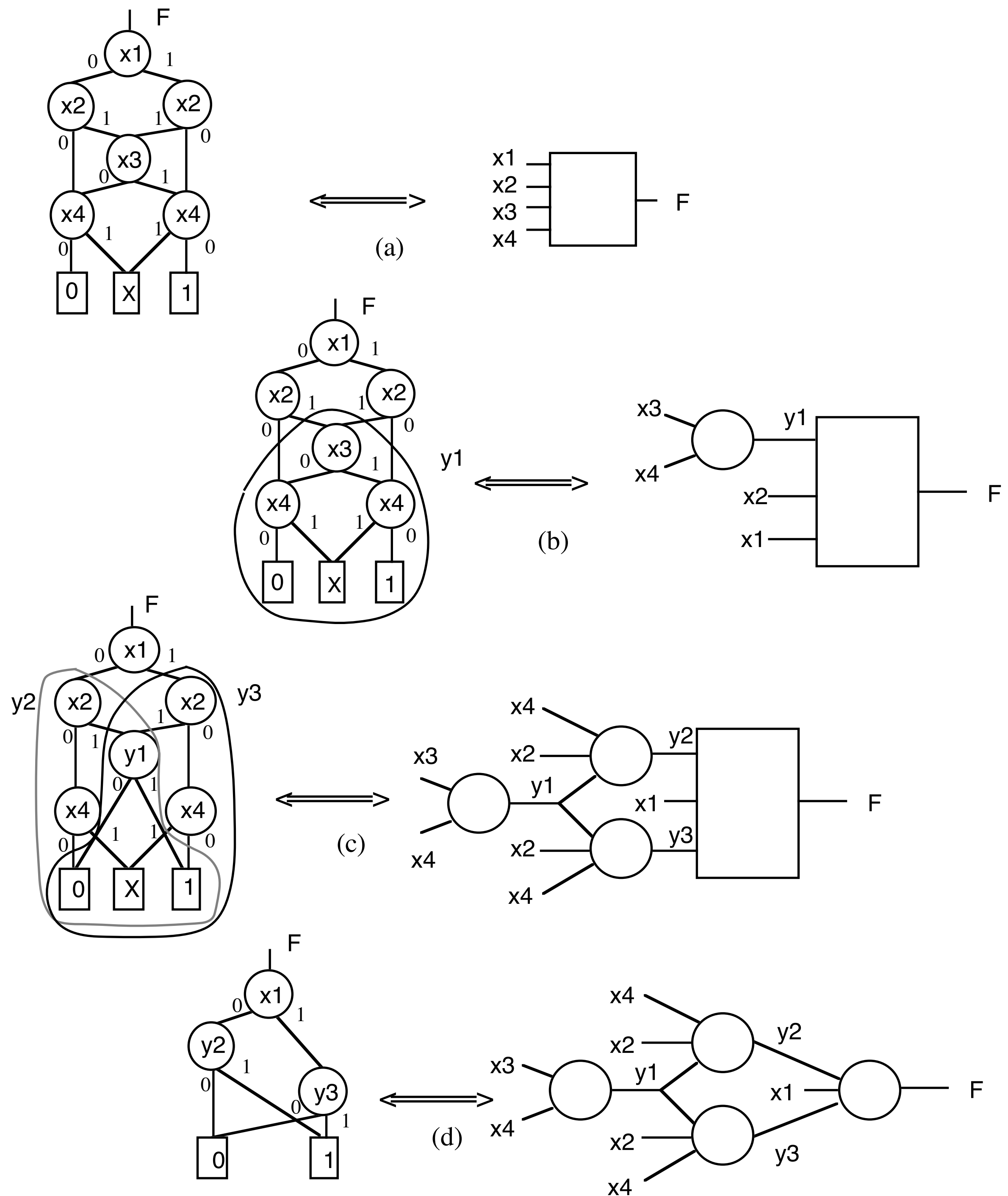


Figure 7.3. Example of direct decomposition of a MBD.

7.1.4 Experimental Results - Direct Decomposition

We have evaluated the algorithm on a set of benchmark circuits for multi-level synthesis. A first experiment addresses the effect of the reordering step on the decomposition. Since it is clear that for this method a larger MBD should lead to a larger Boolean network after decomposition, we present only one example for illustration purposes.

The circuit description was read and converted into a MBD. The initial ordering is the sequence the variables appear in the input file. The initial MBD and the MBD reduced by the re-ordering techniques shown in chapter 4 are decomposed for several choices of the decomposition parameters. Table 7.1 shows the results in terms of size of the Boolean network, total number of vertices and number of connections. The circuit is the Z4.PLA, from the Berkeley PLAs benchmark set.

Z4.PLA	MBD size: 28 nodes						MBD size: 66 nodes						MisII
var/sh	2/2	3/2	4/2	2/3	3/3	4/3	2/2	3/2	4/2	2/3	3/3	4/3	
Net	24	12	8	24	12	9	62	60	56	62	54	50	12
MBD	128	73	49	128	73	65	354	350	335	354	326	320	64
Arcs	60	36	26	60	36	32	160	158	150	160	154	151	30

Table 7.1. Direct decomposition of Z4.PLA.

The field *Net* indicates the number of subfunctions generated (nodes in the Boolean network). *MBD* is the sum of the size of the MBDs of all subfunctions. *Arcs* is the number of connections in the Boolean network. Thus, *Net* is an estimation of the final gate count of the circuit, *MBD* is an estimation of the complexity of the subfunctions and *Arcs* is an estimation of the interconnection complexity. The field *var/sh* indicates the *variable_threshold* and *father_threshold* used, respectively. We have seen that *node_threshold* had little influence in this example. One reason is that the reduction of the MBD size increases the sharing and therefore it becomes a dominant parameter when low *father_threshold* values are used. The same reasoning is valid for low values of *variable_threshold*. Increasing the value of these parameters leads to a greater influence of the *node_threshold*, but this may be not interesting for library based designs. A decomposition driven by the size of the MBD of the subfunctions is better suited for technologies where this parameter is important, like selector-based FPGAs, for example.

The results confirm the interest in the MBD reordering as a prior step for direct decomposition. The ratio between the MBD sizes and the ratio between the decomposition results are quite similar. This does not mean that they are directly proportional. However, a larger MBD usually leads to a larger Boolean network derived by direct decomposition. We can see also that the *variable_threshold* has a major influence in the size of the Boolean network obtained. The larger the number of variables allowed by subfunction, the smaller the number of nodes of the Boolean network.

These results must be analyzed with care. A larger Boolean network does not always denotes a worse solution. Indeed, some synthesis systems such as MisII [Bra87], BOLD [Bos87] and

Olympus [Dem90], among others, decompose the multi-level optimized network into two input functions before mapping, which temporarily increases its size. Although some global information of the optimized multi-level network may be lost, the new network is composed by simpler subfunctions and its quality is roughly the same of the previous one. The point here is that the quality of the solution is more related to the optimality of the multi-level circuit, which is expressed both in terms of the number of subfunctions and also by the complexity of each of them. For this reason we have included also the accumulated size of the subfunctions' MBDs, that is an estimation of their complexity, and the number of interconnections, which is an estimation of the routing complexity.

In our experiments, the difference between the networks generated from the MBD with the original ordering and the reordered one, using the same decomposition parameters (i.e., same *father_threshold* and *variable_threshold*) is significant, and indicates that the original MBD M_{Z4} uses more information than it is needed to describe the Boolean function. We can not properly say that M_{Z4} contains redundant information, because no node in M_{Z4} may be eliminated without changing its functionality. But we can say that M_{Z4} is not *optimized* with respect to the ordering. In this example, we know that the reordered MBD $M_{Z4\text{-}O}$ is the optimum solution (see table 4.2, chapter 4), but in general there is no way to compute the best ordering without using some brute force method. The heuristics proposed in chapter 4, however, produce very good average results.

To get a point of reference for comparison, we present the same data for an algebraic decomposition of Z4.PLA obtained with MisII. We can see that the direct decomposition produced similar results when *variable_threshold* and *father_threshold* are set to (4,3) and (4,2), respectively. The best result is obtained for parameters (4,2). In this case, a subfunction is extracted either if it has four variables in its support or if it has more than two variables (single variable subfunctions are not extracted) and a fanout greater than two. Four variables is still a reasonable support size for library based designs, and this example compares favorably with respect to MisII, which is a standard reference in terms of logic synthesis.

Table 7.2 provides a more extended comparison with MisII. The best results are indicated for each evaluation factor. We can see most of times the direct decomposition with *variable_threshold* of four and a *father_threshold* of two provides the best results. This indicates that for these evaluation parameters the direct decomposition provides interesting results. Another important point is that it is surely a faster decomposition method than the kernel factorization used by MisII. The algebraic manipulation performed by MisII may be at best *O(n)* with respect to the number of variables [McG89]. This, however, is not true for all the cases. The method presented here is clearly linear with respect to the number inputs, because each MBD level is scanned only once. For each level, all subMBDs with roots at that level are traversed once. A simplification that must be taken into account is that the MBD is

stepwise decomposed, due to the deletion of the nodes by the extraction process. We estimate that the manipulation of subMBDs can be much faster than the kernel extraction and algebraic decomposition performed by MisII.

	Direct Decomposition									MisII		
	var=2 shar=2			var=3 shar=2			var=4 shar=2					
	Nodes	MBD	Arcs	Nodes	MBD	Arcs	Nodes	MBD	Arcs	Nodes	MBD	Arcs
5xp1	65	276	162	52	237	143	**38**	**187**	**119**	42	244	120
alu3	55	231	135	35	168	106	**28**	**142**	**95**	**28**	192	145
bw	102	420	279	90	382	266	**63**	**276**	227	65	400	**216**
dekoder	17	61	39	15	55	37	12	**42**	34	**10**	55	**30**
dk27	22	74	49	15	51	40	13	**43**	35	**11**	55	**30**
f51m	62	275	163	57	260	153	**37**	**197**	**117**	42	243	119
fo2	15	60	36	11	48	31	**8**	**35**	**24**	12	56	32
misex1	34	148	92	32	139	90	**18**	**91**	68	**18**	118	**67**
misex2	86	322	184	54	223	153	42	**181**	136	**30**	190	**113**
r53	21	95	55	19	89	53	**14**	**69**	**41**	**14**	85	**41**
r73	41	193	113	39	187	111	31	**159**	96	**28**	193	**93**
r84	57	265	155	54	256	152	44	**223**	**137**	**37**	337	140

Table 7.2. Decomposition benchmarks.

7.1.5 Path Oriented Decomposition

This decomposition method is based on the theorem 3.1, which states that there is a correspondence between paths in a MBD and cubes of the function it represents. Indeed, each path in a MBD corresponds to a cube of a disjoint cover of the function. Following this analogy, we can introduce the idea of *covering paths* of a MBD as a way of covering the function it represents.

Definition 7.7. A path that connects the MBD root to the **1** terminal is called a *ON-path*. In a similar way we define a *DC-path* for the **X** terminal and a *OFF-path* for the **0** terminal.

Definition 7.8. A MBD M is *covered* by a set of MBDs SM = $\{M_i\}$ if each path of M appears at least in one element SM.

Therefore, covering all paths of a MBD is equivalent to cover the function it represents. One immediate way of doing that is to generate a sum of disjoint MBD cubes (section 5.2), where each cube corresponds to a MBD path. However, such cubes are not prime, and the resulting

sum of products will be not useful for a direct implementation. What we propose here is to cover the MBD with a set of subfunctions that are directly extracted from the diagram.

Definition 7.9. A *path oriented decomposition* of a MBD M is the re-expression of M in terms of a set of subfunctions $\{SF_i\}$, each one represented by a MBD M_i, such that M is covered by $\{M_i\}$.

The idea is illustrated here by an example. Consider our familiar MBD shown in figure 7.4. We identify in the MBD the paths that connect the root to the **1** terminal. They are labeled using the letters *a, b, c*, etc. To each MBD arc the labels of the paths that it belongs to are indicated in the figure. The disjoint cubes associated to the MBD paths are indicated in a Karnaugh diagram. It is interesting to remark the *slicing effect* of the variable ordering on the cover. The root variable, for instance, must be present in any path and, therefore, all cubes of the cover depends on it. This can be visualized in the Karnaugh diagram by the fact that no cube is expanded over x_1, the root variable. As the first MBD level (index 1) is *complete*, the cubes of the cover are also sliced with respect to x_2, the variable associated to index 1. We say that a MBD level l_i is *complete* if all its paths contains one variable with index i. In this case, the path cover of the function is *sliced* with respect to that variable. In figure 7.4, the cubes are sliced with respect to x_1, x_2 and x_4, which can be easily checked in the Karnaugh diagram.

Figure 7.4. Path cover of a MBD.

To decompose the MBD into a set of subfunctions we must find a way to select subsets of the MBD paths and to cover them with extracted subfunctions. Our goal is to exploit the graphical nature of the MBD in order to obtain a fast algorithm. The process consists basically in the following steps.

1. Select a node *n* of the MBD M according some cost function. Let S be the set of paths that pass by *n*, which is called the *splitting node*.

2. Builds a new MBD QD with the set of paths S. All arcs of QD that are not defined are pointed to the **0** terminal.

3. Let S^1 be the set of remaining paths of M. Builds a new MBD R with S^1, and point all undefined arcs to the **0** terminal.

4. The sum of the MBDs QD and R is a cover of M:

$$M = QD + R$$

In the example of figure 7.4, if we select the node labeled x_3 we have S = { *b, c, d, e* } and S^1= {*a, f*}. The resulting MBDs QD and r are shown in figure 7.5 (a) and (b), respectively.

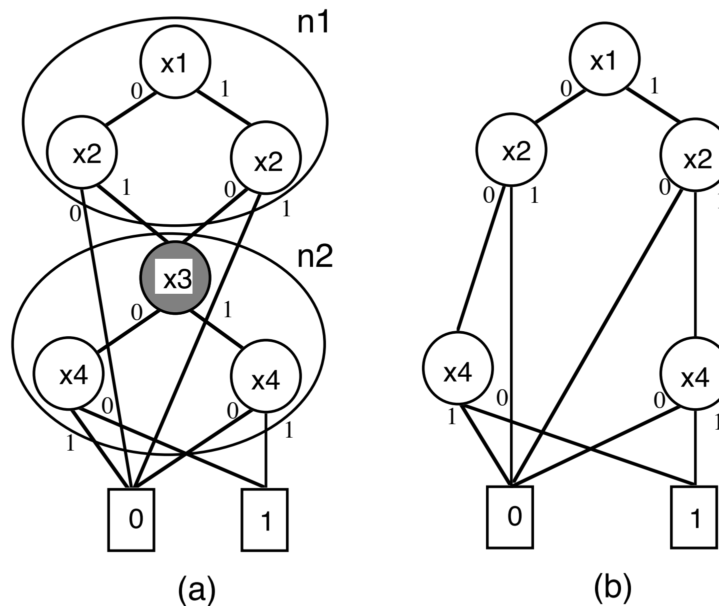


Figure 7.5. Path oriented decomposed MBDs (a) QD and (b) R.

The path oriented decomposition has some interesting properties.

Proposition 7.1. The path oriented decomposition of a MBD relies on an algebraic division of the function it represents.

Proof. We must just to show that the path oriented decomposition produces three subfunctions *q*, *d* and *r*, with $f = q{\cdot}d \vee r$ and sup(q) ∩ sup(d) = ∅. The procedure described above re-express the original MBD in terms of the sum of two MBDs QD and R. MBD QD is defined by all the paths that cross a selected node *n*, which divides these in two set of subpaths: PU, the set of paths that go from the root up to *n* (*upper paths*) and PL, the set of paths that go

from n to the terminal (*lower paths*). All the paths that do not cross node n are redirected to the **0** terminal in QD. The set of paths P covered by QD is obtained by concatenating each upper path with every lower path. If we assume that # is the concatenation operator, then we have P = PU#PL. Thus, each path p in QD is decomposed into a upper path $pu \in$ PU and a lower path $pl \in$ PL. Since each p corresponds to a cube c of a disjoint cover C of the function denoted by QD, the concatenation of pu and pl defines an AND decomposition of c into two cubes cu and cl: $c = cu \cdot cl$. Cubes cu and cl are obtained from subpaths pu and pl in the same way that c is generated by p. Thus, the concatenation of paths corresponds to the product of cubes, and the cover C of QD's function can be expressed as C = CU·CL, where CU is the set of cubes corresponding to the upper paths and CL is the set of cubes associated to the lower paths. From the ordered nature of the MBD is evident that CU and CL depends on different variables. As a set of parallel paths in a MBD describes a sum of cubes, CU and CL are, in fact, two sum of products that are disjoint between themselves, i.e., $\sup(CU) \cap \sup(CL) = \varnothing$. Thus, the path oriented decomposition of a MBD M corresponds to an algebraic division of a function f denoted by M, where subfunctions q = CU, d = CL and r is denoted by the remaining MBD R. ◊

In figure 7.5 the subfunctions q and d are indicated by subgraphs n_1 and n_2. The set of paths contained in the original MBD of figure 7.5 is given below.

a = $x_1 \cdot x_2 \cdot x_4$
b = $x_1 \cdot \bar{x}_2 \cdot x_3 \cdot x_4$
c = $x_1 \cdot \bar{x}_2 \cdot \bar{x}_3 \cdot \bar{x}_4$
d = $\bar{x}_1 \cdot x_2 \cdot x_3 \cdot x_4$
e = $\bar{x}_1 \cdot x_2 \cdot \bar{x}_3 \cdot \bar{x}_4$
f = $\bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_4$

The node n chosen covers paths { b, c, d, e }. The subpaths from the root up to n, PU, are associated to the sum of products CU = $x_1 \cdot \bar{x}_2 \vee \bar{x}_1 \cdot x_2$, and the subpaths from n to the **1** terminal, PL, define the function CL = $x_3 \cdot x_4 \vee \bar{x}_3 \cdot \bar{x}_4$. The MBD R is formed by the remaining paths { a, f }. Thus, the decomposition produces:

$$f = q \cdot d \vee r$$
$$q = x_1 \cdot \bar{x}_2 \vee \bar{x}_1 \cdot x_2$$
$$d = x_3 \cdot x_4 \vee \bar{x}_3 \cdot \bar{x}_4$$
$$r = x_1 \cdot x_2 \cdot x_4 \vee \bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_4$$

In general, the choice of the splitting node is not trivial. One possible criterion is the choice of the node which covers the largest number of paths. But in this case, the root node covers all paths, which is certainly not a good choice. The next candidates are frequently the root son's.

Therefore, the application of this criterion tends to produce a Shannon decomposition, that will reproduce the MBD structure in the Boolean network.

A simple criterion is to select a node that produces functions q and d with similar support lengths. Splitting the support into similar length subsets will produce a balanced network for q and d subfunctions if we use the support length as a estimation of the function complexity. However, r will be not balanced with respect to q and d. In fact, if we select only one splitting node at a time the decomposed network can be highly unbalanced, as illustrates figure 7.6.

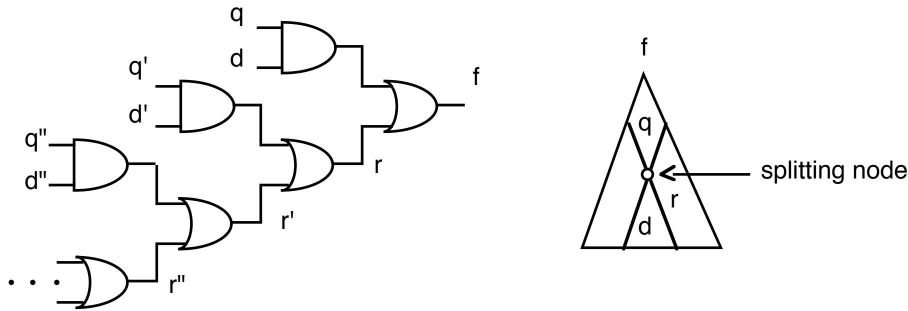


Figure 7.6. Unbalanced path decomposition.

An alternative approach is to select a set of splitting nodes at a time, referred here as the *splitting line*. This will reduce the complexity of r, increasing the fanin of the OR gate, as shown in figure 7.7. As r is simpler, the depth of the unbalanced part is smaller. In some cases the splitting line covers all paths of the MBD and r disappears, leading to a completely balanced decomposition.

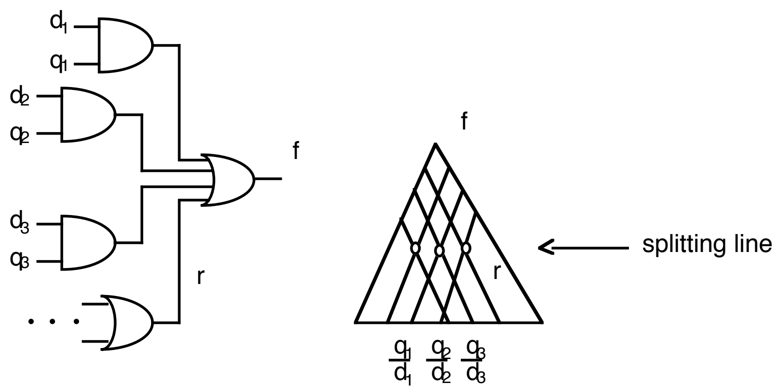


Figure 7.7. Balanced path decomposition.

The balanced path decomposition re-express f in terms of a sum-of-products:

$$f = q_1 \cdot d_1 \vee \cdots \vee q_i \cdot d_i \vee \cdots \vee q_k \cdot d_k \vee r$$

where k is the number of splitting nodes. Thus, if the splitting line has k nodes, f is decomposed into a new function of $2k$ (balanced) or $2k + 1$ (unbalanced) variables.

This method may be used both for the factorization or for the decomposition of a function. In the case of the factorization, we simply compute the new subfunctions q, d and r and recursively apply the algorithm on them. The algorithm for path oriented factorization is presented in figure 7.8.

```
function MBD_path_fact (mbd) : FactoredForm
var mbd: MBD_TYPE;
begin
  var q,d,r:    MBD_TYPE;
      v:        MBD_NODE;

  /* check for stop condition */
  if (support(mbd) <= *min_support*)
    return(mbd);
  else if (simple_fform(mbd))
    return(snode_fact(mbd));
  /* compute the splitting node and the subfunctions q,d,r */
  v = select_split_node(mbd);
  d = copy_mbd(v);
  q = get_up_paths(mbd,v);
  r = get_rest_function(mbd,v);
  return(MBD_path_fact(d)·MBD_path_fact(q) v MBD_path_fact(r));
end;
```

Figure 7.8. MBD path factorization algorithm.

Two stop conditions are checked. One is the number of variables in the support of the MBD. The other one is the identification of simple factored forms, which is made using the concept of *supernodes* [Cal92]. A supernode is a simple AND/OR factored form in which only one node may appear at each MBD level. When this special case is identified, the factorization is straightforward. The selection of the splitting node is done by taking, among the nodes of the MBD level corresponding to the variable that divides the support of the MBD into two similar length subsets, the node that covers the larger number of paths and also that has the larger number of fathers. An special case that is considered is the occurrence of *dominators* nodes [Kar89]. A node n_d dominates another node n_s if all the paths from the root up to n_s pass through n_d . If the MBD as a dominator for the **0** terminal, then it has a simple disjoint OR decomposition. It the MBD as a dominator for the **1** terminal, then it has a simple disjoint AND decomposition. Thus, if we find a dominator for a terminal node, it is selected as splitting node.

Subfunction d is just a copy of the subMBD that has the splitting node as root. Subfunction q is composed by the sum of all the paths from root of the MBD up to the splitting node n. In each of such paths, n is replaced by the terminal **1**. Each path corresponds to a cube that is made prime according to the method described in section 5.1.4. Subfunction q is obtained by ORing all these cubes. The rest r is obtained by replacing n by terminal **0** in f and by reducing it. In the section 7.1.6 some examples of path oriented factorization are provided.

The path oriented decomposition is more complicated. In the factorization, the subfunctions are treated independently and there is no sharing among them. In the case of the decomposition, since the subfunctions are extracted and are associated to new nodes in the network, identical subfunctions must be merged, to avoid useless duplication of logic. Moreover, the factorization is a process applied to single output functions, while the decomposition may be applied to multiple output functions. In this case, each MBD root is associated to an output variable, which is *not* the controlling variable denoted by the root node index.

All the decomposition process is performed within a single MBD. Any time a new subfunction *sf* (q, d or r) is extracted, it is compared against the functions denoted by the roots of the MBD. If there is no equivalent function then a new intermediate variable is created to represent *sf*, the MBD of *sf* is merged into the global MBD and the root of *sf* becomes a new root of the MBD. If there is some MBD root that is equivalent to *sf* or to its complement, then the output variable associated to that root is used to represent *sf*. The extraction of d type subMBDs is completed by replacing the d root by a new node that depends on the variable created or chosen to represent the subMBD, which is similar to process executed in direct decomposition. The algorithm for path decomposition is presented in figure 7.9.

