TECHNOLOGY MAPPING FOR VLSI CIRCUITS EXPLOITING BOOLEAN PROPERTIES AND OPERATIONS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Frédéric Mailhot
December 1991

© Copyright 1991
by
Frédéric Mailhot

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Giovanni De Micheli
(Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Mark A. Horowitz

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Robert W. Dutton

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies & Research

Abstract

Automatic synthesis of digital circuits has gained increasing importance. The synthesis process consists of transforming an abstract representation of a system into an implementation in a target technology. The set of transformations has traditionally been broken into three steps: high-level synthesis, logic synthesis and physical design.

This dissertation is concerned with logic synthesis. More specifically, we study technology mapping, which is the link between logic synthesis and physical design. The object of technology mapping is to transform a technology-independent logic description into an implementation in a target technology. One of the key operations during technology mapping is to recognize logic equivalence between a portion of the initial logic description and an element of the target technology.

We introduce new methods for establishing logic equivalence between two logic functions. The techniques, based on Boolean comparisons, use Binary Decision Diagrams (BDDs). An algorithm for dealing with completely specified functions is first presented. Then we introduce a second algorithm, which is applicable to incompletely specified functions. We also present an ensemble of techniques for optimizing delay, which rely on an iterative approach. All these methods have proven to be efficient both for run-times and quality of results, when compared to other existing technology mapping systems.

The algorithms presented have been implemented in a technology mapping program, Ceres. Results are shown that highlight the application of the different algorithms.

Acknowledgements

I would like to thank Giovanni De Micheli, my advisor, for his guidance and support. He has been my mentor during my years at Stanford. I am also grateful to Mark Horowitz and Robert Dutton who read my dissertation and whose comments provided more polish to the final work. I appreciated the interactions with Abbas El Gamal, who recognized the importance of FPGA's.

I would like to mention some of the students with whom I interacted during the years at Stanford: David Ku, Maurizio Damiani, Thomas Truong, Polly Siegel, Jerry Yang, David Filo, Rajesh Gupta, Claudionor Coelho. Working with them has been both fun and stimulating. The list would not be complete without Michiel Ligthart, who spent a year and a half at Stanford as a Philips/Signetics visiting scholar. I would like to give special thanks to Polly Siegel and Thomas Truong, who took the time to read and review this dissertation with great care.

I would also like to thank my officemates, John Vlissides and Faheem Akram, without whom my years in CIS would not have been the same. The presence of Lilian Betters has also made CIS more agreeable.

This work was supported in part by DEC and NSF under a PYI award, and by ARPA under grant J-FBI-88-101. The financial support from the National Research Council of Canada and from Quebec's Fonds Canadien pour l'Aide a la Recherche is also acknowledged. I am also greatly indebted to the University of Sherbrooke for their help in financing my doctorate.

Throughout the years, my parents have always been supportive and encouraging. They deserve all my gratitude.

Finally, I keep for the end the person who made all this possible, Marie Papineau, to

whom I have the joy of being married. She has been a constant support, an inspiration, and a dearest friend.

Contents

Abstract **iv**

Acknowledgements **v**

1 Introduction **1**

1.1 Automatic digital circuit synthesis 2

1.1.1 Logic synthesis: a historic perspective 2

1.1.2 High level synthesis 4

1.1.3 Physical design 5

1.1.4 The Olympus Synthesis System 6

1.2 Terminology and notation 10

1.3 Contributions 13

1.4 Thesis outline 14

2 Technology mapping **16**

2.1 The technology mapping problem 17

2.2 Previous work 22

2.2.1 Rule-based systems 23

2.2.2 Algorithmic-based systems 25

2.3 Comparison of rule-based and algorithmic methods 36

3 An algorithmic approach to technology mapping **38**

3.1 Decomposition 39

3.2 Partitioning 43

3.3 Covering 47

4 Boolean matching **52**

4.1 Matching logic functions 53

4.2 Use of Binary Decision Diagrams 54

4.3 A simple Boolean matching algorithm 55

5 Matching completely specified functions **58**

5.1 Search space reduction 58

5.1.1 Unateness property 59

5.1.2 Logic symmetry 60

5.2 Determination of invariant properties 65

5.2.1 Unateness extraction 66

5.2.2 Symmetry classes determination 67

5.3 Results 68

6 Matching incompletely specified functions **77**

6.1 Using *don't care* information 78

6.2 Compatibility graph 79

6.3 Computation of relevant *don't care* sets 87

6.3.1 Image-based computation of *don't care* sets 89

6.3.2 Approximations in image computations 95

6.3.3 Relations to testability 97

6.4 Results 97

7 Performance-driven technology mapping **101**

7.1 Delay model 102

7.2 The difficulty with timing-driven technology mapping 103

7.3 Repartitioning 104

7.4 Redecomposition 106

7.5 Re-covering 109

7.6 Iterative mapping 110

7.7 Buffering/repowering 111
7.8 Results 112

8 Conclusion and future directions **116**

Bibliography **119**

List of Tables

5.1 Mapping results for area (No *don't cares* , Actel library Act1, restricted set of gates commercially distributed) 72
5.2 Mapping results for area (No *don't cares* , Complete Act1 library) 73
5.3 Mapping results for area (No *don't cares* , Complete Act2 library) 74
5.4 Mapping results for area (No *don't cares* , LSI Logic library) 75
5.5 Mapping results for area (No *don't cares* , Library derived from the Quicklogic Master cell) 76
6.1 Number of k-input cells used in mapping 30 benchmarks with the full Act1 and the LSI Logic libraries (depth = 5) 82
6.2 Mapping results for area (Using *don't cares* , Actel library) 99
6.3 Mapping results for area (Using *don't cares* , LSI Logic library) 100
7.1 Mapping results for delay (No *don't cares* , Actel library) 114
7.2 Mapping results for delay (No *don't cares* , LSI Logic library) 115

List of Figures

1.1 The Olympus Synthesis System 9
2.1 Illustration of the recursive covering definition 19
2.2 A simple covering example 20
2.3 Two transformation rules from LSS 24
2.4 Tree pattern and associated trie (adapted from [HO82]) 27
2.5 Two trees and their associated tries (adapted from [AGT89]) 28
2.6 Aho-Corasick matching automaton (adapted from [AGT89]) 29
2.7 Patterns for a simple library 30
2.8 Aho-Corasick automaton for patterns of a simple library 31
2.9 Example of pattern-based covering 32
2.10 Example of a DAG, a leaf-DAG and a tree 34
3.1 Algorithm for network decomposition 41
3.2 Two-input decomposition 42
3.3 Algorithm for network partitioning 44
3.4 Circuit partitioning 46
3.5 Graph covering 48
3.6 Graph of all possibles covers of j 49
3.7 Algorithm for network covering 50
4.1 Simple algorithm for Boolean matching 56
5.1 Distribution of symmetry classes C_i 64
5.2 Algorithm for fast Boolean matching 65
5.3 Algorithm for unateness extraction 70
5.4 Algorithm for symmetry extraction 71

6.1 Two pass-transistor implementations of $\mathcal{F} = (a + b)c$, $\mathcal{DC} = \overline{b}\overline{c}$ 79
6.2 Mapping *majority* with LSI Logic library elements 80
6.3 Matching compatibility graph for 3-variable Boolean space 81
6.4 Paths from vertex 5 in the matching compatibility graph 84
6.5 Algorithm for compatibility graph traversal using *don't cares* 86
6.6 Example of a partially mapped network 88
6.7 Algorithm for dynamic controllability *don't cares* calculation 89
6.8 Subnetwork generating the inputs of F 91
6.9 Algorithm for image-based *don't care* calculation 93
6.10 Shannon cofactor and generalized cofactor 94
6.11 Width of a Boolean network $\mathcal{N}$ 95
6.12 Illustration of *don't cares* limiting heuristic 96
7.1 Delays in a subcircuit 105
7.2 Delays in a subcircuit after gate duplication 106
7.3 Algorithm BDD to Boolean network conversion 108
7.4 BDD and corresponding Boolean network 109
7.5 Pseudocode for iterative delay optimization 111
7.6 Example of area/delay tradeoffs 113

Chapter 1

Introduction

Automatic synthesis of logic circuits has gained increasing importance in the digital-circuit-design environment. Synthesis of digital circuits is now widely used because it allows rapid generation of high quality circuitry. The requirement for fast turn-around mandates that synthesis tools must produce results quickly, which then allows system designers to easily experiment with different design options. The necessity for high quality means that automatically generated circuits must meet or exceed the level of performance achieved by custom, hand-designed systems.

In order to fulfill these goals of efficiency and quality, highly abstract descriptions of target systems become the vehicle of choice. The synthesis process is typically broken into a sequence of steps which gradually transform the abstract description into an actual implementation. The transformation steps are subdivided into three classes: operations on abstract representations, operations on logic descriptions and operations on geometric representations. The first class is called high-level synthesis, the second class logic-level synthesis and the last physical-design synthesis. Logic-level synthesis is further subdivided into technology-independent and technology-dependent operations. Technology mapping, which is the subject of this thesis, represents the technology-dependent step.

In this chapter, we review current automatic design methodologies. We briefly overview the origins of logic synthesis, including two-level and multi-level formulations. We explain how current synthesis systems include both high level and logic level synthesis, as well as physical design. We briefly describe *Olympus* as a test-case example

of a synthesis system incorporating high-level and logic-level operations. We emphasize the importance of technology mapping in the general synthesis framework, and describe the contributions of the research we did on that subject. We introduce terminology which will be used later in the thesis, and conclude the chapter with an outline of the remainder of the thesis.

1.1 Automatic digital circuit synthesis

The quality of automatically synthesized logic circuits depends on the transformations at each level of the synthesis process. Therefore, before addressing the specifics of technology mapping, we describe the different steps involved in current automatic synthesis systems. We will review the operations carried out at each step, looking at existing systems and tracing their origins. We start with logic synthesis, then briefly present high-level synthesis, followed by physical design. As a case example, we then describe Stanford's *Olympus Synthesis System*, a vertically integrated, automatic synthesis system.

1.1.1 Logic synthesis: a historic perspective

The origins of logic synthesis date back to the 1950's, when the emergence of the transistor made the implementation of logic circuits easier. In the mid-1950's, the first generation of tube-based digital computers was fully operational, making rudimentary design automation possible for the next generation of transistor-based computers [Pre72, CK56, Sco58]. An instructive measure of the heightened interest in logic operations was the rapidly increasing number of publications in the field during that decade [Hol61]. Quine introduced a method of simplifying two-level logic expressions, which was later improved by McCluskey [Qui52, McC56b]. Methods for minimizing two-level logic representations were studied extensively at that time, because sum-of-products were the most common way of implementing transistor-based digital circuits.

In parallel with the study of two-level logic optimization, multi-level representations were also investigated. Ashenhurst introduced the idea of logic decomposition [Ash59],

which is the basis for recognizing multi-level logic expressions. His method made possible *simple decompositions*, which reexpress a logic function $f(x_1, x_2, \ldots, x_n)$ as a combination of two new functions $g(x_i, \ldots, x_j, \Phi)$, $\Phi(x_k, \ldots, x_l)$ where $\{x_i, \ldots, x_j\} \cup \{x_k, \ldots, x_l\} = \{x_1, x_2, \ldots, x_n\}$ and $\{x_i, \ldots, x_j\} \cap \{x_k, \ldots, x_l\} \equiv \emptyset$. Curtis expanded the method to allow *complex decompositions* [Cur61, Cur62], that is, decompositions where the second condition is relaxed (inputs of g and Φ can be common to both functions). Roth and Karp presented an algorithm implementing both simple and complex decompositions [RK62]. Their system could potentially generate the optimum logic circuit implementing a user-provided logic description. The major limitation was that only small circuits could be processed, because of the computational complexity of the algorithm.

Schneider and Dietmeyer built a system for multiple-output function synthesis based on the decomposition techniques of Karp and Roth [SD68]. Muroga introduced the idea of permissible functions at the beginning of the 1970's [MKLC89]. Liu was concerned with the automatic synthesis of MOS networks [Liu77c, Liu77b]. Most of these methods were computationally too complex at the time for being effectively used on large circuits.

Until the end of the 1970's, most logic optimization systems would be able to process circuits with limited number of inputs and outputs. With the introduction of the Programmable Logic Array (PLA) and the advent of VLSI during the 1970's [MC81], approximate methods for two-level minimization became very important, because the size of the circuits made the use of exact solutions very difficult [BHMSV84]. In 1974, Hong *et al.* developed MINI, a program based on an efficient algorithm for prime implicant generation [HCO74]. Arevalo presented an approximate two-level minimization method targeted at PLAs [AB78]. Svoboda introduced an efficient heuristic for two-level logic simplification [SW79], which was later implemented as program PRESTO [Bro81]. Brayton *et al.* introduced ESPRESSO, a combination of heuristics that proved very successful at simplifying two-level logic expressions [BM84]. Dagenais presented an exact minimization algorithm for two-level logic expressions [DAR86]. Gurunath and Biswas described an efficient two-level minimization algorithm relying on the generation of only a small portion of all the possible prime cubes [GB87]. Since PLAs were the technology

of choice for hardware implementation of combinational logic, the problem of simplifying the hardware was one of simplifying two-level representations and then taking logic information into account at the layout level, to allow compression (or *folding*) of the array [DSV83].

With the main technology for digital circuits shifting from NMOS to CMOS, and the appearance of gate arrays, PLAs proved insufficient to meet the more stringent timing and area requirements. Two-level logic minimization was augmented to multi-level logic optimization. Brayton and McMullen proposed a decomposition technique suited for multi-level logic operations [BM82, BHMSV84]. Many multi-level logic optimization techniques and systems were developed in the 1980's [DJBT81, GBdGH86, BDK+86, BRSVW87, BHJ+87]. One of the new problems that appeared with multi-level logic synthesis was that the transformation from the optimized logic description to a target technology was no longer a transformation into a well-defined, regular PLA structure. In addition to simplifying the logic description of circuits in a technology-independent set of operations, logic synthesis systems had to be extended to take the target technology into account, and technology mapping became the necessary technology-dependent step for automatic synthesis of digital circuits.

The purpose of technology mapping is to transform an arbitrary multiple-level logic representation into an interconnection of logic elements selected from a fixed library of elements. Technology mapping is a very crucial step in the synthesis of semi-custom circuits for different technologies, such as sea-of-gates, gate-arrays, standard cells, or field programmable gate arrays (FPGA). The quality of the final implementation, both in terms of area and performance, depends heavily on this step.

1.1.2 High level synthesis

It is possible to use logic synthesis tools directly, and enter the descriptions of desired circuitry at the logic level. However, the operation of describing a circuit at the logic level can be very time-consuming. With the very tight time-to-market schedules required by today's digital systems, facilitating the designers' creative process has become of prime importance. Therefore, there is a need for methods to efficiently describe digital

systems. From the beginning, the goal of high level synthesis has been to allow one to describe digital systems at a very abstract level, thus eliminating the tedious description at the logic level.

Typical high level synthesis systems operate on behavioral descriptions, which specify the operations the target system is to carry out, without the need to specify in full detail the logic circuit that will implement the target system. The role of high level synthesis systems is therefore to operate on the abstract specifications given by designers and generate a logic description that satisfies those initial functional requirements. Logic descriptions produced by high-level synthesis systems are in general simply blueprints for the required functionality. As a result, logic synthesis is typically used after high-level synthesis to optimize the logic descriptions.

Starting at the end of the 1970's, many systems have been created to address the need for more abstract descriptions, and were typically targeted at very specific applications. MIMOLA was one of the first such system [Zim79]. HAL was targeted at the synthesis of telecommunication ASIC circuits [PKG86]. The Cathedral system is aimed at Digital Signal Processing (DSP) applications [NvMM88]. Other high level synthesis systems developed in that period include the *System Architect's Workbench* [TDW+89], the *Yorktown Silicon Compiler* [BCD+88], USC's ADAM system [PPM86], UC Irvine's synthesis system [BG87], CADDY/DSL [CR89]. At Stanford, *Hercules* and *Hebe* were developed as the high-level end of the *Olympus Synthesis System* [Ku91].

1.1.3 Physical design

The usefulness of high-level and logic level synthesis hinges on the availability of tools that generate the geometric description of the target circuits. These tools are called physical design systems. The geometric description they produce is typically used to drive the physical implementation, either because it directly represents fabrication masks (*e.g.* in semi-custom designs), or it specifies which actions to take to create the final circuit (*e.g.* which fuses to blow in a FPGA).

The earliest physical design tools were targeted at automatic layout systems for gate arrays and standard cells, the Engineering Design System of IBM being among the

first such systems [San87]. Because of its computational complexity, the physical design problem was quickly divided into smaller, more amenable subproblems. A current customary subdivision includes partitioning (and floorplanning), placement, global routing and local routing. Placement involves the positioning of geometrical objects in a two-dimensional plane. Routing is related to the creation of interconnection links between placed objects. Since placement and routing are themselves computationally complex, the goal of partitioning is to isolate smaller circuit portions on which placement and routing can be applied. Among partitioning techniques, Kernighan and Lin's heuristic for bipartite partitioning [KL70], and Kirkpatrick *et al.*'s simulated annealing method [KGV83] are the best known. Current research targets multiple way partitioning [YCL91]. Solutions to the placement problem include using slicing structures [Ott82], min-cut algorithms [Bre77, Lau80] and rectangular dualization methods [HSM82]. Interest in wire routing started in the early 1960's [Lee61]. A major milestone was the idea of reserving *channels* for routing, introduced by Hashimoto and Stevens [HS71]. Channel routing was extensively studied during the 1980's [Bur86]. Global routing is also an important component in today's physical design systems [KMS86]. With the availability of placement and routing tools, symbolic layout and automatic cell generation have become a viable path from semi-custom designs logic descriptions to layout [New87, Uv81, MH87, MD88, HHLH91].

1.1.4 The Olympus Synthesis System

Most synthesis systems nowadays incorporate both high-level and logic synthesis operations. *Olympus* is an example of the current trend [DKMT90]. It is composed of an ensemble of tools that can be classified as members of either the high-level or logic synthesis domains. Figure 1.1 is a diagram representing the current tools in *Olympus*.

The entry point of the system is a behavioral description in a hardware description language, HardwareC. The behavioral description is processed by program *Hercules*, which parses the description, does some compiler-like optimizations (*e.g.* dead code elimination, constant propagation), and generates SIF (Sequencing Intermediate Format), an intermediate representation of the operations and data dependencies implied by the

original specification. This intermediate representation can be simulated by means of *Ariadne*, which allows validation of the HardwareC description against the desired behavior. *Hebe* operates on the same representation (SIF), performing resource allocation, scheduling and control generation. *Hebe* allows *design space exploration*, which lets users of the system experiment with different design decisions and evaluate their impact on the final implementation. *Hebe* produces a logic description in SLIF (Structure/Logic Intermediate Format), which consists of Boolean equations, delay elements, and possibly hierarchy.

The logic description is passed to *Mercury*, a logic synthesis framework. *Mercury* allows for simple logic operations like constant propagation (*sweep*) and merging of logic equations (*elimination*). It contains a simple two-level optimization procedure, called SANKA [Fil91], based in part on the POP algorithm [DHNSV85]. *Mercury* also allows users to operate on the hierarchy of the logic descriptions. *Mercury* contains an interface to other logic synthesis systems (*e.g.* MISII and Autologic^{TM}). It allows a logic description written in SLIF to be optimized by means of these other synthesis tools, and then to be read back. *Mercury* contains an event-driven logic simulator. The logic simulator has different built-in timing models: load-dependent with assumed loads (for technology-independent logic descriptions), load and drive-dependent (for technology-dependent logic descriptions), and zero-delay combinational logic (for compatibility with the results of *Ariadne*).

Once logic descriptions have been optimized, they are transformed by *Ceres*, the subject of this thesis. *Ceres* is a technology mapper which binds technology-independent logic descriptions to target technologies. The target technologies are represented by annotated libraries, which contain descriptions of available logic gates. The logic description of each gate includes its logic function, and other parameters like area, input capacitance, output drive, etc. *Ceres* shares its internal data representation with *Mercury*, and accordingly incorporates some of the functionality of *Mercury* (*e.g.* constant propagation). In addition, for the more specific tasks involved in technology mapping, *Ceres* includes algorithms for logic decomposition, logic equivalence verification for both completely and incompletely specified functions, and area/delay optimization. *Ceres* ultimately transforms technology-independent descriptions, creating new representations based on target

libraries. The resulting netlists are then passed to physical design tools for placement and routing. Many physical design tools are currently available both from industry and academia, and therefore are not included in the *Olympus* system. This dissertation presents the ideas and algorithms implemented by program *Ceres*.

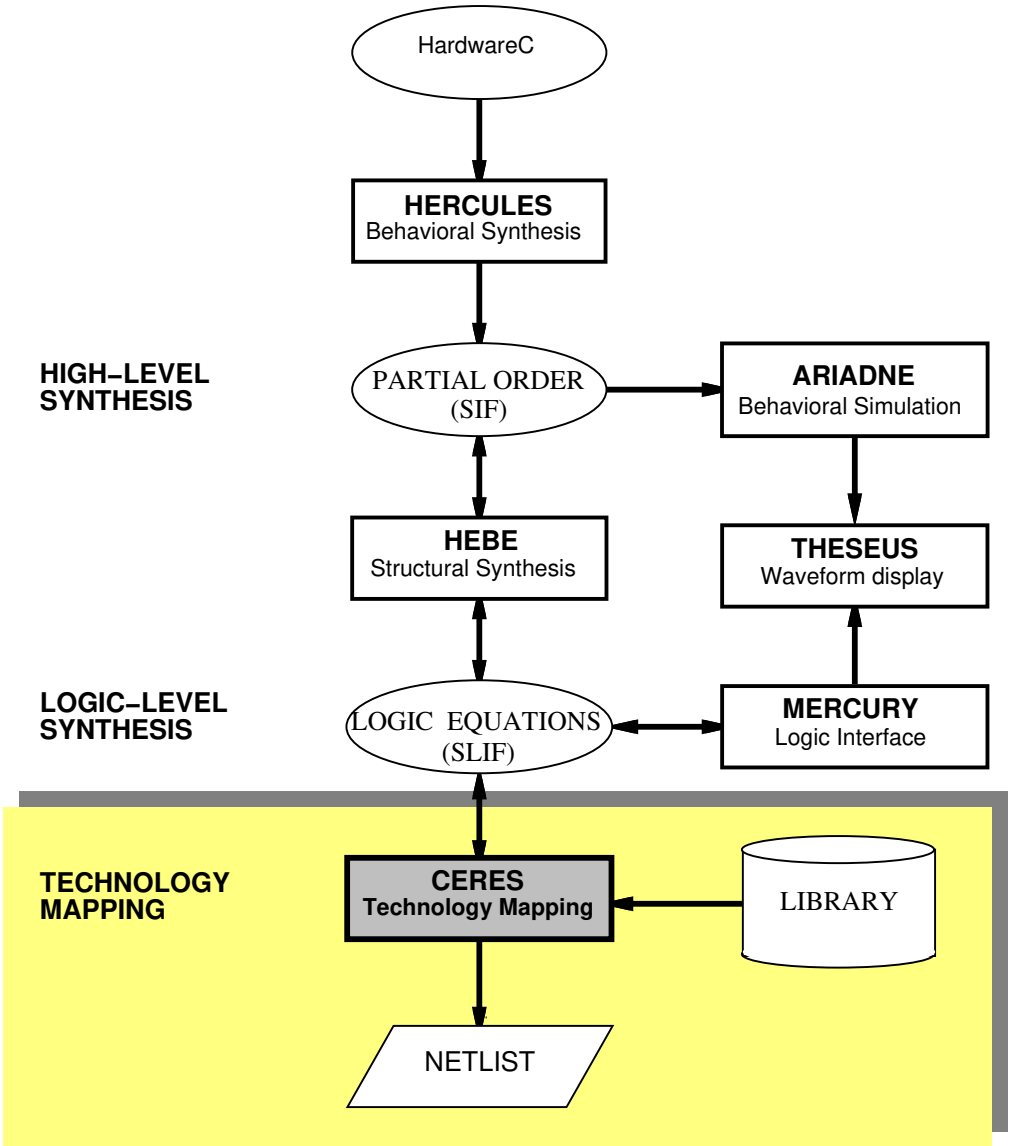


Figure 1.1: The Olympus Synthesis System

1.2 Terminology and notation

Descriptions of automatic synthesis systems have a very specific vocabulary and terminology depending on the level of abstraction. Each of the three levels of abstraction presented earlier, high-level synthesis, logic synthesis and physical design, has its own representation and modeling paradigms. Since in the remainder of this thesis, we focus on the logic level of automatic synthesis of digital circuits, we will define the terminology which will be used throughout.

Customary representations at the logic level involve Boolean algebra, and relations between Boolean functions. Technology mapping involves the technology-dependent set of operations in logic synthesis systems. Technology-dependent and technology-independent operations are closely linked. Therefore, the terminology and notation we use in this thesis are strongly related to those used in descriptions of technology-independent logic synthesis operations. In subsequent chapters of this thesis, we rely on the following definitions and assumptions:

Definition 1.1 $\mathcal{B} = \{0, 1\}$ *is the Boolean domain.*

We denote *Boolean variables* by subscripted strings (*e.g.* x_i, x_j, y_k). *Boolean variables* can take on values from the set $\mathcal{B}$.

Definition 1.2 *The* Boolean operators *'+', '*' and '!' represent the logic disjunction (OR), conjunction (AND) and inversion (NOT), respectively. By default, we denote '*' by a white space. '!' is also represented by an appended apostrophe "'".*

The *phase* of a *Boolean variable* x_i indicates whether the value of x_i is to be used directly or complemented (inverted). This is denoted by a 0 or 1 superscript: x_i^1 means the direct phase of x_i, and x_i^0 means the complemented phase. When used without superscript, x_i means x_i^1. Symbol x_i^0 is also represented by x_i' and $\overline{x_i}$.

Definition 1.3 *A* Boolean function $\mathcal{F}(x_1, \ldots, x_n)$ *is a composition of* Boolean operators. *A single output* Boolean function *is a function* $\mathcal{F} : \mathcal{B}^n \rightarrow \mathcal{B}$. *An* m*-output* Boolean function *is a function* $\mathcal{F} : \mathcal{B}^n \rightarrow \mathcal{B}^m$. *We denote by* x_i *the variable associated with the output of* $\mathcal{F}_i$ [1].

[1]Note that x_i cannot be one of the inputs of $\mathcal{F}_i$ (*i.e.* $x_i \notin \{x_1, \ldots, x_n\}$).

A *Boolean function* $\mathcal{F}$ is said to be *unate* in *Boolean variable* x_i if x_i appears always in only one phase in the expression of $\mathcal{F}$. $\mathcal{F}$ is said to be *positive* (or *negative*) *unate* in x_i if only x_i (or x_i') appears in the expression of $\mathcal{F}$. $\mathcal{F}$ is *binate* in x_i if the variable appears in both phases in the expression of $\mathcal{F}$.

Definition 1.4 *An* input vector $\underline{\mu}$ *of a single output Boolean function* $\mathcal{F} : \mathcal{B}^n \rightarrow \mathcal{B}$ *is one element of the domain* $\mathcal{B}^n$. *Input vector* $\underline{\mu}$ *of* $\mathcal{F}(x_1, \ldots, x_n)$ *is represented by a product* $\underline{\mu} = \prod_{i=1}^{n} x_i^{p_i}$ *where* $p_i \in \{0, 1\}$ *is the phase of variable* x_i.

Definition 1.5 *We define the* ON-set *of Boolean function* $\mathcal{F}$ *as the set of input vectors* $\{\underline{\mu_j}, j = 1, \ldots, \mid ON \mid\}$ *of* $\mathcal{F}$ *for which* $\mathcal{F}(\underline{\mu_j}) = 1$. *The input vectors of the* ON-set *are also called minterms. The* OFF-set *of Boolean function* $\mathcal{F}$ *is defined as the set of input vectors* $\{\underline{\mu_j'}, j = 1, \ldots, \mid OFF \mid\}$ *such that* $\mathcal{F}(\underline{\mu_j'}) = 0$.

Definition 1.6 *The* image *of* A *through* $\mathcal{F} : \mathcal{B}^n \rightarrow \mathcal{B}^m$ *is the subset of* $\mathcal{B}^m$ *reached by* $\mathcal{F}(A)$, *where* A *is a subset of* $\mathcal{B}^n$. *The* range *of* $\mathcal{F}$ *is the image of* $\mathcal{B}^n$ *through* $\mathcal{F}$ *(*$A = \mathcal{B}^n$*).*

Boolean functions can be *completely* or *incompletely specified*. Completely specified functions follow the previous definition. Incompletely specified functions have their range $\mathcal{B}$ extended to $\widetilde{\mathcal{B}} = \{0, 1, X\}$, where X means either 0 or 1.

Definition 1.7 Don't care sets *represent the conditions under which a Boolean function takes the value* X.

Don't care conditions occur in Boolean logic either because some combination of inputs of a Boolean function never happen (the domain of $\mathcal{F}(x_1, \ldots, x_n)$ is smaller than $\mathcal{B}^n$), or because some outputs of $\mathcal{F}$ are not observed. The first class of *don't care* conditions is called *controllability don't cares*, and the second class, *observability don't cares*. Both are described more thoroughly in Chapter 6.

Definition 1.8 *A Boolean function* $\mathcal{F}$ *is* implementable *by a Boolean function* $\mathcal{G}$ *if for each input combination either* $\mathcal{F}$ *and* $\mathcal{G}$ *have the same value, or else* $\mathcal{F}$ *has a value* X. *Alternatively, we say that* $\mathcal{G}$ *is* compatible *with* $\mathcal{F}$.

Definition 1.9 *The* support *of a Boolean function $\mathcal{F}(x_1, \ldots, x_n)$ is the set of variables $\{x_1, \ldots, x_n\}$ used in the expression of $\mathcal{F}$.*

Definition 1.10 *A* Boolean network *is an ensemble of interrelated Boolean functions. We represent a Boolean network by a set of N Boolean variables $\mathcal{V} = \{y_1, \ldots, y_N\}$ and a set of Boolean functions $\{\mathcal{F}_1, \ldots, \mathcal{F}_N\}$ such that $\mathcal{N} = \{y_i = \mathcal{F}_i, i = 1, \ldots, N\}$ where $y_i = \mathcal{F}_i$ represents an assignment of a single-output Boolean function for every Boolean variable. Functions $\mathcal{F}_i, i = 1, \ldots, N$, have $K_i \leq N$ inputs (i.e. $\mathcal{F}_i : \mathcal{B}^{K_i} \rightarrow \mathcal{B}$), each input corresponding to a Boolean variable of $\mathcal{V}$.*

Subsets of Boolean networks are also Boolean networks. Boolean networks are represented by graphs $G(V, E)$ where the vertex set V is in one-to-one correspondence with the set of Boolean variables $\mathcal{V} = \{y_1, \ldots, y_N\}$, and E is the set of edges $\{e_{ij} \mid i, j \in \{1, \ldots, N\}\}$ such that e_{ij} is a member of the set E if $y_i \in$ support($\mathcal{F}_j$). Such networks are acyclic by definition.

We respectively call fanin and fanout the in-degree and the out-degree of vertices in a Boolean network.

Definition 1.11 Primary inputs *are Boolean variables of a Boolean network that depend on no other variable (i.e. $\mathcal{F}_i$ is the identity function for* primary inputs*).* Primary outputs *are Boolean variables on which no other variable depends.*

Definition 1.12 *The* depth *of a Boolean network is the maximum number of vertices in a directed path between any primary input and any primary output.*

Definition 1.13 Collapsing *is the action of reducing the depth of a Boolean network to one.* Partial collapsing *corresponds to collapsing subnetworks.*

Definition 1.14 *The* Boolean behavior *of a n-input, m-output Boolean network is the Boolean function $\mathcal{F} : \mathcal{B}^n \rightarrow \widetilde{\mathcal{B}}^m$ corresponding to the collapsed Boolean network.*

Two Boolean networks are equivalent if there is a one-to-one correspondence between their respective primary inputs and primary outputs, and if their corresponding Boolean behaviors are compatible.

1.3 Contributions

The research presented in this thesis concerns technology-dependent transformations at the logic level in automatic synthesis of digital circuits. The goal of technology mapping, as this operation is called, is to transform arbitrary Boolean networks into networks that rely only on a predefined, restricted set of Boolean functions. The restricted set of Boolean functions is called a *library*, and represents logic gates available in some target technology. Given a cost metric defined in terms of the restricted set of Boolean functions, the quality of the transformed network increases as its associated cost decreases. Typically, the cost metric involves area and delay of the gates implementing the Boolean functions in the library. As will be described in Chapter 2, one of the key operations in the transformation process is to recognize logic equivalence between two arbitrary Boolean functions. Since comparisons are done between Boolean functions with supports labeled differently, this operation, called matching, is difficult.

The contributions of this work are two-fold. First, new algorithms for matching are introduced. Second, we present an iterative formulation of technology mapping for performance optimization.

Matching is a key operation in technology mapping systems, and we present two new classes of algorithms. The first one is based on Boolean techniques for equivalence detection. These techniques are more powerful than existing methods, and therefore produce better quality solutions. The second class of algorithms addresses the problem of matching Boolean functions in the presence of *don't care* information. This is a new, and very important addition to technology mapping operations, as it merges operations that were traditionally technology independent with technology-dependent step of logic synthesis. Better solutions can be obtained, since optimization decisions are made with a clearer knowledge of the target technology.

Performance is of utmost importance in digital circuits. We present an iterative framework based on specific transformations of mapped circuits. The effect of applying these transformations is to progressively reduce the critical delay in the implemented circuit, possibly at the expense of increased area.

All the algorithms presented in this thesis have been implemented in a software

system, *Ceres*. Results are presented at the end of Chapters 5, 6 and 7 to exemplify the value of the algorithms.

1.4 Thesis outline

The remainder of his thesis is structured as follows. In Chapter 2, we formally present the intrinsically difficult problems related to technology mapping. We then review previous approaches to solving these problems. In particular, we briefly review rule-based and algorithmic systems, explaining how these solutions deal with the computationally complex issues. We conclude the chapter by comparing these two approaches, and discussing their respective limitations.

Chapter 3 introduces the general divide-and-conquer approach used in our solution to technology mapping. We describe the basic steps of technology mapping: partitioning, decomposition, and covering. We introduce a new method for solving the covering problem.

In Chapter 4 we focus on our contributions to the efficient solution of the matching problem, a key element of the covering problem. We introduce Boolean matching, and explain how the quality of the solutions it finds is improved over that of previous systems. We briefly review binary decision diagrams (BDD), as they constitute the major data structure of the logic operations involved during Boolean matching. We first present a simple algorithm for matching which can be used for establishing the equivalence of very small logic functions in the presence of *don't care* information. This simple algorithm exemplifies the computational complexity of Boolean matching, and indicates the need for careful search space reduction when processing large logic functions.

In Chapter 5, we explain how to reduce the potentially very large search space. The method is based on binary decision diagrams (BDD), and on the use of logic symmetry. We present a method for computing and exploiting logic symmetry during Boolean matching. We justify the usefulness of logic symmetry as an efficient way of reducing the search space, particularly when dealing with standard logic parts offered by semi-custom and ASIC vendors. We conclude Chapter 5 with comparative results with respect to other technology mapping systems.

