

Constructive Multi-Level Synthesis by Way of Functional Properties

by

Victor Nikolayevich Kravets

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2001

Doctoral Committee:

Professor Kareem A. Sakallah, Chair

Professor Richard R. Brown

Professor Edward S. Davidson

Professor John P. Hayes

Doctor H. Peter Hofstee

Professor Marios C. Papaefthymiou

© Victor Nikolayevich Kravets 2001
All Rights Reserved

To my family.

Acknowledgments

Many people have made my Ph.D. experience exciting and enjoyable. This work would be impossible without their support and encouragement throughout my years at the University of Michigan.

First and foremost I am indebted to Professor Karem Sakallah for his impact on my professional development. His effort has been instrumental in this work from my very first days in the program. Interacting with him I was able to shape my thinking, and learn how to carry research from an initial idea to a matured concept. His ability to capture the essence of seemingly complex matters, and turn it into a simple presentation of truth and beauty have always amazed me. His constructive criticism has also made me work harder and strive for excellence.

I am much obliged to the members of my dissertation committee who helped me shape this work at its various stages. Interaction with Professor John Hayes inspired me to emphasize mathematical rigor in the studied concepts. Discussions with Professor Edward Davidson helped me maintain a practical outlook on the problem. I am thankful to Professor Marios Papaefthymiou and Professor Richard Brown for their continual involvement in my progress. Special acknowledgments go to Dr. Peter Hofstee for agreeing to be on my committee, and generously sharing his time and energy to improve this work. I have found his creative insights very helpful and amiable.

I owe a debt of gratitude to my colleagues in the Advanced Computer Architecture Laboratory who helped me professionally and personally. I am especially thankful to Mike Riepe for his help in my early research and his continual encouragement. I had shared many enjoyable professional and recreational moments with David Van Campenhout. He never failed to respond with his good technical advice, good will and good humor—whichever was needed. I was lucky to spend countless hours discussing this work, and the field of design automation in general, with Juan Antonio Carballo. His ability to provide feedback offering right form on presentation and display of research have always been a source of admiration. I am thankful to Wee Teck Ng for being in

touch with my progress, and making sure I stay fresh through my running regiment. Other members of the ACAL have contributed to a pleasant and supportive environment—I would like to extend warm thanks to Jeff Bell, Koushik Das, Brian Davis, Jim Dundas, Steve Raasch, Tim Strong.

I am grateful to Leyla Nazhandali for reading drafts of this dissertation. Her careful attention to a detail helped me shorten the tedious writing process. I would also like to thank Professor Igor Markov for contributing with his valuable suggestions. Jesse Whittemore and Gi-Joon Nam provided me with corrections of the early draft of the dissertation, and I am thankful for their help.

The research presented in the dissertation was supported in large by the Semiconductor Research Corporation. Aside from the funding, SRC has provided me with a chance to interact with the research community outside of the University of Michigan. I am grateful for this opportunity.

My boundless gratitude goes to my parents. Without their persistent support and care I could not have envisioned completion of this process.

Table of Contents

Dedication	ii
Acknowledgments	iii
List of Figures	ix
List of Tables	xi
List of Appendices	xii
Chapter 1	
Introduction	1
1.1 VLSI Circuit Synthesis	1
1.2 Multi-Level Synthesis: A Short Review	4
1.2.1 Early work	4
1.2.2 The advent of practical synthesis	6
1.2.3 Addressing “serialization” in synthesis	8
1.3 Motivation	10
1.4 Overview	11
Chapter 2	
Notation and Preliminaries	13
2.1 Sets, Relations, and Functions	13
2.2 Boolean Algebra	16
2.3 Boolean Functions	17
2.3.1 Operations on Boolean functions	18
2.3.2 Expansions of Boolean functions	20
2.4 Partial Specification of Boolean Functions	22

2.5	Boolean Function Intervals	24
2.6	Binary-Decision Diagrams	28
2.7	Boolean Networks	29
2.8	Summary	31

Chapter 3

Constructive Synthesis	32
3.1	Constructive Synthesis Flow 32
3.2	M32: A Prototype Constructive Synthesis Program 34
3.2.1	Algorithm overview 34
3.2.2	<i>SelectFunction</i> 35
3.2.3	<i>GenerateDivisor</i> 36
3.2.4	<i>IntroduceGates</i> 38
3.2.5	<i>Substitute</i> 40
3.2.6	Synthesis example 41
3.3	Empirical Evaluation 42
3.3.1	Comparative performance analysis 42
3.3.2	Effects of richer gate libraries 45
3.3.3	Dynamic execution highlights 46
3.4	Constructive Synthesis Objectives 49
3.5	Summary 51

Chapter 4

Decomposition Space in Constructive Synthesis	52
4.1	Symbolic Formulation of Decomposition 52
4.1.1	Generic decomposition template 52
4.1.2	Computation of composition function 53
4.1.3	Computation of decomposition functions 56
4.2	Enforcing Practical Decomposition Constraints 59
4.2.1	Effect of fan-in constraints 59
4.2.2	Modified decomposition template 61
4.2.3	Effect of support constraints 65
4.3	Computational Considerations 69
4.3.1	Exploring common computational core 69
4.3.2	Dealing with large sets of decomposition variables 70

4.4 Summary	71
-----------------------	----

Chapter 5

Semantic Structure of Boolean Functions	72
5.1 Introduction and Prior Work	72
5.2 Symmetry as Permutations	74
5.3 Classical First-Order Symmetries	77
5.4 Higher-Order Symmetries	79
5.5 Symmetries Under Phase Assignment	84
5.6 Characterization of Function Symmetry in Benchmark Circuits	87
5.7 Summary	91

Chapter 6

Libraries for the Decomposition Patterns	92
6.1 Approach	92
6.2 Library for a Single Pattern Function	93
6.3 Library for a Decomposition Pattern	96
6.4 Computed Libraries	98
6.5 Summary	101

Chapter 7

Practical Issues	102
7.1 Constructive Synthesis Flow	102
7.1.1 Selection of a function to decompose	103
7.1.2 Selection of decomposition variables	104
7.1.3 Computation of decomposition functions	106
7.1.4 Logic re-expression	107
7.2 Implementation Issues	110
7.2.1 Computation of pattern function	111
7.2.2 Decomposition cache	112
7.2.3 BDD management	114
7.2.4 Evolving Boolean network	118
7.3 Experimental Analysis	119
7.3.1 Setup	119
7.3.2 Results	120

7.3.3	Evaluation.....	122
7.3.4	Highlights.....	124
7.4	Summary.....	126
Chapter 8		
	Conclusions	127
8.1	Summary.....	127
8.2	Contributions	128
8.3	Future Work	129
	Appendices	131
	Bibliography	159

List of Figures

Figure 1.1:	Major synthesis steps in the design of digital integrated circuits. This dissertation is concerned with multi-level logic synthesis.	2
Figure 1.2:	Effects of wiring on circuit area and delay	9
Figure 3.1:	Schematic illustration of constructive synthesis	33
Figure 3.2:	Overall algorithmic flow in M32	34
Figure 3.3:	Example calculation of structural cost of a divisor according to formula (3.1)	37
Figure 3.4:	1-level lookahead search tree used in divisor selection; costs are based on the implementation network of Figure 3.3-a	38
Figure 3.5:	Meta rules describing variable support selection and construction of the circuit.	39
Figure 3.6:	Synthesis of the full adder in M32 and SIS-1.2	41
Figure 3.7:	<code>Cordic</code> relative layout areas (shown to the same scale)	45
Figure 3.8:	Dynamic execution curves for the <code>clip</code> circuit using two different covers.	47
Figure 3.9:	Two incrementally-different implementations of the <code>cordic</code> circuit	48
Figure 3.10:	Integration of decomposition type and library via semantic properties of a functional specification.	50
Figure 4.1:	Generic decomposition step	53
Figure 4.2:	Illustration of the decomposition template $f(\mathbf{x}) = h(\mathbf{g}(\mathbf{x}_g), \mathbf{x}_h)$	61
Figure 4.3:	Circuit resulting from two successive decomposition steps; it has two topological levels of logic	62
Figure 4.4:	BDD for $G(\Gamma)$ which encodes all 3-to-2 decomposition functions for $f = (a\bar{b}d + a\bar{e} + adc + b\bar{e}d + \bar{b}c\bar{e})$ of Example 4.8. (Inversion bubbles on graph edges denote complementation of the functions associated with their rooted subgraphs.)	65
Figure 4.5:	Bounds on the number of distinct cofactors as a function of the number of decomposition variables for $n = 8$	67

Figure 4.6:	Dependence of the distinct cofactor counts on the number of decomposition variables.	68
Figure 5.1:	Illustration of function symmetry	73
Figure 5.2:	Symmetry induced hierarchical partition of variables for function $f(a, b, c, d, x, y) = abxy + cdxy$	75
Figure 5.3:	Limitation of symmetry groups to represent all invariant permutations of a function	76
Figure 5.4:	Construction of first-order symmetry partition P for a function f with support X	77
Figure 5.5:	Symmetry structures for two example functions.	81
Figure 5.6:	Symmetry under phase assignment. $\{\bar{a}, b\}$ and $\{a, \bar{b}\}$ represent two equivalent ways of denoting the symmetry of f with respect to a and b	85
Figure 5.7:	A multiplexer circuit and its semantic symmetry structures	86
Figure 5.8:	Hierarchical symmetry partition of $\tau 481$. Each subtree is annotated at its root with the number of variable permutations it represents.	89
Figure 5.9:	High level structure of the C499 single-error-correcting circuit	90
Figure 7.1:	Heuristic for the selection of a function to decompose. Numbers annotating the unimplemented nodes denote their complexity. Node v_1 is chosen first for decomposition because it originates the path with the highest logic complexity (path P with complexity 120).	103
Figure 7.2:	Selection of the decomposition variables	104
Figure 7.3:	Ranking and cofactor requirements for the s -to- t decomposition patterns when selecting symmetric decomposition variables	105
Figure 7.4:	The re-express algorithm to select composition function when decomposition functions are given.	109
Figure 7.5:	Composition function selection in re-express for the 3-to-2 reduction.	110
Figure 7.6:	Algorithm to create pattern function $F(\mathbf{x}_g, Z)$ for a given $f(\mathbf{x}_g, \mathbf{x}_h)$	111
Figure 7.7:	Mapping involved in caching of decomposition solutions	113
Figure 7.8:	Interaction between unique tables in M31.	115
Figure 7.9:	Effect of the new variables location on the BDD size of composition function	117
Figure 7.10:	Evolving Boolean network and its state attributes	118
Figure 7.11:	Schematics for the $\tau d73$ circuit, synthesized with SIS-1.2 and M31.	124
Figure 7.12:	A 16-bit adder circuit generated by M31	125
Figure B.1:	Example of observability don't cares for f_2 ; they show that y_2 is redundant.	142
Figure B.2:	Illustration of Boolean relations in capturing flexibility of a Boolean network.	144

List of Tables

Table 2.1:	Some properties of the cofactor operation (c is a cube)	19
Table 2.2:	Some properties of variable abstraction operations.	19
Table 3.1:	Pre-layout synthesis results	44
Table 3.2:	Post-layout synthesis results	44
Table 3.3:	Results from mapping to a richer technology library	46
Table 5.1:	Summary of symmetry characterization of benchmark circuits	87
Table 6.1:	Characteristics of module libraries necessary for support-reducing symmetric decomposition	95
Table 7.1:	Symmetry-induced partition of variables in benchmark circuits	120
Table 7.2:	M31 synthesis results: without and with the pre-computed symmetry library	121
Table 7.3:	SIS-1.2 synthesis results: without and with the pre-computed symmetry library	121
Table 7.4:	Characteristic of the M31 and SIS-1.2 circuits as estimated by SIS-1.2 after they were mapped into mcnc library	122
Table 7.5:	Characteristics of the adders synthesized by M31	125

List of Appendices

Appendix A

Theorem Proofs	132
--------------------------	-----

Appendix B

Modelling Boolean Network Optimization With Intervals	141
---	-----

Appendix C

Symmetry Structures	146
-------------------------------	-----

Appendix D

Synthesis Data	152
--------------------------	-----

Chapter 1

Introduction

In this dissertation we investigate the computer-automated multi-level logic synthesis of combinational circuits. This is a major step in the computer-aided design (CAD) flow of integrated circuits and plays a significant role in determining overall circuit quality. In this chapter we establish context for this problem, briefly review previous synthesis efforts, and outline the remainder of this dissertation.

1.1 VLSI Circuit Synthesis

Very Large Scale Integration (VLSI) technology has been the key enabler for implementing modern digital systems. Today's microprocessors, memories, and application-specific integrated circuits (ASICs) are the beneficiaries of a steady doubling, over the last thirty years, of transistor counts every 18 months (known as Moore's law). This unprecedented increase in integration levels has led to dramatic reductions in production costs and significant increases in performance and functionality. The design of such highly complex systems was also critically dependent on the use of CAD tools in all phases of the design process: synthesis, optimization, analysis, and verification. This dissertation addresses one of the synthesis steps in this automatic design flow, namely the creation of a low-level structural description of a design from a more abstract form. The major synthesis steps in this design flow are depicted in Figure 1.1.

The starting point of design synthesis is typically a textual description, in an appropriate hardware description language (HDL), of the desired functional behavior. At this level, the design is specified in terms of abstract data manipulation operations which are organized into larger blocks

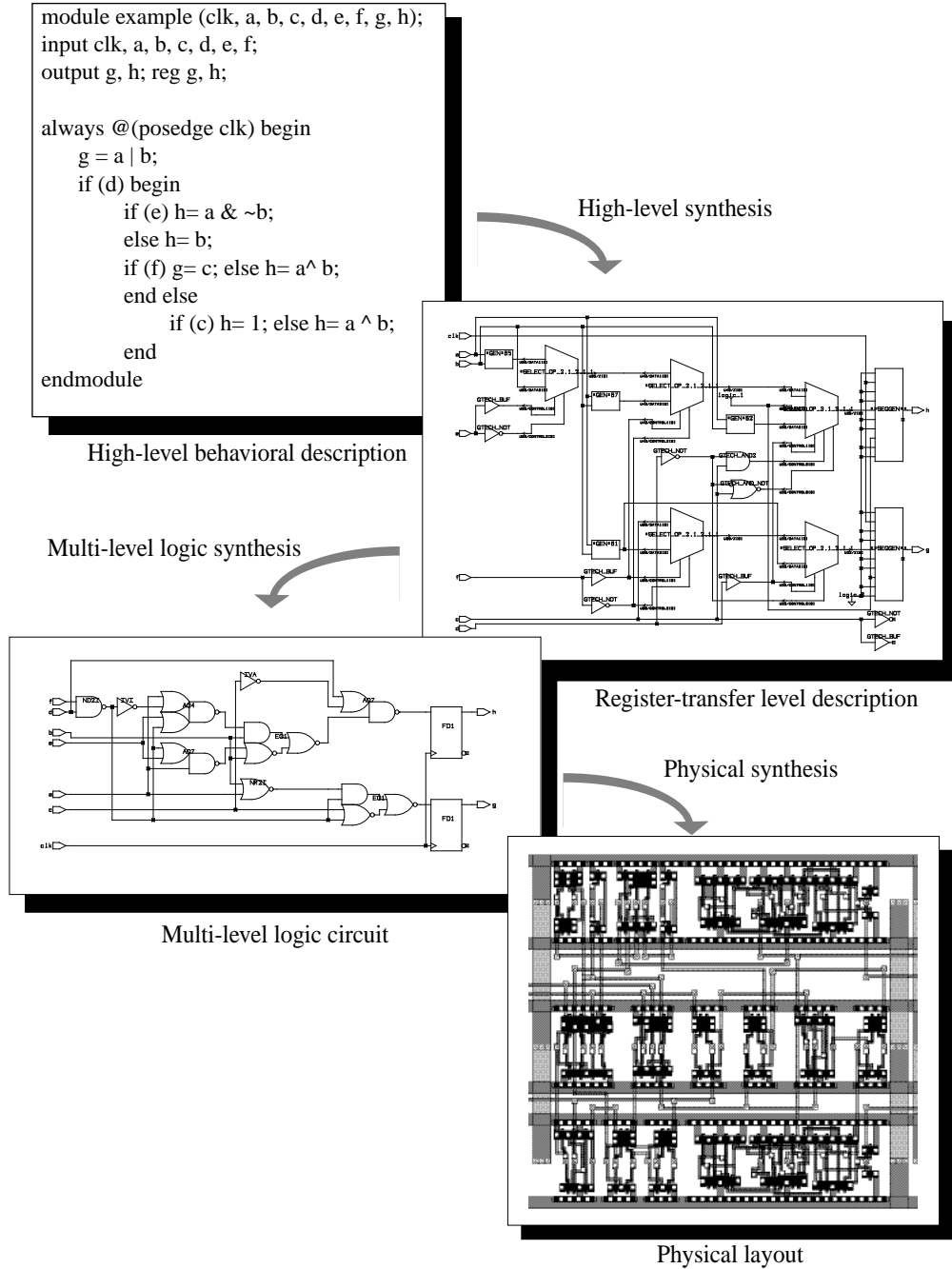


Figure 1.1: Major synthesis steps in the design of digital integrated circuits. This dissertation is concerned with multi-level logic synthesis

using control constructs. High-level synthesis transforms such a description into an appropriate structural representation at the register-transfer level (RTL). Typical RTL components include data storage elements (registers, memories, etc.), functional modules (adders, shifters, etc.), and data steering logic (busses, multiplexors, etc.). The next major synthesis step creates multi-level logic gate realizations for each of the combinational (i.e. memoryless) parts of the RTL description.

Such multi-level logic synthesis is the primary focus of this dissertation. The primitive building blocks used in such synthesis are typically 3- to 4-input single-output cells from a pre-characterized technology library. The final synthesis step generates a complete layout of the design by placing and routing its gate-level implementation, and by synthesizing a suitable power/ground distribution grid and a clock tree.

Each of the above synthesis steps (high-level, logic, and physical) involves a multiple-objective optimization that seeks an appropriate trade-off among the design's area, delay, testability, and more recently, power consumption. Area minimization leads to increased chip yields, and hence lower costs, as smaller circuits can be manufactured more reliably, and are easier to fit on a chip; smaller circuits also often have decreased delay. Delay minimization creates faster circuits which are essential in high-performance computing applications. Improving the testability properties of a circuit can lead to higher reliability and reduced testing costs. Finally, minimizing power consumption has become crucial with the proliferation of hand-held and portable computing devices, and is becoming a major issue in high-performance designs as well.

These design objectives interact in complex ways. Synthesizing a circuit that optimizes across a set of these objectives is a difficult task due to the tremendously large space of potential solutions. Finding a solution in this space that meets the specified objective(s) may, therefore, be computationally quite expensive, if not impossible. In the face of such complexity, most synthesis approaches resort to a serialization of the design creation process by approximating, or entirely ignoring, some of the contributing components of the various optimization objectives. For example, in physical synthesis, layout generation is serialized into the steps of placement, global routing, and detailed routing. Placement is done by making certain assumptions about the routing requirements and the resulting placement solution becomes a constraint for the subsequent routing optimization. In most cases, this is an acceptable strategy that yields good layouts. In some cases, however, the placement constraints preclude the successful routing of the design or lead to routing solutions that do not meet the delay objectives. In such cases, it is necessary to iterate the placement/routing steps until an acceptable solution meeting all objectives is found.

This same serialization paradigm is currently the pre-dominant way for dealing with the complexity of multi-level logic synthesis. Specifically, the synthesis process is split into two phases: a *technology-independent* global restructuring of the RTL logical specifications followed by a *technology mapping* of the resulting structure to a specified cell library. The technology-independent optimizations work on logic representations that do not directly model, and hence are uncon-

strained by, the particular primitive building blocks in this library. The technology mapping phase, on the other hand, is constrained by the structure produced in the technology-independent phase and can only achieve local optimizations as it makes choices to produce the gate-level implementation. Iteration between these two phases may, therefore, be necessary to achieve “closure,” i.e. to satisfy all optimization objectives, especially delay.

There are two fundamental concepts influencing research in multi-level synthesis, as well as synthesis in general: derivation of flexibility in the implementation of a design, and exploiting this flexibility when optimizing the implementation. One source of flexibility is the incomplete specification of a design, or the parts within it. Thus, the implementation changes remain consistent with the specification. The other source of flexibility is invariant transformations which leave the behavior of the actual implementation unchanged. Most research has been done regarding the second source of flexibility as it perceived to be a more difficult problem and to have a more significant impact on the design quality.

1.2 Multi-Level Synthesis: A Short Review

The quest for the automatic synthesis of logic circuits has a long history. In this section we highlight the salient milestones from the last five decades of research and development in this area. We divide the presentation into three parts: early theoretical work in the fifties and sixties, tool development and wide-spread adoption in the seventies and eighties, and modern research efforts that address the limitations of existing approaches in the new world of deep submicron (DSM) ICs.

1.2.1 Early work

Early research in combinational synthesis was primarily concerned with finding optimal forms for realizing a given Boolean function.

Two-level synthesis. Synthesis algorithms were first sought for the two-level logic minimization problem. Quine [91] proposed the first solution to this problem in the 1950s; his method was subsequently improved by McCluskey [80], and has since become known as the Quine-McCluskey two-level minimization procedure. The essence of this procedure is a systematic exploration of the search space of two-level circuits seeking a realization with minimal area. The enumerative nature of such an approach makes it exponentially complex in both space and time, and limits its applicability to relatively small functions with, typically, a dozen or fewer inputs. The advantage of two-

level forms is that they can be directly implemented in VLSI using programmable logic structures—PLAs and PALs [45]—whose areas and delays can be estimated with high accuracy. However, general use of two-level synthesis is hampered by the computational infeasibility of optimally synthesizing large functions in two levels, and by the practical technological limits on the maximum fan-in and fan-out of logic gates. In addition, it can be easily shown that certain multi-level realizations are both smaller and faster than the corresponding optimal two-level forms. Despite these shortcomings, exact and approximate two-level synthesis is sometimes used as a step in multi-level synthesis algorithms.

Multi-level synthesis. Research in multi-level synthesis emerged soon after the initial solutions to the two-level minimization problem were stated. Similar in spirit to those of the two-level problem, the original multi-level approaches were based on a systematic exploration of the solution search space. The dominant view at that time was that two-level circuits were a special case of multi-level circuits, and that the algorithmic solution to the former should generalize to solve the latter.

The fundamental notion in multi-level synthesis is that of *functional decomposition*, i.e. the possibility of expressing a given Boolean function f in terms of a set of other, perhaps simpler, functions g_1, \dots, g_k . Ashenhurst [2] was the first to derive a condition for checking whether a function f has a non-trivial decomposition satisfying the template:

$$f(x_1, x_2, \dots, x_n) = h(g(x_1, \dots, x_s), x_{s+1}, \dots, x_n)$$

His observation laid the foundation for *classical* decomposition theory, which was shortly generalized by Curtis [35], and Roth and Karp [92], to handle other, more complex, decomposition forms. These works represent the first accounts of complete multi-level synthesis algorithms. The general approach was a search procedure that examined all possible decompositions lexicographically, pruning the search by some simple lower bounds on circuit cost, and terminating when a minimum-cost realization was found.

Several other enumeration techniques for multi-level synthesis were explored in the 1960s. Hellerman [58] proposed an algorithm that enumerated all directed acyclic graphs, and tested whether each generated graph implements the desired function. The advances in two-level minimization motivated Lawler [72] to generalize the notion of two-level prime implicants to the multi-level case. His approach showed how these multi-level implicants can be used to obtain “absolutely minimal” factored forms. Gimpel [50] proposed an optimal algorithm for synthesis of three-level networks in terms of NAND gates. Gimpel’s approach is similar in spirit to the work of

Lawler: it generalized the two-level enumeration approach to three levels. Davidson presented a branch-and-bound algorithm for NAND network synthesis [38]. The algorithm constructs a network realization by a sequence of local decisions starting from the primary outputs, and incrementally introduces new gates.

Most of this early work on multi-level synthesis, while theoretically significant, failed to achieve the elusive goal of generating optimal circuits. The complexity of exhaustively enumerating the solution space limited the applicability of these approaches to very small circuits, and rendered them impractical for general-purpose synthesis.

1.2.2 The advent of practical synthesis

The growing complexity of VLSI in the late seventies necessitated new scalable synthesis techniques that sought approximate, rather than optimal, multi-level circuit solutions. Most synthesis tools in use today are based on the premise that the search for optimal solutions is intractable, and are designed, instead, to find acceptable sub-optimal realizations. These tools typically operate on a multi-level representation of the functions being synthesized, continually transforming it until a satisfactory solution is found, and can be roughly classified into two broad categories based on the granularity of transformations used. *Local* transformation approaches modify the current “solution” incrementally by making appropriate changes in its immediate neighborhood. In contrast, *global* transformation approaches seek good multi-level topologies by making large-scale changes to the implementation structure while disregarding technological considerations; a second “mapping” phase insures compliance of the resulting multi-level structure with technology constraints.

Local transformation approaches. Original local optimization methods perform rule-based transformations, which are a set of *ad hoc* rules that are applied iteratively to patterns found in the network of logic gates. In the local optimization method each rule introduces a transformation by replacing a small subgraph of several gates in the network with another subgraph which is functionally equivalent but has a simpler realization according to some cost function. Initially the network consists of AND, OR, INV gates; decoders, multiplexers, adders, etc. After the simplification step these primitives are translated into an interconnection of INV to NAND gates through a sequences of transformations. Technology specific transformations are then applied as a final step in the process. Such transformations have limited optimization capability since they are local in nature, and do not have global view on the design. Examples of systems based on this approach are LSS [37] and LORES/EX [63].

In the later years interest in local transformations has shifted to a more rigorous theory of don't cares [4]. They arise from the structural and external properties of a network, and are used to describe flexibility required to locally optimize a network node. In general, the use of don't cares is a difficult task since their number may become unmanageably large even for a small neighborhood of a node. Furthermore, it is not always clear from which part of a circuit the don't cares should be extracted. Various algorithms have been proposed to extract subsets of don't cares (see e.g., [36, 96]). Optimization with don't cares has a close relation to other optimization methods, such as transduction [88], redundancy removal [33] and global flow analysis [5, 6]; techniques based on these methods have been successfully implemented in BooleDozer [111], a synthesis tool from IBM.

Global transformation approaches. The computational limitations of the classical theory for functional decomposition motivated the development of algorithms which are effective in partitioning complex logic functions. These ideas are based on the notion of *algebraic* factorization applied to sum-of-products (SOP) expressions; the technique is described in [14] and [21]. Algebraic decomposition techniques have experienced the most success to date in the field of multi-level synthesis. They are capable of handling large combinational blocks, and produce very good results for control logic. However, representing logic of higher level abstraction with SOP forms makes it difficult to explore the structural flexibility of the original description: It can lead to the loss of a compact description of the original equations, and algebraic decomposition is too restrictive to rediscover their structure. Examples of systems which rely on the algebraic techniques are MIS [20], SOCRATES [3], and more recently SIS [103]. In more recent years much attention has been also given to AND-XOR decompositions [116, 30, 59, 41].

The advent of binary decision diagrams (BDDs) [24] and their variants rekindled interest in classical decomposition techniques. In recent years researchers have successfully applied Roth and Karp decomposition in FPGA synthesis [28, 71, 87, 99, 121]. These approaches decompose a function recursively until each of the generated subfunctions meets a given fan-in constraint, typically 5. However, since fan-in count is the only notion of node complexity in these approaches, they do not extend easily to a library-specific synthesis. A number of approaches have also been developed which explore the structure of the decision diagram representation of a given function [8, 41, 122, 124]. The close relation between BDDs and multiplexer circuits has also lead to several approaches to synthesis of pass transistor logic (PTL) [7, 12, 30, 75]; they are primarily based on a mapping of (decomposed) BDDs to PTL.

1.2.3 Addressing “serialization” in synthesis

Most of the current tools for multi-level synthesis split the optimization process into technology independent and technology dependent stages. The technology independent phase involves deciding how to partition the logic. The goal is to create a technology independent representation of a set of Boolean functions in multilevel form. The technology dependent stage is then responsible for implementing each partition of the logic. The advantages of this approach to multi-level synthesis are basically: 1) speed, since all computational steps are carefully designed to insure that they do not have worst-case exponential run time behavior, and 2) flexibility, since no assumptions are made about the logic specifications being synthesized. Such multi-stage approaches to complex optimization problems are common in electronic design automation (e.g. placement followed by routing), and are usually necessitated by the difficulty of solving these problems conjointly.

This “serialization” also has several shortcomings – the most serious being an inadequate model of interconnect at the logic restructuring stage, and the non-incremental nature of the topological/functional transformations. Indeed, it implies that decisions made in earlier stages must necessarily be based on loose estimates of what later stages can accomplish. At the same time, the solutions produced by early stages place limitations on the degrees of freedom when improving the final implementation of a design. For two-stage logic synthesis, decisions made during the technology-independent stage significantly determine the structure of a circuit. They are made with no regard for the downstream technology. When the technology characteristics become available in the mapping stage it is often too late to augment the effects of these decisions to improve circuit quality.

To illustrate the impact of a technology—wires in particular—on circuit we conducted a controlled experiment that compared the layouts of combinational circuits that have the same active areas but different interconnect patterns. (The layouts were generated using the Epoch [43] standard cell place and route tools for a two-layer 0.5 μ m CMOS IC process.) The plot in Figure 1.2 shows that the total routing area, as well as delay per logic level as functions of average topological wire length given by

$$\text{Avg. topological wire length} = \frac{\sum_{n=1}^{\text{\# Edges}} L(n)}{\text{\# Edges}}$$

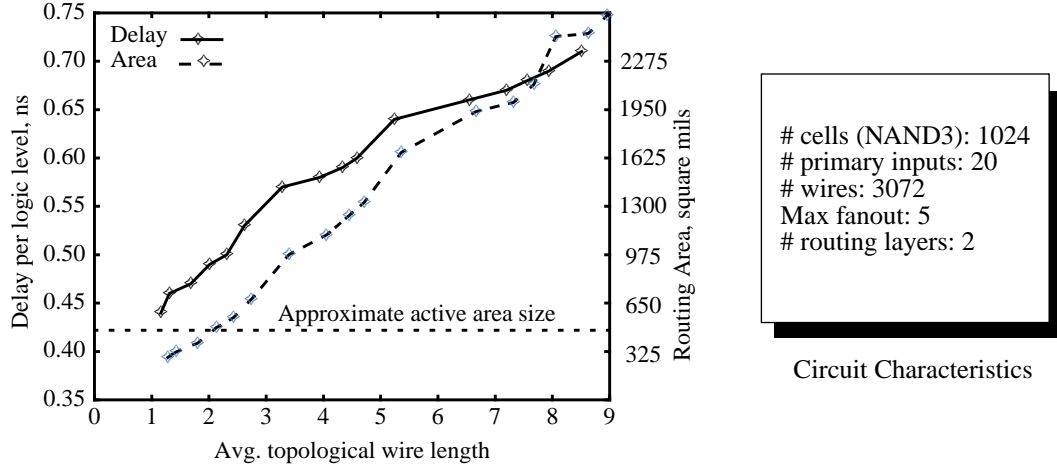


Figure 1.2: Effects of wiring on circuit area and delay

where $L(n)$ is the number of topological levels crossed by wire n and #Edges is the total number of wires in the circuit. As the figure clearly shows, routing area increases with increasing topological complexity, and begins to exceed active area when topological complexity is around 2. Similarly, signal delay per logic level increases with increased topological complexity. While these results may be specific to the particular IC technology and physical design system used in the experiment, they nevertheless confirm the general belief that wiring can be a significant contributor to area and delay. This wiring problem becomes more important with advances in CMOS technology since interconnections are becoming a major concern in today's high-performance, high-density ASIC designs [66].

The back-annotated approaches, which perform resynthesis after technology specific information is extracted from the mapped circuit, compensate partially for this problem. Given a *sign-off* information, these approaches would typically resynthesize the circuit through critical section correction [51, 125, 3, 82, 44, 107]. While this yields improvement in circuit quality, technology-independent and technology-dependent transformations still remain disconnected. In [74], the authors address this problem by dynamically modifying the set of AND2/INV decompositions while deleting others based on the actual cost function used in technology mapping. It allows technology-independent transformations to be part of the technology mapping. However, these transformations do not exhibit global knowledge about circuit structure and functionality

Realizing the need for synthesis to account for the “physical” information of back-end tools, Synopsys makes it possible to choose an appropriate wire-load model [70]. The wire-load models specified in the Synopsys technology library are based on statistical data which is design and process technology-dependent. Thus, inaccuracies in wire-load models can lead to synthesized

designs which are pessimistic, unroutable, or don't meet tight constraints after routing is performed. The synthesis process in Synopsys also relies on the methodology of technology-independent transformations, which is not suited to accounting for the final wire lengths of a design.

1.3 Motivation

It is widely acknowledged that current electronic design automation must be evolved to handle the challenges and opportunities of finer-featured fabrication processes. These methodologies are fundamentally premised on the principle of *separation of concerns*: a complex design flow is serialized into a sequence of manageable steps that are loosely coupled. In this scenario, decisions made in the early stages of design flow become binding constraints on later stages. Such serialization potentially yields less optimal designs than a methodology that simultaneously considers all design aspects. This is unavoidable, however, due to the practical infeasibility of concurrent optimization of all design parameters, and is deemed acceptable as long as the constraints that are fed forward can be met. The methodology breaks down completely, however, when these constraints become unsatisfiable; the typical action in such cases is an iteration that revisits earlier design stages to change suspected problematic decisions. Such iteration has become particularly necessary between the logical and physical synthesis steps due to the inability of layout synthesis to satisfy timing requirements, i.e. achieve *timing closure*.

Ideally, the time-wasting iteration between logic and layout synthesis in today's design methodologies could be eliminated by fusing these stages to simultaneously optimize the logical structure as well the spatial placement of a circuit. This, however, may be formidable task due to reasons of computational complexity. We formulate a more modest objective: to more directly relate the *functional structure* of a logic specification to the ultimate *topological* and *physical structures* of its physical realization.

We conjecture that functional specifications have global semantic attributes that can be profitably used to induce a favorable structural implementation, while reducing the run time complexity of the synthesis process. These attributes can have a profound effect on the suitability of one decomposition type over another. They can be further utilized to study requirements on the functionality of library primitives to make a particular decomposition type effective. The effect of the integration leads to improved synthesis quality, reflecting the global functional properties in the final circuit structure.

1.4 Overview

This work contains a study of an approach to multi-level synthesis. In Chapter 2 we introduce the essential notation, and derive fundamental concepts for this dissertation. The chapter introduces the notion of intervals, later used as a basic vehicle for functional decomposition.

Chapter 3 describes the *constructive* synthesis approach, which operates on a network. The basic idea in this approach is to constructively build the implementation network from primary inputs to primary outputs by interleaving the logic decomposition and library binding steps. The advantage of this approach is that it is able to consider the structural implications of candidate decompositions. The chapter describes a prototype implementation of this approach in the M32 synthesis tool. The tool manipulates sum-of-products expressions, relying on a form of algebraic division. The main goal of this implementation is to understand the implications of this constructive synthesis flow (e.g. its ability to control implementation structure) as opposed to fine tuning individual steps. The exercise helps us to point out the limitations of algebraic division as decomposition tool since it bears no relation to the nature of the functions being synthesized. In particular, the manner in which decomposition is performed and the content of the cell library can have a profound impact on the solution quality. Unrestricted Boolean transformations, on the other hand, are infeasible except for small-scale problems.

The observations made in Chapter 3 motivate the remainder of this dissertation. We conjecture that the inherent complexity of the unrestricted Boolean transformations can be addressed with a strategy that ties 1) the semantic properties of the functions being synthesized, 2) the structural attributes of the implementation network, 3) and carefully chosen decomposition primitives. To realize this integration, Chapter 4 introduces a novel formulation of decomposition. It is designed with the intent to capture complete decomposition flexibility arising in the constructive synthesis flow. Its symbolic formulation makes this formulation particularly suited for the implementation in the latest technology of decision diagrams

In Chapter 5 we study semantic structure of a function in terms of symmetries. The study in this chapter is motivated by our earlier argument that when guided by knowledge of the semantic structure of a function, synthesis can yield more “natural” implementations of the function. Thus, they are studied with the intention of establishing the key relation between desirable decompositions and the library of primitives which makes such decompositions possible. Specifically, symmetries allow us to define a decomposition type that restricts its decomposition functions to a

small library of pre-characterized symmetric primitives, and yet achieves a constrained form of decomposition; this is illustrated in Chapter 6.

We validate our synthesis argument in Chapter 7 which focuses on the implementation aspect of the studied techniques. The chapter shows that the efficient implementation of the symbolic formulation of decomposition is possible by leveraging recent advances in the representation of functions. In particular we describe an implementation in the M31 prototype tool, which links functional and implementation structures through a carefully-planned decomposition in terms of pre-computed decomposition patterns; the tool is implemented primarily to capture the symmetry structure. The empirical evaluation of M31 shows that relating semantic properties of a function to the final implementation is a powerful approach to synthesis.

A perspective on this work and directions for future research are given in Chapter 8.

Chapter 2

Notation and Preliminaries

This chapter provides the notation, definitions, and theoretical foundations needed throughout this dissertation. A quick review of basic set theory is followed by a summary of the primary techniques for representing and manipulating Boolean functions. We then discuss the important topic of partial specification of Boolean functions and their various representations. One particular representation, the Boolean function interval, is treated in detail because of its centrality to many of the computational tasks we describe in the rest of this thesis. The chapter concludes with a brief summary of the main data structures used for the symbolic representation of Boolean functions and digital circuits, namely binary-decision diagrams and Boolean networks.

2.1 Sets, Relations, and Functions

There are many excellent books providing comprehensive coverage of set theory. Among those are two classic works by Fraenkel [46] and Halmos [52]; they are suggested for further reading, as this section provides only the minimum notation and definitions needed to motivate further concepts.

Sets. A *set* is a collection of objects called *elements*, or *members*. If a is a member of set A then we write $a \in A$; similarly subset membership is denoted with \subseteq . Sets can be manipulated with various operations, the most common being:

$$A \cup B =_{\text{df}} \{a \mid a \in A \text{ or } a \in B\} \quad \text{union}$$

$$A \cap B =_{\text{df}} \{a \mid a \in A \text{ and } a \in B\} \quad \text{intersection}$$

$$\bar{A} =_{\text{df}} \{a \mid a \notin A\} \quad \text{complementation}$$

These operations can be applied in various sequences and to arbitrary numbers of sets. They are usually applied to make new sets out of other sets. For example, the operations of intersection and complement enable us to subtract one set from an other:

$$A - B =_{\text{df}} A \cap \bar{B} \quad \text{difference}$$

We often find ourselves needing to impose an ordering on the elements of a set. Such ordered sets are denoted with angle brackets instead of braces. For any number n of objects a_1, \dots, a_n , an ordered set $\langle a_1, \dots, a_n \rangle$ is also called an n -tuple. The precise definition of an ordered set is not really important so long as the following condition is satisfied:

$$\langle a_1, \dots, a_n \rangle = \langle b_1, \dots, b_n \rangle \Leftrightarrow a_i = b_i, 1 \leq i \leq n$$

We may also refer to an ordered set as a vector, and enclose its elements in parentheses. We can remove one element from an ordered set to obtain another ordered set of lower cardinality. We denote this subtraction operation by:

$$\langle a_1, \dots, a_n \rangle \setminus a_i =_{\text{df}} \langle a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n \rangle$$

Repeated application of this operation allows for the elimination of more than element.

For any sets A_1, \dots, A_n , not necessarily distinct, the set of all n -tuples

$$A_1 \times \dots \times A_n =_{\text{df}} \{ \langle a_1, \dots, a_n \rangle \mid a_i \in A_i, 1 \leq i \leq n \}$$

is their *Cartesian product*. When the sets are identical, the Cartesian product can be expressed in shorthand as:

$$A^n =_{\text{df}} \underbrace{A \times \dots \times A}_{n \text{ times}}$$

Note that A^n is a set.

The *power set* of a set A , written 2^A , is the set of subsets of A , i.e.

$$2^A =_{\text{df}} \{ A_i \mid A_i \subseteq A \}$$

The notation 2^A serves to remind us that the number of elements in the power set is $2^{|A|}$, where $|A|$ denotes number of elements in A .

Relations. Many of the computational approaches we use later often rely on grouping set elements according to certain properties. This grouping is based on a fundamental mathematical concept known as a *relation*. Relations may exist among elements within a single set, as well as elements from distinct sets.

Given two, not necessarily distinct, sets A and B , a *binary relation* R from A to B is a subset of their Cartesian product $A \times B$. Higher-order, n -ary relations are subsets of n -tuples from the Cartesian product of n sets. In this work, however, we are primarily concerned with binary relations. An important type of binary relation is one that is defined from a set onto itself. Such a relation $R \subseteq A \times A$ is an *equivalence relation* on A if for any members a , b and c of A the following three conditions are satisfied:

1. $\langle a, a \rangle \in R$ *reflexivity*
2. if $\langle a, b \rangle \in R$ then also $\langle b, a \rangle \in R$ *symmetry*
3. if $\langle a, b \rangle \in R$ and $\langle b, c \rangle \in R$ then $\langle a, c \rangle \in R$ *transitivity*

An equivalence relation partitions a set into subsets. Formally, subsets A_1, \dots, A_k form a *partition* of set A if the following two conditions are satisfied:

$$(i \neq j) \Rightarrow A_i \cap A_j = \emptyset, \quad 1 \leq i, j \leq k$$

$$\bigcup_{1 \leq i \leq k} A_i = A$$

The subsets A_i induced by an equivalence relation are called its *equivalence classes*. A relation which is reflexive and symmetric, but not transitive, is a *compatibility relation*. If the symmetry condition in the above definition is replaced with *anti-symmetry*— $\langle a, b \rangle \in R$ and $\langle b, a \rangle \in R$ implies $a = b$ —then the relation defines a *partial order* on the set. When two elements a and b satisfy the partial order relation we write $a \leq b$. If all pairs in the partial order are comparable, then the order is *total*.

Functions. A *function* f from set A to set B is a mapping, denoted $f : A \rightarrow B$, that associates with each element from A exactly one element from B ; it is a special binary relation. The sets A and B are called, respectively, the *domain* and *co-domain* of f . The *range* of f is the subset of B that f can assume. Unlike an un-restricted relation, each element from the domain of a function appears in exactly one pair relating it to a range element. When the range of a function is identical with its co-domain, it is called an *onto* mapping.

Viewing functions as sets of ordered pairs, we can manipulate them to define new types of relations. Let $g : A \rightarrow B$ and $f : B \rightarrow C$ be functions such that $\text{range}(g) \subseteq B$. The *composition* of g by f is then defined to be the set of ordered pairs:

$$f \circ g =_{\text{df}} \{ \langle a, f(g(b)) \rangle \mid a \in A, b = g(a) \} \quad (2.1)$$

An alternative notation for the composition of g by f is $f(g(\cdot))$. Similarly, the *restriction* of $f : A \rightarrow B$ to subset C of its domain is defined as the set of pairs:

$$f_C =_{\text{df}} \{ \langle a, f(a) \rangle \mid a \in C, C \subseteq A \} \quad (2.2)$$

Functions can be also used to model set membership. For a subset B of set A such a function is defined as a mapping $f : A \rightarrow \{0, 1\}$ such which $f(a) = 1$ if $a \in B$, and $f(a) = 0$ otherwise. Functions of this type are commonly referred to as *characteristic functions* of the corresponding set.

2.2 Boolean Algebra

Boolean algebra and its properties is the primary mathematical model underlying logic synthesis algorithms. In this section, we give a brief overview of the algebraic structure of Boolean algebra. For more detailed treatments, the reader is referred to the extensive literature on this subject including [23, 39, 93].

Boolean algebra is an algebraic structure $(B, +, \cdot)$ in which B is a set, called the *carrier* of the algebra, symbols $+$ (*join* or OR) and \cdot (*meet* or AND) are binary operations, and elements of B satisfy the following list of postulates:¹

1. For all $a, b \in B$ the following equalities hold

$$a + b = b + a \qquad a \cdot b = b \cdot a \qquad \text{commutativity}$$

2. For all $a, b, c \in B$ the following equalities hold

$$a + (b \cdot c) = (a + b) \cdot (a + c) \qquad a \cdot (b + c) = (a \cdot b) + (a \cdot c) \qquad \text{distributivity}$$

3. For the $+$ operation there exists an identity element $0 \in B$, and for the \cdot operation there exists an identity element $1 \in B$ such that

$$a + 0 = a \qquad a \cdot 1 = a \qquad \text{identity}$$

4. For every element $a \in B$ there exists a complement element $\bar{a} \in B$ such that

$$a + \bar{a} = 1 \qquad a \cdot \bar{a} = 0 \qquad \text{existence of complement}$$

Complementation can also be viewed as a unary operation that returns the *complement* or *inverse* of the element it is applied to.

It can be shown that a Boolean algebra induces a partial order relation \leq on the elements of its carrier B . Although the meaning of this relation may vary depending on the nature of the constructed Boolean algebra, it always remains true that for all $a, b \in B$ the following holds:

1. This is one of many possible axiomatic definitions of Boolean algebra. This particular set of postulates is due to Huntington [62].

$$a \leq b \Leftrightarrow a\bar{b} = 0 \Leftrightarrow \bar{a} + b = 1 \quad (2.3)$$

Stone showed in [113] that Boolean algebras have the same structure as the set algebra over a power set 2^A for some finite set A . In this algebra, union \cup and intersection \cap are direct counterparts to the join and meet operations, whereas the empty set \emptyset and A then correspond to the 0 and 1 elements of B . Without loss of generality we can therefore analyze complex Boolean algebra concepts by visualizing them as set operations. The ability to reason in terms of sets becomes very useful when working with Boolean functions which are described next.

2.3 Boolean Functions

Fundamentally, logic synthesis can be viewed as the guided manipulation of sets of Boolean functions to achieve desirable forms that are suitable for circuit realizations. An n -variable Boolean function $f(x_1, \dots, x_n)$ is defined as the mapping [23, 93]

$$f : B^n \rightarrow B$$

where each x_i is an input variable taking values from B . The input variables can also be viewed as a vector $\mathbf{x} =_{\text{df}} (x_1, \dots, x_n)$ whose domain is B^n , and are called the *support* of f . For digital circuit applications, the carrier is commonly restricted to be the 2-element set $B = \{0, 1\}$ where 0 and 1 correspond to the low and high logic levels, respectively. Functions of this algebra are also referred to as *switching functions*.¹ A collection of these functions may be organized in an l -tuple, thereby forming a vector of functions:

$$\mathbf{f} =_{\text{df}} (f_1, \dots, f_l)$$

The vector \mathbf{f} may also be treated as a function $\mathbf{f} : \{0, 1\}^n \rightarrow \{0, 1\}^l$ with a *multi-valued* codomain. In this work, we refer to such a function as a *multiple-output function*, and treat each of its components as a *single-output function* f_i .

An element m in the domain B^n is an ordered string of n 0s and 1s, and is called a *minterm*, or *point*; B^n contains a total of 2^n minterms. The *weight* of a minterm is the numbers of 1s in it. A k -subset of variables from the support of f specifies a *subdomain* B^k . A point in such a subdomain can be viewed as a minterm on the subset of k variables; it is also a *cube* with respect to the entire B^n domain. If a variable x_i is not in the support of f , or if assignments to x_i do not affect the value of f , then f is *vacuous* in x_i . Recalling that switching functions are mappings from B^n

1. Rudeanu terms functions over 2-element carriers *truth functions* [93, p. 29].

to B , the number of such functions (possibly vacuous in some variables) is 2^{2^n} ; the number of functions increases to $2^{l \cdot 2^n}$ in the multiple-output case.

Values of a *completely specified* multiple-output function partition the function's domain into 2^l subsets, some possibly empty. For a single-output function the partition consists of the *on-set* and *off-set* defined as:

$$\begin{aligned} \text{on-set}(f) &=_{\text{df}} \{m \in B^n \mid f(m) = 1\} \\ \text{off-set}(f) &=_{\text{df}} \{m \in B^n \mid f(m) = 0\} \end{aligned}$$

When the off-set is empty, the function is the constant $\mathbf{1}$; analogously, an empty on-set implies that the function is the constant $\mathbf{0}$. Observe that a single-output Boolean function is nothing but a characteristic function of its on-set.

2.3.1 Operations on Boolean functions

It is well known that the set of n -variable Boolean functions, as well as appropriately selected subsets, form Boolean algebras (see e.g. [23, 54]). In particular, equation (2.3), which is stated in terms of Boolean variables, is also applicable to Boolean functions and defines a partial order on them. Given functions $f(\mathbf{x})$ and $g(\mathbf{x})$, $f(\mathbf{x}) \leq g(\mathbf{x})$ indicates that $f(\mathbf{x})$ precedes $g(\mathbf{x})$ in the partial order. Furthermore, the “less than or equal” relation between $f(\mathbf{x})$ and $g(\mathbf{x})$ can be expressed by the following two equivalent forms:

$$f(\mathbf{x}) \leq g(\mathbf{x}) \Leftrightarrow f(\mathbf{x}) \cdot \overline{g(\mathbf{x})} = \mathbf{0} \Leftrightarrow \overline{f(\mathbf{x})} + g(\mathbf{x}) = \mathbf{1} \quad (2.4)$$

Recalling that the algebra of sets is also a Boolean algebra, it is often useful to view operations on Boolean functions in terms of corresponding set operations. Specifically, the meet (\cdot) and join ($+$) of functions f and g between can be seen, respectively, as the intersection and union of their on-sets. For each function f , its complement \bar{f} is a function whose on-set is the off-set of f .

Aside from the basic Boolean algebra operations on n -variable functions, there are other operations that are instrumental to many synthesis algorithms. One such operation, called the *cofactor* [15],¹ restricts variable x_i of function f to a constant and is usually denoted as:

$$\begin{aligned} f_{\bar{x}_i}(x_1, \dots, x_i, \dots, x_n) &=_{\text{df}} f(x_1, \dots, 0, \dots, x_n) && \text{negative cofactor} \\ f_{x_i}(x_1, \dots, x_i, \dots, x_n) &=_{\text{df}} f(x_1, \dots, 1, \dots, x_n) && \text{positive cofactor} \end{aligned}$$

1. Over the years various names were used in reference to the cofactor; they include *Boolean quotient* [23, p. 53], *ratio* [49], *x_i -residue* [81, p. 169], and *restriction* [24].

Table 2.1: Some properties of the cofactor operation (c is a cube)

Distributivity	$(f + g)_c = f_c + g_c$	$(f \cdot g)_c = f_c \cdot g_c$
Approximation	$\bar{c} + \bar{f} = \bar{c} + \bar{f}_c$	$c \cdot f = c \cdot f_c$
Complementation	$(\bar{f})_c = \bar{f}_c$	
Containment	$f \leq g \Rightarrow f_c \leq g_c$	

Whenever $f_{\bar{x}_i} \leq f_{x_i}$ we say that f is *monotone increasing* in x_i ; conversely, if $f_{x_i} \leq f_{\bar{x}_i}$ then f is *monotone decreasing* in x_i . f is *vacuous* in x_i if $f_{\bar{x}_i} = f_{x_i}$. It is important to note that these cofactors are themselves functions whose support does not include the variable x_i . To emphasize this point, we may sometimes denote the explicit dependence of a cofactor on the remaining variables by $f_{x_i} = g(\mathbf{x} \setminus x_i)$. Function monotone in all of its variables is *unate*.

The cofactor operation can be applied to more than one variable at once. Applying it with respect to a cube c yields the cofactor f_c which is obtained by setting appropriate function variables to either 0 or 1 as specified by c . In general, cofactors of n -variable functions are related as given in Table 2.1. The notion of a cofactor also enables variable abstraction according to the two definitions below:

$$\exists x_i f \stackrel{\text{df}}{=} f_{\bar{x}_i} + f_{x_i} \quad \text{existential abstraction}$$

$$\forall x_i f \stackrel{\text{df}}{=} f_{\bar{x}_i} \cdot f_{x_i} \quad \text{universal abstraction}$$

The result of existentially abstracting x_i from f is the smallest function, in the partial order, that contains f and is independent of x_i . Analogously, the result of universally abstracting x_i from f is the largest function that is contained in f and is independent of x_i . Such single-variable abstraction extends in a straightforward manner to the abstraction of variable vectors. Some of the useful properties for these two operations are given in Table 2.2.

Table 2.2: Some properties of variable abstraction operations

Complementation	$\exists x_i \bar{f} = \bar{\forall x_i f}$	$\forall x_i \bar{f} = \bar{\exists x_i f}$
Commutativity	$\exists x_i \exists x_j f = \exists x_j \exists x_i f$	$\forall x_i \forall x_j f = \forall x_j \forall x_i f$
Distributivity	$\exists x_i (f + g) = \exists x_i f + \exists x_i g$	$\forall x_i (f \cdot g) = \forall x_i f \cdot \forall x_i g$
Non-commutativity	$\forall x_i \exists x_j f \neq \exists x_j \forall x_i f$	
Non-distributivity	$\exists x_i (f \cdot g) \neq \exists x_i f \cdot \exists x_i g$	$\forall x_i (f + g) \neq \forall x_i f + \forall x_i g$
Containment	$f \leq \exists x_i f$	$\forall x_i f \leq f$

One of the recurring themes in this dissertation is the identification and judicious use of a function's *semantic structure*. Informally speaking, semantic structure refers to any functional property that holds irrespective of any particular representation. An example of such properties is the afore-mentioned unate functions. Other examples that we will discuss later are various types of symmetry. In general, we view semantic structure as a reflection of the inherent relations among a function's cofactors.

2.3.2 Expansions of Boolean functions

Expansions often serve as an essential component in the representation and effective manipulation of Boolean functions. They allow functions to be expressed in a variety of forms to suite particular computations. Alternatively, they may simply be used to obtain compact, or canonical, representations of functions. Two important expansions, with applications in synthesis, are described below.