The input to the decomposition algorithm is a list of variables that is initially the list of output variables of the function. Each variable in *varlist* stands for a single output function. All functions are represented by a single MBD M. Thus, every variable in *varlist* is associated to a root node of M. The algorithm treats one output at a time. Since *varlist* is updated along the algorithm, we make a copy of its initial contents to avoid modifying it at the same time it is scanned by the top level loop (***foreach*** *v* ***in*** *vl*). For each variable, the splitting line of its respective MBD is computed. If the MBD has a **1** (**0**) terminal dominator a special function is called that decomposes it into a two inputs AND (OR) subfunction and call recursively *MBD_path_dcmp* for both subfunctions. The special case is in fact a simplified version of the decomposition algorithm. If the function has no dominator, then the subfunctions q and d are computed for each node in the splitting line. They are stored together with the splitting node as a triple (v, d, q) in *pairlist* (list of pairs of functions). After all splitting nodes are processed then the rest r is computed. The procedure *treat_subfunctions* takes *pairlist* and *varlist* and performs the extraction and re-insertion of the new subfunctions in the global MBD. If a subfunction (d or q) has an equivalent one in the MBD either in direct or in complemented form then the equivalent one is used, otherwise a new variable is created and the subfunction is merged in the MBD. *treat_subfunctions* returns a list with the literals that represent the subfunctions in the form ((y_{d1}, y_{q1}) ... (y_{dk},y_{qk})). If the r functions exists, then a new variable y_r representing it is created and included in *pairlist*. Finally, the new decomposed function for the current variable *var* is created based on the elements in *pairlist*. For each pair (y_{di}, y_{qi}) an

AND MBD M representing the function M = $y_{di} \cdot y_{qi}$ is created. For the y_r variable a single variable MBD is built. Then all MBDs are ORed and the resulting MBD is associated to the current variable var.

```
function MBD_path_dcmp (varlist): VARIABLE_LIST;
var varlist: VARIABLE_LIST;
begin
  var f,q,d,r,v:    MBD_TYPE;
      rec:          Boolean;
      pairlist:     List;

  rec = false;
  vl = copy_list(varlist);
  foreach var in vl begin
    f = get_mbd(var);
    split_line = get_split_line(f, dom);
    if dom then return(simple_dcmp(f,dom,varlist));
    else begin
      rec = true;
      foreach v in split_line begin
        d = copy_mbd(v);
        q = get_up_paths(f,v);
        pairlist = add (v,d,q) to pairlist;
      end;
      r = get_rest_function(f,split_line);
      pairlist = treat_subfunctions(varlist, pairlist);
      if r then add (new_var(r),1) to pairlist;
      set_mbd(var, build_anor(pairlist));
    end;
    if rec then MBD_path_decmp(varlist);
    else return (varlist);
  end;
```

Figure 7.9. Path oriented decomposition algorithm.

7.1.6 Experimental Results - Path Oriented Decomposition

We have applied the path decomposition method to the same benchmark set used to evaluate the direct decomposition. We reproduce the results of direct decomposition in table 7.3 in order to compare them with the path oriented one.

The results of path decomposition are quite erratic. As we can see from table 7.3, in a few cases it wins (r84, fo2 and dekoder), while at the other extreme it gives very bad results (bw, 5xp1 and f51m). It is an indication that the method is very sensitive to the MBD topology. Of course, the size of the MBD plays an important role, but its specific topology is an important factor too. This suggest that some improvements may be obtained by detecting which topological factors most influences the results and by looking for variable orderings that maximize them. Another alternative that could be tried is the selection of overlapping splitting nodes, which eventually may belong to the same path. This will provide more freedom in the search of a minimal path covering.

	Direct Decomposition									Path Decomposition		
	var=2 shar=2			var=3 shar=2			var=4 shar=2					
	Nodes	MBD	Arcs	Nodes	MBD	Arcs	Nodes	MBD	Arcs	Nodes	MBD	Arcs
5xp1	65	276	162	52	237	143	**38**	**187**	**119**	87	427	236
alu3	55	231	135	35	168	106	**28**	**142**	**95**	49	244	138
bw	102	420	279	90	382	266	**63**	**276**	**227**	228	1040	568
dekoder	17	61	39	15	55	37	12	**42**	34	**7**	47	**25**
dk27	22	74	49	15	51	40	**13**	**43**	**35**	16	73	41
f51m	62	275	163	57	260	153	**37**	**197**	**117**	116	518	278
fo2	15	60	36	11	48	31	8	35	24	**4**	**30**	**16**
misex1	34	148	92	32	139	90	**18**	**91**	**68**	48	229	130
misex2	86	322	184	54	223	153	**42**	**181**	**136**	44	276	185
r53	21	95	55	19	89	53	**14**	**69**	**41**	19	101	53
r73	41	193	113	39	187	111	**31**	**159**	**96**	46	286	153
r84	57	265	155	54	256	152	44	223	137	**26**	**219**	**102**
z4	24	128	60	12	73	36	**8**	**49**	**26**	12	80	40

Table 7.3. Comparison between Direct x Path oriented decomposition.

This approach has some interesting aspects. The extraction of d type functions is executed simultaneously for all outputs. Also, by controlling the number of cut nodes we may control also the depth of the circuit. Moreover, the decomposition tends to produce several functions with similar supports, which may be useful for the multi-level minimization.

7.1.7 Boolean Division

Boolean decomposition techniques are potentially more powerful than structural ones. They can exploit the whole Boolean space and, thus, can produce optimum solutions. The price to pay is that the time to compute good solutions is longer. In the Boolean decomposition there are much more candidate subfunctions for decomposition, and their evaluation is quite complex. We discuss in this section a Boolean division method based on MBDs. First, let us restate the definition of Boolean division.

Definition 7.10. Let f and d be two given functions. The *Boolean division* of f by d consists in finding functions q and r such that

$$f = q \cdot d \vee r$$

such that the supports of the q and d are not disjoint, i.e., $\sup(q) \cap \sup(d) \neq \emptyset$.

In the general case, f is an incompletely specified function $f_{min} \leq \tilde{f} \leq f_{max}$, while d is supposed to be a completely specified function. Then, q and r may be expressed by a set of inequalities:

$$\begin{array}{lll} \mathbf{0} & \leq q & \leq \overline{d} \vee f_{max} \\ f_{min} \cdot \overline{(q \cdot d)} & \leq r & \leq f_{max} \end{array}$$

Their graphical representation is presented in figure 7.10. Function q can range from **0** (in this case $r = f$) up to $\overline{d} \vee f_{max}$. The upper limit indicates the complement of the region where d is **1** and f is **0** - the forbidden region for q. Function r must at least to cover the region not covered by $q \cdot d$ (left hand inequality) and on the other extreme it is limited by f_{max}. It can not be greater than f_{max}, otherwise it will introduce **1**'s in points of the domain where f is **0**. In the figure, the lower limit of r is the internal polygon with dashed line. The square in the center of f indicates its don't care region. The r's upper limit is the square with thicker line. The empty square at left, tagged with letter 'd' is the forbidden region for q, i.e., q's OFF-set. Its ON-set is the indicated by the shaded region inside the intersection of rectangles d and q. All the rest is the q's don't care set. If we know d, then the Boolean division can be performed by the following steps:

1 - compute $\tilde{r} = \{ r_{on}, r_{off}, r_{dc} \}$
2 - minimize $\tilde{r}$ with respect to its don't care set, which produces r.
3 - compute $\tilde{q} = \{ q_{on}, q_{off}, q_{dc} \}$
4 - minimize $\tilde{q}$ with respect to its don't care set, generating function q.

The minimization is performed with the algorithms presented in chapter 5. r is computed in the following way:

1. $r_{on} = f_{min} \cdot \overline{d}$
2. $r_{dc} = f_{max} \cdot d$

The idea is to make r_{dc} as large as possible in order to provide more alternatives to the logic minimizer. Thus, we initially suppose that q will intersect all points $\{\mathbf{x} \in d^{-1}(1) \cap f_{max}^{-1}(1)\}$, such that $f_{max} \cdot d \cdot q = f_{max} \cdot d$. For the same reason $fmax$ is chosen instead of f_{min}. The logic minimization of $\tilde{r} = \{ r_{on}, r_{off}, r_{dc} \}$ yields the *rest* function r. The $\tilde{q}$ is computed as follows:

1. $q_{on} = f_{min} \cdot \overline{r}$
2. $q_{dc} = \overline{d} \vee r$

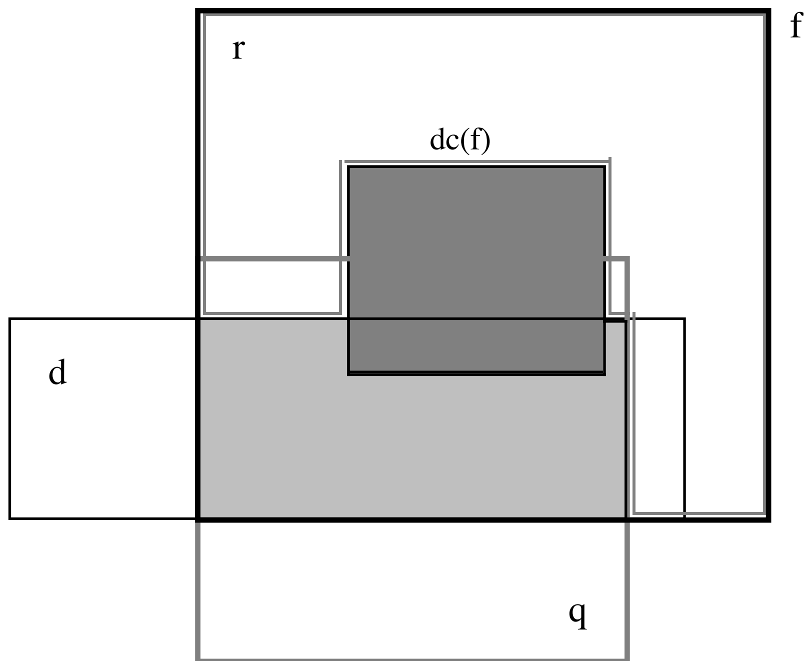


Figure 7.10. Graphic representation of the Boolean division.

Since r is already known, we can use it in computing $\tilde{q}$. The minimum ON-set that must be covered by $q{\cdot}d$ is the portion of f_{min} that is not covered by r. q_{dc} is given by everything that lies outside d plus the portion of r that eventually intersects d. The Boolean division algorithm is presented in figure 7.11. It is just a stepwise description of the ideas presented above.

```
function MBD_bool_div (f,d): MBD_PAIR
var f,d : MBD_TYPE;
begin
  var r, q, rdc, qdc:     MBD_TYPE;

  fm = f(dc->0);
  fM = f(dc->1);
  rdc = fM•d;              /* don't care set for r       */
  r = build_mbd(fm,rdc);   /* build r                    */
  MBD_minimize(r);         /* logic minimize r           */
  qdc = r v not(d);        /* compute don't care for q   */
  q = build_mbd(fm, qdc);  /* build q                    */
  MBD_minimize(q);         /* logic minimize q           */
  return (q,r);            /* return quocient and rest   */
end;
```

Figure 7.11. Algorithm for Boolean division.

Since the Boolean division is based on logic minimization, which in general is independent of the type of logic representation adopted, it can be used to implement functional decomposition. As the logic minimization is a complex task, the functional decomposition of large MBDs can be very time consuming. In this work, we restrict our attention to its application in the factorization of node functions in a Boolean network. Although there is no consensus about

how complex a node function can be (the only restriction is that it is a single output function), in practice we may control its complexity by selecting an appropriate decomposition method. In the methods presented in this section, one can easily restrict the complexity of node functions by limiting the number of variables in their support.

7.1.7.1 Boolean Divisors

The main problem in the Boolean factorization is the choice of a good divisor function. The fact that the Boolean division provides a larger solution space implies that the set of bad solutions is also larger. Since the selection of candidates is based on heuristics, there is no guarantee that the solution will be the best one (that produces functions q, d and r such that the total number of literals is minimum) or even that it will be always better that the algebraic division ([Bra87a] presents some efficient algebraic factorizations algorithms). However, as algebraic division is faster, it is possible to try both and select the best solution.

We study here two heuristics for finding Boolean divisors. The first one is based on the concept of *MBD-kernels*.

Definition 7.11. A *MBD-kernel* of a MBD M is any MBD M_k such that M_k is a cube-free primary divisor of M.

A MBD is *cube-free* is there are no literals $x_i^{e_i} \in$ support(M) such that $M < x_i^{e_i}$, where $e_i \in \{0, 1\}$ is the phase of variable x_i. To establish a correspondence with the kernel definition, note that if $M < x_i^{e_i}$ then M may be evenly divided by $x_i^{e_i}$. Clearly, a MBD-cube M_c is not cube-free, because $M_c < x_i^{e_i}$ for all literals $x_i^{e_i} \in$ support(M_c). The levels of a MBD-kernel are defined in the same faction as for kernels. Since computing all kernels is usually too costly, we select a divisor among the MBD-kernels of level 0. The algorithm is based on [Bra87], and is presented in figure 7.12.

The algorithm successively factors out variables in the support of the function up to find a cube-free expression composed by a sum of products where no literal appears in more than one cube. Initially, the algorithm is called with the MBD and its support as parameters. Each variable in the support is processed in turn. First, the MBD is cofactored with respect to one variable in the direct phase. If there is a set of literals $\{x_i^{e_i} \mid x_i^{e_i} < f_{x_1}\}$, the function is degenerated with respect to them (*make_cube_free*) and the cube composed by the conjunction of these literals is assigned to *cube* variable. If there exists one or more literals in the support of the cube that do not belong to the current support, then f_{x_1} is discarded because it was already processed in previous step. In the case that f_{x_1} does not produce a MBD-kernel of level 0, the process is repeated for f_{x_0}.

```
function MBD_kernel_0 (f, sup) : MBD_TYPE
var f :      MBD_TYPE;
    sup :    LIST;

begin
  var fx1, fx0, cube :  MBD_TYPE
  /* for all variables in the list sup */
  while ((var:=pop(sup)) != NULL) begin
    fx1 := cofactor(f, var, 1);          /*degenerate f: var=1  */
    if (! is_cube?(fx1)) then begin      /* if not cube         */
      fx1 := make_cube_free(fx1,cube);   /* cube={xi|fx1 < xi}  */
      if (support(cube) ⊆ sup) then      /*if lit not processed */
       return(MBD_kernel_0(fx1,sup));    /* continue on fx1     */
    end;
    /* if fx1 does not contains a MBD_kernel0, look at fx0 */
    fx0 := cofactor(f, var, 0);          /*degenerate f: var=0  */
    if (! is_cube?(fx0)) then begin      /* if not cube         */
      fx0 := make_cube_free(fx0,cube);   /* cube={xi|fx0 < xi}  */
      if (support(cube) ⊆ sup) then      /*if lit not processed */
       return(MBD_kernel_0(fx0,sup));    /* continue on fx0     */
    end;
  end;
  /* if no MBD_kernel0 found, return f */
  return (f);
end;
```

Figure 7.12. Algorithm for finding one level 0 MBD-kernel.

The second heuristic to find Boolean divisors try to estimate the gain of a divisor by analyzing its OFF-set. The idea is that the OFF-set of the divisor d becomes the don't care set of the quotient function q. According to the algorithm *MBD_Bool_div()*, $q_{on} = f_{on}$ and $q_{dc} = \overline{d} \vee r$. Therefore, one way to evaluate a candidate divisor d would be to verify how much f is simplified if we make $f_{dc} = \overline{d}$ and minimize f. Since logic minimization is costly, some approximations are considered instead. One possibility is to generate a sum-of-products of f, add the don't care $\overline{d}$ to f and check how much cubes of the ON-set of f may be expanded with respect to the new DC-set. In figure 7.13 we present an algorithm that uses this technique to select a Boolean divisor for f.

First, the sum-of-products of the function (MBD-cover) is computed. The MBD-kernels of level 0 are taken as candidate divisors. Each candidate k is then evaluated. The don't care of f is set to the OFF-set of k. Then, every cube of f is tested for expansion with the new don't care set and the total number of literals deleted from the expanded cubes is computed. The gain is the difference between the number of literals saved and the number of literals of the MBD-cover of k. This allows taking into account the complexity of k in the divisor selection. The kernel which produces the highest gain is selected as divisor.

```
function sel_Bool_div (f) : MBD_TYPE
var f :       MBD_TYPE;

begin
  var sop:             MBD_COVER;
      kl:              KERNEL_LIST;
      lit:             integer;
      gain, maxgain:   integer;
      div:             MBD_TYPE;

  sop := MBD_sop (f);
  kl := MBD_kernels_0(f);
  foreach k in kl begin
    fdc := not(k);
    foreach cube in sop
      lit := lit + number of deleted literals;
    gain := lit - literals(k);
    if (gain > maxgain) then begin
      gain = maxgain;
      div = k;
    end;
  end;
  return(div);
end;
```

Figure 7.13. Algorithm for the selection of a Boolean divisor.

7.1.7.2 Factorization Examples

We present in this section the application of some factorization techniques over a set of small functions obtained from the literature. The experiments compare the use of different approaches:

- ALG - algebraic factorization based on weak division [Bra87]
- PATH - path oriented factorization
- B-ANY-K0 - Boolean factorization using a MBD-kernel of level 0 as divisor
- B-OFF-K0 - Boolean factorization using OFF-set based divisor selection.

Function 1: 36 literals and 8 cubes

$$f = x_1 \cdot x_2 \cdot x_3 \cdot x_4 \cdot x_5 \cdot x_6 \lor x_1 \cdot x_2 \cdot \bar{x}_5 \cdot \bar{x}_6 \lor x_1 \cdot x_2 \cdot \bar{x}_3 \cdot \bar{x}_4 \lor x_3 \cdot x_4 \cdot \bar{x}_5 \cdot \bar{x}_6 \lor \bar{x}_3 \cdot \bar{x}_4 \cdot x_5 \cdot x_6 \lor \bar{x}_1 \cdot \bar{x}_2 \cdot x_3 \cdot x_4 \lor \bar{x}_1 \cdot \bar{x}_2 \cdot x_5 \cdot x_6 \lor \bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_3 \cdot \bar{x}_4 \cdot \bar{x}_5 \cdot \bar{x}_6$$

ALG: 28 literals

$$f = (x_4 \cdot x_3 \lor x_6 \cdot x_5 \lor \bar{x}_6 \cdot \bar{x}_5 \cdot \bar{x}_4 \cdot \bar{x}_3) \cdot \bar{x}_1 \cdot \bar{x}_2 \lor (\bar{x}_4 \cdot \bar{x}_3 \lor x_4 \cdot x_3 \cdot x_2 \cdot x_1) \cdot x_5 \cdot x_6 \lor (x_2 \cdot x_1 \lor x_4 \cdot x_3) \cdot \bar{x}_5 \cdot \bar{x}_6 \lor \bar{x}_4 \cdot \bar{x}_3 \cdot x_2 \cdot x_1$$

PATH: 48 literals

$$f = \bar{x}_2 \cdot \bar{x}_4 \cdot \bar{x}_1 \cdot (\bar{x}_6 \cdot \bar{x}_3 \cdot \bar{x}_5 \lor x_3 \cdot x_6) \lor x_2 \cdot x_4 \cdot x_1 \cdot (x_6 \cdot x_3 \cdot x_5 \lor \bar{x}_3 \cdot \bar{x}_6) \lor$$

$\bar{x}_2 \cdot x_4 \cdot \bar{x}_1 \cdot (\bar{x}_5 \cdot x_3 \cdot x_6 \vee x_5) \vee (\bar{x}_2 \cdot \bar{x}_4 \cdot x_1 \vee x_2 \cdot \bar{x}_4 \cdot \bar{x}_1) \cdot \bar{x}_5 \cdot x_3 \cdot x_6 \vee$
$(\bar{x}_2 \cdot x_4 \cdot x_1 \vee x_2 \cdot x_4 \cdot \bar{x}_1) \cdot x_5 \cdot \bar{x}_3 \cdot \bar{x}_6 \vee x_2 \cdot \bar{x}_4 \cdot x_1 \cdot (x_5 \cdot \bar{x}_3 \cdot \bar{x}_6 \vee \bar{x}_5)$

B-ANY-K0: 30 literals

$f = ((\bar{x}_4 \vee \bar{x}_6) \cdot (\bar{x}_1 \vee \bar{x}_4 \cdot \bar{x}_3 \vee \bar{x}_6 \cdot \bar{x}_5) \vee x_6 \cdot x_5 \cdot x_4 \cdot x_3) \cdot ((x_6 \cdot x_5 \vee \bar{x}_6 \cdot \bar{x}_5 \cdot \bar{x}_4 \cdot \bar{x}_2 \cdot \bar{x}_1) \cdot \bar{x}_3$
$\vee \bar{x}_5 \cdot x_4 \cdot x_3 \vee x_2 \cdot x_1) \vee (x_6 \cdot x_5 \vee x_4 \cdot x_3) \cdot \bar{x}_1 \cdot \bar{x}_2$"

B-OFF-K0: 27 literals

$f = (x_3 \cdot (x_6 \cdot x_5 \cdot x_4 \cdot x_2 \cdot x_1 \vee \bar{x}_2 \cdot \bar{x}_1) \vee \bar{x}_6 \cdot \bar{x}_5 \vee \bar{x}_4 \cdot \bar{x}_3 \cdot x_1) \cdot (\bar{x}_4 \cdot \bar{x}_3 \cdot \bar{x}_2 \cdot \bar{x}_1 \vee x_4 \cdot x_3 \vee x_2 \cdot x_1)$
$\vee x_6 \cdot x_5 \cdot (\bar{x}_4 \cdot \bar{x}_3 \vee \bar{x}_2 \cdot \bar{x}_1)$

Function 2: 15 literals and 5 cubes

$f = \bar{x}_1 \cdot x_3 \cdot x_5 \vee x_1 \cdot x_2 \cdot x_3 \vee x_1 \cdot x_2 \cdot \bar{x}_5 \vee x_1 \cdot x_3 \cdot x_4 \vee x_1 \cdot x_4 \cdot \bar{x}_5$

ALG: 8 literals

$f = (x_4 \vee x_2) \cdot x_1 \cdot (x_3 \vee \bar{x}_5) \vee \bar{x}_1 \cdot x_3 \cdot x_5$

PATH: 8 literals

$f = x_1 \cdot (x_4 \vee x_2) \cdot (\bar{x}_5 \vee x_3) \vee \bar{x}_1 \cdot x_3 \cdot x_5$

B-ANY-K0: 7 literals

$f = ((x_4 \vee x_2) \cdot x_1 \vee x_5 \cdot \bar{x}_1) \cdot (\bar{x}_5 \vee x_3)$

B-OFF-K0: 7 literals

$f = (x_1 \cdot (x_4 \vee x_2) \vee x_5 \cdot \bar{x}_1) \cdot (\bar{x}_5 \vee x_3)$

Function 3: 17 literals and 6 cubes

$f = \bar{x}_1 \cdot \bar{x}_2 \cdot x_3 \cdot x_5 \vee x_1 \cdot \bar{x}_2 \cdot \bar{x}_4 \vee x_1 \cdot x_3 \cdot x_6 \vee \bar{x}_3 \cdot \bar{x}_4 \vee x_2 \cdot \bar{x}_5 \cdot x_6 \vee x_2 \cdot x_4$

ALG: 14 literals

$f = (\bar{x}_3 \vee x_1 \cdot \bar{x}_2) \cdot \bar{x}_4 \vee (x_6 \cdot \bar{x}_5 \vee x_4) \cdot x_2 \vee (x_6 \cdot x_1 \vee x_5 \cdot \bar{x}_1 \cdot \bar{x}_2) \cdot x_3$

PATH: 20 literals

$f = \bar{x}_1 \cdot x_3 \cdot (x_4 \cdot \bar{x}_5 \cdot (x_6 \vee x_2) \vee x_5 \cdot (\bar{x}_4 \vee x_2)) \vee x_1 \cdot x_3 \cdot (\bar{x}_6 \cdot (\bar{x}_2 \cdot \bar{x}_4 \vee x_2 \cdot x_4) \vee x_6) \vee$
$\bar{x}_3 \cdot (x_4 \vee \bar{x}_2)$

B-ANY-K0: 19 literals

$$f = (\bar{x}_2 \vee x_1 \vee x_4)\cdot((x_6\cdot x_1 \vee x_5\cdot \bar{x}_1)\cdot x_3 \vee x_4\cdot x_2) \vee (\bar{x}_3 \vee x_6\cdot \bar{x}_5)\cdot(x_2 \vee \bar{x}_3\cdot \bar{x}_4) \vee x_1\cdot \bar{x}_4\cdot \bar{x}_2$$

B-OFF-K0: 15 literals

$$f = \bar{x}_2\cdot(x_1\cdot \bar{x}_4 \vee x_5\cdot \bar{x}_1\cdot x_3) \vee x_6\cdot(x_1\cdot x_3 \vee \bar{x}_5\cdot x_2) \vee \bar{x}_3\cdot \bar{x}_4 \vee x_4\cdot x_2$$

Function 4: 41 literals and 13 cubes

$$f = x_2\cdot x_8 \vee x_2\cdot x_9 \vee x_1\cdot x_2\cdot x_5\cdot \bar{x}_7 \vee x_1\cdot x_2\cdot x_5\cdot x_6 \vee x_1\cdot x_2\cdot \bar{x}_5\cdot x_7 \vee x_4\cdot \bar{x}_5\cdot x_7 \vee x_4\cdot x_5\cdot x_6 \vee x_4\cdot x_5\cdot \bar{x}_7 \vee x_1\cdot x_3\cdot \bar{x}_5\cdot x_7 \vee x_1\cdot x_3\cdot x_5\cdot x_6 \vee x_1\cdot x_3\cdot x_5\cdot \bar{x}_7 \vee x_3\cdot x_9 \vee x_3\cdot x_8$$

ALG: 16 literals

$$f = (x_3 \vee x_2)\cdot(((x_6 \vee \bar{x}_7)\cdot x_5 \vee x_7\cdot \bar{x}_5)\cdot x_1 \vee x_8 \vee x_9) \vee ((\bar{x}_7 \vee x_6)\cdot x_5 \vee x_7\cdot \bar{x}_5)\cdot x_4$$

PATH: 20 literals

$$f = ((\bar{x}_2\cdot \bar{x}_9\cdot x_3 \vee \bar{x}_9\cdot x_2)\cdot \bar{x}_8\cdot(x_1 \vee x_4) \vee \bar{x}_2\cdot x_4\cdot \bar{x}_3)\cdot(x_7\cdot(x_6 \vee \bar{x}_5) \vee x_5\cdot \bar{x}_7) \vee (x_2 \vee x_3)\cdot(x_9 \vee x_8)$$

B-ANY-K0: 14 literals

$$f = (x_7\cdot \bar{x}_5 \vee x_6\cdot x_5 \vee \bar{x}_7\cdot x_5)\cdot((x_2 \vee x_3)\cdot x_1 \vee x_4) \vee (x_8 \vee x_9)\cdot(x_2 \vee x_3)$$

B-OFF-K0: 19 literals

$$f = (x_1\cdot(x_5 \vee x_7)\cdot(\bar{x}_7 \vee x_6) \vee x_8 \vee x_9 \vee x_7\cdot \bar{x}_5\cdot x_1)\cdot(x_2 \vee x_3) \vee x_5\cdot x_4\cdot(\bar{x}_7 \vee x_6) \vee x_7\cdot \bar{x}_5\cdot x_4$$

Function 5: 21 literals and 6 cubes

$$f = \bar{x}_1\cdot x_3\cdot x_4\cdot \bar{x}_5 \vee \bar{x}_1\cdot x_2\cdot x_3\cdot \bar{x}_4 \vee x_2\cdot \bar{x}_3\cdot x_4\cdot x_5 \vee x_1\cdot x_2\cdot \bar{x}_3 \vee x_1\cdot \bar{x}_3\cdot x_5 \vee x_1\cdot \bar{x}_3\cdot x_4$$

ALG: 14 literals

$$f = ((x_5\cdot x_4 \vee x_1)\cdot x_2 \vee (x_4 \vee x_5)\cdot x_1)\cdot \bar{x}_3 \vee (\bar{x}_4\cdot x_2 \vee \bar{x}_5\cdot x_4)\cdot \bar{x}_1\cdot x_3$$

PATH: 23 literals

$$f = (\bar{x}_1\cdot \bar{x}_5\cdot x_2 \vee x_1\cdot x_5\cdot \bar{x}_2)\cdot \bar{x}_4\cdot x_3 \vee x_2\cdot \bar{x}_1\cdot \bar{x}_2\cdot x_5 \vee \bar{x}_2\cdot x_1\cdot x_4\cdot \bar{x}_5 \vee \bar{x}_2\cdot \bar{x}_1\cdot x_4\cdot x_3 \vee x_1\cdot \bar{x}_4\cdot x_2$$

B-ANY-K0: 14 literals

$$f = (x_2 \vee x_5 \vee x_4)\cdot(((x_5\cdot x_4\cdot \bar{x}_3 \vee \bar{x}_4\cdot x_3)\cdot x_2 \vee \bar{x}_5\cdot x_3)\cdot \bar{x}_1 \vee \bar{x}_3\cdot x_1)$$

B-OFF-K0: 13 literals

$$f = (x_3 \cdot \bar{x}_1 \vee \bar{x}_3 \cdot x_1) \cdot (\bar{x}_5 \cdot x_4 \vee \bar{x}_4 \cdot x_2) \vee x_5 \cdot \bar{x}_3 \cdot (x_1 \vee x_4 \cdot x_2)$$

These results are summarized in table 7.4. The field *Lit* shows the number of literals of the sum-of-products form. The best results are indicated in bold type characters. We may see that the Boolean division with OFF-set based selection of the divisor obtained the best result most of times. In one case, algebraic factorization wins. One advantage of algebraic factorization is that it guarantees that the factored form will have less literals than the sum-of-products, due to its algebraic properties. In the Boolean methods presented here there is no way of asserting that. However, empirical results indicates that Boolean factorization usually performs better than the algebraic one. Path oriented factorization have in average a poorer performance. In some cases increasing the number of literals with respect to initial sum-of-products description, as in examples 1, 3 and 5. Clearly, this kind of factorization should be further developed to produce interesting results.