Chapter 6 presents methods that consider *don't care* information during the matching operation. We introduce the *matching compatibility graph* as a way to recognize logic equivalence in the presence of incompletely specified functions. We describe how *don't care* information is derived from the subject circuit. We briefly present how the introduction of *don't cares* enhances the testability of the final implementation. We complete Chapter 6 by presenting technology mapping results using *don't cares* .

Chapter 7 is concerned with circuit performance issues. We present an ensemble of three performance enhancement techniques: gate duplication, redecomposition and partitioning modification. We also explain how the three techniques are integrated, using an iterative framework as the driving mechanism. Results of timing-driven operations conclude the chapter.

Finally, Chapter 8 concludes the thesis with a summary of the contributions, and a discussion of future extensions.

Chapter 2

Technology mapping

In this chapter, we first present a general description of the technology mapping problem. Then we formally define the technology mapping problem, focusing on the intrinsically complex aspects. Previous solutions fall into two classes: rule-based and algorithmic systems. We review the two classes of solutions, and finally conclude this chapter by comparing them and highlighting their limitations.

The goal of technology mapping is to transform a technology-independent description of a logic circuit into a technology specific representation. The technology dependent implementation should optimize some cost metric, typically delay and/or area (and sometimes power dissipation). Technology-independent descriptions, obtained, for example, after logic synthesis, are typically expressed in terms of Boolean equations (and possibly other components like delay elements). Target technologies are represented by technology specific libraries, which are comprised of individual gates implemented in the target technology. Each library is typically described by enumerating all available logic cells[1] (also called gates), where each gate description lists the logic function together with additional information related to its electrical and physical characteristics (area, input load, input to output delay, power requirements, etc.).

The fundamental operation in most existing technology mapping systems restricts library elements to single-output, combinational gates. Therefore, multi-output gates,

[1] Some technologies can be characterized by a single master cell, or by some general property common to all logic functions available, *e.g.* fanin limit. This is particularly true with some current FPGA technologies.

registers and buffers are rarely directly taken into account. The apparent limitation of this assumption is not as restrictive as it may first seem. Most systems use post-processing steps to add sequential elements and buffers. Similarly, it is possible to replace groups of gates by multi-output gates as a post-processing step, following a pass where only single-output gates are taken into account.

2.1 The technology mapping problem

Technology-independent logic descriptions of logic circuits can be cast as a *Boolean Network*, *i.e.* as a set of Boolean equations. These equations do not necessarily represent logic functions of gates in the target technology under consideration. For example, the technology-independent optimization step of a logic synthesis system might choose a complex equation (*e.g.* $x = a + b(c + d + e)$) for the description of the optimized circuit, whereas in the target technology there is no gate implementing that complex logic function (*e.g.* there may not be an OR-AND-OR gate in the library). Furthermore, even when there is a corresponding gate in the library for each equation generated by the logic optimization step, there is no guarantee that the replacement of the equation by the corresponding library element optimizes some particular figure of merit, such as area or delay, because the equations are generated without knowledge of the target library. Therefore, the goal of technology mapping is to find an ensemble of interconnected gates in the target technology that realizes the same logic function as the original circuit. In addition, the final realization should be optimized in terms of some cost metric provided by the library.

This transformation implies two distinct operations: recognizing logic equivalence between two logic functions, and finding the best set of logically equivalent gates whose interconnection represents the original circuit.

The first operation is called *matching*, and is defined as follows:

Definition 2.1 *Given two logic functions* $\mathcal{F} : \mathcal{B}^n \rightarrow \mathcal{B}$ *and* $\mathcal{G} : \mathcal{B}^n \rightarrow \mathcal{B}$, $\mathcal{F}$ *and* $\mathcal{G}$ match *if* $\mathcal{F} \equiv \mathcal{G}$ *for some input assignment* $\{\{x_1, y_{\pi(1)}\}, \{x_2, y_{\pi(2)}\}, \ldots, \{x_n, y_{\pi(n)}\}\}$, *where variables* $\{x_1, \ldots, x_n\}$ *are the inputs of* $\mathcal{F}$, *variables* $\{y_1, \ldots, y_n\}$ *are the inputs of* $\mathcal{G}$, *and* π *is a bijection from the integer domain* $\{1, \ldots, n\}$ *to itself (i.e.* π *is a permutation).*

Matching involves two steps: equivalence checking and input assignment. Checking for logic equivalence can be expressed as a *TAUTOLOGY* problem: given $\mathcal{F}$, $\mathcal{G}$ and the set of variables $Y = \{y_1, y_2, \ldots, y_n\}$, verify that $\overline{\mathcal{F}(Y) \oplus \mathcal{G}(Y)} = 1$ for all 2^n binary assignments of the variables of Y. *TAUTOLOGY* has been proven to be NP-complete [GJ79]. Input assignment is also computationally complex. In the worst case, it entails the examination of all combinations of n inputs, which represents $n!$ operations.

The second operation is called *covering*, and is defined as:

Definition 2.2 *Given a Boolean network $\mathcal{N}$ and a restricted set of Boolean functions (called a library) $\mathcal{L} = \{\mathcal{G}_i : \mathcal{B}^{K_i} \rightarrow \mathcal{B}; i = 1, 2, \ldots, L\}$, a* cover *is a network $\mathcal{M}$, represented by $\mathcal{M} = \{x_i = \widehat{\mathcal{G}}_j, i = 1, \ldots, \mid \mathcal{M} \mid; j \in \{1, \ldots, L\}\}$ where $\widehat{\mathcal{G}}_j$ is an instance of $\mathcal{G}_i \in \mathcal{L}$, $\mid \mathcal{M} \mid$ is the number of vertices in $\mathcal{M}$, and $\mathcal{M}$ is equivalent to $\mathcal{N}$.*

Covering involves finding an alternate representation of a Boolean network using logic elements which have been selected from a restricted set. Note that before covering is considered, the equivalence between elements of the restricted set and portion of the Boolean network is established using the matching operation described above. In its most general form, the covering process can be described as follows.

Consider a primary output v_o of network $\mathcal{N}$. Let $G_o \subseteq \mathcal{L}$ be the subset of library cells having the property that their output is equivalent to v_o when their inputs are either a subset of vertices in $\mathcal{N}$ (excluding v_o) or a Boolean function of a subset of variables of $\mathcal{N}$. Then the set G_o represents all matches for the primary output v_o. If a match $\mathcal{G}_k \in G_o$ is chosen, then all variables in its support must be available through some other matches that are chosen. Consider a network $\mathcal{N}'$ obtained from $\mathcal{N}$ by removing the primary output v_o from $\mathcal{N}$ and adding all the variables in the support of $\mathcal{G}_k$ as primary outputs of $\mathcal{N}'$. Then the new network $\mathcal{N}'$ must be covered as well. This operation can be applied recursively until all the primary outputs of $\mathcal{N}'$ correspond directly to the primary inputs of $\mathcal{N}$ (Figure 2.1). Given a cost associated with each library element, the optimum cover minimizes the overall cost of the alternate representation.

Let us neglect the computational complexity of finding G_o and $\mathcal{N}'$ at each recursion step. The choice of an element $\mathcal{G}_k \in G_o$ implies that the inputs of $\mathcal{G}_k$ are available as the primary outputs of the corresponding network $\mathcal{N}'$. Therefore, the selection of a

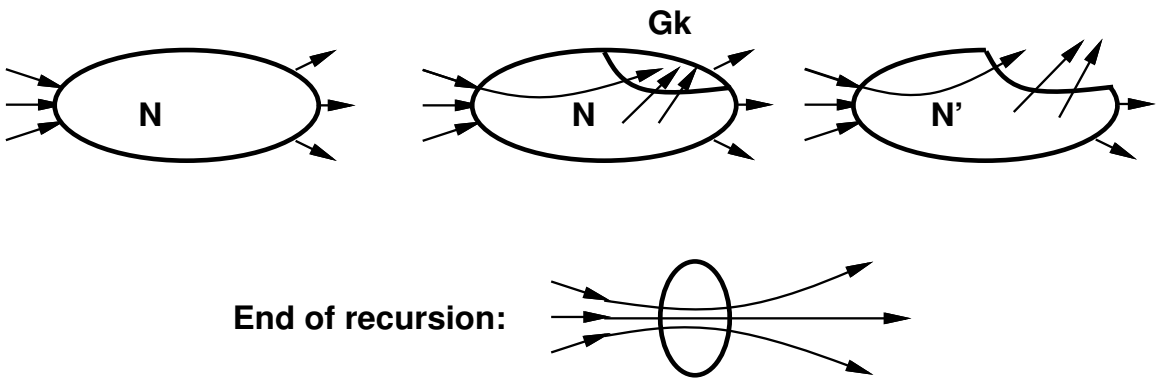


Figure 2.1: Illustration of the recursive covering definition

matching element $\mathcal{G}_k$ implies that subsequent selections must make the primary outputs of $\mathcal{N}'$ (or alternatively, the inputs of $\mathcal{G}_k$) available. If it were possible to recursively list all choices for matches $\mathcal{G}_k$, it would be necessary to describe how choosing a matching gate $\mathcal{G}_k$ in turn implies other matches which generate the inputs of $\mathcal{G}_k$. Solving the set of implications generated in this process is a type of covering problem, which has been termed *binate covering* [Rud89a], *covering with cost* [RH67], and *covering with closure*. The covering problem has been shown to be equivalent to a *SATISFIABILITY* problem, with the additional requirement that each variable in each clause to satisfy has an associated cost [Rud89a].

In the general covering formulation, $\mathcal{N}'$ does not need to be a subset of $\mathcal{N}$. That is, inputs to the chosen gate $\mathcal{G}_k$ do not need to correspond to vertices in the network $\mathcal{N}$. A *restricted covering* problem can be formulated, where the inputs to the matching gates must pre-exist in the initial Boolean network $\mathcal{N}$, *i.e.* the inputs to any matching cell $\mathcal{G}_k \in G_o$ must correspond to a vertex of the Boolean network $\mathcal{N}$. The restricted covering problem has been considered by [Keu87, Rud89a] because it is simpler to handle: no computation of $\mathcal{N}'$ is required. Under the restricted covering assumption, Boolean network $\mathcal{N}$ is defined by a constant set of vertices throughout the covering process. However, such a problem is still computationally complex, because it again involves implications among the selection of the matches, and therefore it is still a binate

covering problem.

As an example of the occurrence of binate covering in the *restricted covering* problem, let us consider Figure 2.2, which shows a simple network, $\mathcal{N}$, with a set of target logic gates, $\mathcal{L}$. We are interested in transforming $\mathcal{N}$ into a new network, $\mathcal{M}$, where each Boolean function belongs to the restricted set $\mathcal{L}$. For $\mathcal{M}$ to be equivalent to $\mathcal{N}$, the

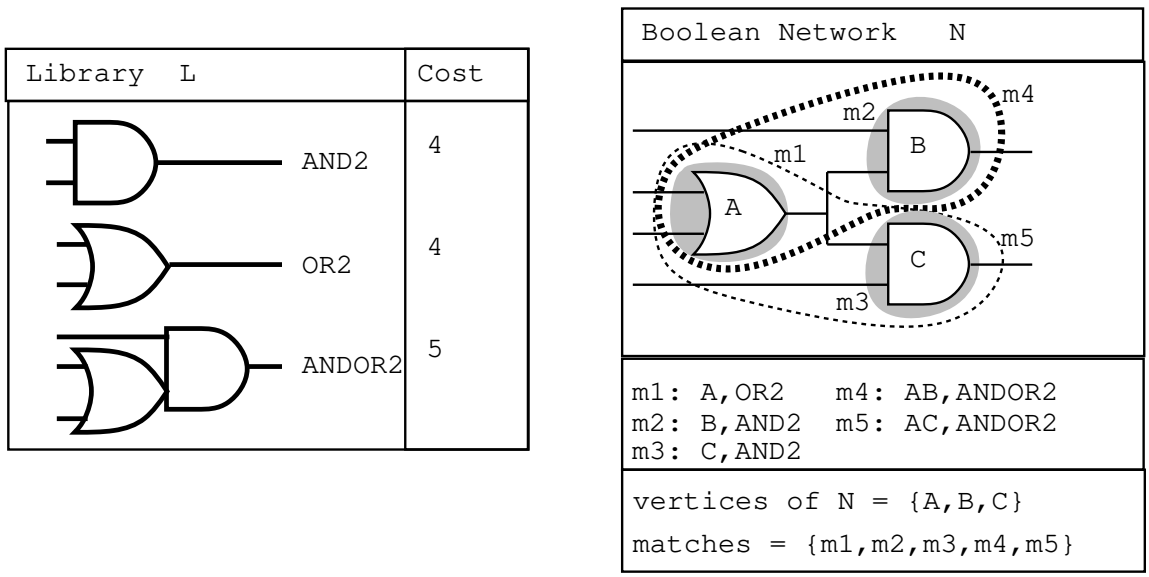


Figure 2.2: A simple covering example

logic functionality of their respective primary outputs has to be compatible. The primary outputs of the transformed network $\mathcal{M}$ must be implemented by gates selected from the library $\mathcal{L}$. Each variable in the support of these gates also must be implemented by some gate from the library, unless the variable is a primary input. This has to be recursively true, so that all variables used in $\mathcal{M}$ are defined.

Each vertex in $\mathcal{N}$ can be represented by at least one element in $\mathcal{L}$, and possibly more than one. Conversely, an element in $\mathcal{L}$ can possibly represent more than one vertex of $\mathcal{N}$. Therefore, there are usually many choices of gates in $\mathcal{L}$ that can generate a valid network $\mathcal{M}$.

For example, in Figure 2.2, function A can be represented either by a single two-input

OR gate (OR2), or together with function B (or C) by a AND-OR gate (ANDOR2). We can associate a binary variable with the choice of a particular gate cover, for example m_1 represents whether the two-input OR gate is chosen to cover A ($m_1 = 1$), or whether it is not chosen ($m_1 = 0$). Similarly, in Figure 2.2, m_4 represents the use of the AND-OR gate to cover functions A and B, and m_5 represents and AND-OR gate covering A and C.

Therefore, we can represent the requirement for A to be covered at least by one gate in $\mathcal{L}$ by the Boolean equation $(m_1 + m_4 + m_5)$. Similarly, the covering requirements for B and C can be represented by Boolean equations $(m_2 + m_4)$ and $(m_3 + m_5)$ respectively. For all vertices in the original network to be covered in the alternate representation, the three Boolean equations above must be true at the same time, *i.e.* $((m_1 + m_4 + m_5)(m_2 + m_4)(m_3 + m_5) = 1)$.

In addition to these requirements, each cover in the alternate network must have its inputs generated by other covers (unless the inputs are primary inputs). For instance, choosing a two-input AND gate to cover B requires that the output of A be available, which is the case only if a two-input OR gate covers A. The choice of the two-input AND gate covering B is represented by Boolean variable m_2, and the two-input OR gate covering A is represented by m_1. This relation can thus be expressed as $m_2 \rightarrow m_1$. Similarly, $m_3 \rightarrow m_1$. These additional conditions must be valid simultaneously with the first covering conditions. Therefore, to solve the covering problem, we must find a solution to the following Boolean equation: $(m_1 + m_4 + m_5)(m_2 + m_4)(m_3 + m_5)(m_2 \rightarrow m_1)(m_3 \rightarrow m_1) = 1$, which can be rewritten as $(m_1 + m_4 + m_5)(m_2 + m_4)(m_3 + m_5)(\overline{m_2} + m_1)(\overline{m_3} + m_1) = 1$. We can see from this equation why this is a *binate* covering problem: some variables (m_2 and m_3) appear in both their ON and OFF phases. For every cube that satisfies the equation $(m_1 m_2 m_3 \overline{m_4}\,\overline{m_5}, m_1 m_2 \overline{m_3}\,\overline{m_4} m_5, m_1 \overline{m_2} m_3 m_4 \overline{m_5}, \overline{m_1}\,\overline{m_2}\,\overline{m_3} m_4 m_5)$, the cost is calculated as the sum of the costs of variables used with a positive phase (12,13,13,10 for the example[2]). The satisfying minterm with minimum cost is chosen as the solution to the covering problem ($\overline{m_1}\,\overline{m_2}\,\overline{m_3} m_4 m_5$).

SATISFIABILITY has long been known to be NP-complete [Coo71, GJ79]. *Binate*

[2]Remember that each variable with positive phase corresponds to selecting a gate from the library.

covering, which involves the additional requirement of minimal cost is, therefore, computationally harder than *SATISFIABILITY*.

2.2 Previous work

In order to find acceptable solutions to the computationally complex problems intrinsic to technology mapping, simplifications and heuristic approaches have been used in the past. Two classes of methods exist: rule-based and algorithmic. Rule-based approaches utilize expert systems to operate on a technology rule-base. A network is modified by successively applying relevant transformation rules to each of its elements. The operation continues until either no transformation can be applied or the network has reached some acceptable state, *i.e.* the evaluation of some metric of the network gives a cost lower than some predefined value. A rule-base is created with rules that encapsulate the allowed transformations on a network as pairs of logically equivalent configurations, with one preferred element in the pair. Rules are typically very specific to the target technologies. Therefore modifying or adding rules in the rule set of a technology (or creating a rule set for a new technology) is usually not simple.

Algorithmic-based approaches use constructive methods to transform networks from generic Boolean logic into technology-specific design. They follow a sequence of systematic operations that change the original logic description into an implementation that conforms to the target technology.

There are currently many systems that support technology mapping to produce actual implementations of circuits. Both ruled-based and algorithmic-based categories are well represented. Often, the technology mapping operation is one step in a more general logic synthesis system. We will now review previous logic synthesis systems, concentrating on the technology mapping aspect. We will first consider rule-based systems, then follow with algorithmic-based ones.

2.2.1 Rule-based systems

One of the first systems to tackle the technology mapping problem was LSS [DJBT81, DBJT84, JTB+86], which started at IBM in the early 1980's. LSS uses local transformations to first simplify a logic description, and then expresses it in terms of the target technology. Rules are used as the underlying transformation mechanism. For example, Figure 2.3 represents two such rules (adapted from [DBJT84]). The first rule, *NTR4*, indicates that two successive AND gates with one input in common are equivalent to one AND gate with a non-inverting buffer in front. The second rule, *NTR3*, similarly indicates that two connected AND gates are equivalent to three connected AND gates, two of them with one common input. Rules *NTR3* and *NTR4* represent two different types of operations, one that removes logic gates (*NTR4*), and one that adds logic gates (*NTR3*). On any given set of adjacent gates, every applicable rule is evaluated by estimating the cost of the resulting circuit, assuming the rule was applied. Rules which improve the cost of the circuit are accepted as soon as they are found: it is a greedy operation.

Initially, in LSS, the target technology could be described only with simple gates [DJBT81]. Eventually, more complex gates were introduced, but recognizing the complex gates (*e.g.* XORs) involved adding many rules to deal with all possibilities [DBJT84]. LSS has grown over the years from a simple logic synthesis program to a system capable of taking testability and timing information into account during technology mapping.

TMS, another rule-based technology mapping system, was also developed at IBM during the same period [GLWH84]. Specifically designed to aid the transformation of circuits from one technology to another, TMS relied on an intermediate representation, *GLN* (General Logic Notation). Rules represented allowed transformations from one logic configuration to another, but did not allow for transformation of some simple gates into more complex ones. For example, parity functions could not be extracted from interconnections of nands or exclusive-ors.

TRIP is a technology mapping system developed at NEC [SBK+87] which does both technology mapping and logic optimization. To detect equivalence between circuit configurations and rule patterns, TRIP uses simulation to compare the initial circuit to the modified circuit. TRIP also relies on user-aided partitioning to keep the size of the

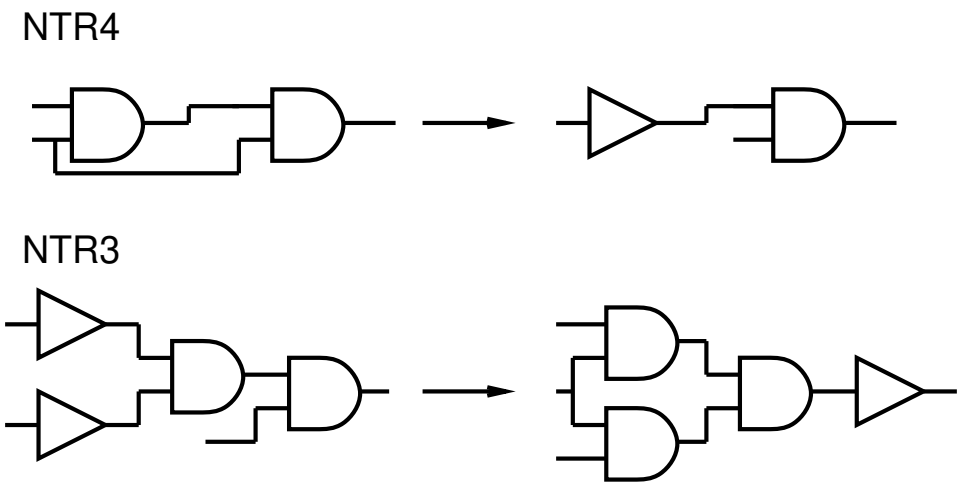


Figure 2.3: Two transformation rules from LSS

designs manageable.

LORES/EX was developed at Mitsubishi [ISH+88], and bears some similarities to TRIP. LORES/EX differs from TRIP in that it is a logic synthesis system which uses different classes or rules during different phases of operation. In particular, it initially uses *standardizing* rules, which change the logic representation into a pseudo-canonical form, where only certain gate configurations are allowed. This has the advantage of reducing the number of patterns the rules need to recognize, and therefore decreases the size of the rule-base and simplifies the process of applying the rules. Since run time grows rapidly with the size of the descriptions, LORES/EX relies on partitioning to reduce the size of the circuits it applies the rules on.

Recently, Autologic, a commercial ruled-based system for logic synthesis and technology mapping, was released by Mentor Graphics. Autologic relies on *signatures* to recognize logic equivalence between different configurations. *Signatures* are in essence truth tables. They are extracted from the portion of the network to be replaced, and compared to those of the library elements. The use of *signatures* implies that rule application in Autologic is based on Boolean comparisons. However, since the entire truth table is represented, its application is restricted to logic functions with a small number of inputs.

Socrates, originally developed at General Electric [GBdGH86] is a rule-based system that exists in a class of its own[3]. *Socrates* stands apart from the other rule-based systems because it is a hybrid method between rule-based and algorithmic-based. It is a complete system for logic optimization where the technology-independent portion is algorithmic-based, and the technology-dependent portion is rule-based. Therefore, for the technology mapping classification, we categorize it as a rule-based system. *Socrates* introduced a new concept in rule-based systems, the idea of a state search. All rule-based systems reviewed previously are greedy. They apply a transformation rule only if it lowers the cost metric of the network. Consequently, it is possible for these systems to get stuck in local optima. *Socrates* uses the state search as a way to avoid that problem. The state search is a mechanism which allows the system to systematically evaluate the application of a succession of rules, and choose to apply a rule with immediate negative impact if it leads to a better solution after further rule transformations. The state search is characterized by its breath B, *i.e.* the number of rules that are evaluated in parallel, and by its depth D, *i.e.* the number of successive rules evaluated before a decision is made. In theory, by choosing $D = \infty$ and $B =\mid \mathcal{RB} \mid$ (the size or the rule-base), all possible rule applications could be considered, with the global optimum being selected in the end. In practice, since the size of the state search grows as B^D, the breath and depth of the state search is limited. But the method still makes it feasible to overcome local optimum solutions, possibly finding a better final realization.

2.2.2 Algorithmic-based systems

The second class of systems is based on algorithmic approaches. We review this type of approaches in detail, since the technology mapping system presented in this dissertation is algorithmic-based. In particular, we carefully explain pattern matching, which was the main method used for establishing logic equivalence in previous algorithmic-based systems, to contrast with Boolean matching, the new method proposed in Chapters 4, 5 and 6.

The idea of algorithmic based technology mapping started from the realization that

[3] *Socrates* was integrated into the initial synthesis system offered by Synopsys.

logic function matching is similar to code generation in software compilers [Joh83, TJB86]. For both compiling and technology mapping, an initial description must be transformed into an expression involving only a restricted class of operations. In the most general situation, the initial description is a directed acyclic graph (DAG) in which nodes represent arbitrary operations, and edges represent data dependencies. Then, the final description is a DAG where the operations attached to nodes are limited to those belonging to the restricted class. The final expression should be optimum; that is, given a cost associated with each operation in the restricted class, the total cost of the final expression should be minimum.

In the case of software compilers, the initial description is an abstract syntax tree, the restricted class is the target computer's instruction set, and the optimum representation implies the fastest execution of the resulting code. In the case of function matching in Boolean networks, the initial description is a Boolean function, the restricted set is the set of logic gates available in a target technology, and the optimum representation gives the smallest delay between primary inputs and primary outputs. Kahrs showed that library selection in silicon compilers is analogous to code generation in software compilers, and he presented a simplified directed graph matching algorithm that covered abstract syntax trees with parameterized library elements [Kah86].

In the area of software compilers, code generation has received considerable attention [Flo61, SU70, AJ76, BS76, AJU77]. DAG matching (which in software compilers appears during the generation of code for expressions with common subexpressions) has been shown to be computationally complex [BS76, AJU77, GJ79]. Therefore, an often used simplifying assumption for code generation algorithms is to assume that only trees are processed rather than DAGs. Since this is a crucial step in compilers, tree matching has been extensively studied [And64, Nak67, Red69, AJ76, HO82].

In the area of technology mapping, in 1987 Keutzer presented a tree matching based algorithm [Keu87] that relied on the tree matching system *twig* which was developed for compilers [AG85]. The code generator-generator *twig* was designed in 1985 by Aho *et al.*, as an efficient method for tree matching in compilers [AG85, Tji85, AGT89]. *Twig* uses a method based on a tree pattern-matching algorithm proposed by Hoffman [HO82], coupled with dynamic programming [Bel57, BD62, AJ76]. Each tree is described by a

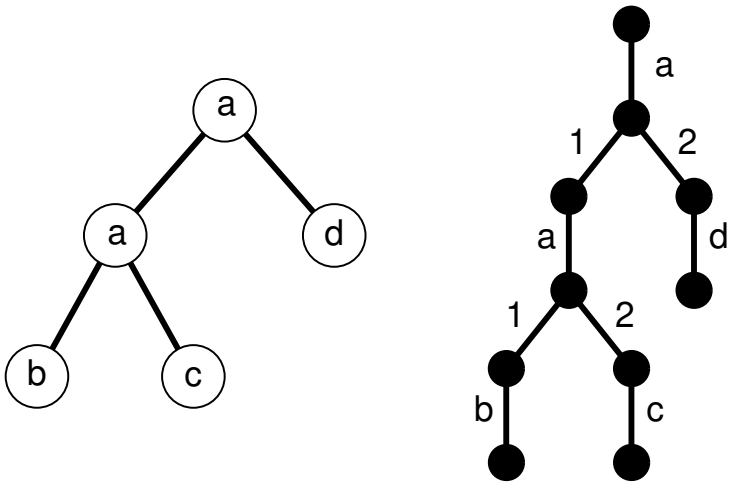


Figure 2.4: Tree pattern and associated trie (adapted from [HO82])

set of strings where each path from the root of the tree to each leaf is represented by a string in which node symbols alternate with branch numbers. For example, the tree in Figure 2.4 can be represented by the set of strings $S = \{a1a1b, a1a2c, a2d\}$, assuming the branches in the tree are numbered from left to right.

In the technology mapping domain, each library element is represented by a target tree, each with its associated strings. The set of strings derived from all target trees is then used to create an Aho-Corasick string-matching automaton, *i.e.* there is a single matching automaton built for a given library. For example, the two trees in Figure 2.5 form the following pattern set:$\{$*+1v, +2!1+1v, +2!1+2v, +2v*$\}$. Each string represents a branch of one or more trees (*e.g.* $+1v$ represents branches *t1.1* and *t2.1* of trees *t1* and *t2*). The strings in the pattern set are then used to generate the matching automaton shown in Figure 2.6. Initially, the automaton has a single state, 0. Then, starting from the 0^{th} state, each string is processed in turn, one character at a time, a new state being added for each of the characters that is not recognized by the automaton being built (*e.g.* states 1,2,3 are added when processing characters *+,1,v* of the first string *+1v*). Each edge between two states is annotated with the character processed when the second state is added. Each state corresponding to the last character of a string is annotated

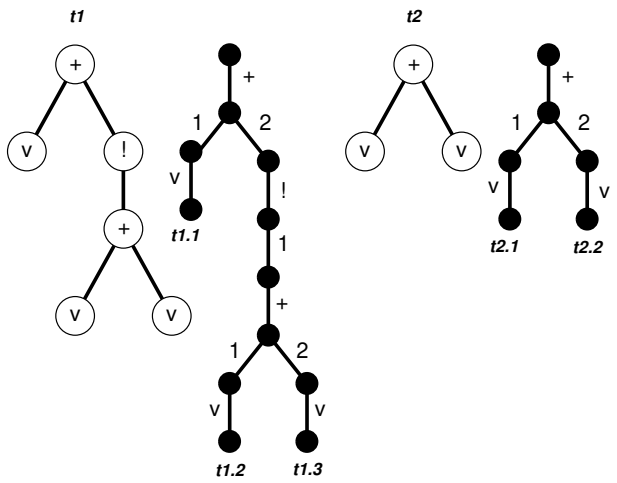


Figure 2.5: Two trees and their associated tries (adapted from [AGT89])

with an *output function* representing the matched string (*e.g.* state 3 recognizes string $+1v$, which represents branches *t1.1* and *t2.1* of trees *t1* and *t2*). When all strings from the pattern set are processed, the automaton is revisited to generate the *failure function*. This function indicates the state to go to when the running automaton reaches a state where the processed character does not correspond to any outgoing edge. The failure function is generated as follows. The automaton, which has a tree structure at this point, is traversed using a breadth-first search. All the states s_i reached by the 0^{th} state are initially annotated with $f_i = 0$, where f_i is the failure function at state s_i. Then, for each state s_j reached by state s_k on input character φ, the failure function f_j is set to point to the state reached by f_k with character φ. For example, in Figure 2.6, the failure function of state s_7 is $(f_7 = 1)$, since $(f_6 = 0)$ and s_0 reaches s_1 on input "+". As the failure functions are created, output functions of states with $f_j \neq 0$ are updated with the output functions of the state corresponding to f_j. For example, when state s_9 in Figure 2.6 is processed, it is found that $f_9 = 3$. Therefore, the output function of s_9 is augmented with the output function of s_3 (*t1.2* is augmented with *t1.1, t2.1*).

The matching automaton is constructed once for any given library. We now consider its use in recognizing trees. The process uses both top-down and bottom-up traversals.

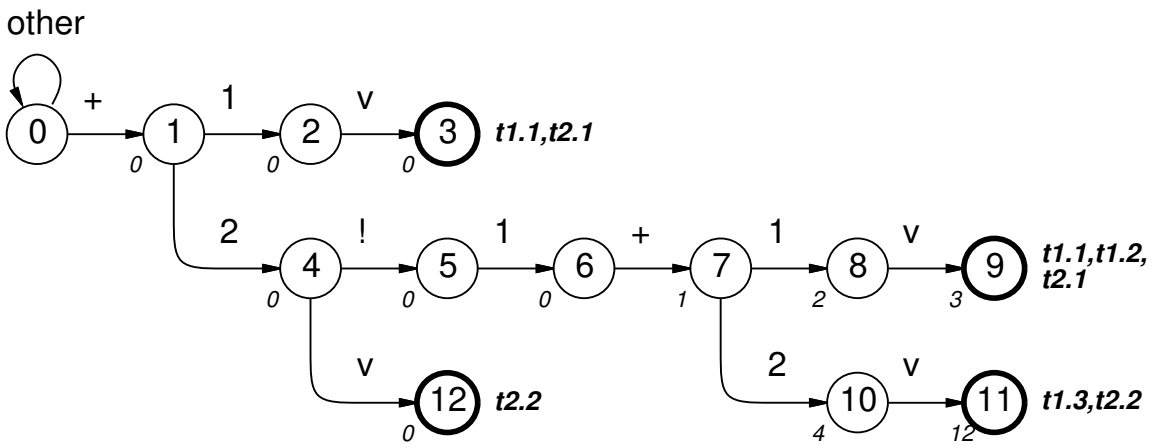


Figure 2.6: Aho-Corasick matching automaton (adapted from [AGT89])

Top-down traversal is used for detecting trees (with the Aho-Corasick automaton), while bottom-up traversal finds the best tree representation for a particular subtree (based on dynamic programming).

For a given tree to match (or subject tree), each node of the tree is processed in a depth first manner through the matching automaton, the path from the root to the processed node being encoded in a string as explained above. Each time the automaton reaches an accepting state, the current node of the tree being processed is annotated with the path that leads to it, together with the pattern tree(s) to which the accepted path belongs to. When the subtrees under a node accept the entire pattern set of a pattern tree $\mathcal{T}$, then that node matches the pattern tree $\mathcal{T}$. For each node in the tree, the best pattern tree is chosen among all the matching trees. For bottom up tree traversals, it has been shown that this leads to the selection of the optimum set of covering patterns [AJ76]. This is called dynamic programming.

Keutzer adapted *twig* to recognize logic patterns in Boolean networks [Keu87]. He introduced the idea of representing the Boolean network by a forest of trees, where the trees are *Boolean functions* expressed as interconnected two-input NAND gates. To that end, he partitioned the network at multi-fanout vertices. Library elements were similarly

represented by NAND2 trees. The representation of the logic network and library by NAND2 trees made possible the use of *twig*. From a software compiler point of view, this tree representation eliminated expressions with common subexpressions, which is equivalent, in a graph theoretical framework, as reducing the complexity of the *binate covering* problem.

Figure 2.7 shows a typical set of patterns for matching a library.