Shannon expansion. Using the cofactor notation, the *Shannon expansion* of function a f is:

$$f = \bar{x}_i f_{\bar{x}_i} + x_i f_{x_i}$$

This expansion dates back to the work of Boole [10, Chapter V], and is also known as Boole's expansion theorem. Its formal proof by induction can be found in [54, p. 98]. In design automation, it is used extensively as a basic step when working with Boolean functions. For any function f , iterative application of the expansion with respect to variables from f results in the function's *Shannon expansion tree*.

Orthonormal expansion. The Shannon expansion is a restricted form of a more general expansion, known as an orthonormal expansion. This expansion is defined over a set of k n -variable functions $\{t_0(\mathbf{x}), \dots, t_k(\mathbf{x})\}$ whose on-sets partition their domain B^n such that:

$$(i \neq j) \Rightarrow t_i(\mathbf{x}) \cdot t_j(\mathbf{x}) = \mathbf{0}, \quad 0 \leq i, j \leq k \quad \text{orthogonality}$$

$$\sum_{0 \leq i \leq k} t_i(\mathbf{x}) = \mathbf{1} \quad \text{normality}$$

If a set of functions meets the above two conditions, then we say that it forms an *orthonormal basis* in B^n . Given a Boolean function $f(\mathbf{x})$ and an orthonormal basis $\{t_0(\mathbf{x}), \dots, t_k(\mathbf{x})\}$, the *orthonormal expansion* of f is the expression

$$f(\mathbf{x}) = \sum_{0 \leq i \leq k} t_i(\mathbf{x}) \cdot f_i(\mathbf{x}) \quad (2.5)$$

As shown in [23, p. 49], for the above identity to hold it must be that all subfunctions f_i satisfy:

$$t_i(\mathbf{x}) \cdot f(\mathbf{x}) \leq f_i(\mathbf{x}) \leq \overline{t_i(\mathbf{x})} + f(\mathbf{x}) \quad (2.6)$$

The orthonormal expansion over a given basis is, therefore, not unique. For example, using the orthonormal basis $\{x_i, \bar{x}_i\}$ yields the expansion

$$f(\mathbf{x}) = \bar{x}_i \cdot f_0(\mathbf{x}) + x_i \cdot f_1(\mathbf{x})$$

where $\bar{x}_i \cdot f(\mathbf{x}) \leq f_0(\mathbf{x}) \leq x_i + f(\mathbf{x})$ and $x_i \cdot f(\mathbf{x}) \leq f_1(\mathbf{x}) \leq \bar{x}_i + f(\mathbf{x})$.

For synthesis applications, it is useful to apply a slightly modified version of (2.5). Let $\mathbf{x} = (\mathbf{y}, \mathbf{z})$, i.e. partition the input vector into two sub-vectors. Using an orthonormal basis $\{t_0(\mathbf{y}), \dots, t_k(\mathbf{y})\}$ whose components are vacuous in \mathbf{z} , the expansion of $f(\mathbf{x})$ is now given by the expression:

$$f(\mathbf{y}, \mathbf{z}) = \sum_{0 \leq i \leq k} t_i(\mathbf{y}) \cdot f_i(\mathbf{y}, \mathbf{z}) \quad (2.7)$$

where

$$t_i(\mathbf{y}) \cdot f(\mathbf{y}, \mathbf{z}) \leq f_i(\mathbf{y}, \mathbf{z}) \leq \overline{t_i(\mathbf{y})} + f(\mathbf{y}, \mathbf{z}) \quad (2.8)$$

While the subfunctions $f_i(\mathbf{y}, \mathbf{z})$ can be chosen arbitrarily as long as they satisfy these bounds, in the context of synthesis it may be desirable to make them vacuous in the basis variables \mathbf{y} . With such a choice, these subfunctions become unique as shown by the following lemma:

Lemma 2.1 *For a given function $f(\mathbf{y}, \mathbf{z})$ and a basis function $t_i(\mathbf{y})$, the subfunction $f_i(\mathbf{y}, \mathbf{z})$ is vacuous in \mathbf{y} and satisfies (2.8) if and only if it is equal to the cofactor $f_m(\mathbf{y}, \mathbf{z})$, where m is a minterm on \mathbf{y} such that $t_i(m) = 1$.*

Proof of this result is given in Appendix A. An immediate consequence of the lemma is that the orthonormal expansion becomes unique and takes the form:

$$f(\mathbf{y}, \mathbf{z}) = \sum_{\substack{0 \leq i \leq k \\ t_i(m) = 1}} t_i(\mathbf{y}) \cdot f_m(\mathbf{y}, \mathbf{z})$$

There are many other expansion forms described in the literature. For example, expansions can be defined over a Boolean ring, replacing the $+$ operation with \oplus [39].¹ However, the orthonormal expansion is sufficient for our purposes, as it is general enough to enable the derivation of any combinational circuit. In this sense, it subsumes all other expansions.

1. This result is made possible due to the fact that Boolean algebra can be defined over a *Boolean ring* letting join and meet to be \oplus and AND, respectively [112,113].

2.4 Partial Specification of Boolean Functions

In addition to completely specified functions, it is necessary in synthesis applications to model and reason about partially specified functions. Such functions arise naturally in this context as they represent the flexibility available to synthesis algorithms for optimizing circuit implementations. Partially specified functions have many representations that differ in their expressiveness, compactness, and the ease (or difficulty) of manipulating them.

The most expressive, but least compact, representation is an explicit listing (a set) of all the completely specified functions that are possible choices for a given partially specified function. For example the set $F(x_1, x_2) = \{x_1, \bar{x}_1 + x_2\}$ can be used to indicate two possible choices for a two-input single-output partially specified function. Similarly, the function set $G(x_1, x_2) = \{\langle x_1, \bar{x}_1 x_2 \rangle, \langle x_1 + \bar{x}_2, \bar{x}_1 \rangle, \langle x_1 + \bar{x}_2, \bar{x}_1 + x_2 \rangle\}$ indicates three choices for a two-input two-output function. Such explicit representations are not very useful in practice, however, because of their potentially exponential size in the number of inputs and outputs.

One possible approach to deal with this complexity while providing reasonable expressiveness is to drop the many-to-one restriction imposed on the mapping of functions, and to represent a partially specified function by a many-to-many relation R from B^n to B^l : $R \subseteq B^n \times B^l$. In the field of synthesis and verification such relations are known as *Boolean relations* [22].¹ Although implied by the notion of function mapping, Boolean relations must be *well-formed* in that each point from B^n must be in a relation with at least one point from B^l . For computational purposes, a Boolean relations is usually modeled by an associated characteristic function $f : B^n \times B^l \rightarrow B$ such that $f(m, m') = 1$ if and only if $\langle m, m' \rangle \in R$.

To illustrate Boolean relations, consider the characteristic function

$$f(x_1, x_2, z_1, z_2) = \bar{x}_1 \bar{x}_2 \bar{z}_1 \bar{z}_2 + \bar{x}_1 \bar{x}_2 z_1 z_2 + \bar{x}_1 x_2 \bar{z}_1 z_2 + x_1 x_2 z_1 + x_1 z_1 \bar{z}_2$$

for a Boolean relation that represents a two-input two-output partially specified function (the function inputs are $\langle x_1, x_2 \rangle$ and its outputs are $\langle z_1, z_2 \rangle$). This relation² is an implicit representation of a set of *compatible* (completely specified) functions that are suitable choices for implementing the

1. In classical functional analysis functions whose behavior is not unique in a point of their domain are known as *multiple-valued functions* [68, Part II]. In the field of design automation this term, however, is reserved for the functions whose range is defined on multiple-valued logics.

2. When clear from context we will refer to a relation and its characteristic function synonymously.

partially specified function. Compatible functions for the above Boolean relation are enumerated in the table below:

Boolean relation		Compatible functions				
$\langle x_1, x_2 \rangle$	$\langle z_1, z_2 \rangle$	$\langle x_1, x_2 \rangle$	$f(x_1, x_2)$			
00	00, 11	00	00	00	11	11
01	01	01	01	01	01	01
10	10	10	10	10	10	10
11	10, 11	11	10	11	10	11

The table reveals a set which consists of four compatible functions:

$$\{ \langle x_1, \bar{x}_1 x_2 \rangle, \langle x_1, x_2 \rangle, \langle x_1 + \bar{x}_2, \bar{x}_1 \rangle, \langle x_1 + \bar{x}_2, \bar{x}_1 + x_2 \rangle \}$$

The compactness of the Boolean relation representation comes at the price of reduced expressiveness: not all function sets can be captured by Boolean relations. For example, the three-element function set obtained by removing $\langle x_1, x_2 \rangle$ from the above set cannot be represented by a Boolean relation.

When used to model single-output partially specified functions, Boolean relations turn out to be equivalent to two other commonly-used representations: don't-care specifications, and Boolean function intervals. For example, the Boolean relation $\bar{x}_1 + \bar{x}_2 \bar{z} + x_2 z$ represents four possible completely specified functions as shown by the truth table below:

Boolean relation		Compatible functions				
$\langle x_1, x_2 \rangle$	z	$\langle x_1, x_2 \rangle$	$f(x_1, x_2)$			
00	0, 1	00	0	0	1	1
01	0, 1	01	0	1	0	1
10	0	10	0	0	0	0
11	1	11	1	1	1	1

These same four functions can be equivalently captured by a don't-care representation in which:

- rows in the above truth table where the output can be either 0 or 1 (the first two) are used to define a don't-care set¹ $z_d = \bar{x}_1$,
- rows in which the output must be 1 are used to define an on-set $z_{on} = x_1 x_2$, and
- rows in which the output must be 0 are used to define an off-set $z_{off} = x_1 \bar{x}_2$.

1. Again, we are representing this set by its characteristic function and liberally speaking of this function as being the actual set.

These three sets form a partition of the input domain, and any two of them suffice as the third can be easily derived (e.g. $z_{off} = \overline{(z_{on} + z_d)}$). In addition, a care set z_c , rather than a don't-care set, can be specified where $z_c = \bar{z}_d$. In any case, the Boolean relation representing the partial specification of a single-output function can be represented by a pair of completely specified functions defined on the domain of the input variables.

A second equivalent representation that also requires a pair of completely specified functions to model a single-output partially specified function is an interval in the function space of n -variable Boolean functions. This is the representation we adopt in this work, and it is further developed in the following section. While they have the same expressiveness and similar computational attributes (for single-output functions) to Boolean relations and don't-care specifications, Boolean function intervals arise naturally while solving sets of Boolean constraints [23]. As we demonstrate throughout this work, many optimizations that are performed during synthesis can be cast in this fashion and their solutions emerge naturally as Boolean function intervals.

2.5 Boolean Function Intervals

A function *interval*¹ is a set of completely specified Boolean functions defined as

$$[l(\mathbf{x}), u(\mathbf{x})] =_{\text{df}} \{f(\mathbf{x}) \mid l(\mathbf{x}) \leq f(\mathbf{x}) \leq u(\mathbf{x})\} \quad (2.9)$$

The functions $l(\mathbf{x})$ and $u(\mathbf{x})$ represent two distinguished members of the interval, namely its *lower* and *upper bounds*, respectively. An interval is non-empty if and only if $l(\mathbf{x}) \leq u(\mathbf{x})$ is satisfied. Examples of function intervals include the subfunctions arising in orthonormal expansion. For instance, the subfunction $f_i(\mathbf{y}, \mathbf{z})$ in (2.8) is a non-unique partially specified function that can be expressed as the function interval $[t_i(\mathbf{y}) \cdot f(\mathbf{y}, \mathbf{z}), \overline{t_i(\mathbf{y})} + f(\mathbf{y}, \mathbf{z})]$.

Function intervals cannot represent arbitrary sets of completely specified functions. However, they are particularly useful in synthesis since they frequently represent the entire set of completely specified functions that satisfy a given constraint. As such, it is advantageous to develop some facility, an *interval algebra*, for symbolically manipulating function intervals.

For an interval $[l, u]$ and its member function f the following properties are readily established from the partial order relation [95]:

1. In [88] function intervals are termed *permissible functions*.

$$\begin{array}{ll}
l \leq f & f \leq u \\
l \cdot f = l & u + f = u \\
l \cdot \bar{f} = 0 & u + \bar{f} = 1 \\
\bar{f} \leq \bar{l} & \bar{u} \leq \bar{f} \\
\bar{l} + \bar{f} = \bar{l} & \bar{u} \cdot \bar{f} = \bar{u} \\
\bar{l} + f = 1 & \bar{u} \cdot f = 0
\end{array} \tag{2.10}$$

These properties provide an insight into the computational nature of the interval bounds. In particular, they yield the following expressions of the bounds:

$$l = \prod_{f \in [l, u]} f \quad \text{lower-bound}$$

$$u = \sum_{f \in [l, u]} f \quad \text{upper-bound}$$

These basic properties allow us to derive useful computational forms for manipulating function intervals.

Being *sets of Boolean functions*, function intervals can be combined using both Boolean operators (OR, AND, inverse, etc.) as well as set operators (union, intersection). Given intervals $F = [l_1(\mathbf{x}), u_1(\mathbf{x})]$ and $G = [l_2(\mathbf{x}), u_2(\mathbf{x})]$, and a Boolean operator $*$, we define $F * G$ to be the set

$$F * G =_{\text{df}} \{f * g \mid f \in F, g \in G\}$$

As the following lemma shows, these sets turn out to be intervals themselves and can be efficiently computed by operating solely on the bounds of the intervals.

Lemma 2.2 [95] $[l_1(\mathbf{x}), u_1(\mathbf{x})] + [l_2(\mathbf{x}), u_2(\mathbf{x})] = [l_1(\mathbf{x}) + l_2(\mathbf{x}), u_1(\mathbf{x}) + u_2(\mathbf{x})]$

$$[l_1(\mathbf{x}), u_1(\mathbf{x})] \cdot [l_2(\mathbf{x}), u_2(\mathbf{x})] = [l_1(\mathbf{x}) \cdot l_2(\mathbf{x}), u_1(\mathbf{x}) \cdot u_2(\mathbf{x})]$$

$$\overline{[l(\mathbf{x}), u(\mathbf{x})]} = [\overline{u(\mathbf{x})}, \overline{l(\mathbf{x})}]$$

It is interesting to point out that although the functions in an interval form a Boolean algebra, the intervals themselves do not form a Boolean algebra. This is because the interval complement operation does not obey the complement laws of Boolean algebra.

Another useful Boolean operation is the cofactor of an interval. Let $H(\mathbf{x} \setminus x_i) = F_{x_i}$ be the cofactor of interval $F(\mathbf{x}) = [l(\mathbf{x}), u(\mathbf{x})]$ with respect to x_i which is defined as follows:

$$H(\mathbf{x} \setminus x_i) = [l, u]_{x_i} =_{\text{df}} \{f_{x_i} \mid l \leq f \leq u\}$$

Interval cofactor can be efficiently computed according to the following lemma:

Lemma 2.3 $[l(\mathbf{x}), u(\mathbf{x})]_{x_i} = [l(\mathbf{x})_{x_i}, u(\mathbf{x})_{x_i}]$

This lemma, whose proof is given in Appendix A, basically states that cofactoring a function interval yields another function interval in the $(n - 1)$ -dimensional Boolean function domain obtained by setting $x_i = 1$. Computationally, this implies that we can cofactor an interval by just cofactoring its bounds.

Viewed as sets, function intervals can be combined with set operators. Specifically, the *interval intersection* $F \cap G$ is the set composed of those functions that belong to both F and G . As the following lemma shows, interval intersection can be computed by performing Boolean operations on interval bounds.

Lemma 2.4 $[l_1(\mathbf{x}), u_1(\mathbf{x})] \cap [l_2(\mathbf{x}), u_2(\mathbf{x})] = [l_1(\mathbf{x}) + l_2(\mathbf{x}), u_1(\mathbf{x}) \cdot u_2(\mathbf{x})]$

The union of two intervals, i.e. the set that consists of functions that belong to either interval, is not necessarily an interval. We define the *interval union* $F \cup G$ to be the smallest interval containing the union of the two intervals. The following lemma gives a computational form for interval union in terms of Boolean operations on their bounds.

Lemma 2.5 $[l_1(\mathbf{x}), u_1(\mathbf{x})] \cup [l_2(\mathbf{x}), u_2(\mathbf{x})] = [l_1(\mathbf{x}) \cdot l_2(\mathbf{x}), u_1(\mathbf{x}) + u_2(\mathbf{x})]$

It is sometimes useful to extract the subset of a function interval $[l(\mathbf{x}), u(\mathbf{x})]$ containing only those functions that are vacuous in x_i . Let $\nabla x_i[l(\mathbf{x}), u(\mathbf{x})]$ be defined as follows:

$$\nabla x_i[l(\mathbf{x}), u(\mathbf{x})] =_{\text{df}} [l(\mathbf{x}), u(\mathbf{x})]_{\bar{x}_i} \cap [l(\mathbf{x}), u(\mathbf{x})]_{x_i} \quad (2.11)$$

Recalling the definition of interval cofactors from Lemma 2.3, note that the function interval defined above is in the $(n - 1)$ -dimensional Boolean function space obtained by eliminating x_i from the variable domain.

Theorem 2.6 *The function interval $\nabla x_i[l(\mathbf{x}), u(\mathbf{x})]$, defined in (2.11), represents the set of all functions $f(\mathbf{x}) \in [l(\mathbf{x}), u(\mathbf{x})]$ that are vacuous in x_i . In other words, for all functions $f(\mathbf{x})$ that are vacuous in x_i it must be that:*

$$f(\mathbf{x}) \in \nabla x_i[l(\mathbf{x}), u(\mathbf{x})] \Leftrightarrow f(\mathbf{x}) \in [l(\mathbf{x}), u(\mathbf{x})]$$

An immediate consequence of this theorem, which is proved in Appendix A, is that the interval $[l(\mathbf{x}), u(\mathbf{x})]$ contains functions that are vacuous in x_i if and only if $\nabla x_i[l(\mathbf{x}), u(\mathbf{x})]$ is non-empty. It is also important to note that the interval $\nabla x_i[l(\mathbf{x}), u(\mathbf{x})]$ is constructed in the $(n - 1)$ -dimensional function domain in which variable x_i is absent, i.e. it can be denoted as the function

as a Efficient computational forms for (2.11) can be readily obtained by applying Lemma 2.3 and Lemma 2.4:

$$\begin{aligned}
 \nabla x_i[l(\mathbf{x}), u(\mathbf{x})] &= [l_{\bar{x}_i}, u_{\bar{x}_i}] \cap [l_{x_i}, u_{x_i}] \\
 &= [l_{\bar{x}_i} + l_{x_i}, u_{\bar{x}_i} \cdot u_{x_i}] \\
 &= [\forall x_i l, \exists x_i u]
 \end{aligned} \tag{2.12}$$

Finally, these results for a single variable extend by induction to an arbitrary number of variables (see Corollary A.5 on page 134).

As mentioned in the previous section, function intervals and don't care specifications are equivalent in the sense that they denote the same set of completely specified functions that represent the possible choices of a partially specified function. Let F be a single-output partially specified function given by the interval $[l, u]$, and let f, r, c , and d denote, respectively, the characteristic functions of its on-, off-, care-, and don't-care sets. These five functions are related according to:

$$\begin{aligned}
 f &= l && \text{on-set is same as lower bound} \\
 r &= \bar{u} && \text{off-set is complement of upper bound} \\
 d &= u\bar{l} && \text{don't-care set is difference between bounds} \\
 c &= l + \bar{u} && \text{care set is complement of don't-care set}
 \end{aligned}$$

A situation that arises often during synthesis is the need to find the set of functions that are identical to a given function in a subset of that function's domain. Specifically, given the function $f(\mathbf{x})$ and a care set $c(\mathbf{x})$, find all functions $g(\mathbf{x})$ such that $g(\mathbf{x}) = f(\mathbf{x})$ when $c(\mathbf{x}) = 1$; when $c(\mathbf{x}) = 0$, the functions $g(\mathbf{x})$ are unspecified. The solution set turns out to be the function interval given in the following theorem:

Theorem 2.7 *Let $f(\mathbf{x})$, $c(\mathbf{x})$, and $g(\mathbf{x})$ be functions as described above. Any function $g(\mathbf{x})$ such that*

$$g(\mathbf{x}) \in [f(\mathbf{x}) \cdot c(\mathbf{x}), f(\mathbf{x}) + \overline{c(\mathbf{x})}] \tag{2.13}$$

satisfies the stated requirements.

The formal derivation of this result is given in Appendix A, and illustrations of its use in a variety of network optimization scenarios are given in Appendix B. Indeed, it can be argued that most synthesis tasks can be reduced to the derivation of a care set that models the “fixed” portion of the network being synthesized, followed by application of (2.13) to determine the “flexibility” that exists in designing the “variable” portion of the network. In some cases, the desired result requires fur-

ther manipulation of the interval in (2.13), e.g. to extract a sub-interval whose functions are vacuous in certain variables.

2.6 Binary-Decision Diagrams

Representing Boolean functions with a binary-decision diagram (BDD) was originally proposed by Lee [73] and Akers [1]. However, it was only with the work of Bryant [24] in 1986 that BDDs became widely used. His work brought out the canonical nature of BDDs in representing Boolean functions. The work also introduced effective algorithms to manipulate them. Since then the use of BDDs has entered virtually every area of synthesis and verification.

A binary-decision diagram represents a Boolean function as a rooted directed acyclic graph (DAG). It has two types of vertices: *terminal* and *non-terminal*. Terminal vertices are leaves in the graph corresponding to the 0 and 1 values of a function; they have no outgoing edges. All other nodes in the BDD are non-terminal, and they represent a Shannon expansion about some variable x_i . Each non-terminal node can be viewed as a root node of some non-constant function f , and it has a $lo(x_i)$ and a $hi(x_i)$ child. If v is a non-terminal node with index i , then its f_v function is

$$f_v = \bar{x}_i \cdot f_{lo(v)} + x_i \cdot f_{hi(v)}$$

To represent a function with a BDD a total order is imposed on its variables. Node variables on each of the root-to-terminal paths obey this order. A reduced BDD is constructed using two reduction rules:

1. Nodes whose two edges point to the same child are deleted.
2. Isomorphic subgraphs are shared.

BDDs are unique for a given variable ordering and hence provide canonical forms for the representation of Boolean functions. This canonicity makes them well suited for symbolic manipulation. They are also useful of representing large combinatorial sets. A comprehensive treatment of BDDs can be found in [26].

Variable ordering is known to have a dramatic effect on the size of a BDD. Unfortunately there is no known method which can quickly detect an optimal variable ordering. Most of the variable ordering techniques rely on heuristics. The earlier work on this subject couples variable ordering techniques with topological information from the circuit for which the BDD has been constructed [47, 78, 84]. A later heuristic, based on variable sifting, was developed by Rudell [94]. In the sifting algorithm, each variable is moved up and down to greedily find its best location. Since its

introduction the algorithm has been integrated into many BDD packages and applications in various flavors [59, 65, 83, 90].

The re-ordering techniques of any typical package are applied once by an explicit function call, or they can be invoked *dynamically* [94] by an implicit function call triggered by some memory consumption criteria. In many applications such asynchronous re-ordering has a profound effect on the resources consumed during the manipulation of BDDs. However, the dynamic re-ordering of variables still remains an expensive operation, and may take a significant part of a computation. Applications which are aware of variable orders implied by the intrinsic nature of the problem are therefore the most desired approach to reducing BDD sizes. For some functions however, the size of a BDD may be exponential in the number inputs regardless of the variable ordering; an n -bit multiplier is an example of such functions [25].

Once the BDD for a function is constructed many operations on it have good characteristics. For example, taking function complement, or checking if the function is satisfiable can be done in constant time. In general the space and time requirements for the binary operations are proportional to the number of nodes in the two composed BDDs. Deciding if two functions are equivalent requires a graph isomorphism check, whose time complexity for the labelled DAG is linear in the number of nodes. The check is even more efficient when the two given functions depend on the same set of variables. If two such functions are equivalent, a typical BDD package implementation would ensure that they reside in the same memory space. This is achieved by virtue of a *unique table* [13], which guarantees that at any time there are no isomorphic subgraphs. Thus the equivalence check takes constant time. The unique table also allows a single *multi-rooted* DAG to represent all created functions. To reduce memory consumption, modern BDD packages also attempt to share not only isomorphic subgraphs, but subgraphs of their complement functions as well; thus subgraphs for f and \bar{f} are identical.

2.7 Boolean Networks

The Boolean network [19] is the basic data structure used by multi-level synthesis algorithms. This section begins with the definition of a Boolean network. A Boolean network is a directed acyclic graph whose connections model the dependence between a set of Boolean functions. Its *primary inputs* are nodes identifying externally controllable signals. Similarly, its *primary outputs* are nodes identifying externally observable signals. All other nodes in the network are *internal* or *constant*, and are collectively called *logic* nodes. For every node i in the network (except primary

inputs) there is an associated representation of a Boolean function f_i , and a Boolean variable y_i . It is also convenient to have variables x_i and z_i as synonyms to the variables of primary inputs and primary outputs, respectively. Functions of the output nodes are single-variable functions of some other node in the network, i.e. $f_i(y_j) = y_j$. We also impose a convention that variables of primary output nodes must not be used in any function. If variable y_i is in the support of some node function f_i then there must be a node in the network corresponding to y_i . Nodes in the Boolean network can also have associated don't care functions d_i expressed in terms of don't cares given for the network outputs, and other implicit don't cares derived from the network structure.

There is an oriented connection from node i to node j , denoted (i, j) , if and only if f_j is not vacuous in y_i . A node i is a *fan-in* of node j if there is a connection (i, j) ; similarly, a node i is a *fanout* of node j if there is a connection (j, i) . The (i, j) connections in the network define a partial order on the network nodes such that (i, j) implies $i \leq j$. A total order induced by this partial order is called *topological*. With each node in the network we may associate a *depth*, which corresponds to the maximum number of connections leading to the node from a primary input node. The *network depth* is defined as a maximum depth over all fan-ins to output nodes. A *wire* corresponds to an edge in the network, and its *topological length* is the difference between node depths connected by the wire.

Function f_i associated with node i is referred to as the node's *local* function. It is expressed in terms of the immediate fan-ins. For each node i in the Boolean network there is also an implicit *global* function f_i^* that expresses the functionality of its output signal in terms of a subset of primary inputs. The function can be computed by recursively traversing node fan-ins and applying the composition operation, starting from a given node i . Formally, the global function f_i^* of a node i expressed in terms of primary inputs I is computed as

$$f_i^* = \begin{cases} y_i & i \in I \\ f_i(f_{i_1}^*, \dots, f_{i_k}^*) & \text{otherwise} \end{cases} \quad (2.14)$$

where f_i is the local function associated with node i , and $f_{i_j}^*$'s are global functions of its fan-in nodes. Equation (2.14) readily extends to the computation of global functions expressed in terms of internal node variables. This is achieved by simply including in I any desired set of the intermediate variables.

2.8 Summary

In this chapter we introduced the elementary foundation for the rest of the dissertation. The fundamental concepts of Boolean functions were introduced illustrating their set-theoretical nature. The summarized basic manipulation operations on Boolean functions will be used throughout the dissertation. Partially specified Boolean functions were then discussed. They are inherent to synthesis algorithms. The concept of intervals was developed to address partially specified functions. The expressive power of intervals was demonstrated in the optimization of Boolean networks.

Chapter 3

Constructive Synthesis

In this chapter we describe a constructive synthesis approach that differs from current practice in logic synthesis technology in one important respect: instead of serializing the steps of technology-independent decomposition and technology mapping, the constructive approach interleaves decomposition and mapping throughout the synthesis process. This synthesis flow yields an implementation network that evolves incrementally from the primary inputs towards the primary outputs, and makes it possible for future decomposition decisions to be made more judiciously by accounting for the structure of the partial implementation created thus far. We illustrate this synthesis flow by means of a prototype tool that experimentally demonstrates its benefits, and that gauges its feasibility as an alternative to traditional synthesis.

3.1 Constructive Synthesis Flow

One of the commonly-cited shortcomings of the dominant two-stage synthesis flow is that it fails to properly account for the effects of wiring in deep submicron (DSM) ICs. This is readily explained by the built-in bias of traditional synthesis towards optimizing active logic (gates) and the fact that wires in DSM chips can no longer be ignored as major contributors to area and delay. Constructive synthesis attempts to address this shortcoming by intertwining technology-independent Boolean optimization and technology-dependent mapping in an algorithmic flow that is cognizant of the structural implications of optimization decisions. The overall process is depicted schematically in Figure 3.1. The primary data structure underlying the process is a Boolean network η that is continually modified as synthesis proceeds to reflect the evolving implementation structure of the functions being synthesized. Each node v in this network has an associated Bool-

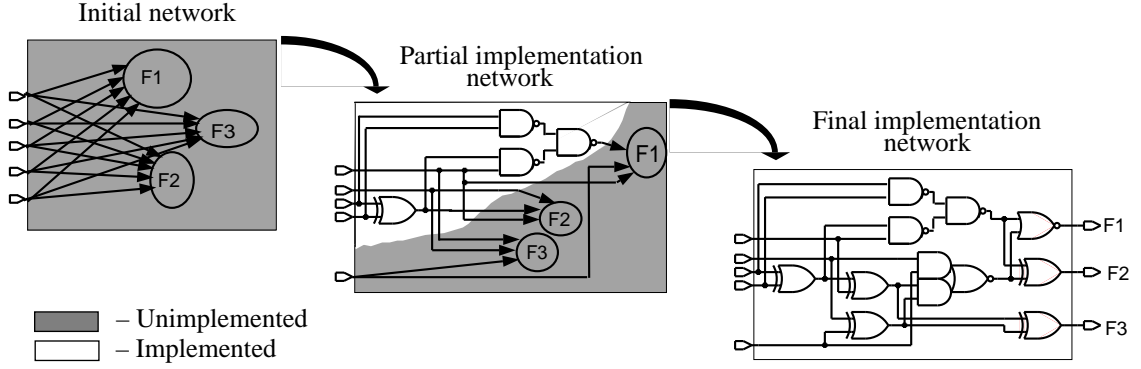


Figure 3.1: Schematic illustration of constructive synthesis

ean function f_v that computes the logic value at the node's output in terms of the logic values on its inputs; it is the local function of the node. A node is considered to be *implemented* if its local function is equivalent to one of the functions in a given gate library L , and *unimplemented* otherwise. To insure an onto mapping from the gates in L to the nodes in the network, the library must, at a minimum, have inverter and pass-through wire gates as well as any 2-input gate that makes it functionally complete (e.g. AND2 or NOR2).

As the synthesis process evolves, the functions of unimplemented nodes are successively decomposed in terms of those of already implemented nodes, resulting in a series of *partial implementation* networks. The decomposition is closely tied with the given library of decomposition primitives, and leads to the creation of new implemented nodes, i.e. nodes that correspond to library primitives. Each implemented node v introduces a new variable y_v which can now be used to simplify the functions of downstream unimplemented nodes. The effect of these successive decompositions is an expansion of the network from the primary inputs towards the primary outputs as more implemented nodes are created and as the functional complexity of unimplemented nodes is reduced. The synthesis process terminates when all nodes become implemented yielding a *final implementation* network.

The constructive creation process facilitates management of the structure of the evolving implementation. Indeed, by decomposing unimplemented portions of the logic directly in terms of library primitives, as opposed to the delayed binding of conventional approaches, the structure of the implementation can be globally controlled yielding higher quality results that can be incrementally modified to satisfy varying area/speed objectives. Additionally, by identifying useful functional properties, the functional content of gate libraries can be chosen, and coupled with

```

Boolean network  $\eta$ ;
set of SOP expressions  $F$ ;      // Set of local functions, in SOP form, of unimplemented nodes;
                                // Initialized to SOP specification of output functions to be
                                // synthesized;

while exists  $f_i \in F$  do {
     $k \leftarrow \text{SelectFunction}()$ ;      // Determine largest  $f_k$ ,  $1 \leq k \leq n$ ;
     $P \leftarrow \text{GenerateDivisor}(f_k)$ ; // Select atomic divisor  $P$ ;
     $y_v \leftarrow \text{IntroduceGates}(P)$ ;   //  $v$  is output node of subcircuit implementing  $P$ ;
     $F \leftarrow \text{Substitute}(P, y_v)$ ;   // Re-express  $F$  in terms of the newly implemented logic;
}

```

Figure 3.2: Overall algorithmic flow in M32

decomposition, more judiciously to produce implementations that reflect the semantic nature of the functions being synthesized.

3.2 M32: A Prototype Constructive Synthesis Program

In this section we provide a rather informal description of M32, a synthesis tool that was developed to evaluate constructive synthesis. The tool uses cube-list (i.e., SOP) representations for all functions, employs a slightly extended algorithm for algebraic division [103], and assumes a primitive library consisting of 2-input NAND gates. These particular implementation choices were motivated by a desire to quickly develop a basic understanding of the major tradeoffs in constructive synthesis. Specifically, we wanted to assess the effect of such a synthesis flow on the structural “quality” of the resulting implementations, and to contrast it with the quality of implementations obtained using traditional synthesis. In later chapters, we develop a more general symbolic framework for constructive synthesis that removes the implementation limitations of M32 and allows for a more fundamental understanding of its benefits. To avoid cluttering the discussion, we omit defining many of the common notions for manipulating SOP expressions. They can be found in virtually any modern book on logic synthesis (see e.g. [18]).

3.2.1 Algorithm overview

The main loop in M32 is shown in the pseudo-code of Figure 3.2. The algorithm operates on two primary data structures: the implementation network η described earlier, and the set of functions F corresponding to η ’s unimplemented nodes. Initially, η consists of primary input nodes and

unimplemented nodes that correspond to each output function being synthesized; the set F is assumed to be given in SOP form for each of these output functions.

In each iteration, the functions of unimplemented nodes are examined and one of them, f_k , is selected for a decomposition step. An *atomic divisor* P is extracted from f_k by an appropriate division procedure. The divisor is subsequently implemented by a small subcircuit of 2-input NAND gates leading to the expansion of the implemented part of η . The decomposition step is completed by substituting the variables y_v of the newly-created nodes into the functions of the unimplemented nodes. The iteration stops, signaling completion of the synthesis process, when F becomes empty.

It is helpful to re-iterate that the main difference between this constructive flow and traditional synthesis is that functional decomposition (*GenerateDivisor* and *Re-express*) and technology mapping (*IntroduceGates*) are interleaved throughout the synthesis process. This, in turn, allows constructive synthesis to consider the structural implications of candidate decompositions. A detailed description of how this is actually done in M32 is described next. In this description we use the following definitions. A literal on a node variable y_v will be denoted as \dot{y}_v . The depth of literal \dot{y}_v , $depth(\dot{y}_v)$, is the topological depth of its corresponding node v in η ; the depth of a primary input is defined to be 0. When calculating the structural cost of a divisor (see below), the depth of a negative literal \bar{y}_v is taken to be the same as that of its positive counterpart y_v ; this is justified by noting that at the stage of selecting divisors, phase assignments in their implementation may not be known. The depth of an SOP expression E , written $depth(E)$, is the maximum depth of any of its literals plus one. The set of literals appearing in E will be denoted $support(E)$. The number of times a literal \dot{y}_v occurs in E will be denoted by $occurrence(\dot{y}_v, E)$, and $size(E)$ will denote the number of literal occurrences in E .

3.2.2 *SelectFunction*

The order in which the functions of F are decomposed clearly affects the final synthesized implementation. In each iteration of the algorithm, the set of nodes that are candidates for decomposition is the subset of unimplemented nodes whose fan-ins are already implemented. The *SelectFunction* routine identifies the next function to decompose by greedily choosing the candidate node whose function has the largest number of literals in its cube representation. This choice is motivated by the expectation that larger, richer, functions yield more divisors that can be shared among the unimplemented nodes. More sophisticated selection strategies can be easily envisioned, especially when the initial specification is a multi-level network.

3.2.3 *GenerateDivisor*

Like most modern synthesis algorithms, M32 relies on an efficient division procedure to decompose a function f into the form $pq + r$. The most commonly-used division procedure is weak division [20] which is applied to an SOP expression for f and is equivalent to excluding all Boolean transformations except for the AND-over-OR distributive law. While primarily motivated by the need for a fast division operation, weak division has been empirically shown to yield acceptable decompositions in practice. Still, the quality of the divisors, as measured by the total number of literals in the resulting factored form, can be improved by judicious application of additional Boolean transformations. The division operation in M32 augments the distributive law with the annihilation ($a \cdot \bar{a} = 0$) and idempotency ($a \cdot a = a$) properties to generate better decompositions. This additional flexibility comes at a modest computational cost.

Example 3.1 Suppose that we would like to obtain a factored form of

$$f = abcg + abe + acde + \bar{a}\bar{b}cd + \bar{a}\bar{b}\bar{e} + \bar{a}c\bar{e}g + cdg$$

A possible algebraic decomposition of f obtained using weak division is

$$f = (ab + d + \bar{a}\bar{e})cg + abe + acde + \bar{a}\bar{b}cd + \bar{a}\bar{b}\bar{e}$$

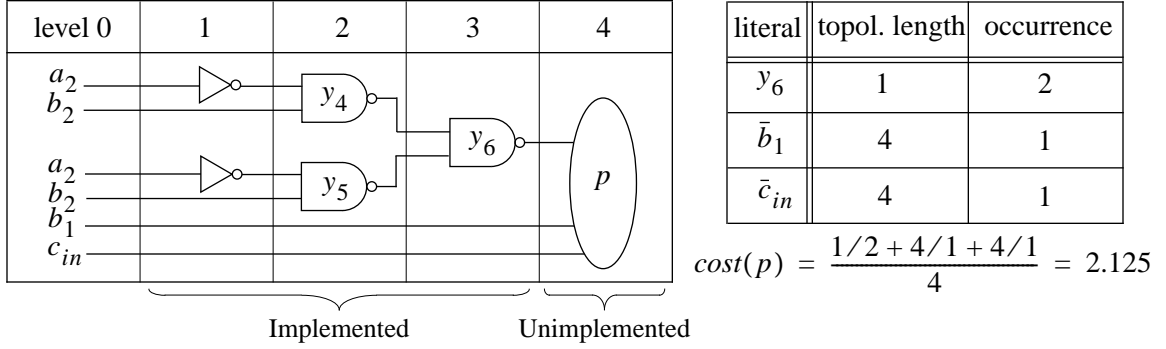
which has a literal cost of 21. Use of annihilation and idempotency yields the more compact decomposition $f = (ab + cd + \bar{a}\bar{e})(ae + cg + \bar{a}\bar{b})$ whose literal cost is only 12. Unlike the algebraic decomposition which yields factors with disjoint support, this decomposition produces factors that have joint support. ■

Divisor selection in M32 is accomplished through successive factorization, using the distributive law, of repeated literals from an SOP expression f . Among all possible divisors, the one returned by *GenerateDivisor* is found by greedily choosing and further factoring the least-cost divisor at each level of literal factorization. This amounts to a 1-level lookahead strategy which was experimentally found to be adequate. Divisor cost is computed according to the formula:

$$\text{cost}(f/\dot{x}) = \frac{\sum_{\dot{y}_v \in \text{support}(f/\dot{x})} \frac{[\text{depth}(f/\dot{x}) - \text{depth}(\dot{y}_v)]}{\text{occurrence}(\dot{y}_v, f/\dot{x})}}{\text{size}(f/\dot{x})} \quad (3.1)$$

where \dot{x} is a candidate literal in expression f and f/\dot{x} is the quotient resulting from algebraic division of f by \dot{x} . Intuitively, the difference $\text{depth}(f/\dot{x}) - \text{depth}(\dot{y}_v)$ corresponds to the topological wire length from the implemented node v to the unimplemented node of f/\dot{x} . The simple reasoning behind this heuristic cost function is that we would like to minimize the literal-weighted sum of wire lengths connecting to the divisor (numerator part), while keeping the divisor large (denom-

$$s_2 = \bar{a}_1(\underbrace{\bar{b}_1 y_6 + \bar{c}_{in} y_6}_p) + \bar{b}_1 \bar{c}_{in} y_6 + b_1 c_{in} \bar{y}_6 + a_1 y_3 \bar{y}_6$$

a) levelized network part for the fan-in connections of divisor p b) evaluated cost of divisor p **Figure 3.3: Example calculation of structural cost of a divisor according to formula (3.1)**

inator part). The numerator part tends to be smaller when literals repeat more often, thereby also suggesting that more factoring of the divisor is possible. Divisors chosen according to this metric account for the structure of the partial implementation network and allow for a more orderly construction process than is possible with two-stage synthesis. The chosen divisors are further improved by invoking the annihilation and idempotency transformations to modify the resulting quotient and reduce the literal cost of the decomposition. The following examples illustrate the structural cost calculation and the divisor selection process.

Example 3.2 In synthesizing an n -bit adder, the unimplemented function of the second sum bit may be given by the SOP expression

$$s_2 = \bar{a}_1(\bar{b}_1 y_6 + \bar{c}_{in} y_6) + \bar{b}_1 \bar{c}_{in} y_6 + b_1 c_{in} \bar{y}_6 + a_1 y_3 \bar{y}_6$$

We can then evaluate the cost of its candidate divisor

$$s_2/\bar{a}_1 = p = \bar{b}_1 y_6 + \bar{c}_{in} y_6$$

by considering the fan-in connections corresponding to the literals of p . These are the incoming connections from the already-implemented part of the circuit, and therefore their topological length can be accurately determined from the levelized circuit (see Figure 3.3). For example, the topological length of \bar{b}_1 is $depth(p) - depth(\bar{b}_1) = 4 - 0 = 4$ and it occurs once in p yielding a contribution of 4 to the numerator of (3.1). The structural cost of using p as a divisor is thus found to be 2.125. ■

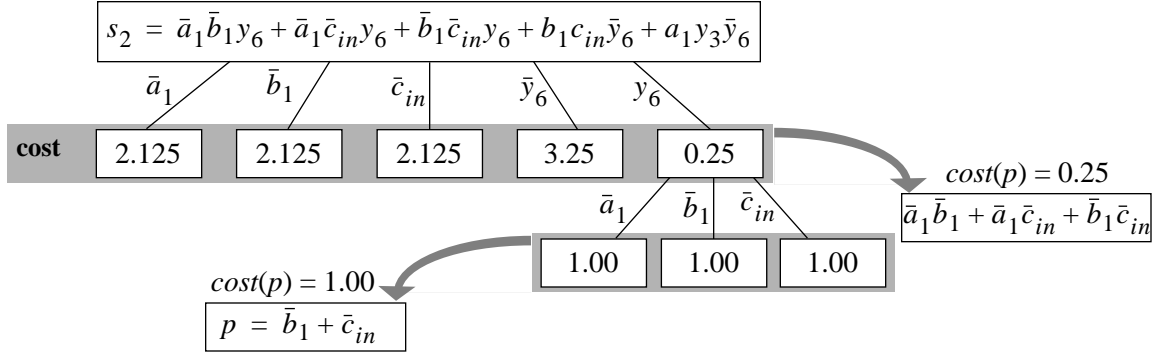


Figure 3.4: 1-level lookahead search tree used in divisor selection; costs are based on the implementation network of Figure 3.3-a

Example 3.3 The divisor whose cost we just calculated is not the best among all possible divisors of s_2 . The hunt for the best divisor, using the structural cost metric (3.1), is shown in Figure 3.4 with the aid of a search tree. The root of the tree has six branches—the result of factoring out six different literals. The least cost divisor at this level is $\bar{a}_1 \bar{b}_1 + \bar{a}_1 \bar{c}_{in} + \bar{b}_1 \bar{c}_{in}$; its cost is 0.25 and is, therefore, selected for subsequent factoring. At the second level of the search tree, all divisors end up having the same cost of 1.00. Thus, one of them, say $p = \bar{b}_1 + \bar{c}_{in}$, is selected arbitrarily and checked for further factorization, which is not possible in this case. Thus, the process terminates and returns this as the atomic divisor. ■

3.2.4 IntroduceGates

Unlike conventional technology mappers that operate on an intermediate “optimized” Boolean network obtained in a prior technology-independent phase, M32 ties its creation of gates with functional decomposition. As soon as a divisor p is found by *GenerateDivisor*, *IntroduceGates* proceeds to map it to the given gate library. The mapping process is also different from those used in conventional synthesis tools: no intermediate *subject graph* is constructed. Similarly to *GenerateDivisor*, this procedure is aware of the structural implications of its choices, and involves iterating the following steps until p is fully implemented by library gates:

1. A gate from the technology library with suitable variable support from p is selected for instantiation as node v
2. The node v , along with possible fan-in inverters, is instantiated and added to η ; variable y_v is associated with the new node v
3. The divisor p is re-expressed in terms of y_v

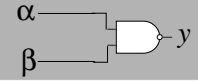
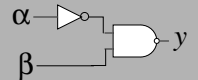
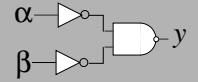
Meta rule	Phase			Implemented as
	p_y	p_α	p_β	
1. $\bar{y} \leftarrow \alpha\beta$ $y \leftarrow \bar{\alpha} + \bar{\beta}$	0 1	1 1	1 1	
2. $\bar{y} \leftarrow \bar{\alpha}\beta$ $y \leftarrow \alpha + \bar{\beta}$	0 1	1 1	0 0	
3. $\bar{y} \leftarrow \bar{\alpha}\bar{\beta}$ $y \leftarrow \alpha + \beta$	0 1	0 0	0 0	

Figure 3.5: Meta rules describing variable support selection and construction of the circuit

The implementation of *IntroduceGates* was limited to a small technology library defined by $L = \{wire, INV, NAND2\}$. Thus, the gate type selection step in the above procedure is unnecessary. Enhancements to *IntroduceGates* that extend its capabilities to richer gate libraries are possible, although the division-based decomposition of SOP expression does not seem to integrate naturally with complex libraries. The selection of a suitable variable support in step 1 is also simplified by the library L . Candidate variable subsets are determined by considering all associative groupings of literal pairs in p , and the best pair is selected. Again, quality in this context is estimated by a structural metric: a pair of literals which instantiates the node v of least depth is greedily selected.

The depth of v in η is derived from the depth of nodes in the subexpression E of an associative grouping while accounting for the possible presence of an inverter on v 's fan-in lines. The presence of an inverter on the fan-in lines is determined by matching subexpression E to the NAND2 function under input/output phase assignments. Due to the small number of possible matching combinations when using only NAND2 gates, we enumerate them as *meta rules* in Figure 3.5. To break ties between the candidate supports we use estimated signal arrival times based on the following delay model:

$$delay = \tau_g + n \times \tau_o \times C_p \quad (3.2)$$

where τ_g is the intrinsic gate delay, n is the gate fan-out, τ_o is fanout delay, and C_p is the capacitance on a fanout pin. In the SIS-1.2 system this model is known as the *library model*. In our experiments we the nominal delay and capacitance were taken from the MCNC library [cite].

After a gate's support and its phases are determined, it is introduced into the circuit by establishing its fan-in connections and placing an inverter (either a new one or by fanning out from a

previously introduced one) on each connection which has negative (0) input phase. No inverter is placed on the gate output regardless of the output phase.

Finally, divisor P is re-expressed in terms of this newly introduced gate, leading to a reduction in its size. The process continues until P is reduced to a single literal. This literal is then returned by the *IntroduceGates* routine, and is later used as a substitute for P in F .

3.2.5 Substitute

At each iteration of the algorithm, the routine *Substitute* re-expresses functions f_1, \dots, f_n of the unimplemented nodes from η in terms of the newly introduced nodes. The substitutions applied to the functions are based on the division operation. If $pq + r$ is the result dividing f by p then substitution re-expresses f as $\dot{y}_v q + r$, where \dot{y}_v is a new literal; the $pq + r$ factored forms define the feasible substitutions. In addition to the distributive, annihilation, and idempotency laws that are used in the division operation, *Substitute* also applies cube reduction implemented using the sharp product $A\#B$ defined as $A\bar{B}$ [23]. Application of this operation is performed if it facilitates division with respect to a given divisor. The example in Section IV illustrates how the use of cube reduction can lead to a better design.

Substitute re-expresses f_1, \dots, f_n in two steps: (1) substituting literal \dot{y}_v ($\dot{\bar{y}}_v$) for the p (\bar{p}) function; and (2) substituting variables of the newly introduced nodes to stand in for their gate functions, modulo the input phase assignments determined in *IntroduceGates*. These steps are applied to each of the functions f_1, \dots, f_n individually. The routine performs substitution selectively to minimize topological wire length according to (3.1). Thus, quotients containing fewer cubes than possible may get selected: cubes of the quotient containing undesirable literals may get omitted. In the first step, *Substitute* uses the literal returned by the *IntroduceGates* routine, which corresponds to the output node of a subcircuit implementing p . The second step of the routine implements substitutions of finer granularity. Substitution of gate functions in this step is performed for each of the nodes in topological order.

Note that substitution using only the distributive law does not require step (1), since it is subsumed by step (2). We should also point out that the “enhanced” division procedure used in M32 may not yield a unique quotient. This is illustrated in the example below, where annihilation gives rise to two distinct decompositions:

Example 3.4 Suppose $f = ab\bar{c}\bar{e} + \bar{a}bcd + a\bar{c}d\bar{e} + \bar{a}cde$, and let $p = a\bar{c}\bar{e} + \bar{a}cd$ be a divisor of f . We can then have two different quotients $q_1 = b + \bar{c}d + ce$ and $q_2 = b + ad + \bar{a}e$. Thus either

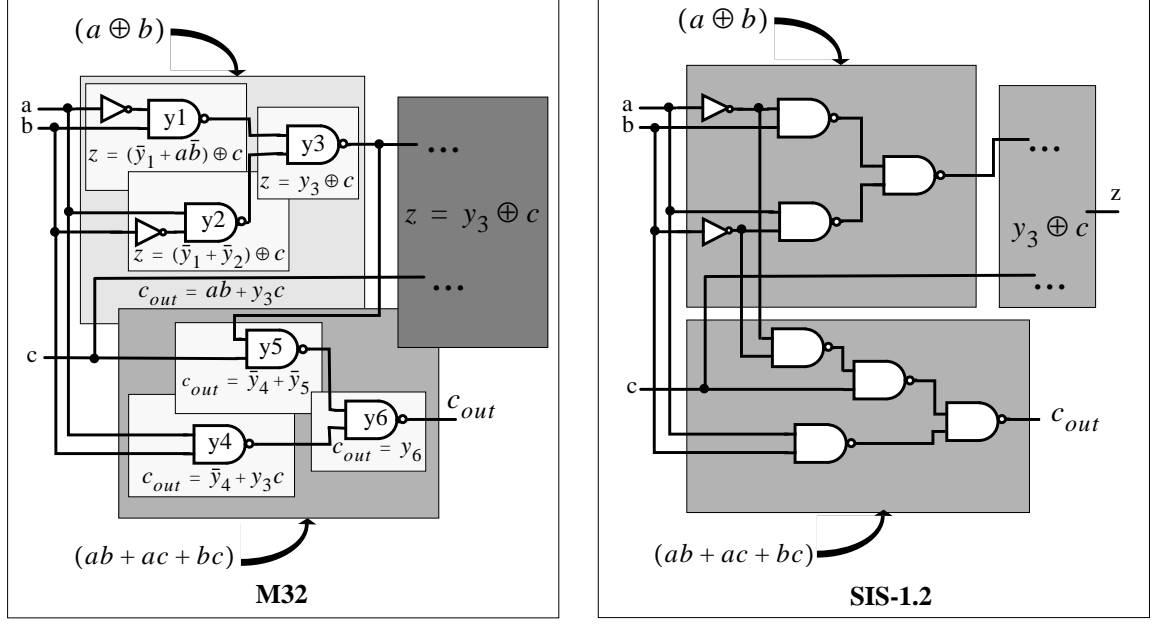


Figure 3.6: Synthesis of the full adder in M32 and SIS-1.2

substitution $f = y_v(b + \bar{c}d + ce)$ or $f = y_v(b + ad + \bar{a}e)$ is feasible. ■

3.2.6 Synthesis example

We illustrate the operation of M32 by tracing the synthesis steps for a full adder. M32 first reads the functional specification

$$z = a \oplus b \oplus c = \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}\bar{c} + abc \quad (3.3)$$

$$c_{out} = ab + ac + bc \quad (3.4)$$

of the circuit to be synthesized in the two-level `pla` format [123]. Thus, $F = \{z, c_{out}\}$ is the set of unimplemented nodes to be synthesized. The algorithm then selects function z since it has more literals than c_{out} . The subexpression P selected from z by the *GenerateDivisor* routine is $\bar{a}b + a\bar{b}$. *GenerateDivisor* selects this divisor since it has the least cost, with the assumption that signal c arrives later than signals a and b . This expression is then implemented using *Introduce-Gates* by instantiating NAND2 gates y_1 , y_2 , and y_3 through the following sequence of transformations of P :

$$P = \bar{a}b + a\bar{b} \xrightarrow{2} \bar{y}_1 + a\bar{b} \xrightarrow{2} \bar{y}_1 + \bar{y}_2 \xrightarrow{1} y_3$$

where the number above each arrow indicates the matching meta rule used from Figure 3.5.

The *Substitute* function then replaces $\bar{a}b + a\bar{b}$ by y_3 (complement substitution is also tried) in both z and c_{out} , yielding:

$$z = \bar{c}y_3 + c\bar{y}_3 \quad (3.5)$$

$$c_{out} = ab + cy_3 \quad (3.6)$$

Note that $a\bar{b} + \bar{a}b$ is not an algebraic divisor of $ab + ac + bc$. Therefore substitution based on weak division alone would not bring any changes to c_{out} . The *Substitute* routine makes substitution in c_{out} possible by through cube reduction using the sharp product. Specifically, *Substitute* determines that the division becomes possible if cubes ac and bc are reduced to $a\bar{b}c$ and $\bar{a}bc$ respectively. This is a valid transformation since the result of $ac\#a\bar{b}c$ and $bc\#\bar{a}bc$, which is in both cases cube abc , is covered by the cube ab . It is now easy to see that $\bar{a}b + a\bar{b}$ divides expression $ab + a\bar{b}c + \bar{a}bc$, representing the same carry-out function $c_{out}: ab + (a\bar{b} + \bar{a}b)c$.