Example	Lit	Alg	Path	One-k0	Off-k0
1	36	28	48	30	**27**
2	15	8	8	**7**	**7**
3	17	**14**	20	19	15
4	41	16	20	**14**	19
5	21	14	23	14	**13**

Table 7.4. Factorization results.

7.1.8 Comments

In this section we have presented some decomposition and factorization methods based on MBDs. Among then, the direct decomposition and the path oriented decomposition and factorization relies on the particularities of the MBD topology, while the Boolean factorization techniques does not explicitly depend on the MBD representation. The direct and path oriented decompositions are alternative ways to split a MBD into a set of subfunctions. Since they are relatively fast, one can try both of them and select the best result. The Boolean division presented here is too costly to be applied to large circuits, but it is useful for the decomposition and factorization of node functions in a Boolean network, which have smaller complexity.

The results of the direct decomposition indicate that it is an interesting decomposition method. The Boolean networks it produces are of reasonable complexity and the algorithm is very fast. Of course, to be effective this method needs a previous MBD reordering to reduce the MBD size. Thus, the reordering time must be taken into account too. The algorithm is flexible in the

sense that it is easy to tune the complexity of the node functions, defined in terms of the number of variables in the support of the function . *Fine granularity* decomposition may be obtained by restricting the support size to two variables, for instance. This produces a larger network but with simpler node functions. Another decomposition parameter, the amount of sharing of a subfunction - estimated by the number of fathers of the subfunction root - accounts for the reduction of the network size, since it identifies subfunctions that are used several times in the multi-level representation. It should be used concurrently with the support size, as an alternative cost function. In this case, a subMBD is extracted both if its support has the expected size or if its number of fathers reaches the specified threshold. If the cost function to be minimized is the global MBD size (the sum of the MBD size of each subfunction), then the size of the subMBDs may replace the support size as the main decomposition parameter, used concurrently with the subfunction sharing.

The path oriented decomposition and factorization has not produced so good results. But we believe that is can be further improved by a deeper analysis of the splitting nodes selection, which may consider also the MBD supernodes. As supernodes represent simple factorable functions, they should not be destroyed by the splitting nodes.

The Boolean factorization techniques where developed just to show that MBDs can also be used in this field. The logic minimization algorithms may profit from the speed of the logic operations performed on MBDs. We presented also a method that allows the extraction of kernels directly from MBDs, here called MBD-kernels, which provides good candidates for logic division. The selection of the divisors was developed based on the MBD-covers of the functions. We believe, however, that is possible to develop methods for the evaluation of candidate divisors that works directly on the MBD representation, without need of converting it to a sum-of-products form.

The time complexity of the direct decomposition is linear with respect to the MBD size. The path oriented decomposition is more complex. Each node of the *q* subfunctions is checked for redundancy, which is computed by checking for implication between its son's functions. As the implication is proportional to the product of the size of both son's functions, the time complexity is bound by $O(n^2)$, where n is the MBD size. This is still acceptable for a large class of practical circuits. The Boolean methods involve logic minimization, which is a NP-complete problem. We do not have a precise estimation of the time complexity of the minimization heuristics developed in chapter 5, but empirically we may state that this is not practical for large circuits, but they can be applied to node functions, which are simpler.

Finally, it should be noted that in the case of direct and path oriented decomposition the BDD extensions proposed in the literature, namely the strong canonical form and the negative edges, would be useful. The negative edges allow a fast verification of the existence and use

of the complement of the extracted subfunctions in both methods. This is implemented here by calling an equivalence checking algorithm. The strong canonical form will be useful for the path oriented decomposition because it keeps the global MBD always updated. Here this is implemented by reducing the set of subfunctions extracted, which takes more time.

7.2 Multi-level Minimization

One useful feature of two-level expressions is the existence of an accurate cost function, that closely correlates to the implementation cost. For multi-level circuits, the cost function is not so clear. In fact, there is no simple answer to the question: “how good is one multi-level implementation?”. The problem is that we are not able to find or to make a good estimation of *the* optimum solution for a multi-level circuit. Consider the case of area minimization, for instance. A multi-level circuit is optimum if it has the smallest possible area. But there is a large set of factors that may affect the final circuit surface. Some of them may be treated at the logic level. Others are related to specifics technological features, as the preference for certain logic gates and the availability of several layers for interconnections, which eases the work of the placement and routing tools. These information have different nature and their integration in a unified synthesis environment is long term research work.

The search of simple cost functions to guide logic synthesis tool leads to the definition of abstract costs at the logic level that may lead to *good* implementations. A common accepted abstraction of the circuit cost is the total number of literals of the Boolean network. Some experiences [Lig88] have demonstrated that this in fact can be a good estimation for the circuit complexity for a large set of problems. In this case, one criteria for optimality may be found by extending the concepts of primality and irredundance to the multi-level case.

Definition 7.12. A Boolean network η is *prime* if no literal can be removed from any cube of any node function cover without changing the Boolean function it represents. It is *irredundant* if no cube can be deleted from any node function cover without changing the Boolean function denoted by η.

One potential problem with this cost function is that it does not take into account the connection’s complexity, which plays an important role for larger circuits. Some recent works have addressed the problem of interconnection complexity at the logic level. [Abo90] tries to simplify the connections by introducing an ordering in which the variables enter in the subfunctions trees and also by controlling the fanout of subfunctions obtained along the decomposition. [Hwa89] models the interconnection problem by the number of interconnection among logic blocks. It proposes a tree like decomposition where the number of interconnection between the blocks is reduced. [Ped91] addresses the interconnection problem during the technology mapping step. It simulates the placement of the cells and make

estimation of the interconnection costs based on a *virtual* placement. It is a quite sophisticated work, and it is striking that this as well as the other approaches have produced so little improvement over the synthesis based on the literal count. This fact seems to indicate that the solutions found with the literal count cost function may be near to the optimum. These information must be taken with care, however, because we can not discard the possibility that this lack of significant improvements comes from our limitations in dealing with the complexity of modeling technological features at the logic level. In general, we found that the literal count is still a good estimation of the circuit cost and we adopt it here as the cost function to be minimized. The interconnection complexity may be indirectly minimized if we look for logic transformations that produce subfunctions with smaller supports.

There are several minimization methods oriented to the reduction of the literal count and to the elimination of redundant logic in a Boolean network. All these minimization can be modeled in terms of don't cares [Bra89]. There are globally two kind of don't cares. *External don't cares* refers to the set of input vectors for which the output of the function is not meaningful. A classical example is the BCD (Binary Coded Decimal) to binary code conversion, where the input values corresponding to decimal digits above 9 are don't cares because they never occur. The *internal* or *implicit* don't cares [Bar88][Bra89] are non-meaningful input conditions derived from the multi-level nature of the circuit. There are two kinds of implicit don't cares: the *observability* (ODC) and the *satisfiability* (SDC) don't cares [Bra89]. Both of them can be used together with the external don't care to simplify the node functions in the Boolean network and, by consequence, reduce the logic complexity of the overall circuit. We make now a brief survey on the main techniques used to simplify multi-level circuits.

Muroga [Mur89] has developed a method called TRANSDUCTION (TRANSformation and reDUCTION) that computes the set of *permissible functions* of each node and uses it to simplify the circuit. The *permissible functions* of a node are all functions that can be assigned to the node without modifying the output functions of the circuit. Node functions are expressed in terms of the primary inputs. As the set of permissible functions may be huge, the minimization relies on a subset of them, called the Compatible Set of Permissible Functions (CSPF). An interesting property of the CSPF is that it allows the simultaneous simplification of distinct nodes, without need of recomputing the CSPF of the other nodes at each transformation. The CSPF is computed by traversing the network from the outputs up to the inputs. The output nodes that have no fanout have their don't care sets initialized with the external don't cares. This method was first conceived to work on two inputs NOR circuits, and was extended to work with general node functions in [Mat89] and [Sav90].

The *global flow* [Bra83][Ber88b] is a method that applies compiler techniques to the multi-level logic minimization problem. The idea is to use logic implications in the form $y_i = b_i$ -> $y_j = b_j$, where y_i and y_j are input or internal nodes of the Boolean network and

b_i and $b_j \in \{0,1\}$. In the original method [Bra83] the network should be specified in terms of a simple gate, e.g., NOR gates. In this case, if one gate input is set to 1, then the output must be 0, independent of the other gate inputs: $y_i = 1 \rightarrow y_j = 0$. The method was generalized in [Bra88] to deal with more than one type of gate. The implications are used to compute the *forcing sets*, which establish a relationship among signals in the network. Based on the forcing sets the frontier of influence of a signal are computed and the gates and/or connections in this region may be transformed, simplifying the circuit.

Another interesting approach uses automatic test pattern generation (ATPG) techniques to logic optimize the circuit [Bry89][Jac89]. The idea is that ATPG algorithms can identify and locate faults in the network that can not be tested, the redundant faults. The fact that the outputs are insensitive to those faults indicates the existence of redundant logic in the circuit. Removing all redundant logic produces a logic optimized circuit that is 100% testable for single stuck-at faults.

The single stuck-at fault is the most used fault model for testing circuits. It assumes that there is only one fault at a time in the network, and it is modeled by setting one fanout stem to a constant value, 0 or 1. To detect such fault, the ATPG system look for an input vector that sets the fanout stem to the opposite value and at the same time propagates it to at least one output. Most ATPG algorithms use fault simulation and logic implication to propagate the logic signals through the network to verify if the fault is testable. In this process, sometimes the logic implications produce logic contradictions, e.g., a signal that is set to two distinct values. In this case, the algorithm backtracks and modify a previous logic assertion and repeats the process. When the number of backtracks is too high, the algorithm is interrupted and the fault is referred to as an aborted fault. If the number of backtracks is high, the algorithm may have a poor performance. If there are aborted faults, it may not be able to find all redundant faults. For those reasons, ATPG techniques were lagged behind other approaches in the logic minimization area. This situation changed with the significant improvements discovered by Schultz [Sch88] and incorporated to the SOCRATES ATPG system. Based on the FAN algorithm [Fuj85], SOCRATES pioneered the use of contrapositive (or backward) implications in the ATPG context. This and other improvements allowed for the detection of all irredundant faults with a backtrack limit of ten, and no aborted faults. Moreover, experimental results show that this heuristics are of linear complexity, which indicates that ATPG based logic minimization may be quite effective for circuits that have redundant logic.

TRANSDUCTION, Global Flow and the ATPG approaches depend on the circuit topology and on the type of the gates to compute meaningful information for the network minimization. A somewhat different approach was proposed in BOLD [Bos87], which is based on multi-level tautology checking. The method is an extension of the ESPRESSO [Bra84] algorithm to deal with the multi-level case. The algorithm manipulates the node function's covers, expanding

and reducing them while controlling the consistency of the transformations through multi-level tautology checking. It not only eliminates literals and cubes from the node's covers but also introduce new variables in the support of the node functions, executing multi-level resubstitution. Its performance relies on the efficiency of the tautology checking algorithms, which takes advantage of the locality of the transformations to perform fast Boolean verification. The resulting network is multi-level prime and irredundant, 100% single stuck-at fault testable and the test set is provided as a by product of the minimization.

All these methods use the don't care sets in a implicit way. On the other hand, MisII [Bra87] performs multi-level minimization by explicitly computing subsets of the observability [Sav90] and satisfiability [Sal89] don't cares for each node function. It then uses a two-level minimizer (ESPRESSO) to simplify the node functions and reduce the network complexity. It was shown in [Sav90] that this approach not only may simplify the node functions but also can introduce new variables in their support, re-expressing them in a simpler way. Since both SDC and ODC may be huge in practical circuits, the choice of an adequate subset of them is a very important problem. [Sal89] presents some filters for the selection of subsets of the SDC. In short, when minimizing a node n, they propose to build the SDC(n) using only the nodes whose fanin is a subset of the fanin of n. In [Sav90] a technique based on the TRANSDUCTION method is presented to compute subsets of the ODC.

In this work, we opted to implement this later technique for the minimization of multi-level circuits. Besides of producing results of similar quality to the other approaches, this method allows us to apply the minimization techniques presented in chapter 5 to simplify the node functions. In the next sections we present a more detailed definition of the SDC and ODC sets, followed by some examples of its application.

7.2.1 Implicit Don't Care Sets

We introduce the SDC by means of an example. Consider the function:

$$f(x_1,x_2,x_3,x_4) = x_1 \vee x_2 \cdot (x_3 \vee x_4)$$

When decomposing this function, suppose we introduce a subfunction $y = x_3 \vee x_4$. Then, f can be expressed as:

$$f(x_1,x_2,x_3,x_4,y) = x_1 \vee x_2 \cdot y$$

The introduction of y in the support of f doubles the size of its domain. But y is not a primary input, it represents an intermediate function that is itself expressed in terms of the primary inputs. Clearly, the relation established by the equation $y = x_3 \vee x_4$ must be respected in all points of the new domain of f. This means that the conditions that leads both to $y = 0$ and

$x_3 \vee x_4 = 1$ or to $y = 1$ and $x_3 \vee x_4 = 0$ in fact never occur and these points have undefined values in the new domain of f. Figure 7.14 illustrates this example by means of a Karnaugh diagram.

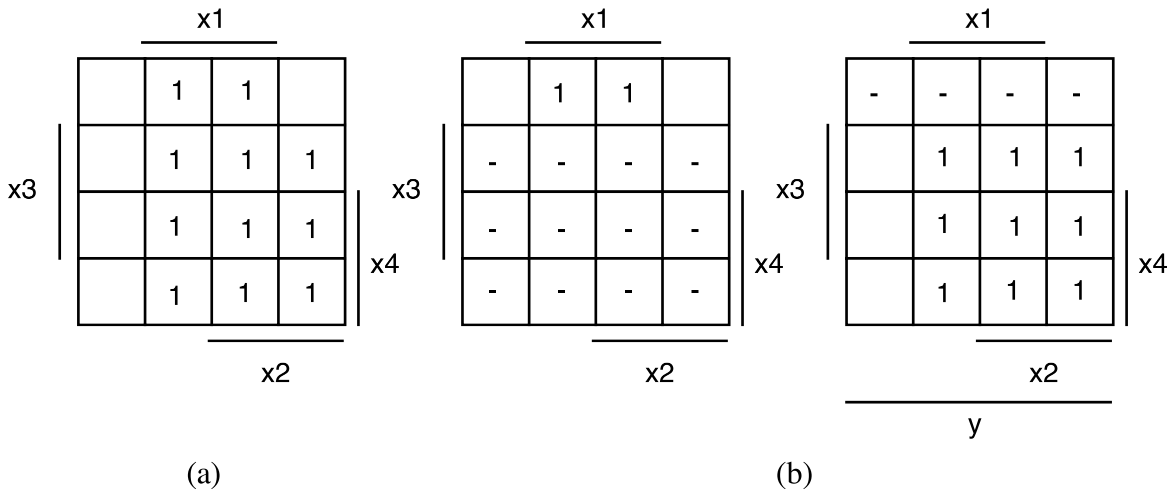


Figure 7.14. Effect of introducing an intermediate variable.

Figure 7.14(a) shows the original function f, while 7.14(b) shows the function with its new support. Note that the number of care points remains the same. Thus, each new intermediate variable introduced doubles the size of the function domain by adding to it a set of don't care vertices. The don't cares are distributed along the two new subdomains $y = 0$ and $y = 1$ according to the equation that relates y to the variables on the previous Boolean space.

The SDC is thus a set of *virtual points* in the extended Boolean space of the multi-level function. In fact, the variables that define the Boolean space will never be set to values that correspond to vertices in the SDC because the relation

$$y_i = f_i(\mathbf{X}, \mathbf{Y})$$

must be *satisfied* for all intermediate variables y_i on the Boolean network. The contribution of a node n_i to the global SDC is noted SDC(n_i) and is defined based on the fact that we can not have $y_i \neq f_i(\mathbf{X}, \mathbf{Y})$:

$$\text{SDC}(n_i) = y_i \oplus f_i, \text{ where } \oplus \text{ is the exclusive-or operation.}$$

The global SDC for a network η is given by the sum of the contribution of each non-primary input node of η :

$$\text{SDC}(\eta) = \sum y_i \oplus f_i, \text{ for all } n_i \in \eta \setminus \{\text{primary input nodes of } \eta\}$$

In figure 7.15 we give an example of the SDC for a simple Boolean network. The Karnaugh diagram shows f expressed in terms of x_1, x_2 and y. The SDC due to node y is obtained by:

$$\mathrm{SDC}(y) = y \oplus x_1.x_2 = \bar{y}\cdot x_1.x_2 + y\cdot(\bar{x}_1 \vee \bar{x}_2)$$

Simplifying f with this SDC we obtain $f = x_1$.

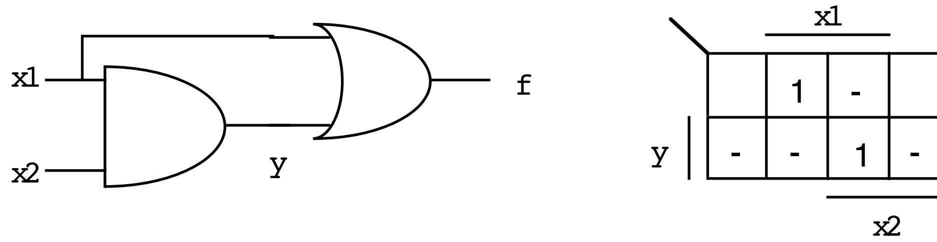


Figure 7.15. SDC for a simple network.

The ODC is the don't care that arise from the fact that for some input vectors the output functions of the network do not depend on some intermediate variables. This occurs due to the blocking of the paths that connect that variable to the outputs by other intermediate variable values. The ODC for a node n_i with respect to an output f_k can be computed by finding all the points in the extended Boolean subspace where $f_k(y_i = 0) = f_k(y_i = 1)$. This means that for those points we can set $y_i = 0$ or $y_i = 1$ and the value of f_k is not affected. Thus, in these cases f_k is independent of y_i. They can be computed using the concept of *Boolean difference* of a function f with respect to a variable y_i :

$$\frac{\partial f}{\partial y_i} = f_{y_i} \oplus f_{\bar{y}_i}$$

Since the exclusive-or operation gives the set of points where two functions differs, the ODC of a node n_i with respect to an output f_k may be computed by finding the complement of the Boolean difference between f_k and y_i:

$$\mathrm{ODC}(n_i) = \overline{\frac{\partial f_k}{\partial y_i}}$$

The ODC for multiple output functions is the conjunction of the ODCs for each output.

$$\mathrm{ODC}(n_i) = \Lambda\left(\overline{\frac{\partial f_k}{\partial y_i}}\right), \text{ for all } f_k \in \{\text{primary outputs of } \eta\}$$

This equation states that if some function f_k depends on y_i for some input vector $\mathbf{x_j}$, then $\mathbf{x_j}$ can not be added to $\mathrm{ODC}(n_i)$. The example from figure 7.15 can be minimized also by the ODC. Computing the $\mathrm{ODC}(y)$, we find:

$$\mathrm{ODC}(y) = \overline{f_y \oplus} f_{\bar{y}} = \overline{(x_1 \vee y)}|_{y=1} \oplus (x_1 \vee y)|_{y=0} = \overline{1 \oplus} x_1 = x_1$$

The new function associated to y is presented in figure 7.16. Minimizing y we have either $y = 0$ or $y = x_1$. Replacing y by any one of these values produces $f = x_1$.

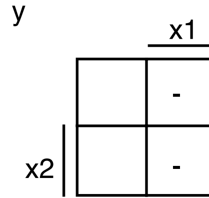


Figure 7.16. ODC for node y.

7.2.2 Node Minimization

The method we have implemented consists in computing subsets of the SDC and ODC and using the minimization techniques presented in chapter 5 to minimize the node functions. The SDC subset that is computed is based on the subset support filter proposed at [Sal89]. First we have tried to compute the whole SDC for a node to check if the MBD representation could be compact enough to allow treating the general case. However, we could quickly verify that it becomes too large even using MBDs. For instance, for the Z4 circuit with only 14 subfunctions, the MBD of one subfunction with the non filtered SDC reached 322 nodes, while the MBD of the same subfunction built with the filtered SDC had only 9 nodes. The problem here is not only the size of the MBD, but also the fact that most of times the non filtered don't cares are not useful for the simplification of the node function.

Let us define a notation convention: the SDC *from* a node n_i is its contribution to the global SDC of the circuit, obtained by the equation SDC(n_i) = $y_i \oplus f_i$. The SDC *for* a node n_i is the SDC used to simplify it. It is obtained by computing the contribution of all nodes n_k such that support(n_k) $\subseteq$ support(n_i). The first step before computing the SDC for any node is to setup the SDC from every node of the network. Then the SDC for a node n_i is computed by simple ORing the SDCs of nodes whose support is a subset of n_i.

As we have seen above, the ODC of a node n_i may be found by computing the Boolean difference of each output with respect to y_i. Of course, we can directly apply the Boolean difference to an output f_k only if f_k depends explicitly on y_i. Otherwise, if y_i is in the transitive fanin of f_k then there is a chain rule that can be used, but it becomes complex quickly. Suppose that f_k depends on y_1, ..., y_n, and y_n depends explicitly on y_i. Then:

$$\frac{\partial f_k}{\partial y_i} = \frac{\partial f_k}{\partial y_1}\frac{\partial y_1}{\partial y_i} \oplus \cdots \oplus \frac{\partial f_k}{\partial y_n}\frac{\partial y_n}{\partial y_i} \oplus$$

$$\frac{\partial^2 f_k}{\partial y_1 \partial y_2}\frac{\partial y_1}{\partial y_i}\frac{\partial y_2}{\partial y_i} \oplus \frac{\partial^2 f_k}{\partial y_1 \partial y_3}\frac{\partial y_1}{\partial y_i}\frac{\partial y_3}{\partial y_i} \oplus \cdots \oplus \frac{\partial^2 f_k}{\partial y_{n-1} \partial y_n}\frac{\partial y_{n-1}}{\partial y_i}\frac{\partial y_n}{\partial y_i} \oplus$$

$$\cdots \oplus$$

$$\frac{\partial^n f_k}{\partial y_1 \cdots \partial y_n} \frac{\partial y_1}{\partial y_i} \cdots \frac{\partial y_n}{\partial y_i}$$

The subset of the ODC used here is computed using a simplified model derived form [Hac89] and [Bra89]. Let

$$\mathrm{E_{i,j}} = \{ \overline{\frac{\partial f_j}{\partial y_i}}(\mathbf{X}) \}$$

be the observability don't care of node n_i with respect to a node n_j in the immediate fanout of n_i. In fact, it is the ODC subset associated to edge (n_i, n_j). Moreover, if f is a Boolean function and n_i an intermediate node in a Boolean network, *restrict*(f, n_i) is the operation that produces a new function g obtained by degenerating f with respect to all variables in the transitive fanout of n_i. Then, the recursive formula below gives a simple and efficient approximation of the ODC for a node n_i, called OD(n_i):

- if node n_i is a primary output, then OD(n_i) = external don't care for n_i, otherwise

$$\mathrm{OD}(n_i) = \bigwedge_{n_k \in fanout(n_i)} restrict(E_{i,k} + OD_k,\ n_i)$$

The algorithm to perform multi-level minimization is presented in figure 7.17.

```
function MLL_minimize(BN, dc): BOOL_NETWORK
var BN:        BOOL_NETWORK;
    dc:        DC_TYPE;          /* SDC, ODC or BOTH */
begin
  var m:           MBD_TYPE;
      nodelist:    list;

  if (dc ∈ {SDC,BOTH})
    then set_up_SDC(BN);         /* SDC(ni) = yi⊕fi */
  nodelist = order_nodes(BN);    /* minimization order */
  foreach node n in nodelist begin
    /* build the incompletely specified function of n */
    case dc:
      SDC: mon = n.mbd; mdc = n.sdc;
      ODC: mon = n.mbd; mdc = DO(n);
      BOTH: mon = n.mbd; mdc = n.sdc v DO(n);
    end case;
    n.mbd = MBD_minimize(m);     /* minimize the node function */
    update_BN(BN,n);             /* update fanin and SDC of n */
  end;
end;
```

Figure 7.17. Multi-level minimization algorithm.