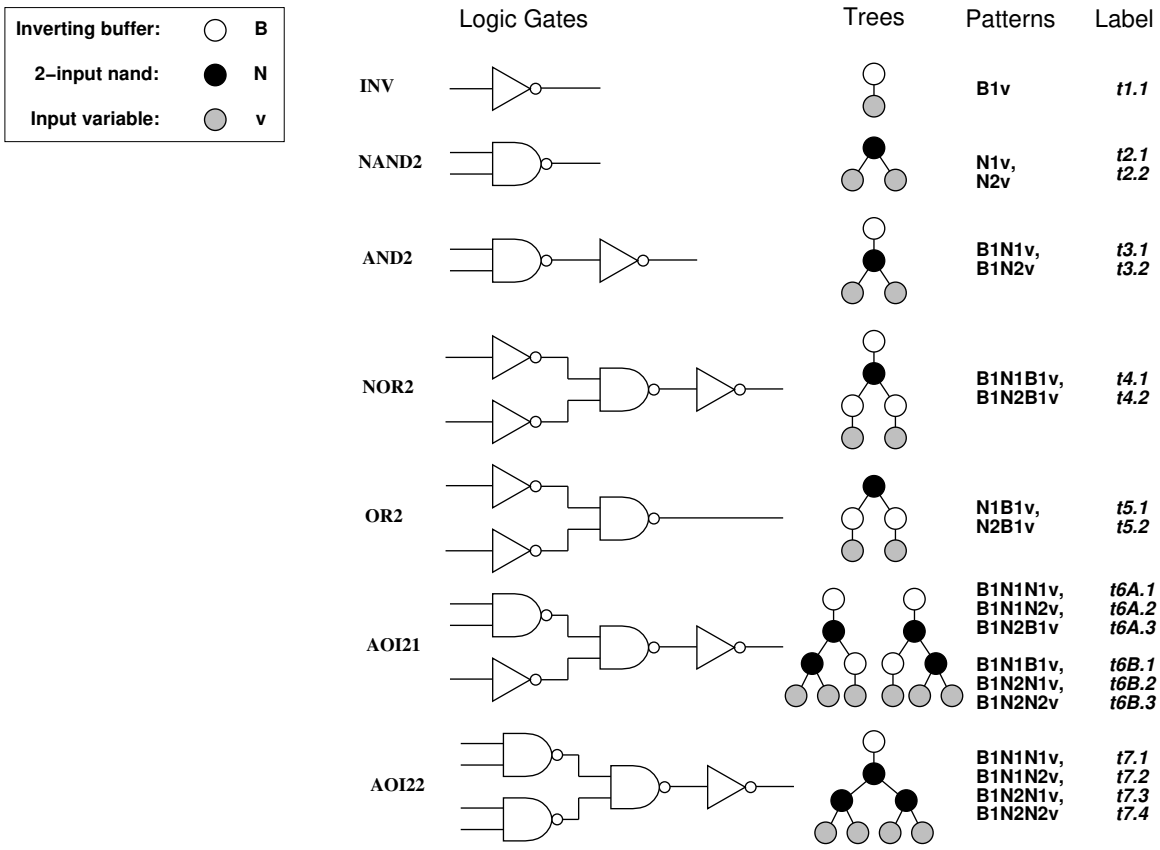


Figure 2.7: Patterns for a simple library

These patterns represent a NAND decomposition of the logic functions representing the gates of the target library. It is assumed that the technology mapping system generates all possible patterns for each logic expression representing a gate in the library. Note in particular that gate AOI21 has 2 representative patterns. Figure 2.8 represents the associated automaton. Each accepting state in the automaton recognizes a set of patterns representing portions of target trees (*e.g.* state 27 recognizes string *t2.1*, which represents half of a NAND2 gate).

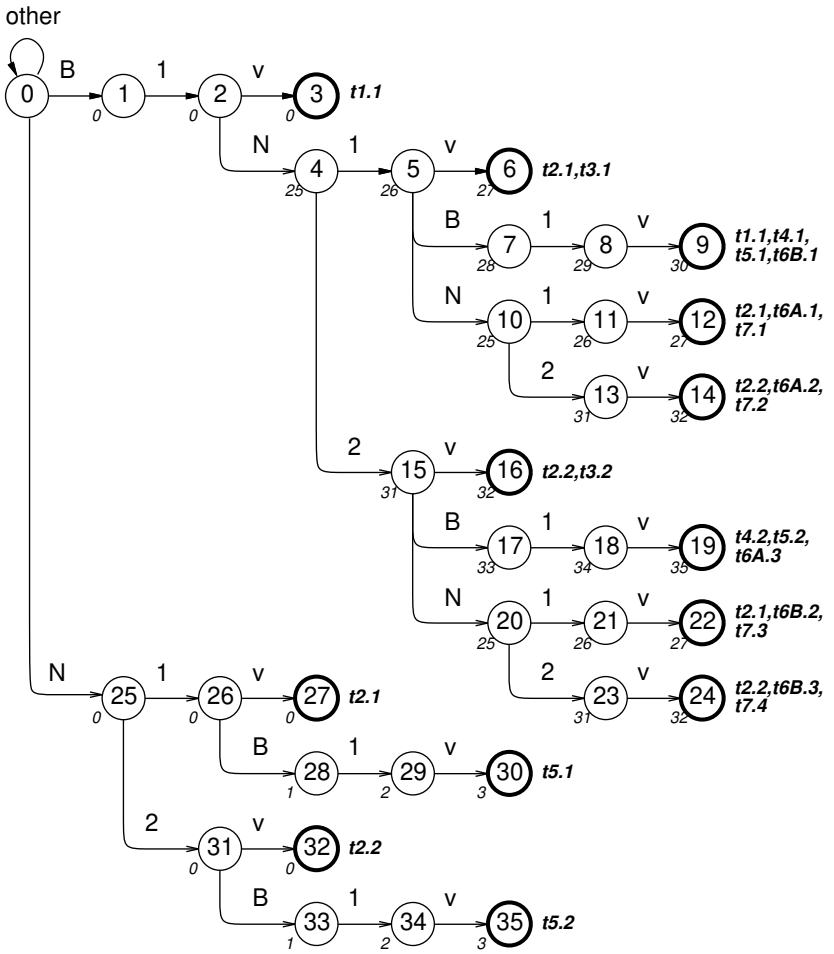
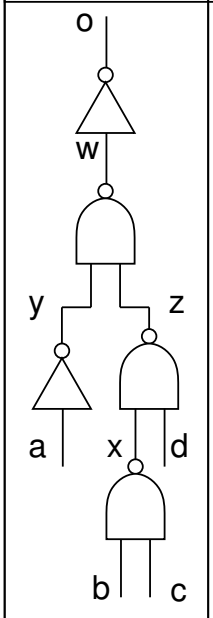
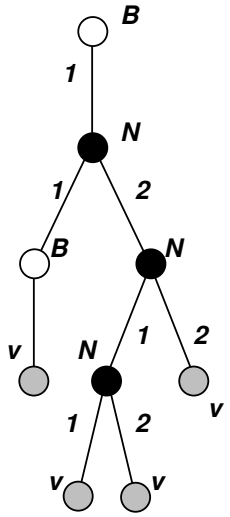


Figure 2.8: Aho-Corasick automaton for patterns of a simple library

Given a technology-independent network to transform into a target technology, the *matching* operation becomes a problem of finding library trees equivalent to tree structures in the network. *Twig* was used to match the network trees with library trees. For example, Figure 2.9 shows how a NAND2-decomposed network is represented by a tree, called the subject graph. The strings generated from the subject graph are passed to the automaton in Figure 2.8. At each node of the tree, if all strings of a pattern are detected, then that pattern is a match. It is assumed that a NAND2 gate is always available in the library, therefore there is guaranteed to be at least one match at each node of the tree. Each match is evaluated as the cost of the matching gate plus the cost of the best matches at the inputs. In the example of Figure 2.9, all nodes have a single matching gate except for node o, which has three matches.





Vertex	Match	Gate	Cost
x	*t2*	**NAND2(b,c)**	**NAND2**
y	*t1*	**INV(a)**	**INV**
z	*t2*	**NAND2(x,d)**	**2 NAND2**
w	*t2*	**NAND2(y,z)**	**3 NAND2 + INV**
o	*t1*	**INV(w)**	**3 NAND2 + 2 INV**
	t3	**AND2(y,z)**	**2 NAND2 + AND2 + INV**
	t6B	**AOI21(x,d,a)**	**NAND2 + AOI21**

Figure 2.9: Example of pattern-based covering

Twig uses dynamic programming to find the best set of library elements to represent the subject tree $\mathcal{ST}$. The algorithm operates on the tree from the leaves of $\mathcal{ST}$ towards the root. At each vertex v of the $\mathcal{ST}$, all library patterns isomorphic to sub-trees rooted at v are detected, and the library element yielding the best cost is selected. The cost of vertex v is calculated as the cost of the matching library element plus the cost of all vertices of $\mathcal{ST}$ corresponding to inputs of the matching pattern. It is assumed that there is always at least one matching pattern for any vertex v of $\mathcal{ST}$. The use of dynamic programming in *twig* ensures that each tree in the network is optimally covered by library elements. It is important to note that NAND2 tree decompositions are in general not unique. Therefore, the quality of the overall solution depends on the particular NAND2 decomposition of a network tree, as well as on the partition of the network into trees.

Keutzer's approach, although very efficient, was limited by the very operation that made it effective. NAND tree matching did not allow recognition of gates with repeated literals such as exclusive-ors (XOR), multiplexers, and majority gates. Another limitation is centered around generation of complement signals. In logic circuits, complements of signals play an important role. It is sometimes better to complement a signal, and propagate the new value to the fanouts of the original signal. The DAGON formulation in terms of NAND2 pattern trees made it difficult to change the polarity of signals, because inverters were not easily added to the strings processed by the string automaton.

In MIS, Rudell built on the ideas of Keutzer by looking at graph-matching [DGR+87, Rud89a]. He proposed an algorithm for graph-matching that relies on tentative binding between pattern graphs and the subject graph, using back-tracking when subject graphs failed to match complete pattern graphs. The graph-matching algorithm was applied only to leaf-DAG patterns, that is, graphs where only leaves can have outdegree greater than one (Figure 2.10).

Given the set of matches found by the graph-matching algorithm for each node in the Boolean network, Rudell studied how to choose the best set of matches, that is, the covering problem. Since graph-covering is NP-complete, Rudell used a tree-covering approximation. Invoking the *principle of optimality*, he based his covering technique on dynamic programming, which is guaranteed to find the optimum solution for trees. Rudell also addressed the problem of modifying the polarity of signals to find better

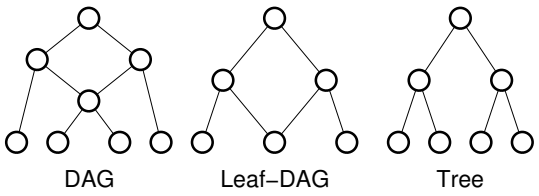


Figure 2.10: Example of a DAG, a leaf-DAG and a tree

solutions. He proposed an *inverter-pair* heuristic, which consisted of adding inverter pairs at the output of each gate in the original network. This allowed phase assignment to be done automatically by the dynamic programming-based covering. A special gate consisting of an inverter pair with no cost was also added to the library, making possible the elimination of unnecessary inverter pairs. The *inverter-pair* heuristic produced the best phase assignment for every single-fanout vertex. However, this did not guarantee optimality because it is not possible to locally find the best phase assignment at multi-fanout vertices. For all systems following Keutzer's partitioning approach, optimality is lost at multi-fanout vertices.

To alleviate that problem, Rudell proposed the *cross-tree phase assignment* heuristic, which simply indicated at multi-fanout points if an inverter had already been allocated. In that case, further use of the inverter came without any area penalty.

Finally, Rudell chose a different partitioning scheme to isolate the pattern trees. Starting from one primary output, he isolated a logic cone going all the way to the primary inputs. Then all the other primary outputs were processed in turn, each one becoming the tip of a logic cone whose support consisted of primary inputs or gates within the cones already processed. The advantage of this partitioning method is that very large trees are processed initially. However, the results that produced are dependent on the order in which the primary outputs are processed.

In addition to area-driven technology mapping, Rudell studied delay-driven transformations. He first considered gates with one fixed load, then extended the idea to gates with variable input loads. One solution is to extract all possible input loads from the cells

in the library, and extend the covering to find the best cost for each possible input load at each vertex in the subject graph. This extension guarantees that the final solution will produce the best arrival times. Since some libraries have cells with many different input loads, Rudell introduced the idea of *load binning*, which discretizes all possible input loads into a finite set of bins. Then, only the loads selected for the bins are considered during covering. A final extension to the delay-driven algorithm is to take area into account. Then among the bins with solutions meeting a given timing constraint, the one with the smallest area is chosen. If no solution meets the timing constraint, the minimum area solution is chosen among all minimum delay solutions.

Many other technology mappers have used pattern matching to determine logic equivalence. TECHMAP , from University of Colorado at Boulder, also uses structural comparisons. TECHMAP differs from DAGON and Berkeley's MIS in that it tries two different NAND2 decompositions to get better results for delay. It also uses algebraic symmetries to reduce the number of comparisons during the matching step. Its results are comparable to those of MIS [MJH89].

SKOL, written by Bergamaschi [Ber88], partitions library cells by the number of cubes in the corresponding logic function. The technology mapping step is done on Boolean factors instead of on an AND/OR decomposed network. It applies a peephole optimization where the best library element is chosen for each sub-circuit.

McMAP, developed by Lisanke *et al.* [LBK88], starts from the same ideas as DAGON and MIS. It breaks the initial network into simple AND/OR gates and then operates on trees. A special *gate_phase* operation is used to minimize the number of inverters. It also uses gate merging to create larger gates from smaller ones. However, the merge operation is highly dependent on the order in which the vertices are processed. Therefore, a number of random orderings are tried to improve the quality of the results.

MACDAS, introduced in 1986 by Sasao [Sas86], is a multi-level synthesis system that uses two-bit encoding and multi-valued simplification. Its main goal is to reduce the number of AND/OR gates under fanin constraints. The fanin constraints are applied as a last step, using *factorization on the network*, which divides AND/OR gates with large fanins into an interconnection of smaller AND/OR gates. This step can be viewed as

a simplified technology mapping step since the resulting circuit conforms to a technology dependent constraint (fanin limitation). However, it is limited in scope since only AND/OR gates are generated, and in general, target technologies include more complex functions than AND/OR gates. Therefore the usefulness of the technology mapper in MACDAS is limited when used stand-alone.

CARLOS, by Mathony and Baitinger, is a multi-level logic design system which includes a technology mapper [MB88]. The cells in the target technologies are restricted to NAND and NOR gates, inverters, and combinations of AND and OR gates with a maximum fanin of 4. As in MACDAS, the technology mapping operation is limited to very simple libraries.

Recently, a technology mapping system using Binary Decision Diagrams (BDDs) has been presented by Sato *et al.* [STMF90]. It is based on the use of permissible functions coupled with Boolean operations.

Most of the algorithmic systems exposed earlier are based on recognizing similar structures between the Boolean network and cells in the library. Although efficient, this has the disadvantage of making difficult and sometimes impossible to recognize logic gates with repeated inputs. For example the majority gate (represented by $F = ab + bc + ac$) cannot be found by any of these pattern-based systems.

In the following chapters, a new method for matching will be presented which will allow finding such matches. Based purely on Boolean operations, it operates as easily with functions involving repeated literals as it does with functions of single literals.

2.3 Comparison of rule-based and algorithmic methods

In principle, rule-based approaches can find solutions very close to the global optimum. Given powerful enough rules, and enough trials and modifications of these rules on the network, it is possible for the network to reach a state where no more rules can be applied without increasing the overall cost. Obviously, the rules and rule-application mechanism must be be powerful enough so that the system can escape a local optimum. But that comes at the price of having to run these systems for a very long time, without a clear idea of how close to optimum a solution is. Rule creation is also a difficult process,

especially when rules begin to interact with one another. The completeness of the rule set is necessary if solutions that are close to the global optimum are to be found (or equivalently, to avoid local minima). The performance of rule-based systems is closely tied to the quality of the rule set for the particular technology. The creation of rules for new technologies can be a difficult task, especially when some unusual configurations are made possible by new technologies.

Algorithmic based approaches on the other hand, rely on provable properties of algorithms and logic functions. Their execution time is typically much shorter than that of rule-based systems, with results of comparable or better quality. One advantage of algorithmic approaches is that it is possible to study beforehand the type and quality of solutions. Therefore bounds on both the run time and the quality of the resulting implementation can be found. However, this characteristic of algorithmic-based approaches turns the advantage to rule-based systems for problems that are not well-defined. Therefore, rule-based systems are well suited for problems which require *ad hoc* techniques, and algorithmic-based systems are preferred for solving problems with provable properties. In this dissertation, we consider technology mapping of combinational logic, which is a problem with known characteristics. We therefore base our operations on algorithms.

Most current algorithmic-based technology mapping systems use pattern matching to recognize logic equivalence between library cells and portions of Boolean networks. The advantage of pattern matching is fast execution, but it restricts the number of possible matches. Therefore, the optimality of the overall implementation is limited, when library cells with repeated literals (or equivalently, non series-parallel gates) are employed.

We propose a more powerful method for detecting logic equivalence. Our method is based on Boolean operations, and therefore is not limited by expressions with repeated literals. We further introduce the use of *don't care* information during logic equivalence detection, to allow further improvement in the number of detected equivalences. The resulting number of successful matches gives better quality solutions.

Chapter 3

An algorithmic approach to technology mapping

It was shown in the previous chapter that there exist two intrinsically difficult problems to technology mapping, *matching* and *covering*. The two problems are intertwined within technology mapping: the goal of *covering* is to produce the best set of *matching* library cells that implements the functionality of a given Boolean network. Therefore, any solution to the technology mapping problem must integrate solutions to both *matching* and *covering*.

Since solving the *general covering* problem is computationally intractable, we will solve the *restricted covering* problem presented in the last chapter. We follow the assumption made by previous systems [Keu87, Rud89a] that during *covering*, inputs to chosen cells correspond to existing vertices in the Boolean network. This has the advantage of eliminating the need to restructure the Boolean networks for each possible matching element. Since the structure of the initial Boolean network does not change, it is then possible to decouple the solutions to *matching* and *covering*: *covering* has no effect on the structure of the Boolean network, and therefore has no influence on the set of library cells found during the *matching* process.

We now present a divide-and-conquer approach to the technology mapping problem, where *matching* and *covering* are handled in separate steps. Given an initial Boolean network, we first find the set of subnetworks that match library cells (*i.e.* we solve the

matching problem first). Then, we select a set of matching library cells which, when properly connected, produce a Boolean network equivalent to the initial one (*i.e.* we then solve the *covering* problem). The remainder of this chapter focuses on the approach of solving the *covering* problem. Algorithms for *matching* are explained in detail in Chapters 4, 5 and 6.

In addition to decoupling the *matching* and *covering* steps, the *restricted covering* problem also has some implications on the types of network that are amenable to technology mapping. Since each matching library element must establish a correspondence between its inputs and existing vertices in the Boolean network, each element must be related to an integer number of vertices in the network. Otherwise, the logic function of some matching element would not correspond to the subnetwork induced by any subset of the vertices. As a result, the use of such elements would require an adjustment of the Boolean network by adding to it additional vertices to cover the mismatch. However, this addition of vertices would contradict the initial *restricted covering* premise, which assumed a constant set of vertices.

The Boolean behavior of a network must be preserved across the technology mapping process. Assuming that every vertex of the original network is necessary in the expression of its Boolean behavior, and given that vertices are involved as indivisible objects in matches with library elements, this implies that every Boolean function assigned to a vertex must be included in at least one Boolean function corresponding to a library element. This condition precludes vertices representing very complex Boolean functions from being matched. Therefore, Boolean networks must be preprocessed so that the *restricted covering* problem can be solvable. In particular, vertices representing large Boolean functions are broken into interconnections of vertices representing smaller Boolean functions. This preprocessing step is called decomposition.

3.1 Decomposition

The need for decomposition is a side-effect of choosing to solve the restricted covering problem. Decomposition is a necessary pre-processing step applied to the Boolean network beforehand, making possible the solution of the covering problem on a fixed set of

vertices.

The ultimate goal of technology mapping is to obtain the best solution possible. Because of the inherent complexity of the problems involved in technology mapping, an exact solution is not possible. Therefore, heuristics are used to arrive at a good solution. One sensible assumption is that finding more matches will produce better solutions. Therefore, a secondary goal of decomposition is to break the initial network into vertices that represent the smallest possible Boolean functions. We call these Boolean functions *base functions*, and they consist of two-input functions with unate inputs, *e.g.* two-input AND/OR/NAND/NOR functions. Most target libraries contain the base functions. In the remainder of this thesis we assume that the library under consideration includes the base functions.

As a result of decomposition, each function assigned to a vertex of the Boolean network can be represented by at least one, but possibly many elements of the library. Decomposition increases the granularity of the network, provides the covering step with more possibilities, and thus, leads to a better final mapping.

Decomposition can be viewed as a restructuring of the initial Boolean network. For any given network, there are many ways to create an equivalent network where the vertices correspond only to base functions. Since the covering operation will not change the structure of the network it is operating on (inputs to matches correspond to existing vertices in the network), the results of decomposition will bias the quality of the final solution. The problem is that it is difficult to evaluate the quality of the final implementation, since decomposition is performed independently of the target technology.

Different techniques can be used to decompose the network. Both disjoint and non-disjoint Boolean decompositions are possible [Ash59, Cur62, RK62]. Algebraic methods, involving for example kernel extraction, are another technique [BM82]. Recently, Bochman *et al.* introduced still another technique, a restricted non-disjoint Boolean decomposition method based on *functional groupability* [BDS91].

If we assume technology-independent restructuring has already taken place, then the decomposition does not need to be so powerful that it would radically change the initial structure. Therefore all that is needed is to break factored forms into two-input functions. For example, $\{x = a(b+c)\}$ becomes $\{x = ay; y = b+c\}$. But even simple splitting of

existing structures often has more than one solution. For example, $\{x = a + b + c + d\}$ can be expressed as $\{x = a + y_1; y_1 = b + y_2; y_2 = c + d\}$, or as $\{x = y_1 + y_2; y_1 = a + b; y_2 = c + d\}$. Depending on the target library and on the input arrival times, one decomposition can lead to a better solution than the other.

In the technology mapping system presented in this thesis, we use two different decomposition algorithms. The first one is used as a preprocessing step before covering. The second decomposition algorithm is used during delay optimization, and its goal is to restructure a portion of a network such that time critical paths in the network are shortened. This algorithm, more specific to iterative performance improvement, is presented in detail in Chapter 7.

```
decompose_all(network) {                    /* Network-wide decomposition */
    (∀ vertex v_j ∈ 𝒩) {                    /* Process every vertex */
        f_j = function_of(v_j)              /* Get function assigned to vertex */
        v_j = decomp(f_j) }                 /* Decompose function and return new assigned vertex */
    return }

decomp(f) {                                 /* Function decomposition procedure */
    op = operator_of(f)                     /* Get operator of top factor */
    if (op == LITERAL) {                    /* If top element is a literal, return */
        return(vertex_of(f)) }
    g = set_of_factors_of(f)                /* Get factors immediately under top one */
    (∀g_i ∈ g) {                            /* Decompose all factors found above */
        g_i = function_of(decomp(g_i)) }
    while (| g |> 1) {                      /* Process factorlist until only one left */
        g1 = next_element_of(g)             /* Get first two factors of the list */
        g2 = next_element_of(g)
        n = new_vertex(op,g1,g2)            /* Create a new vertex with function ((g1) op (g2)) */
        add_at_the_end(g,function_of(n)) }  /* Add new literal corresponding to new vertex */
                                            /* Note: adding at the end of the list generates a balanced decomposition, */
                                            /* adding at the beginning of the list produces an unbalanced decomposition. */
    return(vertex_of(next_element_of(g))) }
```

Figure 3.1: Algorithm for network decomposition

The first decomposition algorithm performs a simple splitting of the large equations generated by technology-independent logic synthesis operations. Each vertex that has a corresponding equation with more than 2 inputs is broken between factored forms, or between elements of factored forms (Figure 3.1). Additional vertices are added when needed in the Boolean network. This simple splitting reduces all vertices of the Boolean network to vertices representing simple two-input equations. This modified network is

then used as the starting point for covering.

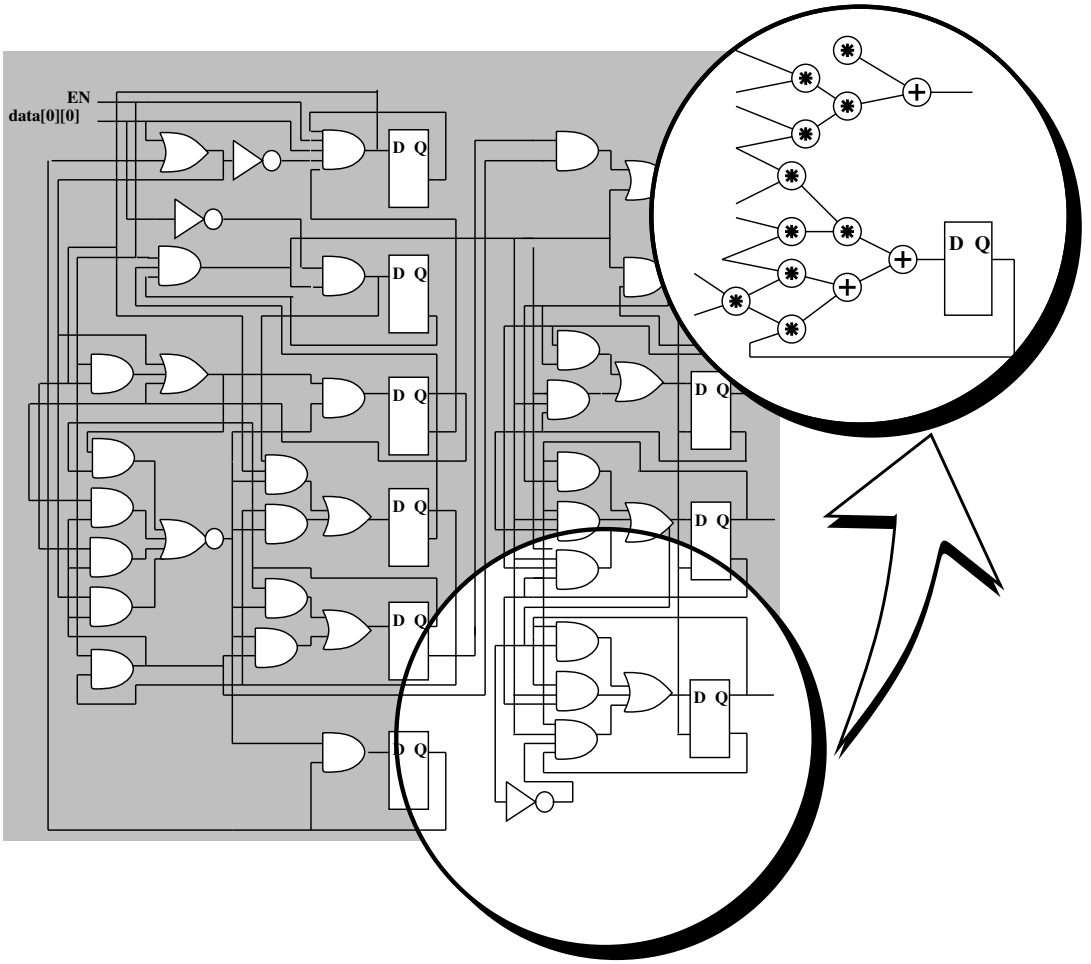


Figure 3.2: Two-input decomposition

Decomposition is recursively applied to each vertex v_i of the Boolean Network that is expressed by a complex Boolean function $\mathcal{F}_i$, *i.e.* $\mathcal{F}_i$ is not a base function. Decomposition yields an equivalent Boolean network, where each vertex is a *base function*. (Figure 3.2).

3.2 Partitioning

Even solving the simplified restricted covering problem is computationally complex. In particular, to solve the restricted covering problem one must solve the binate covering problem, which was shown in Chapter 2 to be harder than SATISFIABILITY. Algorithms have been proposed to solve the binate covering problem, but they are restricted to binate clauses with at most a few hundred variables [PS90, LS90]. Boolean networks can contain tens of thousands of vertices. Each vertex will match at least once, but more likely more than once with an element of the library. Therefore, in the worst case, the binate clauses encountered during technology mapping can contain on the order of 10^5 variables. This is clearly beyond the capabilities of current binate covering solvers. Therefore, more simplifications must be introduced to solve the fixed network covering problem.

It has been shown [Keu87] that polynomial-time solutions to the binate covering problem exist when the clauses representing matching library elements come from a Boolean network that is a tree. When dealing with trees instead of with DAGs, a polynomial-time solution to the binate covering problem exists, for the following reason: DAGs differ from trees in that DAGs have vertices with multiple fanouts, and trees do not. Recall the clauses that need to be satisfied in the general binate covering problem. If DAGs are covered, then multi-fanout vertices will possibly be represented by more than one variable being set to 1 in the set of clauses (*e.g.* vertex A in Figure 2.2 is represented twice, by both variables m_4 and m_5, in the best solution $\overline{m_1}\,\overline{m_2}\,\overline{m_3}m_4m_5$). More than one matching gate of the library can be used to represent the functionality of a multi-fanout vertex. Given that each of these matches can be chosen at the same time, then the number of possibilities is a product of all the matches coming from different multi-fanout branches. In the case of a tree, only one match will be chosen to represent any one vertex. Therefore, the number of possibilities is a sum of the possible matches at a certain vertex.

Isolating trees within a Boolean network ensures that polynomial algorithms can be used to find the best cover for each tree. Therefore, the goal of partitioning is to isolate subnetworks for which the binate covering problem can be solved efficiently. The limitation of this simplification is that finding an exact solution to subsets of a larger

problem does not guarantee an exact solution to the larger problem. Therefore, covering trees exactly does not guarantee an optimal cover for the overall circuit.

After decomposition has been applied to the entire Boolean network, we have a network consisting of interconnected 2-input, single-output Boolean functions. The circuit to be transformed must then be partitioned into single-output subnetworks. Partitioning is a heuristic step that transforms the technology mapping problem for multiple-output networks into a sequence of sub-problems involving single-output networks. Partitioning is performed initially before covering, and also as a part of the iterative improvement of a mapped network. We comment briefly here on the former case. The latter is described in Chapter 7.

Single-output subnetworks are created by traversing the network from the primary outputs to the primary inputs, creating new partitions for every primary output, and for every multi-fanout vertex traversed (Figure 3.3).

```
partition_all(network) {                                  /* Network-wide partitioning */
    (∀ vertices v_j ∈ 𝒩) {                                /* Process every vertex */
        mark_as_not_in_partition(v_j) }                   /* Used to insure each vertex belongs to a single partition */
    (∀ Primary output vertex v_o ∈ 𝒩) {                   /* Process every vertex */
        Γ_o = create_partition()                          /* New empty partition */
        partition(Γ_o,v_o) }                              /* Start a new partition */
    return }

partition(Γ,v) {                                          /* Recursive partitioning */
    if (is_primary_input(v) {                             /* Do not include PI in partitions */
        return }
    insert_in_partition(Γ,v)                              /* Insert vertex in partition */
    s = fanin_set_of(v)                                   /* Get fanin list of vertex */
    (∀s_i ∈ s) {                                          /* Process each fanin */
        if (fanout_number(s_i) > 1) {                     /* Break partitions at multi-fanout vertices */
            if (not_in_partition(s_i) {                   /* Make sure vertex is free */
                Γ_r = create_partition()
                partition(Γ_r,s_i) } }                    /* Recur */
        else {
            Γ_r = Γ
            partition(Γ_r,s_i) } }                        /* Recur */
    return }
```

Figure 3.3: Algorithm for network partitioning

The partitions generated by the previous algorithm are subnetworks. When partitioning is done, each vertex of a subnetwork has either a single fanout or is the output of the subnetwork. *Ad hoc* techniques are used to map sequential elements and I/Os. Therefore, partitioning is also used to isolate the combinational portion of a network from the sequential elements and I/Os. The circuit connections to the sequential elements are removed during the partitioning step. This effectively creates two distinct classes of partitions, combinational and sequential. At the end of the partitioning step, the circuit is represented by a set of combinational circuits that can be modeled by *subject graphs*, and a set of generic sequential elements (Figure 3.4).

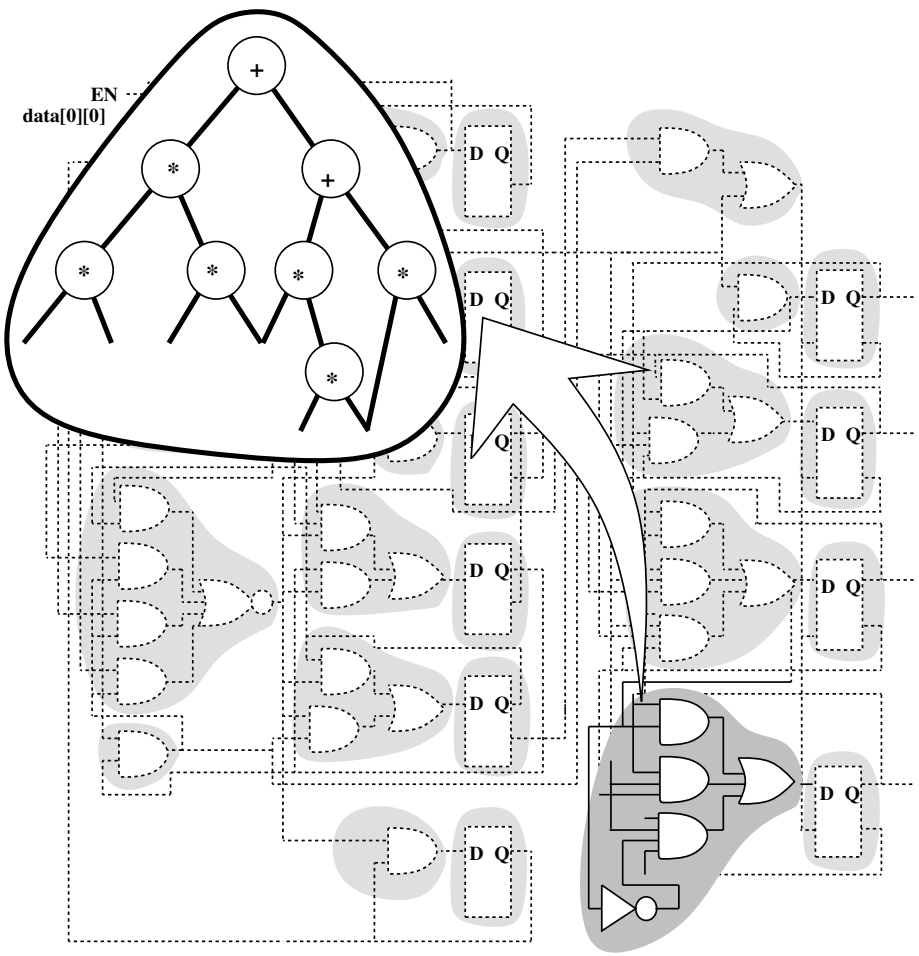


Figure 3.4: Circuit partitioning

3.3 Covering

Given that Boolean networks are first decomposed into a set of vertices whose corresponding Boolean functions are two-input base-functions, and that single-output subnetworks with no non-reconverging multi-fanout vertices are isolated, it is now possible to solve the restricted covering problem. Note that the subject graphs thus isolated are not necessarily trees, as in [Keu87, Rud89a]. We call the vertices of the subject graph with zero indegree *leaves*.

A minimum cost cover is found for each decomposed subject graph. The chosen cover represents the best set of library elements that is logically equivalent to the subject graph. Our solution to the covering problem is based on dynamic programming. Starting at the leaves of the subject graph, which represent the subnetwork inputs, the best solution at each vertex of the subject graph is found. At each vertex v, sub-graphs rooted at v are isolated (Figure 3.5).

During the covering step, matching is used to establish if there is an entry in the library that is functionally equivalent to that sub-graph. After resolving whether there is a logically equivalent element in the library, equivalent library elements are evaluated, and the best one is chosen. Choosing the best set of library elements representing a sub-graph is done as follows.

We denote by Γ_f a subject graph whose single output vertex is ν_f. We consider here the covering of a subject graph Γ_f that optimizes some cost criteria (*e.g.* area or timing). For this purpose we use the notions of *cluster* and *cluster function.*

A *cluster* is a connected sub-graph of the subject graph Γ_f, having only one vertex with zero out-degree ν_j (*i.e.* having only a single output). It is characterized by its depth (longest directed path to ν_j) and number of inputs. The associated *cluster function* is the Boolean function obtained by collapsing the Boolean expressions associated with the vertices into a single Boolean function. We denote all possible clusters rooted at vertex ν_j of Γ_f by $\{\kappa_{j,1}, \ldots, \kappa_{j,N}\}$.