The next step in the *Substitute* routine is to see if any of the local functions of nodes y_1 , y_2 or y_3 can be used to re-express either z or c_{out} . No further substitutions are possible in this case. This completes the first iteration of the algorithm in Figure 3.2.

On the next iteration of the loop either the unimplemented node for c_{out} or z can be selected, since both of their functions have the same number of literals. Figure 3.6 depicts the execution of the algorithm with the assumption that c_{out} is selected. c_{out} is completely implemented in this iteration of through the sequence of transformations:

$$c_{out} = ab + y_3c \xrightarrow{1} \bar{y}_4 + y_3c \xrightarrow{1} \bar{y}_4 + \bar{y}_5 \xrightarrow{1} y_6$$

The final implementation of the circuit has 9 NAND2 gates and 4 inverters. By way of comparison, synthesizing the same circuit using SIS-1.2 (rugged script) requires one more NAND2 gate.

3.3 Empirical Evaluation

To determine the potential of constructive synthesis approach we conducted an experimental evaluation of M32 on publicly available benchmark circuits. The primary goal of these experiments was to see if the method has sufficient merit, compared with conventional synthesis flows, to warrant further exploration. In particular, we were most interested in assessing the impact of the structural metrics as a guide for “simultaneous” decomposition and mapping, while recognizing that the specific implementation choices we made for these computations may not be optimal.

3.3.1 Comparative performance analysis

We evaluated the performance of M32 by synthesizing a set of circuits selected from the MCNC benchmarks [123] and comparing the results against SIS-1.2 [103]. The benchmarks were first

minimized using ESPRESSO [16] prior to multilevel synthesis in M32 or SIS-1.2. We used the `delay` script in SIS-1.2 which is based on the *clustering script* proposed in [114] and is targeted towards technology-independent minimization of circuit delay. Both systems used the minimal gate library $L = \{wire, INV, NAND2\}$, and delay of the implementations obtained by both systems was estimated using the *library* delay model of SIS-1.2 and parameters from the `mcnc.genlib` library.

Table 3.1 compares the generated circuits in terms of the number of gates, levels of logic, topological complexity and estimated pre-layout delay. The results in this table suggest the following observations:

- Even though minimization of gate count is not a primary objective in the M32 system, it generates implementations with fewer gates in all but two cases. In some cases the reduction in gate count is almost 50%.
- M32-generated circuits have consistently fewer logic levels, in several cases being almost half as deep as SIS-generated circuits.
- The topological complexity of M32-generated circuits is consistently lower than that of SIS-generated circuits. The metric is similar to the fanout range suggested by Vaishnav and Pedram [118] for controlling routing complexity during technology-independent logic synthesis. The topological complexity, however, should not be interpreted as the only judge of circuit quality. This becomes apparent from the post-layout results in Table 3.2.
- The pre-layout circuit delays of M32-generated circuits are consistently lower than those of SIS-generated circuits, the average improvement in delay being about 30%.

The run times of M32 were comparable to those of SIS-1.2 for all benchmarks, suggesting that the use of more powerful decompositions and substitutions is computationally feasible.

To get a better indication of synthesis quality, the netlists produced from M32 and SIS-1.2 were laid out using the Epoch standard cell place and route tools [43] from Cascade Design Automation. The layouts were generated using cells in a $0.5\mu\text{m}$ CMOS process with two layers of metal, and allowing over-cell routing. I/O pins were distributed around the perimeter of the standard cell block. Delays were computed using the Epoch static timing analyzer TACTIC. These results, shown in Table 3.2, indicate a 23% average improvement in total area, routing length and post-layout delay for M32-generated circuits. The layouts generated for a representative circuit, the `cordic` benchmark, are shown in Figure 3.7.

Table 3.1: Pre-layout synthesis results

Circuit Name				SIS-1.2				M32				Norm.
	Inp.	Outp.	Cubes	Gates	Levels	Compl.	Delay	Gates	Levels	Compl.	Delay	
z4ml	7	4	59	75	12	1.96	17.6	44	8	1.55	11.6	0.65
vda	17	39	793	1573	29	2.48	62.3	1115	16	1.56	34.3	0.55
inc	7	9	42	152	20	2.20	30.2	133	10	1.55	17.2	0.56
count	35	16	184	311	17	2.55	24.1	201	13	2.19	21.1	0.87
ldd	9	19	70	139	13	1.96	18.9	110	10	1.84	15.4	0.81
b9	41	21	141	176	12	1.69	17.8	177	10	1.60	14.8	0.83
ex4	128	28	620	690	20	1.73	28.0	665	15	1.43	19.3	0.68
cordic	23	2	1180	182	18	1.65	23.4	133	11	1.39	14.6	0.62
cps	24	109	654	2162	35	2.52	61.9	1625	17	1.93	41.3	0.66
duke2	22	29	120	743	20	2.17	35.0	534	13	1.68	24.8	0.70
vg2	25	6	110	239	16	1.88	21.6	152	11	1.36	15.5	0.71
apex2	39	3	438	564	33	2.92	44.6	484	18	1.83	25.4	0.56
sqrt8	8	4	88	79	14	1.84	18.3	76	12	1.72	17.6	0.96
bw	5	28	110	232	16	1.91	28.9	206	9	1.58	18.3	0.63
clip	9	5	167	240	26	2.55	31.3	309	14	1.75	17.5	0.55

Table 3.2: Post-layout synthesis results

Circuit Name	SIS-1.2			M32			Improvement M32 / SIS-1.2		
	Total area (mil ²)	Routing length (μm)	Delay (ns)	Total area (mil ²)	Routing length (μm)	Delay (ns)	Total area	Routing length	Delay
z4ml	33.59	6592.0	2.66	21.55	4212.2	1.81	0.64	0.63	0.68
vda	1159.20	326400.4	7.45	715.642	189921.6	5.87	0.61	0.58	0.78
inc	77.58	17041.5	4.21	62.24	14039.6	2.52	0.80	0.82	0.59
count	137.7	28675.4	3.63	81.51	16941.6	3.62	0.59	0.59	0.99
ldd	60.73	12220.2	2.96	48.09	9123.3	2.18	0.80	0.74	0.73
b9	74.95	15136.6	2.88	76.93	15519.5	2.10	1.02	1.02	0.72
ex4	311.72	66272.5	4.16	294.92	60746.6	3.15	0.94	0.91	0.75
cordic	70.82	13996.6	3.55	54.59	10751.4	1.98	0.77	0.76	0.55
cps	1414.44	393693.7	8.19	1032.2	271116.8	5.74	0.72	0.68	0.70
duke2	414.28	106240.6	4.87	273.10	65001.5	3.80	0.65	0.61	0.78
vg2	100.47	20847.5	3.20	63.17	12526.6	2.15	0.63	0.60	0.67
apex2	283.02	68251.2	6.89	237.54	54155.8	4.67	0.90	0.79	0.67
sqrt8	34.44	7245.9	2.74	32.14	6640.8	2.41	0.94	0.91	0.87
bw	111.25	24798.6	3.71	99.46	22616.8	2.46	0.89	0.91	0.66
clip	116.79	25814.6	5.87	156.59	34502.7	3.71	1.34	1.33	0.63
Average Improvement:							0.81	0.78	0.71

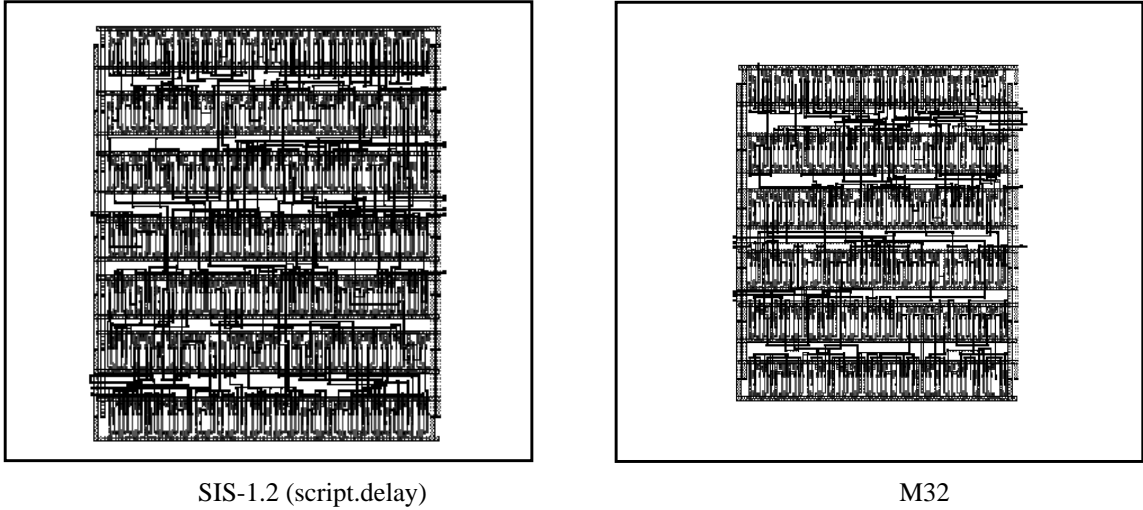


Figure 3.7: Cordic relative layout areas (shown to the same scale)

3.3.2 Effects of richer gate libraries

The most severe implementation choice in M32 was the restriction of the primitive library to NAND2 gates. To insure that the comparative edge M32 has over SIS is due to its overall constructive flow and not to the severe handicap imposed on SIS by limiting it to a NAND2-only library, we conducted one more experiment. In this experiment, a richer gate library is used in both tools. However, since the only library supported by M32 is the simple NAND2/INV library, we had to resort to a less-than-ideal work-around to generate circuits based on other libraries. Using the SIS-1.2 `map` command, the NAND2/INV circuits produced by M32 were technology-mapped to the `mcnc.genlib` library. The same mapping process was also applied to the NAND2/INV circuits generated by SIS-1.2. The results of this experiment are shown in Table 3.3. The columns labeled “Area” and “Delay” record, respectively, the active areas and circuits delays of each implementation as reported by the mapper.

An examination of these results indicates that, overall, the circuits produced from M32 are still faster and smaller than those produced from SIS-1.2. However, the improvement is not as pronounced as it was for the NAND2/INV library. This outcome is hardly surprising since the above mapping process is decidedly antithetical to the constructive synthesis philosophy of M32 and undoes many of its gains. Specifically, mapping by tree covering is ill-suited to a highly optimized DAG and we conjecture that exact DAG covering may have produced better results. It must be noted, however, that DAG covering is notoriously difficult and no published algorithms that can effectively handle large circuits have been demonstrated. This strengthens our belief that a more

Table 3.3: Results from mapping to a richer technology library

Circuit	SIS				M32			
	NAND2/INV		mcnc.genlib*		NAND2/INV		mcnc.genlib*	
Name	Area	Delay	Area	Delay	Area	Delay	Area	Delay
z4ml	142	17.6	131	12.0	84	11.6	85	11.2
vda	2731	62.3	2188	21.6	1870	34.3	1558	20.1
inc	268	30.2	223	13.8	236	17.2	189	10.8
count	521	24.1	394	14.4	340	21.1	277	17.2
ldd	237	18.9	198	12.7	194	15.4	167	12.1
b9	280	17.8	224	12.6	297	14.8	261	10.8
ex4	1149	28.0	888	18.5	1078	19.3	843	15.6
cordic	301	23.4	244	17.2	216	14.6	163	11.1
cps	3641	61.9	2862	29.5	2753	41.3	2368	20.0
duke2	1263	35.0	1039	18.6	898	24.8	779	15.6
vg2	394	21.6	297	12.9	239	15.5	174	14.1
apex2	963	44.6	797	28.9	797	25.4	649	21.5
sqrt8	133	18.3	111	12.4	130	17.6	110	11.5
bw	395	28.9	333	14.3	349	18.3	328	11.3
clip	413	31.3	338	23.9	521	17.5	437	15.7

* Technology mapping was done using the SIS-1.2 command `map -s -n 1 -AFG -p`

natural approach for solving this problem is the extension of the M32 algorithm to handle richer gate libraries directly. This effort is described in subsequent chapters.

3.3.3 Dynamic execution highlights

We illustrate M32's operation using two benchmark circuits, `clip` and `cordic`, to highlight the dynamic nature of the constructive approach to synthesis.

One advantage of building a circuit implementation in this manner is that we are able to estimate progress during the execution of the algorithm. Indeed, at any given point we may take as a measure of progress the fraction of logic implemented so far to the original logic specification. Specifically, let F represent the remaining unimplemented logic at a certain point during synthesis, and F_0 denote the initial functional specification. The percent of logic implemented by a partial circuit can then be measured as $((|F_0| - |F|) / |F_0|) \cdot 100\%$, where $|\cdot|$ denotes function size in a given representation. Since M32 uses the SOP form to represent functions, a suitable size measure is the total number of literals in an SOP expression. A visual indication of progress is provided by a *dynamic execution curve* that shows the percentage of implemented logic as a function of circuit size, in gates, at each synthesis iteration.

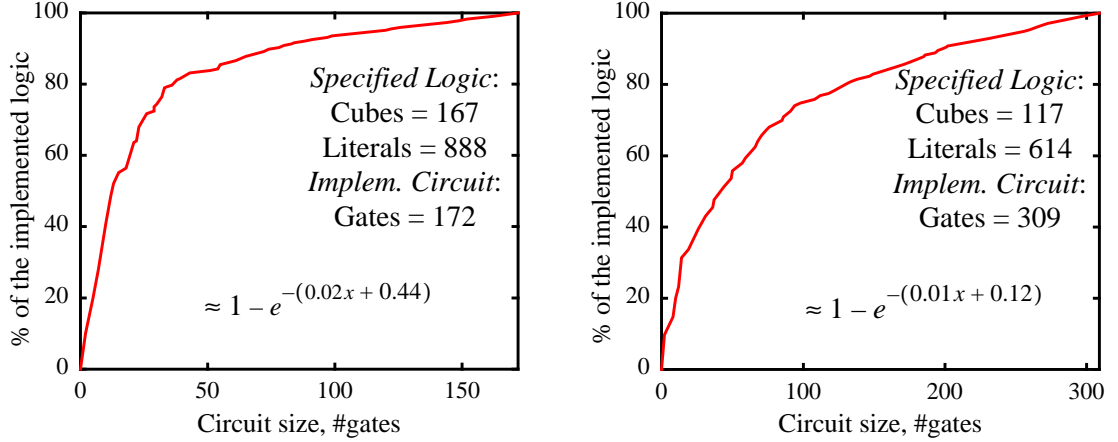
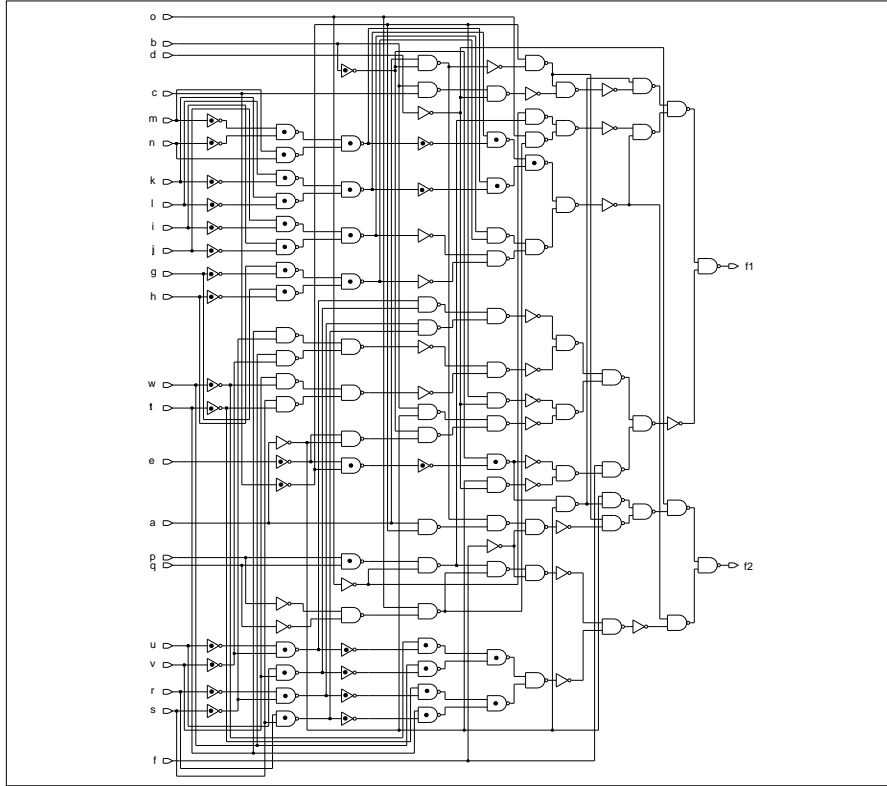


Figure 3.8: Dynamic execution curves for the `clip` circuit using two different covers

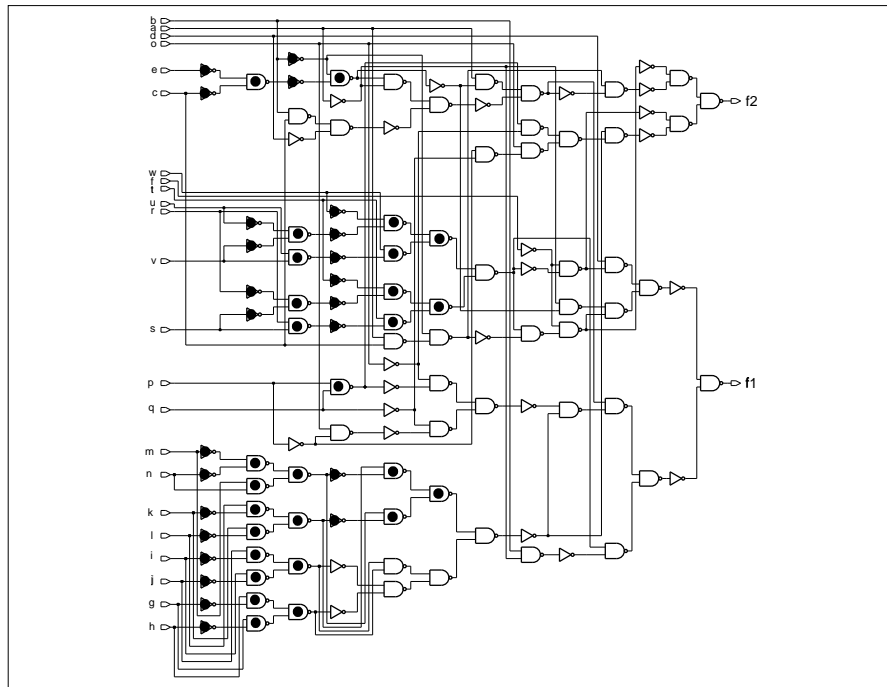
An example of this curve for two different M32 synthesis runs for the `clip` circuit is shown in Figure 3.8. Two distinct covers were used as the initial starting points. In both cases, the dependence of “synthesis progress” on circuit size seems to fit the exponential template $1 - e^{-(ax+b)}$ whose coefficients can be viewed as indicators of synthesis quality for a given cover. The behavior of the curve is dominated by a , denoting the average rate at which the amount of unimplemented logic is reduced at each iteration. These curves suggest that we should strive to make a as large as possible, while keeping b relatively small. In general the value of the coefficient a has much greater impact on final circuit size than the size of the initial form used to represent a function. In the Figure 3.8 experiments, these coefficients vary due to the distinct covers used in the synthesis of the `clip` circuit.

One rather odd outcome of this experiment is that the larger initial cover produced a smaller implementation. This essentially defies the common practice of many multi-level synthesis algorithms of applying a two-level minimization step before, or during (such as done in Boolean division [119]), decomposition of a function. It would seem that the best synthesis algorithms would be less sensitive to the initial form representing a function, or would have a better quality measures evaluating the form.

The ability of constructive synthesis to dynamically adjust the implementation topology is illustrated in Figure 3.9 for the `cordic` benchmark. The two variants shown in the figure were forced to diverge after the 28th iteration of the synthesis loop (about 1/3 of the total). The gates marked with ● in both variants correspond to the common portion of the implemented schematics. The implementation in part (a) of the figure corresponds to the results shown in Table 3.1 and



a) Synthesis with topological constraints: 133 gates, 11 levels



b) Synthesis with relaxed structural constraints: 116 gates, 13 levels

Figure 3.9: Two incrementally-different implementations of the cordic circuit

Table 3.2, and reflects the incorporation of topological complexity constraints. The implementation in part (b) was generated by relaxing these constraints after the 28th iteration. This incremental synthesis capability can prove invaluable when the generated netlists marginally fail to meet specifications and must be fine tuned in the neighborhood of a given solution

3.4 Constructive Synthesis Objectives

Within the constructive synthesis paradigm there are many choices to be made. In particular, the manner in which decomposition is performed and the content of the library can have a profound impact on solution quality. In M32 we used algebraic division and a NAND2 library; the main goal was to understand the implications of this novel constructive synthesis flow (e.g. its ability to control implementation structure) as opposed to fine tuning its individual steps. The exercise helped point out the limitations of a division-based decomposition approach since because of its reliance on particular functional forms (e.g. SOP) that bear no relation to the semantic nature of the functions being synthesized. Unrestricted Boolean transformations, on the other hand, are impractical except for small-scale problems. The simple example below illustrates the difficulty in applying Boolean properties to sum-of-products expressions to obtain good decomposition forms.

Example 3.5 Consider the decomposition of the following function

$$f = \bar{a}\bar{b}cd + \bar{a}b\bar{c}d + a\bar{b}\bar{c}d + abc\bar{d} + \bar{b}\bar{c}de + ab\bar{d}e + ac\bar{d}e + bc\bar{d}e$$

It is not difficult to check that $a \oplus b \oplus c$ is a good divisor for the function. Using this divisor the function f can be decomposed into following equations:

$$f = (sum + e) \cdot (c_{out} \oplus d)$$

$$sum = a \oplus b \oplus c$$

$$c_{out} = ab + ac + bc$$

These equations are especially attractive when complex modules, such as full adders, are present in the implementation library. Unfortunately, deriving them from the original equations requires an arsenal of Boolean properties, whose application is non-obvious. Indeed, it is easy to check that flattening these equations under distributivity, annihilation, and idempotence yields the SOP form $\tilde{f} = f + \bar{a}\bar{b}de + \bar{a}\bar{c}de$, which is clearly different from the original f – it is redundant in the two cubes $\bar{a}\bar{b}de$ and $\bar{a}\bar{c}de$. Removing these two cubes would recreate the original f , but requires the use of the additional absorptive property of Boolean algebra. This example also illustrates that minimal SOP form may not be the best starting point for the decomposition of expressions. ■

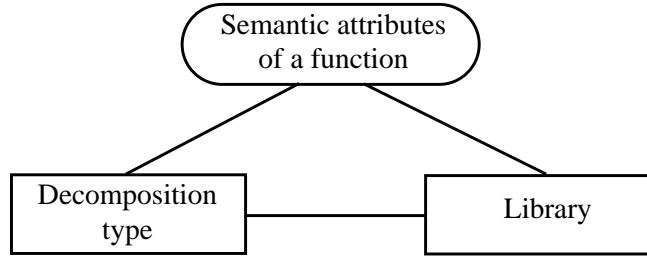


Figure 3.10: Integration of decomposition type and library via semantic properties of a functional specification

To address the inherent computational complexity of decomposition in the unrestricted Boolean domain we therefore suggest a Boolean decomposition strategy that ties together:

1. the *semantic attributes* of the functions being synthesized
2. the *structural attributes* of the implementation network
3. the *functional content* of the basic decomposition primitives

The central idea behind this integration is the premise that functional specifications have global semantic attributes that can be profitably used to induce a favorable structural implementation, while reducing the run time complexity of the synthesis process. These attributes can have a profound effect on the suitability of one decomposition type over another. They can be further utilized to study requirements on the functionality of library primitives to make a particular decomposition type effective. Thus, by judiciously coupling the decomposition type with a library using semantic attributes of a function we are able to merge the traditionally separate library-independent and library-aware synthesis stages (see Figure 3.10). The effect of such integration is the reflection of global functional properties in the implemented circuit structure.

Establishing a direct relation between the functional structure of a logic specification and the ultimate topological and physical structure of its physical realization is, therefore, our primary objective in the following chapters. In particular, we address the key points needed to establish this relation in the following order:

- An understanding of the complete decomposition space available in constructive synthesis. This implies that decomposition operations should be defined to be independent of the actual representation of a function.
- Gaining insights into the semantic properties of functions, particularly symmetry.
- Evaluation of the semantic attributes (symmetry) of a function and their implications on the decomposition type and decomposition primitives.

- Leveraging available technology in scalable software implementations.

3.5 Summary

In this chapter we introduced a new approach to synthesis based on a constructive paradigm that differs in a fundamental way from the two-stage synthesis style in common use today. This paradigm was validated by creating a prototype synthesis tool that was used to assess its potential and to suggest ways to improve it. This investigation clearly showed the inherent benefits of constructive synthesis. It also highlighted the disadvantages of coupling decomposition too closely to a specific representation of the functions being synthesized. The experiment also led us to conjecture that a much closer integration of the semantic attributes of functions, circuit structure, and primitive libraries would make constructive synthesis more effective. This integration essentially amounts to inferring appropriate decomposition types and decomposition primitives from the semantic properties of a function and motivates the work described in the remainder of this dissertation.

Chapter 4

Decomposition Space in Constructive Synthesis

This chapter provides a formal analysis of the decomposition choices that arise in constructive synthesis. They are described symbolically, and are not tied to a specific representation of a Boolean function. Thus the available decomposition choices remain general enough to allow circuit implementations which capture semantic properties of a given function, rather than its representation form. The symbolic formulation of decomposition is then discussed in the context of practical synthesis constraints targeting its computational efficiency.

4.1 Symbolic Formulation of Decomposition

In this section we use a symbolic model for functional decomposition that allows us to pose and answer several key questions related to scalable synthesis, including the existence of a decomposition, and the existence of universal primitives that allow the decomposition of certain classes of functions.

4.1.1 Generic decomposition template

Given an n -variable Boolean function $f(\mathbf{x})$, and k n -variable Boolean functions $g_1(\mathbf{x}), \dots, g_k(\mathbf{x})$, we say that f has an n -to- k decomposition with respect to $g_1(\mathbf{x}), \dots, g_k(\mathbf{x})$ if and only if there exists a k -variable function h such that

$$f(\mathbf{x}) = h(g_1(\mathbf{x}), \dots, g_k(\mathbf{x})) \quad (4.1)$$

A pictorial representation of this decomposition template is shown in Figure 4.1; h will be referred to as the *composition function*, whereas g_1, \dots, g_k will be called the *decomposition functions*. These functions introduce intermediate variables y_1, \dots, y_k into the network that serve as

the support of the composition function. The decomposition is *support-reducing* if $k < n$. The k decomposition functions can be viewed as a single multi-output decomposition function $\mathbf{g}(\mathbf{x}) \equiv (g_1(\mathbf{x}), \dots, g_k(\mathbf{x}))$, and the intermediate variables can be represented by a k -vector $\mathbf{y} \equiv (y_1, \dots, y_k)$. Referring to Figure B.2 on page 144, the multi-output function $\mathbf{g}(\mathbf{x})$ corresponds to block **A** in the hierarchical connections schematic; function $h(\mathbf{y})$, on the other hand, corresponds to block **B** in the schematic.

The decomposition template in (4.1) is sufficiently general to encompass all types of functional decomposition described in the literature, including simple and complex disjunctive and non-disjunctive decompositions [35]. As we show later, support-reducing decompositions in terms of fan-in bounded decomposition functions are particularly attractive from a practical perspective. Before such restrictions are imposed, though, we show in the remainder of this section the relations that must exist between the composition and decomposition functions for equation (4.1) to hold.

4.1.2 Computation of composition function

To determine if the decomposition in (4.1) exists, we can solve for $h(\mathbf{y})$ in terms of $\mathbf{g}(\mathbf{x})$ and $f(\mathbf{x})$. The solution, in general, is not unique and can be expressed as a function interval. The interval solution for $h(\mathbf{y})$ corresponds to a partially specified function, whose flexibility is modeled in terms of the following function:

$$c(\mathbf{x}, \mathbf{y}) = (y_1 \equiv g_1(\mathbf{x})) \cdot \dots \cdot (y_k \equiv g_k(\mathbf{x})) \quad (4.2)$$

This is a characteristic function (discussed in Chapter 2) which captures the consistent input-output assignments of the subcircuit corresponding to the decomposition functions. In recent years characteristic functions have been used to describe the flexibility that arises in design optimization. Viewed as Boolean relations, Brayton and Somenzi [22] described how they can be used to compute the flexibility in optimizing hierarchical designs. Savoj used the output characteristic function to study flexibility in the optimization of Boolean networks in the context of observability relations [97].

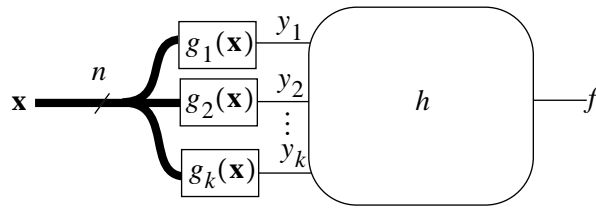


Figure 4.1: Generic decomposition step

Function $c(\mathbf{x}, \mathbf{y})$ represents the constraints introduced by the decomposition functions which can be viewed as a care set when selecting $h(\mathbf{y})$. Indeed, for each point $(\hat{\mathbf{x}}, \hat{\mathbf{y}})$ from this set the value of $h(\hat{\mathbf{y}})$ must agree with the value of $f(\hat{\mathbf{x}})$; at all other points outside of $c(\mathbf{x}, \mathbf{y})$ the values of $h(\mathbf{y})$ can be chosen arbitrarily. This flexibility in selecting $h(\mathbf{y})$ can be described by means of a partially specified function $H(\mathbf{y})$ which identifies all valid selections for $h(\mathbf{y})$. We perform the derivation of $H(\mathbf{y})$ in two steps: first, the function $H(\mathbf{x}, \mathbf{y})$ is defined within the extended domain of function $c(\mathbf{x}, \mathbf{y})$, and then its domain is reduced to obtain $H(\mathbf{y})$.

The first step readily leads to the following expansion for $H(\mathbf{x}, \mathbf{y})$:

$$H(\mathbf{x}, \mathbf{y}) = [c(\mathbf{x}, \mathbf{y}) \cdot f(\mathbf{x}), \overline{c(\mathbf{x}, \mathbf{y})} + f(\mathbf{x})] \quad (4.3)$$

This solution is obtained by direct application of Theorem 2.7 and its validity is easily checked by considering two cases: $c(\mathbf{x}, \mathbf{y}) = 1$ and $c(\mathbf{x}, \mathbf{y}) = 0$. The dependence of $H(\mathbf{x}, \mathbf{y})$ on the \mathbf{x} variables in the interval of (4.3) makes the solution unconditionally consistent, implying that for a given $c(\mathbf{x}, \mathbf{y})$ there is always a non-empty $H(\mathbf{x}, \mathbf{y})$. However, the dependence on \mathbf{x} in $H(\mathbf{x}, \mathbf{y})$ is not consistent with the decomposition template in (4.1) – the template restricts the composition function h to be vacuous in the \mathbf{x} variables, while $H(\mathbf{x}, \mathbf{y})$ is not. To make $H(\mathbf{x}, \mathbf{y})$ vacuous in the \mathbf{x} variables we must ensure that for any given assignment $\hat{\mathbf{y}}$, values of $H(\mathbf{x}, \mathbf{y})$ agree for all assignments $\hat{\mathbf{x}}$. This requirement can be satisfied by invoking Theorem 2.6 and equation (2.12) yielding the function interval:

$$H(\mathbf{y}) = \nabla_{\mathbf{x}} H(\mathbf{x}, \mathbf{y}) = [\exists \mathbf{x}(c(\mathbf{x}, \mathbf{y}) \cdot f(\mathbf{x})), \forall \mathbf{x}(\overline{c(\mathbf{x}, \mathbf{y})} + f(\mathbf{x}))] \quad (4.4)$$

The existential and universal quantification of \mathbf{x} from the lower and upper interval bounds corresponds to the removal of these variables from the support of possible composition functions h making them functions of just the intermediate variables \mathbf{y} , thereby reflecting the structure of Figure 4.1. If the interval (4.4) is non-empty, then there is a Boolean function h which is vacuous in \mathbf{x} , and which belongs to this interval. We say that decomposition *exists* when the interval is non-empty; otherwise the decomposition does not exist. In the examples below we illustrate how the choice of decomposition functions can affect the existence of a decomposition.

Example 4.1 Let $f(x_1, x_2) = x_1 \oplus x_2$, $g_1(x_1, x_2) = x_1$, $g_2(x_1, x_2) = \bar{x}_1 + \bar{x}_2$, and $g_3(x_1, x_2) = x_2$. The input-output characteristic function of these decomposition functions is:

$$c(\mathbf{x}, \mathbf{y}) = (y_1 \equiv x_1) \cdot (y_2 \equiv \bar{x}_1 + \bar{x}_2) \cdot (y_3 \equiv x_2)$$

Using this function we compute the lower bound for the interval in (4.4) as

$$\begin{aligned} \exists \mathbf{x}(c(\mathbf{x}, \mathbf{y}) \cdot f(\mathbf{x})) &= c(0, 0, y) \cdot f(0, 0) + c(0, 1, y) \cdot f(0, 1) \\ &\quad + c(1, 0, y) \cdot f(1, 0) + c(1, 1, y) \cdot f(1, 1) \end{aligned}$$

$$= \bar{y}_1 y_2 \bar{y}_3 \cdot 0 + y_1 y_2 \bar{y}_3 \cdot 1 + \bar{y}_1 y_2 y_3 \cdot 1 + y_1 \bar{y}_2 y_3 \cdot 0$$

and the upper bound as

$$\begin{aligned} \forall \mathbf{x}(\overline{c(\mathbf{x}, \mathbf{y})} + f(\mathbf{x})) &= (y_1 + \bar{y}_2 + y_3 + 0) \cdot (\bar{y}_1 + \bar{y}_2 + y_3 + 1) \\ &\quad \cdot (y_1 + \bar{y}_2 + \bar{y}_3 + 1) \cdot (\bar{y}_1 + y_2 + \bar{y}_3 + 0) \end{aligned}$$

These bounds define an interval of sixteen different composition functions h which can be used for the decomposition of f :

$$H(y_1, y_2, y_3) = [y_1 y_2 \bar{y}_3 + \bar{y}_1 y_2 y_3, \bar{y}_1 \bar{y}_2 + y_1 y_2 + \bar{y}_2 \bar{y}_3 + y_2 y_3]$$

For instance, $h_1(y_1, y_2, y_3) = \bar{y}_1 y_3 + y_1 y_2$ and $h_2(y_1, y_2, y_3) = y_1 y_2 + y_2 y_3$ are two functions from this interval that represent two possible decompositions of f as can be readily verified by substitution in (4.1). ■

Example 4.2 Let $f(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_3$, $g_1(x_1, x_2, x_3) = \bar{x}_1 + \bar{x}_2$, and $g_2(x_1, x_2, x_3) = x_3$. The $H(y_1, y_2)$ interval in this case is:

$$H(y_1, y_2) = [y_1 + y_2, \bar{y}_1 y_2]$$

which is empty¹ since its upper bound is less than its lower bound. Thus, f cannot be decomposed in terms of the given decomposition functions. If, however, the first decomposition function is replaced with $g_1(x_1, x_2, x_3) = x_1 \oplus x_2$, then (4.1) yields the following interval:

$$H = [y_1 \oplus y_2, y_1 \oplus y_2]$$

representing a unique composition function. ■

In general, it is always possible to find k decomposition functions that make the interval in (4.4) non-empty. For example, when $k \geq n$ the decomposition functions can be selected by assuming that at least n of these functions are trivial pass-through wires corresponding to the support of f . The remaining decomposition functions can be selected arbitrarily since their output signals can be assumed redundant in $h(\mathbf{y})$. Similarly, when $k < n$ the interval can be made non-empty by letting one of the decomposition functions correspond to $f(\mathbf{x})$. Such trivial selections of decomposition functions have little practical value, though, and additional constraints on decomposition functions must be imposed to make such decompositions useful in synthesis. These additional constraints, and their impact on synthesis quality, are addressed later.

1. $[y_1 + y_2, \bar{y}_1 y_2]$ is not empty when $y_1 = 0$. For a decomposition to exist, however, the interval must be non-empty unconditionally.

4.1.3 Computation of decomposition functions

Equation (4.4) can be used to compute sets of decomposition functions that will guarantee the existence of a decomposition according to the template in (4.1). Computation of such decomposition functions forms the starting point for the problem of library construction which we discuss later in Section 4.2. To solve for the decomposition functions, we begin by noting that an arbitrary n -variable Boolean function can be expressed in terms of 2^n binary coefficients that denote the function value at each point in its variable space. Thus, we can express $g_j(\mathbf{x})$ as:

$$g_j(\mathbf{x}) = \sum_{i=0}^{2^n-1} \gamma_{ij} \cdot m_i(\mathbf{x}) \quad (4.5)$$

where $\gamma_{ij} \in \{0, 1\}$ and $m_i(\mathbf{x})$ is the minterm on \mathbf{x} whose bits form the decimal value i ; for example, $m_2(ab) = a\bar{b}$ and $m_9(cbda) = c\bar{b}\bar{d}a = a\bar{b}c\bar{d}$. Using $\Gamma =_{\text{df}} [\gamma_{ij}]$ to denote the $2^n \times k$ matrix of coefficients representing the k decomposition functions, the care set $c(\mathbf{x}, \mathbf{y})$ can be re-written as:

$$C(\mathbf{x}, \mathbf{y}, \Gamma) = \prod_{j=1}^k \left[y_j \equiv \sum_{i=0}^{2^n-1} \gamma_{ij} \cdot m_i(\mathbf{x}) \right] \quad (4.6)$$

An assignment to the Γ coefficients in the above encoding induces a binary relation between the minterm spaces of the \mathbf{x} and \mathbf{y} variables. There is a total of $2^k \cdot 2^n$ such assignments, each of which completely defines a multi-output function $\mathbf{g} : B^n \rightarrow B^k$.

In our next step we make use of the following result, derived from the interval in (4.4):

$$\begin{aligned} & [\exists \mathbf{x}(c(\mathbf{x}, \mathbf{y}) \cdot f(\mathbf{x})), \forall \mathbf{x}(\overline{c(\mathbf{x}, \mathbf{y})} + f(\mathbf{x}))] \\ \Leftrightarrow & \exists \mathbf{x}(c(\mathbf{x}, \mathbf{y}) \cdot f(\mathbf{x})) \leq \forall \mathbf{x}(\overline{c(\mathbf{x}, \mathbf{y})} + f(\mathbf{x})) && \text{interval definition} \\ \Leftrightarrow & \overline{\exists \mathbf{x}(c(\mathbf{x}, \mathbf{y}) \cdot f(\mathbf{x}))} + \forall \mathbf{x}(\overline{c(\mathbf{x}, \mathbf{y})} + f(\mathbf{x})) = 1 && (2.4) \text{ identity} \end{aligned}$$

Substituting (4.6) in the last line of the above derivation, and universally quantifying \mathbf{y} , we obtain

$$G(\Gamma) = \forall \mathbf{y}(\overline{\exists \mathbf{x}(C(\mathbf{x}, \mathbf{y}, \Gamma) \cdot f(\mathbf{x}))} + \forall \mathbf{x}(\overline{C(\mathbf{x}, \mathbf{y}, \Gamma)} + f(\mathbf{x}))) \quad (4.7)$$

which is a Boolean function that encodes all feasible decomposition functions \mathbf{g} . The universal quantification of the \mathbf{y} variables in (4.7) ensures that the computed decomposition functions remain valid for all combinations of their output values.

Example 4.3 We apply formula (4.7) to compute 3-to-2 decomposition solutions for the function $f(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_3$. The space of 3-to-2 decomposition functions is encoded as:

$$\begin{aligned}
C(\mathbf{x}, \mathbf{y}, \Gamma) = & (y_1 \equiv (\gamma_{01}\bar{x}_1\bar{x}_2\bar{x}_3 + \gamma_{11}\bar{x}_1\bar{x}_2x_3 + \gamma_{21}\bar{x}_1x_2\bar{x}_3 + \gamma_{31}\bar{x}_1x_2x_3 + \\
& \gamma_{41}x_1\bar{x}_2\bar{x}_3 + \gamma_{51}x_1\bar{x}_2x_3 + \gamma_{61}x_1x_2\bar{x}_3 + \gamma_{71}x_1x_2x_3)) \cdot \\
& (y_2 \equiv (\gamma_{02}\bar{x}_1\bar{x}_2\bar{x}_3 + \gamma_{12}\bar{x}_1\bar{x}_2x_3 + \gamma_{22}\bar{x}_1x_2\bar{x}_3 + \gamma_{32}\bar{x}_1x_2x_3 + \\
& \gamma_{42}x_1\bar{x}_2\bar{x}_3 + \gamma_{52}x_1\bar{x}_2x_3 + \gamma_{62}x_1x_2\bar{x}_3 + \gamma_{72}x_1x_2x_3))
\end{aligned}$$

Together with f , this function is now used to construct formula (4.7). For the part corresponding to the lower bound in the formula we have:

$$\begin{aligned}
\exists \mathbf{x}(C(\mathbf{x}, \mathbf{y}, \Gamma) \cdot f(\mathbf{x})) = & (y_1 \equiv \gamma_{01}) \cdot (y_2 \equiv \gamma_{02}) \cdot 0 + (y_1 \equiv \gamma_{11}) \cdot (y_2 \equiv \gamma_{12}) \cdot 1 + \\
& (y_1 \equiv \gamma_{21}) \cdot (y_2 \equiv \gamma_{22}) \cdot 1 + (y_1 \equiv \gamma_{31}) \cdot (y_2 \equiv \gamma_{32}) \cdot 0 + \\
& (y_1 \equiv \gamma_{41}) \cdot (y_2 \equiv \gamma_{42}) \cdot 1 + (y_1 \equiv \gamma_{51}) \cdot (y_2 \equiv \gamma_{52}) \cdot 0 + \\
& (y_1 \equiv \gamma_{61}) \cdot (y_2 \equiv \gamma_{62}) \cdot 0 + (y_1 \equiv \gamma_{71}) \cdot (y_2 \equiv \gamma_{72}) \cdot 1
\end{aligned}$$

Similarly, for the upper bound we have:

$$\begin{aligned}
\forall \mathbf{x}(\overline{C(\mathbf{x}, \mathbf{y}, \Gamma)} + f(\mathbf{x})) = & ((y_1 \oplus \gamma_{01}) + (y_2 \oplus \gamma_{02}) + 0) \cdot ((y_1 \oplus \gamma_{11}) + (y_2 \oplus \gamma_{12}) + 1) \cdot \\
& ((y_1 \oplus \gamma_{21}) + (y_2 \oplus \gamma_{22}) + 1) \cdot ((y_1 \oplus \gamma_{31}) + (y_2 \oplus \gamma_{32}) + 0) \cdot \\
& ((y_1 \oplus \gamma_{41}) + (y_2 \oplus \gamma_{42}) + 1) \cdot ((y_1 \oplus \gamma_{51}) + (y_2 \oplus \gamma_{52}) + 0) \cdot \\
& ((y_1 \oplus \gamma_{61}) + (y_2 \oplus \gamma_{62}) + 0) \cdot ((y_1 \oplus \gamma_{71}) + (y_2 \oplus \gamma_{72}) + 1)
\end{aligned}$$

$G(\Gamma)$ can now be formed combining these two “terms” yielding:

$$\begin{aligned}
G(\Gamma) = & \forall \mathbf{y} [((y_1 \oplus \gamma_{11}) + (y_2 \oplus \gamma_{12})) \cdot ((y_1 \oplus \gamma_{21}) + (y_2 \oplus \gamma_{22})) \cdot \\
& ((y_1 \oplus \gamma_{41}) + (y_2 \oplus \gamma_{42})) \cdot ((y_1 \oplus \gamma_{71}) + (y_2 \oplus \gamma_{72})) \\
& + ((y_1 \oplus \gamma_{01}) + (y_2 \oplus \gamma_{02})) \cdot ((y_1 \oplus \gamma_{31}) + (y_2 \oplus \gamma_{32})) \cdot \\
& ((y_1 \oplus \gamma_{52}) + (y_2 \oplus \gamma_{52})) \cdot ((y_1 \oplus \gamma_{61}) + (y_2 \oplus \gamma_{62}))]
\end{aligned}$$

Quantifying out \mathbf{y} we finally have:

$$\begin{aligned}
G(\Gamma) = & ((\bar{\gamma}_{11} + \bar{\gamma}_{12}) \cdot (\bar{\gamma}_{21} + \bar{\gamma}_{22}) \cdot (\bar{\gamma}_{41} + \bar{\gamma}_{42}) \cdot (\bar{\gamma}_{71} + \bar{\gamma}_{72}) + \\
& (\bar{\gamma}_{01} + \bar{\gamma}_{02}) \cdot (\bar{\gamma}_{31} + \bar{\gamma}_{32}) \cdot (\bar{\gamma}_{52} + \bar{\gamma}_{52}) \cdot (\bar{\gamma}_{61} + \bar{\gamma}_{62})) \\
& \cdot ((\bar{\gamma}_{11} + \gamma_{12}) \cdot (\bar{\gamma}_{21} + \gamma_{22}) \cdot (\bar{\gamma}_{41} + \gamma_{42}) \cdot (\bar{\gamma}_{71} + \gamma_{72}) + \\
& (\bar{\gamma}_{01} + \gamma_{02}) \cdot (\bar{\gamma}_{31} + \gamma_{32}) \cdot (\bar{\gamma}_{52} + \gamma_{52}) \cdot (\bar{\gamma}_{61} + \gamma_{62})) \\
& \cdot ((\gamma_{11} + \bar{\gamma}_{12}) \cdot (\gamma_{21} + \bar{\gamma}_{22}) \cdot (\gamma_{41} + \bar{\gamma}_{42}) \cdot (\gamma_{71} + \bar{\gamma}_{72}) + \\
& (\gamma_{01} + \bar{\gamma}_{02}) \cdot (\gamma_{31} + \bar{\gamma}_{32}) \cdot (\gamma_{52} + \bar{\gamma}_{52}) \cdot (\gamma_{61} + \bar{\gamma}_{62})) \\
& \cdot ((\gamma_{11} + \gamma_{12}) \cdot (\gamma_{21} + \gamma_{22}) \cdot (\gamma_{41} + \gamma_{42}) \cdot (\gamma_{71} + \gamma_{72}) + \\
& (\gamma_{01} + \gamma_{02}) \cdot (\gamma_{31} + \gamma_{32}) \cdot (\gamma_{52} + \gamma_{52}) \cdot (\gamma_{61} + \gamma_{62}))
\end{aligned}$$

This is a function of 1812 on-set minterms, each corresponding to a feasible 3-to-2 decomposition. However, only 99 of these define non-trivial (i.e. with no decomposition function, or its complement, corresponding to f) decomposition solutions that are invariant under complementation of the decomposition functions. This number can be further reduced by discarding solutions whose

decomposition functions have redundant signals. Some of the more interesting solutions are:

Solution A:	Solution B:	Solution C:
$g_1 = x_1 \oplus x_2$	$g_1 = \bar{x}_1 x_2 + x_1 \bar{x}_3$	$g_1 = x_1 \bar{x}_2$
$g_2 = x_3$	$g_2 = \bar{x}_1 x_3 + x_1 x_2$	$g_2 = x_1 x_3 + \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3$ ■

Example 4.4 Similarly to the previous example, application of (4.7) to $f(x_1, x_2, x_3) = \bar{x}_1 x_2 + x_1 x_3$ also yields 99 3-to-2 non-trivial decomposition solutions invariant under output complementation. We list some of them below:

Solution A:	Solution B:	Solution C:
$g_1 = x_1 + x_2$	$g_1 = \bar{x}_1 x_2$	$g_1 = \bar{x}_1 x_2$
$g_2 = x_1 x_3$	$g_2 = x_1 x_3 + x_2 x_3$	$g_2 = x_1 x_3$ ■

Equation (4.7) encompasses all the decomposition solutions for a given function f . To find the decomposition solutions for an arbitrary n -variable function f , we introduce a vector of 2^n encoding coefficients $\Phi =_{\text{df}} [\varphi_i]$ to express the universe of n -variable functions as:

$$F(\mathbf{x}, \Phi) = \sum_{i=0}^{2^n-1} \varphi_i \cdot m_i(\mathbf{x}) \quad (4.8)$$

Note that a complete assignment to Φ represents a particular completely specified function f ; partial assignments to Φ denote families of functions. Re-writing (4.7) in terms of $F(\mathbf{x}, \Phi)$ we obtain:

$$G(\Phi, \Gamma) = \forall \mathbf{y} (\overline{\exists \mathbf{x} (C(\mathbf{x}, \mathbf{y}, \Gamma) \cdot F(\mathbf{x}, \Phi))} + \forall \mathbf{x} (\overline{C(\mathbf{x}, \mathbf{y}, \Gamma)} + F(\mathbf{x}, \Phi))) \quad (4.9)$$

which is a Boolean function that encodes all feasible decomposition functions \mathbf{g} for any given function f .

Example 4.5 We can use equation (4.9) to compute 3-to-2 decompositions for all 3-variable functions. The universe of these functions can be encoded as:

$$\begin{aligned} F(\mathbf{x}, \Phi) = & \varphi_0 \cdot \bar{x}_1 \bar{x}_2 \bar{x}_3 + \varphi_1 \cdot \bar{x}_1 \bar{x}_2 x_3 + \varphi_2 \cdot \bar{x}_1 x_2 \bar{x}_3 + \varphi_3 \cdot \bar{x}_1 x_2 x_3 \\ & + \varphi_4 \cdot x_1 \bar{x}_2 \bar{x}_3 + \varphi_5 \cdot x_1 \bar{x}_2 x_3 + \varphi_6 \cdot x_1 x_2 \bar{x}_3 + \varphi_7 \cdot x_1 x_2 x_3 \end{aligned}$$

Using this encoding in (4.9) we can identify all 3-to-2 decompositions for every assignment to the Φ coefficients. Indeed, computation of $G(\Phi, \Gamma)$ shows that there are non-trivial decompositions for every function in $F(\mathbf{x}, \Phi)$ induced by the assignments to Φ coefficients. In particular, assignments [01101001] and [00110101] to the $\Phi = [\varphi_0 \varphi_1 \varphi_2 \varphi_3 \varphi_4 \varphi_5 \varphi_6 \varphi_7]$ coefficients induce the decomposition solutions computed for $x_1 \oplus x_2 \oplus x_3$ and $\bar{x}_1 x_2 + x_1 x_3$ in Example 4.3 and Example 4.4, respectively. Note that assignments to the Φ coefficients for the two functions have

the same number of 0s and 1s, implying a structural equivalence between their two functions which, in turn, explains why they have an equal number of decomposition solutions (ninety nine). ■

4.2 Enforcing Practical Decomposition Constraints

Although the decomposition template in (4.1) is very general, its computational complexity makes its use difficult for scalable synthesis. In this section we address the complexity problems by imposing practical fan-in constraints on the decomposition and composition functions. These constraints are reflected in a modified decomposition template that is used to define appropriate decomposition patterns deduced from the semantic structure of a function.

4.2.1 Effect of fan-in constraints

Equation (4.7) requires, in the worst case, the construction of a Boolean characteristic function of $2^n \cdot k$ encoding coefficients. Our first decomposition constraint, therefore, aims at reducing the exponential in n and is satisfied by requiring the support of the $\mathbf{g}(\mathbf{x})$ functions to be bounded by s , where s is the maximum allowable fan-in of the underlying implementation technology; in current CMOS processes, s is typically four. Such a requirement eliminates $k \cdot (2^n - 2^s)$ encoding coefficients from (4.6), thereby exponentially reducing its computational burden. This reduction, however, introduces an additional algorithmic component whose goal is to identify a suitable subset of s -out of- n variables from the input vector \mathbf{x} that can act as the support of the functions $\mathbf{g}(\mathbf{x})$.

Although for a fan-in of at least 2 it is always possible to find decomposition functions (e.g. trivial wire functions) which make interval (4.7) non-empty, given k decomposition functions, the number of decomposition solutions varies greatly depending on the chosen fan-in constraint s . In the worst case, given some k , such that $k < n$, the decomposition may become infeasible if s is chosen to be too small. The effect of varying the fan-in parameter s is illustrated in the example below.

Example 4.6 Suppose that we would like to decompose the function of the second sum bit in an n -bit adder into two decomposition functions. The five-input sum-bit function is given in the following factored form:

$$s_1 = a_1 \oplus b_1 \oplus (a_0 b_0 + c_0(a_0 \oplus b_0))$$

Let $C(\mathbf{x}, \mathbf{y}, \Gamma)$ encode the space of decomposition functions for $k = 2$ (as in Example 4.3). We can then compute the set of all decomposition solutions for s_1 using (4.7). Indeed, function $G(\Gamma)$, computed in (4.7), contains 1,116,591,939 non-trivial pairs of decomposition functions assuming invariance under complementation of their outputs. Restricting the fan-in of the decomposition functions to four, the number of possible solutions reduces to just 795. Further restriction to a fan-in of three yields only these four solutions:

Solution A: $g_1 = a_1 \oplus b_1$ $g_2 = c_0 a_0 + c_0 b_0 + a_0 b_0$	Solution B: $g_1 = b_0 \oplus a_1 \oplus b_1$ $g_2 = c_0 a_0 \bar{b}_0 + \bar{c}_0 \bar{a}_0 b_0$
Solution C: $g_1 = a_0 \oplus a_1 \oplus b_1$ $g_2 = \bar{c}_0 a_0 \bar{b}_0 + c_0 \bar{a}_0 b_0$	Solution D: $g_1 = c_0 \oplus a_1 \oplus b_1$ $g_2 = \bar{c}_0 a_0 b_0 + c_0 \bar{a}_0 \bar{b}_0$

■

In general, the number of decomposition functions k required to make (4.7) non-empty increases significantly as the fan-in constraint becomes stricter. Indeed, the existence of a decomposition according to (4.7) is a function of s and k according to the following table:

s	k	existence of decomposition
$\geq n$	$\geq n$	exists
$\geq n$	$< n$	exists
$< n$	$\geq n$	exists
$< n$	$< n$	may exist

(4.10)

Existence of decomposition in the first three cases is easily demonstrated by choosing trivial decompositions. The fourth case in the table suggests that restricting the values of s and k to be smaller than the support of the function f may preclude its decomposition.