The algorithm is a straightforward application of the concepts presented above. First the SDC from each node n is computed and stored in the field *n.sdc*. Then, the nodes are ordered

according to its fanin cardinality. The ordering in which the nodes are processed is important for the minimization purposes. We adopt here the heuristic of minimizing first the nodes with larger fanin, since they are the first to be discarded by the fanout support filter and are not used by the nodes with smaller fanin. Next, each node of the network is processed. There are three possibilities concerning the use don't cares: only SDC; only ODC or both of them. The incompletely specified node function is built by adding (in this case, *dc* always overrides other values) the don't cares to original node function. Then it is minimized using one of the techniques presented in chapter 5. Finally, the new SDC from *n* and its fanin are updated.

7.2.3 Experimental Results

We have applied the minimization algorithm over a set of benchmark examples. The results are shown in table 7.5. For each example, the circuit is read in the multi-level format and the MBD of each node function is built. The circuit is then minimized with respect to its satisfiability don't care set. Next, the circuit is minimized with respect to its observability don't care set. Since our goal is the synthesis on library based designs, the node functions were simplified using the two-level minimization techniques presented in chapter 5. To speed up the node minimization we have used the reordering techniques to reduce the MBD size before the node minimization.

	Initial				SDC				SDC + ODC			
Circuit	Size	Lit	MBD	Arc	Size	Lit	MBD	Arc	Size	Lit	MBD	Arc
Z4	14	66	100	44	11	58	71	34	10	56	67	29
DK27	10	37	56	31	10	30	50	29	10	30	50	29
P82	19	166	201	85	18	144	215	96	18	144	215	96
M2	19	185	225	147	19	163	223	139	19	163	223	139
X9DN	16	173	226	105	16	160	213	99	16	160	213	99
F51M	194	381	769	381	190	371	751	369	190	371	751	369
RD53	12	62	110	43	11	55	71	34	11	55	71	34
P82	19	166	201	85	18	144	215	96	18	144	215	96

Table 7.5. Multi-level minimization benchmarks.

We present four cost functions for each case:

- *size* is the number of nodes of the Boolean network
- *lit* is the total number of literals of node's covers
- *MBD* is the sum of the MBD size of the node functions
- *arc* is the number of interconnections in the network.

We may check that the total number of literals is effectively reduced, as expected. The number of nodes of the network, however, in several cases is not reduced. The MBD size and the number of connections usually are reduced too, but in some cases they are increased. This is a rather natural outcome of the type of minimization adopted, since one of the two-level minimization goals is the reduction of the number of literals. The MBD size is not important in this case. On the other hand, the increase in the number of connections indicates a possible augmentation of the routing costs.

Curiously, most of times the use of the ODC does not improve the solution. This does not means that it is useless, but probably that the examples taken were not particularly sensitive to ODC techniques. It should be considered also that the minimization was done with a subset of the ODC, which surely limits the efficiency of the method.

Another point that should be stressed is that even using the simplified models, the ODC and SDC may be still very large. Several examples could not be processed by the LISP prototype because the size of the MBD with the don't cares reaches sometimes a thousand nodes by subfunction. Even using the reordering techniques, it could take several hours for the prototype to minimize the circuit. The examples presented above were processed in a few minutes, at most.

The minimization complexity is not a direct function of the circuit size. It is rather a combination of factors. The fanin of each node plays an important role. Since we use the subset support filter, the larger the fanin of one node the larger is the number of nodes that contributes to its don't care. The worst case is when we have some nodes with a large fanin and several others with a small one. In this case, the don't care of the nodes with large fanin may be very large, specially because it will depend on too much variables. On the other hand, a large circuit composed by small fanin subfunctions may processed quite fast, because the subset support filter will limits the size of the node's SDC. It is the case of F51M, which is presented above to exemplify this point. It has 194 nodes, but all of then have at most 2 inputs. This is a extreme case, where the subset support filter is too restrictive. The improvements, by consequence, are not significant. These limitations can be reduced by providing the users with a set of operations to manipulate the Boolean network in order to obtain a topology that is more adequate to the use of those techniques. Some examples of such operations are node collapsing, node factorization and node decomposition. Another interesting possibility is to develop minimization techniques oriented to the reduction of the literals in the support of the function, which should reduce the connection cost and possibly eliminate more nodes from the network.

7.2.4 Comments

In this section we have touched the surface of a large research field that is multi-level logic minimization. There are several distinct methodologies to deal with this problem. All of them may be modeled as a minimization problem over the ODC and SDC of a network. They differ in the way they compute the subsets of the don't cares for the minimization.

In this context, we have studied the application of MBDs in this research field. There is several ways to apply MBDs here. Some works have already addressed the application of BDDs with the TRANSDUCTION method [Mat89]. Global flow, on the other hand, does not seems to be able to produce better results than the other minimization techniques described here and thus was discarded as a candidate for MBD application. Multi-level tautology based minimization and ATPG techniques could surely be improved by the use of MBDs, but they rely on an approach different than that adopted in this work. Thus, we have chosen to implement a method based on node function simplification. It consists in computing subsets of the ODC and SDC for each node of the network, which is then logic minimized. We have developed two methods for the minimization of the incompletely specified MBDs and both of them could be used with this technique. Since this chapter deals with the synthesis on library based designs, the experimental results presented here are related to the application of two-level MBD minimization algorithms. However, it will be not difficult to extend the technology mapping algorithm described in next section to work with FPGAs. In this case, the MBD size minimization techniques could be used to simplify node functions.

The algorithms for computing subsets of the don't cares were taken from the literature. They constitute by themselves an interesting and complex research field that was not treated here. The results show that this technique effectively reduces the complexity of some cost functions of a multi-level network, like the literal and node count. Finally, the efficacy of the algorithm may be improved by developing new algorithms to obtain better approximations of the don't care set, together with a set of operations for reshaping the network in order to guide the minimization process.

7.3 Technology Mapping

The logic decomposition and minimization of a multi-level circuit constitutes the technology independent phase of the synthesis. Once the logic independent optimization is done, the next step is the technology dependent phase of the design. It is usually called technology mapping, or simple mapping, for short.

The mapping consists in covering the network by implementing each node with a set of gates selected from the target library, while trying to meet the design constraints. Circuit delay, global area and testability are typical cost functions in digital circuit design. Ideally, the circuit

should have a minimum area and delay and to be 100% testable. In practice, however, a common goal is to minimize one cost function, area for example, and keep the others within the design constraints.

Technology mapping can be divided into two main tasks: gate matching and network covering. Gate matching consists in, given a Boolean function *f*, find all the possible gates configurations that can implement it. In the general case the matching may include inverters at the inputs and/or at the output of the gate. Network covering is the process of transforming the network into an acceptable design by selecting which set of gate matchings minimize the target cost function. Node collapsing and node decomposition are typical manipulations performed along the covering step. Several approaches have been reported in the literature. The mapping tools can be classified according to their matching and to the covering approaches. We may have:

- *structural* or *functional* (also called Boolean) matching
- *direct* or *recomposing* network covering

DAGON [Keu87], one of the first mappers, treats the matching as a graph isomorphism problem. It is based on the techniques of code minimization used in compilers. The circuit is decomposed into a set of primitive gates, as NANDs, NORs and inverters. The library cells are also decomposed with the same primitives. Gate matching is checked by comparing the tree representation of the gate and of the function to verify if they are isomorphic. We call it here *structural matching*, because we do not compare the functions but the structures used to represent them. A similar approach was adopted in MisII [Bra87]. The main difference is the introduction of duplicated inverters in each connection of the network in order to deal with the phase assignment problem. MisII have produced better results than DAGON, but both are limited due to the tree representation of the gates.

CERES [Mai88] is a mapper that uses Boolean matching to select the gates. Here functions and gates are compared using the Shannon decomposition. Herein we refer to it as *functional matching* because we compare Boolean functions instead of their structural descriptions. This approach is more flexible and allows to identify a wider range of functions including EXORs and majority functions, for example. However, the use of Shannon expansion to check a wide range of input orderings is costly. SKOLL [Ber88] introduces an interesting alternative to the gate matching problem. Gates and functions are described by sum-of-products (SOPs) represented by string of integers, where each integer is the number of literals of the respective cube. This allows a fast comparison between functions, with the restriction that repeated literals can not be represented and only sum-of-products can be used. DIRMAP [Leg88] proposes another interesting approach. The Boolean network is minimized with MisII and each node is directly mapped to a set of gates. It follows a greedy approach and is based on

the availability of autodual libraries, i.e., libraries in which each gate has its dual. It produces better results than the mapper of MisII with a simpler algorithm. The matching itself is not described, and so is difficult to estimate the restrictions of this method. However, when autodual libraries are available, the results in terms of inverter count are good due to the preservation of the global phase assignment information provided by the technology independent minimization.

There is a class of mappers based on local transformations [Gil84][Geu85][Lis88][Lis90]. [Gil84] and [Geu85] use a similar approach that consists in identifying gate configurations that may be replaced by equivalent ones optimized in some faction. They are limited by the local nature of their approach. McMap [Lis88][Lis90] does not look for pre-defined patterns but iterates over the circuit splitting and merging sets of gates trying to simplify the solution. To avoid local minimums it performs multiple trials with random selected starting points. It does not provide a comparison with existing tools, but the reported computing times indicate that the algorithm is very fast.

Other mapping systems that may be cited are GATEMAP [Pit89] and TECHMAP [Mor89]. GATEMAP is in fact a multi-level synthesis system that accepts a behavioral logic description as input and performs logic optimization and technology mapping. The mapping step is executed along the function decomposition using a fast Rademacher-Walsh transform to match subfunctions and gates as early as possible. A further optimization step is performed over the mapped circuit using techniques similar to [Geu85]. This process is quite costly in terms of CPU time, and the results obtained are similar to those of [Geu85]. TECHMAP realizes the mapping over a 2 input NANDs decomposed network. Some interesting features are the NAND decomposition oriented to delay optimization and the definition of a set of testability preserving circuit transformations.

Most algorithmic mappers perform *recomposing* covering. The optimized multi-level network is initially decomposed into elementary gates from the library (NORs, NANDs...). The covering consists in breaking down the network into a forest of trees and optimally mapping each tree using a dynamic programming technique [Aho83]. Nodes are collapsed in order to provide alternative matching possibilities. *Direct* covering consists in taking the optimized multi-level network as the starting point and trying to match directly the nodes against gates from the library. Nodes that can not be mapped are decomposed. DIRMAP [Leg88] and ASYL [Sak90] are examples of the direct covering approach. In [Leg88] it was shown that if we dispose of an autodual library the direct covering can be more effective due to the preservation of the global phase assignment optimization previous performed in the logic minimization step. In general, we can expect the direct covering to give better results since it does not destroys the optimized multi-level structure of the circuit. However, recomposing covering can eventually

produce better results by providing alternative starting points to the mapping problem. It can also be directly applied to networks of simple gates to re-map it to another technology.

In this section we discuss a technology mapping technique based on MBDs [Jac93a] that explores the advantages of both functional and structural matching and both direct and recomposing covering. Each node in the Boolean network has its associated function represented by a MBD. The gates in the library are also described by MBDs. The idea is to use the flexibility of the Boolean approach and improve the speed of the matching by the use of MBDs. Structural matching is used to deal with some special cases. The covering is basically a recomposing one but with a support for node decomposition in the case of direct covering.

7.3.1 Technologic Features

The target technology provides the gates that will be used in the mapping. There are some features of the technology that have a direct influence over the mapping method and over the cost functions. The library establish the functional resources that will be employed to cover the network, while the technology itself defines some parameters that must be considered in the selection process.

The cost of a given gate can vary from one technology to another. A good example is the difference between NMOS and CMOS NANDs and NORs. In NMOS, NORs are preferable in terms of speed and surface, while in CMOS the use of NANDs is more interesting. This kind of information shall be automatically taken into account by the cost function used to select the gates.

The library has a strong influence on the mapping method. The DIRMAP [Leg88] method, for example, needs an autodual library. The basic information that must be provided by the library is what kind of logic gates are available. We may enumerate three main library models:

- *topological libraries* which consist in the set of functions obtained with a matrix of transistors that can be connected to an input signal, deleted or replaced by short circuits.
- *programmable libraries* which contain one or a few programmable gates that can realize several distinct functions.
- *static libraries*, that provides the user with a set of pre-defined gates.

Programmable and static libraries are widely used, while topological ones are less common. SYLON-DREAM [Che90] is an example of a system that maps the Boolean network into topological libraries. It uses the synthesis with negative gates method [Iba71] to implement the node functions at the transistor level.

Cell generators and FPGAs are examples of programmable libraries. The mapping based on cell generators [Ber88a] relies on a different approach. As in the case of the Xilinx devices, the idea is to put as much as possible logic into a single cell, respecting the constraints imposed by the technology.

The libraries most used in practice are static ones. They offer a set of pre-defined primitives that implements the most commonly used logic functions. Among others, the elementary AND (NAND), OR (NOR) and NOT gates, simple sum of products and products of sums, and also some special functions like XNORs, XORs and majority functions. Standard cells and gate-arrays are the most popular examples of static libraries.

The mapping proposed here addresses static libraries. Its importance comes from the fact that it is the most used and it covers a large set design styles. Indeed, the first approach for the mapping into ACTEL devices was to describe all possible gates generated by the ACTEL programmable cell in a static library and then map the Boolean network using that library.

The first problem is how to describe the gates for the mapping tool. Each gate must contains relevant information for the mapping cost functions. In the LISP prototype of our system the gates were described by the following set of information.

Gate example:

- gate name - ex: AOI22
- INPUTS: a list of the gate inputs - ex: (INPUTS (A1 A2 B1 B2))
- OUTPUT: the name of the output - ex: Z
- SIZE: the area value of the gate - ex: 4
- DELAY: the maximum delay of the gate for a standard load - ex: 20 ns
- CONNECTIONS: the relation number of inputs/number connections of the gate ex: 4/6
- EQUATION: the expression that describes the gate logic function
- ex: (EQU (~ (+ (* A1 A2) (* B1 B2))))
- MBD: the MBD of the function.

The size, delay and connections can be used by cost functions in the mapping. We use a simplified delay model, where all inputs are supposed to have the same delay. The connections value express the number of wires that can be saved using a given gate. It is usually directly proportional to the gate area. The equation is an algebraic expression used for printing purposes. The MBD holds the gate function and is used in the matching process.

7.3.2 The Matching Problem

The matching of a function with a gate is one of the key problems in the mapping. Each gate can implement a family of functions, according to some transformations that can be applied to its inputs and to the output. We can point out the following set of transformations [Dav91]:

- S_n, the symmetric group of permutations of the n variables; it contains $n!$ elements.
- C_2^n, the group of complementation of variables containing 2^n elements.
- C, the complementation of the function; the group contains just two elements: the function itself and its complementation.

Let π be a permutation of the input variables and $\partial = \mathbf{x}^{(\mathbf{e})}$ a complementation of them. Let μ be the composition of both transformations: $\mu = \partial(\pi(\mathbf{X}))$. The matching problem can be stated as follow:

- given f, a Boolean function, and a library $L = \{G_i\}$, find a subset of gates $\Omega = \{\beta \in L\}$ such that:

$$f(\mathbf{X}) \equiv \beta(\mu\,(\mathbf{Y})) \text{ or } f(\mathbf{X}) \equiv \overline{\beta}\,(\mu\,(\mathbf{Y})) \qquad (1)$$

where $\equiv$ means a tautology.

The set Ω in fact contains functions that describe the logic behavior of a set of gates. The matching will establish a correspondence among variables of the gate function and variables of the function being matched. If Ω is empty, then there is no gate in the library that can implement function f.

Example: let $f = x_2 \vee \bar{x}_1$, $\mathbf{X} = \{ x_1, x_2 \}$, be the function to be matched against an AND gate described by $\beta = A_1 \cdot A_2$. Then, if $A1 = \bar{x}_1$, $A2 = x_2$, we have:

$$\overline{\beta}\,(\bar{x}_1, x_2) = f(x_1, x_2)$$

In this case we have $\pi = i$ (identity) and $\partial = \mathbf{x}^{(0,\,1)}$. The hardware associated consists in the AND gate with one inverter connected at the input pin A_1 and another connected at the output.

Definition 7.13. A *gate match* of a function f on a library L is described by:

- a logic gate $\beta \in L$
- a mapping π: $\mathbf{X} \rightarrow \mathbf{A}$, where $\mathbf{A}$ is the set of input pins of the gate.
- a set of input inverters, one for each 0 value in the phase vector $\mathbf{e}$ of $\partial = \mathbf{x}^{(\mathbf{e})}$
- an optional output inverter in the case where $\overline{f}\,(\mathbf{X}) \equiv \beta_i(\mu\,(\mathbf{Y}))$

Let f be the function associated to node y_i in the Boolean network. The gate match of f is called the *direct match* of y_i . The gate match of $\bar{f}$ is called the *complement match* of y_i .

The transformations can be illustrated as shown in figure 7.18. There are $2n!2^n$ possible transformations. Using Boolean matching we should apply each possible permutation of the input variables of the function to the gate pins and compare the result with the original function. In practice, however, the number of comparisons can be greatly reduced by considering the symmetry classes of a function.

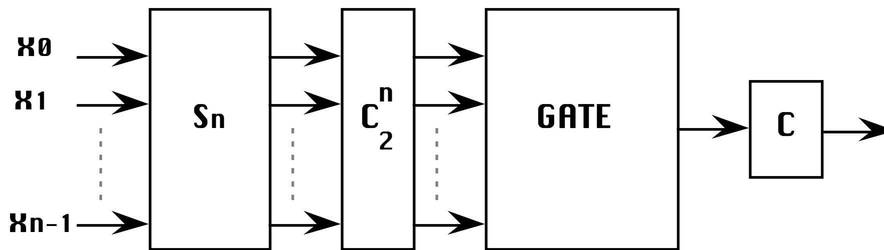


Figure 7.18. Input and output transformations

Definition 7.14: two variables $\{x_i,x_j\}$ are *symmetric* if:

$$f(x_1,..,x_i,..,x_j,..,x_n) \equiv f(x_1,..,x_j,..,x_i,..,x_n)$$

The symmetric property is transitive. If (x_i, x_j) and (x_j, x_k) are symmetric, then (x_i, x_k) are symmetric too. A *symmetric class* is a set of variables that are symmetric in a function. The resulting simplification in the matching comes from the fact that in a symmetry class the ordering of the variables is irrelevant and, thus, these variables need not to be permuted. For example, consider the function $f = (x_1 \vee x_2 \vee x_3)\cdot(x_4 \vee x_5)\cdot x_6$. The variables $\{x_1,x_2,x_3\}$ and $\{x_4 ,x_5\}$ form two symmetry classes, and they need not to be permuted among themselves in the matching. Although symmetric variables are not permuted, the symmetry classes of same cardinality of the matching functions must be permuted. If S_i is the number of symmetry classes with i variables, and k the cardinality of the largest symmetry class then the total number of input combinations to be checked is:

$$\prod_{i=1}^{k}(S_i\,!)$$

This is true for the general case. However, we can further prune the search space of the solutions to the matching problem if we extend the concept of symmetry to the symmetry classes themselves. This is a straightforward generalization of definition 7.14:

Definition 7.15. Let S be the set of symmetry classes of a function f. Let S_i and S_j be two symmetry classes of same cardinality. S_i and S_j are *symmetric* if:

$$f(S_0,..,S_i,..,S_j,..,S_{m-1}) \equiv f(S_0,..,S_j,..,S_i,..,S_{m-1})$$

An example of this property is a sum-of-products form with no repeated literals. The cubes define symmetry classes and they can be permuted without changing the associated function. As in the case of symmetric variables, the symmetry classes with this property do not need to be permuted, reducing the search space of possible matchings. In fact, this is a library dependent parameter. For all standard cells libraries we have experimented, this property holds. In this case, the matching reduces to function classification followed by tautology checking.

7.3.3 Matching with MBDs

The MBD structure, i.e., the MBD graph without node values and unlabelled arcs, represents a family of functions. This is exemplified in figure 7.19. The family of functions F denoted by the MBD is:

$$F = x^{(a)} \cdot (y^{(c)} \cdot t_0 \vee y^{(d)} \cdot t_1) \vee x^{(b)} \cdot t_1$$

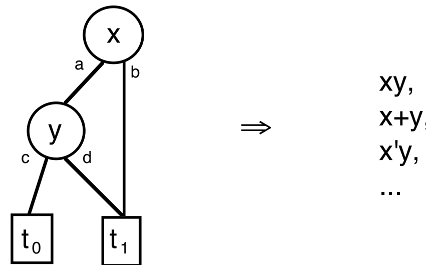


Figure 7.19. MBD structure and its family of functions

Choosing values for the labels $\{ a, b, c, d \}$ and the terminals $\{t_0, t_1\}$, we can derive the family of functions F = $\{ x \cdot y, x \vee y, \bar{x} \vee y, ... \}$. It is not the set of all functions of two variables, but is the set of functions that can be obtained from an AND (or OR) gate by introducing inverters on each input and at the output and also by permuting the inputs.

Comparing the MBD of figure 7.19 with figure 7.18, we can establish the following correspondences:

- the MBD structure represents the gate
- the set S_n stands for the different input variable orderings on the MBD
- the set C corresponds to the two different terminal node value assignments
- the set C_2^n is associated to the arc labels.

The matching of a function and a gate represented by their MBDs can be executed by first finding two isomorphic MBD structures and then extracting from them the variables and output

phases. The isomorphism between the graphs can be found by classifying the MBDs according their symmetry classes. Let MBD_f be the MBD of the node function to be matched. Then the MBD matching can be performed as follows:

1. identify the symmetry classes (*SymCs*) of the MBD_f
2. sort the classes in the descending order of cardinality
3. match the function MBD against the gates that have the same number of *SymCs* and the same *SymCs* cardinality.

This algorithm is based on the fact that two functions *f* and *g* can match only if they have the same number of symmetry classes and if for each symmetry class of *f* there is a corresponding symmetry class in *g* with the same cardinality. For example:

$$f = (x_1 \vee x_2 \vee x_3) \cdot (x_4 \vee x_5 \vee x_6)$$
$$g = (x_1 \vee x_2 \vee x_3 \vee x_4) \cdot (x_5 \vee x_6)$$

have the same number of *SymCs* (2), but they have different cardinality: (3,3) and (4,2) respectively, and thus do not match.

If a gate has *SymCs* that are not symmetric then they must be permuted during the matching. For the libraries where the *SymCs* are symmetric, the three steps above are enough to verify the matching. In particular, step 3 is reduced to a Boolean verification between the functions for all possible phase assignments of input variables.

7.3.4 MBD classification

To find the *SymCs* of a MBD, we re-interpret definition 7.14 with respect to a MBD.

Definition 7.16. Two variables $\{x_i, x_j\}$ are *symmetric* in a MBD if they can be exchanged in the MBD ordering without modifying the topology of the graph.

This can be easily checked. The MBD is a canonical description of a Boolean function. If we change the function, the MBD must be changed accordingly. Thus, if we build a new MBD with two variables exchanged in the ordering and the new graph is isomorphic to previous one, then the function denoted by the MBD is the same for both orderings and, according to definition 7.14, the variables are symmetric. Figure 7.20 gives an example of symmetry on a MBD. Variables x_1 and x_3 are symmetric in the function denoted by the MBD. We can see that when they are exchanged the MBD remains exactly the same.

As we are considering the possibility of any combination of input variables complementation, we extend the symmetry property to deal with all possible phase assignments.

Definition 7.17: Two variables x_i and x_j are *phase free symmetric* if they can be made symmetric by an adequate input phase assignment. *Phase free symmetry classes* are defined in the same way.

For example, let $f = \bar{x}_1 \cdot x_2$. Since $\bar{x}_1 \cdot x_2. \neq x_1 \cdot \bar{x}_2$, x_1 and x_2 are not symmetric. But they are *phase free* symmetric because if we complement either x_1 or x_2 (but not both) then the symmetry property holds. This is equivalent to say that if $\bar{x}_1$ is replaced by a new function y such that y = $\bar{x}_1$, then we have $f = y \cdot x_2$, and x_2 and y are symmetric.

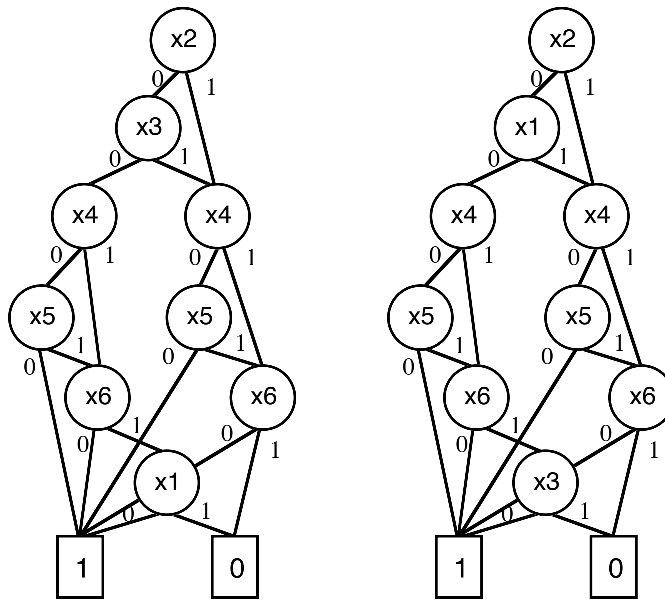


Figure 7.20. Two symmetric variables on a MBD.