As an example, consider the Boolean network (after an AND/OR decomposition):

$$\begin{aligned} f &= j + t \\ j &= xy \end{aligned}$$

$$
\begin{aligned}
x &= e + z \\
y &= a + c \\
z &= \overline{c} + d
\end{aligned}
$$

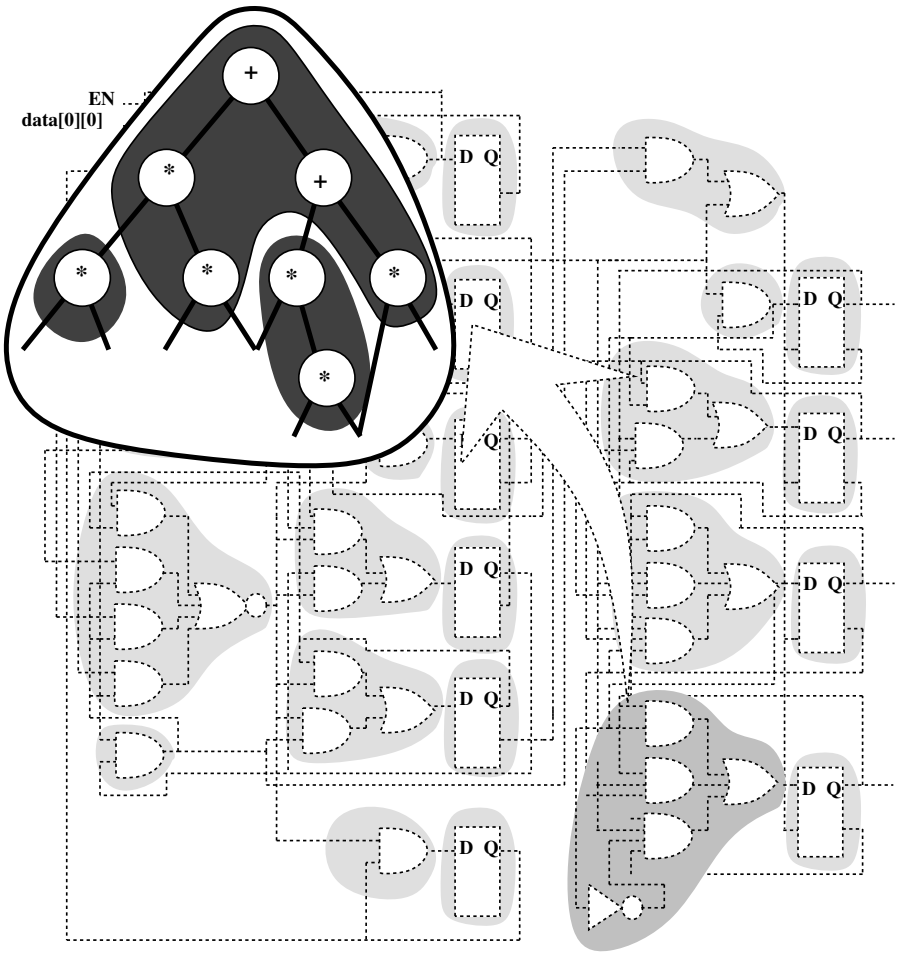


Figure 3.5: Graph covering

There are six possible cluster functions containing the vertex ν_j of the subject graph Γ_f (Figure 3.6):

$$\begin{aligned}
\kappa_{j,1} &= xy \\
\kappa_{j,2} &= x(a+c) \\
\kappa_{j,3} &= (e+z)y \\
\kappa_{j,4} &= (e+z)(a+c) \\
\kappa_{j,5} &= (e+\overline{c}+d)y \\
\kappa_{j,6} &= (e+\overline{c}+d)(a+c)
\end{aligned}$$

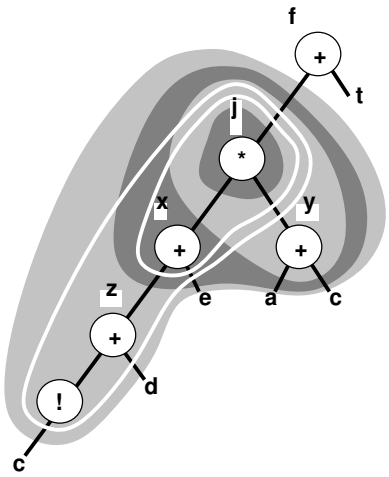


Figure 3.6: Graph of all possibles covers of j

The covering algorithm attempts to match each cluster function $\kappa_{j,k}$ to a library element. A cover is a set of clusters matched to library elements that represents the subject graph, which optimizes the overall area and/or timing. The area cost of a cover is computed by adding the cost of the clusters corresponding to the support variables in the cluster function of $\kappa_{j,k}$ to the cost of the library element corresponding to the cluster $\kappa_{j,k}$ under consideration. For each vertex ν_j in a subject graph Γ_f, there is always at least one cluster function $\kappa_{j,k}$ that matches, because the base functions (*e.g.* AND/OR) exist

in the library, and the network was decomposed accordingly during the decomposition phase. When matches exist for multiple clusters, for any tree-like decomposition the choice of the match of minimal area cost guarantees minimality in the total area cost of the matched sub-graph [Keu87, DGR+87]. The covering algorithm is implemented by procedure *cover* shown in Figure 3.7.

```
cover(top,equation,list,depth) {                                  /* Reached search depth: stop recursive expansion */
        if (depth  = max_depth) {
        return
    }
    if (equation is empty) {                                      /* First recursion on vertex 'top' */
        if (top not yet mapped) {
            equation = get_equation_from(top)                     /* Copy equation defining top originally */
            list = get_support_from(equation)                     /* Copy original support variables */
            cover(top,equation,list,1)                            /* Start expanding the equation */
            set_vertex_mapped(top)                                /* Covering of top is done */
        }
        return }
    while list is not empty {                                     /* Expand all input variables sequentially */
        if (vertex(list) is a primary input) skip this one        /* Do not try to expand primary inputs */
        if (vertex(list) is not mapped) {                         /* Condition for evaluating the best cost */
            cover(vertex(list),NULL,NULL,depth) }                 /* Map input if it's not mapped already */
        else if (any fanout(vertex(list)) does not reconverge at top) {   /* Stop at non-reconverging multi-fanout vertices */
            cover(vertex(list),NULL,NULL,depth) }
        else {                                                    /* Expand the current input variable */
            new_list = get_support_from(equation(vertex(list))    /* Support of the equation for the current variable */
            replace current list element by new_list              /* Augment the current list of support variables */
            new_equation = merge(equation,vertex(list))           /* Eliminate current variable into current equation */
            cover(top,new_equation,new_list,depth+1)              /* Recur on the new (expanded) equation */
            put back list in original state }                     /* Get back the input variable list before expansion */
        list = next_element_from(list) }                          /* Get next input variable */
    check_if_in_library(top,equation,list)                        /* Matches equation and a library element */
                                                                  /* and updates best cost cover */
    return }
```

Figure 3.7: Algorithm for network covering

The cost of any required inverters is also taken into account during the covering step. Each vertex ν_j, when mapped, is initially annotated with two library elements: the first one, C_{ON}, gives the best cost for generating the ON set f_j, and the second one, C_{OFF}, gives the best cost for generating the OFF set $\overline{f_j}$. As soon as variable j is used as an input to a gate that is being mapped, then C_{ON} or C_{OFF} is selected according to the required phase of j. If the same variable j is needed at a later stage with the opposite

phase, then an inverter is automatically taken into account in the cost computation.

The timing cost of a cover can be computed in a similar way by considering a constant delay model. To compute the local time at the vertex ν_j, the propagation delay through a cluster is added to the maximum of the arrival times at its inputs [Rud89a]. When matches exist for multiple clusters, then for any tree-like decomposition, the choice of the match of minimal local time guarantees minimality of the total timing cost of the matched sub-graph.

When considering a more realistic load-dependent delay model, the problem becomes more complex. The propagation delay through a library cell matching a particular vertex now depends on the load on its output. Since covering is solved using dynamic programming, a vertex is always processed before its fanouts, so the output load of a vertex is not known when it is being processed. One solution is to use load-binning [Rud89a, Rud89b]. Then, dynamic programming is used to find the best match at a vertex *for all possible load values*. This can be done by determining beforehand the number of different input loads among all inputs of all elements of a given target technology.

The above covering method, based on load-dependent delay model, will find the fastest implementation. A more important goal is to find the smallest implementation that meets given timing constraints. This was also considered in [Rud89a], but the proposed solution requires arrival time binning, which for exact results implies very large numbers of bins. Approximate solutions are therefore proposed, which relax the number of bins, at possibly a cost in the precision of the delay calculation.

The binning methods are compatible with the covering algorithm presented here. But we opted for a simpler, possibly more powerful method for dealing with load-dependent delay. We propose an iterative refinement technique, where circuits are first mapped for area, and then successively remapped for delay along critical paths. The iterative method is presented in more detail in Chapter 7.

Chapter 4

Boolean matching

Matching is one of the two intrinsically complex problems in technology mapping. In Chapter 3, we showed how *covering* and *matching* are decoupled through the *restricted covering* assumption. The quality of the solution found during covering is contingent on the number and quality of matches. Matching is used during the covering phase to verify that a particular single-output subnetwork is logically equivalent to an element of the library. Therefore, the algorithm adopted to solve the matching problem is of prime importance to the quality of the final implementation.

In this chapter, we consider the matching operation. We define more clearly the general problem to be solved, *i.e.* logic equivalence in the presence of *don't care* information. We introduce and justify the use of Boolean techniques as a powerful method for matching. We briefly review binary decision diagrams (BDD), since they constitute the most appropriate logic representation for the logic operations involved in the Boolean matching process. We introduce a simple algorithm for Boolean matching, which is powerful enough to accept both completely and incompletely specified logic functions. The requirements for efficiency when operating on functions with large number of inputs lead to the more powerful methods introduced in Chapters 5 and 6.

4.1 Matching logic functions

We formulate the matching problem as a problem of checking for logic equivalence between a given Boolean subnetwork (the *cluster function*, introduced in Section 3.3) and the Boolean function representing a library element. Since the support of the cluster function differs from that of the library element, we must find a one-to-one assignment between the variables in the support of the cluster function and the variables in the support of the library function. In general, any variable in one support can be assigned to any variable in the other support. Therefore, all permutations of the variables are possible.

We also consider phase-assignment in conjunction with the matching problem, because they are closely interrelated in affecting the cost of an implementation. In digital circuits it is often possible to change the polarity of a signal at no or low cost, and therefore it is important to detect logic equivalence between two functions after eliminating the input and output phase information. The need to account for inverters thus shifts from the matching step to the covering step, where the cost of additional inversion is considered.

Finally, we want to exploit *don't care* conditions during the matching operation. The importance of the use of *don't care* conditions in multiple-level logic synthesis is well recognized [BBH+88]. Since cluster functions are extracted from Boolean networks, their environment (*i.e.* the rest of the network) will often create *don't care* conditions on the inputs or the output of the cluster. We consider here *don't care* conditions specified at the network boundary or arising from the network interconnection itself [MB89]. Taking the *don't care* information into account during matching broadens the class of equivalent logic functions to which a cluster function belongs. This translates into more possible matches with library elements, and more degrees of freedom during covering.

We denote the cluster function $\mathcal{F} : \mathcal{B}^n \rightarrow \mathcal{B}$ by: $\mathcal{F}(x_1, \ldots, x_n)$. It is an n-input, single-output logic function. We denote the phase of variable x_i by: $\phi_i \in \{0, 1\}$, where $x_i^{\phi_i} = x_i$ for $\phi_i = 1$, $x_i^{\phi_i} = \overline{x}_i$ for $\phi_i = 0$. We denote the *don't care* set of the *cluster function* by: $\mathcal{DC}(x_1, \ldots, x_n)$. We assume $\mathcal{DC}$ is a given for now, and defer its calculation to Chapter 6. We denote the library by: $\mathcal{L} = \{\mathcal{G}_i : \mathcal{B}^{K_i} \rightarrow \mathcal{B}; i = 1, \ldots, \mid \mathcal{L} \mid\}$, where K_i is the number of inputs for library element $\mathcal{G}_i$. The elements $\mathcal{G}_i$ (or $\mathcal{G}$ for short) are multiple-input, single-output functions. We then define the matching problem as follows:

Given a cluster function $\mathcal{F}(x_1, \ldots, x_n)$, an associated *don't care* set $\mathcal{DC}(x_1, \ldots, x_n)$, and a library element $\mathcal{G}(y_1, \ldots, y_n)$, matching consists of finding an ordering $\{i, \ldots, j\}$ and a phase assignment $\{\phi_1, \ldots, \phi_n\}$ of the input variables of $\mathcal{F}$, such that either equation (4.1) or (4.2) is true:

$$\mathcal{F}(x_i^{\phi_i}, \ldots, x_j^{\phi_j}) = \mathcal{G}(y_1, \ldots, y_n) \tag{4.1}$$

$$\overline{\mathcal{F}}(x_i^{\phi_i}, \ldots, x_j^{\phi_j}) = \mathcal{G}(y_1, \ldots, y_n) \tag{4.2}$$

for each value of $(y_1, \ldots, y_n)$ and each *care* value of $(x_i^{\phi_i}, \ldots, x_j^{\phi_j}) \notin \mathcal{DC}$, *i.e.* either equation (1) or (2) holds for all minterms in the *care* set of $\mathcal{F}$.

If no such ordering and phase assignment exists, then the element $\mathcal{G}$ does not match the cluster function $\mathcal{F}$. Furthermore, if no element in the library $\mathcal{L} = \{\mathcal{G}_i : \mathcal{B}^{K_i} \rightarrow \mathcal{B}; i = 1, \ldots, \mid \mathcal{L} \mid\}$ matches $\mathcal{F}$, then $\mathcal{F}$ cannot be covered by the library $\mathcal{L}$. Note that when the library contains the base function, then any vertex v of the Boolean network must have at least one associated cluster function that is covered by a library element: the base function into which v was initially decomposed.

Let us define the *NPN-equivalent* set of a function $\mathcal{F}$ as the set of all the functions obtained by input variable Negation, input variable Permutation and function Negation [Mur71]. We then say that a function $\mathcal{F}$ matches a library element $\mathcal{G}$ when there exists an NPN-equivalent function that is tautological to $\mathcal{G}$ modulo the *don't care* set.

For example, any function $\mathcal{F}(a, b)$ in the set: $\{a + b, \overline{a} + b, a + \overline{b}, \overline{a} + \overline{b}, ab, \overline{a}b, a\overline{b}, \overline{a}\overline{b}\}$ can be covered by the library element: $\mathcal{G}(x_1, x_2) = x_1 + x_2$. Note that in this example $\mathcal{G}(x_1, x_2)$ has $n = 2$ inputs, and can match $n! \cdot 2^n = 8$ functions [MJH89].

4.2 Use of Binary Decision Diagrams

The matching algorithms presented here are all based on Boolean operations. The advantage of using Boolean operations over the structural matching algorithm used by others is that logic equivalence can be established regardless of the representation. For example, $f_1 = ab + ac + bc$ and $f_2 = a(b\overline{c} + \overline{b}c) + bc$ are logically equivalent, but structurally entirely different. Previous approaches used matching on trees or graphs representing the AND/OR (or equivalent) decomposition of a Boolean Factored Form (BFF). These

algorithms could not detect logic equivalence, since no graph operation can transform the BFF of f_1 into f_2 without taking advantage of the function's Boolean properties. It is important to note that different representations of Boolean functions arise because factoring is not unique, and even different canonical forms, such as *sum of products*, can represent the same function. Therefore, a covering algorithm recognizing matches independently from the representation can yield matches of better quality than matches obtained through structural matching techniques.

Our algorithms use Binary Decision Diagrams (BDDs) as the basis for Boolean comparisons. BDDs are based on Shannon cofactors. A logic function f is iteratively decomposed by finding the Shannon cofactors of the variables of f to form the BDD [Ake78, Bry86]. We use BDDs in the form proposed by Bryant, where a fixed ordering of the variables is chosen during Shannon decomposition [Bry86]. Elsewhere, these have been called *ordered binary decision diagrams*, or OBDDs for short [BRB90]. Bryant also introduced procedures to reduce the size of BDDs. For the purposes of technology mapping, where a BDD representation of a portion of the circuit to map is to be used only once, the computational cost of reducing BDDs is comparable to the cost of doing a single comparison between unreduced BDDS. Therefore we exploit a simple way of comparing unreduced, ordered BDDs.

4.3 A simple Boolean matching algorithm

A Boolean match can be determined by verifying the existence of a match of the input variables such that the cluster function $\mathcal{F}$ and the library element $\mathcal{G}$ are a tautology. Tautology can be checked by recursive Shannon decomposition [BHMSV84]. The two Boolean expressions are recursively cofactored generating two decomposition trees. The two expressions are a tautology if they have the same logic value for all the leaves of the recursion that are not in the *don't care* set. This process is repeated for all possible orderings of the variables of $\mathcal{F}$, or until a match is found.

The matching algorithm is described by the recursive procedure *simple_boolean_match* shown in Figure 4.1, which returns TRUE when the arguments are a tautology for some

variable ordering. At level n of the recursion, procedure *simple_boolean_match* is invoked repeatedly, with arguments the cofactors of the n^{th} variable of $\mathcal{G}$ and the cofactors of the variables of $\mathcal{F}$, until a match is found. In this case the procedure returns TRUE. If no match is found, the procedure returns FALSE. The recursion stops when the arguments are constants, in the worst case when all variables have been cofactored. The procedure returns TRUE when the corresponding values match (modulo the *don't care* condition). Note that when a match is found, the sequence of variables used to cofactor $\mathcal{F}$ in the recursion levels 1 to N represents the order in which they are to appear in the corresponding library element. The algorithm is shown in Figure 4.1.

```
simple_boolean_match(f,g,dc,var_list_f,var_list_g,which_var_g) {
    if (dc == 1) return(TRUE)                                   /* If leaf value of DC = 1, local match */
    if ( f and g are constant 0 or 1) return ( f == g )         /* If leaf value of f and g, matches if f == g */
    gvar = pick_a_variable(var_list_g,which_var_g)              /* Get next variable from the list of variables of g */
    remaining_var_g = get_remaining(var_list_g,which_var_g)     /* Get list of unexpanded variables of g */
    which_var_f = 1                                             /* Starting pointer for variables of f */
    while ( which_var_f ≤ size_of(var_list_f)) {                /* Try all unexpanded variables of f in turn */
        fvar = pick_a_variable(var_list_f,which_var_f)          /* Get next variable to expand */
        remaining_var_f = get_remaining(var_list_f,which_var_f) /* Update the list of unexpanded variables of f */
        f0 = shannon_decomposition(f,fvar,0)                    /* Find Shannon cofactor of f with (fvar = 0) */
        f1 = shannon_decomposition(f,fvar,1)                    /* Find Shannon cofactor of f with (fvar = 1) */
        g0 = shannon_decomposition(g,gvar,0)                    /* Find Shannon cofactor of g with (gvar = 0) */
        g1 = shannon_decomposition(g,gvar,1)                    /* Find Shannon cofactor of g with (gvar = 1) */
        dc0 = shannon_decomposition(dc,fvar,0)                  /* Find Shannon cofactor of dc with (fvar = 0) */
        dc1 = shannon_decomposition(dc,fvar,1)                  /* Find Shannon cofactor of dc with (fvar = 1) */

        if (simple_boolean_match(f0,g0,dc0,                     /* Verify that the cofactors of f and g */
            remaining_var_f,remaining_var_g,which_var_g+1)      /* are logically equivalent */
            and simple_boolean_match(f1,g1,dc1,
            remaining_var_f,remaining_var_g,which_var_g+1)) {
            return(TRUE) }
        else if (simple_boolean_match(f1,g0,dc0,                /* If the previous check failed, */
            remaining_var_f,remaining_var_g,which_var_g+1)      /* verify that f is equivalent to the complement of g */
            and simple_boolean_match(f0,g1,dc1,
            remaining_var_f,remaining_var_g,which_var_g+1)) {
            return(TRUE) }
        which_var_f = which_var_f + 1 }
    return(FALSE) }
```

Figure 4.1: Simple algorithm for Boolean matching

Note that in the worst-case all permutations and phase assignments of the input variables are considered. Therefore, up to $n! \cdot 2^n$ different ordered BDDs may be required for each match. Furthermore, all library elements with n or fewer inputs need to be

considered in turn, since *don't care* information might reduce the effective number of inputs. The worst-case computational complexity of the algorithm makes the matching procedure practical only for small values of n. Therefore, better algorithms for Boolean matching are required for more general cases when logic functions with larger number of inputs are considered.

In the following two chapters, we introduce new matching methods based on Boolean operations, which are computationally more efficient than the algorithm just described. In Chapter 5 we consider completely specified cluster functions (*i.e.* we assume $\mathcal{DC} = \emptyset$). We show that we can exploit the symmetry and unateness properties of the Boolean function to significantly reduce the search space, yielding an average computational complexity which is much lower than the upper bound discussed earlier. Chapter 6 expands this method to take *don't care* information into consideration during matching operations to improve the quality of the final result.

Chapter 5

Matching completely specified functions

In this chapter we consider the matching problem for completely specified functions, *i.e.* we neglect *don't care* information. This simplification makes possible the use of some properties of Boolean functions that otherwise would not be usable. In particular, there are invariants in completely specified functions that are not in the presence of *don't cares*. Unateness and symmetry are two such properties. We propose to use these two properties of Boolean functions to speed-up the Boolean matching operation, without hampering the accuracy or completeness of the results. We introduce the two properties as key elements to search space reduction in Section 5.1. We explain how the properties are extracted in Section 5.2. We conclude the chapter with benchmark results.

5.1 Search space reduction

The *simple Boolean matching* algorithm presented in Chapter 4 is computationally expensive for two reasons. First, $n!$ permutations of n inputs are needed before two functions can be declared non-equivalent. Second, for each permutation, all 2^n input phase-assignments are required before logic equivalence is asserted. Since all input permutations and phase-assignments must be tried before two logic functions are declared

different, then for any arbitrary n-input cluster function this implies that $n! \cdot 2^n$ comparisons are necessary in the worst case, *i.e.* whenever a match to a library element fails.

We now look into methods for reducing both the number of permutations and the number of phase-assignments during the process of determining logic equivalence. The number of required phase-assignments is reduced by taking the unateness property into account. The number of input permutations is reduced by using symmetry information. Note that the computational complexity is intrinsic to the Boolean matching problem; therefore, the worst case number of comparisons is still $n! \cdot 2^n$ for any arbitrary cluster function. However, we will show that the upper bound on complexity is related to the functionality of the library elements, and that most commercially available libraries are constituted of elements that imply much smaller upper bounds. Therefore, for most cluster functions, the worst-case bound is much less than $n! \cdot 2^n$. In addition, the average cost of Boolean matching is much lower than the worst-case bound and it is shown experimentally to be competitive with other matching techniques.

5.1.1 Unateness property

To increase the efficiency of the Boolean matching process, we take advantage of the fact that the phase information of unate variables is not needed to determine the logic equivalence. Therefore we define a transformation Υ that complements the input variables that are negative unate. For example, any function $\mathcal{F}(y_1, y_2)$ in the set: $\{y_1 + y_2, \overline{y_1} + y_2, y_1 + \overline{y_2}, \overline{y_1} + \overline{y_2}, y_1 y_2, \overline{y_1} y_2, y_1 \overline{y_2}, \overline{y_1}\,\overline{y_2}\}$ can be represented by the set: $\{y_1 + y_2, y_1 y_2\}$. Note that the phase information still must be kept for binate variables, where both the positive and negative phases are required to express $\mathcal{F}$. By using the transformation Υ, we reduce the information required for the matching operation and therefore also reduce its computational cost. In particular, since the phase of unate variables is predefined, the number of input phase-assignments required for an n-input Boolean function decreases from 2^n to 2^b, where b is the number of binate variables.

As a result of using the unateness property, we redefine the matching problem as follows:

Given a cluster function $\mathcal{F}(x_1,\ldots,x_n)$ and a library element $\mathcal{G}(y_1,\ldots,y_n)$, find an ordering $\{i,\ldots,j\}$ and a phase assignment $\{\phi_k,\ldots,\phi_l\}$ of the binate variables $\{k,\ldots,l\}$ of $\mathcal{F}$, such that either (5.1) or (5.2) is true:

$$\Upsilon(\mathcal{F}(x_i,\ldots,x_k^{\phi_k},\ldots,x_l^{\phi_l},\ldots,x_j)) \equiv \Upsilon(\mathcal{G}(y_1,\ldots,y_n)) \tag{5.1}$$

$$\Upsilon(\overline{\mathcal{F}}(x_i,\ldots,x_k^{\phi_k},\ldots,x_l^{\phi_l},\ldots,x_j)) \equiv \Upsilon(\mathcal{G}(y_1,\ldots,y_n)) \tag{5.2}$$

The unateness property is also important for another aspect of search space reduction. Since unate and binate variables clearly represent different logic operations in Boolean functions, any input permutation must associate each unate (binate) variable in the cluster function to a unate (binate) variable in the function of the library element. This obviously affects the number of input variables permutations when assigning variables of the cluster function to variables of the library element. In particular, it implies that if the cluster function has b binate variables, then only $b!\cdot(n-b)!$ permutations of the input variables are needed. Therefore the worst-case computational cost of matching a cluster function with b binate variables is $b!\cdot(n-b)!\cdot 2^b$.

5.1.2 Logic symmetry

One additional factor can be used to reduce the number of required input permutations. Variables or groups of variables that are interchangeable in the cluster function must be interchangeable in the function of the library element. This implies that logic symmetry can be used to simplify the search space.

Variables are symmetric if they can be interchanged without affecting the logic functionality [McC56a]. Techniques based on using symmetry considerations to speed-up algebraic matching were also presented by Morrison in [MJH89], in a different context.

Definition 5.1 *Logic symmetry is represented by the binary relation $\mathcal{SR}_\mathcal{F}$ on the set of inputs $\{x_1,\ldots,x_n\}$ of $\mathcal{F}$, where $\mathcal{SR}_\mathcal{F} = \{\{x_i,x_j\} \mid \mathcal{F}(x_1,\ldots,x_i,\ldots,x_j,\ldots,x_n) \equiv \mathcal{F}(x_1,\ldots,x_j,\ldots,x_i,\ldots,x_n)\}$. In the following, we write $\mathcal{SR}_\mathcal{F}(x_i,x_j)$ to indicate that $\{x_i,x_j\}$ belongs to $\mathcal{SR}_\mathcal{F}$.*

Theorem 5.1 *The symmetry property of completely specified functions is an equivalence relation.*

Proof: An equivalence relation is reflexive, symmetric and transitive [Liu77a]. From the definition of $\mathcal{SR}_{\mathcal{F}}$, it is obvious that relation $\mathcal{SR}_{\mathcal{F}}(x_i, x_i)$ is true, thus $\mathcal{SR}_{\mathcal{F}}$ is reflexive. By definition, $\mathcal{SR}_{\mathcal{F}}$ is symmetric. Finally, if both $\mathcal{SR}_{\mathcal{F}}(x_i, x_j)$ and $\mathcal{SR}_{\mathcal{F}}(x_i, x_k)$ are true, then $\mathcal{SR}_{\mathcal{F}}(x_j, x_k)$ is also true, because $\mathcal{SR}_{\mathcal{F}}(x_i, x_j)$ means x_j can replace x_i in $\mathcal{F}$, and in that case $\mathcal{SR}_{\mathcal{F}}(x_i, x_k) \equiv \mathcal{SR}_{\mathcal{F}}(x_j, x_k)$. Therefore, $\mathcal{SR}_{\mathcal{F}}$ is transitive. ■

Corollary 5.1 *The symmetry property $\mathcal{SR}_{\mathcal{F}}$ of the input variables of Boolean equation $\mathcal{F}$ implies a partition of the variables of $\mathcal{F}$ into disjoint subsets.*

Proof: It follows from $\mathcal{SR}_{\mathcal{F}}$ being an equivalence relation. ■

Definition 5.2 *A* symmetry set *of a function $\mathcal{F}$ is a set of variables of $\mathcal{F}$ that belongs to the binary relation $\mathcal{SR}_{\mathcal{F}}$.*

Two variables x_i and x_j of $\mathcal{F}$ belong to the same symmetry set if $\mathcal{SR}_{\mathcal{F}}(x_i, x_j)$.

Let us consider for example function $\mathcal{F} = x_1x_2x_3 + x_4x_5 + x_6x_7$. The input variables of $\mathcal{F}$ can be partitioned into three disjoint sets of symmetric variables: $\{x_1, x_2, x_3\}$, $\{x_4, x_5\}$ and $\{x_6, x_7\}$.

Symmetry sets are further grouped into symmetry classes.

Definition 5.3 *A* symmetry class $C_i, i \in \{1, 2, \ldots\}$ *is an ensemble of symmetry sets with the same cardinality i and $S_i =\mid C_i \mid$ is the cardinality of a symmetry class C_i.*

In the previous example, there are two symmetry classes: $C_2 = \{\{x_4, x_5\}, \{x_6, x_7\}\}$ and $C_3 = \{x_1, x_2, x_3\}$, with $S_2 = 2$, $S_3 = 1$. Note that all the other symmetry classes are empty, and therefore $\forall_{i \neq 2,3}\ S_i = 0$.

The symmetry properties are exploited in technology mapping as follows. Before invoking the mapping algorithm, the symmetry classes of each library element are calculated once. Symmetry classes are used in three different ways to reduce the search space during the matching phase. First, they are used as a filter to quickly find good candidates for matching. A necessary condition for matching a cluster function $\mathcal{F}$ with a library element $\mathcal{G}$ is that both have exactly the same symmetry classes. Hence only a small fraction of the library elements must be checked by the full Boolean comparison algorithm to see if they match the cluster function.

Second, symmetry classes are used during the variable ordering. Once a library element $\mathcal{G}$ that satisfies the previous requirement is found, the symmetry sets of $\mathcal{F}$ are compared to those of $\mathcal{G}$. Then, only assignments of variables belonging to symmetry sets of the same size can possibly produce a match. Since all variables from a given symmetry set are equivalent, the ordering of the variables within the set is irrelevant. This implies that permutations need only be done over symmetry sets of the same size, *i.e.* symmetry sets belonging to the same symmetry class C_i. Thus the number of permutations required to detect a match is: $\prod_{i=1}^{q}(S_i!)$, where q is the cardinality of the largest symmetry set, and S_i was defined above as the cardinality of a symmetry class C_i.

For example, let us enumerate the permutations for matching functions $\mathcal{F} = y_1y_2(y_3 + y_4) + y_5y_6$ and $\mathcal{G} = i_1i_2 + (i_3 + i_4)i_5i_6$. Function $\mathcal{F}$ has one non-empty symmetry class, $C_2(\mathcal{F})$, which contains three symmetry sets, $\{y_1, y_2\}$, $\{y_3, y_4\}$, and $\{y_5, y_6\}$. We associate a name, η_i, with each of the symmetry sets: $C_2(\mathcal{F}) = \{\{y_1, y_2\}, \{y_3, y_4\}, \{y_5, y_6\}\} \equiv \{\eta_1, \eta_2, \eta_3\}$ (*i.e.* we represent the pair of symmetric variables $\{y_1, y_2\}$ by η_1, the pair $\{y_3, y_4\}$ by η_2, etc.). Similarly, function $\mathcal{G}$ has only one non-empty symmetry class, C_2, with cardinality $S_2 = 3$. We associate a name, ξ_j, with the symmetry sets of $\mathcal{G}$: $C_2(\mathcal{G}) = \{\{i_1, i_2\}, \{i_3, i_4\}, \{i_5, i_6\}\} \equiv \{\xi_1, \xi_2, \xi_3\}$. We then use the labels η and ξ to represent the different permutations of symmetry sets. The cardinality of the symmetry class C_2 is $S_2 = 3$, and therefore there are $S_2! = 6$ possible assignments of symmetry sets of $\mathcal{F}$ and $\mathcal{G}$:

$$(\eta_1, \xi_1), (\eta_2, \xi_2), (\eta_3, \xi_3)$$
$$(\eta_1, \xi_1), (\eta_2, \xi_3), (\eta_3, \xi_2)$$
$$(\eta_1, \xi_2), (\eta_2, \xi_1), (\eta_3, \xi_3)$$
$$(\eta_1, \xi_2), (\eta_2, \xi_3), (\eta_3, \xi_1)$$
$$(\eta_1, \xi_3), (\eta_2, \xi_1), (\eta_3, \xi_2)$$
$$(\eta_1, \xi_3), (\eta_2, \xi_2), (\eta_3, \xi_1)$$

Only the last assignment, where the variables of $\mathcal{F}$ and $\mathcal{G}$ are paired as $\{\{y_1, y_2\}, \{i_5, i_6\}\}$, $\{\{y_3, y_4\}, \{i_3, i_4\}\}$, $\{\{y_5, y_6\}, \{i_1, i_2\}\}$, make functions $\mathcal{F}$ and $\mathcal{G}$ logically equivalent.

The third use of symmetry classes is during the Boolean comparison itself. Boolean

comparisons are based on iterative Shannon cofactoring. Without symmetry considerations, for an n-input function $\mathcal{F}$, up to 2^n cofactors are needed. But since variables of a symmetry set are freely interchangeable, not all 2^n cofactors are different. For example, given $F = abc$, where $\{a, b, c\}$ are symmetric, then the cofactor of $\{a = 0, b = 1, c = 0\}$ is equivalent to the cofactor of $\{a = 1, b = 0, c = 0\}$.

Theorem 5.2 *Given a function $\mathcal{F}$ with a symmetry set containing m variables $\{y_1, \ldots, y_m\}$, only $m + 1$ of the 2^m cofactors $\mathcal{F}\,|_{y_1^{\phi_1} \ldots y_m^{\phi_m}}$ are different.*

Proof: We define $\mathcal{A}_k(m, v)$, an assignment where v out of the m variables are set to 1, and the remaining $m - v$ variables are set to 0. There are $\begin{pmatrix} m \\ v \end{pmatrix}$ such $\mathcal{A}_k(m, v)$ assignments. By definition, the m variables are symmetric. Therefore, any pair of variables of the symmetry set can exchange their values without changing $\mathcal{F}$. As a result, any assignment $\mathcal{A}_k(m, v); (k = 1, \ldots, \begin{pmatrix} m \\ v \end{pmatrix})$ will correspond to the same value of $\mathcal{F}$, because any assignment $\mathcal{A}_k(m, v)$ can be transformed into any assignment $\mathcal{A}_j(m, v)$ by simple pairwise permutations of the assigned values. Therefore, only assignments $\mathcal{A}_k(m, v)$ with different values of v can potentially make $\mathcal{F}$ different. Since $0 \leq v \leq m$, then v can take only $m + 1$ different values (*i.e.* $0, 1, \ldots, m$ variables can be set to 1), and consequently there are at most $m + 1$ different cofactors. ■

Assuming the n variables of $\mathcal{F}$ are partitioned into k symmetry sets of size $n_1, \ldots, n_k$ (where $\sum_{i=0}^{k} n_i = n$), then the number of necessary cofactors is $\prod_{i=0}^{k}(n_i + 1) \leq 2^n$.