To reflect a fan-in bound of s on the decomposition functions, we modify the generic decomposition template in (4.1) to become:

$$f(\mathbf{x}) = h(g_1(\mathbf{x}_1), \dots, g_k(\mathbf{x}_k)) \quad (4.11)$$

where the \mathbf{x}_i 's are composed from s -subsets of \mathbf{x} . When the decomposition variables \mathbf{x}_i for each g_i are given we can use the computational form in (4.7) to find all feasible sets of k decomposition functions. Solving (4.7) for large k , however, is still computationally expensive even for small fan-in bounds. In the next section we show how the number of Γ coefficients can be reduced further by modifying the decomposition template.

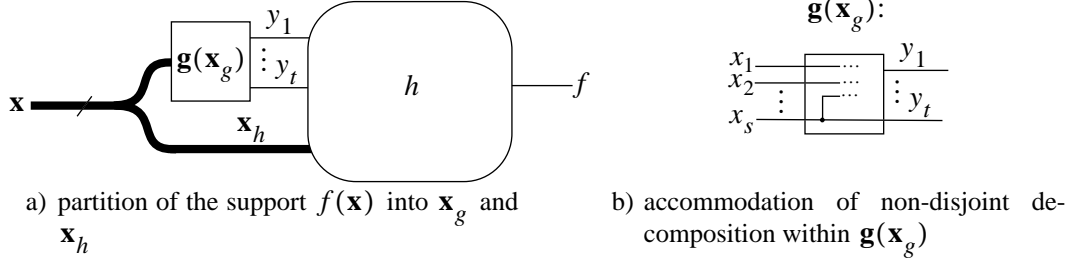


Figure 4.2: Illustration of the decomposition template $f(\mathbf{x}) = h(\mathbf{g}(\mathbf{x}_g), \mathbf{x}_h)$

4.2.2 Modified decomposition template

The template in (4.11) imposes a fan-in bound of s on the decomposition functions, but allows them to have different supports. If, instead, we require all k decomposition functions to have the same support we obtain a new decomposition template as shown in Fig. 4.2. This template partitions the input variables into two sets, \mathbf{x}_g and \mathbf{x}_h , with $|\mathbf{x}_g| = s$ (see Fig. 4.2-a):

$$f(\mathbf{x}_g, \mathbf{x}_h) = h(g_1(\mathbf{x}_g), \dots, g_t(\mathbf{x}_g), \mathbf{x}_h) \quad (4.12)$$

The functions $g_1(\mathbf{x}_g), \dots, g_t(\mathbf{x}_g)$, or collectively $\mathbf{g}(\mathbf{x}_g)$, can be assumed to be library primitives, possibly pass-through wires. Within this template, the generality of the decomposition in (4.11) is still preserved if we change the algorithmic flow in which the equation is solved: instead of simultaneously finding all k decomposition functions for which (4.11) holds, using (4.12) we find them iteratively over subsets of \mathbf{x} whose size is bounded by s .¹ A particular solution for each of the subsets presents us with a collection of decomposition functions, which can be viewed as a multi-output module. An encoding of a complete set of such solutions uses at most $t \cdot 2^s$ Γ coefficients.

While the structure in Fig. 4.2-a appears to correspond to disjoint decomposition [92], allowing pass-through wires as decomposition functions effectively accommodates, through the iterative application of (4.12), non-disjoint decompositions as well. This is illustrated in Figure 4.2-b, where signal x_s is used not only in the non-trivial functions of $\mathbf{g}(\mathbf{x}_g)$ but also as the trivial identity function $g_t(\mathbf{x}_g) = x_s$. This wire can, therefore, be reused in subsequent decomposition steps. The effect of such reuse preserves the full generality of the decomposition template (4.11).

Example 4.7 Consider the following initial decomposition of $f(a, b, c, d) = ac + \bar{a}d + b\bar{d} + \bar{b}d$:

1. Of course, in this new flow, the output signals y_i of newly-introduced decomposition functions become immediately available for subsequent decompositions; they become part of the \mathbf{x} vector.

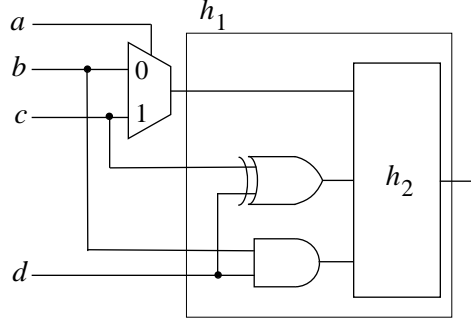


Figure 4.3: Circuit resulting from two successive decomposition steps; it has two topological levels of logic

$$f(a, b, c, d) = h_1(\bar{a}b + ac, b, c, d)$$

This decomposition conforms to template (4.12) with $\mathbf{x}_g = \{a, b, c\}$ and

$$\begin{aligned} g_1(a, b, c) &= \bar{a}b + ac \\ g_2(a, b, c) &= b \\ g_3(a, b, c) &= c \end{aligned}$$

Note that the use of identity decomposition functions, yields a composition function $h_1(y_1, b, c, d)$ whose support overlaps with the original decomposition variables $\mathbf{x}_g = \{a, b, c\}$. Thus, in the next decomposition iteration, function $h_1(y_1, b, c, d)$ can be decomposed using a new set of decomposition variables $\mathbf{x}_g = \{b, c, d\}$ that overlap the original set yielding an overall structure which is non-disjoint. Specifically, $h_1(y_1, b, c, d) = h_2(y_1, b \oplus d, cd)$ with

$$\begin{aligned} g_1(b, c, d) &= b \oplus d \\ g_2(b, c, d) &= cd \end{aligned}$$

The circuit schematic resulting from these two decomposition steps is depicted in Figure 4.3 and clearly reflects a non-disjoint decomposition in inputs b and c of the original function. ■

Using the new decomposition template (4.12), determination of all feasible decomposition functions can be carried out similarly to the procedure that led to (4.7). In particular, the direct fans of the composition function h are now represented by the vector $\langle \mathbf{x}_h, \mathbf{y} \rangle$ which plays the role of \mathbf{y} in the original template. We can, therefore, readily express the set of all feasible decomposition functions by inspection as:

$$G(\Gamma) = \forall \mathbf{x}_h \forall \mathbf{y} (\exists \mathbf{x}_g (\overline{C(\mathbf{x}_g, \mathbf{y}, \Gamma)} \cdot f(\mathbf{x})) + \forall \mathbf{x}_g (\overline{C(\mathbf{x}_g, \mathbf{y}, \Gamma)} + f(\mathbf{x}))) \quad (4.13)$$

Note that the care set in the above equation is written as a function of just the decomposition variables \mathbf{x}_g and the outputs of the decomposition functions \mathbf{y} . This is justified by the fact that the

care set corresponding to the \mathbf{x}_h wires is really a trivial constraint ($\mathbf{y}_h = \mathbf{x}_h$) that is handled by a simple re-labeling (i.e. using \mathbf{x}_h as an alias for \mathbf{y}_h).

The computational effort in (4.13) can be further reduced by noting that the partition of the input variables into two sets allows $f(\mathbf{x})$ to be written as an orthonormal expansion with respect to the decomposition variables:

$$f(\mathbf{x}_g, \mathbf{x}_h) = \sum_{i=0}^{2^s-1} m_i(\mathbf{x}_g) \cdot f_i(\mathbf{x}_h) \quad (4.14)$$

In this expansion, the minterms on \mathbf{x}_g serve as the orthonormal basis, and expansion coefficients $f_i(\mathbf{x}_h)$ are cofactors with respect to \mathbf{x}_g which are functions of just \mathbf{x}_h , i.e. they are vacuous in \mathbf{x}_g (cf. Lemma 2.1). In general, the $f_i(\mathbf{x}_h)$ cofactors can be arbitrarily complex functions of the \mathbf{x}_h variables. However, their significance in the decomposition lies in their relation to each other rather than their individual dependence on \mathbf{x}_h . The dependence on \mathbf{x}_h can, therefore, be abstracted away allowing us to replace the cofactors by a set of coefficients $Z =_{\text{df}} [\zeta_i]$ such that the same coefficient is associated with identical cofactors. Thus, for any value of n , we encode a particular n -variable function using at most 2^s encoding coefficients allowing us to re-write (4.14) as:

$$F(\mathbf{x}_g, Z) = \sum_{i=0}^{r-1} M_i(\mathbf{x}_g) \cdot \zeta_i \quad (4.15)$$

where Z is a vector of r independent encoding coefficients ($1 \leq r \leq 2^s$), and $M_i(\mathbf{x}_g)$ denotes a maximal set of minterms on \mathbf{x}_g with identical cofactors. In other words, the on-sets of the $M_i(\mathbf{x}_g)$ functions are equivalence classes induced by the equality relation of cofactors. It is interesting to note that the $(s+r)$ -variable function $F(\mathbf{x}_g, Z)$ in (4.15) represents an infinite class of functions $f(\mathbf{x}_g, \mathbf{x}_h)$ since each coefficient ζ_i corresponds to a cofactor which is a function of potentially an arbitrarily large number of variables \mathbf{x}_h . We will refer to $F(\mathbf{x}_g, Z)$ as a *pattern function*.

Substituting $F(\mathbf{x}_g, Z)$ for $f(\mathbf{x})$ in (4.13) we finally obtain the following efficient computational form for all feasible decompositions of $f(\mathbf{x})$:

$$G(\Gamma) = \forall Z \forall \mathbf{y} (\overline{\exists \mathbf{x}_g (C(\mathbf{x}_g, \mathbf{y}, \Gamma) \cdot F(\mathbf{x}_g, Z))} + \forall \mathbf{x}_g (\overline{C(\mathbf{x}_g, \mathbf{y}, \Gamma)} + F(\mathbf{x}_g, Z))) \quad (4.16)$$

Note that in addition to replacing $f(\mathbf{x})$ by $F(\mathbf{x}_g, Z)$, we must also replace the abstraction of the \mathbf{x}_h variables in (4.13) with the abstraction of the Z coefficients. Note also that equations (4.13) and (4.16) compute identical sets of feasible decomposition solutions; the computation in (4.16), however, is much more efficient as it involves the abstraction a set of coefficients whose size is

bounded by 2^s , as opposed to (4.13) which involves the abstraction of $n - s$ variables for arbitrarily large n .

Example 4.8 The orthonormal expansion of the function

$$f(a, b, c, d, e) = a\bar{b}d + a\bar{e} + adc + b\bar{e}d + \bar{b}c\bar{e}$$

with respect to $\{a, b, c\}$ is easily shown to be

$$\begin{aligned} f(a, b, c, d, e) = & \bar{a}\bar{b}\bar{c} \cdot \mathbf{0} + (a\bar{b}\bar{c} + a\bar{b}c + abc) \cdot (d + \bar{e}) + \\ & (\bar{a}b\bar{c} + \bar{a}bc) \cdot (d\bar{e}) + (ab\bar{c} + a\bar{b}c) \cdot (\bar{e}) \end{aligned}$$

Thus, we can represent f by the pattern function

$$\begin{aligned} F(a, b, c, \zeta_0, \zeta_1, \zeta_2, \zeta_3) = & \bar{a}\bar{b}\bar{c} \cdot \zeta_0 + (a\bar{b}\bar{c} + a\bar{b}c + abc) \cdot \zeta_1 + \\ & (\bar{a}b\bar{c} + \bar{a}bc) \cdot \zeta_2 + (ab\bar{c} + a\bar{b}c) \cdot \zeta_3 \end{aligned}$$

where $\zeta_0, \zeta_1, \zeta_2$ and ζ_3 are independent encoding coefficients.

To find all feasible 3-to-2 decompositions of this function, we first encode the two 3-input decomposition functions g_1 and g_2 with sixteen coefficients $\gamma_{01}, \dots, \gamma_{71}$ and $\gamma_{02}, \dots, \gamma_{72}$, respectively. The corresponding care set can now be expressed as

$$\begin{aligned} C(\mathbf{x}_g, \mathbf{y}, \Gamma) = & \\ (y_1 \equiv & (\gamma_{01}\bar{a}\bar{b}\bar{c} + \gamma_{11}\bar{a}\bar{b}c + \gamma_{21}\bar{a}b\bar{c} + \gamma_{31}\bar{a}bc + \gamma_{41}a\bar{b}\bar{c} + \gamma_{51}a\bar{b}c + \gamma_{61}ab\bar{c} + \gamma_{71}abc)) \cdot \\ (y_2 \equiv & (\gamma_{02}\bar{a}\bar{b}\bar{c} + \gamma_{12}\bar{a}\bar{b}c + \gamma_{22}\bar{a}b\bar{c} + \gamma_{32}\bar{a}bc + \gamma_{42}a\bar{b}\bar{c} + \gamma_{52}a\bar{b}c + \gamma_{62}ab\bar{c} + \gamma_{72}abc)) \end{aligned}$$

which, together with the pattern function derived earlier, are now substituted in (4.16) to give us the complete expression for $G(\Gamma)$. Computation of the lower bound term of (4.16) has the form:

$$\begin{aligned} \exists \mathbf{x}_g (C(\mathbf{x}_g, \mathbf{y}, \Gamma) \cdot F(\mathbf{x}_g, Z)) = & (y_1 \equiv \gamma_{01}) \cdot (y_2 \equiv \gamma_{02}) \cdot \zeta_0 + (y_1 \equiv \gamma_{11}) \cdot (y_2 \equiv \gamma_{12}) \cdot \zeta_3 \\ & + (y_1 \equiv \gamma_{21}) \cdot (y_2 \equiv \gamma_{22}) \cdot \zeta_2 + (y_1 \equiv \gamma_{31}) \cdot (y_2 \equiv \gamma_{32}) \cdot \zeta_2 \\ & + (y_1 \equiv \gamma_{41}) \cdot (y_2 \equiv \gamma_{42}) \cdot \zeta_1 + (y_1 \equiv \gamma_{51}) \cdot (y_2 \equiv \gamma_{52}) \cdot \zeta_1 \\ & + (y_1 \equiv \gamma_{61}) \cdot (y_2 \equiv \gamma_{62}) \cdot \zeta_3 + (y_1 \equiv \gamma_{71}) \cdot (y_2 \equiv \gamma_{72}) \cdot \zeta_1 \end{aligned}$$

The upper bound term is computed analogously yielding:

$$\begin{aligned} \forall \mathbf{x}_g (\overline{C(\mathbf{x}_g, \mathbf{y}, \Gamma)} + F(\mathbf{x}_g, Z)) = & ((y_1 \oplus \gamma_{01}) + (y_2 \oplus \gamma_{02}) + \zeta_0) \cdot ((y_1 \oplus \gamma_{11}) + (y_2 \oplus \gamma_{12}) + \zeta_3) \\ & \cdot ((y_1 \oplus \gamma_{21}) + (y_2 \oplus \gamma_{22}) + \zeta_2) \cdot ((y_1 \oplus \gamma_{31}) + (y_2 \oplus \gamma_{32}) + \zeta_2) \\ & \cdot ((y_1 \oplus \gamma_{41}) + (y_2 \oplus \gamma_{42}) + \zeta_4) \cdot ((y_1 \oplus \gamma_{52}) + (y_2 \oplus \gamma_{52}) + \zeta_1) \\ & \cdot ((y_1 \oplus \gamma_{61}) + (y_2 \oplus \gamma_{62}) + \zeta_3) \cdot ((y_1 \oplus \gamma_{71}) + (y_2 \oplus \gamma_{72}) + \zeta_1) \end{aligned}$$

Substitution of these two terms and universal abstraction of the four Z coefficients and two \mathbf{y} variables finally yields the desired sixteen-coefficient function $G(\Gamma)$ that encodes all feasible decomposition functions. The BDD for this function is shown in Figure 4.4. The function has a total of

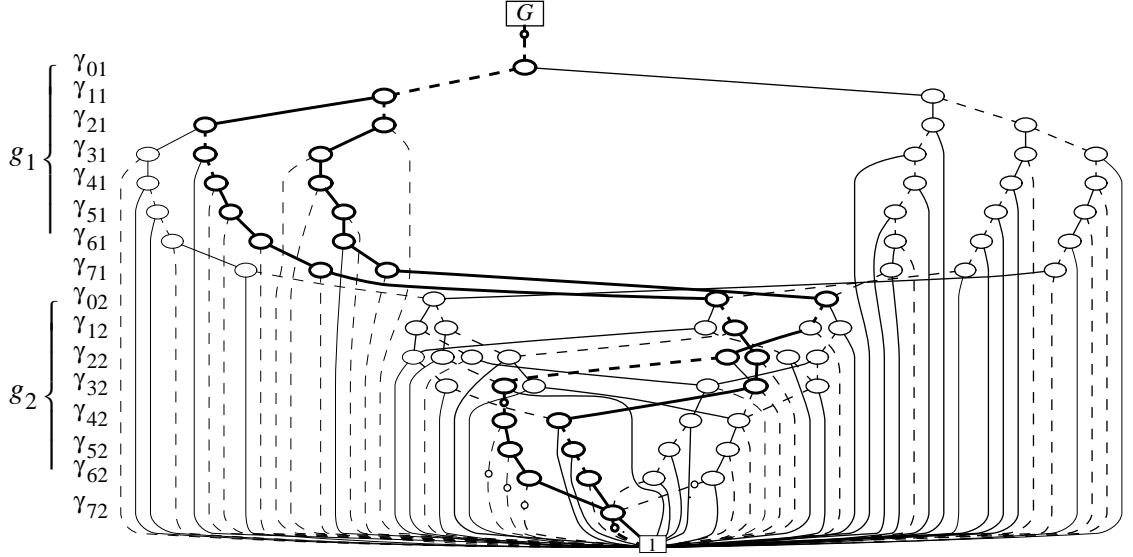


Figure 4.4: BDD for $G(\Gamma)$ which encodes all 3-to-2 decomposition functions for $f = (a\bar{b}d + a\bar{e} + adc + b\bar{e}d + \bar{b}c\bar{e})$ of Example 4.8. (Inversion bubbles on graph edges denote complementation of the functions associated with their rooted subgraphs.)

24 on-set minterms, each corresponding to a pair of possible decomposition functions. Assuming order-invariance of the decomposition functions, and invariance of their complements, the number of solutions reduces to just three:

Solution A:	Solution B:	Solution C:
$g_1 = a\bar{b} + \bar{a}b + bc$	$g_1 = a\bar{b} + \bar{a}b + bc$	$g_1 = a + \bar{b}c$
$g_2 = \bar{a}b + b\bar{c} + \bar{a}c$	$g_2 = a + \bar{b}c$	$g_2 = \bar{a}b + b\bar{c} + \bar{a}c$

These three solution are highlighted on the BDD of Figure 4.4. ■

4.2.3 Effect of support constraints

The introduction of a fan-in constraint for the decomposition functions enabled us to define decomposition according to (4.12), a computationally efficient, yet very general, template. The generality of this template, however, also leads to a great variety of decomposition patterns that become available during the synthesis process. In addition to the fan-in bound s we can classify these decomposition patterns according to the number of decomposition functions t used during a decomposition step. The decomposition template (4.12) establishes a strong relation between these two parameters in the sense that their relative values determine the degree of flexibility available for selecting the composition function h . Indeed, letting t be smaller than s may imply the non-existence of h , and therefore the non-existence of a decomposition. On the other hand, letting t to be equal to or larger than s may present us with a vast number of choices for h , making it difficult

to find a good composition function which improves synthesis quality. The former choice, i.e. $t < s$, leads to a support-reducing decomposition, whereas the latter choice, i.e. $t \geq s$, yields support-maintaining or increasing decompositions.

Support-reducing decomposition defines a class of circuit structures whose width, for each output, decreases monotonically as the circuit is traversed forward from the primary inputs. From a synthesis perspective, this is particularly attractive since each decomposition step guarantees a reduction in the complexity of the remaining unimplemented logic as measured by the cardinality of its input signals. As mentioned earlier, however, support reduction using fan-in limited decomposition functions places a restriction on the class of functions that can be decomposed according to (4.12). Specifically, the existence of an s -to- t decomposition requires that the number of distinct cofactors induced by the minterm space of the decomposition variables be $\leq 2^t$. Such a requirement is easily justified by an information theoretic argument: t binary-valued output lines can distinguish at most 2^t equivalence classes of input patterns. This counting argument has been used extensively in the classical theory for disjoint decomposition, and underlies the notion of column multiplicity in partition tables [35]. The existence of a support-reducing decomposition, thus, requires that the number of distinct cofactors of the function $f(\mathbf{x})$ in (4.12) be at most 2^{s-1} .

Example 4.9 We use the function $f(a, b, c, d, e) = a\bar{b}d + a\bar{e} + adc + b\bar{e}d + \bar{b}c\bar{e}$ from Example 4.8 to illustrate how the existence of decomposition depends on the choice of decomposition variables. The three decomposition variables in the earlier example induce four distinct cofactors 0 , $d + \bar{e}$, $d\bar{e}$, and \bar{e} , confirming the existence of a 3-to-2 reduction. On the other hand, selecting a, b, e as the decomposition variables yields 5 distinct cofactors (0 , 1 , d , c , and dc) implying the non-existence of a 3-to-2 decomposition in those variables. ■

The number of distinct cofactors induced by s decomposition variables is bounded by $\min(2^s, 2^{2^{n-s}})$. This bound is easy to establish by noting that:

- the orthonormal expansion in (4.14) has 2^s terms, each possibly corresponding to a distinct cofactor function; and
 - the universe of the $(n-s)$ -variable cofactor functions in (4.14) has $2^{2^{n-s}}$ functions.
- As s approaches n this number eventually becomes smaller than 2^s .

Figure 4.5 illustrates the relationship between the number of distinct cofactors and the number of decomposition variables. The shaded areas in the figure represent the number of possible cofactors that a function can have in s variables. For small s , this number is bounded by 2^s ; for large

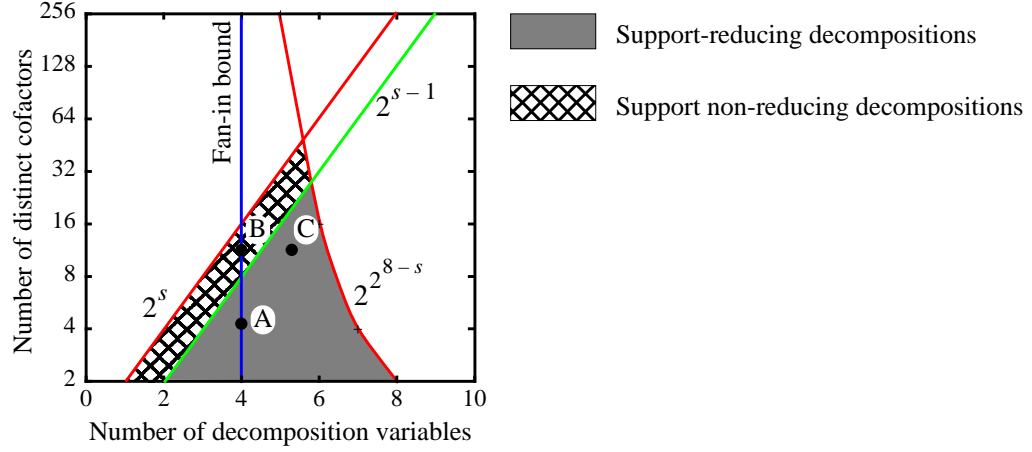


Figure 4.5: Bounds on the number of distinct cofactors as a function of the number of decomposition variables for $n = 8$

s , it is bounded by $2^{2^{n-s}}$. The number of distinct cofactors in the cross-hatched and dark-shaded regions is, respectively, greater than and less than or equal to 2^{s-1} . Thus, if the number of distinct cofactors for a given function falls in the cross-hatched region, support-reducing decomposition would not be possible. If, on the other hand, that number falls in the dark-shaded region, then support reduction is guaranteed.

In the synthesis scenario suggested by (4.12), we assume that we are given a relatively small fan-in bound s (say 3 or 4) that reflects a technological requirement (e.g. maximum number of transistors that can be stacked). If the number of distinct cofactors for our function at this fan-in bound is less than 2^{s-1} (point A in the figure), then it is possible to perform a support-reducing decomposition step. If the function has more distinct cofactors at that bound (point B), then support reduction cannot be accomplished. It must be noted, however, that support reduction can always be achieved if the fan-in bound is relaxed (e.g. by moving horizontally from point B to point C).

To illustrate the above concepts with some concrete examples, consider the decomposition possibilities for the 8th product bit of a 6-bit multiplier and the 6th sum bit of an n -bit adder shown in Figure 4.6. In this figure, cofactors are computed for each possible subset of s decomposition variables ($1 \leq s \leq 12$). The number of distinct cofactors are then determined and their minimum and maximum are plotted against s . The 2^{s-1} support-reducing threshold line is also plotted.

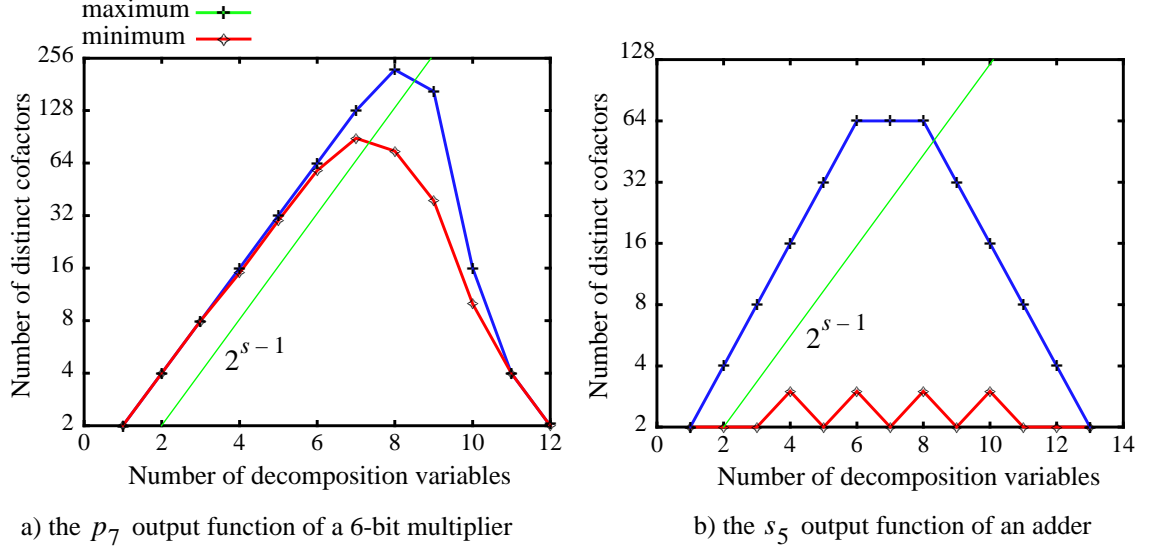


Figure 4.6: Dependence of the distinct cofactor counts on the number of decomposition variables

As the multiplier function plot in part (a) clearly shows, support reduction is not possible for $s \leq 7$. Indeed, the support-reducing threshold line intersects the “minimum distinct cofactors” curve at some s between 7 and 8. Thus support reduction can only be achieved by using groups of 8 or more decomposition variables. Note that this exceeds half the number of inputs. In general, multiplier functions do not simultaneously admit a fan-in bounded and support-reducing decomposition.

The adder function plot in part (b), on the other hand, has a completely different character. The gap between the minimum and maximum number of distinct cofactors at each value of s is much wider than in the case of the multiplier function. In addition, support-reducing decompositions are possible for any value of $s \geq 2$. These results hold regardless of the adder size.

An analysis of the MCNC benchmarks [123] shows that the majority of their functions admit support-reducing decompositions for small values of s , usually less than five. If the number of decomposition variables required to achieve support-reduction is larger than five, then the function is likely non-symmetric and monotone in most of its variables. Existence of a support-reducing decomposition in a typical function from this suite of benchmarks is very sensitive to the choice of decomposition variables, necessitating their careful selection during the decomposition process.

Finally, it is interesting to note that the support-reduction requirement, in the presence of a fan-in bound, establishes a direct link between the structure of a function (namely the number of its distinct cofactors in a subset of its inputs) and the structure of an implementation of that function

(namely the strict reduction in topological width). This ability to reflect functional structure in the resulting implementation topology is one of the distinctive features of our synthesis flow. As we'll see in Chapter 7, this feature is one of the major factors that lead to improved overall synthesis quality.

4.3 Computational Considerations

The core computation of decomposition solutions in (4.16) involves two nearly identical terms. In this section we explore some optimizations that yield further computational savings in the process of solving for feasible decompositions.

4.3.1 Exploring common computational core

It is possible to simplify the computation of (4.16) by observing that the terms corresponding to the lower and upper bounds in the formula can be simplified, and by noting that they also have virtually identical structure. The first step of the simplification multiplies out the two functions corresponding to the interval's lower bound. Let vector γ_i denote the i^{th} row of coefficients in the $\Gamma = [\gamma_{ij}]$ matrix, and let function $\sigma(i)$ identify the index of a coefficient in Z corresponding to a cofactor $f_i(\mathbf{x}_h)$. We then have the following result:

Theorem 4.1

$$G(\Gamma) = \forall Z \forall \mathbf{y} \left(\overline{\exists \mathbf{x}_g \left(\sum_{i=0}^{2^s-1} m_i(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_i) \cdot \zeta_{\sigma(i)} \right)} + \overline{\exists \mathbf{x}_g \left(\sum_{i=0}^{2^s-1} m_i(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_i) \cdot \bar{\zeta}_{\sigma(i)} \right)} \right)$$

The two summations in the above formula are identical with the exception of complemented Z coefficients, suggesting that during the computation of $G(\Gamma)$ one summand can be constructed from the other by a simple complementation of Z coefficients.

There is also another, more significant, observation that follows directly from the structure of the computational form in Theorem 4.1, which allows further reduction of G 's computational effort. It follows from the fact that the difference in the computation of the $G(\Gamma)$ for any two s -to- t problem instances is attributed only to the function $\sigma(i)$. Thus, it has a computational core common to all problem instances. This observation suggests that we can compute a generic form $G(\Gamma, Z)$ where (for example) $\sigma(i) = i$, and then obtain a $G(\Gamma, Z)$ that is specific to a given problem instance by appropriately remapping the Z coefficients. The abstraction of the Z coeffi-

cients can then be applied to obtain an encoding of all feasible decomposition functions for the problem instance. The example below illustrates the significance of the $\sigma(i)$ function in $G(\Gamma, Z)$.

Example 4.10 We can view $G(\Gamma, Z)$ as an intermediate result when computing $G(\Gamma)$ using (4.16). For the function of Example 4.8 we may write $G(\Gamma, Z)$ as:

$$G(\Gamma, Z) = \prod_{k=0}^3 \left[\prod_{i=0}^7 ((\gamma_{i1} \equiv \phi_0(k)) + (\gamma_{i2} \equiv \phi_1(k)) + \bar{\zeta}_{\sigma(i)} + \prod_{i=0}^7 ((\gamma_{i1} \equiv \phi_0(k)) + (\gamma_{i2} \equiv \phi_1(k)) + \zeta_{\sigma(i)}) \right]$$

where $\phi_i(k)$ is a function returning the value of the i th bit of an integer k , and $\sigma(i)$ is a function remapping integer i into some other integer value. One may easily observe that the only variant in the above $G(\Gamma, Z)$ is function $\sigma(i)$. Intuitively, $\sigma(i)$ tells us what equivalence classes (i.e. factors) minterm $m_i(\mathbf{x}_g)$ belongs to. In our example the following set of tuples defines this function:

$$\{\langle 0, 0 \rangle, \langle 1, 3 \rangle, \langle 2, 2 \rangle, \langle 3, 2 \rangle, \langle 4, 1 \rangle, \langle 5, 1 \rangle, \langle 6, 3 \rangle, \langle 7, 1 \rangle\}$$

Note that the function is consistent with the pattern function $F(a, b, c, \zeta_0, \zeta_1, \zeta_2, \zeta_3)$ in Example 4.8; e.g., minterms 2 and 3 are associated with coefficient ζ_2 . ■

4.3.2 Dealing with large sets of decomposition variables

As we have seen in Section 4.1.3 the number of Γ coefficients needed to encode the decomposition functions grows exponentially in the number of decomposition variables. We can eliminate this exponential dependence by assuming that:

1. the number of decomposition functions t is small, and
2. not all of the feasible decomposition solutions need to be computed

These two assumptions allow us to obtain a constrained form for the computation of function $G(\Gamma)$ that uses fewer Γ coefficients.

To obtain the reduction in the number of Γ coefficients, we begin by adopting expansion (4.6) for the modified decomposition template (4.12); we re-write it as

$$C(\mathbf{x}_g, \mathbf{y}, \Gamma) = \sum_{i=0}^{2^s-1} [m_i(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_i)] \quad (4.17)$$

Each vector γ_i in the above formula denotes the i th row of coefficients in the $\Gamma = [\gamma_{ij}]$ matrix; operation \equiv in (4.17) is performed bit-wise. Recalling the discussion in Section 4.1.3 on page 56,

an assignment to the Γ coefficients in $C(\mathbf{x}_g, \mathbf{y}, \Gamma)$ maps each minterm $m_i(\mathbf{x}_g)$ into a point $\hat{\mathbf{y}}$ from the minterm space of \mathbf{y} . The number of Γ coefficients used by this mapping is $t \cdot 2^s$, where s is the number of variables in \mathbf{x}_g . However, we may obtain a substantial reduction in the Γ coefficients by replacing each minterm function $m_i(\mathbf{x}_g)$ with a factor $M_i(\mathbf{x}_g)$ whose on-set forms a subset of an equivalence class. This way, instead of mapping a single minterm to a point in the space of \mathbf{y} , a subset of minterms gets mapped. The mapping is reflected in the following generalization of (4.17):

$$C(\mathbf{x}_g, \mathbf{y}, \Gamma) = \sum_{i=0}^r [M_i(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_i)] \quad (4.18)$$

As such, the number of Γ coefficients used in (4.18) is reduced to $t \cdot r$. Thus, the new form of $C(\mathbf{x}, \mathbf{y}, \Gamma)$ suggests an efficient computation of subsets of feasible decomposition functions \mathbf{g} .

The efficient computation of $G(\Gamma)$ using (4.18) is supported by our two assumptions. The first assumption leads to a small t . The second assumption, on the other hand, helps to keep r small. Indeed, by letting the on-set of each factor $M_i(\mathbf{x}_g)$ be an equivalence class we ensure that the $r \leq 2^t$ relation is satisfied. Whenever the s -to- t reduction is feasible the computed $G(\Gamma)$ encodes a non-empty set of decomposition functions.

4.4 Summary

In this chapter we have studied the decomposition choices available in constructive synthesis. We began with the generic decomposition template which allows all implementations of a function. The template was then modified to accommodate simple, yet effective, support-reducing and fan-in-bounded practical constraints. The modified template was linked to the semantic properties of Boolean functions. The analysis of the available decomposition choices was performed symbolically enabling us to analyze them without being encumbered by particular functional representations.

Chapter 5

Semantic Structure of Boolean Functions

The subject we explore in this chapter is motivated by our earlier argument that when guided by knowledge of the semantic structure of a function, synthesis can yield more “natural” implementations of the function. We view the symmetry of a function as one way to capture its semantic structure. Our main objective is therefore to obtain efficient computational methods for symmetry identification. We propose a generalization of classical symmetry that allows for the simultaneous swap of groups of variables and we show that it captures more of a function’s invariant permutations with only a modest increase in computational requirements. We apply the new symmetry definition to the analysis of a large set of benchmark circuits and provide extensive data showing the existence of substantial symmetries in those circuits.

5.1 Introduction and Prior Work

Symmetries usually refer to permutations of an object’s parameters that leave it unchanged. They provide insights into the structure of the object that can be used to facilitate computations on it. They can also serve as a guide for preserving that structure when the object is transformed in some way. The object we study here is an n -variable Boolean function and the symmetries we explore are variable permutations, with possible complementation, that leave the function unchanged (see Figure 5.1).

The study of symmetries in Boolean functions dates back to Shannon [105] who recognized that symmetric functions have particularly efficient switch network implementations. Since then several attempts were made to devise synthesis procedures for symmetric functions [42, 67]. These

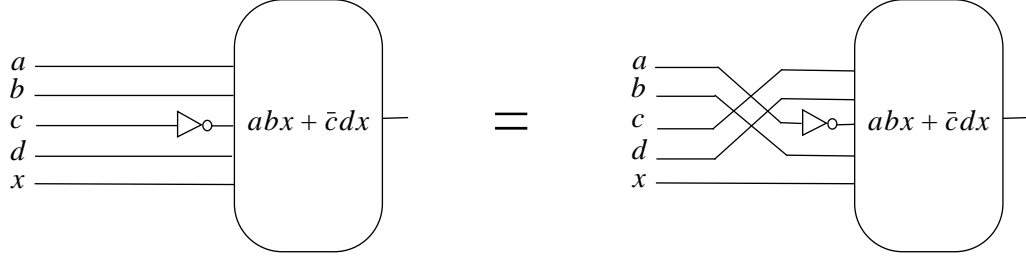


Figure 5.1: Illustration of function symmetry

efforts, however, failed to yield practical synthesis tools and are generally viewed as inapplicable to the types or sizes of functions typically encountered in today's design automation environments. In recent years, the increasing use of BDDs for the manipulation of Boolean functions has sparked renewed interest in the study of function symmetries. In [64], for example, the authors showed that the size of a BDD can be reduced by using a variable order that places symmetric variables contiguously. This observation led to the development of sifting procedures for dynamic BDD variable ordering based on function symmetries [89, 102]. Symmetries were also utilized to improve the efficiency of functional equivalence checking for functions with unknown input correspondence [32, 85], and in the context of Boolean matching [60, 77, 115].

Much of the existing literature on symmetry is based on function invariance under swaps of variable pairs in the function's support; we'll refer to this type of symmetry as *classical symmetry* to distinguish it from the more general symmetry we describe in this chapter. For completely-specified functions, classical symmetry can be represented as a partition on the set of variables: variables that belong to a given block of that partition are equivalent, i.e. symmetric, whereas variables that belong to different blocks are non-equivalent, i.e. non-symmetric. The blocks of such a partition are commonly referred to as the function's *symmetry groups*, and variables within a symmetry group are equivalent in the sense that they can be permuted arbitrarily without changing the value of the function. The advent of BDDs led to the development of efficient symbolic methods for the identification of a function's symmetry groups. The computational core in such algorithms is the check that determines the equivalence of a pair of variables; larger symmetry groups are then built incrementally using transitivity. Many of the recently-proposed techniques for symmetry identification achieve their efficiency through careful analysis of the structure of the BDD that represents the function [86, 89, 102]. In [117], the authors approach this problem by using the generalized Reed-Muller transform to speed-up the computation of symmetries. A notable exception to the commonly-used definition of symmetry was proposed in [85]. Rather than invariance under swaps of variables, symmetry is defined in terms of equivalence among arbitrary subspaces (i.e. cofac-

tors) of the function.

We define symmetry as an invariance under *arbitrary* variable permutation rather than invariance under swaps of variable pairs. Under this broader definition, partitions on the set of variables fail to capture all the invariant input permutations. We explore the relation between symmetry groups and variable permutations in Section 5.2 and highlight the inherent limitations of variable partitions as a means of representing arbitrary variable permutations. As an alternative to the explicit, and computationally infeasible, listing of all invariant permutations, we propose an efficient hierarchical extension to the notion of symmetry groups—a *hierarchical partition*—that allows us to represent a larger (but not necessarily the complete) set of invariant variable permutations. These hierarchical partitions represent *higher-order* symmetries that arise from simultaneously swapping groups of, rather than single, variables. In Section 5.3 we formally state the conditions under which the classical *first-order* symmetries exist and provide computational procedures for the construction of the corresponding flat partition. In Section 5.4 we generalize these conditions to define the higher-order symmetry and show how the corresponding hierarchical partition can be computed efficiently. In Section 5.5 we expand the notion of invariance to include the assignment of inversion phases to the function inputs. In Section 5.6 we report on the results of applying hierarchical partitioning to a large set of benchmarks; we also provide detailed analyses of a few benchmarks to show that additional symmetries, missed by hierarchical partitioning or hidden through netlist flattening, can still be found. We conclude in Section 5.7 by recapping the main contributions and suggesting several possible extensions.

5.2 Symmetry as Permutations

Consider the six-variable function $f(a, b, c, d, x, y) = abxy + cdx$. It is relatively straightforward to show that its classical symmetry groups are $\{a, b\}$, $\{c, d\}$, and $\{x, y\}$. (The exact procedure for computing these groups is described in Section 5.3.) To appreciate the need for a broader notion of symmetry it is useful to view these symmetry groups as an *implicit* representation of the variable permutations that leave the function unchanged. Specifically, a group G_i consisting of n_i variables corresponds to $n_i!$ permutations; the total number of permutations represented by all groups is the product of the number of permutations for each of the individual groups. Thus, the three-group partition on the variables of this function corresponds to the eight ($2! \times 2! \times 2!$) permutations:

$$\{\langle abcdxy \rangle, \langle abcdyx \rangle, \langle abdcxy \rangle, \langle abdcyx \rangle, \langle bacdx y \rangle, \langle bacdyx \rangle, \langle badcxy \rangle, \langle badcyx \rangle\} \quad (5.1)$$

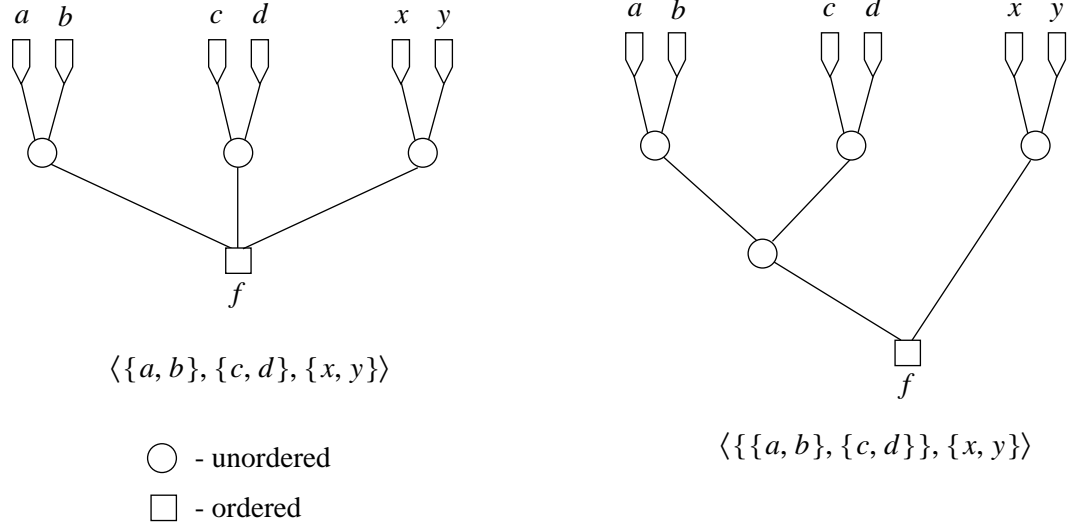


Figure 5.2: Symmetry induced hierarchical partition of variables for function $f(a, b, c, d, x, y) = abxy + cdxy$

Direct substitution of each of these permutations in the expression for the function confirms that they do indeed leave it unchanged. We will refer to such permutations as the function's *invariant permutations*. In addition, we will encode the “flat” partition that induced them by an ordered list of un-ordered groups:

$$\langle \{a, b\}, \{c, d\}, \{x, y\} \rangle \quad (5.2)$$

This encoding emphasizes the fact that the permutations in (5.1) are generated from variable swaps that are strictly *within*, and not across, groups.

Further examination of this function, however, reveals that it remains invariant under the following additional set of eight permutations

$$\{ \langle cdabxy \rangle, \langle cdabyx \rangle, \langle dcabxy \rangle, \langle dcabyx \rangle, \langle cdbaxy \rangle, \langle cdbayx \rangle, \langle dcbaxy \rangle, \langle dcbayx \rangle \} \quad (5.3)$$

which are not captured by the flat partition in (5.2). Note that each of these permutations can be derived from a corresponding permutation in (5.1) by swapping the groups $\{a, b\}$ and $\{c, d\}$. Thus, a suitable encoding that acts as an implicit representation for all sixteen permutations in (5.1) and (5.3) is the following *hierarchical* partition on the set of variables:

$$\langle \{ \{a, b\}, \{c, d\} \}, \{x, y\} \rangle \quad (5.4)$$

In both (5.2) and (5.4) we use angle brackets to indicate a fixed order (single permutation) and curly brackets to indicate all possible orders of the enclosed elements. A pictorial representation of the flat and hierarchical partitions in (5.2) and (5.4) is shown in Figure 5.2.

This small example serves to illustrate several important points that motivate our desire for a

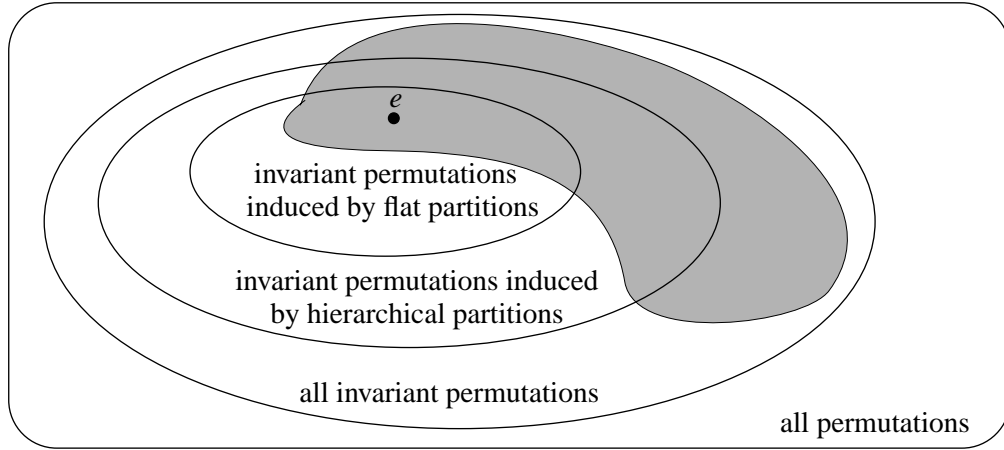


Figure 5.3: Limitation of symmetry groups to represent all invariant permutations of a function

fresh exploration of functional symmetries:

- Function invariance under unrestricted variable permutations expands the classical notion of symmetry by identifying more *structure* in functions than can be inferred from simple variable swaps.
- Functional structure may be specified by an explicit listing of all invariant permutations. However, such a listing may be infeasible due to the exponentially large number of such permutations. Compact implicit representations of this structure include flat and hierarchical partitions on the variable set that act as “stylized permutation generators.” The quality of an implicit representation of invariant permutations can be measured in terms of the number of permutations it generates: representation R_i is deemed superior to representation R_j if it corresponds to a larger number of invariant permutations; in some sense, R_i identifies more of a function’s structure than R_j . An ideal representation would identify the complete set of invariant permutations.
- In addition to compactness, an implicit representation should be efficiently computable. Compact representations whose construction procedures require exponential run times are as infeasible as explicit listings of invariant permutations.

Figure 5.3 depicts a Venn diagram that establishes the relation between the classical and new definitions of function symmetry. The universe is taken to be the entire set of variable permutations and e is used to denote the identity permutation, i.e. the normal variable order. The permutations induced by flat and hierarchical partitions of the variables are thus seen to be subsets of all invariant permutations. In addition, the permutations induced by hierarchical partitioning are clearly

<pre> P ← { }; while X ≠ ∅ do { select a ∈ X; X ← X \ a; G_a ← {a} for ∀b ∈ X do { if a and b are symmetric then G_a ← G_a ∪ {b}; } X ← X \ G_a; P ← P ∪ {G_a}; } </pre>	<pre> // initialize partition P; // while not all variables in X are partitioned; // pick a representative variables; // initialize new variable group G_a; // extend G_a with variables symmetric to a; // remove G_a variables from X; // and update partition P; </pre>
--	--

Figure 5.4: Construction of first-order symmetry partition P for a function f with support X

seen as a superset of those induced by flat partitioning. The shaded subset is meant to represent an alternative implicit representation of invariant permutations that is distinct from those based on partitions on the variable set.

In the remainder of this chapter we develop the concept of hierarchical partitions on the set of variables as a means of characterizing functional structure which extends the classical notion of symmetry. It is important, however, to keep in mind that, while provably superior to flat partitioning, hierarchical partitioning may still be too limited in its ability to capture a sizeable subset of invariant permutations. It does, however, serve as a catalyst for exploring other implicit representations of a function's invariant permutations, a point that we will allude to later when we analyze some of the benchmark circuits.

5.3 Classical First-Order Symmetries

First-order symmetries correspond to function invariance under swaps of variable pairs. Specifically, if variables a and b in the support of function f satisfy the condition:

$$f(\dots, a, \dots, b, \dots) = f(\dots, b, \dots, a, \dots) \quad (5.5)$$

then we say that f has a first-order symmetry between variables a and b . These two variables are also said to form a symmetry group $\{a, b\}$. It is well known, as can be readily shown using Shannon expansion, that condition (5.5) is equivalent to the following equality constraint on the function's cofactors:

$$f_{\bar{a}b} = f_{a\bar{b}} \quad (5.6)$$

Equation (5.6) serves as a computational check for first-order symmetry between variables a and b in function f . It also defines an equivalence relation on the set of variables that can be used to partition the set into its equivalence classes, i.e. symmetry groups, in quadratic time. A sketch of such a procedure, which is composed of two nested loops that iterate on the variables, is shown in Fig. 5.4. The procedure uses X to denote the set of variables, and P to represent the desired partition on X , i.e. the set of symmetry groups formed from X . In the outer `while` loop a variable a from X is chosen and used to *seed* a new symmetry group G_a . In the inner `for` loop, the symmetry of this variable to each other variable b in X is checked; if b is found to be symmetric to a , it is added to G_a . Before proceeding with the next pass of the outer loop, the variables collected in G_a are deleted from X ($X \setminus G_a$) and G_a is added to P . The procedure terminates when X becomes empty.

The extension of these classical results to higher-order symmetries is facilitated by adopting a matrix formulation of the symmetry check in (5.6). As in Chapter 4, let $m_i(x_1 x_2 \dots x_n)$ denote the i^{th} minterm function on the specified ordered set of variables. Cofactors of a function f with respect to variables a and b can now be expressed as the 2×2 matrix:

$$F_{\langle a \rangle, \langle b \rangle} = \begin{bmatrix} f_{m_0(a), m_0(b)} & f_{m_0(a), m_1(b)} \\ f_{m_1(a), m_0(b)} & f_{m_1(a), m_1(b)} \end{bmatrix} = \begin{bmatrix} f_{\bar{a}\bar{b}} & f_{\bar{a}b} \\ f_{a\bar{b}} & f_{ab} \end{bmatrix} \quad (5.7)$$

When clear from context, we will also adopt the following shorthand notation for this matrix:

$$F_{\langle a \rangle, \langle b \rangle} = \begin{bmatrix} f_{0,0} & f_{0,1} \\ f_{1,0} & f_{1,1} \end{bmatrix} \quad (5.8)$$

where $f_{i,j}$ is implicitly understood to stand for $f_{m_i(a), m_j(b)}$.

Comparison of (5.7) or (5.8) with (5.6) immediately suggests that (5.6) is equivalent to requiring the cofactor matrix $F_{\langle a \rangle, \langle b \rangle}$ to be symmetric, i.e.:

$$F_{\langle a \rangle, \langle b \rangle}^T = F_{\langle a \rangle, \langle b \rangle} \quad (5.9)$$

where the superscript T denotes matrix transpose. This result should not be too surprising since condition (5.6) expresses function invariance under the swap of two variables, which in the matrix formulation corresponds to interchanging rows and columns.

5.4 Higher-Order Symmetries

Swaps of variable pairs can be extended in a straightforward manner to swaps of groups of ordered variables. For example, if variables a, b, c , and d in the support of function f satisfy the condition:

$$f(\dots, a, \dots, b, \dots, c, \dots, d, \dots) = f(\dots, c, \dots, d, \dots, a, \dots, b, \dots) \quad (5.10)$$

then we say that f has a second-order symmetry between ordered variable groups $\langle a, b \rangle$ and $\langle c, d \rangle$. This type of symmetry can be conveniently expressed by the symmetry group $\{\langle a, b \rangle, \langle c, d \rangle\}$. The invariance in (5.10) corresponds to the *simultaneous* swap of a with c and b with d and is readily shown to be equivalent to the following constraint on the function's cofactors:

$$F_{\langle ab \rangle, \langle cd \rangle}^T = F_{\langle ab \rangle, \langle cd \rangle} \quad (5.11)$$

where

$$F_{\langle ab \rangle, \langle cd \rangle} = \begin{bmatrix} f_{m_0(ab), m_0(cd)} & f_{m_0(ab), m_1(cd)} & f_{m_0(ab), m_2(cd)} & f_{m_0(ab), m_3(cd)} \\ f_{m_1(ab), m_0(cd)} & f_{m_1(ab), m_1(cd)} & f_{m_1(ab), m_2(cd)} & f_{m_1(ab), m_3(cd)} \\ f_{m_2(ab), m_0(cd)} & f_{m_2(ab), m_1(cd)} & f_{m_2(ab), m_2(cd)} & f_{m_2(ab), m_3(cd)} \\ f_{m_3(ab), m_0(cd)} & f_{m_3(ab), m_1(cd)} & f_{m_3(ab), m_2(cd)} & f_{m_3(ab), m_3(cd)} \end{bmatrix} \quad (5.12)$$

Such an invariance has a special property that when applied twice it reproduces the identity. In [55] it is referred to as *involution* symmetry.