Therefore, a simple method for detecting if two variables are symmetric is to build two MBDs for the two respective orderings and check if they are isomorphic. If we suppose we have the original MBD already computed, then the cost of this operation is proportional to the cost to build a new MBD plus the cost of the equivalence checking. It is better than the method proposed in [Mai88], which uses Shannon decomposition to perform tautology checking. A still more efficient method is to apply the incremental techniques in order to avoid the cost of building a new MBD. If we exchange two adjacent variables we can easily check if the MBD was modified or not. In fact, we can check it even without exchanging the variables. The idea is expressed in the following theorem.

Theorem 7.1. Let S be the set of subgraphs of nodes associated to two adjacent variables in the MBD ordering (x_i, x_{i+1}) which have at least 2 nodes. Each subgraph has one root and can lead to 2, 3 or 4 different subfunctions in the MBD. x_i and x_{i+1} are *phase free* symmetric either if all s in S lead to only 2 subfunctions in the MBD or if they lead to three

subfunctions: two subfunctions are associated to paths $\bar{x}_i\,\bar{x}_j$ and $x_i\,x_j$ in the graph and the third one is associated to paths $\bar{x}_i x_j$ and $x_i\,\bar{x}_j$.

Proof: two Boolean variables x_i and x_j divide the Boolean space into four subspaces, respectively: (1) $\bar{x}_i\,\bar{x}_j$, (2) $\bar{x}_i x_j$, (3) $x_i\,\bar{x}_j$ and (4) $x_i\,x_j$. Each subspace corresponds to a path in the subgraphs s ∈ S. Exchanging x_i and x_j the subspaces (1) and (4) remains unchanged, while subspaces (2) and (3) are exchanged. To be symmetric on these variables, the subfunctions at subspaces (2) and (3) must be equivalent. This condition is represented here as (2) ≡ (3). In this case we have at most three different subfunctions associated to subspaces (1), (2), (3) and (4): one for (1), another for (4) and a third one for (2) and (3). If either (1) ≡ (3) or (1) ≡ (2) ≡ (3) or (2) ≡ (3) ≡ (4) there are only two different subfunctions and the paths in each s ∈ S lead to only two different nodes (each node is associated to a subfunction). If we have only (2) ≡ (3) then the paths in s ∈ S lead to three different nodes. ◊

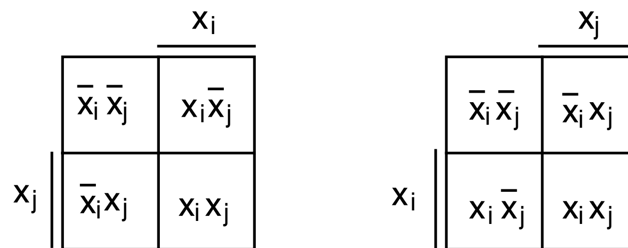


Figure 7.21. Effect of exchanging variables.

This proof can be visualized with the help of Karnaugh like diagram, as in figure 7.21. Two variables divide the function domain in four subspaces or subfunctions. If the variables are exchanged, then the subspaces where $x_i \neq x_j$ are also exchanged in the table. If x_i and x_j are symmetric, then $f(\ldots,x_i,x_j,\ldots) = f(\ldots,x_j,x_i,\ldots)$, and both diagrams must be symmetric with respect to the diagonal. This implies that subspaces $\bar{x}_i x_j$ and $x_i\,\bar{x}_j$ must be equivalents. Figure 7.22 show some examples of symmetric variables in two variable functions.

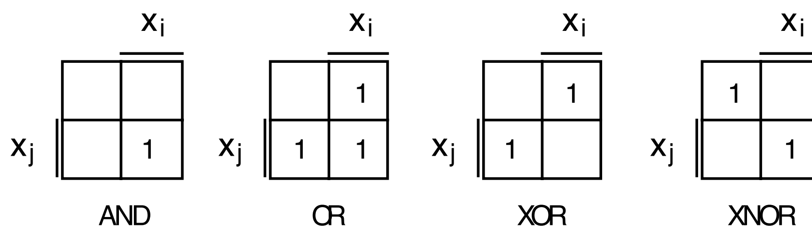


Figure 7.22. Symmetric variables examples.

Thus, we can detect the symmetry property between two adjacent variables in a MBD just by inspection. The subgraph topologies that correspond to symmetric and adjacent variables in a

MBD are presented in figure 7.23. The subgraph (a) corresponds to the case where three of the four subfunctions are equivalent. A simple example is the OR function of two variables. Subspaces $\bar{x}_i x_j$, $x_i \bar{x}_j$ and $x_i x_j$ are associated to function **1** (g in the subgraph) while the subspace $\bar{x}_i \bar{x}_j$ is associated to function **0** (f in the subgraph). This can be indicated in (a) by labeling edges that lead to subfunction g with value 1 and the edges in series that lead to f with value 0. Subgraph of figure 7.23(b) represents the case where $\bar{x}_i x_j \equiv x_i \bar{x}_j$ and $x_i x_j \equiv \bar{x}_i \bar{x}_j$. Subgraph 7.23(c) is the general case where we have only $\bar{x}_i x_j \equiv x_i \bar{x}_j$ (g in the subgraph) and $x_i x_j$ and $\bar{x}_i \bar{x}_j$ are different.

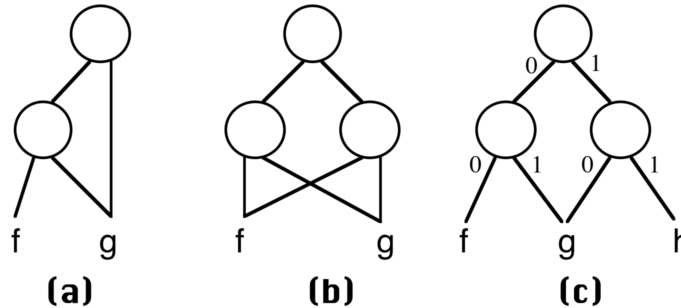


Figure 7.23. Some adjacent variables subgraphs.

Note that when we have a set of subgraphs in the adjacent levels, all of them must be of the same type ((a), (b) or (c) in figure 7.23). This can be visualized in figure 7.20. The MBD represents the function $f = \overline{(x_1 \vee x_2 \vee x_3)(x_4 \vee x_5)x_6}$. Variables x_4 and x_5 are known to be symmetric. In the MBD, we can see that the two subgraphs formed by these variables correspond to the symmetry configuration of figure 7.23(a).

The algorithm to find the phase free symmetry classes in a MBD is based on the incremental techniques presented in chapter 4. The idea is to make a variable *walk* through the MBD and *meet* all the other variables. The algorithm starts with the root variable v_r and test it for phase free symmetry against its right adjacent variable in the ordering (variable with the next index in the MBD). Then, v_r is swapped with its adjacent variable in the ordering. Next, it is compared against its new right adjacent variable in the ordering, and so on. The number of steps is upper bound by $n(n-1)/2$. Usually there are less comparisons due to the transitive property of symmetric variables: if one variable belongs to a *SymCs* it does not need to be further compared. After all variables are pairwise tested, the MBD is classified by putting the *SymCs* with larger cardinality closer to the root. The process is sketched in figure 7.24.

Let us illustrate the process with an example. Consider again the function of figure 7.20, $f = \overline{(x_1 \vee x_2 \vee x_3)(x_4 \vee x_5)x_6}$. We have the *SymCs* $S_0 = \{x_1, x_2, x_3\}$, $S_1 = \{x_4, x_5\}$ and $S_2 = \{x_6\}$. The literal x_6 is considered as a single element *SymC*. In figure 7.25 we show f with an arbitrary ordering. MBDs from (a) to (f) show the first step in the symmetry evaluation

of variable x_4, the original root. In all cases but (d), x_4 and its upper neighbor are not symmetric. In (d) the subgraphs defined by variables x_4 and x_5 follow the configuration of figure 7.23(a). Finally, figure 7.25(g) shows the final classified MBD, with the three *SymCs* sorted in decreasing cardinality.

```
function class_MBD (mbd, order)
begin
  /* find the symmetry classes */
  foreach variable v in order
    if v is not member of any symmetry class already computed
    then  begin
      compare v against all remaining variables
      put all variables symmetric to v in a new symmetry class
      push the SymCs found in the set of symmetry classes S
    end;
  /* sort the symmetry classes */
  sort S in descending order of cardinality;
  /* reorder the MBD */
  foreach symmetry class sc in S
    foreach variable v in sc
      swap the variable in the MBD from its current position up
        to its position in S;
      update the order;
  return (mbd, order)
end;
```

Figure 7.24. Algorithm for MBD classification.

The gates of the library are classified automatically when the library is defined by the user, using the same approach. The goal is to reduce the number of candidate gates for the matching. The gates are classified first by the number of *SymCs* and after by a vector resulting from the juxtaposition of the *SymCs* cardinality (symmetry vector). The gate AOI22 is therefore stored with the set of gates represented by the vector 22, which belongs to the subset of gates that have 2 symmetry classes. The gate AOI322 is associated, respectively, to vector 322 in the subset of gates with 3 symmetry classes, and so on. Thus, when the function to be mapped is classified, its symmetry vector is extracted and used to select a subset of candidate gates from the library. The gate indexing scheme in the library is organized as depicts figure 7.26.

7.3.5 MBD matching

MBD matching is the task of verifying if two MBDs can be made equivalent by an adequate variable phase and function phase assignments. The matching algorithm is similar to that for equivalence checking between MBDs, but modified to deal with the phase assignment.

Definition 7.18: Two MBDs M_i and M_j can be matched if for each node $v_i \in M_i$ there is a correspondent node $v_j \in M_j$ such that:

Figure 7.25. MBD classification example.

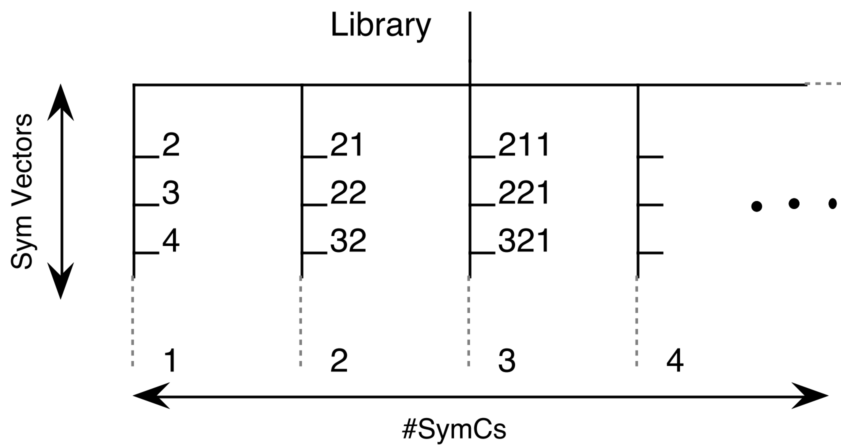


Figure 7.26. Library organization.

(a) either they are the same terminal and the output phase is direct or they are different terminals and the output phase is complemented;

(b) they have the same index and

- low(v_i) can be matched to low(v_j) *and* high(v_i) can be matched to high(v_j) and the variable phase is direct
- low(v_i) can be matched to high(v_j) *and* high(v_i) can be matched to low(v_j) and the variable phase is complemented.

The main difference with respect to equivalence checking is that the phase of the variables are determined during the matching. If the low and high sons of two nodes of two different MBDs can not be matched, the low/high and high/low matching is then verified. This is equivalent to say that one variable was complemented (exchange low/high pointers). This point is illustrated in figure 7.27. Function $f = \bar{x}_1 + x_2$ is matched against gate NOR2. The root nodes x_1 and A_1 can not equivalent, because the low son of x_1 leads to a terminal node and the low son of A_1 is connected to a non-terminal node. However, if the variable associated to node x_1 is complemented, then the low son of $\bar{x}_1$ is connected to a non-terminal node, as do A_1. Note that, if we have several nodes at the same level of the graph (nodes with same index), all of them shall be complemented, i.e., their low and high pointer must be exchanged.

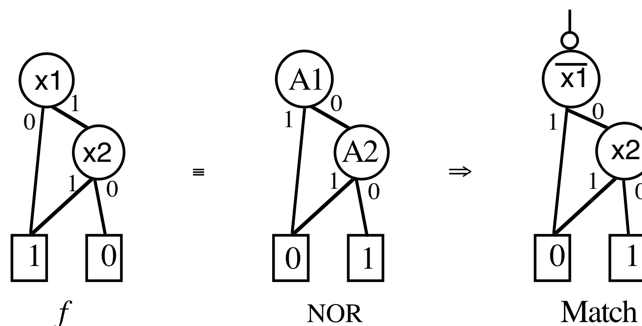


Figure 7.27. Match with phase assignment

The high sons of $\bar{x}_1$ and A_1 are now connected to terminals that have opposite values. Since the output of f was not yet inverted, we invert it. Note that if f was already inverted then the matching fails. Inverting a function represented by a MBD results in the same MBD with the terminal values **1** and **0** exchanged. Thus we have that high($\bar{x}_1$) = high(A_1) = **1**. Proceeding with the matching, we check x_2 against A_2. In this case, they are equivalent and the resulting match indicates that $\bar{f}(\bar{x}_1, x_2)$ matches NOR(A_1,A_2). Thus, we have $A_1 = \bar{x}_1$ and $A_2 = x_2$ and $f = \overline{\text{NOR}}(\bar{x}_1, x_2)$, which indicates that f can be realized by a NOR gate with one inverter connected to input A_1 and another one connected to the gate output.

MBDs can be used to perform both *structural* and *functional matching* . To use the MBD for *structural matching* we must assign to each node input a different index. In this case, even if we have repeated input variables, the MBD will reflect the gate structure. If we do not impose

different indices to the inputs, then the MBD is a Boolean function and we perform *functional matching*. Figure 7.28 shows these two approaches for a XOR gate. In the graph 7.28 (a) we have assigned to four *node inputs* $\bar{x}_1, x_2, x_1$ and $\bar{x}_2$ four different indices. In this case the MBD reflects the sum-of-products form $\bar{x}_1 \cdot x_2 \vee x_1 \cdot \bar{x}_2$. One can verify that the subgraph defined by the nodes inside n_1 corresponds to the first cube and the subgraph with nodes inside n_2 corresponds to the second cube in the sum-of-products form. If we consider n_1 and n_2 as two nodes and define a new MBD with low(n_1) = n_2, high(n_1) = **1**, low(n_2) = **0** and high(n_2) = **1**, we get MBD = $n_1 + n_2$, which correspond to the SOP of the function. In figure 7.28(b) each *variable* has an unique index, and the MBD represents the XOR *function*. These alternative approaches are useful because we can explore the fact that a XOR can be implemented with an AOI22 gate, which is sometimes cheaper than a XOR gate, specially if the inputs are available in both phases.

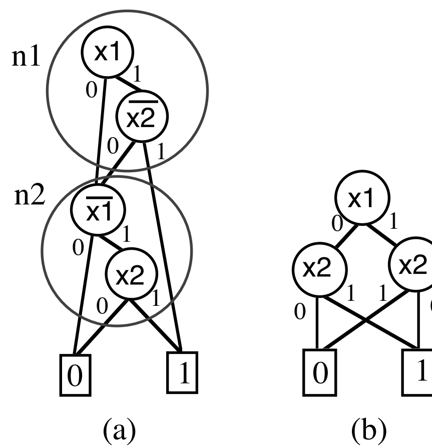


Figure 7.28. Structural (a) and functional (b) representations.

7.3.6 Matching Algorithms

The matching algorithms are sketched in figure 7.29. Function *mach_MBD* takes a MBD and its order and return the set of matches that can implement the function. Each match is formed by the gate and the set of inputs/output inverters generated by the matching algorithm. The function is first classified according to its symmetry classes. Then the *SymCs* cardinality vector is used to address the set of gates in the library that are candidates to match the function. The function *match_gate* takes the MBD of the function to be mapped and the MBD of the gate and return NULL if they can not be matched, otherwise it returns the input assignment with the respective phases and the output phase.

Function *rec_match_gate* does the matching by traversing both MBDs and checking for direct or complemented match at each node. *bddf* and *bddg* are the current nodes being matched. The algorithm computes the phase of each variable in the matching and store it in the array *phv*[]. The phase is positive if the low sons and the high sons can be matched between themselves, and it is negative if the low son of each node matches the high son of the other. It is not

shown in the algorithm, but if the same variable matches with different polarities at different nodes in the MBDs, then the match is unsuccessful. The phase of the function is computed also by *rec_match_gate*. If the match requires that the MBD terminals must be inverted, an inverter is added to the output of the gate. This is symbolically expressed by the statement *invert the output* in the algorithm.

```
function match_BDD (mbd, order): matchlist;
begin
   (mbd,order) := class_BDD (mbd, order);
   vector := get_SCs_vector(mbd);
   G := get_gates(vector);
   foreach gate in G
      if (match = match_gate(mbd, G.mbd, order))
      then
         { match contains the gate and inverters needed }
         put match in matchlist;
   return (matchlist);
end;

function match_gate (bddf, bddg, order): match;
begin
   initialize phase_vector;
   if (rec_match_gate (bddf, bddg, phase_vector))
      then return (build_match ( order, phase_vector))
      else return (NULL);
end;

function rec_match_gate (bddf, bddg, phv)
begin
   case (bddf,bddg) begin
      (they have different indices): return(false);
      (both are terminals):
         if terminals are distinct then
            if (phv[output] = NEGATED) then return(false);
            else  begin
               phv[output] := NEGATED;
               return(true);
            end;
      (only one is a terminal): return (false);
      ((rec_match_gate (lowv (bddf), lowv (bddg), phv) and
         rec_match_gate (highv (bddf), highv (bddg) phv)):
         return(true);
      ((rec_match_gate (lowv (bddf), highv (bddg), phv) and
         rec_match_gate (highv (bddf), lowv (bddg) phv)):
         phv[index(bddf)] := NEGATED;
         return(true);
   end;
end;
```

Figure 7.29. MBD matching algorithms.

7.3.7 Network Covering

The technology independent synthesis uses sophisticated algorithms to produce a logic optimized Boolean network. There are two methods to cover the network with gates from a library: (1) The recomposing covering further decomposes the optimized network in terms of

elementary two input functions (typically: NANDs and NORs). This yields a *fine granularity* network which is composed by simple node functions that are collapsed in order to form larger functions to match complex gates in the library, which reduces the total gate count; (2) The direct covering consists in trying to directly map the logic optimized network into the target technology. The idea is to avoid the lost of the optimized structure produced in the previous step. In this case, when a node function is too complex to be implemented by a single gate in the library then it is decomposed into simpler functions.

The approach adopted here is a combination of both direct and recomposing covering. First we try to map the node functions directly to gates of the library. If the mapping is successful, we still try new mappings by generating new functions that are obtained by collapsing intermediate variables in the support of the node function. This approach allows the algorithm to work properly even if the input circuit is described by elementary gates. Nodes that can not be directly mapped are decomposed into a two input AND/OR tree. To decompose the MBD of a node we generate first a prime and irredundant sum-of-products (SOP) representation of the node function using the techniques shown in chapter 5. The decomposition itself is trivial. The goal is to produce a balanced tree in order to reduce the number of levels of the decomposed node. The covering algorithm then proceeds over to the decomposed function.

The covering algorithm is similar to that presented in the FPGA mapping, based on dynamic programming. Each network output and each multiple fanout node defines a root of a tree like subnetwork. The leaves of the tree are the primary input nodes or multiple fanout nodes. It is mapped optimally by computing the cost of all of its subtrees. As in the case of FPGA mapping, this approach produces non-optimal solutions at the subtrees boundaries. There are two main limitations: (1) some simple gates with multiple fanout are not collapsed even when it is possible to do it and (2) the phase of a mapped function g is optimized only with respect to the subnetwork composed by the nodes in the transitive fanin of g. Case (1) is treated by a pos-processing step that check multiple fanout nodes for collapsing, while case (2) is tackled by a global phase optimization algorithm that is described in the next section.

In the sequel we shall discuss in more detail the covering algorithm. Its pseudo-code is shown in figure 7.30. The function *map_node* computes the match for a node in both phases, direct and complemented. It returns the match of the required phase. The reason for computing both matches is that the cost of a gate match is evaluated by computing recursively the cost of its input nodes matches. When a gate match includes inverters at the inputs, the cost of the inverter is taken into account by computing the cost of the complement match of the input. For example, suppose the function $\bar{y}_1 \cdot y_2$ is matched by an AND gate plus an INVERTER at input y_1. The cost of such match is not "cost(AND) + cost(INVERTER) + the cost of the matchings of y_1 and y_2", but "cost(AND) + the cost of the matchings of $\bar{y}_1$ and y_2".

The first step in *map_node* is to check if the node was already matched. In this case, the match with the desired phase is returned. Otherwise, the matching proceeds by the determination of the cluster functions of the node. The clusters are all the functions obtained by recursively collapsing the single output fanins of the function associated to the node up to some technological constraint, typically the number of inputs.

```
function map_node (n, phase) : MATCH_TYPE
var:
  d_map, c_map: MATCH_TYPE;
  clusters:     set of  MBD_TYPE;
begin
  if (matched(n)) then
    case (phase) begin
      ON: return(n.dir_map);
      OFF: return(n.cmpl_map);
    end;
  clusters := make_clusters(n);
  if (!map_clusters(clusters, d_map, c_map)
    then return(map_node(dcmp_node(n),phase);
  n.dir_map := d_map;
  n.cmpl_map := c_map;
  case phase begin
    ON: return(n.dir_map);
    OFF: return(n.cmpl_map);
  end;
end;
```

Figure 7.30. Covering algorithm.

For example, consider the function f given by:

$$f = y_1 \cdot y_2$$
$$y_1 = x_1 \vee x_2$$
$$y_2 = y_3 \cdot x_3$$
$$y_3 = x_4 \cdot x_5 \vee x_6$$

Its set K of all clusters of f is given by:

$$k_0 = y_1 \cdot y_2$$
$$k_1 = (x_1 \vee x_2) \cdot y_2$$
$$k_2 = y_1 \cdot y_3 \cdot x_3$$
$$k_3 = (x_1 \vee x_2) y_3 \cdot x_3$$
$$k_4 = y_1 . (x_4 \cdot x_5 \vee x_6) \cdot x_3$$
$$k_5 = (x_1 \vee x_2) \cdot (x_4 \cdot x_5 \vee x_6) \cdot x_3$$

If the target technology has a fanin limit of 4, for example, then only clusters k_0, k_1, k_2 and k_4 will be generated. If node corresponding to variable y_2 has multiple fanout, then only clusters k_0 and k_1 will be generated for function f. The clusters here are shown as algebraic expression just for easy of understanding. In fact each cluster is a MBD obtained by the *compose* [Bry86]

operation. This is illustrated in figure 7.31, where the MBD of k_1 is derived from the MBDs of k_0 and y_1.

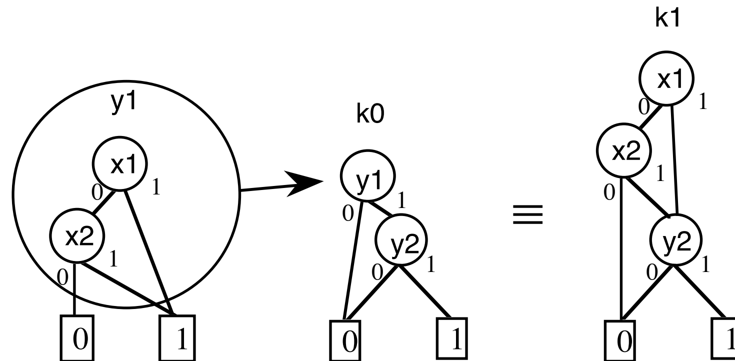


Figure 7.31. Composing y_1 and k_0 to obtain k_1.

Function *map_clusters* takes each cluster and matches it to the gates in the library. It evaluates the matchings by recursively calling *map_node* for each cluster input. For the gate inputs with inverters assigned to the algorithm calls *map_node* with the *phase* parameter set to OFF, which means that it looks for the complemented input function. Primary inputs have a special treatment. The direct mapping of a primary input has no cost, while its complement has the cost of an inverter. *Map_clusters* returns both the direct and the complement matchings of the cluster that has the smaller cost. If the library has only positive gates this means that an inverter is added to the best match output. But in almost all the cases the library provides several alternatives and the gate match produces a set of positive and negative gates with inverters at the inputs and at the output. In this case, the best complement matching is easily found by eliminating an inverter from the output of a gate match.

If *map_clusters* fails to find a match for the clusters then the node function is decomposed into an AND/OR tree of two input functions. The decomposition starts with the generation of a MBD cover of the node function, as described in chapter 5. The decomposition is very simple:

1. create a balanced tree of two input OR node functions with size(MBD_cover) leaves
2. to each leaf associates a MBD cube. Put the cubes with more literals in the leaves that are nearer to the root
3. for each cube create a balanced tree of two input AND node functions.

A simple example is shown in figure 7.32. The sum of products (SOP) consists in three cubes, represented in (a) by the conjunction of their literals. In (b) it is shown the OR tree decomposition. There are three leaves, two with depth 2 and one with depth 1. We choose the leaf with depth 1 to put the cube c_0, which have the larger number of literals. The others are associated to the remaining leaves. Then they are decomposed into two input AND trees. The cubes with larger number of literals will produce deeper AND trees. Putting them in the OR

leaves with smaller depth tend to produce a balanced tree, in which all the signals tend to have the same delay. After decomposing the node into a AND/OR tree, the mapping continues at the root of the tree.