Although in the worst case, logic equations have no symmetry at all, our experience with commercial standard cells and (programmable) gate array libraries shows that the library elements are highly symmetric. We computed the symmetry classes C_i of every element of three available libraries (CMOS3, LSI Logic and Actel), and established the cardinality S_i of each symmetry class C_i extracted. We found that the average cardinality $\overline{S_i}$ of all the symmetry sets of the library cells in the three libraries is less than 2, as shown in Figure 5.1. Therefore, the number of permutations $\prod_{i=1}^{q}(S_i!)$ on the average is close to 1.

Unateness information and symmetry classes are used together to further reduce the search space. Unate and binate symmetry sets are distinguished, since both unateness and

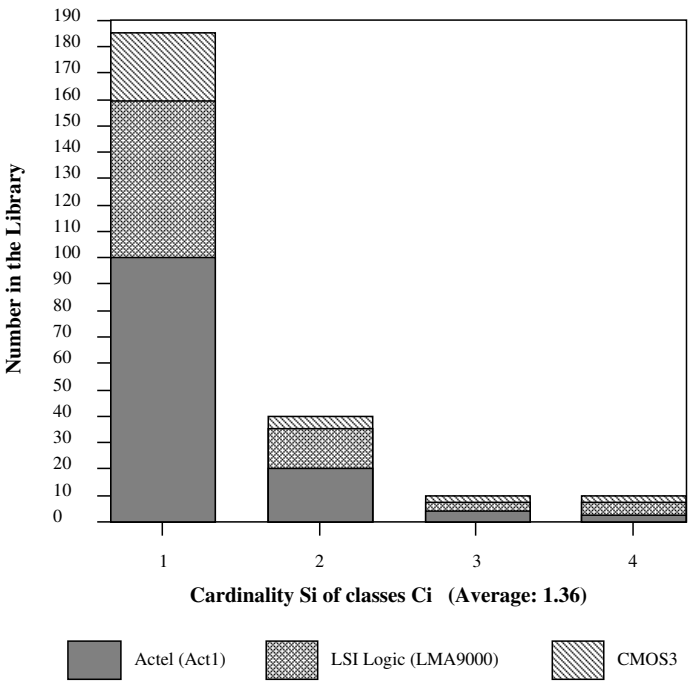


Figure 5.1: Distribution of symmetry classes C_i

symmetry properties must be the same for two variables to be interchangeable. We now compute the worst case bound. We separate symmetry classes C_i into disjoint subclasses C_i^u and C_i^b of unate and binate variables ($C_i^u \cup C_i^b = C_i$ and $C_i^u \cap C_i^b = \emptyset$). Therefore, we split the symmetry classes cardinality into $S_i = S_i^u + S_i^b$, where S_i^u is the cardinality of $C_i^u \in C_i$ and S_i^b is the cardinality of $C_i^b \in C_i$. This further reduces the number of permutations to $\prod_{i=1}^{q} S_i^u! \cdot S_i^b! = \prod_{i=1}^{q} S_i^u! \cdot (S_i - S_i^u)! < \prod_{i=1}^{q} S_i!$. Hence, when considering the phase assignment of the binate variables, at most $\prod_{i=1}^{q} S_i^u! \cdot (S_i - S_i^u)! \cdot 2^{i \cdot (S_i - S_i^u)}$ Boolean comparisons must be made in order to find a match.

As an example, in the Actel library *Act1*, the worst case occurs for the library element $MXT = d_0 c_1 c_3 + d_1 \overline{c_1} c_3 + d_2 c_2 \overline{c_3} + d_3 \overline{c_2 c_3}$, where $S_1 = 7$, and $S_1^u = 4$. In that case, $4! \cdot 3! \cdot 2^3 = 1152 \ll 7! \cdot 2^7 = 645,120$, where $7! \cdot 2^7$ represents the number of comparisons needed if no symmetry or unateness information is used.

Procedure *boolean_match*, a variation on procedure *simple_boolean_match*, is shown

in figure 5.2. It incorporates the symmetry and unateness information to reduce the search space: permutations are done only over symmetry sets of the same size. In addition, symmetry sets of unate and binate variables are separated into distinct classes C_i^u and C_i^b. Then only symmetry sets with the same unateness property are permuted.

```
boolean_match(f,g,f_symmetry_sets,g_symmetry_sets) {
    if ( f and g are constant 0 or 1) {                    /* If leaf value of f and g, matches if f == g */
        return ( f = g ) }
    if ( f_symmetry_sets is empty) {                       /* All variables of current symm set are assigned */
        f_symmetry_sets = get_next_f_symmetry_set()        /* Get next symm set in the list */
        symmetry_size = size_of(f_symmetry_sets)
        while ( symmetry sets of g with                    /* Try all symm sets of g with the same size */
            size symmetry_size have still to be tried) {   /* and the same unateness property */

            g_symmetry_sets = get_next_available_set(g,symmetry_size)
            boolean_match(f,g,f_symmetry_sets,g_symmetry_sets)
            if ( it is a match) return(TRUE)
            else return(FALSE) } }
    fvar = pick_a_variable(f_symmetry_sets)                /* Get variables from compatible symm sets */
    gvar = pick_a_variable(g_symmetry_sets)

    f0 = shannon_decomposition(f,fvar,0)                   /* Find Shannon cofactor of f with (fvar = 0) */
    f1 = shannon_decomposition(f,fvar,1)                   /* Find Shannon cofactor of f with (fvar = 1) */
    g0 = shannon_decomposition(g,gvar,0)                   /* Find Shannon cofactor of g with (gvar = 0) */
    g1 = shannon_decomposition(g,gvar,1)                   /* Find Shannon cofactor of g with (gvar = 1) */

    if ((boolean_match(f0,g0,f_symmetry_sets,g_symmetry_sets))
        and (boolean_match(f1,g1,f_symmetry_sets,g_symmetry_sets))) {
        return(TRUE) }                                     /* Verify that the cofactors of f and g are equivalent */
    else if ((boolean_match(f0,g1,f_symmetry_sets,g_symmetry_sets))
        and (boolean_match(f1,g0,f_symmetry_sets,g_symmetry_sets))) {
        return(TRUE) }                                     /* Verify that f is equivalent to the complement of g */
    else return(FALSE) }
```

Figure 5.2: Algorithm for fast Boolean matching

5.2 Determination of invariant properties

Unateness and logic symmetry are the two invariant properties we utilize for search space reduction during Boolean matching. Since cluster functions represent arbitrary portions of Boolean networks, we preprocess every cluster function to detect possible simplification before the unateness and symmetry properties are extracted.

In particular, the preprocessing step recognizes and eliminates vacuous variables. Recall that an equation $\mathcal{F}$ is vacuous in a variable v_i if the equation can be expressed without the use of v_i [McC86].

Vacuous variables are detected by checking if $\mathcal{F}_{v_i} \equiv \mathcal{F}_{\overline{v}_i}$ for any given variable v_i. When this condition is true, variable v_i is vacuous, and therefore does not influence the value of $\mathcal{F}$. In that case, we arbitrarily set variable v_i to 0 or to 1, to simplify the expression of function $\mathcal{F}$. Shannon decomposition is used for this detection. Since $\mathcal{F}_{v_i}$ and $\mathcal{F}_{\overline{v}_i}$ are derived from the same equation $\mathcal{F}$, both of their supports are subsets of the support of $\mathcal{F}$. Hence the variable ordering of $\mathcal{F}_{v_i}$ and $\mathcal{F}_{\overline{v}_i}$ is unique, and no permutation (nor phase assignment) is needed during the Shannon decomposition.

5.2.1 Unateness extraction

Unateness is the first property to be extracted from Boolean functions. For efficiency reasons, the unateness determination is done in two successive steps.

The first step consists of considering a decomposition of the function $\mathcal{F}$ into base functions represented by a leaf-DAG and detecting the phase of each variable of $\mathcal{F}$. The phase detection proceeds as follows. Starting at the root of the leaf-DAG representing function $\mathcal{F}$, a token representing a positive phase is propagated depth first towards the leaves of the DAG. When a vertex corresponding to a negative unate function is traversed, the phase of the token passed down is complemented. Each variable reached during the graph traversal is annotated with the phase of the current token. This phase-detection operation is implemented in procedure *get_unate* (Figure 5.3), where we assume that all base functions are unate. It would be possible to extend procedure *get_unate* to allow binate vertices as well (by extending the possible values of the token to positive, negative and *both*), but it is unnecessary: we use 2-input unate functions as base-functions during the decomposition step, therefore the leaf-DAG cannot contain binate vertices.

The traversal of the leaf-DAG in procedure *get_unate* takes at most $2n - 1$ steps, where n is the number of leaves: since the network is decomposed into 2-input gates, then each level in the levelized DAG has at most half the number of vertices of the previous level. Therefore, such a DAG with n inputs has at most $n + \frac{n}{2} + \frac{n}{4} + \cdots + 1 =$

$\sum_{i=0}^{\infty} \frac{n}{2^i} - \sum_{i=0}^{\infty} \frac{1}{2^i} + 1 = 2n - 2 + 1 = 2n - 1$ vertices.

All variables used in only one phase are necessarily unate. However, the first operation can falsely indicate binate variables, because the algorithm relies on structure, not on Boolean operations. For example, it would indicate that variable a in $\mathcal{F} = a\overline{c} + ab + \overline{a}bc$ is binate, whereas it is unate ($\mathcal{F}$ is simply a multiplexer $\mathcal{F} = a\overline{c} + bc$). The first step is used because it is a linear complexity operation. But when binate variables are detected, a more powerful technique is required to determine that the variables are really binate.

In the second step, the unateness property of the remaining variables (those which the first step labeled as binate) is detected verifying implications between cofactors [McC86]. The unateness property of these *possibly* binate variables is detected by verifying if $\mathcal{F}_{v_i} \Rightarrow \mathcal{F}_{\overline{v}_i}$ (negative unate variable) or if $\mathcal{F}_{\overline{v}_i} \Rightarrow \mathcal{F}_{v_i}$ (positive unate variable). If neither implication is true, then variable v_i is binate (Figure 5.3).

5.2.2 Symmetry classes determination

Once the unateness information has been determined, symmetry properties are extracted. The transformation Υ, presented in Section 5.1.1, is applied to ensure that symmetry will be detected between unate variables regardless of phase. By definition, two variables are symmetric if interchanging them in the expression of a function does not change the original function. We detect that two variables are symmetric simply by verifying that $\mathcal{SR}_{\mathcal{F}}(x_i, x_j)$ is true for that pair of variables. Since logic symmetry is an equivalence relation, as we established in Section 5.1.2, it is transitive. Therefore, if v_i is symmetric to v_j, and v_j to v_k, the symmetry of v_i and v_k is established without further verification. Similarly, if v_i is symmetric to v_j, and v_j is not symmetric to v_k, then v_i is not symmetric to v_k. As a result, when two variables are symmetric, the symmetry relations of the second variable are identical to those of the first variable, and do not need to be established through additional verification. This implies that it is not always necessary to verify all pairs of variables for symmetry. All pairs of variables must be processed (by verifying that $\mathcal{SR}_{\mathcal{F}}(x_i, x_j)$ is true) only when there is no symmetry. This is the worst case, and $\frac{n(n-1)}{2}$ swaps must be done, where n is the number of inputs to the equation. When a function is completely symmetric, *i.e.* when all inputs to a function $\mathcal{F}$ are symmetric,

then only $n-1$ swaps are needed (Figure 5.4).

The unateness property is used to reduce the number of swaps needed. Assuming b out of the n input variables are binate, then at worst $\frac{b(b-1)}{2} + \frac{(n-b)(n-b-1)}{2}$ swaps are required. At best $n-2$ swaps are needed, when both binate and unate variables are maximally symmetric. In order to verify that swapping two variables $\{v_i, v_j\}$ leaves $\mathcal{F}$ unchanged, it is sufficient to verify that $\mathcal{F}_{v_i\overline{v}_j} \equiv \mathcal{F}_{\overline{v}_i v_j}$. As in the first step, this is done using Shannon decomposition, and a single ordering (and phase assignment) of the variables is sufficient.

Since the phase information is relevant to binate variables, two swaps must be done for each pair of binate variables, one swap for each phase assignment of one of the two variables. Again, Shannon decomposition is used to check if the two instances of the equation are the same, and, as in the first step, only one variable ordering is used.

From an implementation standpoint, symmetry classes are established once for each library element. Each library element is then inserted into a database, using its symmetry sets as the key. Library elements with the same symmetry sets are further grouped by functionality (*e.g.* $\mathcal{G}_1 = y_1 y_2$ and $\mathcal{G}_2 = y_1 + y_2$ are grouped together in a new entry $\mathcal{H}$ of the database corresponding to functions of 2 equivalent, unate inputs).

5.3 Results

Tables 5.1, 5.2 and 5.4 show mapping results for a set of benchmark circuits which are optimized for area, *i.e.* a weighted sum of the matches is minimized. Note that the algorithms presented in this chapter can be used for delay optimization when assuming a constant load model, or when using load-binning, as explained at the end of Chapter 3. However, we decided to use an iterative method to specifically deal with timing issues, and we defer the presentation of the iterative method until Chapter 7.

Ceres was run using various covering depths, which allow trade-offs between run-times and quality of results. We noted that remapping (*i.e.* extracting the logic from a mapped network, and applying technology mapping again) often improves the results. Ceres allows for an arbitrary number of remapping operations. The following results represent the best of two successive mapping operations. Run-times reflect the use of

two complete mapping steps and one logic extraction.

Table 5.1 represents circuits bound to the Actel library, which has a large number of elements with repeated literals. In this case, results show that using Boolean operations for matching leads both to better implementation and faster run-times when compared to pattern matching based comparisons: results range from 3% smaller area with more than 12X faster run-time, to 10% smaller area with 4X faster run-time.

Table 5.2 shows mapping results using the complete Actel (*Act1*) library, which comprises over 700 elements[1].

Table 5.3 gives mapping results using the complete Actel (*Act2*) library, which comprises over 730 elements[2].

Note that in Tables 5.2 and 5.3 we did not include results from program *sis* [SSM+92], for two reasons. First, pattern matching, which is the core matching algorithm in *sis*, does not utilize complex gates with repeated literals. Since most gates in the full *Act1* and *Act2* libraries have repeated literals, the majority of the gates would not have been usable. Second, the number of patterns generated by *sis* needed to encompass all the possible representations would be excessively large. We indeed tried to use program *sis* with the full *Act1* library, but stopped it after 20 hours, as it was still working on the pattern generation for the internal library representation.

Table 5.4 shows results using the LSI Logic 10K library. This table shows again how Ceres can trade-off quality of results for run-time. In particular, results comparable in area to those of *sis* are obtained with 30% faster run-times, or 4% smaller area at a cost of a 2X increase in run-time.

Table 5.5 gives results using a library derived from the Quick Logic master cell [Bak91]. For that technology, as for the full actel libraries, we do not compare the results to those of *sis* because it was not possible to build and use the entire library in program *sis*.

[1]The elements of the Actel library *Act1* are all derived from the master equation $\mathcal{A} = (a+b)(cd+e\overline{d}) + \overline{(a+b)}(fg+h\overline{g})$, using any possible input bridge or stuck-at combination.

[2]The elements of the Actel library *Act2* are derived from the master equation $\mathcal{A} = (a+b)(cdg+e\overline{dg}) + \overline{(a+b)}(fdg+h\overline{dg})$, using any possible input bridge or stuck-at combination.

```
extract_unateness(equation) {
        input_vars = support(equation)                          /* Get the list of inputs of eq */
        foreach(v ∈ input_vars) {                               /* Process each variable */
                reset_unateness(v) }                            /* Initialize v's unateness to unknown */
        get_unate(f,POSITIVE)                                   /* First step */
        foreach(v ∈ input_vars) {                               /* Process each variable */
                if (is_binate(v) {                              /* Leave unate variables */
                        get_binate(f,v) } } }                   /* Second step */

get_unate(f,phase) {
        if (is_empty(f)) {                                      /* no more factored forms (FF) */
                return }
        if (is_annotated(f) .AND. (annotation(f) == phase) {    /* f already reached with that phase */
                return }
        lower_level_phase = phase                               /* Phase underneath FF */
        if (type(f) == VARIABLE) {                              /* Reached a leaf */
                set_unateness(variable(f),phase) }              /* Tag the variable with current phase */
        else if (type(f) == INVERSION) {                        /* Phase is inverted underneath FF */
                lower_level_phase = complement(phase) }

        set_unate(next_level(f), lower_level_phase)             /* Process lower level FF */
        set_unate(same_level(f), phase)                         /* Process same level FF */
        annotate(f,phase)                                       /* Mark f as processed with that phase */
        return }

get_binate(f,v) {
        reset_unateness(v)
        f_0 = shannon_decomposition(equation,v,0)               /* Find Shannon cofactor of f with (v = 0) */
        f_1 = shannon_decomposition(equation,v,1)               /* Find Shannon cofactor of f with (v = 1) */
        fpu = f_1 + f_0'                                        /* Establish implication */
        if (fpu == 1) {                                         /* fpu is a tautology */
                set_unateness(v,POSITIVE)                       /* v is positive unate */
                return }
        fnu = f_0 + f_1'                                        /* Establish implication */
        if (fnu == 1) {                                         /* fnu is a tautology */
                set_unateness(v,NEGATIVE)                       /* v is negative unate */
                return }
        set_unateness(v,BINATE)                                 /* v is binate */
        return }
```

Figure 5.3: Algorithm for unateness extraction

```
extract_symmetry(equation) {
    symm_classes = new_classes()
    input_vars = support(equation)                          /* Get the list of inputs of eq */
    foreach(v ∈ input_vars) {                               /* Process each variable */
        if (is_tagged(v)) { continue }                      /* Skip v if already in a set */
        tag(v)                                              /* Mark v as processed */
        symm_set = new_set()
        add_to_symm_set(symm_set, v)
        foreach(u ∈ untagged_input_vars) {                  /* Compare with remaining variables */
            g1 = shannon_decomposition(equation,v,1)        /* Find Shannon cofactor of eq with (v = 1) */
            g0 = shannon_decomposition(equation,v,0)        /* Find Shannon cofactor of eq with (v = 0) */
            g10 = shannon_decomposition(g1,u,0)             /* Find Shannon cofactor of g1 with (u = 0) */
            g01 = shannon_decomposition(g0,u,1)             /* Find Shannon cofactor of g0 with (u = 1) */
            if (g10 ≡ g01) {                                /* Condition for (v, u) symmetry */
                add_to_symm_set(symm_set, u)                /* Add to current set */
                tag(u) } }                                  /* Mark u as processed */
        add_set_in_classes(symm_classes,symm_set) }
    return }
```

Figure 5.4: Algorithm for symmetry extraction

Circuit	SIS		Ceres					
			depth 3		depth 4		depth 5	
	cost	rtime	cost	rtime	cost	rtime	cost	rtime
C6288	1649	139.0	1425	85.6	1425	86.5	1425	87.1
k2	1308	4408.0	1209	131.9	1209	170.5	1209	253.3
C7552	1250	501.3	1117	97.4	1100	123.6	1062	178.3
C5315	957	264.8	899	79.2	887	107.2	831	151.2
frg2	941	869.3	1049	103.9	1045	141.6	844	193.1
pair	823	179.9	872	69.3	813	90.3	716	120.7
x1	761	630.1	827	95.4	825	150.6	807	328.5
C3540	658	295.0	641	45.3	628	62.0	608	101.9
vda	651	3103.5	543	46.1	543	61.4	543	91.7
x3	637	288.1	639	54.8	620	74.9	527	108.1
rot	570	254.7	583	48.4	582	73.8	551	137.5
alu4	521	1702.5	561	44.9	559	72.0	550	147.7
C2670	431	177.3	379	28.4	359	39.0	317	64.4
apex6	378	65.2	399	23.2	400	30.5	399	45.6
C1355	371	53.2	176	12.5	178	15.1	178	22.2
term1	360	368.0	380	29.3	365	44.6	302	73.0
x4	346	160.9	433	32.5	428	49.0	368	79.5
alu2	297	623.8	325	24.3	329	41.2	320	90.1
frg1	286	83.9	277	30.1	277	51.6	271	120.6
C1908	283	87.2	266	15.9	265	21.5	263	34.0
ttt2	217	167.9	283	23.4	283	37.5	249	69.0
C880	193	47.0	191	11.0	182	14.9	178	23.0
C499	178	51.8	176	9.9	176	14.1	168	21.1
example2	175	50.8	179	11.1	179	14.5	175	21.5
apex7	147	44.6	149	9.8	150	13.3	142	19.3
my_adder	128	38.6	112	8.8	96	11.8	64	14.4
C432	125	35.9	93	7.5	93	10.4	93	17.2
f51m	124	120.0	132	12.2	131	21.2	129	44.7
z4ml	106	96.8	113	10.2	113	17.4	91	33.7
c8	103	45.9	143	11.1	136	15.9	116	22.7
Total	14974 100 %	14955.0 1.0	14571 97.3 %	1213.4 0.08	14376 96.0 %	1677.9 0.11	13496 90.1 %	2715.1 0.18

Table 5.1: Mapping results for area (No *don't cares* , Actel library Act1, restricted set of gates commercially distributed)

Circuit	Ceres					
	depth 3		depth 4		depth 5	
	cost	rtime	cost	rtime	cost	rtime
C6288	1425	89.3	1425	89.2	1425	89.0
k2	1195	158.0	1195	205.4	1195	306.7
C7552	1119	116.8	1085	150.1	1045	250.2
C5315	908	102.0	765	141.8	715	245.8
frg2	1021	129.1	889	174.5	836	270.1
pair	842	88.7	769	122.3	686	180.2
x1	834	121.4	827	200.0	807	140.8
C3540	638	60.0	620	84.9	585	152.8
vda	526	59.4	526	81.3	526	124.2
x3	598	71.3	531	99.8	486	155.3
rot	563	65.3	536	100.3	521	190.3
alu4	559	60.4	546	99.0	520	203.4
C2670	377	39.8	337	56.6	304	106.1
apex6	368	31.2	357	41.2	347	62.2
C1355	178	18.5	178	22.1	176	37.0
term1	369	41.6	305	63.1	272	105.3
x4	396	45.5	360	66.8	342	110.3
alu2	321	35.1	312	58.5	312	122.7
frg1	285	41.3	284	71.5	272	162.5
C1908	270	23.8	269	31.7	265	56.4
ttt2	272	33.7	258	53.1	230	96.0
C880	184	17.9	182	23.5	178	37.1
C499	178	16.8	174	21.4	160	44.4
example2	173	17.4	168	22.4	162	32.1
apex7	146	16.3	134	21.4	125	30.8
my_adder	112	15.1	64	18.9	64	29.1
C432	100	13.3	100	17.6	100	27.8
f51m	128	19.5	123	32.1	112	65.0
z4ml	108	16.6	101	26.7	82	50.4
c8	131	17.9	117	24.4	111	35.6
Total	14324	1583.0	13537	2221.6	12961	3519.6
	100 %	1.0	94.5 %	1.4	90.5 %	2.2

Table 5.2: Mapping results for area (No *don't cares* , Complete Act1 library)

Circuit	Ceres					
	depth 3		depth 4		depth 5	
	cost	rtime	cost	rtime	cost	rtime
C6288	1454	89.9	1454	90.1	1454	89.9
k2	1285	156.8	1051	201.1	1051	304.7
C7552	1155	117.8	1061	154.2	1019	302.8
C5315	960	101.1	826	138.2	745	248.2
frg2	1122	132.8	822	193.2	696	291.2
pair	865	88.6	756	124.2	647	193.0
x1	850	124.7	769	199.6	747	442.7
C3540	600	58.8	574	91.3	540	162.4
vda	585	61.8	539	86.8	539	133.0
x3	670	74.2	526	107.0	449	162.0
rot	606	67.4	504	107.1	473	200.6
alu4	565	69.7	497	106.8	478	215.4
C2670	404	41.6	331	59.7	304	121.5
apex6	337	31.4	299	45.8	297	69.5
C1355	180	19.4	154	22.4	154	46.9
term1	407	43.3	295	69.1	252	116.8
x4	416	46.7	342	73.5	308	118.0
alu2	324	36.3	280	64.6	274	132.7
frg1	277	41.4	274	77.4	277	170.5
C1908	281	24.6	242	33.2	240	75.2
ttt2	293	35.3	235	58.5	207	103.0
C880	162	18.2	143	25.7	141	44.1
C499	178	16.8	140	22.4	130	62.5
example2	179	18.1	161	23.7	156	34.3
apex7	156	17.2	130	23.2	119	31.5
my_adder	127	15.9	64	20.7	64	27.7
C432	101	13.9	102	19.7	101	30.3
f51m	128	20.3	111	35.1	97	70.5
z4ml	98	17.4	89	29.1	80	58.0
c8	146	18.8	106	26.4	93	38.7
Total	14911 100.0 %	1620.2 1.0	12877 86.4 %	2329.8 1.4	12132 81.4 %	4097.6 2.5

Table 5.3: Mapping results for area (No *don't cares* , Complete Act2 library)

Circuit	SIS		Ceres					
			depth 3		depth 4		depth 5	
	cost	rtime	cost	rtime	cost	rtime	cost	rtime
C6288	2429	73.7	2241	53.6	2241	52.4	2241	55.9
k2	2182	149.0	2302	110.5	2302	185.3	2302	340.5
C7552	2699	127.1	2797	95.0	2780	139.0	2749	234.2
C5315	1959	86.1	2103	49.0	2002	114.7	1987	192.3
frg2	2045	122.7	1915	90.5	1869	149.1	1712	254.7
pair	1505	68.7	1529	61.4	1525	96.2	1483	165.7
x1	1410	103.1	1498	85.6	1498	175.7	1491	459.4
C3540	1192	67.6	1208	48.1	1201	78.5	1186	155.8
vda	1039	71.6	1090	45.6	1090	71.7	1090	119.5
x3	1285	68.0	1250	52.0	1254	81.2	1095	130.4
rot	1108	70.7	1120	50.7	1121	91.1	1122	107.1
alu4	1009	78.8	1015	48.5	1015	94.6	999	229.5
C2670	862	45.8	909	22.0	887	49.8	893	93.2
apex6	714	33.6	680	24.5	680	35.2	680	59.3
C1355	561	27.9	404	15.5	404	20.4	402	32.9
term1	721	44.9	706	33.6	687	57.3	573	107.2
x4	690	43.6	734	33.2	734	56.9	643	96.8
alu2	565	44.3	588	29.8	587	56.7	576	140.2
frg1	579	45.3	595	37.2	595	77.2	592	210.0
C1908	592	39.4	600	21.6	600	30.9	587	56.5
ttt2	429	33.5	425	23.1	421	39.2	377	73.3
C880	342	22.5	314	16.8	314	25.0	309	44.7
C499	421	24.6	406	14.0	406	18.7	404	29.2
example2	354	23.6	371	14.7	371	20.9	352	33.1
apex7	283	20.0	285	14.0	285	19.0	268	28.7
my_adder	242	17.8	224	13.4	223	18.1	216	24.4
C432	221	17.8	215	12.4	215	17.9	215	32.9
f51m	234	21.1	197	14.5	197	25.0	191	55.0
c8	237	19.1	221	12.6	218	16.7	215	23.7
cht	211	16.6	204	11.5	204	14.5	188	18.8
Total	28120 100 %	1628.5 1.0	28146 100.0 %	1154.9 0.7	27926 99.2 %	1928.9 1.2	27138 96.3 %	3604.9 2.2

Table 5.4: Mapping results for area (No *don't cares* , LSI Logic library)

Circuit	Ceres					
	depth 3		depth 4		depth 5	
	cost	rtime	cost	rtime	cost	rtime
C6288	1424	112.9	1424	112.8	1424	112.7
k2	1071	135.0	860	178.1	696	271.9
C7552	1017	110.8	917	141.8	845	289.6
C5315	823	89.4	667	118.4	566	240.8
frg2	953	115.4	701	159.8	580	247.3
pair	747	78.7	629	106.6	510	171.4
x1	779	100.6	595	163.9	479	370.5
C3540	539	51.8	467	75.0	439	165.5
vda	491	50.7	410	70.0	314	103.1
x3	599	64.2	431	85.6	367	138.3
rot	514	55.8	395	86.8	332	177.4
alu4	473	51.1	378	86.2	328	194.4
C2670	347	36.4	276	49.2	241	102.3
apex6	287	28.4	252	40.0	239	71.1
C1355	170	18.3	154	20.1	152	36.0
term1	339	35.7	251	55.1	200	98.1
x4	349	38.6	263	59.9	232	104.1
alu2	276	29.9	222	52.4	183	117.8
frg1	275	35.2	231	61.2	189	147.8
C1908	260	21.6	212	28.3	207	67.4
ttt2	240	29.6	189	51.0	168	93.2
C880	151	17.0	138	23.4	133	42.7
C499	170	15.4	154	19.4	152	47.5
example2	155	16.3	131	21.0	121	32.9
apex7	123	15.3	101	20.4	85	29.8
my_adder	112	14.5	64	18.0	64	25.6
C432	82	13.4	77	16.8	75	28.5
f51m	116	17.8	90	29.4	76	65.6
z4ml	96	15.2	66	24.6	60	51.8
c8	126	16.8	95	22.4	77	33.1
Total	13104 100 %	1431.8 1.0	10840 82.7 %	1997.6 1.4	9534 72.8 %	3678.2 2.6

Table 5.5: Mapping results for area (No *don't cares* , Library derived from the Quicklogic Master cell)

Chapter 6

Matching incompletely specified functions

We explained in Chapter 4 that powerful *matching* algorithms correlate to better quality solutions. One possible enhancement is to exploit *don't care* information while asserting logic equivalence. We presented in Chapter 4 a conceptually simple, albeit computationally complex matching algorithm, where *don't care* information is incorporated. Chapter 5 introduced a method to efficiently use Boolean techniques when matching completely specified functions. In the present chapter, we present an efficient method for matching logic functions in the presence of *don't care* information.

The topology of the Boolean network changes during the covering stage. As a result, *don't care* conditions must be dynamically established. Therefore a technology mapping algorithm that exploits *don't care* conditions must involve two tasks: i) computing and updating local *don't care* sets and ii) using the *don't care* information to improve the quality of the mapped circuit. We first present the use of *don't care* sets and we defer their computation to Section 6.3.

6.1 Using *don't care* information

We have considered two approaches to using *don't care* conditions in technology mapping. One uses Boolean simplification before matching a function to a library element, and the other merges simplification and matching in a single step. The former is motivated by the following rationale: *don't care* conditions are usually exploited to minimize the number of literals (or terms) of each expression in a Boolean network. While such a minimization leads to a smaller (and faster) implementation in the case of pluri-cell design style [BCD+88] (or PLA-based design), it may not improve the local area and timing performance in a cell-based design. For example, cell libraries exploiting pass-transistors might be faster and/or smaller than other gates having fewer literals. A pass-transistor based multiplexer is such a gate. Let us assume a function defined by its *on* set $\mathcal{F}$ and its *don't care* set $\mathcal{DC}$:

$$\begin{aligned} \mathcal{F} &= (a+b)c \\ \mathcal{DC} &= \overline{b}\overline{c} \end{aligned}$$

Then $(a+b)c$ is the representation that requires the least number of literals (3), and the corresponding logic gate is implemented in CMOS pass-transistor logic by 4 transistors plus two buffers, *i.e.* 8 transistors total. On the other hand, $a\overline{b}+bc$ requires one more literal (4), but it is implemented by only 4 pass-transistors and one inverter, *i.e.* 6 transistors total (Figure 6.1). Note also that in most cell-libraries $(a+b)c$ would be implemented as an AND-OR-INVERT gate, followed by an inverter, using $6+2=8$ transistors. However, $a\overline{b}+bc$ would still be implemented by pass-transistor logic.

A second example, taken from the Microelectronic Center of North Carolina (MCNC) benchmark *majority*, is also representative of the uses of *don't cares* during matching (Figure 6.2). In that example, the use of *don't cares* yields better matches, and gives an overall lower cost for the resulting circuit. Consider the cluster function $OUT = \overline{ac}\overline{d}+T$ that has an associated *don't care* set $\mathcal{DC} = Td + Tc$. Thus it can be re-expressed as $OUT = \overline{c}\overline{d}(a+T)$. The two expressions have the same number of literals (4), and

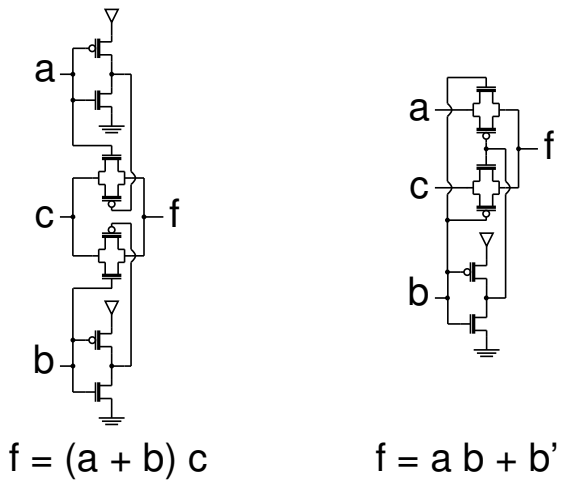


Figure 6.1: Two pass-transistor implementations of $\mathcal{F} = (a + b)c$, $\mathcal{DC} = \overline{b}\overline{c}$

therefore are equally likely to be chosen by a technology-independent simplify operation (which relies on literal count). But only one of the two exists in the library, and that match is essential in finding the best overall cost.

These examples show that applying Boolean simplification before matching may lead to inferior results, as compared to merging the two steps into a single task. For this reason, we directly use *don't care* sets in the Boolean matching step to search for the best implementation in terms of area (or timing).

6.2 Compatibility graph

Boolean matching that incorporates the *don't care* information can be done using the simple matching algorithm presented in Section 4.3. Unfortunately, when *don't care* conditions are considered, the cluster function $\mathcal{F}$ cannot be uniquely characterized neither by unateness properties nor by symmetry classes. Therefore the straightforward techniques, based on symmetry sets, presented in the previous section no longer apply. The simple

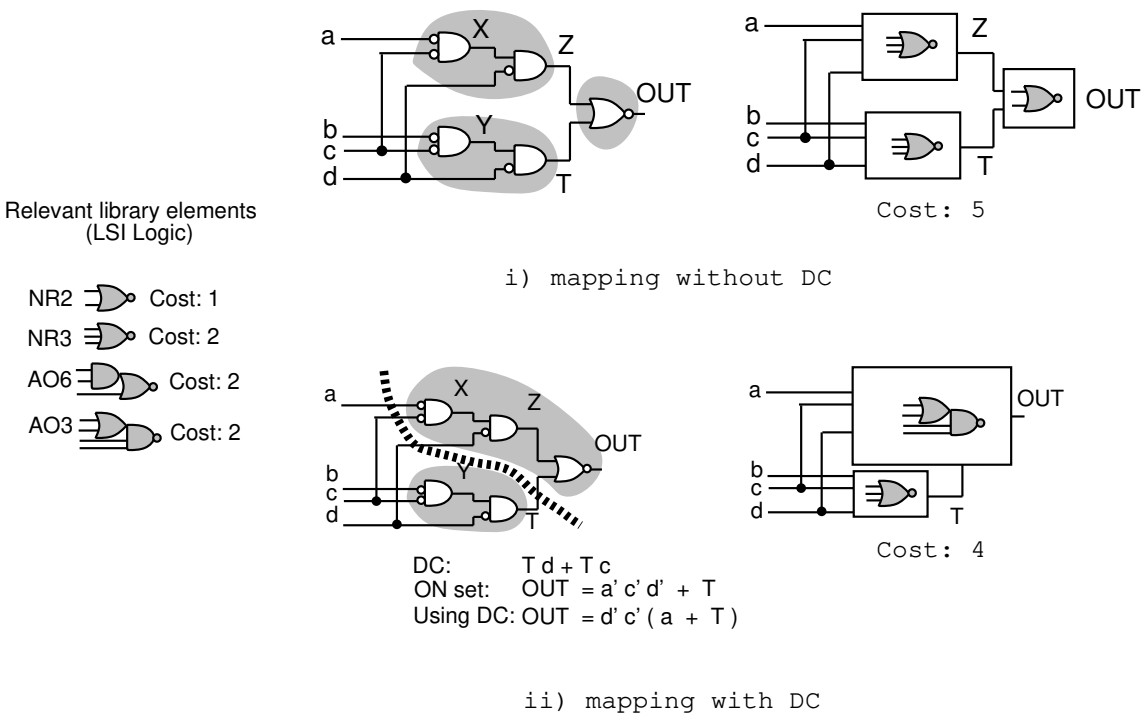


Figure 6.2: Mapping *majority* with LSI Logic library elements

matching algorithm would require in the worst case $n! \cdot 2^n$ variable orderings, each ordering requiring up to 2^n Shannon cofactorings. Therefore the algorithm is likely to be inefficient.