As an example, consider the six-variable function $f = \bar{a}\bar{c}\bar{x} + acx + ab\bar{c}y + \bar{a}cdy + \bar{a}\bar{b}\bar{c}\bar{y} + \bar{a}c\bar{d}\bar{y}$ whose cofactor matrix on $\langle a, b \rangle$ and $\langle c, d \rangle$ is:

$$F_{\langle ab \rangle, \langle cd \rangle} = \begin{bmatrix} \bar{x} & \bar{x} & \bar{y} & y \\ \bar{x} & \bar{x} & \bar{y} & y \\ \bar{y} & \bar{y} & x & x \\ y & y & x & x \end{bmatrix}$$

Since this matrix is symmetric, we can conclude that the function is invariant under the simultaneous swap of a with c and b with d . This is easily verified by direct substitution in the function expression. It is also instructive to examine the cofactor matrix on $\langle a \rangle$ and $\langle b \rangle$:

$$F_{\langle a \rangle, \langle b \rangle} = \begin{bmatrix} (\bar{c}\bar{x} + cdy + c\bar{d}\bar{y}) & (cx + \bar{c}\bar{y}) \\ (\bar{c}\bar{x} + cdy + c\bar{d}\bar{y}) & (cx + \bar{c}y) \end{bmatrix}$$

This matrix is clearly asymmetric; thus, while the function has a second-order symmetry between $\langle a, b \rangle$ and $\langle c, d \rangle$, it does not have a first-order symmetry between a and b . A similar check shows the function to lack first-order symmetry between c and d .

As another example, consider the function

$$g = x(ab(c + d) + cd(a + b)) + \bar{x}(\overline{ab})(\overline{cd}) + y(ab\bar{c}\bar{d} + \bar{a}\bar{b}cd)$$

which has the following cofactor matrix on $\langle a, b \rangle$ and $\langle c, d \rangle$:

$$G_{\langle ab \rangle, \langle cd \rangle} = \begin{bmatrix} \bar{x} & \bar{x} & \bar{x} & y \\ \bar{x} & \bar{x} & \bar{x} & x \\ \bar{x} & \bar{x} & \bar{x} & x \\ y & x & x & x \end{bmatrix}$$

Thus, g has a second-order symmetry between $\langle a, b \rangle$ and $\langle c, d \rangle$. Unlike the previous function, however, g also has the first-order symmetries $\{a, b\}$ and $\{c, d\}$ since the cofactor matrices

$$G_{\langle a \rangle, \langle b \rangle} = \begin{bmatrix} \bar{x}(\overline{cd}) + y(cd) & \bar{x}(\overline{cd}) + x(cd) \\ \bar{x}(\overline{cd}) + x(cd) & x(c + d) + y(\overline{c + d}) \end{bmatrix} \text{ and}$$

$$G_{\langle c \rangle, \langle d \rangle} = \begin{bmatrix} \bar{x}(\overline{ab}) + y(ab) & \bar{x}(\overline{ab}) + x(ab) \\ \bar{x}(\overline{ab}) + x(ab) & x(a + b) + y(\overline{a + b}) \end{bmatrix}$$

are both symmetric. This suggests that besides $\{\langle a, b \rangle, \langle c, d \rangle\}$, three additional second-order symmetry groups exist, namely $\{\langle a, b \rangle, \langle d, c \rangle\}$, $\{\langle b, a \rangle, \langle c, d \rangle\}$, and $\{\langle b, a \rangle, \langle d, c \rangle\}$. All four groups can, thus, be succinctly represented by the single second-order symmetry group on unordered variables $\{\{a, b\}, \{c, d\}\}$ which corresponds to eight invariant variable permutations.

The symmetry structures for these two functions are summarized in Figure 5.5. The structures represent hierarchical partitions on the set of variables and can be generalized to higher orders using the following construction:

1. (*Basis*) Any variable x_i in the function's support is symmetric to itself and forms the symmetry structure $S_i \equiv \{x_i\}$.
2. (*Recursion*)
 - a. If S_1, \dots, S_m are $m \geq 2$ symmetry structures that are pairwise symmetric, then $S \equiv \{S_1, \dots, S_m\}$ is a symmetry structure.
 - b. If $\langle S_1^1, \dots, S_1^k \rangle, \dots, \langle S_m^1, \dots, S_m^k \rangle$ are $m \geq 2$ ordered lists of $k \geq 2$ symmetry structures that are pairwise symmetric, then $S \equiv \left\{ \langle S_1^1, \dots, S_1^k \rangle, \dots, \langle S_m^1, \dots, S_m^k \rangle \right\}$ is a symmetry structure.
3. (*Termination*) If S_1, \dots, S_m is a collection of $m \geq 2$ symmetry structures then $S \equiv \langle S_1, \dots, S_m \rangle$ is a symmetry structure.

Function		$f = \bar{a}\bar{c}\bar{x} + acx + ab\bar{c}y + \bar{a}cdy + \bar{a}\bar{b}\bar{c}\bar{y} + \bar{a}c\bar{d}\bar{y}$	$g = x(ab(c+d) + cd(a+b)) + \bar{x}(\bar{a} + \bar{b})(\bar{c} + \bar{d}) + y(ab\bar{c}\bar{d} + \bar{a}\bar{b}cd)$
Hierarchical Partition	Graphical		
	Symbolic	$\langle \{ \langle a, b \rangle, \langle c, d \rangle \}, \{x\}, \{y\} \rangle$	$\langle \{ \{a, b\}, \{c, d\} \}, \{x\}, \{y\} \rangle$
Invariant Permutations		$\{ \langle abcdxy \rangle, \langle cdabxy \rangle \}$	$\{ \langle abcdxy \rangle, \langle bacdxy \rangle, \langle abdcxy \rangle, \langle badcxy \rangle, \langle cdabxy \rangle, \langle cdbaxy \rangle, \langle dcabxy \rangle, \langle dcbaxy \rangle \}$

○ – unordered □ – ordered (L to R)

Figure 5.5: Symmetry structures for two example functions

When applied to equal-sized variable groups that have disjoint support, the above construction induces a hierarchical partition that can be represented by a tree with two types of nodes:

1. Nodes, depicted as circles, that represent un-ordered sets $\{ \dots \}$
2. Nodes, depicted as rectangles, that represent ordered sets $\langle \dots \rangle$

Let v be a node in such a tree, and let $|v|$ be the cardinality of v , i.e. the size of the set it represents; note that $|v|$ is equal to the number of v 's immediate predecessors in the tree. The number of variable permutations corresponding to v , denoted by $\pi(v)$, can be computed according to the formula:

$$\pi(v) = \begin{cases} |v|! \times \prod_{u \in \text{Pred}(v)} \pi(u) & \text{if } v \text{ is unordered} \\ \prod_{u \in \text{Pred}(v)} \pi(u) & \text{if } v \text{ is ordered} \end{cases} \quad (5.13)$$

The symmetry check in the recursive step of the above construction can be performed by invoking a condition similar to (5.11) on representative variable permutations from each of the two symmetric structures being compared. Specifically, to check structures S_i and S_j for pairwise symmetry, let $\langle x_1 \dots x_p \rangle$ and $\langle y_1 \dots y_p \rangle$ be two variable permutations in their respective supports and $F_{\langle x_1 \dots x_p \rangle, \langle y_1 \dots y_p \rangle}$ be the corresponding cofactor matrix. Then, S_i and S_j are symmetric if

and only if

$$F_{\langle x_1 \dots x_p \rangle, \langle y_1 \dots y_p \rangle}^T = F_{\langle x_1 \dots x_p \rangle, \langle y_1 \dots y_p \rangle} \quad (5.14)$$

Condition (5.14) can be verified by checking the equality of $2^{p-1}(2^p - 1)$ pairs of cofactors on $\langle x_1 \dots x_p \rangle$ and $\langle y_1 \dots y_p \rangle$. Clearly, such a check becomes quite expensive as the structures grow in size. Fortunately, the complexity of the check is reduced to $\frac{1}{2}p(p+1)$ if the structures being checked consist of pairwise symmetric sub-structures. This can be illustrated for $p = 2$ by noting that, when $\{a, b\}$ and $\{c, d\}$ are assumed to be first-order symmetry groups, the two middle columns (resp. rows) of the cofactor matrix in (5.12) become identical. This makes it possible to reduce the size of the matrix to 3×3 by merging rows and columns of equal minterm weight. Such a transformation results in the following matrix form $F_{\langle ab \rangle, \langle cd \rangle}$:

$$\begin{bmatrix} f_{m_0(ab), m_0(cd)} & \{f_{m_0(ab), m_1(cd)}, f_{m_0(ab), m_2(cd)}\} & f_{m_0(ab), m_3(cd)} \\ \left\{ \begin{matrix} f_{m_1(ab), m_0(cd)} \\ f_{m_2(ab), m_0(cd)} \end{matrix} \right\} & \left\{ \begin{matrix} f_{m_1(ab), m_1(cd)}, f_{m_1(ab), m_2(cd)} \\ f_{m_2(ab), m_1(cd)}, f_{m_2(ab), m_2(cd)} \end{matrix} \right\} & \left\{ \begin{matrix} f_{m_1(ab), m_3(cd)} \\ f_{m_2(ab), m_3(cd)} \end{matrix} \right\} \\ f_{m_3(ab), m_0(cd)} & \{f_{m_3(ab), m_1(cd)}, f_{m_3(ab), m_2(cd)}\} & f_{m_3(ab), m_3(cd)} \end{bmatrix}$$

For groups of p symmetric variables, the reduction yields a $(p+1) \times (p+1)$ matrix.

To further reduce the computational cost of constructing a function's hierarchical symmetry partition, we have developed the following necessary condition for two ordered groups of variables to be exchangeable:

Theorem 5.1 *If two ordered disjoint variable groups, $G_1 = \langle x_1, \dots, x_p \rangle$ and $G_2 = \langle y_1, \dots, y_p \rangle$, are symmetric in function f , then variables x_1 and y_1 must be symmetric in function*

$$f^* = \exists G_1 \setminus x_1, G_2 \setminus y_1(f) \quad (5.15)$$

The proof to this theorem is given in Appendix A, and shows how the notion of the cofactor matrix can be used to study symmetric properties of a function.

To illustrate this theorem, consider the function $f = abc + xyz$. According to the theorem, symmetry between $\langle a, b, c \rangle$ and $\langle x, y, z \rangle$ requires that a and x be symmetric in $[\exists b, c, y, z(abc + xyz)] = a + x$ which, trivially, they are. To determine if these two groups are indeed symmetric, we need now to check the symmetry of the cofactor matrix:

$$F_{\langle abc \rangle, \langle xyz \rangle} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

which is symmetric.

To emphasize the fact that the condition in Theorem 5.1 is necessary but not sufficient, consider the function $f(a, b, c, d) = ab\bar{d} + a\bar{b}d + bc\bar{d} + \bar{b}cd$ which has first-order symmetry groups $\{a, c\}$ and $\{b, d\}$. The cofactor matrix on groups $\langle a, c \rangle$ and $\langle b, d \rangle$ is

$$F_{\langle ac \rangle, \langle bd \rangle} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

and is clearly asymmetric. This fact, however, is not detected by the theorem's condition since a and b are trivially symmetric in $[\exists c, d(ab\bar{d} + a\bar{b}d + bc\bar{d} + \bar{b}cd)] = 1$.

Another example illustrates the utility of the condition in the above theorem as a way to prune unnecessary symmetry checks on large sets of variables. It is easy to show that the function $f = ((a \oplus c)x + acy)(b \oplus d)$ has first-order symmetry groups $\{a, c\}$ and $\{b, d\}$. It is also evident, on the other hand, that a and b are not symmetric in the function $[\exists c, d(((a \oplus c)x + acy)(b \oplus d))] = x + ay$. This immediately implies that there are no second-order symmetries between $\{a, c\}$ and $\{b, d\}$, and obviates the need for the more expensive symmetry check implied by (5.11).

It is interesting to note that a variation on Theorem 5.1, in which the existential quantifier \exists is replaced by the universal quantifier \forall , is possible. “Stronger” variations that abstract smaller subsets of variables are also possible and may provide useful trade-offs between runtime efficiency and the accuracy of estimating the function's symmetry. In general, abstracting variables from f gives a function f^* with more invariances in the remaining variables, while preserving their f invariances. This observation provides us with a necessary condition for deriving powerful hints for the identification of symmetry substructures.

For a given function a hierarchical partitioning for a set of its variables may not be unique,

allowing multiple symmetry structures. Such non-uniqueness of the symmetry structures contrasts to the flat partitioning induced by the equivalence relation of classical first order symmetries, which is unique. As an example consider function $f = ab + bc + cd + de + ae$. The function has 5 symmetry structures corresponding to the following 5 hierarchical partitions of its variables: $\langle \{ \langle ac \rangle, \langle be \rangle \}, \{ d \} \rangle, \langle \{ \langle ab \rangle, \langle ed \rangle \}, \{ c \} \rangle, \langle \{ \langle ad \rangle, \langle ce \rangle \}, \{ b \} \rangle, \langle \{ \langle bc \rangle, \langle ed \rangle \}, \{ a \} \rangle$, and $\langle \{ \langle ab \rangle, \langle dc \rangle \}, \{ e \} \rangle$. Observe that permutations from distinct symmetry structures can be composed to derive new invariances of f that are not contained in any of the five listed structures. Indeed, by composing non-trivial permutations of the first four symmetry structures in the listed order we obtain a new *rotational type* [55, 85] permutation $\langle e, a, b, c, d \rangle$. This compositional property of the symmetry structures allows us to list implicitly invariances that cannot be described by a single hierarchical partitioning of a variable set.

5.5 Symmetries Under Phase Assignment

The symmetry condition in (5.14) can be relaxed to describe more involution invariance by allowing the variables being swapped to have selective inversions. Specifically, let $\langle \varphi_1 \dots \varphi_p \rangle$ be a vector of binary phase assignment variables, and replace (5.14) with

$$\exists \varphi_1 \dots \varphi_p (F_{\langle x_1 \dots x_p \rangle}^T \oplus \langle \varphi_1 \dots \varphi_p \rangle, \langle y_1 \dots y_p \rangle = F_{\langle x_1 \dots x_p \rangle \oplus \langle \varphi_1 \dots \varphi_p \rangle, \langle y_1 \dots y_p \rangle}) \quad (5.16)$$

where the \oplus operation is performed bit-wise.

For example, condition (5.14) applied to $f(a, b) = a + \bar{b}$ requires that

$$(F_{\langle a \rangle, \langle b \rangle}^T = F_{\langle a \rangle, \langle b \rangle}) \Leftrightarrow \left(\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \right)$$

which obviously does not hold. On the other hand, applying (5.16) relaxes this requirement, replacing it instead with $(F_{\langle a \rangle, \langle b \rangle}^T = F_{\langle a \rangle, \langle b \rangle}) \vee (F_{\langle \bar{a} \rangle, \langle b \rangle}^T = F_{\langle \bar{a} \rangle, \langle b \rangle})$, i.e.

$$\left(\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \right) \vee \left(\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \right)$$

whose second term is true implying the truth of the entire condition. Thus, the function can be said to have a first-order symmetry between \bar{a} and b , and that $\{\bar{a}, b\}$ is a symmetry group.

The effect of introducing the phase assignment variables in the symmetry check can be seen as a “normalization” of the function that makes it insensitive to the inversion polarity of its inputs. In the matrix formulation of the symmetry check, different assignments to the phase variables corre-

$$\begin{array}{c} a \\ b \end{array} \text{---} \boxed{} \text{---} f = a + \bar{b}$$



Normalized Function Variants		$\begin{array}{c} a \\ b \end{array} \text{---} \boxed{f} \text{---} g_1 = \bar{a} + \bar{b}$	$\begin{array}{c} a \\ b \end{array} \text{---} \boxed{f} \text{---} g_2 = a + b$
Hierarchical Partition	Graphical		
	Symbolic	$\{\bar{a}, b\}$	$\{a, \bar{b}\}$
Invariant Permutations		$\{\langle \bar{a}b \rangle, \langle \bar{b}, a \rangle\}$	$\{\langle a\bar{b} \rangle, \langle b\bar{a} \rangle\}$

Figure 5.6: Symmetry under phase assignment. $\{\bar{a}, b\}$ and $\{a, \bar{b}\}$ represent two equivalent ways of denoting the symmetry of f with respect to a and b

spond to different row orderings; if one or more such orderings yields a symmetric cofactor matrix, the function can be said to have symmetry under phase assignment. An alternative formulation of (5.16) in which the phase assignment variables are associated with the y instead of the x variables is possible and leads to an equivalent requirement on the function cofactors. In this case, however, different phase assignments correspond to different column orderings. For our simple example above, we would deduce that the function is symmetric in a and \bar{b} implying that $\{a, \bar{b}\}$ is a symmetry group. Figure 5.6 illustrates the equivalence of this symmetry group with the one we identified earlier and pictorially shows the corresponding inversions on the function's inputs, the resulting hierarchical partitions, and invariant permutations.

Care must be taken when applying (5.16) to check for higher-order symmetries. Specifically, the assignments available for the phase variables at any level of the partition hierarchy must necessarily be constrained by their assignments at earlier levels. The only flexibility in choosing phase assignments at higher levels of the hierarchy is to reverse the polarity of the support of a symmetry structure; this amounts to choosing one of the two alternative phase assignments propagated from earlier levels of the tree. Symbolically, let $\langle \hat{\phi}_1 \dots \hat{\phi}_p \rangle$ be a phase assignment for which (5.16) held at some level of the hierarchy tree. The symmetry check at subsequent levels in the tree can then be simplified to:

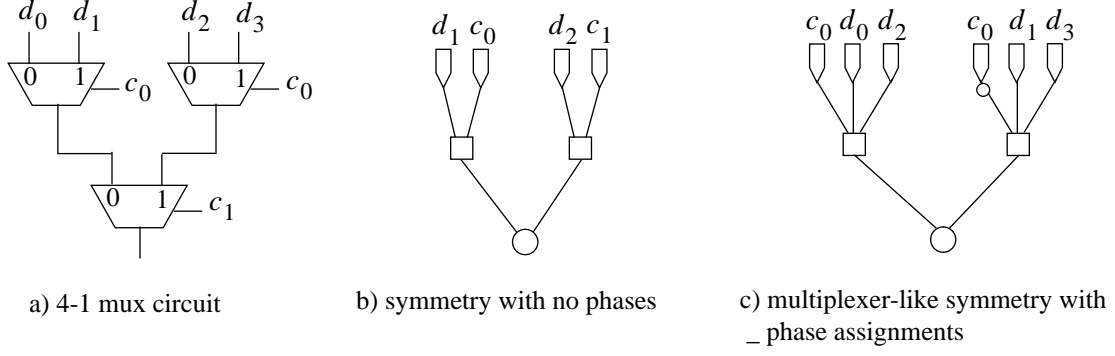


Figure 5.7: A multiplexer circuit and its semantic symmetry structures

$$\begin{aligned}
 (F^T_{\langle x_1 \dots x_p \rangle \oplus \langle \hat{\phi}_1 \dots \hat{\phi}_p \rangle, \langle y_1 \dots y_p \rangle} &= F_{\langle x_1 \dots x_p \rangle \oplus \langle \hat{\phi}_1 \dots \hat{\phi}_p \rangle, \langle y_1 \dots y_p \rangle}) \vee \\
 (F^T_{\langle x_1 \dots x_p \rangle \oplus \overline{\langle \hat{\phi}_1 \dots \hat{\phi}_p \rangle}, \langle y_1 \dots y_p \rangle} &= F_{\langle x_1 \dots x_p \rangle \oplus \overline{\langle \hat{\phi}_1 \dots \hat{\phi}_p \rangle}, \langle y_1 \dots y_p \rangle})
 \end{aligned} \tag{5.17}$$

where complementation of the phase assignment is bit-wise.

As an example of high-order symmetry under phase assignment consider the function $f(a, b, c, d) = \bar{a}b + \bar{c}d$. It has first-order symmetries represented by the symmetry groups $\{\bar{a}, b\}$ and $\{c, \bar{d}\}$. A check of second-order symmetry between these two groups entails the construction of the following two cofactor matrices:

$$F_{\langle \bar{a}b \rangle, \langle c\bar{d} \rangle} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \quad F_{\langle \bar{a}b \rangle, \langle \bar{c}d \rangle} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Since the second matrix is symmetric, we can infer that $\{\{\bar{a}, b\}, \{\bar{c}, d\}\}$ is a symmetry structure for this function.

Symmetry under phase assignments also provides a meaningful extension of higher-order symmetries to the swaps of ordered *non-disjoint* variable groups. We give an example of such symmetry for the 4-1 multiplexer function

$$f = \bar{c}_0 \bar{c}_1 d_0 + c_0 \bar{c}_1 d_1 + \bar{c}_0 c_1 d_2 + c_0 c_1 d_3$$

A circuit implementing this function, and its two symmetry structures are depicted in Figure 5.7. Each of the two symmetry structures in the figure represents a single permutation. The symmetry structure in Figure 5.7-c though, is different from the one in Figure 5.7-b in that its two ordered groups overlap in the c_0 variable. Indeed, it involves swap of a variable, namely c_0 , with its complement. We refer to such symmetry as *multiplexer-like symmetry*. One may also easily verify that

Benchmark suite	Total # of output functions	# of output functions with no symmetries	High-order symmetry statistics				
			# of output functions with high-order symmetries	Max order of symmetry	Max symmetry group size	High-to-first order permutation ratio	
						Min	Max
Multi-level MCNC	1560	76	41	4	64	2	10E13.32
Multi-level ISCAS85	703	85	16	3	12	2	10E6.08
Two-level MCNC	549	68	10	4	64	2	10E90.91

Table 5.1: Summary of symmetry characterization of benchmark circuits

composition of the permutations represented by the symmetry structures of Figure 5.7-b and Figure 5.7-c describes all 8 invariances of the functions under permutation and complementation of its variables.

5.6 Characterization of Function Symmetry in Benchmark Circuits

We performed an extensive study of available benchmark circuits to determine their symmetry partitions based on the generalized symmetry model presented in this chapter. Specifically, we analyzed the 2812 output functions of the 101 logic synthesis and optimization benchmarks available from MCNC [123]. These circuits come from three suites:

- the multi-level MCNC benchmarks
- the multi-level ISCAS-85 benchmarks
- the two-level MCNC benchmarks

The multi-level circuits were flattened before the symmetry partitions of their outputs were computed; thus, the reported symmetry partitions reflect the intrinsic functional structure of these outputs rather than any structural regularity in their circuit implementations. Detailed symmetry profiles tabulated for each output of these circuits can be found in [69]. A summary of these results is shown in Table 5.1.

Several observations can be made from these data. The most striking is the relatively small number of output functions that do not exhibit any symmetries. Considering the fact that some of these functions were generated synthetically to stress synthesis algorithms, this suggests that the majority of functions one is likely to encounter in practical design situations will possess some degree of symmetry. The data also show that a small number of functions have higher order symmetries. In the majority of those cases, the order of symmetry was 2; several functions exhibited symmetries of order 3 and 4. As a measure of the additional symmetries found by hierarchical par-

tioning, we tabulate the ratio of the number of invariant permutations induced by the hierarchical partition to those induced by the first-order partition. This ratio ranged from a minimum of 2 to a maximum of 10^{91} . Finally, symmetry groups ranged in size from a minimum of 2 to a maximum of 64.

We should point out that the symmetry structures were computed under a restriction on the size of ordered groups as well as the variable order within those groups. Specifically, ordered groups chosen for symmetry checks were selected by partitioning the variables into equal-sized subsets using their netlist order. This was done for subset sizes from 2 to 10. This restriction was motivated by the desire to keep the computational effort reasonable, but is otherwise arbitrary.

We also have conducted an experiment to estimate the extent to which invariances in the benchmark circuits are captured with our swap-based symmetry structures. The experiment entailed the computation of symmetries of truth tables for a large number of smaller benchmarks. Specifically we examined a large set of functions whose either on-set or off-set has at most 2,000 minterms. For each of these functions a subject hypergraph was constructed from its truth table [76], and its symmetry was computed using the GRAPE [109] package from the GAP system [48]. Surprisingly, we found no benchmark functions whose symmetries are not expressible with our symmetry structures. This, however, does not imply that they can cover all invariances of a function. In Appendix C on page 151 we give an example function whose symmetry cannot be represented with swaps.

In the remainder of this section we provide a closer examination of the symmetries discovered in four of the benchmark circuits: `t481`, `C432`, `C499`, and `C6288`.

Symmetry characterization of `t481`. This benchmark is interesting because its only output has a 4-level hierarchical symmetry partition (see Figure 5.8). The symmetry involves both ordered and un-ordered groups of variables and requires phase assignment to normalize the function. The number of invariant permutations induced by this partition is 8192 which is 16 times the number of invariant permutations induced by the flat partition of first-order symmetries. The multi-level netlist for this benchmark is quite irregular and large (over one thousand gates). This is at odds with the highly regular symmetry structure shown in Figure 5.8 and suggests that other implementations that are more compact might be possible.

Symmetry characterization of `C432`. Of the seven output functions for this benchmark, only one (`223GAT(84)`) has first-order symmetry. The second-order symmetries exhibited by the other out-

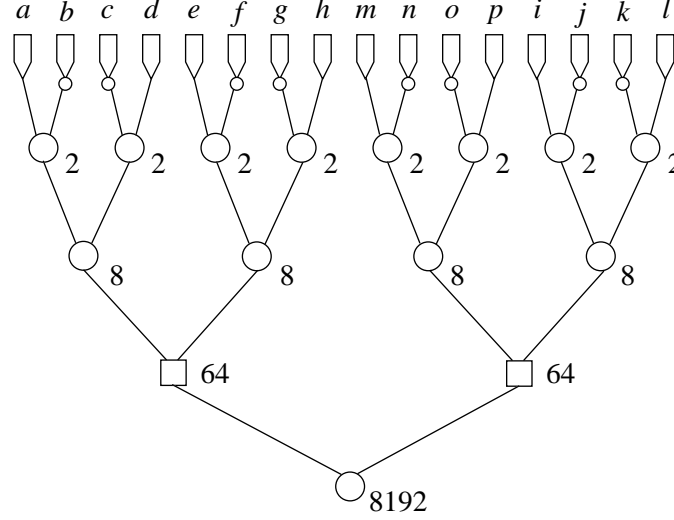


Figure 5.8: Hierarchical symmetry partition of $t481$. Each subtree is annotated at its root with the number of variable permutations it represents

puts are between ordered groups of variables that range in size from 3 to 6. These symmetries correspond to a substantial number of invariant permutations (up to $10^{5.6}$ for two of the outputs) that would have been overlooked by classical symmetry. The detailed hierarchical symmetry partitions for all seven outputs are given in Appendix C on page 146.

Symmetry characterization of C499. C499 has 32 outputs none of which exhibits any symmetry in terms of its 41 inputs. Based on this, one may be led to believe, erroneously, that these functions lack any regularity. Closer examination, however, reveals that a significant amount of symmetry exists in this circuit when its high-level structure is recognized. This structure is depicted in Figure 5.9. The circuit performs single-error-correction [56] and consists of two main modules $M1$ (syndrome generator) and $M2$ (error correction) that are quite regular. The circuit illustrates that completely asymmetric functions may result from the composition of highly symmetric functions. It also suggests that a suitable high-level decomposition might help uncover such latent symmetries. A characterization of the symmetry inherent in C499 is detailed in Appendix C on page 148, where symmetry partitions are derived based on internal signals in addition to primary inputs.

For the module $M1$ we first give symmetries of some of its internal signals D_i and H_i :

$$H_i::\{R, IC_i\}, D_i::\{ID_{i_1}, \dots, ID_{i_{12}}\} \quad (0 \leq i \leq 7, i_j \in \{0, \dots, 31\})$$

These signals then combine to form symmetries of the $M1$ syndrome outputs:

$$S_i::\{D_i, H_i\} \quad (0 \leq i \leq 7)$$

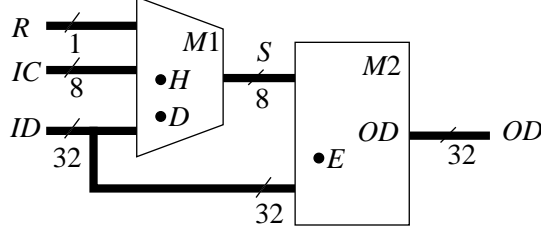


Figure 5.9: High level structure of the C499 single-error-correcting circuit

We first describe the $M2$ equations in terms of the module's internal signals E_i :

$$E_i: \{\dot{S}_0, \dots, \dot{S}_7\} \quad (0 \leq i \leq 31)$$

where dots above S_i 's indicate either complemented or non-complemented phase. Together with ID_i 's we can use the E_i signals to describe the circuit outputs:

$$OD_i: \{E_i, ID_i\} \quad (0 \leq i \leq 31)$$

Symmetry characterization of C6288. Another example that lacks first-order symmetry is the C6288 16-bit multiplier. In fact, no symmetry has been computed for this circuit because it cannot be flattened due to the exponential memory requirements for its BDD. Smaller multipliers that could be flattened showed little first-order symmetry. However, under a remapping of the multiplier's 32 inputs into the domain of its 16^2 partial products, a three-level structure that is rich with symmetries emerges. In fact, this structure can be readily obtained by a partial flattening of the circuit that stops at the first level of 256 AND gates whose output signals correspond to the partial products. By expressing the output functions of the circuit in terms of these signals we obtain functions which are highly symmetric. The detailed characterization of the remapped functions is given in Appendix C on page 150.

The benchmark suggests symmetry profiles for the general multiplier circuit. We formulate the first-order symmetry of the 32 remapped output functions p_k in terms of following formula:

$$p_k = \begin{cases} 1(2) & \text{if } k = 0 \\ 1(2) \dots 1(k+1) & \text{if } 1 \leq k \leq 15 \\ 1(2) \dots 1(30-k) \ 2(30-k+1) \dots 2(15) \ 1(16) & \text{if } 16 \leq k \leq 29 \\ 1(1) \ 2(2) \dots 2(15) \ 1(16) & \text{if } k = 30, 31 \end{cases} \quad (5.18)$$

With each output p_k of the partial multiplier this formula identifies a list of groups using $n(s)$ notation, where n is the number of groups of size s .

5.7 Summary

In this chapter we studied functional symmetry relying on invariance under unrestricted variable permutations. This definition encompasses the classical notion of symmetry, based on variable swaps, as a special case. Our studies are based on a new hierarchical partitioning scheme that generalizes the flat partitioning implied by classical symmetry and yields more invariant permutations. The hierarchical partitioning algorithm is based on the symbolic detection of symmetry in specially-constructed cofactor matrices. The run time efficiency of hierarchical partitioning was shown to be quite reasonable, aided in part by the application of a necessary condition that quickly detects asymmetry. Application of hierarchical partitioning to a large number of benchmark circuits revealed the existence of significant symmetries. Symmetry is not the only functional property that can shed light on a function's semantic structure. In this work it is studied as an example that illustrates how semantic properties of a function can be used to improve synthesis quality.

Chapter 6

Libraries for the Decomposition Patterns

As we had established earlier, the decomposition may or may not exist when imposing support-reducing and fanin-bounding constraints simultaneously (see table (4.10) on page 60). Relaxing at least one of the two constraints, on the other hand, guarantees existence of decomposition. (The effect of imposing these constraints on the existence of decomposition was examined in Chapter 4.) We are interested in the decomposition where both of the constraints are imposed since it breaks function into “smaller pieces,”¹ and therefore is a good indicator of a circuit quality when applied in the constructive synthesis flow. Since the decomposition under these constraints does not always exist, one must be careful how the decomposition primitives are selected. In this chapter we therefore pre-compute libraries for the decomposition patterns which satisfy both of these constraints while ensuring feasible decomposition. Each of the elements in the library is a multi-output module, and can be instantiated into circuit based on the semantic properties of a function; the particular semantic property targeted in this chapter is symmetry.

6.1 Approach

In Chapter 4 we have introduced the notion of a pattern function. It encodes a class of functions with respect to their “decomposition core” inherited from the semantic properties of a function. For example, pattern function

$$F_s(\mathbf{x}_g, \mathbf{Z}) = \sum_{i=0}^s S_i(\mathbf{x}_g) \cdot \zeta_i \quad (6.1)$$

1. measured as the support size of their functions

represents a set of functions which are symmetric in s decomposition variables \mathbf{x}_g ; functions from this set have form:

$$f_s(\mathbf{x}_g, \mathbf{x}_h) = \sum_{i=0}^s S_i(\mathbf{x}_g) \cdot f_i(\mathbf{x}_h) \quad (6.2)$$

where on-set of $S_i(\mathbf{x}_g)$ is composed from minterms of weight i . For the s -to- t pattern functions in (6.1), and a given decomposition pattern s -to- t , we may compute their decomposition primitives only once, and then retrieve them on-demand during synthesis. Clearly, pattern functions may vary for a symmetric property of a function, and therefore, such pre-computation is not limited to one case – several pattern functions may have same semantic structure.

In this chapter we partition arising decomposition possibilities according the desired s -to- t pattern, and the maximum number of the Z variables in its pattern function. A set of modules which makes decomposition feasible for any such s -to- t pattern functions is “functionally complete.” Collectively, these modules form a complete set of decomposition primitives required to implement a particular decomposition inferred by a certain semantic property of a function. We are specifically interested in the support-reducing and fan-in-bounded decomposition for the functions with symmetries.

In the subsequent sections of this chapter we describe how library computation is linked to structure-aware decomposition. We first describe how a library can be computed for a single pattern function symmetric in its decomposition variables (Section 6.2). We then show how to modify the symbolic formulation of decomposition such that *smallest* library is computed for an s -to- t pattern (Section 6.3). The practical significance of computing the smallest library is that when decomposition functions are instantiated from this library they are more likely to get shared as the synthesis evolves. The library computation is done extending our symbolic formulation of decomposition. In Section 6.4 we report the computed libraries, and show how the pre-computation approach extends to other forms of symmetry.

6.2 Library for a Single Pattern Function

To compute libraries for a single pattern function we apply symbolic formulation of decomposition directly. The pattern function $F_s(\mathbf{x}_g, Z)$ will be used to demonstrate this computation. Its expansion in (6.1) describes a class of functions whose s decomposition variables form a symmetry group. A function from this class has at most $s + 1$ distinct cofactors with respect to the min-

term space of decomposition variables: minterms of the same weight have same cofactors. The maximum number on the distinct cofactors implies existence of the decomposition whenever s is a solution to the relation $s + 1 \leq 2^{s-1}$, namely $s \geq 3$. Thus, we can express the cofactors by the $(s + 1)$ -term sum as given in expansion (6.2). The $f_i(\mathbf{x}_h)$ functions in the expansion represent cofactors of f_s with respect to \mathbf{x}_g .

Example 6.1 Suppose we would like to decompose the function

$$f = \bar{a}\bar{b}\bar{c}de + \bar{a}\bar{b}cd + \bar{a}b\bar{c}d + \bar{a}bc\bar{d}e + a\bar{b}\bar{c}d + a\bar{b}c\bar{d}e + ab\bar{c}\bar{d}e + abc\bar{d}$$

with respect to mutually symmetric decomposition variables a, b, c . The equivalent minterms of the decomposition variables induced by symmetry relation along with their cofactors are given in the table below:

i	Factor, $S_i(a, b, c)$	Cofactor, $f_i(d, e)$
0	$\{\bar{a}\bar{b}\bar{c}\}$	de
1	$\{\bar{a}\bar{b}c, \bar{a}b\bar{c}, a\bar{b}\bar{c}\}$	d
2	$\{\bar{a}bc, a\bar{b}c, ab\bar{c}\}$	$\bar{d}e$
3	$\{abc\}$	\bar{d}

According to the table we can write f in the factored form as

$$f = (\bar{a}\bar{b}\bar{c})de + (\bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}\bar{c})d + (\bar{a}bc + a\bar{b}c + ab\bar{c})\bar{d}e + (abc)\bar{d}$$

There is total of four distinct cofactors, and therefore there is a 3-to-2 support-reducing decomposition. ■

Instantiating (4.15) on page 63 with $F_s(\mathbf{x}_g, Z)$ we obtain a computational form for all feasible support-reducing decompositions of n -variable functions that are symmetric in s or fewer variables:

$$G(\Gamma) = \forall Z \forall \mathbf{y} (\exists \mathbf{x}_g (\overline{C(\mathbf{x}_g, \mathbf{y}, \Gamma) \cdot F_s(\mathbf{x}_g, Z)}) + \forall \mathbf{x} (\overline{C(\mathbf{x}_g, \mathbf{y}, \Gamma)} + F_s(\mathbf{x}_g, Z))) \quad (6.3)$$

The above computational form provides us with the decomposition solutions which hold for $F_s(\mathbf{x}_g, Z)$. We assume that function $C(\mathbf{x}_g, \mathbf{y}, \Gamma)$ in (6.3) encodes the space of possible s -to- t reductions. Thus, the resulting function $G(\Gamma)$ encodes a set of feasible t -output decomposition primitives (i.e. modules) which can become member of our symmetric library. As stated, equation (6.3) yields a non-empty set of solutions (i.e. $G(\Gamma) \neq \mathbf{0}$), whenever the number of decomposition relation $t \geq \log(s + 1)$ holds.

We can illustrate the library computation for the symmetric 3-to-2 reduction. The result of exe-

Table 6.1: Characteristics of module libraries necessary for support-reducing symmetric decomposition

Symmetry type		Possible s -to- t reduction	Module library characteristics		
s	Max #distinct cofactors		#Libraries	Size	Example library (composed of s -input symmetric primitives)
2	2	2-to-1	1	3	$\langle \text{AND} \rangle$ $\langle \text{NAND} \rangle$ $\langle \text{XOR} \rangle$
3	2	3-to-1	1	7	$\langle \text{AND} \rangle$ $\langle S_1 \rangle$ $\langle S_2 \rangle$ $\langle \text{NAND} \rangle$ $\langle S_{0,3} \rangle$ $\langle \text{MAJ3} \rangle$ $\langle \text{XOR} \rangle$
3	4	3-to-2	3	1	$\langle \text{NAND} \rangle$, MAJ3
4	2	4-to-1	1	15	$\langle S_0 \rangle \dots \langle S_4 \rangle$ $\langle S_{0,1} \rangle \dots \langle S_{0,4} \rangle$ $\langle S_{1,2} \rangle \dots \langle S_{1,4} \rangle$ $\langle S_{2,3} \rangle$ $\langle S_{2,4} \rangle$ $\langle S_{3,4} \rangle$
4	4	4-to-2	59049	10	$\langle S_{0,1}, S_{0,2} \rangle$ $\langle S_{0,1}, S_{0,3} \rangle$ $\langle S_{0,2}, S_{0,4} \rangle$ $\langle S_{0,3}, S_{2,3} \rangle$ $\langle S_{0,4}, S_{1,4} \rangle$ $\langle S_{0,4}, S_{3,4} \rangle$ $\langle S_{1,2}, S_{2,4} \rangle$ $\langle S_{1,3}, S_{1,4} \rangle$ $\langle S_{1,3}, S_{2,3} \rangle$ $\langle S_{2,4}, S_{3,4} \rangle$
4	5	4-to-3	140	1	$\langle \text{NAND} \rangle$, MAJ4
5	2	5-to-1	1	31	total of 31 single-output modules
5	4	5-to-2	$\approx 1.8^9$	65	total of 65 2-output modules
5	6	5-to-3	420	3	$\langle S_{0,3} \rangle$, MAJ5
5	6	5-to-4	14385	4	$\langle S_{0,3} \rangle$, MAJ5 , ANY5

cutting (6.3) from within the M31 program is given in Appendix D on page 152. As the program indicates the 3-to-2 reduction has three 2-output modules each of which forms a singleton library for this decomposition pattern; these libraries are:

$$\{\langle XOR3, MAJ3 \rangle\}, \{\langle XOR3, SAME3 \rangle\}, \{\langle MAJ3, SAME3 \rangle\} \quad (6.4)$$

where $SAME3(x_1, x_2, x_3) = x_1x_2x_3 + \bar{x}_1\bar{x}_2\bar{x}_3$.

6.3 Library for a Decomposition Pattern

As the relation $t \geq \log(s+1)$ suggests, letting $s = 2$ in the pattern function (6.1) does not yield the support-reducing decomposition, i.e. solving (6.3) for the 2-to-1 reduction gives us $G(\Gamma) = \mathbf{0}$. However, the relation neither implies that there is no 2-to-1 reduction for *any* pattern function symmetric in the two decomposition variables. Indeed, for the pattern functions

$$\begin{aligned} F'_2(x_1, x_2, z_0, z_1) &= S_0(x_1, x_2) \cdot z_0 + S_1(x_1, x_2) \cdot z_0 + S_2(x_1, x_2) \cdot z_1 \\ F''_2(x_1, x_2, z_0, z_1) &= S_0(x_1, x_2) \cdot z_0 + S_1(x_1, x_2) \cdot z_1 + S_2(x_1, x_2) \cdot z_1 \\ F'''_2(x_1, x_2, z_0, z_1) &= S_0(x_1, x_2) \cdot z_1 + S_1(x_1, x_2) \cdot z_0 + S_2(x_1, x_2) \cdot z_1 \end{aligned} \quad (6.5)$$

the 2-to-1 reduction exists since these functions depend only on two cofactor coefficients Z . Our objective in this section is to develop a mechanism which identifies smallest library of multi-output modules that is functionally complete for all decomposition-feasible functions of a given pattern s -to- t .

When support-reducing decomposition in (6.3) yields $G(\Gamma) = 0$ we compute decomposition primitives for the restricted variants of $F_s(\mathbf{x}_g, Z)$. They are the pattern functions which are obtained from (6.1) by imposing a relation between values of its Z variables. Imposing such a relation gives restricted pattern functions in a way similar to how functions in (6.5) are restricted forms of (6.1) (when $s = 2$). For these restricted pattern functions we can compute required libraries, and then construct a complete s -to- t library as their union. However, instead of deriving each of the pattern functions separately, and then computing their libraries, we perform the library construction simultaneously for the pattern functions by extending our symbolic formulation of decomposition.

Throughout the library computation we do not explicitly create all needed pattern functions. Instead, we account for the different pattern functions by extending the symbolic computation in (6.3) with an extra “term” which allows implicitly establish relations between Z variables in $F_s(\mathbf{x}_g, Z)$. This “term” is defined below as a parametrized (with A) constraint function:

$$patt_cnstr(Z, A) = \prod_{i < j}^s ((\zeta_i \equiv \zeta_j) + \bar{\alpha}_{ij}) \quad (6.6)$$

Observe that whenever $\hat{\alpha}_{ij} = 1$ the above constraint may evaluate to 1 only if $\hat{\zeta}_i = \hat{\zeta}_j$.

In general, assignments to the A variables in the $patt_cnstr$ function enforce an equality constraint between the variables of Z , and can be viewed as generators of pattern functions which have stricter form than a given $F(\mathbf{x}_g, Z)$. By restricting the computation of (6.3) to the conditions which satisfy such an induced relation we can relax the computational form (6.3), and provide decomposition solutions of the subclasses of $F(\mathbf{x}_g, Z)$. Such decomposition solutions can be computed simultaneously for all assignment to the A variables by means of the proposition below:

Proposition 6.1 *Let formula in (6.6) encode all possible equality relations between values of the Z coefficients in terms of the assignments to its A variables. For each assignment to A function $G(\Gamma)$ can then be determined using following computational form:*

$$G(\Gamma, A) = \overline{\forall Z \forall \mathbf{y} (\exists \mathbf{x}_g (C(\mathbf{x}_g, \mathbf{y}, \Gamma) \cdot F(\mathbf{x}_g, Z)) + \forall \mathbf{x} (C(\mathbf{x}_g, \mathbf{y}, \Gamma) + F(\mathbf{x}_g, Z)) + patt_cnstr(Z, A))}$$

The proof of the above proposition is based on the observation that $patt_cnstr(Z, A)$ acts as a parametrized (via A) care-set condition extending (6.3). Each assignment \hat{A} gives function $G(\Gamma, \hat{A})$ encoding all feasible decomposition functions for its induced pattern function. Note that an assignment \hat{A} whose variables are all of 0 gives decomposition solutions $G(\Gamma, \hat{A})$ for the pattern function in (6.6).

A characteristic function of all assignments \hat{A} for which $G(\Gamma, \hat{A})$ is not empty can be computed as $\exists \Gamma (G(\Gamma, A))$; in general, the on-set in the computed function can be quite large. We must observe however, that many of the assignments to the A variables are redundant in the encoding $patt_cnstr(Z, A)$ – they repeat other assignments under the transitive closure of the equality relation between Z variables. We discard such solutions by defining a universe of assignments for which the transitive property of the equality is satisfied:

$$T(A) = \prod_{i < k < j} (\alpha_{ik} \alpha_{kj} \rightarrow \alpha_{ij}) \quad (6.7)$$

The above equation provides us with a characteristic function for all assignments valid under the transitive relation between Z variables. Using this function we can restrict our analysis to only to those assignments \hat{A} for which $T(\hat{A}) = 1$ as they subsume redundant assignments; the constrained form of $G(\Gamma, A)$ is then $G(\Gamma, A) \cdot T(A)$.

To reduce further on the number of assignments \hat{A} needed to be considered we make another observation analogous to the transitive relation between Z variables. The observation is based on the fact that some of these assignments define “weaker” equality relation constraints than others. Formally, an assignment \hat{A} is *weaker* than an \hat{B} if and only if relation $\hat{\alpha}_{ij} \leq \hat{\beta}_{ij}$ holds component-wise between values of these assignments, and there exists at least one pair of components such that $\hat{\alpha}_{ij} \neq \hat{\beta}_{ij}$. For the arbitrary assignments to A and B we define such relation symbolically as:

$$less-than(A, B) = \prod_{i < j} (\alpha_{ij} \leq \beta_{ij}) \cdot \overline{\prod_{i < j} (\alpha_{ij} \equiv \beta_{ij})} \quad (6.8)$$

The above formula encodes the “less-than” relation between two domains A and B , which can be used to extract the subset of weakest assignments from a given set of all feasible assignments described by $all(A) = \exists \Gamma(G(\Gamma, A))$. Indeed, the computation

$$weak(B) = \forall A(all(A) \rightarrow (all(B) \cdot \overline{less-than(A, B)})) \quad (6.9)$$

gives an encoding of all weakest pattern functions in terms of B variables for which decomposition exists. It can be easily brought to the form which depends on the A variables. Constraining $G(\Gamma, A)$ with $weak(A)$ we have all feasible decompositions.

For each of the assignments to \hat{A} , such that $weak(A) = 1$, we now have a set of decompositions. These sets provide decomposition primitives which can be used to construct libraries covering all function subclasses.

6.4 Computed Libraries

Table 6.1 summarizes the results of solving (6.3) for all s -to- t support-reducing decompositions where $s \leq 5$. The first two columns in the table characterize the symmetry of the function being decomposed in terms of the number of symmetric decomposition variables s and the maximum number of distinct cofactors with respect to those variables. Column 3 indicates the corresponding achievable support reduction. The remaining columns characterize the module libraries required to realize these decompositions: column 4 is the number of possible minimal-size libraries, column 5 is the number of required s -input cells in each library, and column 6 shows a sample library. The counts in column 4 include only libraries of symmetric cells and assume that libraries consisting of the modules are indistinguishable under complementation and permutation of their outputs. Some of the library module functions listed in column 6 are expressed using the S_a notation, where a is a set of integers identifying the weights of minterms in the on-set of their functions.

We can make the following observations about the results in Table 6.1:

- The libraries in this table represent pre-computed decomposition primitives that map a structural property of the functions being decomposed (symmetry) into a structural property of the circuit implementation (width reduction). Indeed, the complexity of the function being synthesized is reflected directly in the implementation: the support of the most complex symmetric functions (with $s+1$ distinct cofactors) can only be reduced by one, whereas the support of the least complex symmetric functions (with 2 distinct cofactors) can be maximally reduced to 1.
- Whenever $s \geq 3$ and $t \geq \lceil \log(s+1) \rceil$, the libraries for s -to- t reduction are universal in the sense that they will yield the desired decomposition for all functions that are symmetric in s or more variables; they are “functionally complete” for the class of symmetric functions. For instance, there are exactly three universal libraries that enable 3-to-2 decomposition, one of which, $\langle XOR3, MAJ3 \rangle$. If t is too large some of the decomposition output in the s -to- t pattern may become redundant. The 5-to-4 pattern is an example of such redundancy, where the fourth output function (denoted by ANY5 in the module) can be defined arbitrarily.
- For a given s the number of libraries decreases and their size (number of cells) increases with stronger support reduction (smaller t). For $t = 1$, the libraries become unique, up to complementation, and contain $2^s - 1$ cells.
- The libraries in this table can be extended to handle the class of functions that are invariant with respect to complementation of their inputs by placing corresponding inversions on the respective primitive inputs.

When simple symmetries do not exist other structural attributes of a function might be present. In particular, a multiplexer-like symmetry of the form

$$f(\bar{x}_1, x_2, x_3, \dots, x_{2k}, x_{2k+1}, \dots, x_{n-1}, x_n) = f(x_1, x_3, x_2, \dots, x_{2k+1}, x_{2k}, \dots, x_{n-1}, x_n)$$

often arises in datapath circuits. The invariance described by this relation swaps two ordered groups of variables of size k while complementing one variable outside these groups:

$$\{ \langle \bar{x}_1, x_2, x_4, \dots, x_{2k} \rangle, \langle x_1, x_3, x_5, \dots, x_{2k+1} \rangle \} \quad (6.10)$$

In benchmark circuits the most common functions of this type can be described as a sum:

$$f_M(\mathbf{x}_g, \mathbf{x}_h) = \sum_{i=0}^{2^k-1} [\bar{x}_1 \cdot m_i(\mathbf{x}_g^L) + x_1 \cdot m_i(\mathbf{x}_g^R)] \cdot f_i(\mathbf{x}_h) \quad (6.11)$$

where \mathbf{x}_g is composed of x_1 , $\mathbf{x}_g^L = \langle x_2, x_4, \dots, x_{2k} \rangle$ and $\mathbf{x}_g^R = \langle x_3, x_5, \dots, x_{2k+1} \rangle$.

Using the decomposition template (6.11) we can now pre-compute corresponding 3-to-2 libraries using a procedure similar to the one we used in the case of simple symmetries. Specifically, we first express all functions admitting template (6.11) using a suitable encoding function F_M in which the dependence on the non-decomposition variables is eliminated through the introduction of a set of binary encoding coefficients $Z \equiv [\zeta_i]$:

$$F_M(x_1, x_2, \dots, x_{2k+1}, Z) = \sum_{i=0}^{2^k-1} [\bar{x}_1 \cdot m_i(x_2, x_4, \dots, x_{2k}) + x_1 \cdot m_i(x_3, x_5, \dots, x_{2k+1})] \cdot \zeta_i$$

Next, noting that this decomposition template is independent of the “datapath width” m , we reduce it to a width of one by choosing a single representative from each group of “left” and “right” variables \mathbf{x}_g^L and \mathbf{x}_g^R . Without loss of generality we may choose x_2 to represent \mathbf{x}_g^L and x_3 to represent \mathbf{x}_g^R . We then have following cofactors with respect to minterm on the x_1, x_2 and x_3 variables:

i	Set of equivalent minterms	Cofactor, $(F_M)_{m_i(\langle x_1, x_2, x_3 \rangle)}$
0	$\{\bar{x}_1 \bar{x}_2 \bar{x}_3, \bar{x}_1 \bar{x}_2 x_3\}$	$\sum_{i=0}^{2^{m-1}-1} m_i(x_4, \dots, x_{2m}) \cdot \zeta_i$
1	$\{\bar{x}_1 x_2 \bar{x}_3, \bar{x}_1 x_2 x_3\}$	$\sum_{2^{m-1}-1}^{2^m-1} m_i(x_4, \dots, x_{2m}) \cdot \zeta_i$
2	$\{x_1 \bar{x}_2 \bar{x}_3, x_1 \bar{x}_2 x_3\}$	$\sum_{i=0}^{2^{m-1}-1} m_i(x_5, \dots, x_{2m+1}) \cdot \zeta_i$
3	$\{x_1 \bar{x}_2 x_3, x_1 x_2 x_3\}$	$\sum_{2^{m-1}-1}^{2^m-1} m_i(x_5, \dots, x_{2m+1}) \cdot \zeta_i$

We can abstract dependence on the x_4, \dots, x_{2k+1} variables and ζ_i ’s by replacing cofactors in the above table with new ζ'_i coefficients. This replacement allows us to rewrite F_M as

$$F_M(x_1, x_2, x_3, Z) = \bar{x}_1 \bar{x}_2 \cdot \zeta'_0 + \bar{x}_1 x_2 \cdot \zeta'_1 + x_1 \bar{x}_3 \cdot \zeta'_2 + x_1 x_3 \cdot \zeta'_3$$

Substituting $F_M(x_1, x_2, x_3, Z')$ in (4.18) we obtain a computational form similar to (6.3). The solution for this 3-to-2 decomposition yields three possible 2-output modules: $\langle \bar{x}_1 x_2 + x_1 \bar{x}_3, x_1 \rangle$, $\langle \bar{x}_1 x_2 + x_1 \bar{x}_3, \bar{x}_1 x_2 + x_1 x_3 \rangle$ and $\langle \bar{x}_1 x_2 + x_1 x_3, x_1 \rangle$. As follows from the last module, we can

accommodate this type of decomposition by means of a 2-to-1 multiplexer and a wire. (Note that whenever $k = 1$ template (6.11) admits 3-to-1 reduction using just a 2-to-1 multiplexer.) It is interesting to note that restricting our decomposition pattern to three decomposition variables as we did above allows us to avoid computing an exact symmetry structure of the form (6.10) to identify x_1 , x_2 and x_3 . These three decomposition variables can be identified by quantifying out control signal x_1 from $f_M(\mathbf{x})$ which gives a function symmetric in x_2 and x_3 .