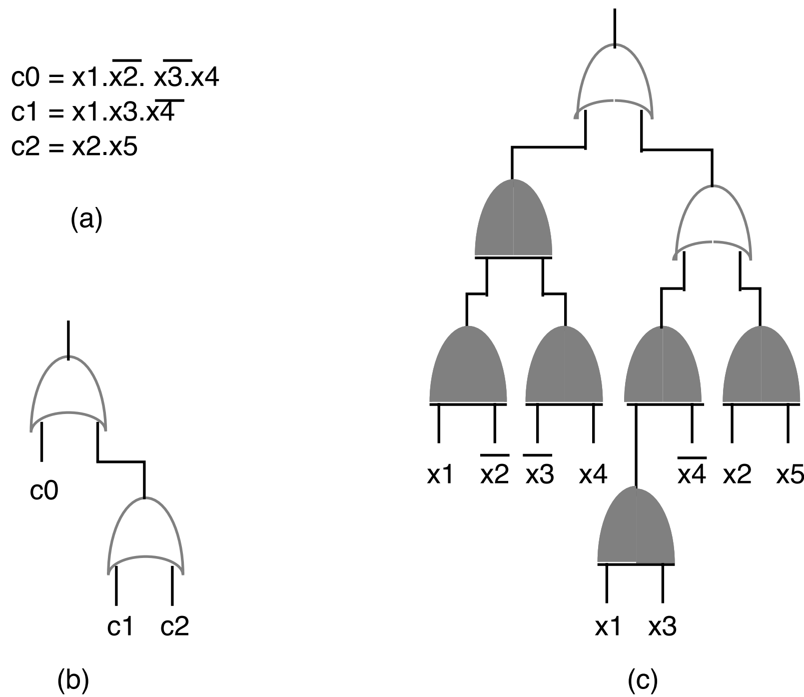


Figure 7.32. Node decomposition with (a) SOP, (b) OR tree and (c) AND/OR tree

After the first step of the network covering the nodes have their direct and complemented matches defined. Some original nodes are collapsed and we have a different network. A second step is then performed in which the final phase of each node is computed. For single fanout nodes, the phase is defined by its fanout node's match. If the fanout gate require a negated input, the complement of the match of the node is selected, otherwise the direct match is chosen. For multiple fanout nodes there are two possibilities: either all fanout gates use the same node phase and then we have the same case of a single fanout node or the fanout gates need both phases. In this case, the cheaper match of the multiple fanout node is chosen and a new inverting node is created to provide the complemented phase. The process starts at the outputs that have no fanin and proceeds up to the network inputs. We use the term *fanout gate* here because the match of the fanout of a node is already selected.

7.3.8 Phase Optimization

The dynamic technique guarantees the optimality of a tree mapping. However, at the tree boundaries some redundant inverters can arise. If we use the gate size as cost function, sometimes an NAND-NOT combination can be chosen instead of a single AND gate, if they have the same cost. These redundancies can be detected and removed at the phase optimization step. The gate configurations are analyzed and if inverters can be saved the gate is complemented or

transformed by applying De Morgan's rules (see chapter 2). As the algorithm works directly on the mapped network, the complementation of a gate requires the presence of the dual or the complement of the gate in the library. Some logic synthesis systems like MisII [Bra87] make the phase optimization on the logic network, without checking the specific gates form the library. Our approach has the advantage that only feasible node complementation are executed. An example of the application of the De Morgan's rule that saves two inverters is presented in figure 7.33.

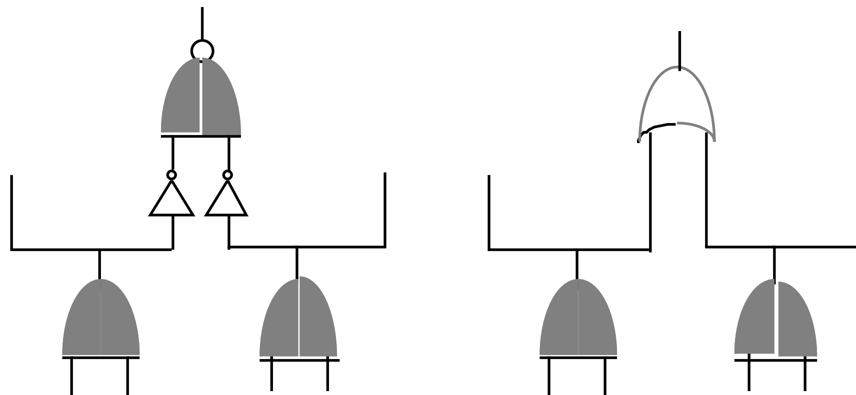


Figure 7.33. Example of the application of De Morgan's rule.

Two approaches were implemented. The first is a greedy one. All gates of the network are analyzed and the gate whose complementation reduces most the inverter count is selected and complemented. The process is repeated until no more gain is obtained. The problem with this method is that it stops at a local minimum. To circumvent this limitation, an stochastic method is used. An initial phase assignment for the network is randomly generated and the greedy method is then applied. A set of *p* trials, where *p* is a user supplied parameter, is executed and the best result is selected. An sketch of the pseudo-code for these algorithms is presented in figure 7.34.

7.3.9 Experimental Results

A set of examples of the MCNC benchmark was tested and the results were compared to MisII, which is a standard point of reference for multi-level synthesis. Both mappers were tuned for area minimization. For the LISP prototype of our mapper the run times are not comparable. Thus only the results are meaningful. To compare only the technology mapping phase we provide the same Boolean networks for both mappers. The circuits were minimized with MisII using the algebraic script supplied with the Octtools package [Oct91]. Two kind of inputs were generated to our mapper: (1) the minimized network and (2) the minimized network decomposed into 2 input ANDs. We used the MisII function *tech_decompose* to generate (2). The library used is the MCNC. We reproduce here the script.algebraic file of MisII that indicates the logic decomposition/minimization executed on the original descriptions.

Algebraic script:

sweep

eliminate 5

simplify -m nocomp -d

resub -a

gkx -abt 30

resub -a; sweep

gcx -bt 30

resub -a; sweep

gkx -abt 10

resub -a; sweep

gcx -bt 10

resub -a; sweep

gkx -ab

resub -a; sweep

gcx -b

resub -a; sweep

eliminate 0

decomp -g *

```
function greedy_phase (network):
begin
  repeat
    select gate g ∈ network that reduces most the inverter count
    invert g
  until no more gain
end;

function stochastic_phase (network, p):
begin
  greedy_phase (network);
  repeat p times
    invs := get_inv_count (network);
    repeat invs times
      invert a random selected gate g ∈ network;
    greedy_phase(network);
  return the best result;
end;
```

Figure 7.34. Pseudo codes for phase minimization.

	MisII			Direct SP			Direct GP			AND2 SP			AND2 GP		
Circuits	A	G	I	A	G	I	A	G	I	A	G	I	A	G	I
Z4	51	17	4	54	15	3	54	16	4	53	18	3	53	18	6
Z9SYM	113	47	13	116	43	9	111	45	11	114	46	11	113	47	12
Z5XP1	152	61	12	154	60	11	153	62	16	155	61	15	153	62	16
SQN	167	62	8	172	62	10	172	65	13	168	63	10	168	63	10
RISC	127	64	21	123	61	19	123	61	19	129	60	18	124	61	19
RD53	67	27	9	66	22	4	67	23	5	66	25	7	67	26	8
RD73	134	55	13	137	53	15	137	54	16	132	50	13	134	53	16
P82	145	59	12	145	58	12	144	60	14	145	58	12	144	59	12
M1	88	38	9	86	36	9	86	37	10	89	37	10	88	38	11
M2	150	66	20	148	60	17	145	61	18	145	62	18	145	62	18
O2	36	16	4	40	12	0	36	16	4	36	16	4	36	16	4
F51M	160	63	11	162	62	15	159	64	17	159	62	15	159	64	17
DEKODER	44	18	5	43	17	4	43	17	4	43	17	4	43	17	4
BW	267	112	23	267	110	21	263	111	22	261	109	22	262	110	23
ALU3	134	56	12	133	54	13	133	54	13	133	52	11	133	52	11
5XP1	158	64	13	163	62	13	159	64	15	158	61	12	159	64	14
Total:	1993	825	189	2009	788	175	1985	810	201	1986	797	185	1981	812	201

Table 7.6. Results for different mappings and the MisII mapper.

Table 7.6 shows the results for direct mapping and for the AND2 decomposed circuit mapping. GP means *greedy_phase* and SP means *stochastic_phase*. **A** is the global area, **G** the total gate count and **I** the inverter count. An interesting result is that direct map with SP produces in all cases the smaller gate count. This indicates a good use of complex gates and a possible reduction of interconnections complexity. *Direct_SP*, however, do not always produces network with global area smaller than *direct_GP*, as it could be expected. This occurs because the cost function used for these algorithms is the inverter count. Thus, if transforming two NANDs into two ANDs saves one inverter, this is done even if the global area is increased (typically: cost(AND) = cost(NAND) + cost(NOT)). In average, the reduction of the number of gates in a network by collapsing NOTs shall produce a final gain in the layout area due to the connection costs. Another interesting observation is that the direct mapping does not improve the average area with respect to AND2 decomposed mapping. This indicates that the decomposition performed by MisII before the mapping do not seems to have an strong impact on the mapping results of our method in terms of global area. Gate count, however, is affected by the decomposition.

7.3.10 Comments

In this section we studied the application of MBDs in the technology mapping problem. It was shown that MBDs can be very interesting to implement Boolean matching techniques. A new and fast technique to identify symmetric variables based on the MBD topology was developed. It is used to classify the functions with respect to its symmetry classes, which is an important task in the Boolean matching problem. Once the functions are classified, the matching is executed in a one step MBD traverse. An interesting feature of the matching algorithm is that it computes at the same time the phase assignment of the function variables and output.

The network covering step is a generalization of two main techniques, the direct covering and the recomposing one. This is intended to provide more flexibility in the application of the algorithm. For instance, as the Boolean mapping is speed up by fast Boolean matching techniques, it is possible to try both approaches, one mapping starting with the optimized multi-level circuit and another one starting with a fine granularity decomposition.

The covering uses the dynamic programming techniques to optimally map each tree of the circuit. As in the case of FPGA mapping (although in a minor extension), it may produce non optimal solutions at the trees boundaries. The solution is to perform a pos-processing optimization step, that tries to merge simple gates that have multiple fanout into their fanouts and also deals with the phase assignment problem. In the case of library based technologies like standard cells, for instance, the absorption of gates by their fanout is more rare. But the phase assignment indeed produces gains, reducing the final network gate count.

The mapping was exemplified using the circuit area as cost function, but it may be extended to deal with the delay cost function. The method is general in the sense that it minimizes an user defined cost function, that may reflect distinct technologic parameters. The problem of mapping for delay and area, for instance, turns out to be the problem of defining a cost function that takes into account both parameters. Of course, there are other *add hoc* features that could be added to improve its performance with respect to specific problems, like the duplication of logic to reduce the delay [Keu90] and so forth. But a well specified cost function should allow to obtain good mapping results.

Chapter 8

Conclusions and Future Work

This chapter present the final considerations about the work accomplished and draws some directions for further research.

The main goal of this work was to study the application of MBDs in the context of multi-level logic synthesis and at the same time develop new methods and algorithms based on them. It was motivated by the observation that the MBD (BDDs and its extensions in general) is a promising logic representation due to its compactness and its canonical features. The research strategy was to build a complete multi-level logic synthesis system using the MBDs to represent logic functions and verify the possible advantages of its use in each field. In this chapter we analyze the results obtained in each logic synthesis phase and propose some topics for future research.

The logic synthesis system built was called LOGOS (LOGic Optimization System), and its structure is presented in the first chapter. It addresses two targets: the synthesis in library based technologies and the synthesis in multiplexor based FPGAs. Library based technologies covers a large range of design styles. Any technology that implements logic circuits in terms of a set of logic gates can be tackled by this approach. Standard cells and gate-arrays are probably the most significant examples. The FPGAs address a different kind of circuit that in not adequately modeled by libraries of gates: the programmable devices. This area had a strong development last years and became a major technology trend for logic synthesis applications. There are several different FPGA architecture [Moo91]. Since we are also interested in analyzing the relationship between logic representation and target technology we restrict our attention to the multiplexor base FPGAs, due to the direct correspondence that exists among 2 to 1 multiplexors and nodes of the MBD. FPGAs and library base synthesis cover a significant

range of the multi-level synthesis area and therefore provided a good feedback concerning the interest in the use of MBDs.

The synthesis on library based designs follows a traditional approach that split the process into two main phases: the technology independent synthesis and the technology mapping. In the technology independent phase we apply algorithmic transformations to decompose the initial description into a set of Boolean functions, generating a Boolean network. Then its complexity is reduced by simplifying the node functions with respect to their don’t cares. The next step is the technology mapping which covers the network with gates from the target library.

There is a current research trend in methodologies that integrate both phases. We believe that this is an interesting long term research subject. For the purposes of this work, however, we opted to follow the traditional approach that still produce the best results.

The first point to consider is the choice of MBDs as logic representation. There are several alternative representation schemes based on BDDs. It is evident, as discussed in chapter 3, that MBDs are advantageous with respect to the case of representing each output as well as the don’t cares in separate BDDs. On the other hand, the alternatives proposed by Minato [Min90] as well as the multiple-valued BDDs [Bra89] where not analyzed and could be subject of further research. We believe that the algorithms developed in this work could be adapted to other types of BDDs. Indeed, it was already shown that this is the case for the minimization of the BDD size with respect to the don’t care set [Lin93]. We do not believe, however, that the alternatives cited above could present some significant improvement over MBDs, and we do believe that the use of a third terminal is in effect quite useful for logic manipulations.

Other extensions like strong canonical form and negative edges could be interesting for some applications. For instance, the decomposition methods developed here would be improved if we adopt these techniques. In the technology mapping, however, the strong canonical introduces some problems because we can not have functions with different orderings at the same time. Also, the generation of prime and irredundant covers presented here would be too costly with negative edges. To take advantage of both approaches, it will be useful to include strong canonical form BDDs with negative edges as an alternative representation in the C++ version of LOGOS. The object oriented programming eases this task by providing multiple inheritance mechanisms that help to create hierarchies of classes in which common behavior is *factored* out from similar entities. The extension of LOGOS to deal with alternative MBD representations is therefore a subject of further research.

One of the features that makes MBDs interesting for logic synthesis is their compactness. Although their worst case cost is exponential, in general they has a reasonable size for most practical functions. However, we have seen that their size depends strongly on the variable

ordering, which in turn depends on the type of the function. Only the class of symmetric functions has MBDs that are insensitive to the variable ordering. The other functions have a MBD size that varies in unpredictable ways with the ordering and there is no simple and general ordering strategy for them.

To deal with this problem we have developed incremental manipulation techniques that allow to generate new orderings without rebuilding the entire MBD at each step. A greedy and a stochastic reordering approaches were developed upon the incremental algorithms which produced very good results. The stochastic approach has found the best ordering for the set of benchmarks where this ordering is known. Although there is no guarantee of finding the best ordering in all cases, the empirical results indicates that the stochastic approach produce solutions that are very hard to improve. The greedy approach produce results that are less effective than the stochastic one, but it is more than one order of magnitude faster than it. The LISP prototype allowed us to deal with small to medium functions, but we have seen that its C++ version is quite fast and can process large problems. Both strategies are useful for practical reasons. We could say that the stochastic method should be used whenever possible, but for very large MBDs the greedy approach is more interesting in terms of CPU time.

The incremental manipulation technique is one of the most important outcomes of this work. It became clear along this research that it can be useful not only for the reduction of the MBD size, but also to help any method that explores the MBD topology to improve its solution. The most important example is the determination of symmetry classes of a function, where the swapping method is used to find all pairs of adjacent variables in the MBD. Another interesting consequence of this method is that it allows the application of stochastic techniques to the manipulation of MBDs, which was successfully applied to the ordering problem. Some synthesis systems like ASYL [Ben92] use these techniques for FPGA mapping, while some BDD packages start to employ it for the reduction of storage requirements.

The minimization of incompletely specified functions is fundamental problem that has a large application in logic synthesis. The use of MBDs for logic minimization had produced interesting results. The two-level covers can be performed in a single bottom-up step. The quality of the prime and irredundant cover generated is similar to those produced by the state of art two-level minimizers like ESPRESSO [Bra84]. The MBD size minimization proposed here is in fact a new research subject that, up to the knowledge of the author, was never tackled before. The algorithm developed here, although the simplifications derived from the filtering of the matching candidates, is still exhaustive in the sense that it try all matchings in the search of the best solution. This may be very time consuming if the number of candidates is large. A further research work in this area consists in finding good heuristics to select subsets of the matching candidates and also to develop alternative techniques to evaluate the gain of each possible solution. In the present version, the gain is evaluated by reducing the MBD and

verifying its size. Some incremental techniques could be developed to avoid the cost of reducing and computing the MBD size.

For library based synthesis two logic decomposition methods were developed. The direct decomposition provides interesting results, because it is fast and the decomposed networks are of reasonable complexity. The path oriented decomposition needs further development to become useful. Both of them can not preclude a previous reordering step, otherwise the results are not interesting. The Boolean factorization method presented does not relies on specific MBD features, but it produces good results and is useful in the factorization of the node functions. In this field some further research can be done in the search of good divisor functions. The method based on the analysis of the OFF set of the candidates could be further enhanced. We believe that the analysis of the OFF set of the function to be factored could lead to interesting results not only for the evaluation of the candidates but also to compute the divisors themselves.

Besides the decomposition methods described here, the field of functional decomposition is a good candidate for the application of MBDs. Several limitations of the ancient methods where related to the exponential complexity of the logic representation adopted (truths tables). The use of MBDs can extend those limits up to practical applications. However, if the nature of the algorithms is not changed, the use of MBDs will just improve the solutions by a constant factor, and the methods will still be inefficient for large problems. The combination of structural and functional techniques could be envisaged to try to overcome this problems.

The multi-level minimization problem was not treated in depth in this work. We have exemplified the use of MBDs for the node minimization approach, but several other alternatives could be tried. One possibility that may be object of future research is the application of the reduction of MBD size with respect to the don't cares in the multi-level optimization for FPGA synthesis. In this case, the mapping tool should be adapted to exploit the particularities of the multiplexor cells. In the approach presented here the MBDs were useful in providing a compact representation for the node functions, which may quickly attain huge sizes. We have also verified that the MBDs of the node functions can be significantly reduced by the reordering heuristics, which improved the performance of our two-level minimization. The MBD can become huge due to the large number of variables introduced in the support of the node function when computing the SDC and ODC of the node. The type of functions generated, nevertheless, is very sensitive to the reordering techniques.

The technology mapping is another field where the use of MBDs present some advantages. We have developed a Boolean mapping technique based on MBDs which yield good results. The Boolean mapping, in general, performs better than the structural mapping proposed by most mappers in the literature. Its main advantage is its ability to deal with a larger spectrum of

candidate gates for matching, which may lead to better solutions. In this case, the main contribution of MBDs was a fast method for symmetry detection between adjacent variables in the diagram that is used to speed up the classification of the functions, an important and costly step in the gate matching problem. The MBDs allow also the matching with simultaneous phase assignment that is executed in a single traverse of the diagram. These techniques lead to an important improvement in the performance of the Boolean mapping.

It should be noted that the fast symmetry detection may be useful in other areas of logic design, as in the synthesis of symmetric functions [Kim91] and so forth.

The concept of symmetric symmetry classes that was introduced here is another important factor in the improvement of the speed of the Boolean matching. For most gates in the usual standard cells libraries this feature applies. The simplification obtained is that this avoid permuting symmetry classes of same carnality in the matching phase, which reduces the cost of the matching.

Other interesting feature of our mapping approach is a flexible network covering which support both direct and recomposing covering. The final phase optimization step reduces the number of inverters by means of deterministic and stochastic manipulations. Deterministic manipulations analyze the circuit and replace a gate either by its complement or by its De Morgan's equivalent when this lead to a reduction of the total number of inverters. The stochastic transformations complement an arbitrarily subset of gates and then apply the deterministic manipulations to reduce the number of inverters. These techniques in general yield good savings in terms of inverters count.

The LISP version of the mapping tool could deal with circuits of standard benchmark sets in reasonable computing time, confirming the efficiency of the algorithms. As is the case in other fields of logic synthesis, its performance is evaluated empirically and its implementation in C++ would allow us to compare it against other Boolean mappers in the literature to get a more precise evaluation.

In the case of FPGA synthesis, we developed a method based on the direct mapping of the MBD into ACTEL cells. The results obtained confirm the assertion that the interrelationship between logic representation and the target technology is an important factor in logic synthesis. Here the association between MBD nodes and multiplexor cells provide us a good cost function to guide the MBD manipulation in order to optimize the final result. The cost function to be minimized is the MBD size, which correlates well with the circuit complexity. To optimize the cost function we apply the reordering techniques, the MBD size don’t care minimization and the subgraph resubstitution.

The subgraph resubstitution technique is an additional resource to reduce the MBD size. Although conceptually interesting, it has a restricted application. Most of times the reduction of the MBD size increases the sharing of nodes and prevent the application of subgraph resubstitution. It remains, however, as an alternative for improvement of some special cases.

The mapping based on the MBD size have produced results comparable to the state of art mappers, obtaining in several cases the best ones. This is mainly due to the reordering techniques that obtained compact diagrams, optimizing the circuit cost. In most cases the best results were obtained with the stochastic reordering, as expected. An interesting point that should be observed is that in some special cases the multi-level decomposed circuit is much simpler than the flattened MBD. We have seen two of such examples in benchmarks. These cases does not invalidate the use of MBDs for the synthesis, because once decomposed, the MBD mapping can be applied to the subfunctions, eventually with support for node collapsing using the cluster functions. As we had already stated above, the multi-level mapping for FPGA applications is a subject for future research in the context of the LOGOS system.

The overall results obtained of this work confirm that BDDs and particularly MBDs are a very interesting logic representation for logic synthesis applications. Moreover, this had become evident in last years due to the intense research on this subject. New methods and new variations of the original BDD representation are continuously being generated. In the field of logic minimization find several new applications, specially in two-level techniques. [Min92] presents an alternative technique for computing prime and irredundant covers of BDDs, using a top-down traverse instead of the bottom-up approach presented here. In [Cou92] a new technique for the implicit enumeration of prime and essential prime implicants of Boolean functions based on BDDs was presented that produced remarkable results. This technique allows the computation of the complete set of primes for functions with hundreds of thousands of implicants in a few seconds. It was already extended to deal with multiple value functions [Lin92]. The implicit enumeration was applied to two-level minimization [Cou93], extending the range of tractable problems. All these methods relies on efficiency of the BDD representation. We believe that this and other methods can be extended and applied in multi-level synthesis producing similar improvements. This is a new and exciting subject for future research.

Chapter 9

References

[Abo90] P. Abouzeid, G. Saucier, F. Poirot. “Lexicographical factorisation minimizing the critical path and the routing factor of multilevel logic”. In: *IFIP Working Conference on Logic and Architecture Synthesis* Paris - May 1990.

[Aho83] A. Aho, J. Hopcroft and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts. 1983.

[Ake78] S. B. Akers. “Binary Decision Diagrams”. In: *IEEE Trans. on Computers,* vol. c-27, no. 6, Jun. 1978.

[Ake79] S. B. Akers. “Functional Testing Using Binary Decision Diagrams”. In: 8th International Symposium on Fault Tolerant Computing, pp. 82-92, June 1979.

[Ash57] R. Ashenhurst. “The Decomposition of Switching Functions”. In: Proceedings of the *International Symposium on Switching Theory*, Harvard University, pp. 74-116, 1957.

[Bab92] B. Babba, M. Crastes and G. Saucier. “Input Driven Synthesis on PLDs and PGAs”. In: Proceedings of *29th Design Automation Conference,* June, 1992.

[Bar86] K. Bartlett, W. Cohen, A. De Geus and G. Hachtel. “Synthesis and Optimization of Multilevel Logic under Timing Constraints”. In: *IEEE Trans. on Computer-Aided Design,* vol. CAD-5, no 4, Oct. 1986.

[Bar88] K. Bartlett *et al.*. “Multilevel Logic Minimisation Using Implicit Don’t Cares”. In: *IEEE Trans. on Computer-Aided Design,* vol. 7, no. 6, June 1988.

[Ben92] T. Benson, H. Bouzouzou, M. Crastes, I. Floricia and G. Saucier. "Synthesis on Multiplexor-based FPGAs using Binary Decision Diagrams". In: Proceedings of *ICCD,* 1992.

[Ber88] R. A. Bergamaschi. "Automatic Synthesis and Technology Mapping of Combinational Logic". In: Proceedings of *IEEE International Conference on Computer-Aided Design,*. Nov. 1988.

[Ber88a] M.R.C.M. Berkelaar, J.A.G. Jess "Technology Mapping for Standard-Cell Generators". In: Proceedings of *IEEE International Conference on Computer-Aided Design,*. Nov. 1988.

[Ber88b] R. L. Berman and L. Trevillyan. "Improved Logic Optimization Using Global Flow Analysis Extended Abstract". In: Proceedings of *International Conference on Computer Aided Design,* 1988.

[Bos87] D. Bostic et al., "The Boulder Optimal Logic Design System". In: Proceedings of *IEEE International Conference on Computer-Aided Design,*. Nov. 1987.

[Bra82] R. K. Brayton, C. McMullen. "The Decomposition and Factorisation of Boolean Expressions". In: Proceeding of *International Symposium in Circuits and Systems (ISCAS-82)* Rome, May 1982.

[Bra83] D. Brand. *Redundancy and Don't Cares in Logic Synthesis*. In: *IEEE Transactions on Computer-Aided Design,* vol. CAD-32, no. 10, pp. 947-52 Oct. 1983.

[Bra84] R. K. Brayton, G. Hachtel, C. McMullen and A. Sangiovanni-Vincentelli. *Logic Minimisation Algorithms for VLSI Synthesis*. Boston, MA: Kluwer Academic, 1984.

[Bra87] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang. "MIS: A Multiple-Level Logic Optimization System". In: *IEEE Trans. on Computer-Aided Design,* vol. CAD-6, no. 6, Nov. 1987.

[Bra87a] R. K. Brayton. "Factoring Logic Functions". In: *IBM Journal of Research and Development,* Vol. 31, No. 2, March 1987.

[Bra88] R. K. Brayton, E. Sentovich, F. Somenzi. "Don't Cares and Global Flow Analysis of Boolean Networks." In: Proceedings of *IEEE International Conference on Computer-Aided Design,*. Nov. 1988.