Another straightforward approach is to consider all the completely specified functions $\mathcal{H}$ that can be derived from $\mathcal{F}$ and its *don't care* set $\mathcal{DC}$. Functions $\mathcal{H}$ are easily computed by adding successively all possible subsets $D_i \in \mathcal{DC}$ to function $\mathcal{F}$. In this case, the symmetry sets can be used to speed-up matching. Unfortunately, there are 2^N possible subsets $D_i \in \mathcal{DC}$, where N is the number of minterms in $\mathcal{DC}$. Therefore this approach can be used only for small *don't care* sets. For large *don't care* sets, a pruning mechanism must be used to limit the search space.

We consider in this section a formalism that allows us to efficiently use *don't care* sets during matching. We first introduce a representation of n-variable functions that exploits

the notion of symmetry sets and NPN-equivalence (defined in Section 4.1) and that can be used to determine matches while exploiting the notion of *don't care* conditions. For a given number of input variables n, let $G(V, E)$ be a graph whose vertex set V is in one-to-one correspondence with the ensemble of all different NPN equivalent classes of functions. The edge set $E = \{(v_i, v_j)\}$ of the graph $G(V, E)$ denotes the vertex pairs such that adding a minterm to a function included in the NPN-class represented by v_i leads to a new function belonging to the NPN-class represented by v_j. Such a graph $G(V, E)$ for $n = 3$ is shown in Figure 6.3.

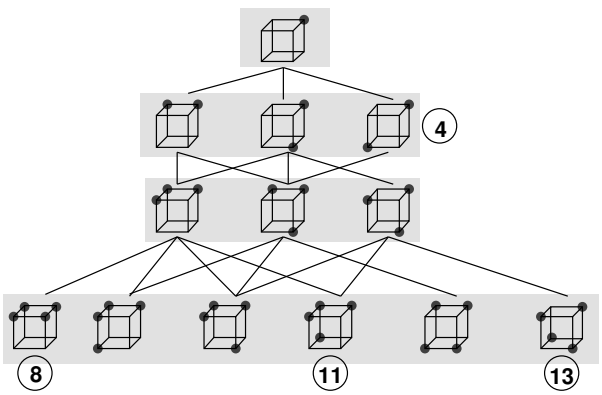


Figure 6.3: Matching compatibility graph for 3-variable Boolean space

Each vertex v_i in the graph is annotated with one function θ_i belonging to the corresponding NPN-equivalence class of v_i. The function θ_i is chosen arbitrarily among the members of the NPN-equivalence class that have the least number of minterms. For example, vertex v_4 in Figure 6.3 corresponds to functions $\{abc + \overline{a}\overline{b}\overline{c}, \overline{a}bc + a\overline{b}\overline{c}, \overline{a}\overline{b}c + ab\overline{c}, a\overline{b}c + \overline{a}b\overline{c}\}$ and their complements. The four functions listed contain 2 minterms, and their complements 6. The representative function θ_4 for vertex v_4 is $\{abc + \overline{a}\overline{b}\overline{c}\}$, but could be any of the 4 functions just enumerated. The set of functions θ_i is used as the basis for establishing relations between vertices v_i. Each vertex v_i is also annotated with the library elements, if any, that match the corresponding function θ_i. When multiple library elements are NPN-equivalent, they all correspond to the same vertex v_i in the compatibility graph.

The graph $G(V, E)$ is called *matching compatibility* graph, because it shows which matches are *compatible* with the given function. Note that the size of the compatibility graph is small for functions of 3 and 4 variables, where there are 14 and 222 different NPN-equivalent functions respectively [Mur71], representing the 256 and $65,536$ possible functions of 3 and 4 variables. Unfortunately, for functions of more than 4 variables, the number of NPN-equivalent functions grows very quickly (functions of 5 and 6 variables have $616,126$ and $\simeq 2 \times 10^{14}$ NPN-equivalent classes respectively [HMM85]), although it is very sparse in terms of the vertices corresponding to library elements. At present, we have implemented techniques for technology mapping using *don't care* conditions for cluster functions of at most 4 variables. From experimental results of mapped networks, we found that the majority of the library elements used have 4 or less variables (see Figure 6.1 for the distribution of the number of inputs of cells used for mapping 30 benchmarks). Therefore, it is a reasonable implementation decision to use *don't cares* only for cluster functions whose fanin is less or equal to 4.

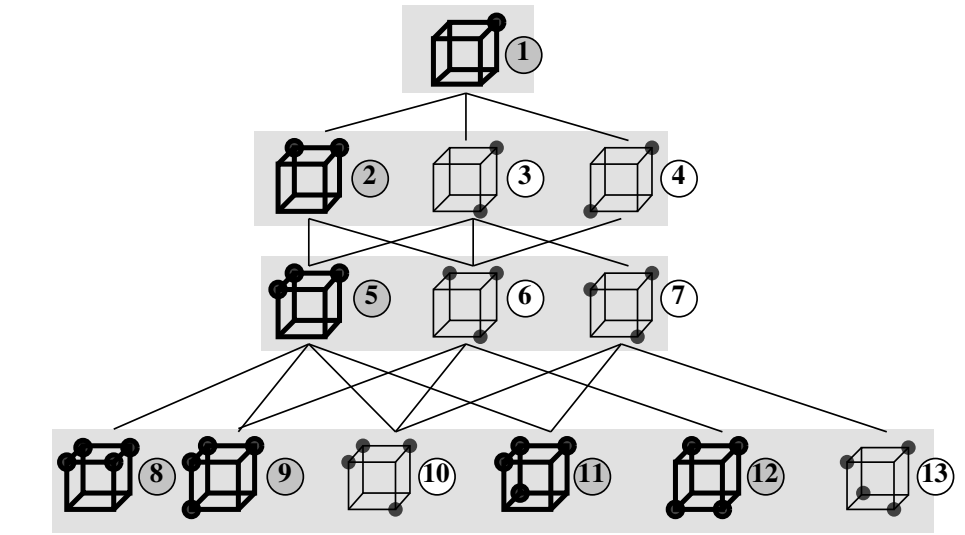
Number of inputs	Number of cells used		Percentage of total	
	Act1	LSI	Act1	LSI
1	305	2110	2.9 %	12.9 %
2	3176	6816	30.7 %	41.8 %
3	2998	2705	29.0 %	16.6 %
4	3685	4583	35.6 %	28.1 %
5	182	103	1.8 %	0.6 %
6	0	3	0.0 %	0.0 %

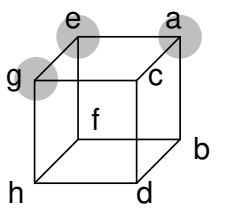
Table 6.1: Number of k-input cells used in mapping 30 benchmarks with the full Act1 and the LSI Logic libraries (depth = 5)

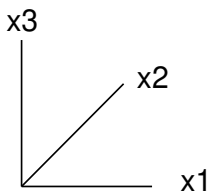
For functions of 4 variables and less, the compatibility graph is constructed once and annotated with the library elements. Each vertex v_i in the graph is also annotated with the paths p_{ij} from vertex v_i to a vertex v_j corresponding to library element $\mathcal{G}_j \in \mathcal{L}$. The set of paths $P_{i\mathcal{L}} = \{p_{i0}, p_{i1}, \ldots, p_{im}\}$ represents all the paths from vertex v_i to all the vertices corresponding to library elements. Each path represents the set of additional minterms differencing the function θ_i corresponding to v_i from the function θ_j of v_j,

where v_j corresponds to a library element. Therefore, checking if a function $\mathcal{F}$ is logically equivalent (modulo the *don't care* set $\mathcal{DC}$) to a library element $\mathcal{G}_j \in \mathcal{L}$ is the same as verifying that vertex v_i (corresponding to function $\mathcal{F}$) has some path p_{ij} to vertex v_j (corresponding to library element $\mathcal{G}_j$), such that the corresponding minterms are in the *don't care* set $\mathcal{DC}$.

For example, Figure 6.4 illustrates all the paths of the vertex labeled 5 in the *matching compatibility graph*. The 8 minterms of the 3-dimensional Boolean space are labeled a through h. The paths represent all the possible additions of minterms to the function θ_5 corresponding to vertex v_5. Note that some paths correspond to the complement of the representative function θ_j associated to the reached vertex v_j. This is indicated by a "~" preceding the vertex number. For this example, vertices $v_1, v_2, v_5, v_8, v_9, v_{11}$ and v_{12} are assumed to correspond to library elements. Therefore, of all the paths of vertex 5, only the shaded ones lead to vertices corresponding to library elements.







abcdefgh	
00000000	5
00000001	9
00000100	11
00000101	~5
00010000	10
00010001	~6
00010100	~7
00010101	~3
00100000	8
00100001	~5
00100100	~5
00100101	~2
00110000	~5
00110001	~2
00110100	~3
00110101	~1

abcdefgh	
01000000	9
01000001	~6
01000100	~6
01000101	~2
01010000	~6
01010001	~4
01010100	~3
01010101	~1
01100000	~5
01100001	~3
01100100	~2
01100101	~1
01110000	~2
01110001	~1
01110100	~1
01110101	constant 1

Figure 6.4: Paths from vertex 5 in the matching compatibility graph

Let us define $\mathcal{M}(v_i)$ as the number of minterms of the representative function θ_i of vertex v_i in a given N-dimensional Boolean space, and the distance between two vertices v_i and v_j as $\mathcal{D}(v_i, v_j) = \mid \mathcal{M}(v_i) - \mathcal{M}(v_j) \mid$. Then the number of paths from vertex v_i to any other vertex (including itself) of the compatibility graph is $2^{(2^N - \mathcal{M}(v_i))}$. This number simply indicates that a Boolean function of N variables can have at most 2^N minterms, of which $\mathcal{M}(v_i)$ are already allocated. Therefore only $2^N - \mathcal{M}(v_i)$ minterms can be added, and there are $2^{(2^N - \mathcal{M}(v_i))}$ possible permutations of these additional minterms.

For a 4-variable compatibility graph, the total number of paths for the entire network is 375,552. This is reasonable from an implementation point of view, since each path is represented by 16 bits, and thus the entire set of paths occupies approximately 750 KBytes. Note that in general not all paths must be stored, since the elements of the library usually represent only a subset of all possible NPN-equivalent classes.

If we consider all possible combinations of minterms, the maximum number of paths $\mid P_{ij} \mid$ between vertices v_i and v_j is

$$\mid P_{ij} \mid = \begin{pmatrix} 2^N - \mathcal{M}(v_i) \\ \mathcal{D}(v_i, v_j) \end{pmatrix} + \begin{pmatrix} 2^N - \mathcal{M}(v_i) \\ 2^N - \mathcal{M}(v_i) - \mathcal{M}(v_j) \end{pmatrix}$$

The first term of the expression for $\mid P_{ij} \mid$ represents all the combinations of minterms that can make a function of $\mathcal{M}(v_i)$ minterms into a function of $\mathcal{M}(v_j)$ minterms, in a N-dimensional Boolean space. The second term of the expression represents the combinations of minterms that yield a function of $2^N - \mathcal{M}(v_j)$, *i.e.* the complement of the functions computed for in the first part. Although this upper bound function grows exponentially, experimental results show that the actual number of paths between any pair of vertices is much smaller. For the 4-variable compatibility graph, the maximum number of paths between any two vertices is 384, corresponding to vertices $v_i = abcd$ and $v_j = abcd + \overline{a}(b\overline{d} + \overline{c}d)$. Given that $\mathcal{M}(v_i) = 1$ and $\mathcal{M}(v_j) = 5$, it is clear that the actual number of paths is much smaller than the worst case of 4004 calculated with the above formula. This is due to the fact that not all combinations of added minterms will make function θ_i logically equivalent to θ_j. In some cases, it is even impossible to reach some library element v_j from vertex v_i. For example, in Figure 6.3, vertex v_4 cannot reach vertices v_8, v_{11}, v_{13}. In addition, some paths do not need to be recorded, because their head vertex does not correspond to a library cell.

The matching of a cluster function $\mathcal{F}$ to a library element is redefined in terms of the compatibility graph as follows. For cluster functions with no applicable *don't care* set, only procedure *boolean_match* is used. Otherwise, procedure *boolean_match* is used to find the vertex $v_{\mathcal{F}} \in G(V, E)$ corresponding to the NPN-equivalence class of cluster function $\mathcal{F} \cap \overline{\mathcal{DC}}$ (*i.e.* , the ON-set of cluster function $\mathcal{F}$). Since the graph represents all possible functions of 4 or less variables, there exists a vertex in the graph which is NPN-equivalent to $\mathcal{F}$. At the same time vertex $v_{\mathcal{F}}$ is found, the algorithm computes the transformation $\mathcal{T}$ representing the input ordering and phase assignment on the inputs and output such that $\mathcal{T}(\mathcal{F}) = \theta_{\mathcal{F}}$. The transformation $\mathcal{T}$ is applied to the *don't care* set $\mathcal{DC}$, to generate a new expression, $\mathcal{T}(\mathcal{DC})$, consistent with the representative function $\theta_{\mathcal{F}}$ of $v_{\mathcal{F}}$. There exists a match to the library cell $\mathcal{G}$ if there is a path in the graph $G(V, E)$ from $v_{\mathcal{F}}$ to $v_{\mathcal{G}}$ (possibly of zero length) whose edges are included in the image $\mathcal{T}(\mathcal{DC})$ of the *don't care* set $\mathcal{DC}$ of $\mathcal{F}$. It is necessary that *don't care* sets are transformed by the operator $\mathcal{T}$ before the path inclusion is checked, because paths in the *compatibility* graph are computed between representative functions θ_i.

The algorithm for graph traversal is shown in Figure 6.5. It is invoked with the vertex found by algorithm *boolean_matching* and the image $\mathcal{T}(\mathcal{DC})$ of the corresponding *don't care* set as parameters. When finished, the algorithm returns the list of all the matching library elements, among which the minimum-cost one is chosen to cover $\mathcal{F}$.

```
dc_match(f,dc) {
    vertex = get_vertex(f)                                  /* Find starting point in the compatibility graph */
    NPN_orientation = cast_to_same_NPN_class(vertex,f)      /* Find how f and the vertex are related */
    NPN_dc = change_NPN(dc,NPN_orientation)                 /* Transform dc as f was into the vertex function */
    for (all paths p_i in vertex) {                         /* Find paths to library elements covered by dc */
        if (included(p_i,NPN_dc)) {
            update_cover_list(vertex,cover_list) } }
    return(cover_list) }
```

Figure 6.5: Algorithm for compatibility graph traversal using *don't cares*

6.3 Computation of relevant *don't care* sets

Don't care sets are classified into two major categories: external DCs, and internal DCs. External DCs are assumed to be provided by the user along with the network specification. They represent conditions that never occur on the primary inputs of the circuit, and conditions that are never sampled on the primary outputs. Internal DCs occur because of the Boolean network structure. They are further classified into *controllability don't cares* and *observability don't cares*. Controllability *don't cares* represent impossible logic relations between internal variables. Observability *don't cares* represent conditions under which an internal vertex does not influence any primary output.

The existence of controllability and observability *don't care* sets represents two different (but complementary) aspects of a network. Controllability *don't care* sets are related to the logic structures in the transitive fanin of a vertex, whereas observability *don't care* sets are related to the logic structures in the transitive fanout of a vertex in the Boolean network. The dynamic programming formulation of technology mapping implies the network to map is modified starting at the primary inputs, and is completed when all primary outputs are processed. The technology mapping operation modifies the logic structure of the network, and potentially modifies the internal *don't care* sets. Therefore, *don't care* sets should be calculated dynamically, as the boundary of the mapped network moves from primary inputs to primary outputs (Figure 6.6).

Controllability *don't care* sets are conceptually easily computed: a vertex is being mapped only when all its predecessors are mapped. Then all the logic functions expressing a vertex are known, and it is straightforward to extract the controllability *don't care* sets from them. For a subset of variables Y, that are inputs to a gate or a subnetwork, the controllability *don't care* sets (CDC) represent the impossible patterns for the variables Y. The CDC sets can be derived from the satisfiability *don't care* sets (SDC) by taking the iterated consensus of SDC on the variables different from Y[1], where the satisfiability *don't care* set is defined as $SDC = \sum i \oplus \mathcal{F}_i$ [BBH+88]. For example, for $x = ab$, the satisfiability *don't care* is $SDC_x = x \oplus (ab) = \overline{x}ab + x(\overline{a} + \overline{b})$. Controllability *don't care* sets can be computed in a straightforward manner from SDC for a particular subset of

[1]Recall that a Boolean network is defined by a set of equations $i = \mathcal{F}_i$. Therefore the condition $(i \neq \mathcal{F}_i) = i \oplus \mathcal{F}_i$ can never occur.

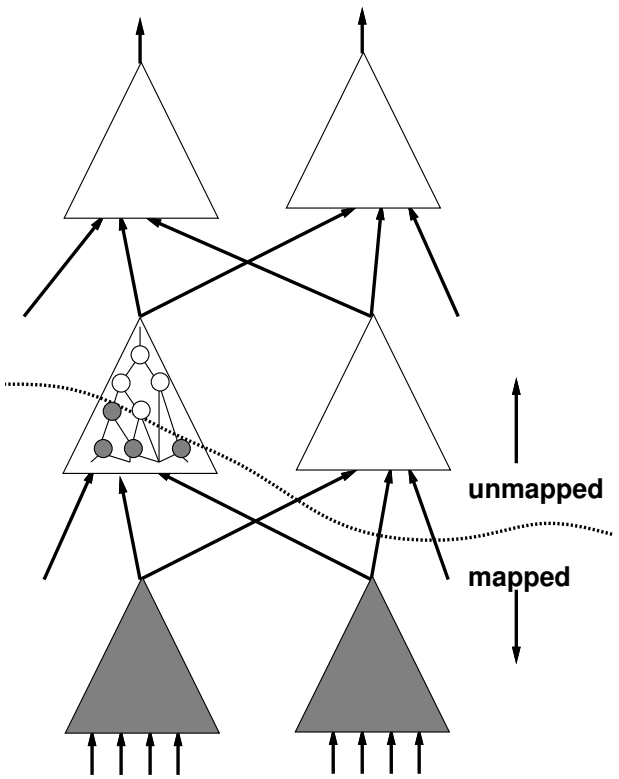


Figure 6.6: Example of a partially mapped network

variables $\{a, b, c, \ldots\}$. Given the satisfiability *don't care* set $SDC = \sum i \oplus \mathcal{F}_i$, each variable j of SDC not in the cutset $\{a, b, c, \ldots\}$ is eliminated by intersecting $SDC\ |_j$ with $SDC\ |_{\overline{j}} = CDC_{\{a,b,c,\ldots\}}(f) = \prod_{i \notin \{a,b,c,\ldots\}} (\sum j \oplus \mathcal{F}_j)\ |_i \cdot (\sum j \oplus \mathcal{F}_j)\ |_{\overline{i}}$. The controllability *don't care* sets are computed dynamically as the mapping operation proceeds through the network (Figure 6.7).

Observability *don't care* sets deal with the successors of vertices. They denote conditions under which a variable does not influence any primary output. For example, in the following network: $x = at, t = b + c$, t is unobservable when $a = 0$ (in that case, $x = 0$ regardless of the value of t). By the very nature of dynamic programming techniques, when a vertex is being processed, its successors are not yet mapped. This

```
update_cdc(vertex,current_cdc) {
        sdc = recursive_xor(vertex,support(current_cdc))                    /* Add all SDCs between vertex and CDC cut set */
        unused_support = unused_support_recursive(vertex,support(current_cdc) /* Get list of cut set vertices now covered by vertex */
        new_cdc = sdc ∪ current_cdc                                         /* Update CDC with SDC previously calculated */
        new_cut_set = vertex ∪ {support(current_cdc) \ unused_support}      /* Update CDC cut set */
        for (v ∈ support(new_cdc), but ∉ new_cut_set)
                new_cdc = consensus(new_cdc, v)                             /* Eliminate unused variables */
        return(new_cdc) }


recursive_xor(eq,vertex,list) {
        eq = equation(vertex)
        sdc = xor(eq,vertex)
        for (v ∈ support(eq), but not in list)
                sdc = sdc ∪ recursive_xor(v,list)
        return(sdc) }


unused_support_recursive(vertex,list) {
        eq = equation(vertex)
        for (v ∈ support(eq) and in list)
                if (last_fanout(v) = vertex)
                        unused_support = unused_support ∪ v
        for (v ∈ support(eq) and not in list)
                unused_support = unused_support ∪ unused_support_recursive(v,list) }
```

Figure 6.7: Algorithm for dynamic controllability *don't cares* calculation

implies the exact observability of a vertex is known only after the mapping is completed. Note that unless the observability *don't cares* are recomputed each time a vertex is modified, it is not possible to use the full ODC set for all the vertices [CM89]. Therefore, compatible subsets of the ODC must be used, as described in [SB90]. Although good algorithms have been proposed to compute compatible observability *don't care* sets [DM91, DM90, SB91], efficient implementations are far from trivial, and we decided not to include them at present. The results reported in Section 6.4 therefore represent only the use of controllability *don't care* sets.

6.3.1 Image-based computation of *don't care* sets

Procedure *update_cdc* of Figure 6.7 has exponential complexity. This exponential behavior occurs because the iterated consensus operation involves recursively establishing the

product $\mathcal{F}_{x_i} \cdot \mathcal{F}_{\overline{x_i}}$ for every variable x_i to be eliminated from $\mathcal{F}$. Note that the consensus operation is also known as the universal quantification: $\forall x_i \; \mathcal{F} \equiv \mathcal{F}_{x_i} \cdot \mathcal{F}_{\overline{x_i}}$. We call $\widetilde{\mathcal{F}}$ the function derived from $\mathcal{F}$ using that procedure, *i.e.* $\widetilde{\mathcal{F}}$ is obtained from $\mathcal{F}$ by eliminating a subset of the input variables of $\mathcal{F}$. The final expression of $\widetilde{\mathcal{F}}$ involves the product of up to 2^{n_e} multiplicands, where n_e is the number of variables eliminated from $\mathcal{F}$. As a result, the consensus operation is not usable for functions with a large number of variables to eliminate.

Coudert *et al.* introduced in 1989 an efficient method for implicit state enumeration in finite state machines [CBM90b, CBM90a], which can be adapted for calculating *don't care* conditions. The method is based on image computation, and it is an efficient way of establishing the characteristic function of the image of a Boolean function. Touati and Savoj have also presented work inspired by the method of Coudert *et al.* [TSL+90, SBT91, Sav91]. We will now review this method and its foundations, and present a useful extension for efficiently calculating subsets of CDC.

When matching a cluster function $\mathcal{F}(x_1, \ldots, x_m)$, we considered an *associated* controllability *don't care* set $\mathcal{DC}(x_1, \ldots, x_m)$ both in algorithms *simple_boolean_match* and *dc_match*. The *don't care* information used in those algorithms must ultimately be expressed as the impossible relations between the inputs of the cluster function being matched. Given that $\mathcal{F}(x_1, \ldots, x_m)$ belongs to a n-input Boolean network $\mathcal{N}$, the set of inputs $\{x_1, \ldots, x_m\}$ of $\mathcal{F}$ can be expressed as the m primary outputs of a subnetwork $\mathcal{N}'$ of $\mathcal{N}$, where the Boolean behavior of $\mathcal{N}'$ is a multi-output function $\Phi : \mathcal{B}^n \rightarrow \mathcal{B}^m$ (Figure 6.8).

Since cluster function $\mathcal{F}$ takes as its inputs the outputs of subnetwork $\mathcal{N}'$, the *don't care* set $\mathcal{DC}$ is simply the set of vectors of $\mathcal{B}^m$ that are not in the *range* of Φ, the m-output logic function representing the Boolean behavior of the subnetwork[2]. The characteristic function $\chi(\Phi)$ is a function of $\mathcal{B}^m \rightarrow \mathcal{B}$, whose value is 1 for vectors of $\mathcal{B}^m$ in the range of Φ, and 0 otherwise. Therefore, the *don't care* information can be expressed simply as $\mathcal{DC} = \overline{\chi(\Phi)}$.

Let us represent $\Phi : \mathcal{B}^n \rightarrow \mathcal{B}^m$ by a vector of single-output Boolean functions: $\Phi = [h_1, h_2, \ldots, h_m]$, each individual function defined as $h_i : \mathcal{B}^n \rightarrow \mathcal{B}$ (the inputs to

[2]Recall the *range* of a function is defined as the image of its entire domain, here $\mathcal{B}^n$

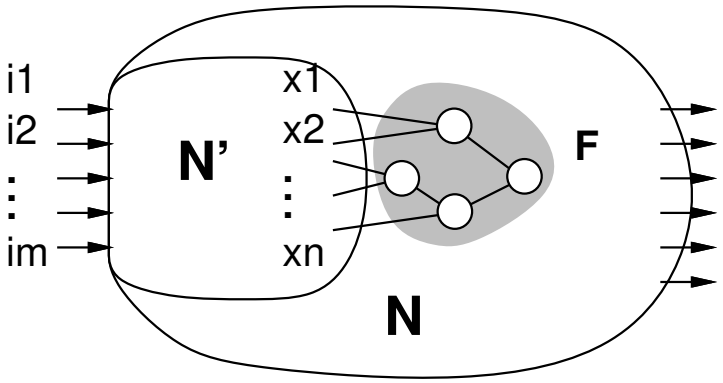


Figure 6.8: Subnetwork generating the inputs of F

each function h_i are the n primary inputs of network $\mathcal{N}$). Recall that the m functions $\{h_1, \ldots, h_m\}$ correspond to the m vertices $\{v_1, \ldots, v_m\}$ of the Boolean network which are related to the inputs[3] $\{x_1, \ldots, x_m\}$ of cluster function $\mathcal{F}$. Therefore, h_i represents the functionality of its associated variable x_i as a function of the primary inputs of the Boolean network. The algorithm introduced by Coudert *et al.* is an efficient method to extract the characteristic function $\chi(\Phi)$ given the vector expression $\Phi = [h_1, h_2, \ldots, h_m]$. It finds all the minterms $x_1^{\phi_1} x_2^{\phi_2} \cdots x_m^{\phi_m}$ of the characteristic function $\chi(\Phi)$ corresponding to every point of $\mathcal{B}^m$ that is in the range of $\Phi = [h_1, h_2, \ldots, h_m]$.

The characteristic function $\chi(\Phi)$ is expressed as $\chi(x_1, \ldots, x_m)$, a function of the m inputs of cluster function $\mathcal{F}$. Each variable x_i of χ corresponds to a function h_i in Φ. The minterms $x_1^{\phi_1} x_2^{\phi_2} \cdots x_m^{\phi_m}$ of χ are extracted by establishing the possible values of functions h_j of Φ. Variable x_i is 1 (0) if its corresponding function h_i is 1 (0), but function h_i is 1 (0) only for a subset σ_i of the domain $\mathcal{B}^n$. Therefore, when h_i has a certain value (say 1), it implies a restriction σ_i of its domain. Restriction σ_i applies to all the other functions h_j of Φ, since all functions of Φ are evaluated in parallel. This restriction of the domains of functions h_j of Φ given the value of one function h_i is the basis for the calculation of the characteristic function χ.

[3] In the rest of this section, we use x_i and v_i interchangeably, for the sake of simplicity.

The extraction of $\chi(\Phi)$ from Φ involves two operations. The first one involves the successive assertion of the functions h_i of Φ to 1 (0), and the derivation of an expression for the minterms of χ in terms of the variables x_i. This effectively computes the range of Φ. The second operation occurs during the range extraction, and involves the calculation of a function h_j given the value of another function h_i.

The first operation can be formally described as follows. Calling $\mathcal{R}^k$ the transformation of a function $\Phi : \mathcal{B}^n \rightarrow \mathcal{B}^k$ into its characteristic function $\chi(\Phi) : \mathcal{B}^k \rightarrow \mathcal{B}$, then $\mathcal{R}^k([h_1, \ldots, h_k]) = x_k \cdot \mathcal{R}^{k-1}([h_1 \mid_{h_k}, \ldots, h_{k-1} \mid_{h_k}]) + \overline{x_k} \cdot \mathcal{R}^{k-1}([h_1 \mid_{\overline{h_k}}, \ldots, h_{k-1} \mid_{\overline{h_k}}])$ [CBM90b]. The previous equation indicates that x_k follows the value of h_k, and that the portion of the domain for which h_k has that value restricts the domain of the remaining functions $h_j, j \neq k$. Transformation $\mathcal{R}^k$ is applied recursively until $k = 1$. Note that in the particular case that function h_k is a tautology (*i.e.* $h_k \equiv 1$), $\mathcal{R}^k([h_1, \ldots, (h_k \equiv 1)]) = x_k \cdot \mathcal{R}^{k-1}([h_1, \ldots, h_{k-1}])$, because when h_k is tautologically true, x_k cannot be 0. Furthermore, h_k being a tautology implies h_k is 1 regardless of its inputs, and therefore imposes no constraint on its domain. Similarly, for h_k tautologically false (*i.e.* $h_k \equiv 0$), $\mathcal{R}^k([h_1, \ldots, (h_k \equiv 0)]) = \overline{x_k} \cdot \mathcal{R}^{k-1}([h_1, \ldots, h_{k-1}])$.

The second operation was called *constraint* in [CBM90b, CBM90a]. Touati *et al.* realized the operation is analogous to Shannon cofactor extraction, and renamed it *generalized cofactor* [TSL+90]. In a traditional Shannon cofactor, the domain of a function F is reduced in half by considering only the subset of the domain where a particular variable x is 1 (or 0). The value of the function F for the other half of the original domain (*i.e.* corresponding to $\overline{x}$) becomes irrelevant, and is assumed to take the same value as that of the restricted domain (Figure 6.10). Then, the traditional formula $F = x \cdot F \mid_x + \overline{x} \cdot F \mid_{\overline{x}}$ is obtained. This concept is extended to restricting the domain of F to the subset making a given function G equal to 1 (or 0). In that case, $F = G \cdot F \mid_G + \overline{G} \cdot F \mid_{\overline{G}}$ (Figure 6.10). Note that in general $F \mid_G$ contains *don't cares* , because the domain of $F \mid_G$ is the subset of the domain of F for which G is equal to 1. Therefore, the subset of the domain of F for which G is equal to 0 corresponds to *don't cares* in $F \mid_G$. These *don't cares* are used to simplify the expression of $F \mid_G$. For example, in Figure 6.10, the general cofactor $F \mid_{\overline{G}}$ is simplified to $F \mid_{\overline{G}} = 1$ when considering the *don't cares*. We summarize the algorithm for the CDC computation in the following pseudocode, adapted from [TSL+90]

(Figure 6.9). Note that BDDs are the preferred representation for all the operations on logic functions carried out in the algorithm.

```
get_cdc(F) {
    input_list = support(F)                          /* Get list of inputs of F
    Φ = bdd_vector(input_list)                       /* Get BDDs of inputs of F in terms of Primary Inputs */
    care_set = R^k-transform(Φ)                      /* Get characteristic function of Φ
    DC = complement(care_set)                        /* DC set is complement of χ(Φ) }

R^k-transform([h_1,...,h_k]) {                       /* Φ is an array of single-output functions */
    if (k == 0) return(1)                            /* The transformation is complete */
    foreach(h_i) {                                   /* If any function is a constant, eliminate it */
        if (h_i ≡ 1) {                               /* h_i is the constant 1 */
            return(x_i · F-transform([h_1,...,h_{i-1},h_{i+1},...,h_k])) }
        if (h_i ≡ 0) {                               /* h_i is the constant 0 */
            return(x̄_i · F-transform([h_1,...,h_{i-1},h_{i+1},...,h_k])) } }
    return(x_k · F-transform([h_1 |_{h_k},...,h_{k-1} |_{h_k}])
    +x̄_k · F-transform([h_1 |_{h_k},...,h_{k-1} |_{h_k}]) }

general_cofactor(F,G) {
    F_G = F · G                                      /* Restrict domain of F to that of G */
    DC = Ḡ                                           /* What is not in the domain of G is don't care */
    if (F ∪ DC ≡ 1) return(1)
    return(simplify(F_G,DC) }                        /* Try to eliminate variables from F_G */
```

Figure 6.9: Algorithm for image-based *don't care* calculation

As an example of the above algorithm, let us assume $\Phi = [h_1, h_2]$, where $h_1 = ab$ and $h_2 = a + b$. We assume variables x_1 and x_2 correspond to functions h_1 and h_2. Then,

$$\begin{aligned}
\chi(\Phi) &= \mathcal{R}^2([h_1, h_2]) \\
&= x_2 \cdot \mathcal{R}^1([h_1 \mid_{h_2}]) + \overline{x_2} \cdot \mathcal{R}^1([h_1 \mid_{\overline{h_2}}]) \\
&= x_2 \cdot \mathcal{R}^1([ab \mid_{a+b}]) + \overline{x_2} \cdot \mathcal{R}^1([ab \mid_{\overline{a}\cdot\overline{b}}]) \\
&= x_2 \cdot \mathcal{R}^1([h_1]) + \overline{x_2} \cdot \mathcal{R}^1([0]) \\
&= x_2 \cdot (x_1 + \overline{x_1}) + \overline{x_2} \cdot \overline{x_1} \\
&= x_2 + \overline{x_1} \\
\Rightarrow \mathcal{DC} &= \overline{\chi(\Phi)} = x_1\overline{x_2}
\end{aligned}$$

Therefore, we see that it is impossible for the two-input AND $h_1 = ab$ to be 1 when the two-input OR $h_2 = a + b$ is 0.