6.5 Summary

For functions that are symmetric in some inputs, we have pre-computed libraries required in the implementation of their decomposition patterns. These decomposition patterns capture the structural properties of functions and reflect them in the implementation structure. This is an illustration of how a semantic structure of a function infers “natural” decomposition patterns, along with their primitives. Such a pre-computation does not have to be restricted to the symmetric properties of a functions, as well is does not have to be limited to the pre-computation of decomposition primitives; for example, primitives for the composition function can be pre-computed.

Chapter 7

Practical Issues

In this chapter we discuss an implementation of the concepts introduced in earlier chapters. We develop a constructive synthesis algorithm based on the decomposition template explored in Chapter 4 and Chapter 6, and discuss its implementation in a prototype synthesis tool called M31. The tool uses the CUDD binary-decision diagram package [110] to perform the necessary symbolic manipulations of Boolean functions. To further validate the argument that functional structure can be used to induce a favorable implementation structure through the intermediary of suitable decomposition functions we present and evaluate the results of synthesizing publicly available benchmarks using M31.

7.1 Constructive Synthesis Flow

In this section we show how the decomposition template (4.12) can be used in a constructive algorithmic flow similar to that of M32's in Figure 3.2 on page 34. Unlike the earlier algorithm, however, the new M31 algorithm makes choices that are not encumbered by particular functional representations (e.g. SOP). Instead, all manipulations are based on a representation-free symbolic form that enables the identification and propitious use of a function's semantic structure. The algorithm operates on an evolving Boolean network by repeatedly applying the following steps:

1. select an unimplemented function to decompose
2. select a set of decomposition variables from the function's support
3. select a set of decomposition primitives and introduce them as gates into the network
4. re-express the forward unimplemented logic in terms of the newly-introduced gates

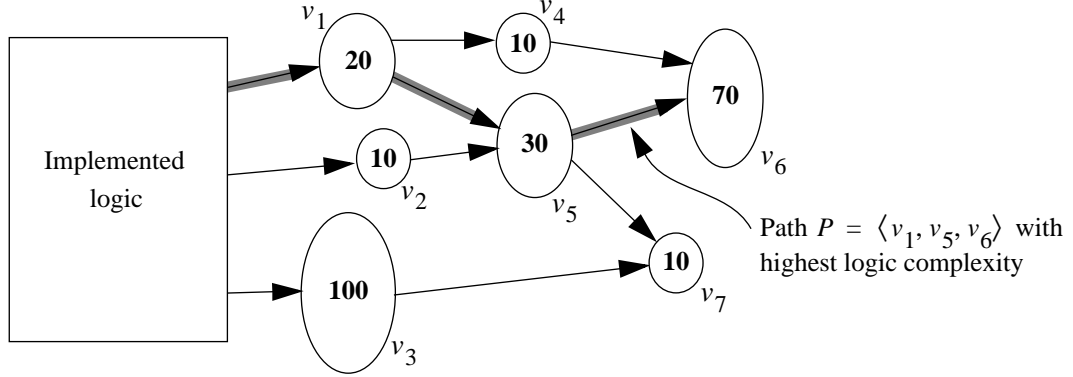


Figure 7.1: Heuristic for the selection of a function to decompose. Numbers annotating the unimplemented nodes denote their complexity. Node v_1 is chosen first for decomposition because it originates the path with the highest logic complexity (path P with complexity 120)

The algorithm terminates when all unimplemented functions have been reduced to single literals, yielding a network that is fully decomposed in terms of library primitives. The choices available and heuristics used in each of these steps are discussed below.

7.1.1 Selection of a function to decompose

The first step in the synthesis algorithm loop selects an unimplemented node to decompose. To aid in this selection, unimplemented nodes are annotated with a complexity measure that is intended to reflect the size of their functions. Using such a measure, larger functions would be expected to have larger implementations compared with smaller functions. Let $G_u = (V_u, E_u)$ be the subgraph of the Boolean network consisting solely of the unimplemented nodes, and let $P = \langle v_1, v_2, \dots, v_n \rangle$ be a path in this subgraph whose starting node v_1 has only implemented nodes as its fan-ins, and whose last node is a primary output. The complexity of path P is defined as the sum of the complexities of its nodes: $\text{Complexity}(P) = \sum_{1 \leq i \leq n} \text{Complexity}(v_i)$.

The node chosen for decomposition is the first-tier node (i.e. node whose fan-ins are already implemented) with the largest path complexity (see Figure 7.1). This choice is motivated by the desire to decompose logic with higher complexity first in the hope that its implementation would be re-used later on in the decomposition of smaller functions without incurring additional area penalties. In addition, choosing nodes based on their path, rather than node, complexity insures that logic that lies on “long” paths is treated first; this tends to reduce overall delay while allowing for the re-use of such logic by shorter paths. Specifically, the implemented logic of larger nodes becomes available to smaller nodes without inadvertently increasing circuit depth, since network depth is dominated by the logic of the larger nodes. Our experiments with other node selection

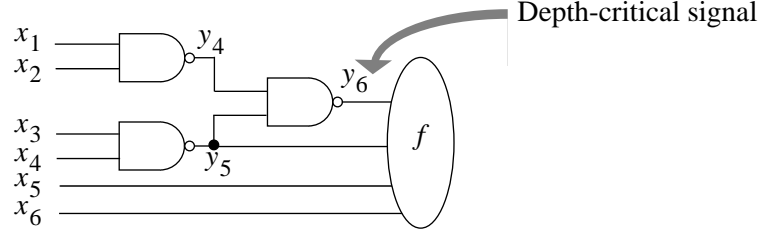


Figure 7.2: Selection of the decomposition variables

heuristics indicate that this “most complex logic first” heuristic is especially suited to minimizing the delay of a circuit.

As a complexity measure, we use the size of a function’s BDD. While there is a vast literature on complexity measures of Boolean functions [120] which can be used in this context, BDD size is particularly convenient as it is readily available in our implementation and seems to work well in practice. Furthermore, the significance of this measure is in its relative, rather than absolute, ranking of the unimplemented functions.

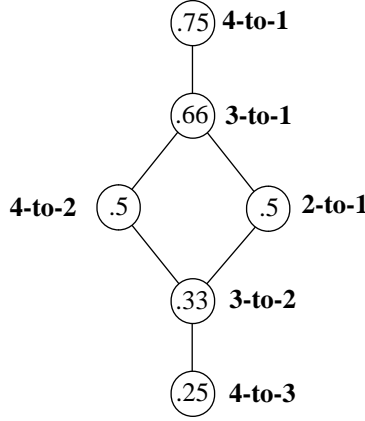
7.1.2 Selection of decomposition variables

The next step after selecting a function to decompose is to determine a suitable subset of its support that will serve as the set of decomposition variables. This is an important step in the synthesis flow as it has a demonstrably significant effect on overall circuit structure. When used in template (4.12) the decomposition variables affect the choices for the decomposition functions as well as the composition function. These choices, in turn, affect the topology of the synthesized circuit. In this section we describe heuristics aimed at selecting good decomposition variables to yield support-reducing decompositions. The heuristics are tied to

- the topological properties of the partially constructed circuit, and
- the semantic properties of the remaining unimplemented logic

We describe below how these properties affect the selection of decomposition variables.

The primary criterion is the selection of decomposition signals whose gate implementation would result in the smallest depth increase of the evolving Boolean network. The topological properties of a partially implemented circuit are therefore used to steer synthesis decisions with the objective of circuit depth minimization. Figure 7.2 helps us to illustrate how a choice of decomposition variables affects network depth. Suppose that we would like to select a set of decomposition variables from f whose implementation has minimum impact on the network depth. Clearly, selecting the y_6 variable would guarantee an increase of the depth of node f , as y_6 ’s connection is



a) ranking based on the reduction ratio

# symm. variables s	max # cofactors	max allowed # cofactors		
		s -to-1	s -to-2	s -to-3
4	5	2	4	8
3	4	2	4	
2	3	2		

b) requirements on the cofactor counts for each of the patterns

Figure 7.3: Ranking and cofactor requirements for the s -to- t decomposition patterns when selecting symmetric decomposition variables

critical for the node. Thus, we would like to avoid selecting y_6 . On the other hand, selecting any subset of variables in the support of f which does not contain y_6 allows us to insert a set of gates implementing their signals without increasing the topological depth of f . The M31 implementation relies on this observation to prioritize variables whose implementation does not increase network depth.

Selection of decomposition variables based purely on the structure of a partially constructed circuit may not be enough though to produce well synthesized circuits. Therefore, in addition to depth considerations, the M31 tool estimates the effect of decomposition variables on the decomposition quality based on the semantic properties of a function. The algorithm relies on a synthesis approach which ties decomposition to the pre-computed decomposition patterns inferred from the intrinsic structure of a function. These decomposition patterns can be efficiently located during synthesis according to the semantic properties of decomposition variables. Thus the selection decision is driven by the desired patterns matching the semantic structure of a function.

We have illustrated such a pre-computation step in Chapter 6 for functions with symmetric properties. The decomposition patterns in the chapter were computed for a class of fan-in-bounded and support-reducing decompositions. Their computation is tied to the symmetric properties of a function, and characterized according to the s -to- t parameters. Each of the decomposition patterns re-encodes s signals of unimplemented logic with t decomposition functions, thereby achieving an s -to- t reduction. We can rank these reductions relating values of s and t parameters. Part (a) of Figure 7.3 depicts such ranking for $s \leq 4$. Requirements on the decomposition patterns

are stated in terms of the cofactor counts in part (b) of the figure. The M31 algorithm selects symmetric decomposition variables which maximizes this ratio, subject to the depth constraints.

In general, a function may not always possess semantic properties matching pre-computed decomposition patterns. The M31 tool in this case relies on a heuristic for the selection of decomposition variables using the global structure of its BDD representation. The heuristic is based on studies in [78, 84] indicating that in a well ordered BDD variables which contribute to the logic complexity the most are positioned closer to the root of a BDD. The results in [78] also suggest that when BDDs are constructed for practical designs these variables correspond to the input signals connecting to the delay-critical circuit parts. Clearly, such a heuristic is sensitive to the variable order in a BDD, suggesting that different circuits can be synthesized simply varying the ordering of BDD variables. Using this heuristic M31 selects the smallest set of decomposition variables enabling support-reducing decomposition. As follows from table (4.10) page 60 on such selection is always possible when the fan-in constraint is removed.

7.1.3 Computation of decomposition functions

For a given set of decomposition variables the decomposition template (4.12) provides a choice of decomposition functions. When the semantic properties of the decomposition variables match pre-computed patterns these choices are readily known. Thus, decomposition signals can be implemented by picking one of the patterns and instantiating it. In the case where the semantic properties of the decomposition variables do not correspond to any of the pre-computed patterns, M31 computes decomposition functions on the fly using our symbolic formulation of decomposition.

In general, we would like to achieve decomposition with as few decomposition functions as possible, i.e. we would like to keep t small. Hence the M31 implementation would select an s -to- t pattern which gives the fewest number of output signals. Such an objective results in fewer connections to the remaining logic. The number of decomposition functions required to achieve decomposition template (4.12) follows directly from the number of distinct cofactors – for the t decomposition functions the number of distinct cofactors induced by minterms of decomposition variables should be at most 2^t . Clearly, decomposition patterns for which 2^t is less than the number of distinct cofactors are infeasible, and therefore will not be selected for decomposition.

When the semantic properties of a function do not match any of the pre-computed patterns, the decomposition functions are computed on the fly, and then constrained by the available library primitives. The constraining is performed by intersecting the $G(\Gamma)$ function with the library primitives encoded as function $L_s(\Gamma_i)$ for each decomposition function i ($1 \leq i \leq t$). Using Γ_i coeffi-

coefficients corresponding to the i th decomposition function (i.e. coefficients of the i th column in the Γ matrix), the $L_s(\Gamma_i)$ function encodes n -input ($n \leq s$) library primitives under permutation and complementation of their inputs, treating each of the encoded functions as an s -input primitive. Each of the $L_s(\Gamma_i)$ encodings is constructed during a library read, and therefore need not to be computed during synthesis. It is possible that the available library of primitives is too stringent to implement a feasible s -to- t reduction, i.e. the library constraining gives $G(\Gamma) = \mathbf{0}$. In this case M31 relaxes the s -to- t reduction by increasing the t parameter. If no s -to- $(s-1)$ library-constrained solution exists, M31 introduces decomposition functions as new unimplemented nodes that will become implemented on the subsequent iterations.

Since within a feasible s -to- t pattern the decomposition functions are not unique, we select them according to a cost function. In the M31 this cost function is implemented as a very simple routine which evaluates decomposition functions by estimating the effect of new gates on the network. The cost function considers increases in the network depth, area, and the number of fan-in connections that result from the new gates. The decomposition function which amounts to smallest increase in these numbers (prioritized in the given order) is then selected. Since some of the gates corresponding to the decomposition functions may already be in the network as the result of earlier iterations; the cost function also accounts for this sharing. When M31 introduces decomposition functions as the unbound nodes the cost function is changed, and it considers fan-in counts and the SOP size¹ of decomposition functions.

The outlined heuristic for selecting decomposition functions can have many other variations. For example, effect of the decomposition functions on [forward] unimplemented logic can be estimated more accurately. In particular, decomposition functions can be selected also trying to consider the extent to which logic of decomposition functions is shared across other unimplemented nodes in the Boolean network. Another possibility is to select decomposition functions to improve semantic properties of the remaining unimplemented logic.

7.1.4 Logic re-expression

After the decomposition functions have been selected we have to perform one more step to complete the decomposition of a function. This step requires the selection of a composition function h . The complete flexibility which arises in selection of h is described by means of intervals in Chap-

1. For the functions of few variable the SOP size tends to provide better estimate for their circuit implementation than the BDD size.

ter 4. As follows from the discussion in that section we are provided with a vast number of choices when picking h . This selection problem is directly related to the minimization of incompletely specified functions, which has been addressed in literature in various contexts.

The most mature account of the minimization of incompletely specified functions is presented in the field of two-level synthesis. Research in this area spans several decades. Two-level minimization of incompletely specified functions has been also studied in multi-level synthesis in the attempt to bypass limitations of algebraic division [11, 40, 119]. In more recent years the problem of minimizing Boolean functions has re-emerged in the context of BDD minimization with don't cares. The effort in this subject originates from the efficient minimization operators *restrict* and *constrain*, introduced by Coudert [34] in 1989; more recent references on this subject include [29, 61, 102, 106].

In spite of the number of available solutions to the minimization problem they all suffer from the limitations which make them inapplicable to our selection of h . Their biggest limitation is that they are not aware of the objectives specific to our decomposition template. In particular, they may produce an h which depends on the decomposition variables. Of course, to avoid this we may do a pre-computation step which first abstracts decomposition variables according to (4.4), and only then do the minimization. Such an approach, however, introduces additional computational effort which could be quite costly. We have therefore decided to develop an efficient algorithm for selecting h which is aware of the problem nature.

Our algorithm is shown in Figure 7.4. Given decomposition functions $\mathbf{g}(\mathbf{x}_g) = (g_1, \dots, g_t)$ and the original function $f(\mathbf{x}_g, \mathbf{x}_h)$, it selects h vacuous in the decomposition variables \mathbf{x}_g . The algorithm is based on the observation that the collection of all 2^t products $\dot{g}_1(\mathbf{x}_g) \cdot \dots \cdot \dot{g}_t(\mathbf{x}_g)$ (where dots denote either complemented or not complemented function) can be viewed as the elementary functions t_i of an orthonormal basis. The complete flexibility in selecting h is then described by the orthonormal expansion, discussed in Section 2.3.2. To ensure that the selected h is independent of the \mathbf{x}_g variables, each subfunction f_i in the expansion must be a cofactor of f with respect to a minterm from t_i ; this requirement is due to Lemma 2.1.

The algorithm in Figure 7.4 presumes an ordering on the g_1, \dots, g_t decomposition functions and is invoked as *re-express*($f, \mathbf{1}, \mathbf{g}, 1$) where $\mathbf{1}$ denotes the space of \mathbf{x}_g minterms, \mathbf{g} is a vector of decomposition functions g_1, \dots, g_t , and 1 is the index of the first decomposition function in the vector. The algorithm builds the composition function h recursively. On each of its recursive calls it tries to refine the minterm space of decomposition variables \mathbf{x}_g with the decomposition

```

function re-express( $f, P, g_1, \dots, g_t, i$ ) {
  if  $\forall m, m' (m, m' \in P \Rightarrow f_m = f_{m'})$  then           // if  $P$  is a subset of an equivalence class,
    return  $f_m$ ;                                           // then return its cofactor;
  assert( $i \leq t$ );                                         // ensure feasibility of decomposition;
  if  $P \leq g_i$  or  $P \leq \bar{g}_i$  then                       // if  $g_i$  is redundant in the subspace  $P$ ,
    return re-express( $f, P, g_1, \dots, g_t, i + 1$ ); // then advance to  $g_{i+1}$ ;
   $h_0 \leftarrow \text{re-express}(f, \bar{g}_i \cdot P, g_1, \dots, g_t, i + 1);$  // find subfunction of  $h$  rooted at  $\bar{y}_i$ ;
   $h_1 \leftarrow \text{re-express}(f, g_i \cdot P, g_1, \dots, g_t, i + 1);$  // find subfunction of  $h$  rooted at  $y_i$ ;
  return  $\bar{y}_i \cdot h_0 + y_i \cdot h_1$ ;                     // return partially constructed  $h$ ;
}

```

Figure 7.4: The re-express algorithm to select composition function when decomposition functions are given

functions. As soon as the algorithm detects a partition P which is a subset of an equivalence class it returns its cofactor. This check is performed by the first statement in the algorithm. As the recursion unravels the cofactor gets conjoined with the y_i signals, re-encoding its factor P . The result of the execution is a composition function h whose support is the output signals of the decomposition functions and \mathbf{x}_h variables. The algorithm also has an assertion statement which fails if the decomposition template of Figure 4.2-a can not be achieved, i.e. no composition function h exists.

Example 7.1 We illustrate execution of the *re-express* algorithm, continuing with Example 4.8. For their function

$$f(a, b, c, d, e) = a\bar{b}d + a\bar{e} + adc + b\bar{e}d + \bar{b}c\bar{e}$$

and a pair of decomposition functions

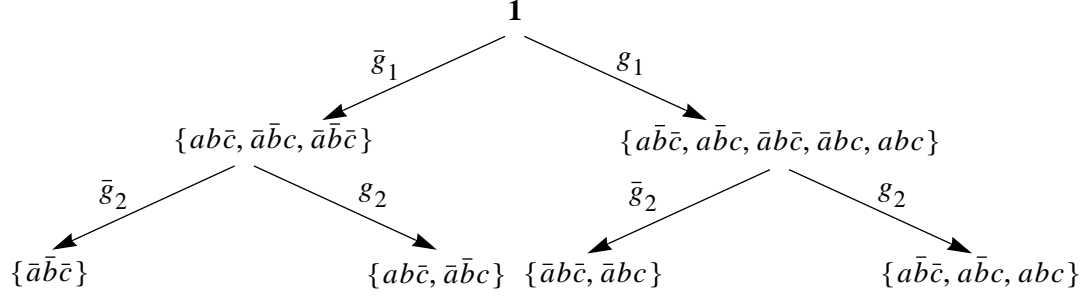
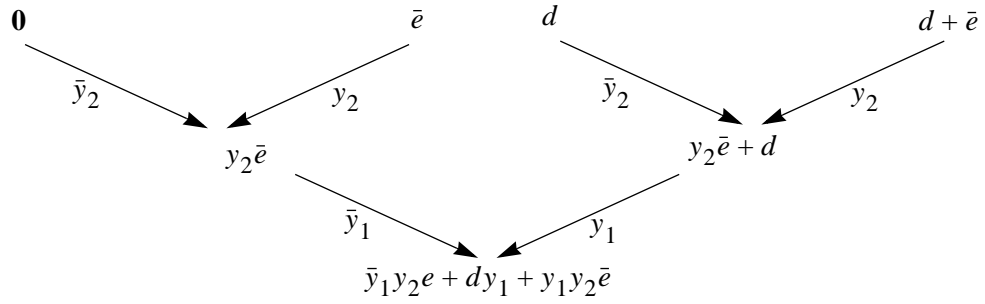
$$g_1 = a\bar{b} + \bar{a}b + bc$$

$$g_2 = a + \bar{b}c$$

we use *re-express* to compute a composition function h . The algorithm computes h recursively. On each recursive call it refines the minterm space of the decomposition variables a , b , and c , trying to identify factors $t_i(a, b, c)$ (see Figure 7.5-a). As the recursion unravels it re-codes these factors with the y_i variables, assigning their phases according to the recursion tree (see Figure 7.5-b). The result of this computation is composition function:

$$h(y_1, y_2, d, e) = y_1d + \bar{y}_1y_2e + y_1y_2\bar{e} \quad \blacksquare$$

In general, unlike to the Figure 7.5 example, the recursion tree may not be balanced. An unbal-

a) partition of minterm space into factors in *re-express*b) composition function creation during recursion unravel in *re-express***Figure 7.5: Composition function selection in *re-express* for the 3-to-2 reduction**

anced recursion tree is the result of flexibility which arises in the selection of h . The algorithm is also sensitive to the order of the decomposition functions – different ordering may result in different h functions.

Using the *re-express* algorithm, M31 attempts to re-express functions of all unimplemented nodes in the network. The re-expression process is performed iteratively for the logic of all nodes which admit support-reducing decomposition for the given set of decomposition functions. Only nodes whose fan-in nodes have been already decomposed and nodes whose support is a superset of the decomposition functions support are considered. Such a complete forward re-expression of logic maximizes node sharing in multi-output circuits. It is worth noting that other criteria are also possible in the selection of composition functions.

7.2 Implementation Issues

Although the symbolic formulation of decomposition is sound in its theoretical foundation, there are some key issues that need to be addressed during its implementation. They are discussed in this section, and are essential for the M31 implementation. We begin by introducing an algorithm to

```

function compute_pattern( $f, \mathbf{x}_g, \mathbf{x}_h$ ) {
     $U \leftarrow \mathbf{1}; F \leftarrow \mathbf{0}; i \leftarrow 0;$ 
    while  $U \neq \mathbf{0}$  do {                                // while minterm space of  $\mathbf{x}_g$  is not empty;
        select  $m \in U;$                                 // select one of the remaining minterms;
         $f_i \leftarrow f_m \equiv f;$                     // compute equivalence class containing  $m$ ,
         $t_i \leftarrow \forall \mathbf{x}_h(f_i);$                     // and use it as a factor  $t_i$  (cf. Corollary 7.2);
         $F \leftarrow F + t_i \cdot \zeta_i;$                 // extend  $F$ ;
         $U \leftarrow U \cdot \bar{t}_i;$                     // subtract minterms of computed factor  $t_i$ ;
         $i \leftarrow i + 1;$                             // increment distinct cofactor count;
    }
    return  $F;$ 
}

```

Figure 7.6: Algorithm to create pattern function $F(\mathbf{x}_g, Z)$ for a given $f(\mathbf{x}_g, \mathbf{x}_h)$

compute the pattern function $F(\mathbf{x}_g, Z)$. It is used as a key to store and retrieve pre-computed decomposition patterns from a cache. The manipulation of BDDs required in the implementation of the symbolic decomposition in the constructive synthesis is then discussed. The section concludes with the implementation of the evolving Boolean network.

7.2.1 Computation of pattern function

The symbolic formulation of decomposition uses the pattern function $F(\mathbf{x}_g, Z)$, defined in equation (4.15). It is constructed for a specific set of decomposition variables \mathbf{x}_g from a given function $f(\mathbf{x}_g, \mathbf{x}_h)$. The construction efficiency of this function is an important factor in the overall synthesis flow, and we therefore focus on a method for its fast generation.

The construction of $F(\mathbf{x}_g, Z)$ can be envisioned as a process which replaces identical cofactors (with respect to minterms from the space of the decomposition variables) of a function with distinct variables ζ_i . A straightforward way of realizing such construction identifies all distinct cofactors by examining them one a time. Such a procedure, however, is guaranteed to be exponential in the number of decomposition variables. We have therefore developed a much more practical algorithm, which ideally meets our synthesis demands.

Our algorithm, called *compute_pattern*, is depicted in Figure 7.6. It is based on the efficient identification of equivalent minterm classes. For a given function f , and a partition of its variables into \mathbf{x}_g and \mathbf{x}_h , it computes the pattern function $F(\mathbf{x}_g, Z)$ as F . The algorithm first initializes the minterm space U of the decomposition variables \mathbf{x}_g to the constant $\mathbf{1}$ function; it also initializes

F to the constant $\mathbf{0}$ function. Each iteration i in the algorithm computes a new factor t_i , and adds the product $t_i \cdot \zeta_i$ to F . On each iteration i the on-set minterms of the factor t_i are also subtracted from the space U . The process continues until all minterms from U belong to one of the factors t_i , i.e. U becomes empty.

When using this algorithm the computation of $F(\mathbf{x}_g, Z)$ requires r number of iterations, where r is the number of distinct cofactors. Although there is an associated computational effort with each of the iterations, our experiments show that it is amortized greatly when the number of iterations is small. Such behavior of the algorithm is especially suited when the decomposition relies on the decomposition variables yielding small cofactor counts, and therefore a small number of iterations. Furthermore, if the feasibility of decomposition depends on a certain threshold r on the number of distinct cofactors, then the number of iterations in the algorithm is bounded by r .

The validity of the *compute_pattern* algorithm rests on the ability to efficiently compute equivalent minterm sets. The result is based on a theorem and corollary, both given below. Let C_i be an equivalence class of minterms induced by the equality relation of cofactors. Then we have following theorem:

Theorem 7.1 *The on-set minterms of $t_i(\mathbf{x}_g)$ form a subset of the equivalence class C_i in $f(\mathbf{x}_g, \mathbf{x}_h)$ if and only if for a minterm m such that $t_i(m) = 1$ following relation holds*

$$t_i(\mathbf{x}_g) \leq (f_m(\mathbf{x}_g, \mathbf{x}_h) \equiv f(\mathbf{x}_g, \mathbf{x}_h)) \quad (7.1)$$

The proof of this theorem is given in Appendix A. A consequence of this result provides us with a computation of an equivalence class C_i for a given minterm m :

Corollary 7.2 *Given function $f(\mathbf{x}_g, \mathbf{x}_h)$ and a minterm m from the space of x_g , the computation*

$$\forall \mathbf{x}_h (f_m(\mathbf{x}_g, \mathbf{x}_h) \equiv f(\mathbf{x}_g, \mathbf{x}_h)) \quad (7.2)$$

yields an equivalence class C_i containing minterm m .

The above result follows immediately from Theorem 7.1 by forcing $f_m(\mathbf{x}_g, \mathbf{x}_h) \equiv f(\mathbf{x}_g, \mathbf{x}_h)$ to be independent of \mathbf{x}_h . In the *compute_pattern* algorithm this corollary is used to select factors t_i composed of the equivalence classes C_i .

7.2.2 Decomposition cache

Up to this point we have studied the significance of the pattern function $F(\mathbf{x}_g, Z)$ in the context of our symbolic formulation of decomposition. This function, however, has another important application in the synthesis flow. Encoding semantic properties of a decomposition core for a function,

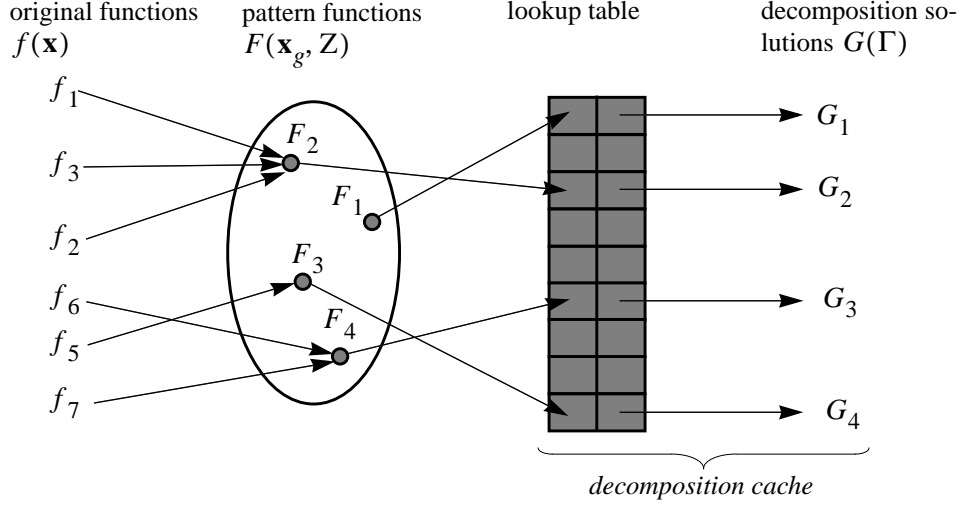


Figure 7.7: Mapping involved in caching of decomposition solutions

it provides a key in the efficient storage and retrieval of decomposition patterns. The pattern function may point either to the pre-computed patterns, or to the patterns which were computed on the fly during a synthesis run. Thus the decomposition solutions $G(\Gamma)$ for a given pattern function need to be computed only once. It turns out that the entire process of storing and retrieving decomposition patterns using the pattern function has a very efficient implementation in the framework of BDDs. The implementation is easily recognized as a conventional cache, and we therefore refer to it as *decomposition cache*.

The topmost view of the mapping involved in the decomposition cache is depicted in Figure 7.7. It has three steps of indirection required to locate the $G(\Gamma)$ encoding of decomposition function. The first mapping step constructs the pattern function for a given f and a set of decomposition variables. The pattern function is constructed using the algorithm *compute_pattern*. The unique table used by the decision diagram algorithms ensures that identical functions not only have same BDDs, but they are also physically the same – they reside in the same memory, and therefore have same addresses. Thus the address of the pattern function provides a key for locating its decomposition solutions, allowing us to consult the decomposition cache in constant time. The decomposition cache then gives a BDD address for the $G(\Gamma)$ function that encodes the decomposition functions. Note that no function f points to the pattern function F_1 in the figure. Such a possibility arises when a decomposition solution is been pre-computed earlier but has not yet been consulted.

Observe that the order in which minterms are selected from U in the *compute_pattern* algo-

rithm affects the constructed functions; different functions may be created for the same $f(\mathbf{x}_g, \mathbf{x}_h)$. Such ambiguity may result into a false cache miss. The M31 tool avoids this ambiguity by implementing *compute_pattern* such that the resulting pattern function is the same for structurally identical $f(\mathbf{x}_g, \mathbf{x}_h)$ functions, i.e. it is canonical in this sense. This canonicity is achieved by first fixing an order on the \mathbf{x}_g variables, and then, on each iteration, selecting minterm $m_i(\mathbf{x}_g)$ from U which has least index i . This way the cache hit is guaranteed when the pattern function computed during synthesis and the pattern function inserted in the cache earlier represent the same semantic property.

The M31 tool implements the decomposition cache by partitioning it into tables according to the s and t parameters. Thus with each s -to- t there is an associated cache table. Such partitioning enables fast lookup of $G(\Gamma)$ according to a desired s -to- t pattern. When the program completes its execution it gives the user the option of saving the state of each decomposition cache. It can then be reconstructed on subsequent runs of the program so that the computed decomposition solutions do not have to be re-computed on each run of the program.

7.2.3 BDD management

In the Section 7.2.2 we have demonstrated the effective application of BDDs to computation reuse. There is also an overhead associated with their manipulation. The foremost reason for this overhead is that the size of BDDs may become very large, thereby impacting software performance. The modified decomposition template addresses this arising complexity by imposing practical decomposition constraints on the decomposition and composition functions. The constructive synthesis flow also provides us with the implementation specific techniques which significantly improve performance of M31; they are described below.

Separate unique tables. The unique table of a BDD manager guarantees that each node is unique; specifically, that there is no other node labeled by the same variable and with the same children. This allows sharing of BDD subgraphs between functions from the same table. However, as a result of such sharing BDDs and their computations may interfere with each other, causing the efficiency of their operations to degrade. For example, a good variable order of one function may cause an increase in the BDD size of another function, and vice versa. This observation motivates us to partition the memory space of BDDs into separate unique tables. In the M31 implementation this partition is achieved by identifying three algorithmic components whose BDDs are relatively independent from each other.

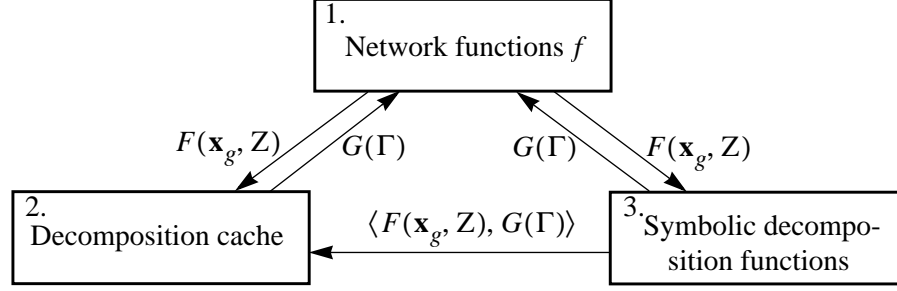


Figure 7.8: Interaction between unique tables in M31

The M31 partition of BDDs into unique tables is depicted in Figure 7.8. Each of the three unique tables corresponds to a block in the figure labeled with its functional content:

1. *Network functions f*. BDDs of the functions corresponding to the network functions. It also defines auxiliary Z variables to enable construction of the $F(\mathbf{x}_g, Z)$ functions.
2. *Decomposition cache*. Functions represented by the unique tables of this block correspond to the decomposition cache. Functions in this table form tuples $\langle F(\mathbf{x}_g, Z), G(\Gamma) \rangle$, where $F(\mathbf{x}_g, Z)$ is used as a key to access $G(\Gamma)$.
3. *Symbolic decomposition functions*. The unique table of this block encompasses all functions and auxiliary variables involved in the symbolic formulation of decomposition. It also contains the library functions $L_s(\Gamma_i)$.

The arrows between blocks in the table indicate their interaction. These interactions arise during synthesis, and may require functions from one table to be expressed in the domain of variables from another table. To realize such translations M31 relies on a routine which performs construction of a new BDD in linear time (in the number of nodes). The complete interaction between the unique tables is described easily in the order of the constructive synthesis flow. For a given network function f and its decomposition variables \mathbf{x}_g , M31 constructs pattern function $F(\mathbf{x}_g, Z)$ using the *compute_pattern* algorithm. In the next step M31 checks the decomposition cache to determine if function $G(\Gamma)$ for $F(\mathbf{x}_g, Z)$ is been already computed. Since the decomposition cache uses a different unique table, the $F(\mathbf{x}_g, Z)$ function needs to be translated in the domain of this table. In case of a cache miss symbolic decomposition of a function is performed, requiring a translation of $F(\mathbf{x}_g, Z)$ into the domain of table 3. After computing the function $G(\Gamma)$, it needs to be inserted into the decomposition cache together with its key $F(\mathbf{x}_g, Z)$. This involves a translation of the $\langle F(\mathbf{x}_g, Z), G(\Gamma) \rangle$ tuple to the domain of the decomposition cache (table 2).

Variable order. The M31 implementation is aware of the potentially detrimental effect of a poor variable ordering in a BDD. It therefore addresses this problem using reordering heuristics provided by the CUDD package [110], as well as careful allocation of variables from within the tool. These methods are applied to the BDDs of network functions, as well as to the functions involved in symbolic decomposition. M31 targets good variable orders from the first steps of synthesis. It also tries to minimize the use CUDD ordering routines as their execution is typically time consuming.

The variable order of the network functions is significant not only from the perspective of a BDD size, but also because it is part of a heuristic to select decomposition variables. (Thus, different variable orders may lead to different circuit implementations.) M31 deduces a good variable order for node functions from the structural properties of the provided network. The deduction is achieved by traversing the network in depth-first-search order starting from the primary outputs, and on each traversal step picking a fan-in node which is furthest from the primary inputs. When the provided network is too shallow to deduce such structure, M31 uses the order of variables supplied in the netlist. If the total number of BDD nodes exceeds a CUDD's pre-defined limit the dynamic variable reordering¹ is triggered.

As the network evolves during synthesis the node functions also change. They become re-expressed in terms of the new variables. It is important to place these new variables in the proper positions in a BDD, or the size of BDDs may increase unreasonably. The M31 implementation maintains a good structure of BDDs by identifying proper locations for newly introduced variables. Indeed, these new variables correspond to the output signals of newly introduced decomposition functions, and they re-encode decomposition variables. Thus, it is reasonable to conclude that when new variables are placed in the former location of decomposition variables the re-encoded BDD inherits the structure of the original BDD. Using this observation M31 places newly introduced variables at the position of the bottom decomposition variables.

The significance of placing new variables in the proper position is illustrated on two practical examples; namely, on carry-out function c_{out} of a 16-bit adder, and on the p_{11} function for the most significant bit in a 6-bit partial multiplier. For the example of the c_{out} function, its BDD variables are ordered by placing the most significant bit variables closest to the root, creating the $a_{15} < b_{15} < \dots < a_0 < b_0 < c_{in}$ order. To perform decomposition of this function, four decomposi-

1. this dynamic reordering can be disabled on the command line

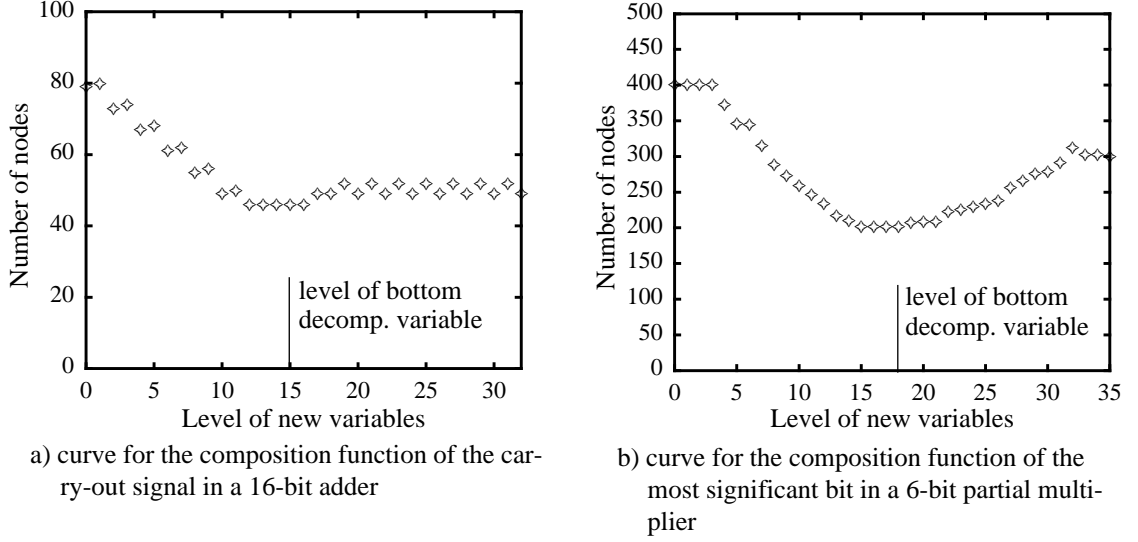


Figure 7.9: Effect of the new variables location on the BDD size of composition function

tion variables were selected – a_9, b_9, a_8, b_8 , and implemented with two majority functions introducing two new variables y_1 and y_2 . After re-expressing c_{out} with these two decomposition functions we evaluated effect of placing the new variable pair at different levels of BDD. The curve plotting the position of new variables against the resulting BDD size is depicted in Figure 7.9-a. One may easily observe that as the new variables get placed further away from the location of decomposition variables the BDD size increases. This effect is even more pronounced for the p_{11} multiplier function. Figure 7.9-b depicts result of choosing different locations for the two new variables implementing the three signals of the partial bit-products $a_5 \cdot b_0, a_0 \cdot b_5$ and $a_1 \cdot b_4$.

Our experiments with M31 have further validated effectiveness of the outlined heuristic for the location of new variables. Virtually all BDDs of the composition function constructed in this way with the *re-express* for the support-reducing decomposition are smaller than the original function. Thus, on each iteration of the constructive synthesis algorithm the representation of unimplemented logic decreases, resulting in the monotonic reduction of BDDs. Such a reduction also eliminating the need for reordering BDD variables.

Underapproximation of decomposition solutions. When the symbolic formulation of decomposition leads to a large BDD for $G(\Gamma)$, we can reshape it by underapproximating its solution space. The result of such underapproximation is a much smaller BDD which encodes fewer decomposition functions. Although such underapproximation eliminates some of the feasible decomposition

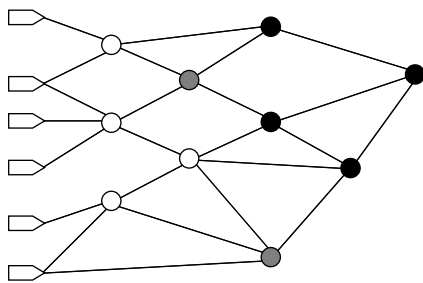
solutions, this loss is not significant since examining all feasible solutions is likely to be an expensive process.

In M31 the underapproximation is invoked when the number of nodes in $G(\Gamma)$ exceeds certain threshold,¹ and before constraining with the $L_s(\Gamma_i)$ library functions. The CUDD package provides several routines to obtain a compact BDD using this approach. In the M31 implementation this underapproximation is achieved using CUDD function `Cudd_SubsetShortPaths`. It extracts a dense subset from a BDD with the shortest paths heuristic. The result of this operation is a compact BDD with the large on-set.

7.2.4 Evolving Boolean network

The evolving Boolean network models the state of the constructive synthesis process. Thus, aside from the traditional realization of a Boolean network, its nodes also have attributes to capture this state. These *state attributes* are associated with each node in the network, and determine their eligibility for decomposition on each iteration of the constructive synthesis algorithm. In what follows we describe the significance of these additional attributes.

Figure 7.10-a depicts small evolving Boolean network. Its nodes are colored according to their states. As it follows from a table in Figure 7.10-b, each node can have up to four states; they are encoded with two binary variables: *fixed* and *active*. Initially only input nodes are marked with the implemented state \circ . Other nodes are unimplemented and marked \bullet , or should not be decomposed and marked \odot . A node may get marked \odot either in a pre-processing step, or it may or during synthesis if it is deemed undecomposable due to some constraints. The \bullet mark denotes an active node which the algorithm is currently decomposing. A node obtains its state \bullet only if all



a) sample network

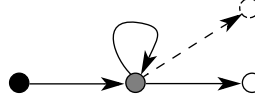
Node	State attributes		Meaning
	Fixed	Active	
●	0	0	unimplemented, not active
●	0	1	unimplemented, active
○	1	0	implemented
⊙	1	1	undecomposable

b) attributes in the network and their meaning

Figure 7.10: Evolving Boolean network and its state attributes

1. a threshold of few thousand nodes

of its fan-in nodes are marked either \bigcirc or \odot , i.e. they are fixed. Any node marked \bullet may eventually turn \odot , if for some reason synthesis process decides not to decompose it further. These semantics define a transition relation between the states of the network nodes. The relation reflects an advancing wavefront of the constructive synthesis algorithm, and is described in terms of the following state transition graph:



As the graph suggests, all nodes in the final network are marked either \bigcirc or \odot .

7.3 Experimental Analysis

The goal of this section is to empirically evaluate the concepts introduced in this work. The implementation of these concepts in M31 allows us to conduct experiments with the benchmark circuits, and evaluate their practical value. In our experimental analysis we use SIS-1.2 as a reference point to judge quality of the M31-synthesized circuits.

7.3.1 Setup

We evaluate the synthesis quality of the M31 and SIS-1.2 programs using two technology libraries: the `mcnc` standard library,¹ and its superset `mcnc+`. We created the `mcnc+` library by extending the `mcnc` library with 3-input exclusive-or, 3-input majority, and 2-to-1 multiplexer gates. These primitives were added to create a “functionally complete” library enabling support-reducing decomposition with respect to the variables possessing simple symmetry, and also multiplexer-like symmetry.

Using these two different libraries, `mcnc` and `mcnc+`, the programs were evaluated on the same set of MCNC combinational benchmarks [123]. These benchmarks vary in the extent of symmetry they have. The detailed symmetry profiles of variable partitions for the benchmark circuits are given in Table 7.1. The table lists the number of symmetry groups of various sizes for a representative sample of the MCNC benchmark circuits. For example, the `z4m1` benchmark with 7 inputs and 4 outputs has 5 groups of 2 symmetric variables and 4 groups of 3 symmetric variables (counted independently for each output). The symmetry ratios listed in the last two columns

1. The standard `mcnc` library is provided with SIS-1.2, and it contains simple and complex `aoi/oai` gates with up to four inputs.

Table 7.1: Symmetry-induced partition of variables in benchmark circuits

Circuit			Symmetry group counts of size n										Symmetry ratio	
Name	Inputs	Outputs	1	2	3	4	5	6	7	8	9	≥10	Min	Max
rd53	5	3					3						1.00	1.00
rd73	7	3							3				1.00	1.00
rd84	8	4								4			1.00	1.00
9symml	9	1									1		1.00	1.00
parity	16	1										1	1.00	1.00
z4ml	7	4		5	4								0.33	1.00
pm1	16	13	4	6	7	1	3		1				0.25	1.00
count	35	16	63	2	1	1	1	1	1	1	1	7	0.20	0.25
pcler8	27	17	24	19	2	2	2	2	2	1		1	0.17	1.00
pm4	16	8	3	10	10	5							0.17	1.00
lal	26	19	22	31	5	2	1	1	1	1			0.12	1.00
sct	19	15	24	34	6	13							0.14	1.00
comp	32	3		48									0.17	1.00
c8	28	18	67	2	1	1	1	1	1	2			0.17	1.00
t481	16	1		8									0.12	0.12
x4	94	71	175	50	39	27	4	9					0.11	1.00
my_adder	33	17		135	17								0.06	1.00
apex7	49	37	220	26	12	4	1	2	8				0.06	1.00
i2	201	1	13			3						5	0.05	0.05

of the table give a rough measure of symmetry in the logic specifications and are computed as the reciprocal of the number of symmetry groups for the least and most symmetric outputs. A symmetry ratio of 1 corresponds to a fully symmetric function (one symmetry group containing all variables), whereas an n -input function that lacks any variable symmetries has a symmetry ratio of $1/n$.

The benchmarks were provided either as two-level or multi-level specifications. M31 constructed symbolic representations from these specifications before commencing the synthesis process. SIS was run with `script.rugged`. Using this script, multilevel circuits were synthesized from both the original specification and from collapsed two-level forms; the results in Table 7.3 report the best variant. For the `my_adder` and `comp` netlists, only the multi-level forms were used since SIS-1.2 was unable to generate their two-level forms due to their large sizes. For the `parity` circuits SIS-1.2 was also not able to complete its synthesis within one hour of the execution.

7.3.2 Results

In Table 7.2 and Table 7.3 we characterize the quality of the circuits synthesized by M31 and SIS-1.2 in terms of their topological properties (node and wire counts, number of logic levels, and aver-

Table 7.2: M31 synthesis results: without and with the pre-computed symmetry library

Circuit	gates		wire count		logic levels		avg. top. wire length		aoi/oai		non-mcnc gates (mcnc+)		
	mcnc	mcnc+	mcnc	mcnc+	mcnc	mcnc+	mcnc	mcnc+	mcnc	mcnc+	maj3	xor3	mux21
rd53	26	10	56	33	6	3	1.34	1.15	5	0	2	2	0
rd73	62	8	125	24	10	3	1.76	1.25	11	0	4	4	0
rd84	71	18	151	45	11	5	1.85	1.27	13	6	4	3	0
9sym	72	13	144	36	14	5	2.07	1.08	15	0	4	4	0
parity	15	8	30	23	4	3	1.53	1.04	0	0	0	7	0
my_adder	206	98	355	278	17	5	2.76	1.29	54	0	74	8	0
pm4	188	47	373	108	19	8	2.39	1.50	27	4	5	7	0
comp	98	78	198	157	17	9	2.31	1.44	13	6	24	0	0
z4ml	18	6	39	18	5	3	1.51	1.67	3	0	3	3	0
t481	23	23	46	40	5	5	1.21	1.20	0	0	2	0	0
pm1	29	29	77	77	4	4	1.56	1.56	1	1	0	0	0
c8	103	66	216	148	5	5	1.52	1.45	0	0	0	0	25
x4	320	235	644	518	8	8	2.37	2.25	44	16	0	0	51
count	91	81	219	195	7	6	1.90	1.88	16	5	0	0	11
pcler8	117	78	251	194	8	7	1.61	1.79	16	10	0	0	12
lal	59	59	119	119	6	6	1.51	1.51	2	2	0	0	0
sct	59	52	117	107	7	6	1.79	1.72	3	2	0	0	3
apex7	358	154	998	325	14	10	1.99	2.17	134	24	0	0	8
i2	95	90	311	302	9	9	1.19	1.18	10	6	0	0	5
Avg.	105.7	60.6	235.2	144.5	9.2	5.7	1.79	1.49	19.3	4.3	6.4	2.0	6.0

Table 7.3: SIS-1.2 synthesis results: without and with the pre-computed symmetry library

Circuit	gates		wire count		logic levels		avg. top. wire length		aoi/oai		non-mcnc gates (mcnc+)		
	mcnc	mcnc+	mcnc	mcnc+	mcnc	mcnc+	mcnc	mcnc+	mcnc	mcnc+	maj3	xor3	mux21
rd53	21	21	47	48	8	8	1.98	1.98	7	7	0	0	1
rd73	40	44	86	92	17	16	2.83	2.80	7	8	0	0	2
rd84	77	76	184	166	13	11	2.15	1.79	27	20	0	0	2
9sym	120	120	310	309	13	13	2.19	2.19	37	36	0	0	1
parity	15	15	15	30	4	4	1.00	1.00	0	0	0	0	0
my_adder	156	156	285	285	35	35	4.91	4.91	19	19	0	0	0
pm4	172	150	399	341	19	19	4.08	3.63	49	36	0	0	5
comp	85	85	168	168	13	13	2.04	2.04	16	16	0	0	0
z4ml	22	22	47	47	10	10	2.49	2.51	8	7	0	0	2
t481	23	23	38	38	5	5	1.21	1.21	0	0	0	0	0
pm1	32	32	64	64	8	8	1.59	1.59	3	3	0	0	0
c8	65	65	168	168	10	10	2.54	2.54	31	31	0	0	0
x4	231	229	514	510	17	16	2.95	2.95	79	77	0	0	2
count	90	90	203	203	20	20	5.39	5.39	24	24	0	0	0
pcler8	69	61	140	128	12	12	3.32	3.61	14	10	0	0	4
lal	65	68	132	140	12	12	2.08	2.09	9	9	0	0	4
sct	52	52	108	108	31	31	6.44	6.44	11	11	0	0	0
apex7	151	151	331	331	15	15	3.10	3.10	41	41	0	0	0
i2	81	81	293	293	8	8	1.31	1.31	9	9	0	0	0
Avg.	82.4	81.1	185.8	182.5	14.2	14	2.82	2.79	20.5	19.1	0.0	0.0	1.2

Table 7.4: Characteristic of the M31 and SIS-1.2 circuits as estimated by SIS-1.2 after they were mapped into `mcnc` library

Benchmark Circuit	Area		Delay		M31/SIS ratio	
	SIS-1.2	M31	SIS-1.2	M31	Area	Delay
rd53	50	56	17.9	14.3	1.12	0.80
rd73	98	75	35.6	17.4	0.76	0.49
rd84	205	107	26.4	22.6	0.52	0.86
9sym	310	84	35.8	18.9	0.27	0.53
parity	75	75	12.4	15.5	1.00	1.25
my_adder	285	667	57.3	26.0	2.34	0.45
pm4	410	289	48.8	42.3	0.70	0.87
comp	168	240	24.0	25.3	1.43	1.01
z4ml	50	59	25.3	14.5	1.18	0.57
t481	53	56	11.8	11.9	1.06	1.01
pm1	87	72	11.1	10.8	0.83	0.97
c8	175	187	27.5	18.5	1.07	0.67
x4	522	624	38.3	22.2	1.19	0.58
count	216	272	58.6	23.3	1.26	0.40
pcler8	142	235	23.5	16.9	1.65	0.72
lal	146	166	22.5	15.6	1.14	0.70
sct	113	129	57.9	16.2	1.14	0.28
apex7	332	358	33.6	29.2	1.08	0.87
i2	296	295	21.7	17.9	0.99	0.82
Avg.	196.4	212.9	31.05	19.98	1.08	0.65

age topological wire length) as well as their utilization of the library primitives (counts of the used complex aoi/oai gates provided by the `mcnc` library, and of the added support-reducing gates). This characterization is done for the circuits synthesized using both `mcnc` and `mcnc+` libraries.

Since the gate count and the number of logic levels give only a rough estimate of the circuit quality we also estimated physical properties of the synthesized circuits. To collect their physical characteristics the circuits synthesized by M31 were evaluated within the SIS-1.2 environment; they were first library-mapped, and then their characteristics were collected. Since the technology mapper in SIS-1.2 does not utilize the extra gates in `mcnc+`, we have limited the mapping process to the `mcnc` library. The results obtained from this mapping process are given in Table 7.4. The table characterizes each of the circuits in terms of SIS-reported area and delay.

7.3.3 Evaluation

The obtained experimental data enables us to make some key observations. We first analyze the results in Table 7.2 and Table 7.3. As the results in these tables show, circuits synthesized by M31 using `mcnc+` library have many fewer logic levels, much lower average topological wire lengths, and fewer connections when compared to SIS-1.2. Such an improved synthesis quality confirms

that a judicious choice of library primitives, coupled with a carefully-tuned decomposition procedure, yields much better synthesis quality than generic decompositions in terms of arbitrary primitives. Indeed, the improvement is most pronounced for the functions rich in symmetries; it reduces gradually as synthesized functions become less symmetric.

The synthesis quality of the M31 program is very much dependent on the library used. This high degree of variation is a direct implication of the tight integration of decomposition with an outfitted library. On the other hand, no such extensive variation can be observed for the circuits synthesized with SIS-1.2 due to the serialization of technology-independent and technology-dependent transformations.