[Bra89a] R. K. Brayton. "Multi-level Logic Synthesis". *ICCAD-89* Tutorial.

[Bra90] Karl S. Brace, Richard L. Rudell, Randal E. Bryant "Efficient Implementation of a BDD Package". In: Proceedings of *27th Design Automation Conference* 1990.

[Bry86] R. E. Bryant. "Graph-Based Algorithm for Boolean Function Manipulation". In: *IEEE Trans. on Computers,* vol. C-35, no. 8, Aug. 1986.

[Bry89] D. Bryan, F. Brglez and R. Lisanke. Redundancy Identification and Removal. In: Proceedings of the *International Workshop on Logic Synthesis*, Research Triangle Park, 1989.

[Cal91] N. Calazans, R. Jacobi, Q. Zhang and C. Trullemans. "Improving Binary Decision Diagrams Through Incremental Reduction and Improved Heuristics". In: Proceedings of *Custom Integrated Circuits Conference*, 1991.

[Cal92] N. Calazans, Q. Zhang, R. Jacobi, B. Yernaux and A. M. Trullemans. "Advanced Ordering and Manipulation Techniques for Binary Decision Diagrams". In: Proceedings of *European Design Automation Conference,* 1992.

[Che90] K. Chen, S. Muroga. "Timing Optimization for Multi-Level Combinational Networks". In: Proceedings of *27th Design Automation Conference*, pp. 339-44, 1990.

[Cor88] Vince Corbin, Warren Snapp. "Design Methodology of the Concorde Silicon Compiler" *Silicon Compilation*, Addison-Wesley, Reading (Mas) 1988.

[Cou92] O. Coudert, J. C. Madre. "Implicit and Incremental Computation of Prime and Essencial Implicants of Boolean Functions", Proceedings of *28th Design Automation Conference,* June, pp. 36-9, 1992.

[Cou93] O. Coudert, J. C. Madre, H. Fraisse. "A New Viewpoint on Two-level Logic Minimization", Proceedings of *30h Design Automation Conference,* June, pp. 625-30, 1993.

[Cur62] A. Curtiss. "A New Approach to the Design of Switching Circuits". Van Nostrand, Princeton, 1962.

[Dag86] M. Dagenais, V. K. Agarwal, N. C. Rumin. "McBOOLE: A New Procedure for Exact Logic Minimisation". In: *IEEE Trans. on Computer-Aided Design,* vol CAD-5, no 1, Jan. 1986.

[Dar81] J. A. Darringer, D. Brand, J. Gerbi, W. Joyner and L. Trevillyan. "Logic Synthesis Through Local Transformations". In: *IBM Journal of Research and Development,* Vol. 25, no. 4, pp. 272-80, July 1981.

[Dar84] J. A. Darringer et al. “LSS: A System for Production Logic Synthesis”. In: *IBM Journal of Research and Development*, Vol. 28, no. 5, pp. 537-45, Sep. 1984.

[Dav69] E. S. Davidson. “An algorithm for nand decomposition under network constraints”. In: *IEEE Transactions on Computers*, C-18, Dec. 1969.

[Dav83] M. Davio, J. Deschamps and A. Thayse. *Digital Systems with Algorithmic Implementation*. John Wiley and Sons, New York, 1983.

[Dav91] M. Davio. “Technology Mapping”. Private communication.

[Dav91a] M. Davio. “Algorithmic Methods in Boolean Design”. Class notes Elec2430, Laboratoire de Microélectronique - DICE, FSA - UCL.

[Dem90] G. De Micheli, D. Ku, F. Maillot, T. Truong. “The Olympus Synthesis System”. In: *IEEE Design & Test of Computers*, Oct. 1990.

[Det87] E. Detjens, G. Ganot, R. Rudell, A. Sangiovanni-Vincentelli, A. Wang. “Technology Mapping in MIS”. In: Proceedings of *IEEE International Conference on Computer-Aided Design,* Nov. 1987.

[Dev88] S. Devadas. “Boolean Decomposition in Multi-Level Logic Optimization”. In: Proceedings of *IEEE International Conference on Computer-Aided Design,*. 1988.

[Dev88a] S. Devadas. “Boolean Decomposition of Programmable Logic Arrays”. In: Proceedings of *Custom Integrated Circuits Conference,* 1988.

[Die69] D. Dietmeyer and Y. Su. “Logic Design Automation of Fan-in Limited NAND Networks.” In: *IEEE Transactions on Computers*, C-18(1), Jan. 1969.

[ElG89] A. ElGamal, J. Green, J. Reyneri, E. Rogoyski, K. El-Ayat and A. Mohsen. “An Architecture for Electrically Configurable Gate Arrays”. In: *IEEE Journal of Solid State Circuits*, 24(2), pp. 394-8, April 1989.

[Erc91] S. Ercolani and G. De Micheli. “Technology Mapping for Electrically Programmable Gate Arrays”. In: Proceedings of *28th Design Automation Conference*, pp. 234-9, 1991.

[Fil90] D. Filo, J. Yang, F. Maillot, G. De Micheli. “Technology Mapping for a Two-Output RAM-Based Field Programmable Gate Array”. In: Proceedings of *European Design Automation Conference,* 1991.

[Fri87] S. J. Friedman, D. J. Supowit. “Finding the Optimal Variable Ordering for Binary Decision Diagrams”. In: Proceedings of *24th Design Automation Conference,* 1987.

[Fuj85] H. Fujiwara. *Logic Testing and Design for Testability*. MIT Press Series in Computer Systems. The MIT Press, Cambrigde, Massachusetts, 1985.

[Fuj88] M. Fujita, H. Fujisawa, N. Kawato. “Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams”. In: Proceedings of *IEEE International Conference on Computer-Aided Design,* 1988.

[Fuj91] M. Fujita, Y. Matsunaga and T. Kakuda. “On Variable Ordering of Binary Decision Diagrams for the Application of Multi-level Logic Synthesis”. In: Proceedings of *European Design Automation Conference,* 1991.

[Gaj88] D. Gajski. *Silicon Compilation*, Addison-Wesley, Reading (Mas) 1988.

[Ger82] J. Gersting. *Mathematical structures for Computer Science*. W. H. Freeman and Company, New York, 1982.

[Geu85] A. J. de Geus, W. Cohen. “A Rule-Based System for Optimizing Combinational Logic”. In: *IEEE Design & Test of Computers*. 1985.

[Geu86] A. J. de Geus. “Logic Synthesis and Optimization Benchmarks for the 1986 Design Automation Conference”. In: Proceedings of *Design Automation Conference - DAC*, 1986.

[Gil84] J. L. Gilkinson, S. D. Lewis, B.B. Winter and A. Hekmatpour. “Automated Technology Mapping”. In: *IBM J. of Res. and Dev*., Vol. 28, No. 5, pp. 546-56, Sept. 1984.

[Gin89] A. Ginetti, C. Trullemans. “A Control Unit Compiler Based on C++”. In: *Brazilian Symposium on the Design of Integrated Circuits*. Rio de Janeiro, 12-14 April 1989.

[Gol59] S. Golomb. “On the Classification of Boolean Functions”. In: *IRE Transactions on Information Theory*, IT-5, pp. 176-86, 1959.

[Gur89] B. Gurunath, N. Biswas. “An Algorithm for Multiple Output Minimisation”. In: *IEEE Trans. on Computer-Aided Design,* vol. 8, no. 9, Sep. 1989.

[Hac89] G. Hachtel, R. Jacoby, P. Moceyunas. “On Computing and Approximating the Observability Don't Care Set”. In: Proceedings of *International Workshop On Logic Synthesis,* Research Triangle Park, North Carolina, USA, May 1989.

[Har65] M. Harrison. *Introduction to Switching and Automata Theory*. McGraw-Hill, New York, 1965.

[Hil92] D. Hill, E. Detjens. “FPGA Design Principles (a tutorial)”. In: Proceedings of *Design Automation Conference - DAC*, pp. 45-6, 1992.

[Hon74] S. J. Hon, R. G. Cain, D.L. Ostapko, “MINI: A heuristic approach for logic minimisation”. In: *IBM J. of Res. and Dev.*, Vol. 18, pp. 443-58, Sept. 1974.

[Hwa89] T.-T. Hwang, R. M. Owens and M. J. Irwin. “Communication Complexity Driven Logic Synthesis”. In: Proceedings of *International Workshop on Logic Synthesis,* Research Triangle Park, North Carolina, USA, May 1989.

[Iba71] T. Ibaraki, S. Muroga. “Synthesis of Networks with a Minimum Number of Negative Gates”. In: *IEEE Trans. on Computer-Aided Design,* vol. C-20, no. 1, Jan. 1971.

[Ish91] N. Ishiura, H. Sawada, S.Yajima. “Minimization of Binary Decision Diagrams Based on Exchanges of Variables”. In: Proceedings of *IEEE International Conference on Computer-Aided Design,*. Nov. 1991.

[Jac89] R. Jacoby, P. Moceyunas, H. Cho, H. Hachtel. “New ATPG Techniques for Logic Optimization”. In: Proceedings of *International Conference on Computer Aided Design,* 1989.

[Jac89a] R. Jacobi. “Silicon Compilation”. *ISSSE-89* Tutorial, Sep. 1989.

[Jac91] R. Jacobi, N. Calazans and C. Trullemans. “Incremental Reduction of Binary Decision Diagrams”. In: Proceedings of *International Symposium on Circuits and Systems* - Singapore, June 1991.

[Jac92] R. Jacobi and A.M. Trullemans. “Generating Prime and Irredundant Covers from Binary Decision Diagrams”. In: Proceedings of *European Design Automation Conference*, 1992.

[Jac93] R. Jacobi and A.M. Trullemans. “A New Logic Optimization Method for Multiplexor Based FPGA Synthesis”. In: Proceedings of *EuroDAC*, pp. 312-17, 1993.

[Jeo92] S. W. Jeong, B. Plessier, G. D. Hachtel, F. Somenzi. “Variable Ordering for Binary Decision Diagrams”. In: Proceedings of *European Design Automation Conference*, 1992.

[Kar88] K. Karplus. “Representing Boolean Functions with If-Then-Else DAGs”, Internal Report UCSC-CRL-88-28, November 30, 1988.

[Kar89] K. Karplus. “Using If-then-else DAGs for Multi-Level Logic Minimization”. In: Proceedings of *Decennial Cltech Conference on VLSI*, 1989.

[Kar91] K. Karplus. “Amap: a Technology Mapper for Selector Based Field-Programmable Gate Arrays”, In: Proceedings of *28th Design Automation Conference*, pp. 244-7, 1991.

[Kar91a] K. Karplus. “Xmap: a Technology Mapper for Table-lookup Field-Programmable Gate Arrays”, In: Proceedings of *28th Design Automation Conference*, pp. 240-3, 1991.

[Keb92] U. Kebsull, E. Schubert, W. Rosenstiel. “Multilevel Logic Synthesis Based on Functional Decision Diagrams”. In: Proceedings of *European Design Automation Conference*, 1992.

[Keu87] K. Keutzer. “DAGON: Technology Binding and Local Optimization by DAG Matching”. In: Proceedings of *24th Design Automation Conference,* 1987.

[Keu90] K. Keutzer, S. Malik and A. Saldanha. “Is Redundancy Necessary to Reduce Delay?”. In: Proceedings of *27h Design Automation Conference,* 1990.

[Kim91] B.-G. Kim, D. L. Dietmeyer. “Multilevel Logic Synthesis of Symmetric Switching Functions”. In: *IEEE Transactions on Computer Aided Design,* Vol. 10, no. 4, pp. 436-46, April 1991.

[Kin89] R. King and P. Banerjee. “ESP: Placement by Simulated Evolution”. In: *IEEE Trans. on CAD*, Vol. 8, No 3, March 1989.

[Knu69] D. Knuth. *The Art of Computer Programming - Fundamental Algorithms*. Addison-Wesley. 1969.

[Law64] E. L. Lawler. “An Approach to Multilevel Boolean Minimization”. In: *Journal of the ACM,* Vol. 11, No 3, July 1964, pp. 283-95.

[Lee59] C. Y. Lee. “Representation of Switching Circuits by Binary Decision Diagrams”. In: *BSTJ,* No. 38, July 1959, pp. 985-99.

[Leg88] M. Lega. “Mapping Properties of Multi-Level Logic Synthesis Operations”. In: Proceedings of *International Conference on Circuit Design*, pp. 257-61, October 1988.

[Lia92] H. Liaw and C. Lin. “On the OBDD-Representation of General Boolean Functions”. In: Proceedings of *IEEE Transactions on Computer-Aided Design,* Vol. 41, No. 6, June 1992.

[Lig88] M. Lightner, W. Wolf. “Experiments in Logic Optimization”. In: Proceedings of *IEEE International Conference on Computer Aided Design,* p.286-89, 1988.

[Lin92] B. Lin, O. Coudert, J. C. Madre. “Symbolic Prime Generation for Multiple-Valued Functions”, Proceedings of *28th Design Automation Conference,* pp. 40-4, June, 1992.

[Lin93] B. Lin. “Minimization of Incompletely Specified BDDs”. Private communication. 1993.

[Lis88] R. Lisanke, F. Brglez, G. Kedem. “McMAP: A Fast Technology Mapping Procedure for Multi-Level Logic Synthesis”. In: Proceedings of *International Conference on Circuit Design*, pp. 252-6, October 1988.

[Lis90] R. Lisanke. “Technology Mapping”. In: Tutorials, IFIP Working Conference On Logic and Architecture Synthesis, Paris, May 1990

[Mai88] F. Maillot and G. De Micheli. “Technology Mapping using Boolean Matching”. In: Proceedings of *IEEE International Conference on Computer-Aided Design,* 1988.

[Mal88] S. Malik, A. R. Wang, R. K. Brayton and A. Sangiovanni-Vincentelli. “Logic Verification using Binary Decision Diagrams in a Login Synthesis Environment”. In *Procee*dings of *IEEE International Conference on Computer-Aided Design,* 1988.

[Mal88a] A. Malik, R. K. Brayton, A. Richard Newton and A. Sangiovanni-Vincentelli. “A Modified Approach to Two-Level Logic Minimization”. In: *IEEE International Conference on Computer Aided Design,* 1988.

[Mal89] A. Malik, R. K. Brayton and A. Sangiovanni-Vincentelli. “Logic Minimization for Factored Forms”. In: *International Workshop in Logic Synthesis,* Research Triangle Park, North Carolina, USA, May 1989.

[Mat88] H.-J. Mathony, U. G. Baitinger. “CARLOS: An Automated Multilevel Logic Design System for CMOS Semi-Custom Integrated Circuits”. In: *IEEE Trans. on Computers,* vol. 7, no. 3, March 1988.

[Mat89] Y. Matsunaga, M. Fujita. “Multi-Level Logic Optimization Using Binary Decision Diagrams”. In: Proceedings of *IEEE International Conference on Computer Aided Design,* November, 1989.

[McC56] E. J. McCluskey. “Minimization of Boolean Functions”. In: *Bell Lab. Technical Journal,* 35, Nov. 1956.

[McG89] P. C. McGeer, Robert K. Brayton. “Efficient Prime Factorization of Logic Expressions”. In: Proceedings of *26th Design Automation Conference,* 1989.

[Min90] S. Minato, N. Ishiura, S. Yajima "Shared Binary Decision Diagrams with Attributed Edges for Efficient Boolean Function Manipulation". In: Proceedings of *27th Design Automation Conference*, 1990.

[Min92] S. Minato. "Fast Generation of Irredundant Sum-of-Products Forms from Binary Decision Diagrams". In: Proceedings of *SASIMI,* 1992.

[Moo91] W. R. Moore and W. Luk. *FPGAs. Oxford 1991 International Workshop on Field Programmable Logic and Applications.* Abingdon EE&CS Books, Abingdon, Holland, 1991.

[Mor89] C. R. Morrison, R. M. Jacoby and G. D. Hachtel. "TECHMAP: Technology Mapping with Delay and Area Optimization". In: *Logic and Architecture Syntheses for Silicon Compilers*, Elsevier Science Publishers B.V. (North-Holland), 1989

[Mur89] S. Muroga, Y. Kambayashi, H. Chi Lai, J. N. Culliney. "The Transduction Method - Design of Logic Networks Based on Permissible Functions". In: *IEEE Trans. on Computer-Aided Design,* vol.38, no 10, Oct. 1989.

[Mur92] R. Murgai, R. Brayton and A. Sangiovanni-Vincentelli. "An Improved Synthesis Algorithm for Multiplexor-based PGA's." In: Proceedings of *29th Design Automation Conference,* June, 1992.

[Ola90] L. Diaz-Olavarieta and S. G. Zaky. "A New Synthesis Technique For Multilevel Combinatinal Circuits". In: Proceedings of *European Design Automation Conference*, pp. 122-7 1990.

[Oct91] OCTTOOLS Distribution 3.5. "Tool Man Pages". Electronics Research Laboratory, University of California, Berkeley. 1991.

[Ped91] M. Pedram and N. Bhat. "Layout Driven Technology Mapping". In: Proceedings of *28th Design Automation Conference,* 1991.

[Pit89] E. B. Pitty. "A Critique of the Gatemap Logic Synthesis System". In: Logic and Architecture Syntheses for Silicon Compilers, Elsevier Science Publishers B.V. (North-Holland), 1989.

[Qui52] W. Quine. "The Problem of Simplifying Truth Functions". In: *American Mathematical Monthly*, Vol. 59, pp. 521-31, 1952.

[Rab88] J. Rabaey and others, "CATEDRAL II: A Synthesis System for Multiprocessor DSP Systems" *Silicon Compilation*, Addison-Wesley, Reading (Mas) 1988.

[Rud86] R. Rudell, A. Sangiovani-Vincentelli. “Multiple-Value Minimization for PLA Optimization”. In: Proceedings of *IEEE International Conference on Computer-Aided Design,*. Nov. 1986.

[Rud87] R. Rudell, A. Sangiovani-Vincentelli. “Exact Minimization of Multiple-Valued Functions for PLA Optimization”. In: IEEE *Transactions on Computer Aided Design*. Vol. CAD-6, no. 5, Sept. 1987.

[Rut65] D. E. Rutherford. “Introduction to Lattice Theory”. Oliver & Boyd LTD., Tweeddale Court, Edinburgh, Scotland. 1965.

[Saa91] Y. Saab and B. Rao. “Combinational Optimization by Stochastic Evolution”. In: *IEEE Trans. on CAD*, Vol. 10, No 4, April 1991.

[Sak90] K. Sakouti and G. Saucier. “A fast and effective technology mapper on an autodual library of standard cells”. In: *IFIP Working Conference on Logic and Architecture Synthesis* Paris - May 1990.

[Sal89] A. Saldanha, A. Wang, R. K. Brayton and A. Sangiovanni-Vincentelli. “Multi-Level Logic Simplification using Don't Cares and Filters”. In: Proceedings of *26h Design Automation Conference,* 1989.

[Sau92] J. Saul. “Logic Synthesis for Aritmetic Circuits Using the Reed-Muller Representation”. In: Proceedings of *European Design Automation Conference*, EDAC, 1992.

[Sav90] H. Savoj, R. Brayton. “The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks”. In: Proceedings of *27h Design Automation Conference,* 1990.

[Sch88] M. H. Schulz, E. Trischtler and T. M. Sharfert. “SOCRATES: A Higly Efficient Automatic Test Pattern Generation System”. In: *IEEE Transactions on CAD*, Vol. 7, no. 1, pp. 126-37, January 1988.

[Sha38] C. E. Shannon. “The Symbolic Analysis of Relay and Switching Circuits”. In: *Transactions of AIEE,* 57, pp. 713-23, 1938.

[Sha49] C. E. Shannon. “The Synthesis of Two-terminal Switching Circuits”. In: *Bell System Technical Journal*, Vol. 28, pp. 59-98, Jan. 1949.

[Wal85] R. Walter, D. Thomas. “A Model of Design Representation and Synthesis”. In: Proceedings of *22nd Design Automation Conference*, 1985.

[Wan92] W. Wan. “A New Approach to the Decomposition of Incompletely Specified Functions Based on Graph Coloring and Local Transformation and Its Application to FPGA Mapping.” Master of Science in Electrical and Computer Engeneering, Portland State University, 1992.

[Woo91] N.S. Woo. “A Heuristic Method for FPGA Technology Mapping Based on the Edge Visibility”. In: Proceedings of *Design Automation Conference*, 1991.

[Xil92] Xilinx, Inc., “Xilinx Programmable Gate Array Data Book”, 1992.

[Ykm89] C. Ykman-Couvreur. “Multi-Level Boolean Optimization for Incompletely Specified Boolean Functions in PHIFACT”. In: *Logic and Architecture Syntheses for Silicon Compilers,* Elsevier Science Publishers B.V. (North-Holland), 1989.

Appendix

The LISP Prototype

The algorithms presented in this work where first developed in Common LISP on Macintosh personal computers. A graphic interfaced was developed in order to help the study and the research on MBDs. This tool was and important factor for the development of this work. Its graphic feedback was a source of insight and inspiration in the topological analysis of MBDs. This appendix describes some of its features.

1. Input Data Formats

LOGOS reads two type of circuit descriptions. The format **MLL** describes multilevel circuits through a set of logic equations while the format **PLA** describes circuits in the sum-of-products form. In the descriptions bellow, the words in italic indicate keywords that should be written exactly as they appears. The system does not distinguinsh lower and upper cases. The text between angle brackets "<>" must be replaced by the required data. Text between square brackets "[]" may repeated several times, but should appears at least once.

1.1 MLL Format

The **MLL** format follows a simple and rigid structure.

:NAME-CIRCUIT <name>
:INPUT-VARIABLES (<list of input variables separated by blanks> *)*
:INTERMEDIATE-VARIABLES (<list of intermediate variables separated by blanks>*)*
:OUTPUT-VARIABLES (<list of output variables separated by blanks> *)*
:EQUATIONS ([<equation> <set>] *)*

In the field ":equations", the <equation> is a quoted string describing a logic equation using the following set of operators:

"=" - assignment operator.
"~" - logic NOT operator.
"+" - logic OR operator.
"*" - logic AND operator, which is optional
"[]" - subexpressions.

The "<set>" parameter indicates ON, OFF or DC set, i.e., the characteristic set that is being described by the equation. Each intermediate or output variable may appears only once, for each set, at the left hand of the equation.

Here is an example of such description:

```
:NAME-CIRCUIT Test
:INPUT-VARIABLES ( x1 x2 x3 x4 )
:INTERMEDIATE-VARIABLES ( y1 y2 )
:OUTPUT_VARIABLES ( z1 z2 )
:EQUATIONS (   " y1 = x1 + x2 " ON
               " y2 = x3x4 + ~y1 " ON
               " y1 = ~y1[x1 + x2] + y1~x1x2 " DC
               " z1 = x1 + y1 " ON
               " x2 = y1 + ~y2 " ON )
```

1.2 PLA Format

The PLA format is a subset of the format used by ESPRESSO. It corresponds to the default type "fd". The input description is as follows.

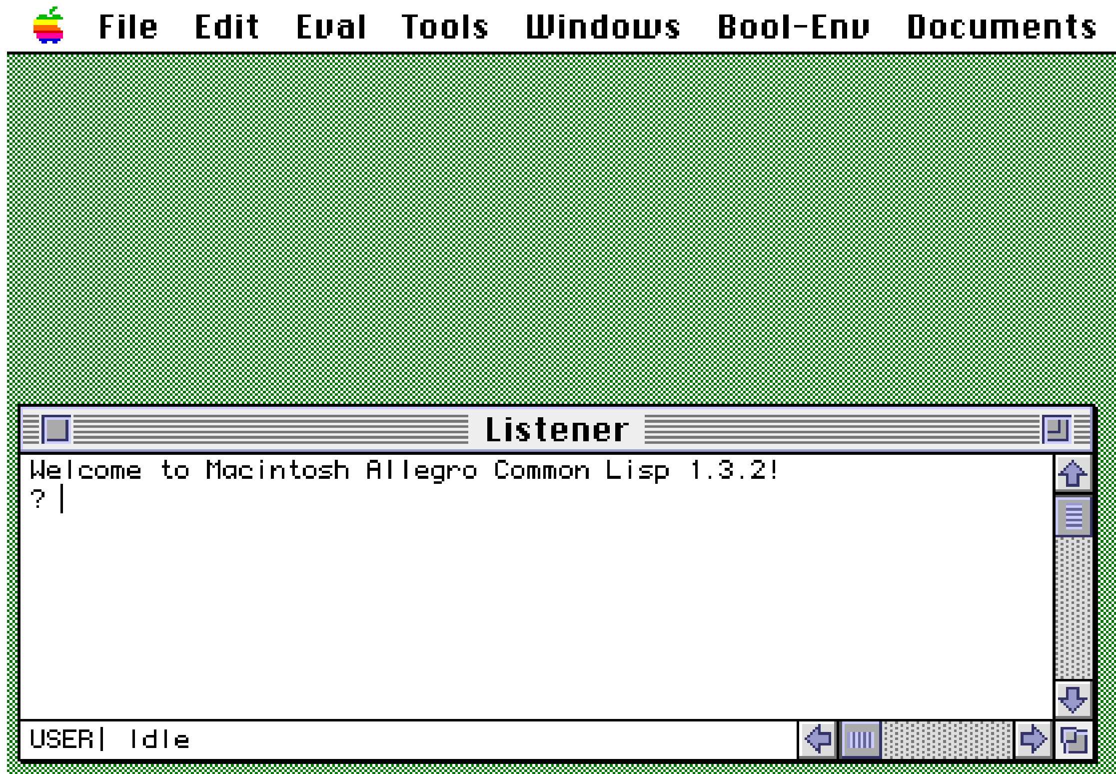
.I <number of inputs>
.O <number of outputs>
.p <number of products>
[<input cube> <output cube> *]*
.e

Input and output cubes are strings of *0's*, *1's* and *-'s*. A "2" is a sinonymous of "-" in the output cubes. Here is an example of single output PLA.