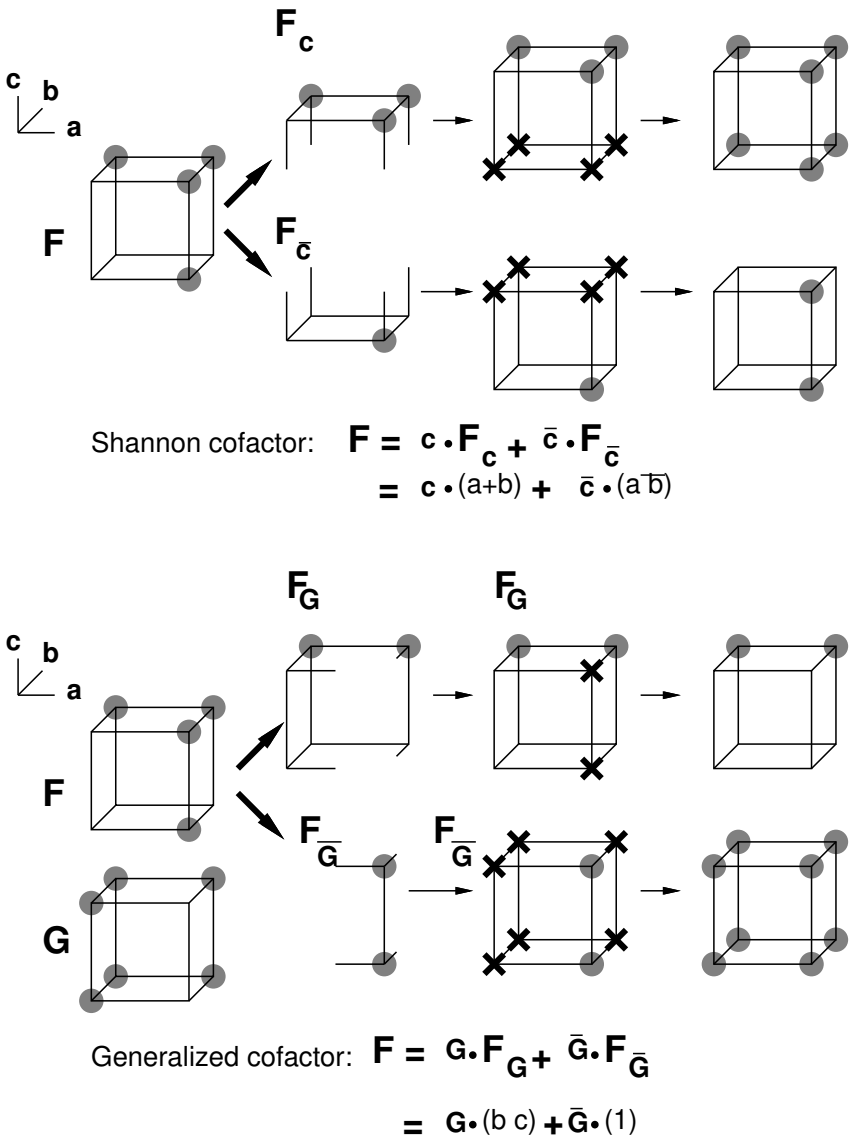


Figure 6.10: Shannon cofactor and generalized cofactor

6.3.2 Approximations in image computations

There is one problem with the image-based computation as presented in the previous section. We assumed that the functions h_i were expressed as a function of the primary inputs of the network. Unfortunately, for many Boolean networks, it is impossible to express the function of every vertex in the network as a function of the primary inputs. In particular, even for a very efficient logic representation like the BDDs, it was shown that certain functions require a BDD whose size is exponential in the number of inputs [Bry86].

Recently, Berman related the size of a BDD to the topology of the Boolean network it represents [Ber91b]. He established a bound on the number of nodes in a BDD, based on the *width* of a network. Specifically, the size of the BDD cannot exceed $n \times m \times 2^{w(\mathcal{N})}$, where $\mathcal{N}$ is a Boolean network with n inputs and m outputs, and a width $w(\mathcal{N})$. The width $w(\mathcal{N})$ represents the cardinality of the largest cutset through the network $\mathcal{N}$ (Figure 6.11).

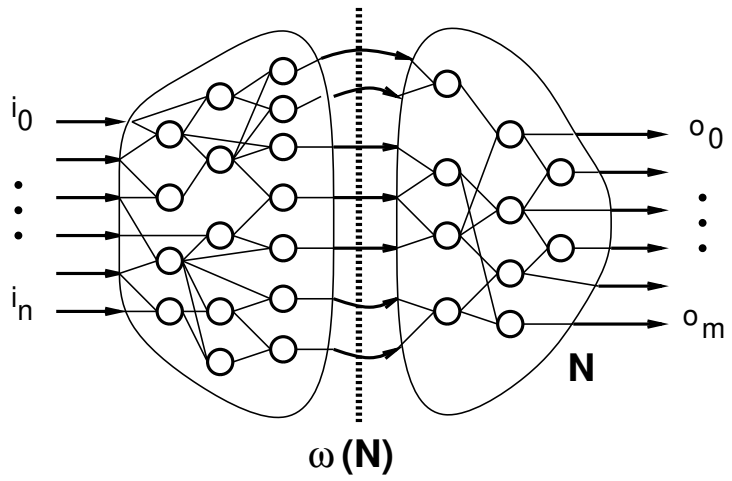


Figure 6.11: Width of a Boolean network $\mathcal{N}$

For matching purposes, *don't care* information represents a source of additional degrees of freedom. It provides a mean of possibly finding more matches, and therefore of potentially improving the quality of the results. Although the use of the entire *don't*

care information entails the most benefits, the matching method does not require the *don't cares* to be complete. Therefore, when BDDs cannot be efficiently calculated, we propose a simplification which reduces the size of the BDDs, at the cost of relinquishing the completeness of the *don't care* information.

Since the bound on the size of a BDD grows rapidly with the width of a Boolean network, we propose a simple heuristic to reduce the width of the networks from which BDDs are calculated. Assume a network whose vertices are topologically ordered. The cutset through the vertices at a certain level k is larger than the cutset at the previous level $k-1$ only if some vertices at level $k-1$ have more than one fanout. This means multiple-fanout vertices contribute to the increase of the width of a circuit, and therefore to the increase in the size of the corresponding BDD. Our simple heuristic consists of expressing the functions h_i using the closest multi-fanout vertices in their transitive fanins. This simplification effectively uses only subnetworks with limited width to extract the BDDs used during the image calculation (Figure 6.12).

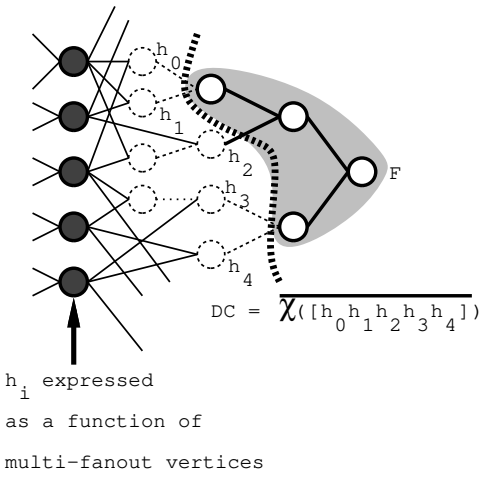


Figure 6.12: Illustration of *don't cares* limiting heuristic

6.3.3 Relations to testability

Note that when $(\mathcal{DC} \cup \mathcal{F} = 1)$ (or $(\mathcal{DC} \cup \overline{\mathcal{F}} = 1)$, then the algorithm finds a match with the constant value 1 (0 in the second case). This is always preferred to any other match, since it has a cost of 0. As a result, for every cell mapped to a library element there exists at least two *controllable* input patterns (*i.e.* it is possible to generate these patterns from the primary inputs), such that the output of the cell is 0 for one pattern and 1 for the other. This is a necessary condition to make a network testable. Assume that the library cells consist of testable gates (*i.e.* such that internal nodes are controllable and observable from the cell input/output pins). Then our method guarantees that the mapped circuit is 100% testable for stuck-at faults with the recently proposed I_{DDQ} testing method [FSTH90, CBL91]. However, cell controllability is not sufficient for achieving 100% testability by using standard testing methods. Indeed, it is possible that the output of a cell is not *observable* at the primary outputs when the *controllable* input patterns are applied to that cell. But by using a post-processing step involving standard ATPG and redundancy removal techniques [BBL89], the mapped network can be made 100% testable for single stuck at faults. The post-processing step could in principle be eliminated by computing observability *don't care* conditions. In practice this goal is hard to achieve, since the network is mapped from primary inputs to primary outputs and the observability of a vertex being mapped depends on portions of the network yet to be mapped.

6.4 Results

Tables 6.2 and 6.3 show mapping results for area, taking *don't care* information into account. For both technologies, the results show that using *don't cares* during the technology binding operation improves the quality of the results when operating on both optimized and unoptimized circuits. Circuits in the first category were optimized using UC Berkeley's *sis* with the standard script, which involves technology independent operations. It is worth noticing that the results of operating on non-optimized circuits using *don't care* information sometimes are better than the ones of optimized circuits mapped

without using *don't cares*. This indicates that the use of *don't care* during the technology mapping phase effectively allows logic restructuring, which is traditionally thought of as a technology-independent operation. Calculation of *don't care* is computationally intensive, and in some cases we used subsets of the full *don't care* set. Note that most of the execution times, when using *don't cares*, is spent calculating the *don't care* information. In Tables 6.2 and 6.3, the top portion represents results using subsets of the *don't care* sets, and the bottom portion represents results using the full *don't care* sets.

Note that we used UC Berkeley's BDD package for calculating the *don't cares*. We found the operations on BDDs to be very efficient. Unfortunately, the use we made of that code, creating and freeing tens of thousands of BDDs during the dynamic *don't care* set extraction, unveiled major memory leaks. The leaks eventually made many processes too large to be handled on our machines. Therefore, we report here only the results of a subset of the benchmarks used in tables 5.1 and 5.4.

	Original circuits				Optimized circuits			
	No DC		With DC		No DC		With DC	
Circuit	area	rtime	area	rtime	area	rtime	area	rtime
t481	1867	748.9	1495	12520.0	31	3.4	28	28.6
frg2	844	194.7	774	2960.7	369	33.4	361	463.3
x1	807	327.5	761	2979.2	172	35.3	166	171.9
alu4	550	147.5	443	1356.9	457	111.3	410	1151.3
apex6	399	45.9	387	431.8	451	77.0	420	682.2
i6	348	25.0	346	607.7	150	18.0	150	199.8
C1908	263	34.1	248	7687.3	195	26.2	193	7279.5
x4	368	79.4	329	517.4	182	13.9	179	140.1
term1	302	73.2	237	721.0	104	27.2	102	146.7
frg1	271	120.0	249	715.2	106	37.4	106	181.2
alu2	320	90.1	254	369.2	253	65.0	212	303.7
ttt2	249	70.9	147	262.2	92	14.3	89	74.3
i5	178	18.7	178	67.8	66	4.1	66	9.2
example2	175	22.0	150	92.5	160	17.9	149	105.9
c8	116	10.8	74	92.5	66	5.6	54	57.2
apex7	142	19.4	122	105.6	129	18.9	102	166.9
cht	124	22.2	111	116.5	84	5.0	84	32.9
9symml	94	33.2	94	114.8	101	36.8	103	116.4
z4ml	91	34.6	35	119.5	33	8.5	25	34.5
sct	83	17.2	80	53.3	35	6.1	33	65.2
lal	77	12.7	75	35.3	39	5.9	37	50.8
Total	7668	2148.0	6589	31926.4	3275	571.2	3069	11461.6
	1.0	1.0	0.86	14.9	0.43	0.27	0.40	5.3
					1.0	1.0	0.94	20.1

Table 6.2: Mapping results for area (Using *don't cares* , Actel library)

	Original circuits				Optimized circuits			
	No DC		With DC		No DC		With DC	
Circuit	area	rtime	area	rtime	area	rtime	area	rtime
t481	3637	944.3	3463	12581.9	61	11.8	65	22.7
frg2	1754	229.4	1525	2928.9	594	40.9	578	408.2
x1	1695	382.2	1560	2939.7	327	43.8	317	169.2
alu4	999	174.3	894	1302.2	836	130.8	785	1060.7
apex6	680	52.2	679	335.8	803	93.0	799	646.3
i6	581	62.3	579	400.0	318	24.8	318	213.7
C1908	596	40.7	588	7701.7	462	32.6	464	7322.0
x4	670	88.7	595	481.9	317	18.7	313	125.7
term1	598	81.7	450	651.3	219	34.0	217	138.6
frg1	583	144.1	506	705.4	227	46.6	226	182.3
alu2	570	102.6	431	328.1	470	75.7	425	264.2
ttt2	453	73.7	302	188.1	193	18.5	186	63.8
i5	356	23.5	356	67.4	198	7.3	198	12.0
c8	249	27.2	182	77.9	128	14.0	123	59.5
apex7	269	23.4	244	192.2	265	23.2	227	139.7
cht	231	26.6	200	83.7	127	7.9	127	22.8
9symml	214	37.6	215	100.3	216	43.5	216	116.5
z4ml	181	42.7	180	127.6	68	12.3	65	39.6
sct	144	20.3	142	47.0	86	9.6	83	61.4
lal	156	16.2	157	36.9	94	9.6	96	48.8
Total	14616	2593.7	13248	31278.0	6009	698.6	5828	11117.7
	1.0	1.0	0.91	12.1	0.41	0.27	0.39	4.3
					1.0	1.0	0.97	15.9

Table 6.3: Mapping results for area (Using *don't cares* , LSI Logic library)

Chapter 7

Performance-driven technology mapping

Previous chapters introduced methods to improve the quality of the matching operation. We also presented a covering algorithm, whose goal is to find a good circuit implementation given a set of matches. The covering method presented in Chapter 3 can base its cost evaluation on area or delay, if a fixed gate-delay model is chosen. For the more general cases where load-dependent delays are considered, we now introduce an ensemble of operations that deal specifically with the optimization of delays through the network, in conjunction with technology mapping.

We first define the delay model we are using to evaluate the quality of the solutions during optimization. We then explain why taking delay into account is different and more difficult than optimizing for area. We follow with a presentation of three operations, repartitioning, redecomposition and re-covering, which are integrated together within an iterative refinement procedure. We conclude the chapter with results demonstrating the area-delay trade-off.

7.1 Delay model

Most commercial libraries use a delay model which is a linearization of the load-dependent gate delay. We base our delay calculation on the same idea, and in the remainder of this chapter, we consider the following delay model for the evaluation of timing characteristics of the network. Let us consider a vertex $v_j, j = \{1, 2, \ldots\}$ of the network.

- δ_j is the intrinsic (unloaded) gate delay;
- C_j is the load capacitance at a gate output;
- $\delta_j + \alpha_j \cdot C_j$ is the total gate delay, where α_j is a parameter of the library element representing its drive capability.
- $a_j = (\delta_j + \alpha_j \cdot C_j + \max_{I_j} a_i)$ is the arrival time at the output of the gate corresponding to v_j, where a_i is the arrival time at a gate input, with gate $v_i \in PI$ = Fanin(v_j).

In our formulation of the problem, we are given the set of arrival times $\{a_i, i \in PI\}$ of the primary inputs PI, together with the set of required times $\{r_o, o \in PO\}$ of the primary outputs PO. For synchronous circuits with period T, we assume the input arrival times to be 0, and the required times at the outputs (*i.e.* register inputs) to be $T - t_{setup}$. We use the concept of slack [HSC82, De 87, De 89], where the slack s_j at a certain vertex v_j corresponds to the difference between the required time[1] at that vertex r_j and the arrival time a_j, *i.e.* ($s_j = r_j - a_j$). Therefore, time critical nets are those with negative slacks. Note that we do not consider false paths [DKM91, MSSB91] in our delay calculations. Therefore, the critical paths reported are an upper bound on the real critical delays through the circuits. Note also that in the current implementation, we do not distinguish between rise and fall delay, although the algorithms presented can be easily extended to deal with separate rise and fall times. We use the worst of the rise and fall delays, and therefore, use a conservative model.

[1]Note that required time at an internal vertex is calculated by back-propagating through the network the required times at the primary outputs.

7.2 The difficulty with timing-driven technology mapping

We already mentioned in Section 3.3 that dynamic programming techniques can be used to optimize timing as well as area. But there is an important difference between the two optimizations: evaluating the area cost of a particular vertex mapping involves only vertices already mapped (predecessors of the vertex), whereas evaluating the timing cost involves also the successors of the vertex being mapped. Successors are needed because the capacitive load on the output of a gate influences its delay. Since the dynamic programming technique implies the successors of a vertex being processed are not yet mapped, then the capacitive load on its output is not known.

Therefore specific methods to deal with delay have to be introduced. *Binning* has been proposed by Rudell [Rud89b], where each vertex is (possibly) mapped for all the possible capacitive loads on its output. We propose a different heuristic solution, involving iterative mapping of the network. The first mapping of the network includes only the optimization of area. Then, the portions of the network that do not meet the timing constraints are iteratively remapped. This method has the advantage that the entire environment of a vertex is known when it is remapped. In particular, the capacitive load the vertex is driving is known exactly.

It is important to remark that a solution under given timing constraints may not exist. Therefore our strategy is to perform a set of transformations that lead to a mapped network that either satisfies the constraints or that cannot be further improved by the transformations themselves.

In order to be efficient, iterative remapping has to be powerful enough to modify substantially the portions of the network that do not meet the timing constraints, *i.e.* the vertices with negative slack. To converge to a good solution in a finite number of steps, it must also be monotonic. We propose an ensemble of three techniques to achieve this goal:

- *repartitioning* modifies the original partition of multi-fanout vertices.
- *redecomposition* changes the two-input decomposition, taking into account delay information.

- *re-covering* applies covering using delay as the cost metric.

We now describe repartitioning and redecomposition in detail. We briefly review re-covering, and then explain how the three operations are integrated in an iterative operation.

7.3 Repartitioning

Repartitioning takes place after a first mapping has been done, using the traditional partitioning technique outlined in Section 3.2. Repartitioning targets multiple-fanout vertices that do not meet the timing constraints. The goal is to change partition block boundaries, by merging subject graphs, to have other (and possibly more) choices when matching (and redecomposing) the vertices along the critical paths. Merging multiple fanout subject graphs means the merged portions have to be duplicated for the other fanouts to achieve the original functionality.

Consider for example the subcircuit in Figure 7.1, where a gate corresponds to a multiple-fanout vertex v_j on the critical path. The original arrival time a_j at its output is: $a_j = \max_{I_j}(a_i) + \delta_j + \alpha_j \cdot C_j$, where a_i is the arrival time on the inputs of gate j, δ_j is the intrinsic delay of gate j, α_j is the fanout-dependent delay factor of gate j, and C_j is the fanout.

Assuming a_l is the latest arriving input, we can reexpress a_j as: $a_j = \delta_l + \alpha_l \cdot C_l + \delta_j + \alpha_j \cdot C_j$ Then if we assume one of the fanouts (say v_c) is on the critical path, it is possible to isolate the critical fanout v_c from the other fanouts of vertex v_j by making a copy $\widetilde{v_j}$ of v_j, and using v_c as the only fanout of $\widetilde{v_j}$. The operation keeps the other fanouts of v_j in their original position. This duplication of vertex v_j, with the new gate driving the critical path only, is shown in Figure 7.2: $a'_j = \delta_l + \alpha_l \cdot (C_l + C_{lj}) + \delta_j + \alpha_j \cdot C_{jk}$, and $a''_j = \delta_l + \alpha_l \cdot (C_l + C_{lj}) + \delta_j + \alpha_j \cdot (C_j - C_{jk})$, where:

a'_j is the new arrival time for the critical path,

a''_j is the new arrival time for the other fanouts of j,

C_{lj} is the input capacitance of j at input l,

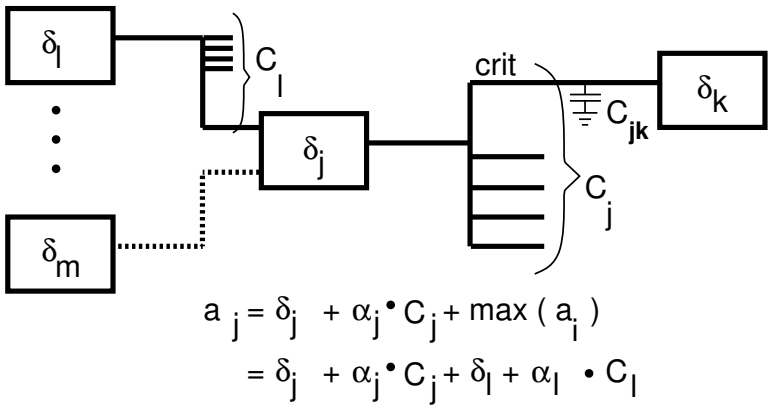


Figure 7.1: Delays in a subcircuit

C_{jk} is the input capacitance of gate k, which is the gate corresponding to the fanout of j on the critical path.

Then, the difference in delay is $\Delta a'_j = a'_j - a_j = \alpha_l \cdot C_{lj} + \alpha_j \cdot (C_{jk} - C_j)$, and $\Delta a''_j = a''_j - a_j = \alpha_l \cdot C_{lj} - \alpha_j \cdot C_{jk}$. The arrival time of the fanin-vertices of v_j are also modified by the duplication process. The difference in delay is $\Delta a'_l = a'_l - a_l = \alpha_l \cdot C_{lj}$.

This example shows some important properties for vertices with multiple fanouts:

- Duplicating gates *per se* reduces delay along the critical paths, when $(\alpha_{i0} \cdot C_{ij} - \alpha_j \cdot C_{jk} \leq 0)$.

  This is usually the case, and it can be verified on a case by case basis.

- The fanins of the duplicated vertex are slowed down by the addition of one gate load $(\alpha_{ik} \cdot C_{ij})$.

For a particular vertex which does not meet the timing constraints, it is therefore simple to verify how much can be gained by duplication, and whether or not the duplication affects other critical nets. In particular, if all the inputs of the vertex to duplicate have a single fanout, then duplication is always a good solution. In addition, the duplicated vertex can now be merged forward into the next partition (it is now a single fanout vertex). Mapping can be redone at this point on the modified partition, possibly improving

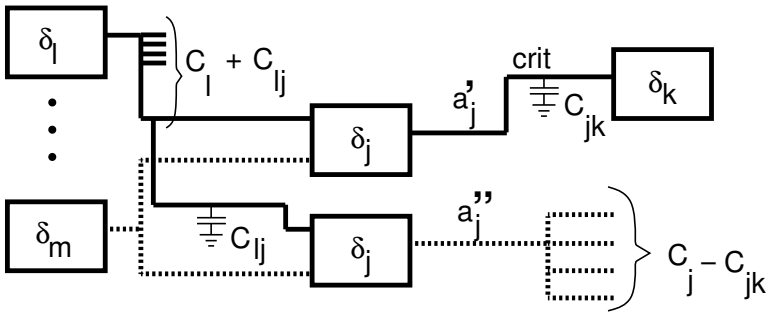


Figure 7.2: Delays in a subcircuit after gate duplication

delay even more.

7.4 Redecomposition

Redecomposition is used alone or in combination with repartitioning. The goal is to restructure the Boolean network in such a way that late arriving signals of a partition are used as inputs of vertices that are closer to the output of the partition. Redecomposition has (like decomposition) two important side effects:

- It influences the list of library elements that may cover a subject graph.
- It influences the critical path through the Boolean network.

The first point is related to the fact that different decompositions might give rise to different *possible* covers. For example, given $f = a + b\overline{c} + \overline{b}c$, the following decompositions imply very different covers:

$$\begin{array}{lcllcl} f_1 & = & a + x & f_2 & = & x + y \\ x & = & y + z & x & = & b\overline{c} \\ y & = & b\overline{c} & y & = & a + z \\ z & = & \overline{b}c & z & = & \overline{b}c \end{array}$$

In particular, the decomposition f_1 allows the Exclusive-OR $x = b\overline{c} + \overline{b}c$ to be mapped, whereas in decomposition f_2, the Exclusive-OR gate cannot be found (because variable a appears as an input to the same gate as $z = \overline{b}c$). We address the first point by heuristically trying to keep repeated literals together during the decomposition. The second point is important because decomposition can be used to push late arriving signals closer or further from the output, *possibly* reducing or lengthening the critical path. This problem has been addressed by Singh [SWBSV88] and Paulin [PP89].

Redecomposition implies unmapping a portion of mapped network, changing its *base function* decomposition, and then remapping the modified block. It is a tentative process, in that mapping the redecomposed partition does not necessarily give better results. We therefore isolate subgraphs being redecomposed, and use the new decomposition only when it produces better results. The evaluation of the value of a redecomposed (and remapped) partition is fairly simple and it involves two steps. First, since the subcircuit under consideration has a single output, we can just compare the arrival times of the original and re-decomposed partition blocks. Second, we check if the input loads have increased, and, if so, if any other critical net was created.

The redecomposition algorithm we are using follows the same principle that Singh proposed [SWBSV88]. One significant difference is that we use BDDs instead of kernel extraction for the decomposition. After a subgraph Γ is isolated for redecomposition, its inputs are ordered in decreasing order of their arrival times. That input list then specifies the order in which the variables are processed during the BDD extraction. After the BDD is reduced, it is then retransformed into a standard Boolean network, which is finally mapped. Procedure *redecompose* transforms a Boolean network into one where the latest arriving inputs are closer to the output (Fig. 7.3).

```
redecomp(eq) {
        eq_list = get_input_list(eq)                          /* Get support of equation */
        order_input_list(eq_list)                             /* Order support by arrival times (latest first) */
        bdd = create_reduced_bdd(eq,eq_list)                  /* Create BDD with previous order */
        get_network_from_bdd(bdd) }                           /* Transform BDD into Boolean Factored form */


get_network_from_bdd(bdd) {
        if (low(bdd) == ZERO) {
                if (high(bdd) == ONE)                         /* f = x */
                        return(create_equation(LITERAL,control_var(bdd))
                        else {                                /* f = x h */
                        eq = create_product_of(control_var(bdd),
                                get_network_from_bdd(high(bdd))
                        return(eq) } }
                        else if (low(bdd) == ONE) {
        if (high(bdd) == ZERO)                                /* f = x' */
                eq = create_complement(control_var(bdd))
                        return(eq)
                        else {                                /* f = x' + h */
                        eq = create_sum_of(complement(control_var(bdd)),
                                get_network_from_bdd(low(bdd))
                        return(eq) } }
                        else {
if (high(bdd) == ZERO) {                                      /* f = x' l */
                eq = create_product_of(complement(control_var(bdd)),
                        get_network_from_bdd(high(bdd))
                return(eq) }
                else if (high(bdd) == ONE) {                  /* f = x + l */
                eq = create_sum_of(control_var(bdd),
                        get_network_from_bdd(low(bdd))
                return(eq) } }
                else {                                        /* f = x' l + x h */
        s1 = create_product_of(complement(control_var(bdd)),
                get_network_from_bdd(low(bdd))
        s2 = create_product_of(control_var(bdd),
                get_network_from_bdd(high(bdd)))
        eq = create_sum_of(s1,s2)
        return(eq) } }
```

Figure 7.3: Algorithm BDD to Boolean network conversion

For example, let us reconsider the Boolean network described in Section 3.3:

$$\begin{aligned} f &= j + t \\ j &= xy \\ x &= e + z \\ y &= a + c \\ z &= \overline{c} + d \end{aligned}$$

Assume vertex v_j does not meet its timing requirement, and that the arrival times of the inputs to the partition block rooted by v_j are: $\{a_a = 10.0, a_c = 12.0, a_d = 5.0, a_e = 7.0\}$. The variable ordering used for creating the BDD representing j would be $\{c, a, e, d\}$. The resulting BDD is shown in Figure 7.4, together with the two-input gate decomposition derived from the BDD.

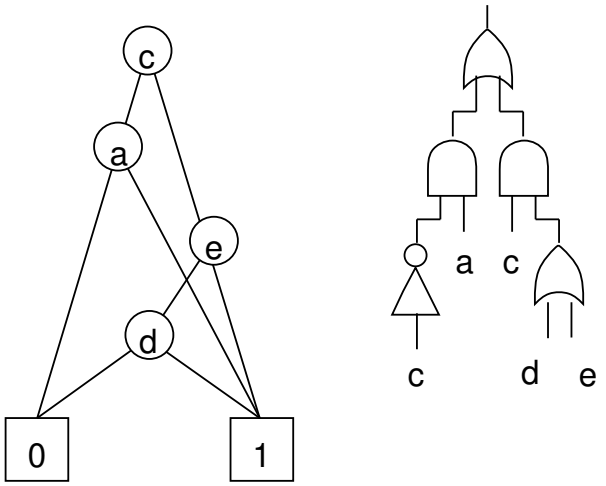


Figure 7.4: BDD and corresponding Boolean network

7.5 Re-covering

After each modification of the network (repartitioning or redecomposition), we apply timing-driven covering. The goal is to find better implementations that take advantage of

the modified network structure. The operation is carried out as follows. All time-critical gates of the modified network are isolated. We then extract the corresponding logic of these time-critical gates, and represent them with unmapped, combinational logic. At this point, the network represents a hybrid of mapped logic (netlist of library elements) and unmapped combinational logic. We then apply the matching and covering techniques described in the previous chapters.

Since the goal of this re-covering step is to optimize the circuit for delay, we use timing as a cost metric. We explained previously that fanout-dependent delays are difficult to evaluate during the covering step, since in general the load on the output of a vertex is not known when it is being matched. During re-covering, some loads are known: those corresponding to gates that are still mapped. For the loads that are not known, we use an estimation of the load which is the number of fanout edges times the average input load of the gates in the library. Therefore, during the re-covering process, the cost evaluation is an approximation of the actual delay.

7.6 Iterative mapping

The techniques outlined above, repartitioning and redecomposition, are integrated in an iterative operation. After a first area-oriented mapping, arrival times and required times are computed for each gate in the network. The required times on the outputs are assumed to be given, and so are the arrival times on the inputs. The difference between arrival time and required time, or slack, is computed for each gate. The gates that have negative slacks are then operated upon in reverse topological order, where primary output gates appear first, and primary input gates appear last.

Redecomposition and repartitioning are used iteratively until the constraints are satisfied (*i.e.* no negative slack) or no more improvement is possible. After each of the two operations, re-covering is applied to take advantage of the modified structure of the network. Since each step is accepted only if it speeds up the target gate without affecting negatively the slacks on surrounding gates, this process is guaranteed to complete in a finite number of steps. Figure 7.5 shows the pseudocode for the iterative operation.

```
iterative_map(network) {
    network_changed = TRUE                          /* Iteration proceeds until no improvement is found */
    while (network_changed) {
        network_changed = FALSE
        compute_timing(network)
                                                    /* Compute slacks for each vertex */
        current_best = duplicate(network)           /* Keep a copy of current network */
        (∀ critical vertices ∈ network) {
            repartition(vertex) }
        unmap_critical_vertices(network)            /* Get combinational logic of all critical vertices */
        re-cover_for_delay(network)                 /* Map all vertices that were just unmapped */
        if (cost(network) < cost(current_best) {
            network_changed = TRUE
            current_best = duplicate(network) }     /* Update bets solution */
        (∀ critical vertices ∈ network) {
            redecompose(vertex) }
        unmap_critical_vertices(network)            /* Get combinational logic of all critical vertices */
        re-cover_for_delay(network)                 /* Map all vertices that were just unmapped */
        if (cost(network) < cost(current_best) {
            network_changed = TRUE
            current_best = duplicate(network) }     /* Update bets solution */
        } }
```

Figure 7.5: Pseudocode for iterative delay optimization

7.7 Buffering/repowering

In addition to the operations outlined in this chapter, buffering and repowering can be used as a last resort to improve timing. Repowering should be used first, using gates with more drive capability for vertices with high fanout that are on critical paths. After repowering, buffering can be used to speed up nets with large fanouts, when neither redecomposition nor repartitioning can be applied, or they would modify other critical nets.

Buffering was studied by Berman *et al.* [BCD89], who proved that the problem is NP-complete even in its simplest form. In their formulation, buffer trees are used as a mean to reduce the required times at gates which drive large capacitive loads. They proposed a polynomial time algorithm for a simplified problem, the *adjacent output* buffering problem, where gates are ranked by required times, and adjacent gates in that rank are adjacent in the buffer tree. Unfortunately, the complexity of the method (N^5), although polynomial, makes it impractical for large number of fanouts.

Touati proposed a simplified buffer tree structure, the LT-tree, noting that the depth of buffer trees is usually limited, and that fully general solutions are seldom necessary [TMBW90, Tou90]. Singh *et al.* proposed a heuristic approach to buffering, based on a divide-and-conquer formulation [SSV90]. Recently, Lin *et al.* proposed a heuristic solution which tries to minimize the area of buffer trees under timing constraints [LMS91].

Buffering is currently done as a postprocess after all the other modifications fail. We rely on external systems to carry that operation, for example Touati's buffering strategy in Berkeley's *sis*. Note that the results presented in the last section of this chapter do not include any buffering.

7.8 Results

Figure 7.6 shows the area/delay trade-off curve for different circuits. Note that all the points on these curves are obtained as successive results during the delay optimization iterations. It is possible for the user to chose between any of these implementation during a single run of the program. When area as well as delay are constrained, the program stops the delay optimization iterations to limit the area increase. Tables 7.1 and 7.2 show the results for the fastest (and usually largest) implementations. Results from program *Ceres* are compared to those of *Sis*, where both systems where used in delay optimization mode.