Comparison of the gate types used by each of the tools shows that SIS-1.2 makes very little use of the extra support-reducing gates added to the MCNC library. M31, on the other hand, instantiates these gates extensively as it repeatedly applies support-reducing decomposition. The utilization of extra gates by the M31 program is roughly proportional to the amount of functional symmetries present in a circuit being synthesized. Such a correlation is the result of applying pre-characterized decompositions to the symmetric decomposition variables.

The run time of M31 for the synthesized benchmarks is under one minute; for the circuits with many symmetries the tool completed its synthesis in a few seconds. The experimental runs show that the utilization of the decomposition cache is a significant factor contributing to the improved efficiency. The run time increases when there are fewer symmetries due the increased computational effort in selecting decomposition variables and decomposition functions. In this case selection of decomposition functions is then more often done on the fly, and is more expensive as their encoding function tend to be larger.

The results in Table 7.4 characterize synthesized circuits in terms of their physical properties. Overall, the data shows that M31 circuits have a much better delay, while their area is only slightly increased. We should point out that the `comp` and `parity` benchmarks, for which M31 has slightly larger delay SIS-1.2, were synthesized from completely different starting points using these tools: while SIS-1.2 used the original multi-level form as a starting point, M31 synthesized the netlist from a flat form.

We must emphasize that all of the synthesized circuits have at least some symmetries, making them good candidates for the M31 implementation. Thus, these benchmarks are especially suited for the M31 synthesis. In particular, all these circuits were synthesized using support-reducing

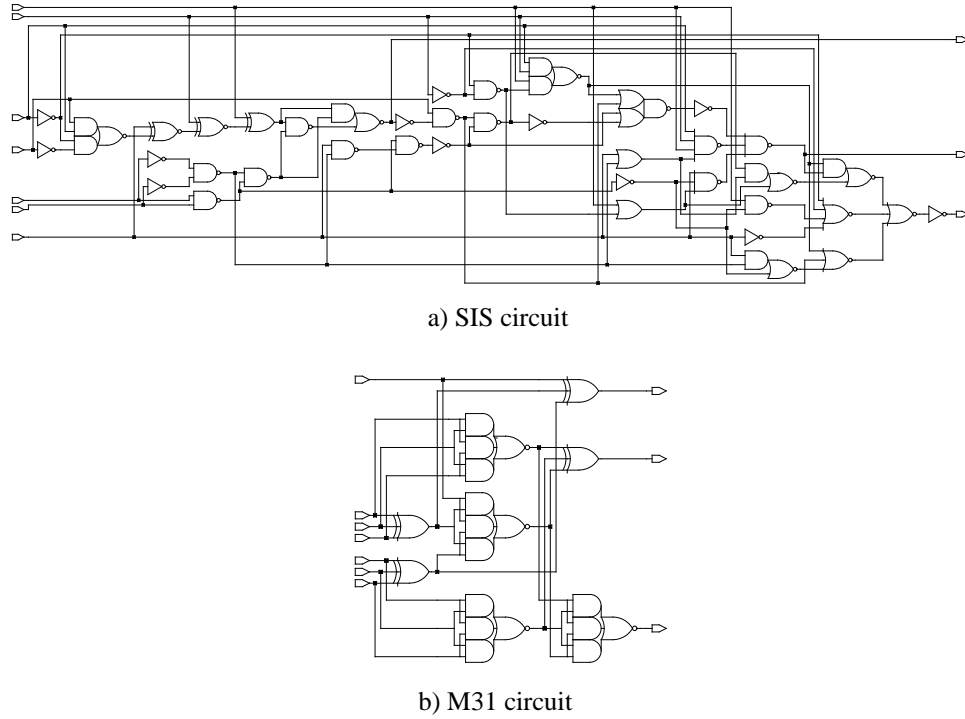


Figure 7.11: Schematics for the `rd73` circuit, synthesized with SIS-1.2 and M31

decomposition selecting at most four decomposition variables. We are aware though that there are many functions which may not have apparent semantic structure, or whose structure is not accounted in the M31 implementation. For these circuits support-reducing decomposition using a small number of decomposition variables may not be a feasible decomposition technique. An example of such functions is a class of non-symmetric unate functions. Their decomposition is likely to require a different decomposition template than the one used in the current M31 implementation. However, our examination of the multi-level MCNC benchmarks shows that the majority of circuits in the suite can be fully synthesized using support-reducing decomposition with a fan-in bound of four. The table in Appendix D on page 157 provides data for the amount of logic synthesized under these constraints using `mcnc+` library.

7.3.4 Highlights

Our further analysis also points to some important observations and features of the M31 synthesis; they are listed below.

Circuits synthesized by M31 often exhibit good structural properties, reflecting their semantic regularity. This is illustrated in Figure 7.11, comparing `rd73` circuits synthesized by SIS-1.2 and M31. The M31 circuit is more compact than the SIS-1.2 circuit; it is also composed of larger blocks, simplifying its layout. As the figure suggests capturing semantic properties of a function

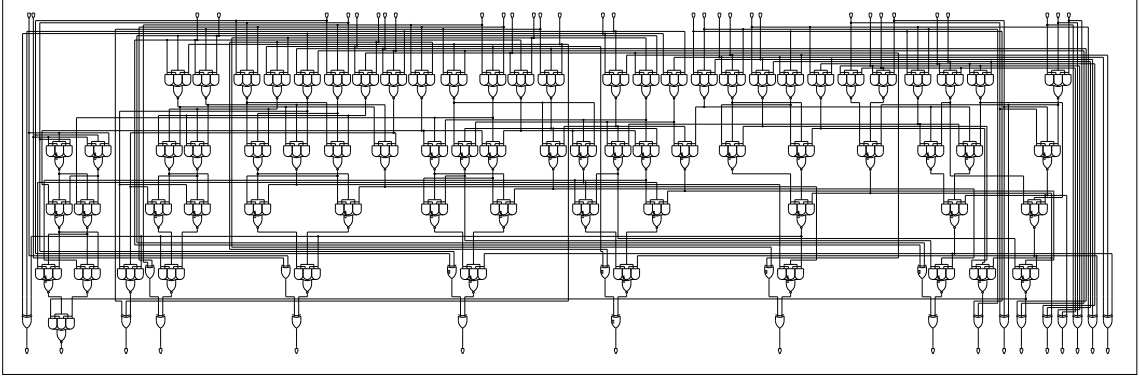


Figure 7.12: A 16-bit adder circuit generated by M31

may have a profound effect on synthesis quality.

There is a significant extent of logic sharing in the M31 circuits. This is due to the constructive synthesis flow in which the gates of partially implemented circuits are made available on the subsequent iterations of the forward, yet undecomposed, logic. The `rd73` circuits in Figure 7.11 illustrates this point – each of its non-output gates has fanout of at least 2.

The results of the M31 synthesis also show that carefully planned automatic synthesis may lead to new circuit structures even for thoroughly studied functions. In Figure 7.12 we give a 16-bit adder synthesized by M31. Its schematic depicts an unconventional variant on the carry-lookahead adder with only five levels of logic. It computes its carry chains without using their conventional generate and propagate encoding. For more detailed insight into its structure we refer the reader to Appendix D on page 154. In Table 7.5 we characterize this class of n -bit adders, for $n = 4, 8, 16, 32, 64$. The area and delay measurements were reported by SIS-1.2 after it mapped each of these adders into the `mcnc` library. The table illustrates a virtually logarithmic increase of their delay in the number of bits.

Although the support-reducing and fan-in-bounded decomposition template used in the M31

Table 7.5: Characteristics of the adders synthesized by M31

Word length, n	Gates count	Logic levels	Area	Delay
4	16	3	109	13.5
8	40	4	273	17.9
16	96	5	667	26.0
32	231	6	1575	25.9
64	526	7	3572	31.1

implementation is by no means complete for all circuits, it is able to synthesize a wide range of practical functions, leading to compact implementations. Examples of such functions include many arithmetic circuits: adders, counters, rotators, etc. Functions which do not admit our constrained decomposition template directly can be synthesized by re-coding them in a larger domain. We have done such a re-coding in Section 5.6 on page 90 for a 16-bit multiplier function. The resulting partial multiplier functions are rich in symmetries. The re-coded function was then synthesized with M31, yielding a 16-bit array style multiplier. The schematic of the synthesized circuit is depicted in Appendix D on page 156.

7.4 Summary

In this chapter we showed how a symbolic formulation of decomposition can be incorporated into a constructive synthesis flow. The constructive synthesis algorithm based on this formulation has been implemented as part of the M31 symmetry-based synthesis tool. The implementation relies on binary-decision diagrams, providing a basic mechanism to manipulate functions in the unrestricted functional domain. An empirical evaluation of M31 shows that linking functional and implementation structures through a carefully-planned decomposition in terms of pre-computed decomposition patterns is an effective synthesis paradigm.

Chapter 8

Conclusions

In this work we addressed the problem of multi-level circuit synthesis with a novel, non-traditional, synthesis approach that infers circuit structure from the semantic properties of a function. The logical progression of reasoning leading to this approach is summarized below. The main contributions of the dissertation are then listed. We conclude with new directions for future work.

8.1 Summary

We began our exploration by addressing current advances in ICs, where wires, and hence circuit structure, become increasingly more important. In response to these new challenges we developed a constructive synthesis approach that is cognizant of the structural implications of decomposition during circuit synthesis. To test the effectiveness of the approach, an algorithm that follows the constructive synthesis paradigm was implemented in the M32 program. M32 manipulates sum-of-products expressions using a variant of algebraic division; the main goal in its implementation was to understand the implications of this novel constructive synthesis flow (e.g. its ability to control implementation structure) as opposed to fine tuning its individual steps. The exercise helped to point out the arbitrariness of algebraic division as a decomposition tool since it bears no relation to the nature of the functions being synthesized. Unrestricted Boolean transformations, on the other hand, are infeasible except for small-scale problems.

We conjectured that a middle ground can be found: a Boolean decomposition strategy that relates structural properties of the functions being synthesized to the structural attributes of the implementation network. To validate this conjecture we began the formulation of a theoretical framework for library-aware constructive logic synthesis. In this framework the decomposition

choices are described symbolically, enabling their natural software implementation in the modern technology of binary decision diagrams. We applied this formulation to the synthesis of functions with symmetries, and showed that significant improvement in circuit quality is possible without undue run time complexity. The improved quality of the synthesized circuits is the result of a coordinated strategy that ties functional structure (symmetries in this case) to appropriately outfitted decomposition patterns through a decomposition procedure that is aware of both.

8.2 Contributions

The primary contributions of this dissertation are summarized below.

- **Constructive synthesis paradigm:** We introduced a constructive synthesis paradigm as an alternative to conventional synthesis. Its algorithm performs decomposition in the context of a given library of decomposition primitives. The algorithm considers the structural implications of candidate decompositions.
- **New theoretical decomposition framework:** We developed a solid theoretical foundation for library-aware structural logic synthesis. It rests on a symbolic formulation of decomposition that covers all decomposition choices in constructive synthesis, i.e. it allows the derivation of all circuits realizing a given function.
- **Exploration of semantic structure:** We showed that rigorous techniques in decomposition exploiting the semantic structure of a functional specification lead to improved synthesis quality – they allow a guided exploration of complex decomposition forms while reducing the run time complexity of the synthesis algorithms. The approach was validated for functions with symmetries.
- **Symmetry identification:** We classified new forms of functional symmetries based on group swaps and developed new algorithms for their identification. We also performed a comprehensive study of the symmetries in publicly available benchmark circuits.
- **Practicality of the approach:** We showed new ways for leveraging the modern technology of binary decision diagrams in a synthesis system. The experimental results showed that circuits synthesized with our prototype implementation often have better delay characteristics than circuits generated by conventional synthesis tools.

8.3 Future Work

The contributions of this dissertation, of course, do not give a complete answer to the multi-level synthesis problem. Various extensions of this work are possible, some of which are given below.

- **Further exploration of semantic properties:** The symmetry insights developed in this dissertation may get further explored in structure-aware functional decomposition to yield “natural” decomposition patterns. The regular structure of datapath-type circuits make such explorations particularly appealing. Beyond symmetries there are other semantic properties of a function which can be explored to improve synthesis quality. For example, unateness of control logic may get explored to limit the decomposition space, and to infer structure of the implementation circuit.
- **Libraries of larger granularity:** In the implementation of our synthesis algorithm we have limited decomposition primitives to those with a small fan-in. Such a restriction can be lifted to explore the effect of larger decomposition blocks on circuit quality. These blocks may come in a variety of netlist topologies, forming a class of implementations for a block. As the constructive synthesis process progresses, an instantiated representative of the class may get replaced with other class members, whose netlist topology is more appropriate in view of the evolved Boolean network; for example, a netlist that has more internal signals available to the forward logic may get instantiated instead.
- **Extraction of a certain semantic core:** There are many functions which do not appear to have much semantic structure. However, slight modifications to their representations may yield a far improved semantic structure. We have illustrated such a modification on a multiplier function by expressing it in terms of the n^2 partial bit-products. The re-expressed function was constructed in terms of the internal signals of the provided network, prompting the question: How does one identify portions of a network which satisfy a certain semantic structure?
- **New decomposition templates:** We have illustrated a symbolic formulation of decomposition that imposes practical fan-in-bounding and support-reducing constraints. For the decomposition of some functions, however, these constraints could be too limiting. Thus, other decomposition templates should be explored. For example, support-maintaining, or support-increasing, decomposition templates allow us to re-encode the function in terms of new signals, and thereby improve its semantic structure. Although,

such decomposition templates do not reduce the circuit width on each of the successive levels, they simplify synthesis on subsequent iterations of the constructive synthesis algorithm.

- **Enumeration of decompositions:** Implementation details of the symbolic formulation of decomposition can be further explored. For example, the M31 approach to caching decomposition solutions may be well suited for enumerating decomposition possibilities, instead of selecting them greedily.

Appendices

Appendix A

Theorem Proofs

Chapter 2 Proofs

Lemma A.1 (Lemma 2.1) *For a given function $f(\mathbf{y}, \mathbf{z})$ and a basis function $t_i(\mathbf{y})$, the subfunction $f_i(\mathbf{y}, \mathbf{z})$ is vacuous in \mathbf{y} and satisfies*

$$t_i(\mathbf{y}) \cdot f(\mathbf{y}, \mathbf{z}) \leq f_i(\mathbf{y}, \mathbf{z}) \leq \overline{t_i(\mathbf{y})} + f(\mathbf{y}, \mathbf{z}) \quad (\text{A.1})$$

if and only if it is equal to the cofactor $f_m(\mathbf{y}, \mathbf{z})$, where m is a minterm on \mathbf{y} such that $t_i(m) = 1$.

Proof: (\Rightarrow). Suppose that f_i satisfies (A.1) and is vacuous in the \mathbf{x} variables. We may then have derivation:

$$\begin{aligned} t_i(\mathbf{y}) \cdot f(\mathbf{y}, \mathbf{z}) &\leq f_i(\mathbf{z}) \leq \overline{t_i(\mathbf{y})} + f(\mathbf{y}, \mathbf{z}) && (\text{A.1}) \text{ theorem constraint} \\ \Rightarrow (t_i(\mathbf{y}) \cdot f(\mathbf{y}, \mathbf{z}))_m &\leq (f_i(\mathbf{z}))_m \leq (\overline{t_i(\mathbf{y})} + f(\mathbf{y}, \mathbf{z}))_m && \\ &&& \text{cofactor containment property (Table 2.1)} \\ \Leftrightarrow (t_i(\mathbf{y}))_m \cdot (f(\mathbf{y}, \mathbf{z}))_m &\leq (f_i(\mathbf{z}))_m \leq (\overline{t_i(\mathbf{y}))}_m + (f(\mathbf{y}, \mathbf{z}))_m && \\ &&& \text{cofactor distributivity (Table 2.1)} \\ \Leftrightarrow 1 \cdot f_m(\mathbf{y}, \mathbf{z}) &\leq f_i(\mathbf{z}) \leq \bar{1} + f_m(\mathbf{y}, \mathbf{z}) && \text{cofactor definition} \\ \Leftrightarrow f_m(\mathbf{y}, \mathbf{z}) &= f_i(\mathbf{z}) && \text{relation } \leq \text{ definition} \end{aligned}$$

Thus, $f_i(\mathbf{z})$ is the unique cofactor $f_m(\mathbf{y}, \mathbf{z})$.

(\Leftarrow). Cofactor $f_m(\mathbf{y}, \mathbf{z})$ is vacuous in the \mathbf{x} variables according to its definition. We must also show that $f_m(\mathbf{y}, \mathbf{z})$ satisfies bounds (2.6). The derivation below establishes this fact:

$$\begin{aligned} m \cdot f(\mathbf{y}, \mathbf{z}) \oplus m \cdot f_m(\mathbf{y}, \mathbf{z}) &= \mathbf{0} && \text{cofactor approximation property (Table 2.1)} \\ \Leftrightarrow m \cdot f(\mathbf{y}, \mathbf{z}) \cdot \overline{f_m(\mathbf{y}, \mathbf{z})} + \overline{m} \cdot f(\mathbf{y}, \mathbf{z}) \cdot f_m(\mathbf{y}, \mathbf{z}) &= \mathbf{0} && \oplus \text{ expansion} \\ \Leftrightarrow (m \cdot f(\mathbf{y}, \mathbf{z}) \cdot \overline{f_m(\mathbf{y}, \mathbf{z})} = \mathbf{0}) \wedge (\overline{m} \cdot f(\mathbf{y}, \mathbf{z}) \cdot f_m(\mathbf{y}, \mathbf{z}) = \mathbf{0}) &&& \text{monotonicity of } + \end{aligned}$$

The derivation holds for any minterm m such that $t_i(m) = 1$. We must therefore have:

$$\left(\sum_{\text{m.s.t. } t_i(m)=1} m \cdot f(\mathbf{y}, \mathbf{z}) \cdot \overline{f_m(\mathbf{y}, \mathbf{z})} = \mathbf{0} \right) \wedge \left(\sum_{\text{m.s.t. } t_i(m)=1} \bar{m} \cdot f(\mathbf{y}, \mathbf{z}) \cdot f_m(\mathbf{y}, \mathbf{z}) = \mathbf{0} \right)$$

monotonicity of +

$$\Leftrightarrow (t_i(\mathbf{y}) \cdot f(\mathbf{y}, \mathbf{z}) \leq f_m(\mathbf{y}, \mathbf{z})) \wedge (f_m(\mathbf{y}, \mathbf{z}) \leq \overline{t_i(\mathbf{y})} + f(\mathbf{y}, \mathbf{z})) \quad (2.3) \text{ identity}$$

The last step in the derivation corresponds to (2.6) ■

Lemma A.2 (Lemma 2.3) $[l(\mathbf{x}), u(\mathbf{x})]_{x_i} = [l(\mathbf{x})_{x_i}, u(\mathbf{x})_{x_i}]$

Proof: (\Rightarrow). We first show that if a function is contained in $[l(\mathbf{x}), u(\mathbf{x})]_{x_i}$ then it is also contained in $[l_{x_i}(\mathbf{x}), u_{x_i}(\mathbf{x})]$. Omitting dependence on the \mathbf{x} variables, suppose that there is some function g such that $g \in [l, u]_{x_i}$, and $g \notin [l_{x_i}, u_{x_i}]$. According to the interval cofactor definition there must also be function f in $[l, u]$ such that $g = f_{x_i}$. Since $l \leq f \leq u$, the containment property of the cofactor operation also implies that $l_{x_i} \leq f_{x_i} \leq u_{x_i}$. This however, contradicts our original assumption that $g \notin [l_{x_i}, u_{x_i}]$. Hence, functions in $[l(\mathbf{x}), u(\mathbf{x})]_{x_i}$ must be also contained in $[l_{x_i}, u_{x_i}]$.

(\Leftarrow). We now show that if a function is contained in $[l_{x_i}, u_{x_i}]$ then it is also contained in $[l, u]_{x_i}$. Suppose that there is function g such that $g \in [l_{x_i}, u_{x_i}]$ and $g \notin [l, u]_{x_i}$. This implies that

$$\forall f (f_{x_i} = g \Rightarrow f \notin [l, u]) \quad (\text{A.2})$$

For the function g , on the other hand, following derivation holds:

$$\begin{aligned} l_{x_i} \leq g \leq u_{x_i} &\Rightarrow x_i \cdot l_{x_i} \leq x_i \cdot g \leq x_i \cdot u_{x_i} && \text{monotonicity of } \cdot \\ &\Rightarrow x_i \cdot l_{x_i} + \bar{x}_i \cdot l_{\bar{x}_i} \leq x_i \cdot g + \bar{x}_i \cdot l_{\bar{x}_i} \leq x_i \cdot u_{x_i} + \bar{x}_i \cdot l_{\bar{x}_i} && \text{monotonicity of } + \\ &\Rightarrow x_i \cdot l_{x_i} + \bar{x}_i \cdot l_{\bar{x}_i} \leq x_i \cdot g + \bar{x}_i \cdot l_{\bar{x}_i} \leq x_i \cdot u_{x_i} + \bar{x}_i \cdot u_{\bar{x}_i} && x_i \cdot l_{x_i} \leq x_i \cdot u_{x_i} \text{ condition} \\ &\Rightarrow l \leq x_i \cdot g + \bar{x}_i \cdot l_{\bar{x}_i} \leq u && \text{Shannon expansion} \end{aligned}$$

The last step of the derivation gives us function $f = x_i \cdot g + \bar{x}_i \cdot l_{\bar{x}_i}$ in the $[l, u]$ interval such that $f_{x_i} = g$. This however, contradicts (A.2) requiring that no such function f is contained in $[l, u]$. Therefore, function g must be in $[l, u]_{x_i}$. ■

Lemma A.3 (Lemma 2.4) $[l_1(\mathbf{x}), u_1(\mathbf{x})] \cap [l_2(\mathbf{x}), u_2(\mathbf{x})] = [l_1(\mathbf{x}) + l_2(\mathbf{x}), u_1(\mathbf{x}) \cdot u_2(\mathbf{x})]$

Proof: We show that function f is contained in $[l_1 + l_2, u_1 \cdot u_2]$ if and only if it is contained in $[l_1, u_1]$ and in $[l_2, u_2]$ by means of following derivation:

$$\begin{aligned} (l_1 \leq f \leq u_1) \wedge (l_2 \leq f \leq u_2) &&& \text{interval definition} \\ \Leftrightarrow (\bar{l}_1 + f = \mathbf{1}) \wedge (\bar{f} + u_1 = \mathbf{1}) \wedge &&& (2.3) \text{ identity} \\ (\bar{l}_2 + f = \mathbf{1}) \wedge (\bar{f} + u_2 = \mathbf{1}) \end{aligned}$$

$$\begin{aligned}
&\Leftrightarrow ((\bar{l}_1 + f) \cdot (\bar{l}_2 + f) = \mathbf{1}) \wedge && \text{monotonicity of product} \\
&\quad ((\bar{f} + u_1) \cdot (\bar{f} + u_2) = \mathbf{1}) \\
&\Leftrightarrow (\bar{l}_1 \cdot \bar{l}_2 + f = \mathbf{1}) \wedge (\bar{f} + u_1 \cdot u_2 = \mathbf{1}) && \text{distributivity} \\
&\Leftrightarrow (l_1 + l_2 \leq f) \wedge (f \leq u_1 \cdot u_2) && (2.3) \text{ identity}
\end{aligned}$$

The last step of the derivation corresponds to $[l_1 + l_2, u_1 \cdot u_2]$. ■

Theorem A.4 (Theorem 2.6) *The function interval $\nabla x_i[l(\mathbf{x}), u(\mathbf{x})]$, defined in (2.11) on page 26, represents the set of all functions $f(\mathbf{x}) \in [l(\mathbf{x}), u(\mathbf{x})]$ that are vacuous in x_i . In other words, for all functions $f(\mathbf{x})$ that are vacuous in x_i it must be that:*

$$f(\mathbf{x}) \in \nabla x_i[l(\mathbf{x}), u(\mathbf{x})] \Leftrightarrow f(\mathbf{x}) \in [l(\mathbf{x}), u(\mathbf{x})]$$

Proof: (\Rightarrow). We need to prove that every vacuous in x_i function $f(\mathbf{x})$ from the abstracted interval $\nabla x_i[l(\mathbf{x}), u(\mathbf{x})]$ is contained in $[l(\mathbf{x}), u(\mathbf{x})]$. This can be done by merely showing that the abstracted interval is contained in the original one. Indeed, the fact follows immediately from the established in (2.12) identity $\nabla x_i[l(\mathbf{x}), u(\mathbf{x})] = [\exists x_i l(\mathbf{x}), \forall x_i u(\mathbf{x})]$, implying that $f(\mathbf{x})$ is also contained in $[l(\mathbf{x}), u(\mathbf{x})]$.

(\Leftarrow). Suppose that vacuous in x_i function $f(\mathbf{x})$ is contained $[l(\mathbf{x}), u(\mathbf{x})]$. The necessary condition of the theorem requires us to show that function it is also contained in the abstracted interval $\nabla x_i[l(\mathbf{x}), u(\mathbf{x})]$. Since $f(\mathbf{x})$ is vacuous in x_i it must be that $f_{\bar{x}_i}(\mathbf{x}) \in [l(\mathbf{x}), u(\mathbf{x})]_{\bar{x}_i}$ and $f_{x_i}(\mathbf{x}) \in [l(\mathbf{x}), u(\mathbf{x})]_{x_i}$. These two cofactors are equal, and therefore according to definition of the interval product and (2.12) must be contained in $\nabla x_i[l(\mathbf{x}), u(\mathbf{x})]$. Furthermore, $f(\mathbf{x}) = f_{\bar{x}_i}(\mathbf{x}) = f_{x_i}(\mathbf{x})$, implying $f(\mathbf{x}) \in \nabla x_i[l(\mathbf{x}), u(\mathbf{x})]$. ■

Corollary A.5 *Let \mathbf{y} be composed from a subset of variables of \mathbf{x} . For an interval $[l(\mathbf{x}), u(\mathbf{x})]$ and all functions $f(\mathbf{x})$ vacuous in \mathbf{y} it must then be:*

$$f(\mathbf{x}) \in \nabla \mathbf{y}[l(\mathbf{x}), u(\mathbf{x})] \Leftrightarrow f(\mathbf{x}) \in [l(\mathbf{x}), u(\mathbf{x})]$$

Proof: (\Rightarrow). According to the Theorem A.4 sufficiency proof making an interval vacuous in a single variable leads to its shrinking. Applying this step iteratively for each variable in \mathbf{y} we must therefore obtain an interval contained in the original interval. This implies that vacuous in the \mathbf{y} variables function $f(\mathbf{x})$ from $\nabla \mathbf{y}[l(\mathbf{x}), u(\mathbf{x})]$ must be also in $[l(\mathbf{x}), u(\mathbf{x})]$.

(\Leftarrow). Suppose that vacuous in the \mathbf{y} variables function $f(\mathbf{x})$ is contained $[l(\mathbf{x}), u(\mathbf{x})]$. The necessary condition of the theorem requires us to show that function is also contained in the abstracted interval $\nabla \mathbf{y}[l(\mathbf{x}), u(\mathbf{x})]$. The proof is done by induction on the number of variables in \mathbf{y} :

(*Basis.*) The case when \mathbf{y} is composed of a single variable is been established as a necessary

condition to Theorem A.4.

(*Induction.*) Suppose now that the condition holds for $\mathbf{y} = (\mathbf{y}_1, \dots, \mathbf{y}_i)$, i.e.

$$f(\mathbf{x}) \in \nabla_{\mathbf{y}}[l(\mathbf{x}), u(\mathbf{x})]$$

According to (2.12) we may re-write above relation as

$$f(\mathbf{x}) \in [\exists \mathbf{y} l(\mathbf{x}), \forall \mathbf{y} u(\mathbf{x})]$$

If function $f(\mathbf{x})$ is also vacuous in variable y_{i+1} then according to reasoning of basis part it must be that

$$f(\mathbf{x}) \in \nabla_{y_{i+1}}[\exists \mathbf{y} l(\mathbf{x}), \forall \mathbf{y} u(\mathbf{x})]$$

Applying (2.12) and commutativity of the abstraction operation we have:

$$f(\mathbf{x}) \in \nabla_{\mathbf{y}} y_{i+1} [l(\mathbf{x}), u(\mathbf{x})]$$

This induction completes proof of the necessary condition. ■

Theorem A.6 (Theorem 2.7) *Let $f(\mathbf{x})$, $c(\mathbf{x})$, and $g(\mathbf{x})$ be functions as described above. Any function $g(\mathbf{x})$ such that*

$$g(\mathbf{x}) \in [f(\mathbf{x}) \cdot c(\mathbf{x}), f(\mathbf{x}) + \overline{c(\mathbf{x})}]$$

Proof: Suppose that $g(\mathbf{x})$ is a completely specified function selected by arbitrarily assigning don't cares of $f(\mathbf{x})$. Then following derivation holds:

$$\begin{aligned} c &\rightarrow (g = f) && \text{care set definition} \\ \Leftrightarrow c &\leq (g \equiv f) && \text{implication property} \\ \Leftrightarrow c(g \oplus f) &= \mathbf{0} && (2.3) \text{ identity} \\ \Leftrightarrow c(\bar{g} \cdot f + g \cdot \bar{f}) &= \mathbf{0} && \oplus \text{ expansion} \\ \Leftrightarrow c \cdot \bar{g} \cdot f + c \cdot g \cdot \bar{f} &= \mathbf{0} && \text{distributivity} \\ \Leftrightarrow c \cdot f \cdot \bar{g} = \mathbf{0} \wedge c \cdot \bar{f} \cdot g &= \mathbf{0} && \text{monotonicity of } + \\ \Leftrightarrow (c \cdot f \leq g) \wedge (g \leq f + \bar{c}) &&& (2.3) \text{ identity} \end{aligned}$$

This two-way derivation shows that $g(\mathbf{x})$ is in the interval if and only if it can be obtained from $f(\mathbf{x})$ restricted to $c(\mathbf{x})$, thereby proving the result. ■

Chapter 4 Proofs

Theorem A.7 (*Theorem 4.1*)

$$G(\Gamma) = \forall Z \forall \mathbf{y} \left(\overline{\exists \mathbf{x}_g \left(\sum_{i=0}^{2^s-1} m_i(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_i) \cdot \zeta_{\sigma(i)} \right)} + \overline{\exists \mathbf{x}_g \left(\sum_{i=0}^{2^s-1} m_i(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_i) \cdot \bar{\zeta}_{\sigma(i)} \right)} \right) \quad (\text{A.3})$$

Proof: The computational form of this theorem is proved by re-writing (4.16). The computation corresponding to the lower bound in that equation can be expanded as:

$$\begin{aligned} C(\mathbf{x}_g, \mathbf{y}, \Gamma) \cdot F(\mathbf{x}_g, Z) &= \left(\prod_{j=1}^t \left[y_j \equiv \sum_{i=0}^{2^s-1} \gamma_{ij} \cdot m_i(\mathbf{x}_g) \right] \right) \cdot \left(\sum_{i=0}^{r-1} M_i(\mathbf{x}_g) \cdot \zeta_i \right) \\ &= \left(\sum_{i=0}^{2^s-1} m_i(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_i) \right) \cdot \left(\sum_{i=0}^{2^s-1} m_i(\mathbf{x}_g) \cdot \zeta_{\sigma(i)} \right) \end{aligned}$$

Multiplying out the above two summation, and applying annihilation we get

$$= \sum_{i=0}^{2^s-1} m_i(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_i) \cdot \zeta_{\sigma(i)} \quad (\text{A.4})$$

We can similarly re-write complement of the upper bound part in (4.16).

$$\begin{aligned} \overline{C(\mathbf{x}_g, \mathbf{y}, \Gamma) + F(\mathbf{x}_g, Z)} \\ &= \overline{C(\mathbf{x}_g, \mathbf{y}, \Gamma) \cdot F(\mathbf{x}_g, Z)} \\ &= \sum_{i=0}^{2^s-1} m_i(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_i) \cdot \prod_{i=0}^{2^s-1} (m_i(\mathbf{x}_g) + \bar{\zeta}_{\sigma(i)}) \\ &= \prod_{i=0}^{2^s-1} \left(\sum_{j=0}^{2^s-1} m_j(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_j) \right) \cdot (\overline{m_i(\mathbf{x}_g)} + \bar{\zeta}_{\sigma(i)}) \end{aligned}$$

Multiplying out each summation with $\overline{m_i(\mathbf{x}_g)} + \bar{\zeta}_{\sigma(i)}$ gives

$$= \prod_{i=0}^{2^s-1} \left(\sum_{j=0}^{2^s-1} m_j(\mathbf{x}_g) \cdot \overline{m_i(\mathbf{x}_g)} \cdot (\mathbf{y} \equiv \gamma_j) + m_j(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_j) \cdot \bar{\zeta}_{\sigma(i)} \right)$$

Re-arranging terms in above expansion we obtain

$$= \prod_{i=0}^{2^s-1} \left[\sum_{\substack{j=0 \\ i \neq j}}^{2^s-1} m_j(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_j) + m_j(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_j) \cdot \bar{\zeta}_{\sigma(i)} \right] + m_i(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_i) \cdot \bar{\zeta}_{\sigma(i)}$$

Absorbing term $m_j(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_j) \cdot \bar{\zeta}_{\sigma(i)}$ we have

$$= \prod_{i=0}^{2^s-1} \left[\sum_{\substack{j=0 \\ i \neq j}}^{2^s-1} m_j(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_j) \right] + m_i(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_i) \cdot \bar{\zeta}_{\sigma(i)}$$

Factoring out the $m_i(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_i) \cdot \bar{\zeta}_{\sigma(i)}$ terms we have

$$= \prod_{i=0}^{2^s-1} m_i(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_i) \cdot \bar{\zeta}_{\sigma(i)} \cdot \left[\sum_{j=0}^{2^s-1} m_j(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_j) \right]$$

Multiplying out each of the 2^s summations in the above expansion we find that all terms in the square brackets are annihilated by $m_i(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_i) \cdot \bar{\zeta}_{\sigma(i)}$ terms, giving

$$= \sum_{i=0}^{2^s-1} m_i(\mathbf{x}_g) \cdot (\mathbf{y} \equiv \gamma_i) \cdot \bar{\zeta}_{\sigma(i)} \tag{A.5}$$

Substituting results of (A.4) and (A.5) into and (4.16) we have (A.1). ■

Chapter 5 Proofs

Theorem A.8 (Theorem 5.1) *If two ordered disjoint variable groups, $G_1 = \langle x_1, \dots, x_p \rangle$ and $G_2 = \langle y_1, \dots, y_p \rangle$, are symmetric in function f , then variables x_1 and y_1 must be symmetric in function*

$$f^* = \exists G_1 \setminus x_1, G_2 \setminus y_1 (f)$$

Proof: The theorem proof is done by contradiction. Suppose groups G_1 and G_2 are symmetric in f , and variables x_1 and y_1 are not symmetric in f^* . Symmetry between G_1 and G_2 implies that the cofactor matrix

$$F_{\langle G_1 \rangle, \langle G_2 \rangle} \equiv \begin{bmatrix} f_{0,0} & \boxed{\begin{matrix} \cdot & \cdot & \cdot \\ f_{i,j} & \cdot & \cdot \end{matrix}} \\ \boxed{\begin{matrix} \cdot & \cdot & \cdot \\ f_{j,i} & \cdot & \cdot \end{matrix}} & \cdot & \cdot & f_{n,n} \end{bmatrix}$$

must be symmetric, where $n = 2^p - 1$. Since in the symmetric matrix we have $\forall i, j [f_{i,j} = f_{j,i}]$ it must also be true that

$$\sum_{i < j} f_{i,j} = \sum_{i < j} f_{j,i}$$

Here the left and right hand sides correspond to the boxed summation of cofactors in the upper-right and lower-left corners of matrix $F_{\langle G_1 \rangle, \langle G_2 \rangle}$. Since x_1 and y_1 occupy the most significant bit positions in their respective groups, it is should be clear that these sums define the cofactors of f^* :

$$f^*_{\bar{x}_1 y_1} = \sum_{i < j} f_{i,j}$$

$$f^*_{x_1 \bar{y}_1} = \sum_{i < j} f_{j,i}$$

But this implies that $f^*_{\bar{x}_1 y_1} = f^*_{x_1 \bar{y}_1}$ contradicting the assumption that x_1 and y_1 are not symmetric in f^* . Therefore x_1 and y_1 must be symmetric in f^* . It is easy to show that the same holds true for any corresponding pair of variables x_i and y_i . ■

Chapter 7 Proofs

Theorem A.9 (Theorem 7.1) *The on-set minterms of $t_i(\mathbf{x}_g)$ form a subset of the equivalence class C_i in $f(\mathbf{x}_g, \mathbf{x}_h)$ if and only if for a minterm m such that $t_i(m) = 1$ following relation holds*

$$t_i(\mathbf{x}_g) \leq (f_m(\mathbf{x}_g, \mathbf{x}_h) \equiv f(\mathbf{x}_g, \mathbf{x}_h))$$

Proof: (\Rightarrow). First we show that if on-set of t_i is a subset of an equivalence class than the relation $t_i \leq (f_m \equiv f)$ must hold for any minterm m on the \mathbf{x}_g variables such that $t_i(m) = 1$. Let M_i be a function whose on-set correspond to the equivalence class C_i , and let f_i be cofactor with respect to minterms from the class. Without loss of generality let us also assume that for the given t_i relation $t_i \leq C_i$ holds. We then have expansion:

$$\begin{aligned} (f \equiv f_m) &= f \cdot f_i + \bar{f} \cdot \bar{f}_i && \text{definition of } \equiv \\ &= (M_0 f_0 + \dots + M_i f_i + \dots + M_n f_n) \cdot f_i + (\bar{M}_0 + \bar{f}_0) \cdot \dots \cdot (\bar{M}_i + \bar{f}_i) \cdot \dots \cdot (\bar{M}_n + \bar{f}_n) \cdot \bar{f}_i && \text{orthonormal expansion} \\ &= (M_0 f_0 + \dots + M_i f_i + \dots + M_n f_n) \cdot f_i + (M_1 + \dots + M_i + \dots + M_n + \bar{f}_0) \cdot \bar{f}_i \\ &\quad \cdot \dots \cdot (M_0 + \dots + M_{i-1} + M_{i+1} + \dots + M_n) \cdot \bar{f}_i \\ &\quad \cdot \dots \cdot (M_0 + \dots + M_i + \dots + M_{n-1} + \bar{f}_n) \cdot \bar{f}_i && \text{orthonormality of } M_i \text{'s} \\ &= M_0 \cdot (f_i \equiv f_0) + \dots + M_i \cdot (f_i + \bar{f}_i) + \dots + M_n \cdot (f_0 \equiv f_n) && \text{distributivity, absorption} \\ &\geq M_i && \text{annihilation, monotonicity of } + \end{aligned}$$

Since $t_i \leq M_i$, and function M_i corresponds to C_i the relation $t_i \leq (f_m \equiv f)$ must also hold.

(\Leftarrow). We now show that if subset t_i is contained in $f \equiv f_m$, then t_i is also contained in some equivalence class C_i induced by the equality of f 's cofactors. Suppose there is no such C_i , implying that there must be a pair of minterms m and m' in the on-set of t_i such that $m \in C_i$ and $m' \in C_j$ ($i \neq j$). Let also assume that $f_i = f_m$ and $f_j = f_{m'}$.

Using to the cofactor containment property, and earlier established expansion the $m' \leq (f \equiv f_m)$ relation gives us:

$$\begin{aligned} 1 &\leq ((M_0 \cdot f_0 + \dots + M_i \cdot f_i + \dots + M_j \cdot f_j + \dots + M_n \cdot f_n) \cdot f_i) + \\ &\quad (\bar{M}_0 + \bar{f}_0) \cdot \dots \cdot (\bar{M}_i + \bar{f}_i) \cdot \dots \cdot (\bar{M}_j + \bar{f}_j) \cdot \dots \cdot (\bar{M}_n + \bar{f}_n) \cdot \bar{f}_i)_{m'} \end{aligned}$$

Applying distributive property of a cofactor, and using orthogonality of M_i 's, we have:

$$1 \leq f_j \cdot f_i + \bar{f}_j \cdot \bar{f}_i$$

This however is impossible since $f_i \neq f_j$ according to the original assumption. The contradiction implies that all minterms in the on-set of t_i must belong to a single equivalence class. ■

Appendix B

Modelling Boolean Network Optimization With Intervals

The time spanning research in the field of synthesis has seen many approaches to exploring degrees of freedom in Boolean networks. As a result there exists a vast vocabulary identifying degrees of freedom with various flavors. These varying approaches to describing flexibility have common nature though: They explore conditions which cannot occur in the network. During optimization of a Boolean network these conditions descend from previously imposed optimization constraints. A common type of such constraints assumes that one part of a network has been fixed while the other has been optimized. In such a scenario the former part of a network gives rise to the degrees of freedom in the optimization of a later part. In the discussion below we show how intervals (described in Chapter 2) can capture fundamental degrees of freedom described in literature.

Single-node flexibility

A common step in the optimization of Boolean networks is to minimize the local function of each node using don't cares derived from the environment of a node. It was shown in [4] that don't cares allow to make node functions vacuous in its redundant fan-in signals. Traditionally the don't cares of a Boolean network are divided into three groups: satisfiability don't cares (SDC), observability don't cares (ODC), and external don't cares (EDC). Both SDC and ODC are related to the circuit structure, and are implicit. They include signal patterns which can never occur in the network. On the other hand, EDC are explicitly provided by a user in terms of the external signals. These three types of don't cares are described below.

Satisfiability don't cares refer to the impossible combinations of variable values for the nodes which topologically proceed some given node. For a given node function f_i and its output signal y_i , they are represented as a sum of don't care functions $y_i \oplus f_i$ for nodes enclosed between primary inputs and a node been optimized. Suppose that the following set of equations defines a Boolean network:

$$s = \bar{x}_1 \bar{x}_2 + x_1 x_2$$

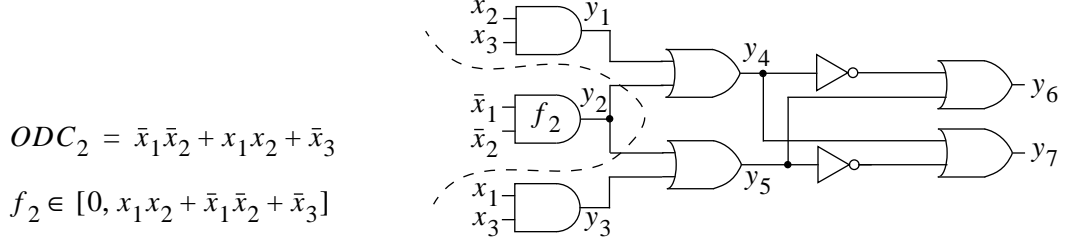


Figure B.1: Example of observability don't cares for f_2 ; they show that y_2 is redundant

$$t = x_3 x_4$$

$$k = \bar{x}_5 \bar{x}_6 + x_5 x_6$$

$$r = \bar{s}k + x_1 x_2 x_3 x_4 k + \bar{x}_1 \bar{x}_2 x_3 x_4$$

When simplifying r , its SDC is

$$d = (s \oplus (\bar{x}_1 \bar{x}_2 + x_1 x_2)) + (t \oplus (x_3 x_4)) + (k \oplus (\bar{x}_5 \bar{x}_6 + x_5 x_6))$$

Values of function r at these points can be assigned arbitrarily. Specifically, the complement of d is a care set function, and therefore according to Theorem 2.7 any new r is in the interval $[r \cdot \bar{d}, r + d]$. One such function in the interval is $r = k\bar{s} + \bar{k}st$.

Observability don't cares complement satisfiability don't cares in that they provide flexibility for single-node optimization in terms of the impossible conditions of a network part which is topologically ahead of a given node. They allow the global function at a node to change without this change becoming observable at the network outputs. These don't cares are illustrated for the example circuit given in [97] using the concept of intervals (see Figure B.1). The circuit could be a complete implementation of a function, or a subcircuit of some combinational block.

Suppose that we would like to minimize node y_2 in terms of the network part which is topologically ahead of y_2 . To compute the ODC for this node we will first compute characteristic function $F(x_1, x_2, x_3, y_2)$ of Boolean relation describing flexibility in selecting f_2 . The characteristic function of the Boolean relation is then used to extract ODC for the node.

The care set arising from this network part is represented as a function below:

$$c = (y_1 \equiv (x_2 x_3)) \cdot (y_3 \equiv (x_1 x_3)) \cdot (y_4 \equiv (y_1 + y_2)) \cdot \\ (y_5 \equiv (y_2 + y_3)) \cdot (y_6 \equiv (\bar{y}_4 + y_5)) \cdot (y_7 \equiv (y_4 + \bar{y}_5))$$

When selecting f_2 the described care set must be obeyed, while the global functions $f_6^* = x_1 + \bar{x}_2 + \bar{x}_3$ and $f_7^* = \bar{x}_1 + x_2 + \bar{x}_3$ preserved. The consistency must hold for all signal values of y_6 and y_7 . These requirements are ensured by selecting a characteristic function $F(x_1, x_2, x_3, y_2)$ from the interval below:

$$\forall y_6, y_7 [(f_6^* \equiv y_6) \cdot (f_7^* \equiv y_7) \cdot c, (f_6^* \equiv y_6) \cdot (f_7^* \equiv y_7) + \bar{c}]$$

Substituting our functions and abstracting variables we obtain:

$$F(x_1, x_2, x_3, y_2) \in [\bar{y}_2 + \bar{x}_1 \bar{x}_2 + x_1 x_2 + \bar{x}_3, \bar{y}_2 + \bar{x}_1 \bar{x}_2 + x_1 x_2 + \bar{x}_3] \quad (\text{B.1})$$

The interval contains a single characteristic function of a Boolean relation representing flexibility in selecting f_2 . (As we will observe later such uniqueness is not incidental when computing ODC.)

The definition of observability don't cares requires the values of y_2 to bear no effect on the output values, or, in other words, on the interval consistency. Therefore, all points of the observability don't cares must be able map to both \bar{y}_2 and y_2 . By abstracting y_2 from the Boolean relation we extract all such don't cares:

$$ODC_2 = \forall y_2 (\bar{y}_2 + \bar{x}_1 \bar{x}_2 + x_1 x_2 + \bar{x}_3) = \bar{x}_1 \bar{x}_2 + x_1 x_2 + \bar{x}_3$$

According to these don't cares

$$f_2 \in [(\bar{x}_1 \bar{x}_2) \cdot \overline{ODC_2}, \bar{x}_1 \bar{x}_2 + ODC_2]$$

or equivalently

$$f_2 \in [0, x_1 x_2 + \bar{x}_1 \bar{x}_2 + \bar{x}_3]$$

implying that y_2 can be selected as constant 0, and is therefore redundant.

Finally, minimization of a local node function f_i with the external don't cares $d_e(\mathbf{x})$ is performed by first expressing external don't cares in the domain of a node's fan-in variables as

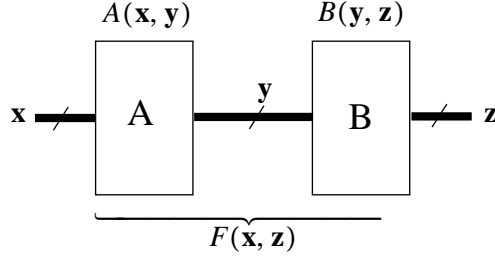
$$d_e(\mathbf{y}) = \exists \mathbf{x} (d_e(\mathbf{x}) \cdot (f_i^*(\mathbf{x}) \equiv f_i(\mathbf{y})))$$

and then combining the re-expressed don't cares with either ODC or SDC. The effect of such inclusion of EDC is an expanded interval providing more minimization flexibility.

Multi-node flexibility

In the last example we used Boolean relations to compute observability don't cares. The computed Boolean relation described a set of all compatible functions f_2 ; they represent allowable functionality for a node. Such computation extends naturally to computing flexibility for a node subset, arising due to the impossible conditions of the forward nodes. Thus, the network is partitioned into a cascade of two subnetworks, fixed and optimized, as illustrated in Figure B.2.

The two interconnected blocks in Figure B.2 depict a hierarchical representation of some Boolean network. Suppose that the functionality of the \mathbf{z} signals is provided as a function of the \mathbf{x}



a) hierarchical connections of a network

$$\begin{aligned} A(\mathbf{x}, \mathbf{y}) &= \forall \mathbf{z} (F(\mathbf{x}, \mathbf{z}) + \overline{B(\mathbf{y}, \mathbf{z})}) \\ &= \exists \mathbf{z} (F(\mathbf{x}, \mathbf{z}) \cdot B(\mathbf{y}, \mathbf{z})) \end{aligned}$$

b) flexibility for block **A**, when block **B** and $F(\mathbf{x}, \mathbf{z})$ are given**Figure B.2: Illustration of Boolean relations in capturing flexibility of a Boolean network**

signals, and that block **B** is fixed while **A** is been optimized. Cerny [27] formulated all possible implementations of **A** for such an interconnection¹ in terms of Boolean relations in the context of Boolean equations. His approach uses the characteristic function of a Boolean relation to model consistent behavior of inputs and outputs for a given block. Let $A(x, y)$, $B(x, y)$ and $F(x, z)$ be such functions for the blocks **A**, **B**, and a complete network, respectively. Cerny shows that all implementations of **A** are expressed in terms of B and F as

$$A(\mathbf{x}, \mathbf{y}) = \forall \mathbf{z} (F(\mathbf{x}, \mathbf{z}) + \overline{B(\mathbf{y}, \mathbf{z})}) \quad (\text{B.2})$$

All functions compatible with A are valid implementations of **A**.

The equation (B.2) derives directly from the interval solution to **A**. Indeed, when B is viewed as a care set function for F the interval solution for A is:

$$A(\mathbf{x}, \mathbf{y}) \in \forall \mathbf{z} [F(\mathbf{x}, \mathbf{z}) \cdot B(\mathbf{y}, \mathbf{z}), F(\mathbf{x}, \mathbf{z}) + \overline{B(\mathbf{y}, \mathbf{z})}] \quad (\text{B.3})$$

Equation (B.2) then follows directly from the proposition below:

Proposition B.1 [27] *If $B(\mathbf{y}, \mathbf{z})$ is characteristic function for a completely specified block **B**, then*

$$\exists \mathbf{z} (F(\mathbf{x}, \mathbf{z}) \cdot B(\mathbf{y}, \mathbf{z})) = \forall \mathbf{z} (F(\mathbf{x}, \mathbf{z}) + \overline{B(\mathbf{y}, \mathbf{z})})$$

According to this proposition, the characteristic function A is unique and can be obtained with two different computational forms. Note that this is consistent with the solution (B.1) for the earlier ODC example – the interval also contains only one function.

The notion of Boolean relations, however, breaks when we reverse the problem solving for block **B** in terms of block **A**. Savoj showed that expressing implementations for **B** with Boolean

1. However in his approach block **B** also depends on \mathbf{x} . This requirement is not arbitrary as it ensures that Boolean relations are well-formed when solving for **B** in terms of **A**.

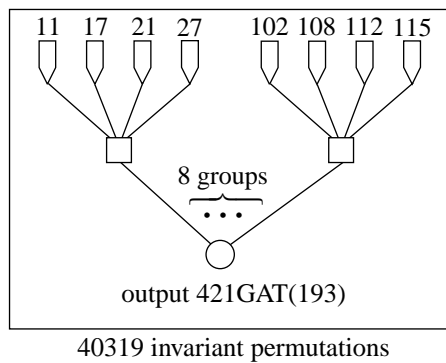
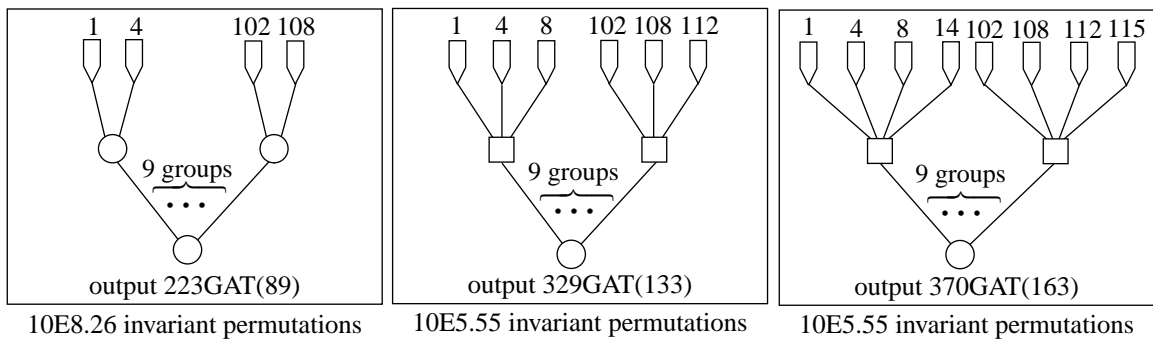
relations is not adequate [97, p. 61]. The reason for the insufficiency of Boolean relations is that they may become not well-formed, which may allow implementations of block that **B** are not consistent with **A**. Cerny avoids this problem by allowing **B** depend on the signals of **x**. Such explicit dependence on **x** ensures that the Boolean relation of $B(\mathbf{x}, \mathbf{y}, \mathbf{z})$ is well-formed. However, this way block **B** is redundant in **x** as any of its signals are also allowed to pass through block **A**. Chapter 4 addresses this problem expressing flexibility for each individual output in block **B** by means of intervals. The the notion of intervals is also used in Chapter 4 to express more general flexibility where only signal dependence between the blocks **A** and **B** is known, allowing them change simultaneously.