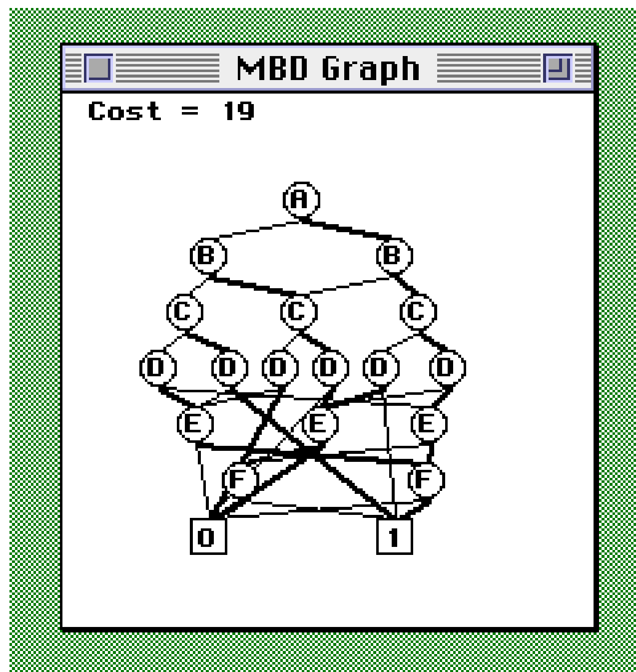
```
.i 4
.o 1
.p 6
0000 0
0001 0
0010 0
01-1 1
100- 1
-110 -
1111 2
.e
```

2. Graphic Interface

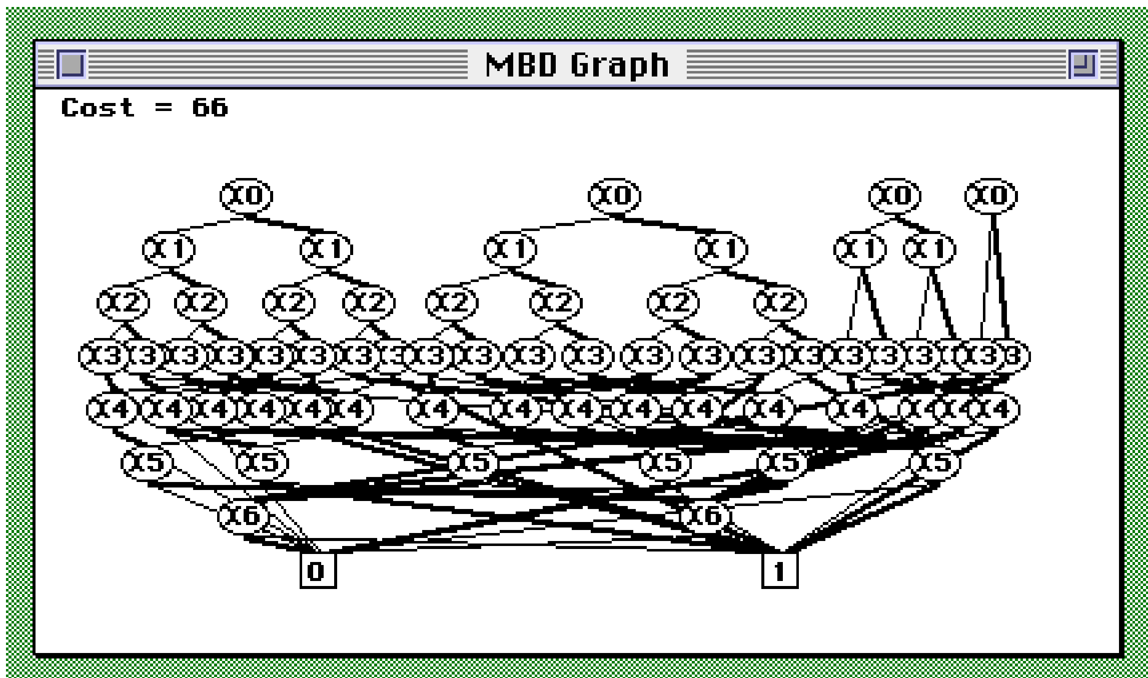
For illustration purposes we show how the graphic interface looks like.



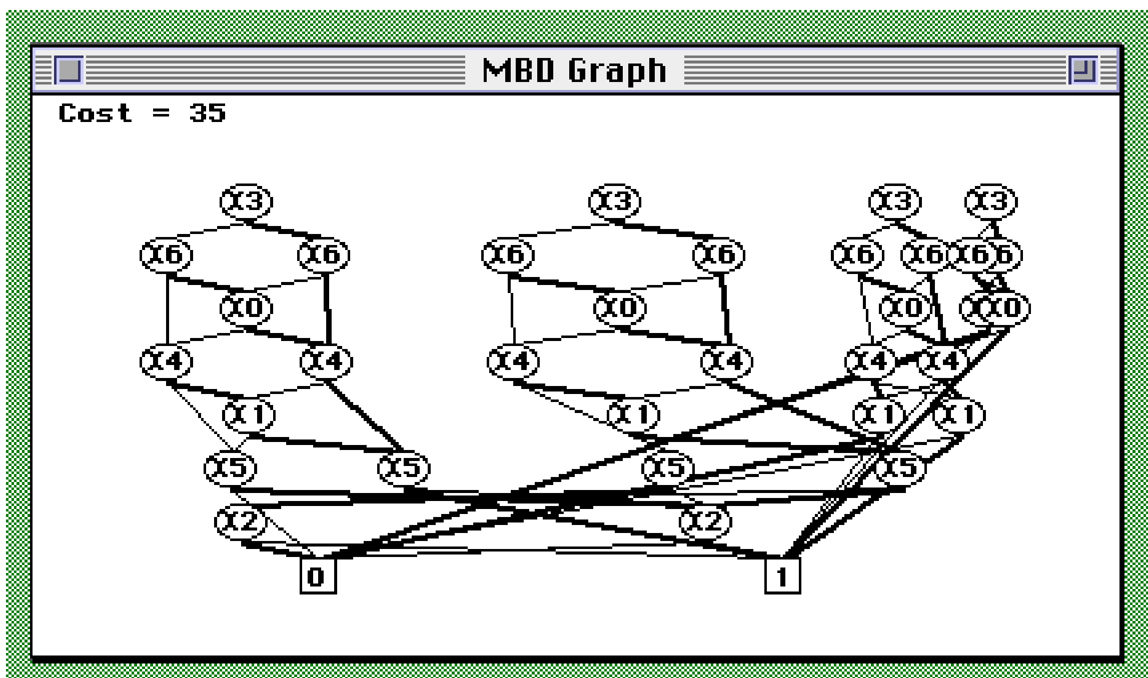
Menu and command window



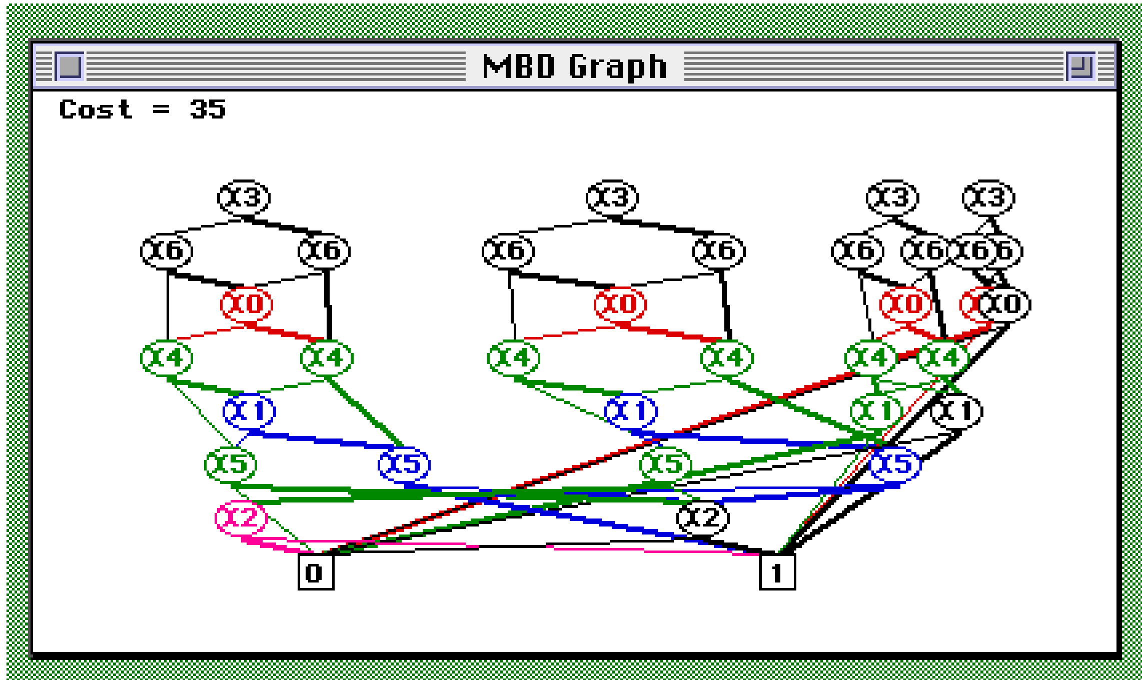
A single-output MBD



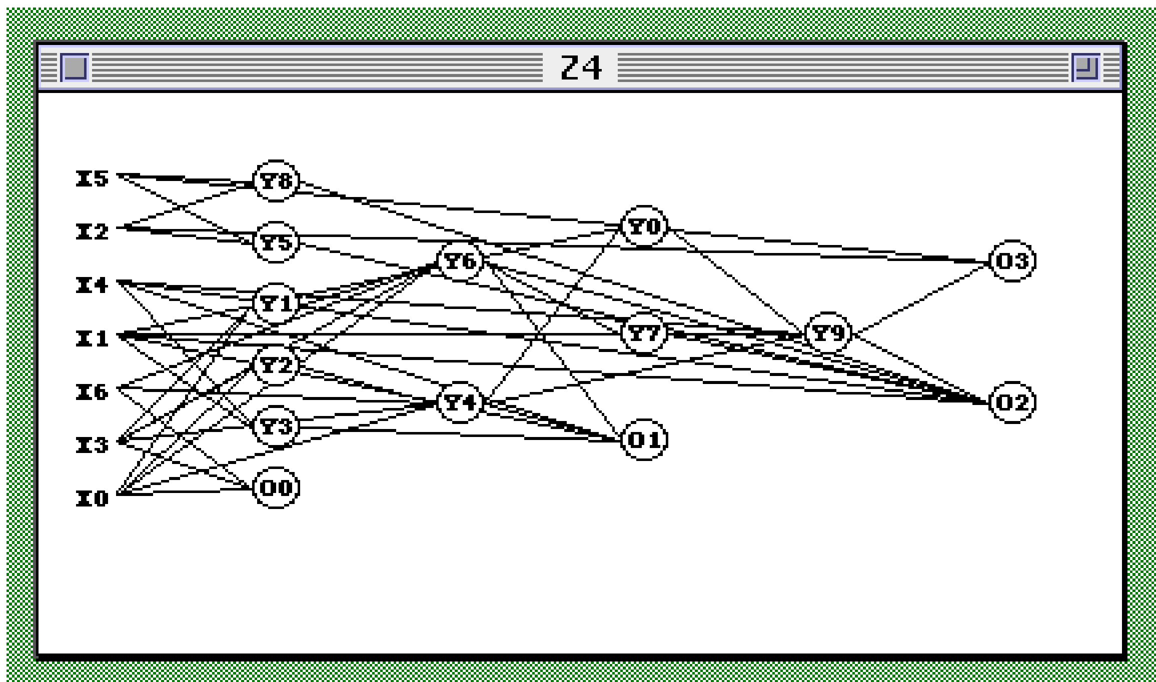
A larger MBD with multiple outputs



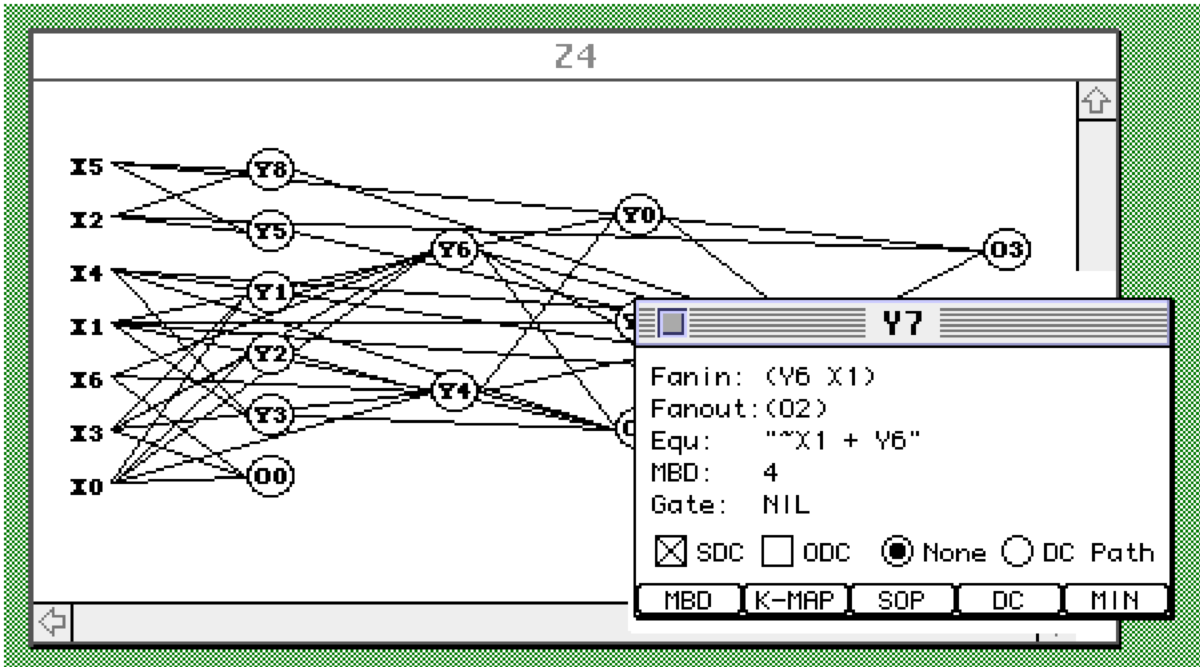
The same MBD after greedy reordering



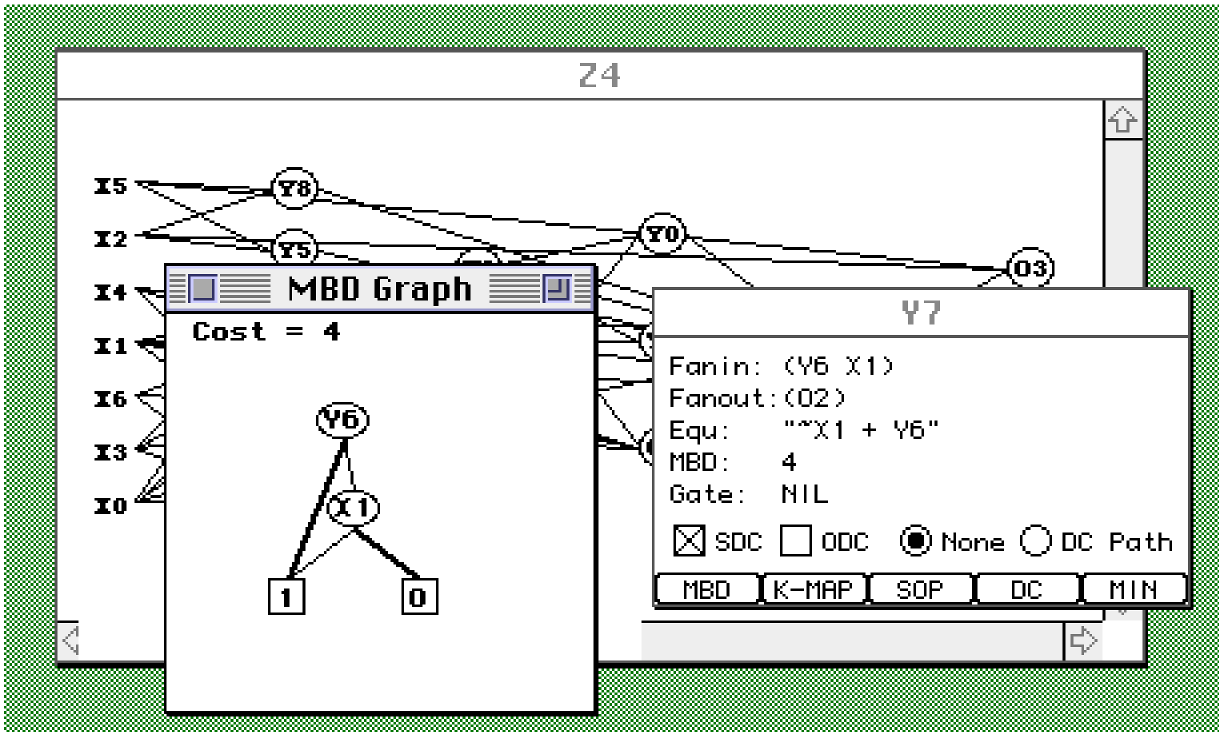
FPGA Mapping. Mapped multiplexor cells presented in different colors.



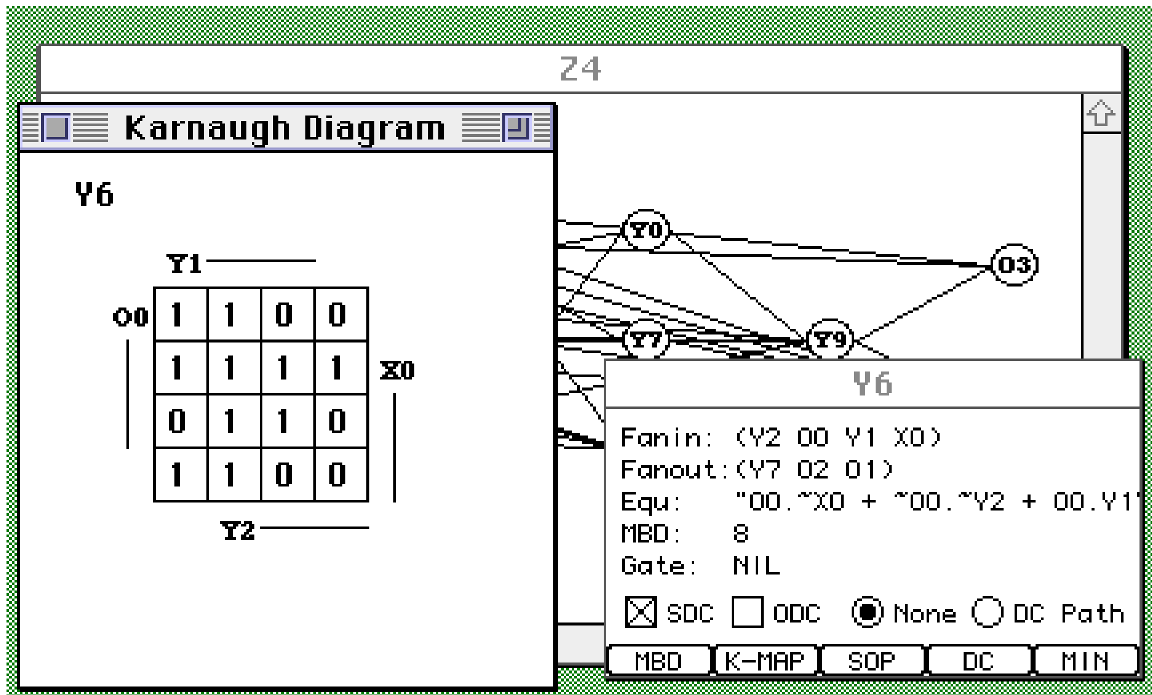
Z4 network



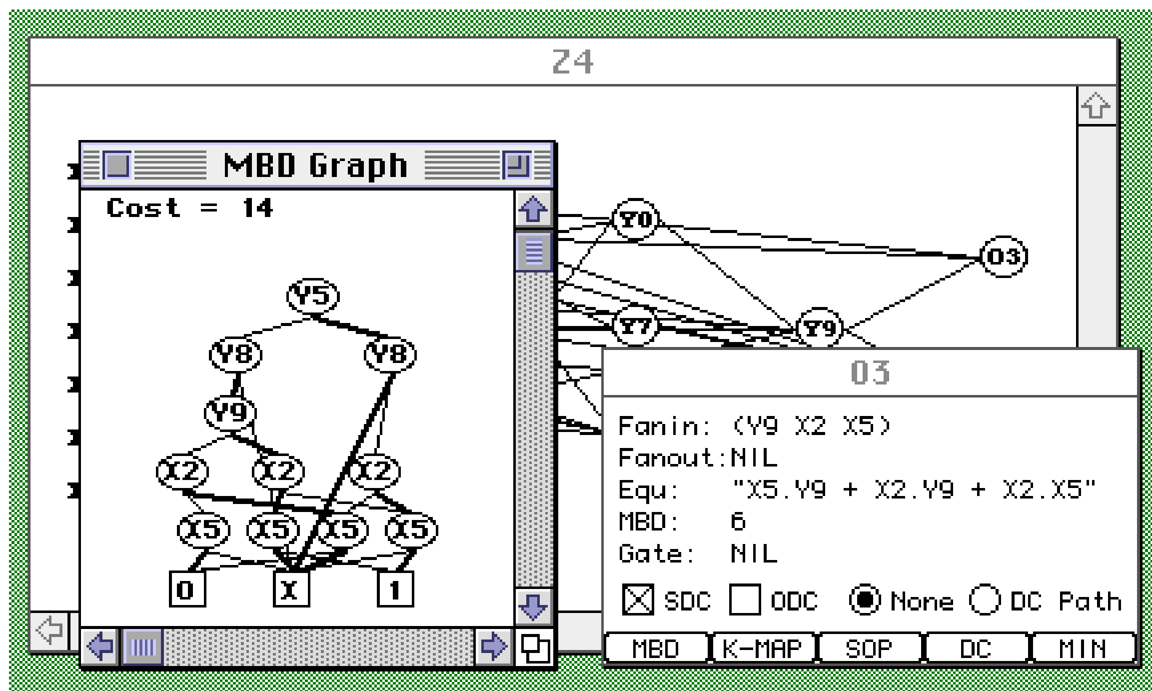
Inspecting nodes of the network



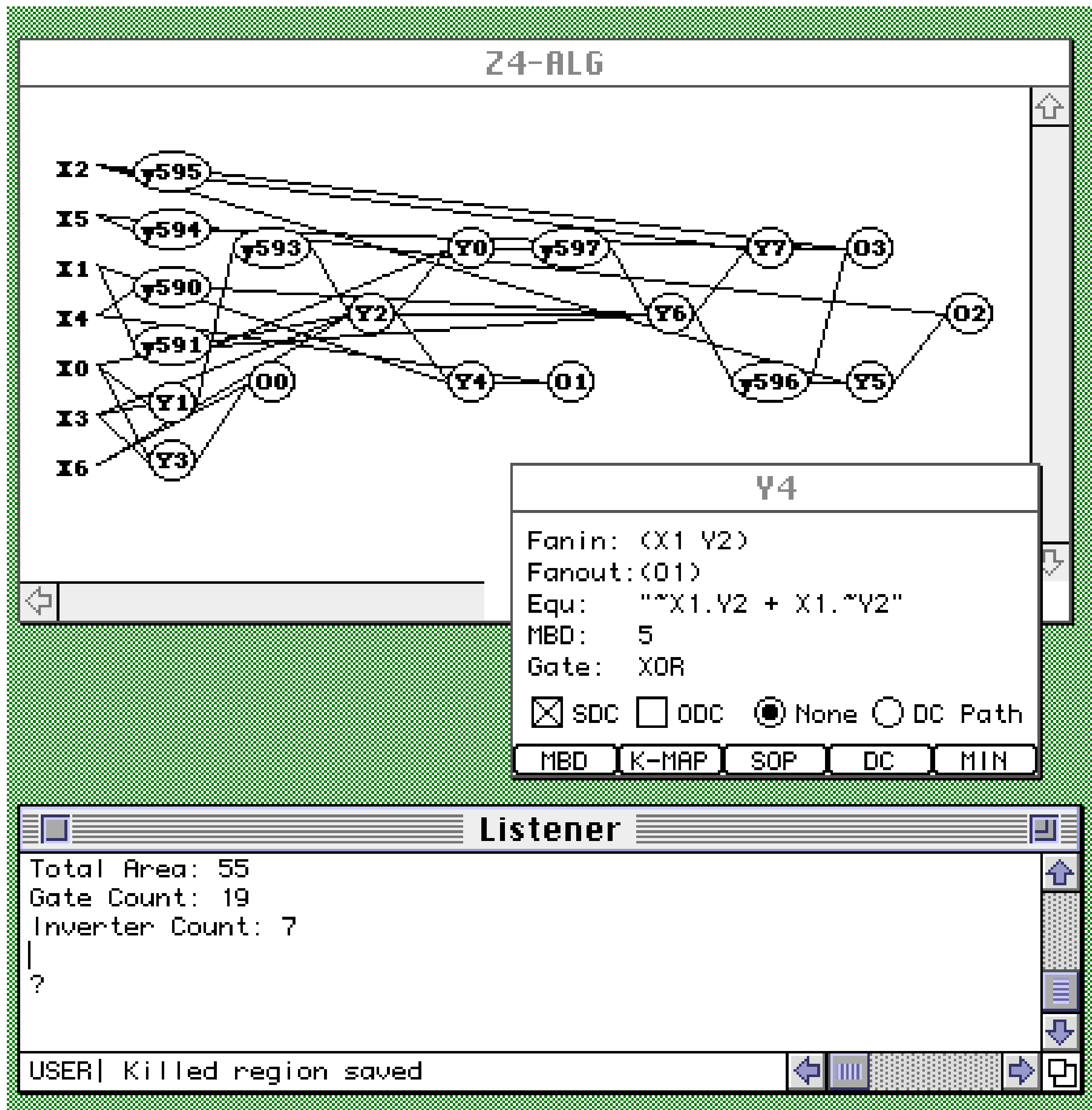
The MBD of a node



The Karnaugh diagram of a node's function



Computing the SDC of a node



Standard Cells mapping. Node window indicates the selected gate, while the command window draw statistics