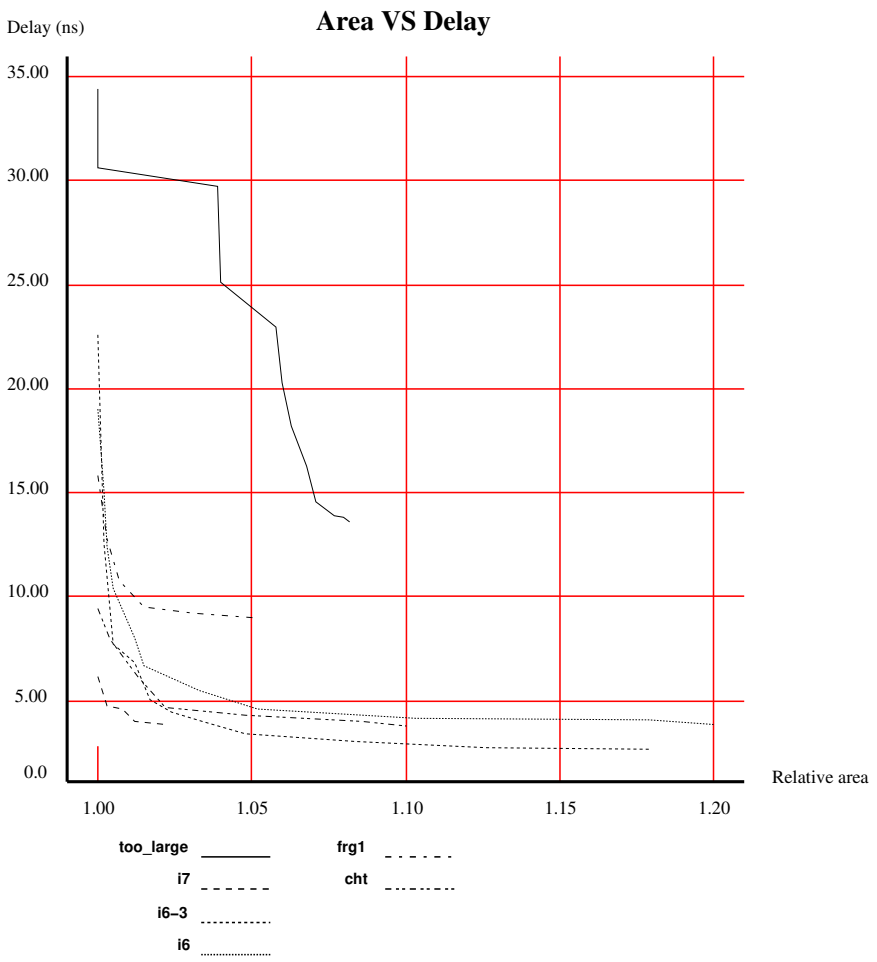


Figure 7.6: Example of area/delay tradeoffs

	Sis			**Ceres**				
				Smallest		Fastest		
Circuit	delay	area	rtime	delay	area	delay	area	rtime
C6288	1063.8	2414	147.0	891.9	1425	835.7	2028	608.7
C7552	346.0	2349.0	214.3	172.7	1062	148.8	1567	433.1
C5315	203.8	1775	179.2	197.8	831	186.3	960	230.5
frg2	164.4	1243	146.5	272.2	844	110.9	918	370.2
pair	169.4	1518	138.8	150.9	716	130.2	1074	271.8
x1	70.8	308	54.1	102.3	807	93.0	851	349.2
C3540	287.4	1066	120.6	290.8	608	265.8	682	155.6
vda	139.0	810	162.7	95.2	543	79.4	669	127.4
x3	106.2	770	103.6	117.2	527	87.0	668	140.0
rot	180.6	617	64.4	318.5	551	192.9	1133	398.1
alu4	250.0	685	71.4	312.0	550	258.9	1183	188.0
C2670	229.2	697	89.9	215.2	317	179.8	342	83.6
apex6	90.0	818	66.0	104.6	399	83.7	583	47.7
C1355	208.2	695	51.2	126.4	176	116.4	253	40.4
term1	96.0	356	58.9	128.1	302	124.8	313	85.8
x4	95.8	444	61.5	101.4	368	65.1	397	104.1
alu2	220.8	339	47.3	287.2	320	199.8	1214	208.5
frg1	77.4	104	28.7	79.4	271	74.4	299	133.4
C1908	191.4	529	66.9	195.6	263	170.6	457	50.8
ttt2	64.4	197	40.7	115.2	249	83.7	306	85.2
C880	139.8	330	46.3	171.1	178	145.1	318	50.9
C499	123.0	327	45.6	128.3	168	114.3	233	27.2
example2	90.6	296	38.8	109.9	175	65.1	251	38.7
apex7	96.8	237	35.8	117.8	142	107.3	177	39.3
my_adder	188.2	160	29.8	157.8	64	148.8	129	29.6
C432	195.0	223	35.7	200.6	93	182.1	168	56.2
f51m	192.0	115	27.5	56.5	129	56.5	129	47.9
c8	55.0	168	33.2	49.4	116	46.5	173	66.4
i10	399.0	2445	247.4	518.1	1232	305.2	2002	348.9
dalu	341.8	1776	161.0	213.0	909	169.2	1224	344.7
count	165.2	144	28.1	166.5	63	83.7	119	12.7
comp	110.2	150	28.2	111.4	60	95.7	250	46.6
i4	49.2	198	43.5	57.8	122	57.8	122	36.1
cht	56.8	188	32.4	32.9	124	32.9	124	22.5
cc	39.4	58	24.6	27.9	31	27.9	31	4.6
Total	7067.0	25735	3000.9	6614.4	15603	5228.3	22394	5522.9
	1.0	1.0	1.0	0.94	0.61	0.74	0.87	1.8

Table 7.1: Mapping results for delay (No *don't cares* , Actel library)

	Sis			**Ceres**				
				Smallest		Fastest		
Circuit	delay	area	rtime	delay	area	delay	area	rtime
C6288	84.2	5963	92.0	106.9	2263	106.9	2263	206.9
C7552	43.0	5210	133.0	34.1	2611	25.5	3072	438.8
C5315	22.7	4113	107.1	30.7	1981	28.6	2208	169.5
x1	7.9	764	29.0	11.4	1695	9.0	1709	462.8
C3540	32.9	2423	69.7	42.1	1214	32.3	2951	462.5
vda	14.3	2260	141.6	19.9	1078	10.7	1432	214.7
x3	11.6	2013	56.7	15.8	1125	8.5	1423	211.9
rot	18.6	1345	39.8	37.4	1117	20.1	1866	341.6
alu4	29.7	1434	40.5	45.5	993	27.8	2399	332.8
C2670	23.8	1705	54.7	32.9	864	22.6	1225	90.0
apex6	9.7	1339	38.3	12.7	674	10.8	721	62.8
C1355	19.6	1373	32.0	15.9	404	15.9	404	21.8
term1	11.0	861	32.0	15.0	598	12.7	745	114.4
x4	10.1	1014	34.2	14.0	670	6.7	765	142.2
alu2	27.5	747	26.4	36.8	568	22.1	1261	250.8
frg1	9.3	223	15.9	15.9	583	9.0	613	237.8
C1908	20.8	1234	39.4	29.3	596	23.9	1348	106.4
ttt2	7.4	528	22.4	15.0	452	8.2	572	113.3
C880	17.1	737	26.0	22.3	309	12.7	1444	124.0
C499	13.6	935	26.8	17.1	406	16.4	884	93.6
apex7	9.6	485	20.1	13.5	270	8.1	516	74.3
my_adder	21.0	348	16.3	39.1	256	17.8	682	97.2
C432	20.1	524	19.8	28.3	202	25.6	256	36.3
f51m	7.6	257	15.4	7.3	244	6.5	248	62.9
c8	6.05	321	18.3	7.3	249	6.1	338	45.6
i10	40.6	5529	143.6	81.3	2638	58.4	2731	283.0
dalu	36.4	3855	94.7	31.9	2090	23.8	3015	535.8
count	18.0	243	15.4	28.5	112	7.6	279	43.2
comp	11.8	251	15.5	11.6	151	11.3	219	19.3
i4	7.2	500	24.8	5.7	208	5.0	300	22.9
cht	6.4	507	18.4	9.5	231	3.8	254	55.0
cc	4.6	163	13.3	5.5	74	3.4	87	10.6
Total	662.3	52420	1580.1	869.0	28268	614.4	39693	5723.9
	1.0	1.0	1.0	1.3	0.54	0.93	0.76	3.6

Table 7.2: Mapping results for delay (No *don't cares* , LSI Logic library)

Chapter 8

Conclusion and future directions

Technology mapping is an important operation in automatic synthesis of digital circuits. The usefulness of automatic tools hinges ultimately on the quality of their results. Technology mapping is the operation that bridges the gap between abstract logic descriptions and electronic gate implementations. Therefore, the quality of the technology mapping step is essential in obtaining a good implementation.

There are two difficult problems intrinsic to technology mapping: covering and matching. In this dissertation a general covering problem was described, and we showed that every technology mapping system, including this one, solves a simpler, restricted covering problem.

We have presented new algorithms for matching, which proved to be more powerful than other existing methods. In particular, we introduced the idea of using Boolean techniques for establishing logic equivalence between two logic functions, which is key to the solution of the matching problem. In a first set of algorithms, we presented a technique for efficiently dealing with completely specified functions, using Boolean operations. These algorithms were based on the use of Binary Decision Diagrams and exploit invariant logic properties, unateness and logic symmetry.

We also introduced an extension of the previous method, which allowed the detection of logic equivalence between incompletely specified functions. This second method, based on the *matching compatibility graph*, extended the capabilities of Boolean matching,

making possible the use of *don't care* information. In general, using *don't cares* increases the number of matches, which usually imply better quality solutions, as shown by the experimental results.

We finally presented an iterative framework for performance-driven technology mapping. We showed how we can iteratively use local transformations, changing the structure of the circuit by duplication and decomposition, and obtaining faster implementations. Results showed that, as the iterations proceeded, it was possible to trade-off area for delay.

The work presented in this dissertation is based on many assumptions and simplifications, which had to be made for efficient algorithms to be found. The simplifications included the restricted covering problem, partitioning at multi-fanout vertices, and matching single-output logic functions. Furthermore, we assumed circuits to be synchronous. We also adopted a delay model based on the worst-case delay through combinational circuits. As new techniques emerge for logic synthesis, some of these simplifications might become unnecessary, and their elimination will open up new possibilities for technology mapping.

Given the current knowledge on technology mapping and logic synthesis, some new developments are possible. Among these, there are two avenues which show promises for the near future. The first one regards asynchronous digital circuits. The use of asynchronous digital circuitry has recently proved to be area-competitive, with improved performance, compared to traditional, synchronous digital design [WH91, Wil90, MBL$^+$89]. Research on automatic synthesis of asynchronous circuits has progressed to the point where systems for technology-independent description and optimization are now possible [THYMM89, ND91]. One problem is in the translation to technology-dependent implementations, which currently needs to be done by hand. A natural extension to actual technology mapping systems would be handling that transformation automatically. However, this application of technology mapping to the asynchronous world is not simple, since logic gates in the final implementation have to be carefully chosen to prevent logic hazards from occurring.

The second promising direction for technology mapping systems is to incorporate false paths information during performance optimization. False paths occur in a combinational

network when the structure of the network makes impossible the sensitization of certain paths through the network. In that case, the evaluation of the critical delays through the circuit has to take into account the existence of false paths in order to obtain the true critical delay [DKM91, MSSB91]. False paths have recently begun to be considered during digital circuit synthesis [Ber91a, MBSVS91]. One pending difficulty is the complexity of identifying which paths are the true critical paths in a given implementation [Wan91]. Given an efficient algorithm for false paths identification, the quality of performance-driven technology mapping could be improved by limiting iterative operations to true critical paths.

Technology mapping represents one step in the automatic synthesis of digital circuits. As the design process changes, the requirements on synthesis systems will evolve. New technologies will require different constraints on technology-specific optimizations. New design paradigms will imply modifying synthesis systems to allow different user-interaction models. Looking at the history of digital circuits tools from the first layout editors to the current use of hardware description languages, one can only guess the possibilities for the next generation of synthesis systems.

Bibliography

[AB78] Z. Arevalo and J. G. Bredeson. A Method to Simplify a Boolean Function in a Near Minimal Sum-of-products for Programmable Logic Arrays. *IEEE Transactions on Computers*, 27:1028, November 1978.

[AG85] A. V. Aho and M. Ganapathi. Efficient Tree Pattern Matching: an Aid to Code Generation. In *Symposium on Principles of Programming Languages*, pages 334–340. ACM, January 1985.

[AGT89] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[AJ76] A. V. Aho and S. C. Johnson. Optimal Code Generation for Expression Trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[AJU77] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code Generation for Expressions with Common Subexpressions. *Journal of the ACM*, 24(1):146–160, January 1977.

[Ake78] S. B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, 27(6):509–516, June 1978.

[And64] J. P. Anderson. A Note on Some Compiling Algorithms. *Communications of the ACM*, 7(3):149–150, March 1964.

[Ash59] R. Ashenhurst. The Decomposition of Switching Functions. In *International Symposium on the Theory of Switching*, pages 74–116. MIT, 1959.

Held 2-5 April 1957.

[Bak91] S. Baker. Quicklogic unveils FPGA. *EE Times*, 639, April 1991.

[BBH+88] K. A. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. Multilevel Logic Minimization Using Implicit Don't Cares. *IEEE Transactions on CAD/ICAS*, 7(6):723–740, June 1988.

[BBL89] D. Bryan, F. Brglez, and R. Lisanke. Redundancy Identification and Removal. In *International Workshop on Logic Synthesis*, page 8.3. MCNC, May 1989.

[BCD+88] R. K. Brayton, R. Camposano, G. De Micheli, R. H. J. M. Otten, and J. van Eijndhoven. The Yorktown Silicon Compiler System. In D. Gajski, editor, *Silicon Compilation*. Addison Wesley, 1988.

[BCD89] C. L. Berman, J. L. Carter, and K. F. Day. The Fanout Problem: From Theory to Parctice. In C. L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the 1989 Decennial Caltech Conference*, pages 69–99. MIT Press, March 1989.

[BD62] R. E. Bellman and S. E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1962.

[BDK+86] R. Brayton, E. Detjens, S. Krishna, T. Ma, P. McGeer, L. Pei, N. Phillips, R. Rudell, R. Segal, A. Wang, R. Yung, and A. Sangiovanni-Vincentelli. Multiple-Level Logic Optimization System. In *International Conference on Computer-Aided Design*, pages 356–359. IEEE, November 1986.

[BDS91] D. Bochman, F. Dreisig, and B. Steinbach. A New Decomposition Method for Multilevel Circuit Design. In *European Design Automation Conference*, pages 374–377. IEEE, February 1991.

[Bel57] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Ber88] R. A. Bergamaschi. Automatic Synthesis and Technology Mapping of Combinational Logic. In *International Conference on Computer-Aided Design*, pages 466–469. IEEE, November 1988.

[Ber91a] R. A. Bergamaschi. The Effects of False Paths in High-Level Synthesis. In *International Conference on Computer-Aided Design*, pages 80–83. IEEE, November 1991.

[Ber91b] C. L. Berman. Circuit Width, Register Allocation, and Ordered Binary Decision Diagrams. *IEEE Transactions on CAD/ICAS*, 10(8):1059–1066, August 1991.

[BG87] F. Brewer and D. Gajski. Knowledge Based Control in Micro-architectural Design. In *Design Automation Conference*, pages 203–209. IEEE, June 1987.

[BHJ+87] D. Bostick, G. D. Hachtel, R. Jacoby, M. R. Lightner, P. Moceyunas, C. R. Morrison, and D. Ravenscroft. The Boulder Optimal Logic Design System. In *International Conference on Computer-Aided Design*, pages 62–69. IEEE, November 1987.

[BHMSV84] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic publishers, 1984.

[BM82] R. K. Brayton and C. T. McMullen. The Decomposition and Factorization of Boolean Expressions. In *International Symposium on Circuits and Systems*, pages 49–54. IEEE, 1982.

[BM84] R. K. Brayton and C. T. McMullen. Synthesis and Optimization of Multistage Logic. In *International Conference on Computer Design*, pages 23–28. IEEE, October 1984.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *Design Automation Conference*, pages 40–45. IEEE, June 1990.

[Bre77] M. Breuer. Min-cut Placement. *Journal of Design Automation of Fault Tolerant Computers*, pages 343–362, October 1977.

[Bro81] D. W. Brown. A State-Machine Synthesizer - SMS. In *Design Automation Conference*, pages 301–304. IEEE, June 1981.

[BRSVW87] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on CAD/ICAS*, 6(6):1062–1081, November 1987.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[BS76] J. L. Bruno and R. Sethi. Code Generation for a One-register Machine. *Journal of the ACM*, 23(3):502–510, July 1976.

[Bur86] M. Burstein. Channel Routing. In T. Ohtsuki, editor, *Layout Design and Verification*, pages 133–167. North-Holland, 1986.

[CBL91] T. J. Chakraborty, S. Bhawmik, and C. J. Lin. Enhanced Controllability For IDDQ Test Sets Using Partial Scan. In *Design Automation Conference*, pages 278–281. ACM/IEEE, June 1991.

[CBM90a] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Using Boolean Functional Vectors. In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification: VLSI Design Methods-II, Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Houthalen, 13-16 Nov. 1989*, pages 179–196. North-Holland, 1990.

[CBM90b] O. Coudert, C. Berthet, and J.C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In J. Sifakis, editor, *Lecture Notes in Computer Science No 407: International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, 12-14 Jun. 1989*, pages 365–373. Springer-Verlag, 1990.

[CK56] S. R. Cray and R. N. Kisch. A Progress Report on Computer Applications in Computer Design. In *Western Joint Computer Conference*, pages 82–85, 1956.

[CM89] K. C. Chen and S. Muroga. SYLON-DREAM: A Multi-Level Network Synthesizer. In *International Conference on Computer-Aided Design*, pages 552–555. IEEE, November 1989.

[Coo71] S. A. Cook. The Complexity of Theorem-Proving Procedures. In *3rd Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[CR89] R. Camposano and W. Rosenstiel. Synthesizing Circuits from Behavioral Descriptions. *IEEE Transactions on CAD/ICAS*, 8(2):171–180, February 1989.

[Cur61] H. A. Curtis. A Generalized Tree Circuit. *Journal of the ACM*, 8:484–496, 1961.

[Cur62] H. A. Curtis. *A New Approach to the Design of Switching Circuits*. Van Nostrand, 1962.

[DAR86] M. R. Dagenais, V. K. Agarwal, and N. C. Rumin. McBoole: A New Procedure for Exact Logic Minimization. *IEEE Transactions on CAD/ICAS*, 5(1):229–237, January 1986.

[DBJT84] J. Darringer, D. Brand, W. Joyner, and L. Trevillyan. LSS: A System for Production Logic Synthesis. *IBM J. Res. Develop.*, September 1984.

[De 87] G. De Micheli. Performance-Oriented Synthesis in the Yorktown Silicon Compiler. *IEEE Transactions on CAD/ICAS*, 6(5):751–765, September 1987.

[De 89] G. De Micheli. Synchronous Logic Synthesis. In *International Workshop on Logic Synthesis*, page 5.2, North Carolina, May 1989.

[DGR+87] E. Detjens, G. Gannot, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. Technology Mapping in MIS. In *International Conference on Computer-Aided Design*, pages 116–119. IEEE, November 1987.

[DHNSV85] G. De Micheli, M. Hoffman, R. Newton, and A. Sangiovanni-Vincentelli. A System for the Automatic Synthesis of Programmable Logic Arrays. In A. Sangiovanni-Vincentelli, editor, *Computer-Aided Design of VLSI Circuits and Systems*. Jai Press, 1985.

[DJBT81] J. Darringer, W. Joyner, C. L. Berman, and L. Trevillyan. Logic Synthesis Through Local Transformations. *IBM J. Res. Develop.*, 25(4):272–280, July 1981.

[DKM91] S. Devadas, K. Keutzer, and S. Malik. Delay Computation in Combinational Logic Circuits: Theory and Algorithms. In *International Conference on Computer-Aided Design*, pages 176–179. IEEE, November 1991.

[DKMT90] G. De Micheli, D. Ku, F. Mailhot, and T. Truong. The Olympus Synthesis System. *IEEE Design & Test of Computers*, pages 37–53, October 1990.

[DM90] M. Damiani and G. De Micheli. Observability Don't Care Sets and Boolean Relations. In *International Conference on Computer-Aided Design*, pages 502–505. IEEE, November 1990.

[DM91] M. Damiani and G. De Micheli. Derivation of Don't Care Conditions by Perturbation Analysis of Combinational Multiple-Level Logic Circuits. In *International Workshop on Logic Synthesis*, page 6.1a. MCNC, May 1991.

[DSV83] G. De Micheli and A. Sangiovanni-Vincentelli. Multiple Constrained Folding of Programmable Logic Arrays: Theory and Applications. *IEEE Transactions on CAD/ICAS*, 2(3):167–180, July 1983.

[Fil91] D. Filo, March 1991. Private communication.

[Flo61] R. W. Floyd. An Algorithm for Coding Efficient Arithmetic Operations. *Communications of the ACM*, 4(1):42–51, January 1961.

[FSTH90] R. R. Fritzemeir, J. Soden, R. K. Treece, and C. Hawkins. Increased CMOS IC Stuck-at Fault Coverage with Reduced IDDQ Test Sets. In *International Test Conference*, pages 427–435. IEEE, 1990.

[GB87] B. Gurunath and N. N. Biswas. An Algorithm for Multiple Output Minimization. In *International Conference on Computer-Aided Design*, pages 74–77. IEEE, November 1987.

[GBdGH86] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel. SOCRATES: A System for Automatically Synthesizing and Optimizing Combinational Logic. In *23rd Design Automation Conference*, pages 79–85. IEEE/ACM, 1986.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GLWH84] J. L. Gilkinson, S. D. Lewis, B. B. Winter, and A. Hekmatpour. Automated Technology Mapping. *IBM J. Res. Develop.*, 28(5):546–555, September 1984.

[HCO74] S. J. Hong, R. G. Cain, and D. L. Ostapko. MINI: A heuristic apporoach for logic minimization. *IBM J. Res. Develop.*, 18:443–458, September 1974.

[HHLH91] C. Y. Hwang, Y. C. Hsieh, Y. L. Lin, and Y. C. Hsu. An Efficient Layout Style for 2-Metal CMOS Leaf Cells and Their Automatic Generation. In *Design Automation Conference*, pages 481–486. IEEE, June 1991.

[HMM85] S. L. Hurst, D. M. Miller, and J. C. Muzio. *Spectral Techniques in Digital Logic*. Academic Press, 1985.

[HO82] C. M. Hoffmann and M. J. O'Donnell. Pattern Matching in Tress. *Journal of the ACM*, 29(1):68–95, January 1982.

[Hol61] P. A. Holst. Bibliography on Switching Theory Circuits and Logical Algebra. *IRE Transactions on Electronic Computers*, pages 638–661, December 1961.

[HS71] A. Hashimoto and J. Stevens. Wire Routing by Optimizing Channel Assignment within Large Apertures. In *Design Automation Conference*, pages 155–169. IEEE, 1971.

[HSC82] R. Hitchcock, G. Smith, and D. Cheng. Timing Analysis of Computer Hardware. *IBM J. Res. Develop.*, 26(1):100–105, January 1982.

[HSM82] W. Heller, G. Sorkin, and K. Maling. The Planar Package for System Designers. In *Design Automation Conference*, pages 253–260. IEEE, June 1982.

[ISH+88] J. Ishikawa, H. Sato, M. Hiramine, K. Ishida, S. Oguri, Y. Kazuma, and S. Murai. A Rule Based Logic Reorganization System LORES/EX. In *International Conference on Computer Design*, pages 262–266. IEEE, October 1988.

[Joh83] S. C. Johnson. Code Generation for Silicon. In *Symposium on Principles of Programming Languages*, pages 14–19. ACM, January 1983.

[JTB+86] W. H. Joyner, L. H. Trevillyan, D. Brand, T. A. Nix, and A. C. Gundersen. Technology Adaptation in Logic Synthesis. In *23rd Design Automation Conference*, pages 94–100. IEEE/ACM, June 1986.

[Kah86] M. Kahrs. Matching a Parts Library in a Silicon Compiler. In *International Conference on Computer-Aided Design*, pages 169–172. IEEE, November 1986.

[Keu87] K. Keutzer. DAGON: Technology Binding and Local Optimization by DAG Matching. In *24th Design Automation Conference*, pages 341–347. IEEE/ACM, June 1987.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vechi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[KL70] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell Systems Technical Journal*, 49(2):291–307, February 1970.

[KMS86] E. S. Kuh and M. Marek-Sadowska. Global Routing. In T. Ohtsuki, editor, *Layout Design and Verification*, pages 169–198. North-Holland, 1986.

[Ku91] D. Ku. *Constrained Synthesis and Optimization of Digital Integrated Circuits from Behavioral Specifications*. PhD thesis, Stanford University, June 1991. Technical report CSL-TR-91-476.

[Lau80] U. Lauther. A Min-cut Placement Algorithm for General Cell Assemblies Based on a Graph Representation. *Journal of Digital Systems*, 4(1):21–34, 1980.

[LBK88] R. Lisanke, F. Brglez, and G. Kedem. McMAP: A Fast Technology Mapping Procedure for Multi-Level Logic Synthesis. In *International Conference on Computer Design*, pages 252–256. IEEE, October 1988.

[Lee61] C. Y. Lee. An Algorithm for Path Corrections and its Applications. *IRE Transactions on Electronic Computers*, pages 346–365, September 1961.

[Liu77a] C. L. Liu. *Elements of Discrete Mathematics*. McGraw-Hill, 1977.

[Liu77b] T. K. Liu. Synthesis of Feed-Forward MOS Networks with Cells of Similar Complexities. *IEEE Transactions on Computers*, 26(8):826–831, August 1977.

[Liu77c] T. K. Liu. Synthesis of Multilevel Feed-Forward MOS Networks. *IEEE Transactions on Computers*, 26(6):581–588, June 1977.

[LMS91] S. Lin and M. Marek-Sadowska. A Fast and Efficient Algorithm for Determining Fanout Trees in Large Networks. In *European Design Automation Conference*, pages 539–544. IEEE, February 1991.

[LS90] B. Lin and F. Somenzi. Minimization of Symbolic Realtions. In *International Conference on Computer-Aided Design*, pages 88–91. IEEE, November 1990.

[MB88] H. J. Mathony and U. G. Baitinger. CARLOS: An Automated Multilevel Logic Design System for CMOS Semi-Custom Integrated Circuits. *IEEE Transactions on CAD/ICAS*, 7(3):346–355, March 1988.

[MB89] P. McGeer and R. K. Brayton. Consistency and Observability Invariance in Multi-Level Logic Synthesis. In *International Conference on Computer-Aided Design*, pages 426–429. IEEE, November 1989.

[MBL+89] A. J. Martin, S. M. Burns, T.K. Lee, D. Borković, and P.J. Hazewindus. The design of an asynchronous microprocessor. In *Decennial Caltech Conference on VLSI*, pages 226–234, 1989.

[MBSVS91] P. C. McGeer, R. K. Brayton, A. L. Sangiovanni-Vincentelli, and S. K. Sahni. Performance Enhancement through the Generalized Bypass Transform. In *International Conference on Computer-Aided Design*, pages 184–187. IEEE, November 1991.

[MC81] C. Mead and L. Conway. *Mead and Conway: Introduction to VLSI Systems*. Addison-Wesley, 1981.

[McC56a] E. J. McCluskey. Detection of Group Invariance or Total Symmetry of a Boolean Function. *Bell Syst. tech J.*, 35(6):1445–1453, November 1956.

[McC56b] E. J. McCluskey. Minimization of Boolean Functions. *Bell Syst. tech J.*, 35(6):1417–1444, November 1956.

[McC86] E. J. McCluskey. *Logic Design Principles With Emphasis on Testable Semicustom Circuits*. Prentice-Hall, 1986.

[MD88] F. Mailhot and G. De Micheli. Automatic Layout and Optimization of Static CMOS Cells. In *International Conference on Computer Design*, pages 180–185. IEEE, October 1988.

[MH87] R. L. Maziasz and J. P. Hayes. Layout Optimization of CMOS Functional Cells. In *Design Automation Conference*, pages 544–551. IEEE, June 1987.

[MJH89] C. R. Morrison, R. M. Jacoby, and G. D. Hachtel. TECHMAP: Technology Mapping with Delay and Area Optimization. In G. Saucier and P. M. McLellan, editors, *Logic and Architecture Synthesis for Silicon Compilers*, pages 53–64. North-Holland, 1989.

[MKLC89] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The Transduction Method: Design of Logic Networks based on Permissible Functions. *IEEE Transactions on Computers*, October 1989.

[MSSB91] P. C. McGeer, A. Saldanha, P. R. Stephan, and R. K. Brayton. Timing Analysis and Delay-Fault Test Generation using Path-Recursive Functions. In *International Conference on Computer-Aided Design*, pages 180–183. IEEE, November 1991.

[Mur71] S. Muroga. *Threshold Logic and its Applications*. John Wiley, 1971.

[Nak67] I. Nakata. On Compiling Algorithms for Arithmetic Expressions. *Communications of the ACM*, 10(8):492–494, August 1967.

[ND91] S. Nowick and D. Dill. Automatic Synthesis of Locally-Clocked Asynchronous State Machines. In *International Conference on Computer-Aided Design*, pages 318–321. IEEE, November 1991.

[New87] A. R. Newton. Symbolic Layout and Procedural Design. In G. De Micheli and A. Sangiovanni-Vincentelli and P. Antognetti, editor, *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, pages 65–112. Martinus Nijhoff, 1987.

[NvMM88] S. Note, J. van Meerbergen, and F. Catthoor H. De Man. Automated Synthesis of a High-Speed Cordic Algorithm with the Cathedral-III Compilation System. In *International Symposium on Circuits and Systems*, pages 581–584. IEEE, June 1988.

[Ott82] R. Otten. Automatic Floor-plan Design. In *Design Automation Conference*, pages 261–267. IEEE, June 1982.

[PKG86] P. G. Paulin, J. Knight, and E. Girczyc. HAL: A Multi-paradigm Approach to Automatic Data-path Synthesis. In *23rd Design Automation Conference*, pages 263–270. IEEE/ACM, June 1986.

[PP89] P. G. Paulin and F. J. Poirot. Logic Decomposition Algorithms for the Timing Optimization of Multi-Level Logic. In *26th Design Automation Conference*, pages 329–333. IEEE/ACM, June 1989.

[PPM86] A. Parker, J. Pizarro, and M. Mlinar. MAHA: A Program for Data Path Synthesis. In *Design Automation Conference*, pages 461–466. IEEE, June 1986.

[Pre72] R. J. Preiss. Introduction. In M. A. Breuer, editor, *Design Automation of Digital Systems: Theory and Techniques*, pages 1–19. Prentice-Hall, 1972.

[PS90] M. Pipponzi and F. Somenzi. An Alternative Algorithm for the Binate Covering Problem. In *European Design Automation Conference*, pages 208–211. IEEE, March 1990.

[Qui52] W. V. Quine. The Problem of Simplifying Truth Functions. *American Mathematical Monthly*, 59(8):521–531, October 1952.

[Red69] R. R. Redziejowski. On Arithmetic Expressions and trees. *Communications of the ACM*, 12(2):81–84, February 1969.

[RH67] S. U. Robinson and R. W. House. Gimpel's Reduction Technique Extended to the Covering Problem with Costs. *IEEE Transactions on Electronic Computers*, 16:509–514, August 1967.

[RK62] J. P. Roth and R. M. Karp. Minimization Over Boolean Graphs. *IBM journal of Research and Development*, 6(2):227–238, April 1962.

[Rud89a] R. Rudell. *Logic Synthesis for VLSI Design*. PhD thesis, U. C. Berkeley, April 1989. Memorandum UCB/ERL M89/49.

[Rud89b] R. Rudell. Technology Mapping for Delay. In *International Workshop on Logic Synthesis*, page 8.2. MCNC, May 1989.

[San87] A. Sangiovanni-Vincentelli. Automatic Layout of Integrated Circuits. In G. De Micheli and A. Sangiovanni-Vincentelli and P. Antognetti, editor, *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, pages 113–195. Martinus Nijhoff, 1987.

[Sas86] T. Sasao. MACDAS: Multi-level AND-OR Circuit Synthesis using Two-Variable Function Generators. In *23rd Design Automation Conference*, pages 86–93. ACM/IEEE, June 1986.

[Sav91] H. Savoj, April 1991. Private communication.

[SB90] H. Savoj and R. K. Brayton. The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks. In *Design Automation Conference*, pages 297–301. ACM/IEEE, June 1990.

[SB91] H. Savoj and R. K. Brayton. Observability Relations and Observability Don't Cares. In *International Conference on Computer-Aided Design*, pages 518–521. IEEE, November 1991.

[SBK+87] S. Suzuki, T. Bitoh, M. Kakimoto, K. Takahashi, and T. Sugimoto. TRIP: An Automated Technology Mapping System. In *Design Automation Conference*, pages 523–529. ACM/IEEE, June 1987.

[SBT91] H. Savoj, R. K. Brayton, and H. Touati. Extracting Local Don't Cares for Network Optimization. In *International Conference on Computer-Aided Design*, pages 514–517. IEEE, November 1991.

[Sco58] N. R. Scott. Notes on Mathematical Logic and Switching Networks. In *Advanced Theory of the Logical Design of Digital Computers*. University of Michigan, June 1958.

[SD68] P. R. Schneider and D. L. Dietmeyer. An Algorithm for Synthesis of Multiple-Output Combinational Logic. *IEEE Transactions on Computers*, 17(2):117–128, February 1968.

[SSM+92] E.M. Sentovich, K.J. Singh, C. Moon, H. Savoj, R.K. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *submitted to the 29th Design Automation Conference*, June 1992.

[SSV90] K. J. Singh and A. Sangiovanni-Vincentelli. A Heuristic Algorithm for the Fanout Problem. In *Design Automation Conference*, pages 357–360. ACM/IEEE, June 1990.

[STMF90] H. Sato, N. Takahashi, Y. Matsunaga, and M. Fujita. Boolean Technology Mapping for Both ECL and CMOS Circuits Based on Permissible Functions and Binary Decision Diagrams. In *International Conference on Computer Design*, pages 286–289. IEEE, October 1990.

[SU70] R. Sethi and J. D. Ullman. The Generation of Optimal Code for Arithmetic Expressions. *Journal of the ACM*, 17(4):715–728, October 1970.

[SW79] A. Svoboda and D. E. White. *Advanced Logical Circuit Design Techniques*. Garland STPM Press, New York, 1979.

[SWBSV88] K. J. Singh, A. R. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Timing Optimization of Combinational Logic. In *International Conference on Computer-Aided Design*, pages 282–285. IEEE, November 1988.

[TDW+89] D. Thomas, E. Dirkes, R. Walker, J. Rajan, J. Nestor, and R. Blackburn. The System Architect's Workbench. In *Design Automation Conference*, pages 40–45. IEEE, June 1989.

[THYMM89] Robert W. Brodersen Teresa H.-Y. Meng and David G. Messerschmitt. Automatic Synthesis of Asynchronous Circuits from High-Level Specifications. *IEEE Transactions on CAD/ICAS*, 8(11):1185–1205, November 1989.

[TJB86] L. Trevillyan, W. Joyner, and L. Berman. Global Flow Analysis in Automatic Logic Design. *IEEE Transactions on Computers*, 35(1):77–81, January 1986.

[Tji85] S. W. K. Tjiang. Twig Reference Manual. Technical report, AT&T Bell Labs, 1985. TR 120.

[TMBW90] H. J. Touati, C. W. Moon, R. K. Brayton, and A. Wang. Performance Oriented Technology Mapping. In W. J. Dally, editor, *Advanced Researcgh in VLSI: Proceedings of the 6th MIT Conference*, pages 79–97. MIT Press, January 1990.

[Tou90] H. Touati. *Performance-Oriented Technology Mapping*. PhD thesis, U. C. Berkeley, November 1990. Memorandum UCB/ERL M90/109.

[TSL+90] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit State Enumaration of Finite State Machines Using BDD's. In *International Conference on Computer-Aided Design*, pages 130–133. IEEE, November 1990.

[Uv81] T. Uehara and W. M. vanCleemput. Optimal layout of CMOS Functional Arrays. *IEEE Transactions on Computers*, 30(5):305–312, May 1981.

[Wan91] A. Wang, November 1991. Private communication.

[WH91] T. E. Williams and M. A. Horowitz. A Zero-Overhead Self-Timed 160ns 54b CMOS Divider. In *International Solid-State Circuits Conference*, pages 98–99. IEEE, February 1991.

[Wil90] T. Williams. Latency and Throughput Tradeoffs in Self-timed Speed-independent Pipelines and Rings. Technical report, Stanford University, August 1990. CSL-TR-90-431.

[YCL91] C. W. Yeh, C. K. Cheng, and T.T. Y. Lin. A General Purpose Multiple Way Partitioning Algorithm. In *Design Automation Conference*, pages 421–426. IEEE, June 1991.

[Zim79] G. Zimmermann. The MIMOLA Design System: Detailed Description of the Software System. In *Design Automation Conference*, pages 56–63. ACM/IEEE, June 1979.