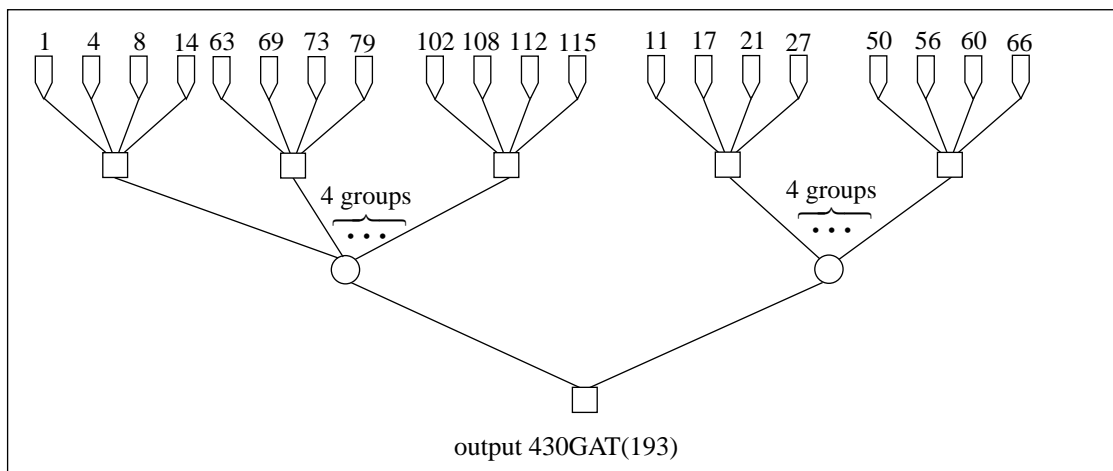
Appendix C

Symmetry Structures

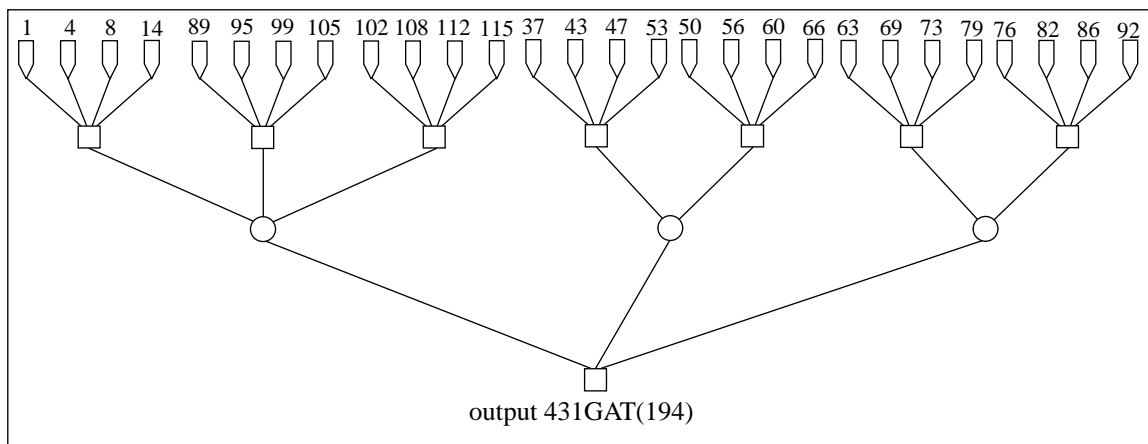
Symmetry structures for C432

Below we give symmetry structures for each of the seven outputs in the $c432$ circuit. They illustrate hierarchical symmetries which arise in datapath circuits.

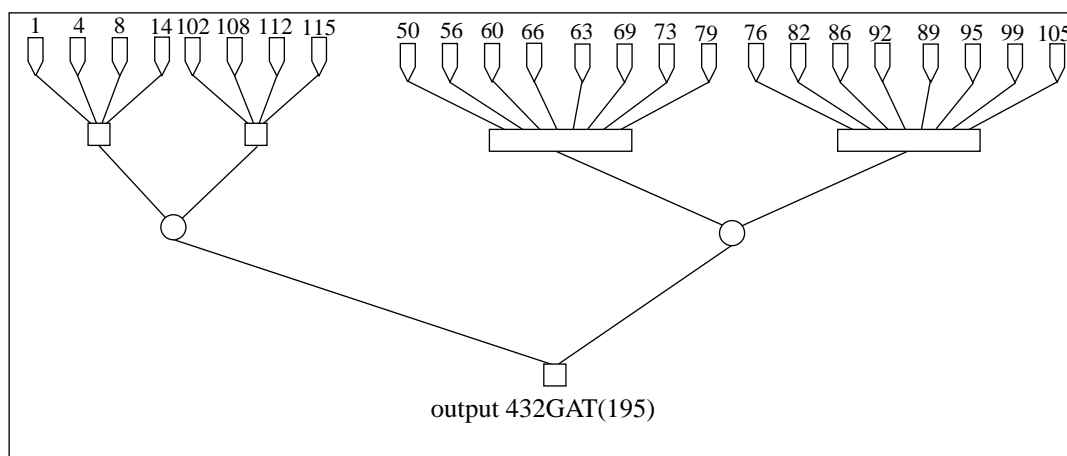




2879 invariant permutations



23 invariant permutations



3 invariant permutations

Symmetry structures for C499 based on its high-level decomposition

Symmetries of the circuit are described, relying on the signal naming from [57].

The *M1* module implements syndrome equations. We first give symmetries for some of its internal signals:

$$\begin{aligned}
 D_0 &: \{ID_{00}, ID_{04}, ID_{08}, ID_{12}, ID_{16}, ID_{17}, ID_{18}, ID_{19}, ID_{20}, ID_{21}, ID_{22}, ID_{23}\} \\
 D_1 &: \{ID_{01}, ID_{05}, ID_{09}, ID_{13}, ID_{24}, ID_{25}, ID_{26}, ID_{27}, ID_{28}, ID_{29}, ID_{30}, ID_{31}\} \\
 D_2 &: \{ID_{02}, ID_{06}, ID_{10}, ID_{14}, ID_{16}, ID_{17}, ID_{18}, ID_{19}, ID_{24}, ID_{25}, ID_{26}, ID_{27}\} \\
 D_3 &: \{ID_{03}, ID_{07}, ID_{11}, ID_{15}, ID_{20}, ID_{21}, ID_{22}, ID_{23}, ID_{28}, ID_{29}, ID_{30}, ID_{31}\} \\
 D_4 &: \{ID_{16}, ID_{20}, ID_{24}, ID_{28}, ID_{00}, ID_{01}, ID_{02}, ID_{03}, ID_{04}, ID_{05}, ID_{06}, ID_{07}\} \\
 D_5 &: \{ID_{17}, ID_{21}, ID_{25}, ID_{29}, ID_{08}, ID_{09}, ID_{10}, ID_{11}, ID_{12}, ID_{13}, ID_{14}, ID_{15}\} \\
 D_6 &: \{ID_{18}, ID_{22}, ID_{26}, ID_{30}, ID_{00}, ID_{01}, ID_{02}, ID_{03}, ID_{08}, ID_{09}, ID_{10}, ID_{11}\} \\
 D_7 &: \{ID_{19}, ID_{23}, ID_{27}, ID_{31}, ID_{04}, ID_{05}, ID_{06}, ID_{07}, ID_{12}, ID_{13}, ID_{14}, ID_{15}\}
 \end{aligned}$$

These are *multi-phase* symmetries (i.e. symmetries which hold under any phase assignments to the group variables). The remaining inputs to this module have the following symmetries:

$$\begin{aligned}
 H_0 &: \{R, IC_0\} & H_2 &: \{R, IC_2\} & H_4 &: \{R, IC_4\} & H_6 &: \{R, IC_6\} \\
 H_1 &: \{R, IC_1\} & H_3 &: \{R, IC_3\} & H_5 &: \{R, IC_5\} & H_7 &: \{R, IC_7\}
 \end{aligned}$$

D_i and H_i form multi-phase symmetries:

$$\begin{aligned}
 S_0 &: \{D_0, C_0\} & S_2 &: \{D_2, C_2\} & S_4 &: \{D_4, C_4\} & S_6 &: \{D_6, C_6\} \\
 S_1 &: \{D_1, C_1\} & S_3 &: \{D_3, C_3\} & S_5 &: \{D_5, C_5\} & S_7 &: \{D_7, C_7\}
 \end{aligned}$$

The S_i syndrome signals have the following symmetries in the *M2* module:

$$\begin{aligned}
 E_{00} &: \{S_0, \bar{S}_1, \bar{S}_2, \bar{S}_3, S_4, \bar{S}_5, S_6, \bar{S}_7\} & E_{16} &: \{S_0, \bar{S}_1, S_2, \bar{S}_3, S_4, \bar{S}_5, \bar{S}_6, \bar{S}_7\} \\
 E_{01} &: \{\bar{S}_0, S_1, \bar{S}_2, \bar{S}_3, S_4, \bar{S}_5, S_6, \bar{S}_7\} & E_{17} &: \{S_0, \bar{S}_1, S_2, \bar{S}_3, \bar{S}_4, S_5, \bar{S}_6, \bar{S}_7\} \\
 E_{02} &: \{\bar{S}_0, \bar{S}_1, S_2, \bar{S}_3, S_4, \bar{S}_5, S_6, \bar{S}_7\} & E_{18} &: \{S_0, \bar{S}_1, S_2, \bar{S}_3, \bar{S}_4, \bar{S}_5, S_6, \bar{S}_7\} \\
 E_{03} &: \{\bar{S}_0, \bar{S}_1, \bar{S}_2, \bar{S}_3, S_4, \bar{S}_5, S_6, \bar{S}_7\} & E_{19} &: \{S_0, \bar{S}_1, S_2, \bar{S}_3, \bar{S}_4, \bar{S}_5, \bar{S}_6, \bar{S}_7\} \\
 E_{04} &: \{S_0, \bar{S}_1, \bar{S}_2, \bar{S}_3, S_4, \bar{S}_5, \bar{S}_6, S_7\} & E_{20} &: \{S_0, \bar{S}_1, \bar{S}_2, S_3, S_4, \bar{S}_5, \bar{S}_6, \bar{S}_7\} \\
 E_{05} &: \{\bar{S}_0, \bar{S}_1, \bar{S}_2, \bar{S}_3, S_4, \bar{S}_5, \bar{S}_6, S_7\} & E_{21} &: \{S_0, \bar{S}_1, \bar{S}_2, S_3, \bar{S}_4, S_5, \bar{S}_6, \bar{S}_7\} \\
 E_{06} &: \{\bar{S}_0, \bar{S}_1, S_2, \bar{S}_3, S_4, \bar{S}_5, \bar{S}_6, \bar{S}_7\} & E_{22} &: \{S_0, \bar{S}_1, \bar{S}_2, \bar{S}_3, \bar{S}_4, \bar{S}_5, S_6, \bar{S}_7\} \\
 E_{07} &: \{\bar{S}_0, \bar{S}_1, \bar{S}_2, S_3, S_4, \bar{S}_5, \bar{S}_6, S_7\} & E_{23} &: \{S_0, \bar{S}_1, \bar{S}_2, S_3, \bar{S}_4, \bar{S}_5, \bar{S}_6, S_7\}
 \end{aligned}$$

$$\begin{array}{ll}
E_{08}:\{S_0, \bar{S}_1, \bar{S}_2, \bar{S}_3, \bar{S}_4, S_5, S_6, S_7\} & E_{24}:\{\bar{S}_0, S_1, S_2, \bar{S}_3, S_4, \bar{S}_5, \bar{S}_6, \bar{S}_7\} \\
E_{09}:\{\bar{S}_0, \bar{S}_1, \bar{S}_2, \bar{S}_3, \bar{S}_4, S_5, S_6, \bar{S}_7\} & E_{25}:\{\bar{S}_0, S_1, S_2, \bar{S}_3, \bar{S}_4, S_5, \bar{S}_6, \bar{S}_7\} \\
E_{10}:\{\bar{S}_0, \bar{S}_1, S_2, \bar{S}_3, \bar{S}_4, S_5, S_6, \bar{S}_7\} & E_{26}:\{\bar{S}_0, S_1, S_2, \bar{S}_3, \bar{S}_4, \bar{S}_5, S_6, \bar{S}_7\} \\
E_{11}:\{\bar{S}_0, \bar{S}_1, \bar{S}_2, S_3, \bar{S}_4, S_5, S_6, \bar{S}_7\} & E_{27}:\{\bar{S}_0, S_1, S_2, \bar{S}_3, \bar{S}_4, \bar{S}_5, \bar{S}_6, S_7\} \\
E_{12}:\{S_0, \bar{S}_1, \bar{S}_2, \bar{S}_3, \bar{S}_4, S_5, \bar{S}_6, S_7\} & E_{28}:\{\bar{S}_0, S_1, \bar{S}_2, S_3, S_4, \bar{S}_5, \bar{S}_6, \bar{S}_7\} \\
E_{13}:\{\bar{S}_0, S_1, \bar{S}_2, \bar{S}_3, \bar{S}_4, S_5, \bar{S}_6, S_7\} & E_{29}:\{\bar{S}_0, S_1, \bar{S}_2, S_3, \bar{S}_4, S_5, \bar{S}_6, \bar{S}_7\} \\
E_{14}:\{\bar{S}_0, \bar{S}_1, S_2, \bar{S}_3, \bar{S}_4, S_5, \bar{S}_6, S_7\} & E_{30}:\{\bar{S}_0, S_1, \bar{S}_2, S_3, \bar{S}_4, \bar{S}_5, S_6, \bar{S}_7\} \\
E_{15}:\{\bar{S}_0, \bar{S}_1, \bar{S}_2, S_3, \bar{S}_4, S_5, \bar{S}_6, S_7\} & E_{31}:\{\bar{S}_0, S_1, \bar{S}_2, S_3, \bar{S}_4, \bar{S}_5, \bar{S}_6, S_7\}
\end{array}$$

The E_i and ID_i signals form the following multi-phase symmetry groups:

$$\begin{array}{llll}
OD_{00}:\{E_{00}, ID_{00}\} & OD_{08}:\{E_{08}, ID_{08}\} & OD_{16}:\{E_{16}, ID_{16}\} & OD_{24}:\{E_{24}, ID_{24}\} \\
OD_{01}:\{E_{01}, ID_{01}\} & OD_{09}:\{E_{09}, ID_{09}\} & OD_{17}:\{E_{17}, ID_{17}\} & OD_{25}:\{E_{25}, ID_{25}\} \\
OD_{02}:\{E_{02}, ID_{02}\} & OD_{10}:\{E_{10}, ID_{10}\} & OD_{18}:\{E_{18}, ID_{18}\} & OD_{26}:\{E_{26}, ID_{26}\} \\
OD_{03}:\{E_{03}, ID_{03}\} & OD_{11}:\{E_{11}, ID_{11}\} & OD_{19}:\{E_{19}, ID_{19}\} & OD_{27}:\{E_{27}, ID_{27}\}
\end{array}$$

Symmetry profile for C6288 expressed in terms of its partial product

Symmetry profiles in the table below are given for a 16-bit multiplier in terms of its partial product signals. Each row in the table identifies a list of group variables that are symmetric using $n(s)$ notation, where n is the number of groups of size s .

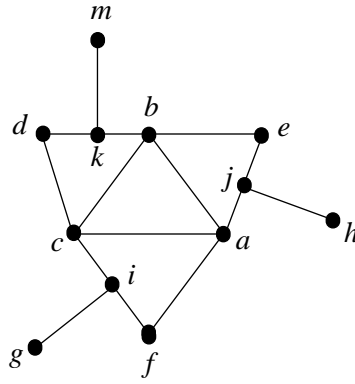
Output	Symmetry Profile
545GAT(287)	1(2)
1581GAT(423)	1(2)
1901GAT(561)	1(2) 1(3)
2223GAT(700)	1(2) 1(3) 1(4)
2548GAT(840)	1(2) 1(3) 1(4) 1(5)
2877GAT(983)	1(2) 1(3) 1(4) 1(5) 1(6)
3211GAT(1128)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7)
3552GAT(1275)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 1(8)
3895GAT(1423)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 1(8) 1(9)
4241GAT(1572)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 1(8) 1(9) 1(10)
4591GAT(1722)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 1(8) 1(9) 1(10) 1(11)
4946GAT(1876)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 1(8) 1(9) 1(10) 1(11) 1(12)
5308GAT(2031)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 1(8) 1(9) 1(10) 1(11) 1(12) 1(13)
5672GAT(2187)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 1(8) 1(9) 1(10) 1(11) 1(12) 1(13) 1(14)
5971GAT(2309)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 1(8) 1(9) 1(10) 1(11) 1(12) 1(13) 1(14) 1(15)
6123GAT(2368)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 1(8) 1(9) 1(10) 1(11) 1(12) 1(13) 1(14) 1(15) 1(16)
6150GAT(2378)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 1(8) 1(9) 1(10) 1(11) 1(12) 1(13) 1(14) 2(15) 1(16)
6160GAT(2383)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 1(8) 1(9) 1(10) 1(11) 1(12) 1(13) 2(14) 2(15) 1(16)
6170GAT(2388)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 1(8) 1(9) 1(10) 1(11) 1(12) 2(13) 2(14) 2(15) 1(16)
6180GAT(2393)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 1(8) 1(9) 1(10) 1(11) 2(12) 2(13) 2(14) 2(15) 1(16)
6190GAT(2398)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 1(8) 1(9) 1(10) 2(11) 2(12) 2(13) 2(14) 2(15) 1(16)
6200GAT(2403)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 1(8) 1(9) 2(10) 2(11) 2(12) 2(13) 2(14) 2(15) 1(16)
6210GAT(2408)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 1(8) 2(9) 2(10) 2(11) 2(12) 2(13) 2(14) 2(15) 1(16)
6220GAT(2413)	1(2) 1(3) 1(4) 1(5) 1(6) 1(7) 2(8) 2(9) 2(10) 2(11) 2(12) 2(13) 2(14) 2(15) 1(16)
6230GAT(2418)	1(2) 1(3) 1(4) 1(5) 1(6) 2(7) 2(8) 2(9) 2(10) 2(11) 2(12) 2(13) 2(14) 2(15) 1(16)
6240GAT(2423)	1(2) 1(3) 1(4) 1(5) 2(6) 2(7) 2(8) 2(9) 2(10) 2(11) 2(12) 2(13) 2(14) 2(15) 1(16)
6250GAT(2428)	1(2) 1(3) 1(4) 2(5) 2(6) 2(7) 2(8) 2(9) 2(10) 2(11) 2(12) 2(13) 2(14) 2(15) 1(16)
6260GAT(2433)	1(2) 1(3) 2(4) 2(5) 2(6) 2(7) 2(8) 2(9) 2(10) 2(11) 2(12) 2(13) 2(14) 2(15) 1(16)
6270GAT(2438)	1(2) 2(3) 2(4) 2(5) 2(6) 2(7) 2(8) 2(9) 2(10) 2(11) 2(12) 2(13) 2(14) 2(15) 1(16)
6280GAT(2443)	1(2) 2(3) 2(4) 2(5) 2(6) 2(7) 2(8) 2(9) 2(10) 2(11) 2(12) 2(13) 2(14) 2(15) 1(16)
6287GAT(2444)	1(1) 2(2) 2(3) 2(4) 2(5) 2(6) 2(7) 2(8) 2(9) 2(10) 2(11) 2(12) 2(13) 2(14) 2(15) 1(16)
6288GAT(2447)	1(1) 2(2) 2(3) 2(4) 2(5) 2(6) 2(7) 2(8) 2(9) 2(10) 2(11) 2(12) 2(13) 2(14) 2(15) 1(16)

A function which has no involution symmetries

Below we give an example function whose symmetry cannot be represented with our swap-based symmetry structures; its SOP representation is

$$ab + ac + af + aj + bc + be + bk + cd + ci + dk + ej + fi + gi + hj + km$$

The semantic structure of the function is captured in the following graph:.



It is constructed such that its vertices are adjacent if and only if their variables belong the same cube of the SOP representation. In the cycle notation [55] the symmetry of this function is

$$(abc)(dfe)(ghm)(ijk)$$

which is clearly not representable with the variable swaps. The example is due to Markov [79].

Appendix D

Synthesis Data

Symmetric library computation

Below the module libraries are computed for the 3-to-2 symmetric decomposition:

```
m31> symm_modules 3 2
Creating Lower bound...bdd size 47
Creating complement of Upper bound...bdd size 47
ORing Lower bound and complement of Upper bound...bdd size 101
Creating G for all pattern functions...bdd size 37
Decoding library solution...bdd size 4
Module libraries are:
  {M1}, {M2}, {M3}
Total of 3 functionally complete module libraries
  where modules are:
    M2 = < x0*x1*x2 + x0'*x1'*x2',
           x0'*x1' + x0'*x2' + x1'*x2' >
        = < S(0,3), S(0,1) >
    M1 = < x0*x1*x2' + x0*x1'*x2 + x0'*x1*x2 + x0'*x1'*x2',
           x0'*x1' + x0'*x2' + x1'*x2' >
        = < S(0,2), S(0,1) >
    M3 = < x0*x1*x2 + x0'*x1'*x2',
           x0*x1*x2' + x0*x1'*x2 + x0'*x1*x2 + x0'*x1'*x2' >
        = < S(0,3), S(0,2) >
Total 3 module types
m31>
```

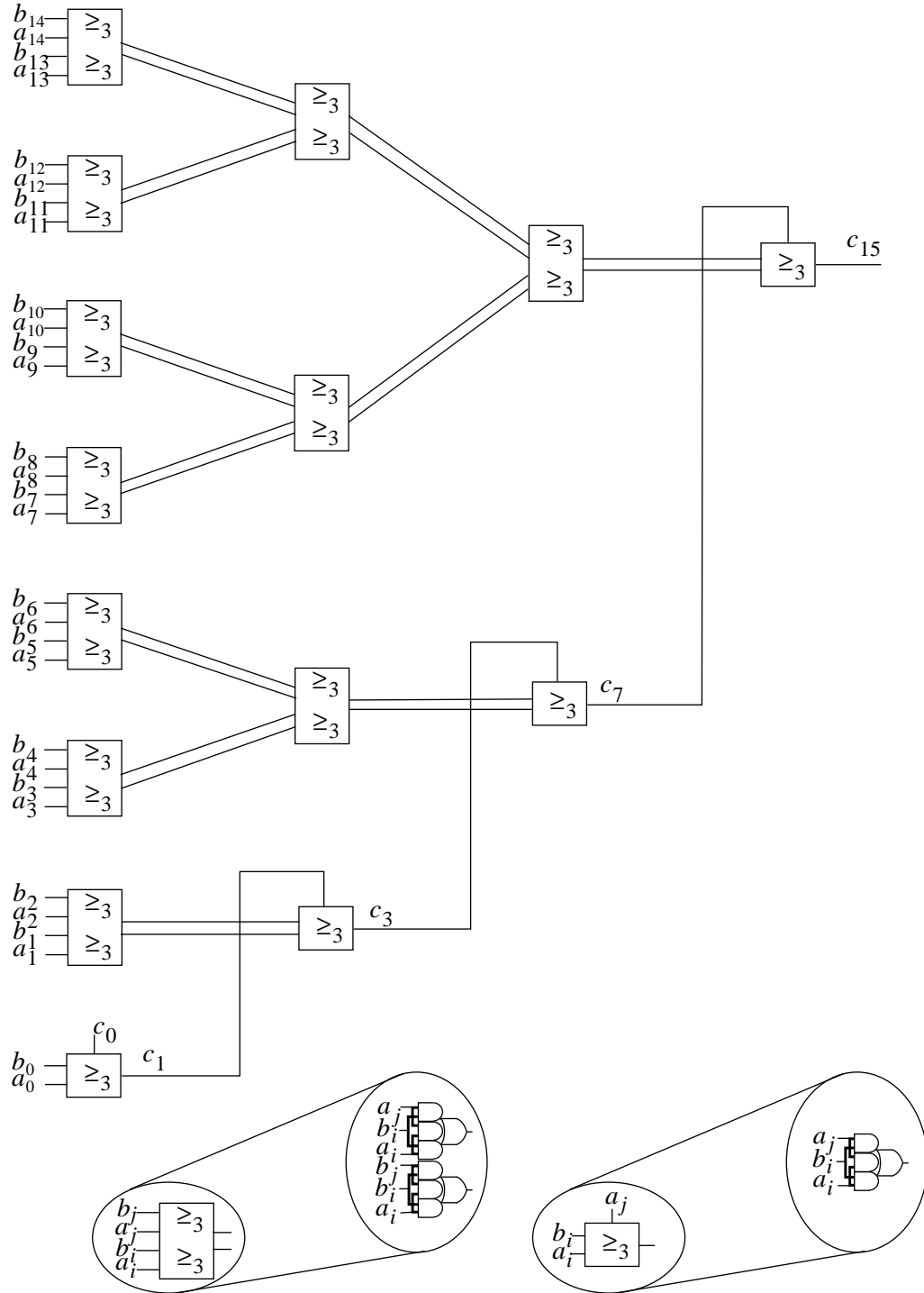
The output shows that there is total of three libraries, each composed of a single 2-output module.

Below the module library are computed for the 2-to-1 symmetric decomposition:

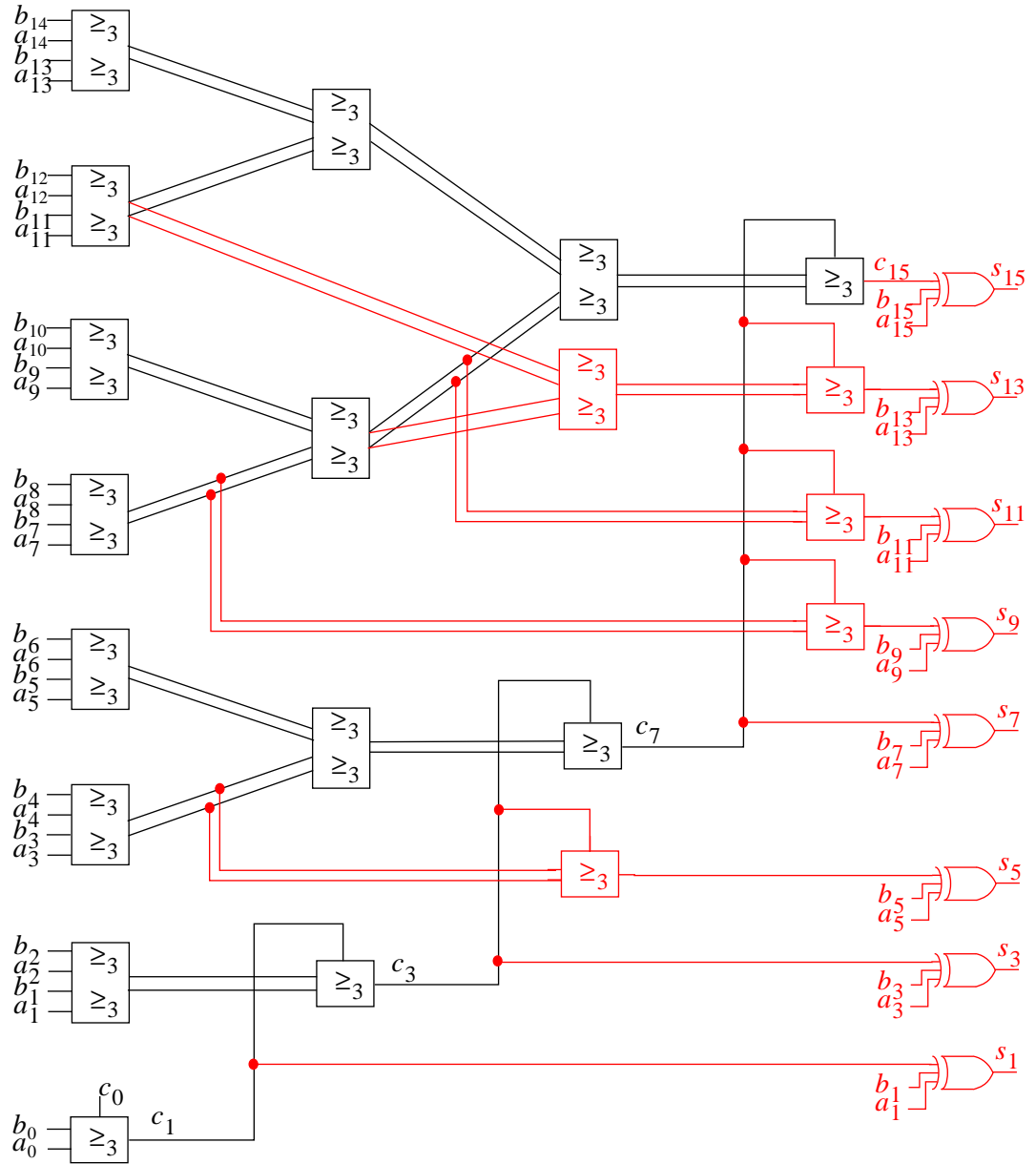
```
m31> symm_modules 2 1
Creating Lower bound...bdd size 13
Creating complement of Upper bound...bdd size 13
ORing Lower bound and complement of Upper bound...bdd size 23
Creating encoding condition for the pattern functions...bdd size 13
Adding encoding condition...bdd size 53
Creating G for all pattern functions...bdd size 16
Deriving feasible pattern functions...bdd size 4
Creating transitive closure...bdd size 5
Deleting subsumed pattern functions...bdd size 4
Deriving weakest pattern functions...bdd size 5, 3 subproblems
Computing complete library.....bdd size 4
Module libraries are:
    {M1,M2,M3}
Total of 1 functionally complete module libraries
    where modules are:
        M3 = < x0*x1 + x0'*x1' >
            = < S(0,2) >
        M2 = < x0'*x1' >
            = < S(0) >
        M1 = < x0' + x1' >
            = < S(0,1) >
Total 3 module types
m31>
```

Structure of the synthesized 16-bit adder

The 16-bit adder synthesized by M31 has 5 levels of logic. It splits computation of the even and odd sum bits into two independent subcircuits. Its carry signals are computed as a tree of majority gates. Such a tree is given below for the c_{15} signal.

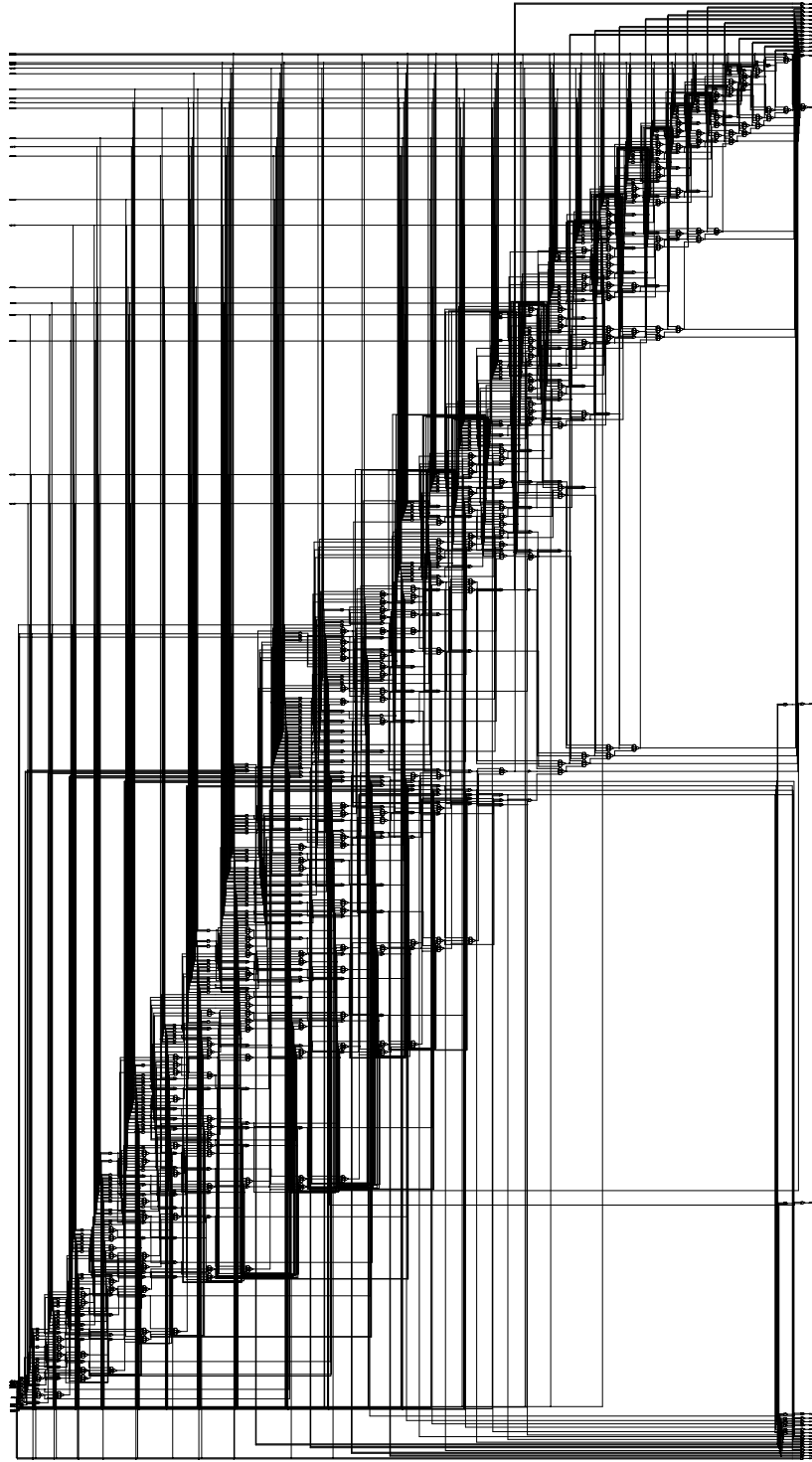


The largest carry trees in the adder correspond to the c_{15} and c_{16} signals; their corresponding depth is $\lceil \log(14) \rceil = 4$ and $\log(16) + 1 = 5$. Tree of the remaining carry signals share subtrees of c_{15} and c_{16} . This sharing is illustrated in the figure below. It depicts computation of odd logic in the adder.



Synthesized 16-bit multiplier

Schematic for a 16-bit multiplier synthesized with M31. Its circuit has a regular, array style structure composed of the majority and the exclusive-or gates. The circuit has 738 gates including 256 2-input AND, 224 3-input XOR, and 224 3-input MAJORITY; it has 31 levels of logic on its longest path.



Amount of synthesized logic

The table below gives data for the amount of logic synthesized with fan-in-bounded (bound of four) and support-reducing decomposition using M31 on the flattened multi-level MCNC benchmark. In the synthesis of these benchmarks the `mcnc+` library (see page 119 of Chapter 7 for its description) was used. The first three columns in the table characterize a benchmark in terms of its name, number of inputs, and number of outputs. Columns 4 and 5 give BDD sizes of the largest network nodes before and after running M31. Column 5 gives number of nodes that were not completely decomposed by M31 due to the fan-in and support-reducing constraints. The last column provides percent of logic synthesized with M31 measured using formula $((|F_0| - |F|)/|F_0|) \cdot 100\%$; $|F_0|$ and $|F|$ in the formula denote, respectively, sizes of shared BDDs for the original logic and the final remaining unimplemented logic.

Circuit			Largest size of a node BDD		# of not fully decomposed nodes	% of synthesized logic
name	inputs	outputs	before	after		
9symml	9	1	24	4	0	100.0
alu2	10	6	86	86	3	4.3
alu4	14	8	135	135	5	3.0
apex6	135	99	107	23	2	97.5
apex7	49	37	42	6	0	100.0
b1	3	4	4	3	0	100.0
b9	41	21	34	6	0	100.0
c8	28	18	13	4	0	100.0
cc	21	20	11	4	0	100.0
cm138a	6	8	6	4	0	100.0
cm150a	21	1	34	3	0	100.0
cm152a	11	1	16	3	0	100.0
cm162a	14	5	17	4	0	100.0
cm163a	16	5	14	4	0	100.0
cm42a	4	10	4	4	0	100.0
cm82a	5	3	7	4	0	100.0
cm85a	11	3	14	4	0	100.0
cmb	16	4	12	4	0	100.0
comp	32	3	47	6	0	100.0
cordic	23	2	48	4	0	100.0
count	35	16	21	4	0	100.0
cu	14	11	18	15	1	81.7
des	256	245	65	44	120	54.9
dalu	75	16	139	157	16	18.8
example2	85	66	28	17	1	97.9
f51m	8	8	20	20	4	16.4
frg2	143	139	59	70	7	90.4
i1	25	16	21	4	0	100.0
i2	201	1	1481	4	0	100.0
i3	132	6	32	4	0	100.0
i4	192	6	132	4	0	100.0

Circuit			Largest size of a node BDD		# of not fully decomposed nodes	% of synthesized logic
name	inputs	outputs	before	after		
i5	133	66	34	4	0	100.0
k2	45	43	222	331	25	14.4
lal	26	19	22	4	0	100.0
majority	5	1	7	6	0	100.0
mux	21	1	36	3	0	100.0
my_adder	33	17	49	4	0	100.0
pair	173	137	177	224	71	50.2
parity	16	1	16	3	0	100.0
pcler8	27	17	28	4	0	100.0
pm1	16	13	12	4	0	100.0
rot	135	107	905	723	28	38.9
sct	19	15	25	4	0	100.0
t481	16	1	78	4	0	100.0
term1	34	10	197	9	1	99.2
too_large	38	3	865	502	3	43.7
ttt2	24	21	32	21	2	85.6
unreg	36	16	12	4	0	100.0
vda	17	39	113	108	30	32.5
x1	51	35	316	50	3	92.0
x2	10	7	16	11	2	59.6
x3	135	99	167	42	2	94.9
x4	94	71	31	6	0	100.0
z4ml	7	4	10	4	0	100.0

Bibliography

Bibliography

- [1] S. B. Akers. Binary Decision Diagram. *IEEE Transactions on Computers*, C-27(6):509-516, June 1978.
- [2] R. L. Ashenhurst. The decomposition of switching functions. In *Proceedings of the International Symposium on the Theory of Switching (April 2-5, 1957)*, pp. 74-116. Annals of Computation Laboratory of Harvard Univ., vol. 29. 1959.
- [3] K. Bartlett, W. Cohen, A. de Geus, and G. Hachtel. Synthesis and optimization of multilevel logic under timing constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, CAD-5(4):582-596, 1986.
- [4] K. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. Multi-level logic minimization using implicit don't cares. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, CAD-7(6):723-740, June 1988.
- [5] C. L. Berman and L. Trevillyan. A global approach to circuit size reduction. In MIT Press, editor, *Advanced Research in VLSI, 5th MIT Conference*, pp. 69-99, 1989.
- [6] C. L. Berman and L. Trevillyan. Global flow optimization in automatic logic design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 10(5):557-562, May 1991.
- [7] V. Bertacco, S. Minato, P. Verplaeste, L. Benini, and G. De Micheli. Decision diagrams and pass transistor logic synthesis. In *International Workshop on Logic Synthesis*.
- [8] V. Bertacco and M. Damiani. The disjunctive decomposition of logic functions. In *Proc. International Conference on Computer-Aided Design*, pp. 78-82, November 1997.
- [9] T. Besson, H. Bouzouzou, M. Crastes, I. Floricica, G. Saucier. Synthesis on multiplexer-based FPGA using binary decision diagrams. In *Proc. International Conference on Computer-Aided Design*, pp. 163-167, November 1992.
- [10] G. Boole. *An Investigation of the Laws of Thought*. Walton, London, 1854. (reprinted by Dover, New York, 1954).
- [11] D. Bostick, C. Hachtel, R. Jacoby, M. Lightner, P. Moceyunas, C. Morrison, and D. Ravenscroft. The Boulder optimal logic design system. In *Proc. International Conference on Computer-Aided Design*, pp. 62-65, November 1987.
- [12] P. Buch, A. Narayan, A. R. Newton, A. Sangiovanni-Vincentelli. Logic synthesis for large pass transistor circuits. In *Proc. International Conference on Computer-Aided Design*, pp. 663-670, November 1997.

- [13] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. 27th Design Automation Conference*, 40-45, June 1990.
- [14] R. K. Brayton and C. McMullen. The decomposition and factorization of Boolean expressions. In *Proc. IEEE International Symposium on Circuits and Systems*, pp. 49-54, May 1982.
- [15] R. K. Brayton, J. D. Cohen, G. D. Hachtel, B. M. Trager, D. Y. Y. Yun. Fast recursive Boolean function manipulation. In *Proc. IEEE International Symposium on Circuits and Systems*, pp. 58-62, May 1982.
- [16] R. K. Brayton, G. D. Hachtel, L. A. Hemachandra, A. R. Newton, and A. L. M. Sangiovanni-Vincentelli. A comparison of logic minimization strategies using ESPRESSO. In *Proc. IEEE International Symposium on Circuits and Systems*, pp. 42-48, May 1982.
- [17] R. K. Brayton and C. McMullen. Synthesis and optimization of multistage logic. In *Proc. International Conference on Computer Design*, pp. 23-28, October 1984.
- [18] R. K. Brayton, C. McMullen, G. D. Hachtel, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers for VLSI Synthesis, 1984.
- [19] R. K. Brayton, C. L. Chen, C. McMullen, R. H. J. M. Otten, and Y. J. Yamour, Automated implementation of switching functions as dynamic CMOS circuits, In *Proc. Custom Integrated Circuit Conference*, pp. 346-350, 1984.
- [20] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic optimization and the rectangle covering problem. In *Proc. International Conference on Computer-Aided Design*, November 1987.
- [21] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 6:1062-1081, November 1987.
- [22] R. K. Brayton and F. Somenzi. Boolean relations and the incomplete specification of logic networks. In *Proc. International Conference on VLSI*, pp. 231-240, August 1989.
- [23] F. M. Brown. *Boolean Reasoning*. Kluwer Academic Publishers, Boston, 1990.
- [24] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(6):677-691, August 1986.
- [25] R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, C-40(2):205-213, February 1991.
- [26] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293-318, September 1992.
- [27] E. Cerny and M. A. Marin. An approach to unified methodology of combinational switching

- circuits. *IEEE Transactions on Computers*, 26(8):745-756, 1977.
- [28] S.-C. Chang and M. Marek-Sadowska. Technology mapping via transformations of functional graphs. In *Proc. International Conference on Computer-Aided Design*, pp. 159-162, November 1992
 - [29] S. Chang, D. I. Cheng, and M. Marek-Sadowska. Minimizing ROBDD size of incompletely specified multiple output functions. In *Proc. European Design and Test Conference*, pp. 620-624, 1994.
 - [30] S. Chattopadhyay, S. Roy, and P. P. Chaudhuri. KGPMIN: An efficient Multilevel Multioutput AND-OR-XOR minimizer. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 16(3):257-265, March 1997.
 - [31] R. Chaudhry, T.-H. Liu, A. Aziz, and J. L. Burns. Area-oriented synthesis for pass-transistor logic. In *Proc. International Conference on Computer Design*, pp. 160-167, 1998.
 - [32] D. I. Cheng and M. Marek-Sadowska. Verifying equivalence of functions with unknown input correspondence. In *Proc. European Design Automation Conference*, pp. 81-85, Paris, France, February 1993.
 - [33] K.-T. Cheng and L. A. Entrena. Multi-level logic optimization by redundancy addition and removal. In *Proc. European Conference on Design Automation*, February 1993.
 - [34] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proc. Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, 1989.
 - [35] H. A. Curtis. *New Approach to the Design of Switching Circuits*. Van Nostrand, Princeton, NJ, 1962.
 - [36] M. Damiani and G. De Micheli. Observability don't care sets and Boolean relations. In *Proc. International Conference on Computer-Aided Design*. pp. 502-505, November 1990.
 - [37] J. Darringer, W. Joyner, L. Berman, and L. Trevillyan. LSS: Logic synthesis through local transformations. *IBM Journal of Research and Development*, 25(4):272-280, July 1981.
 - [38] E. Davidson. An algorithm for NAND decomposition under network constraints. *IEEE Transactions on Computers*, C-18(12):1098-1109, December 1969.
 - [39] M. Davio, J.-P. Deschamps and A. Thayse. *Discrete and Switching Functions*. McGraw-Hill, New York, 1978.
 - [40] S. Devadas. Boolean decomposition in multi-level logic optimization. In *Proc. International Conference on Computer-Aided Design*, pp. 290-293. November 1988.
 - [41] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. Perkowski. Efficient representation and manipulation of switching functions based on ordered Kronecker functional decision diagrams. In *Proc. 31st Design Automation Conference*, pp. 415-419, June 1994.

- [42] C. R. Edwards and S. L. Hurst. A digital synthesis procedure under function symmetries and mapping methods. *IEEE Transactions on Computers*, C-27:985-997, 1978.
- [43] *EPOCH User's Manual*, ver. 3.2. Cascade Design Automation, Bellevue, WA 98006, 1995.
- [44] J. P. Fishburn. LATTIS: An iterative speedup heuristic for mapped logic. In *Proc. 29th Design Automation Conference*, pp. 488-491, June 1992.
- [45] G. Fleisher and L. Maissel. An introduction to array logic. *IBM Journal of Research and Development*, 19:98-109, March 1975.
- [46] A. A. Fraenkel. *Abstract Set Theory*. North-Holland Publishing, Amsterdam, 1976.
- [47] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of Boolean comparison method based on binary decision diagrams. In *Proc. International Conference on Computer-Aided Design*, pp. 2-5, November 1988.
- [48] The GAP Group. *GAP – Groups, Algorithms, and Programming*. ver. 4.2. Aachen, St. Andrews, 2000. (<http://www-gap.dcs.st-and.ac.uk/~gap>)
- [49] M. J. Ghazala. Irredundant disjunctive and conjunctive forms of a Boolean function. *IBM Journal of Research and Development*, vol. 1, pp. 443-458, April 1957.
- [50] J. Gimpel. The minimization of TANT networks. *IEEE Transactions on Electronic Computers*, EC-14:535-541, August 1965.
- [51] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel. SOCRATES: A system for automatically synthesizing and optimizing combinational logic. *Proc. 23th Design Automation Conference*, pp. 580-586, June 1986.
- [52] P. R. Halmos. *Naive Set Theory*. Springer-Verlag, New York, 1974.
- [53] G. Hachtel, R. M. Jacoby, K. Keutzer, and C. R. Morrison. On properties of algebraic transformations and the synthesis of multifault-irredundant circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 11(3):313-321, March 1992.
- [54] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Boston, 1996.
- [55] M. Hall. *The Theory of Groups*. Macmillan, New York, 1957.
- [56] M. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Design and Test of Computers*, 16(3):72-80, July 1999.
- [57] J. P. Hayes. *ISCAS High-Level Models*. <http://www.eecs.umich.edu/~jhayes/iscas/benchmark.html>.
- [58] L. Hellerman. A catalog of three-variable or-invert and and-invert logic circuits. *IEEE Transactions on Electronic Computers*, EC-12(3):198-223, June 1963.

- [59] H. Higuchi and F. Somenzi. Lazy group sifting for efficient symbolic state traversal of FSMs. In *Proc. International Conference on Computer-Aided Design*, pp. 45-49, November 1993.
- [60] U. Hinsberger and R. Kolla. Boolean matching for large libraries. In *Proc. 35th Design Automation Conference*, pp. 206-211, June 1998.
- [61] Y. Hong, P. A. Beerel, J. R. Burch, and K. L. McMillan. Safe BDD minimization using don't cares. In *Proc. 34th Design Automation Conference*, pp. 208-212, June 1997.
- [62] E. V. Huntington. Sets of independent postulates for algebra of logic. *Trans. Amer. Math. Soc.*, vol. 5, pp. 228-309, 1904.
- [63] J. Ishikawa, H. Sato, M. Hiramane, K. Ishida, S. Oguri, Y. Kazuma, and S. Murai. A rule-based reorganization system LORES/EX. In *Proc. International Conference on Computer Design*, pp. 262-266, October 1988.
- [64] S.-W. Jeong, T.-S. Kim, and F. Somenzi. An efficient method for optimal BDD ordering computation. In *International Conference on VLSI and CAD (ICVC'93)*, Taejon, Korea, November 1993.
- [65] G. Kamhi and L. Fix. Adaptive variable ordering for ordered binary decision diagrams. In *Proc. IEEE International Conference on Computer-Aided Design*, pp. 359-365, November 1998.
- [66] K. Keutzer, A. R. Newton, and N. Shenoy. The future of logic synthesis and physical design in deep-submicron process geometries. In *ISPD'97*, pp. 218-224, 1997.
- [67] B.-G. Kim and D. L. Dietmeyer. Multilevel logic synthesis of symmetric switching functions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 10(4):436-446, April 1991.
- [68] K. Knopp, *Theory of Functions*, Dover, New York, 1996.
- [69] V. N. Kravets and K. A. Sakallah. *Generalized symmetries in Boolean functions*. Number CSE-TR-420-00. University of Michigan, Ann Arbor, MI 48109, February 2000.
- [70] P. Kurup and T. Abbasi. *Logic Synthesis Using Synopsys*. Kluwer Academic Publishers, 1997.
- [71] Y. T. Lai, M. Pedram, and Sarma B. K. Vrudhula. BDD based decomposition of logic functions with application to FPGA synthesis. In *Proc. 30th Design Automation Conference*, pp. 642-647, June 1993.
- [72] E. L. Lawler. An approach to multilevel Boolean minimization. *Journal of the ACM*, 11:283-295, July 1964.
- [73] C. Y. Lee. Representation of switching circuits by binary decision diagrams. *Bell Syst. Tech. J.* 38:985-999, 1959.

- [74] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness. Logic decomposition during technology mapping. In *Proc. International Conference on Computer-Aided Design*, pp. 264-271, November 1995.
- [75] T.-H. Liu, A. Aziz, and J. L. Burns. Performance driven synthesis for pass-transistor logic. In *International Workshop on Logic Synthesis*, pp. 255-259, June 1998.
- [76] E. M. Luks. Hypergraph isomorphism and structural equivalence of Boolean functions. In *Proc. Symposium on Theory of Computing*, pp. 652-658, May 1999.
- [77] F. Mailhot and G. D. Micheli. Technology mapping using Boolean matching. In *Proc. European Design Automation Conference*, pp. 180-185, March 1990.
- [78] S. Malik, A. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in logic synthesis environment. In *Proc. International Conference on Computer-Aided Design*, pp. 6-9, November 1988.
- [79] I. L. Markov. Private communication. October, 2000.
- [80] E. J. McCluskey. Minimization of Boolean functions. *Bell System Technical Journal*, 35:1417-1444, April 1956.
- [81] E. J. McCluskey. *Logic Design Principles with Emphasis on Testable Semicustom Circuits*. Prentice-Hall, New Jersey, 1986.
- [82] P. McGeer and R. K. Brayton. The observability don't care set and its approximations. In *Proc. International Conference on Computer Design*, pp. 502-505, November 1989.
- [83] C. Meinel and A. Slobodova. Speeding up variable reordering of OBDDs. In *Proc. International Conference on Computer Design*, pp. 338-343, October 1997.
- [84] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *Proc. 27th Design Automation Conference*, pp. 52-57, June 1990.
- [85] J. Mohnke, P. Molitor, and S. Malik. Limits of using signatures for permutation independent Boolean Comparison. In *Proc. Asia and South Pacific Design Automation*, pp. 459-464, August 1995.
- [86] D. Moller, J. Mohnke, and M. Weber. Detection of symmetry of Boolean functions represented by ROBDDs. In *Proc. International Conference on Computer-Aided Design*, pp. 680-684, October 1993.
- [87] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimum functional decompositions using encoding. In *Proc. Design Automation Conference*, pp. 408-414, June 1994.
- [88] 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*, C-38(10):1404-1424, October 1989.

- [89] S. Panda, F. Somenzi, and B. F. Plessier. Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams. In *Proc. International Conference on Computer-Aided Design*, pp. 628-631, November 1994.
- [90] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *Proc. International Conference on Computer-Aided Design*, pp. 74-77, November 1995.
- [91] W. Quine. The problem of simplifying truth functions. *Amer. Math. Monthly*, 59:521-531, 1952.
- [92] J. P. Roth and R. Karp. Minimization over Boolean graphs. *IBM Journal of Research and Development*, 6(2):227-238, April 1962.
- [93] S. Rudeanu. *Boolean Functions and Equations*. North-Holland, Amsterdam, 1974.
- [94] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. International Conference on Computer Aided Design*, pp. 42-47, November 1993.
- [95] K. A. Sakallah. *Functional abstraction and partial specification of Boolean functions*. Number CSE-TR-255-95. University of Michigan, Ann Arbor, MI 48109, August 1995.
- [96] H. Savoj, H. Touati, and R. K. Brayton. Extracting local don't cares for network optimization. In *Proc. International Conference on Computer-Aided Design*, pp. 514-517, November 1991.
- [97] H. Savoj. *Don't Cares in Multi-Level Network Optimization*. PhD thesis, University of California, Berkeley, 1992.
- [98] J. Saul. An algorithm for the multi-level minimization of Reed-Muller representations. In *Proc. International Conference on Computer-Aided Design*, pp. 634-637, November 1991.
- [99] H. Sawada, T. Suyama, and A. Nagoya. Logic synthesis for look-up table based FPGAs using functional decomposition and support minimization. In *Proc. International Conference on Computer-Aided Design*, pp. 353-358, November 1995.
- [100] P. R. Schneider and D. L. Dietmeyer. An algorithm for synthesis of multiple-output combinational logic. *IEEE Transactions on Computers*, C-17(2):117-128, February 1968.
- [101] C. Scholl. Multi-output functional decomposition with exploration of don't cares. In *Proc. Design, Automation and Test in Europe Conference*, pp. 743-748, February 1998.
- [102] C. Scholl, D. Moller, P. Molitor, and R. Drechsler. BDD minimization using symmetries. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 18(2):81-100, February 1999.
- [103] E. M. Sentovich. *SIS: A system for sequential circuit synthesis*. Number UCB/ERL M92/41. UC Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, May 1992.
- [104] E. M. Sentovich, V. Singhal, and R. K. Brayton. Multiple Boolean Relations. In *Interna-*

tional Workshop on Logic Synthesis, May 1993.

- [105] C. E. Shannon. A symbolic analysis of relay and switching circuits. *AIEE Trans.* 57:713-723, 1938.
- [106] T. R. Shiple, R. Hajati, A. L. Sangiovanni-Vincentelli, and R. K. Brayton. Heuristic minimization of BDDs using don't cares. In *Proc. 31st Design Automation Conference*, pp. 225-231, June 1994.
- [107] K. J. Singh, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *Proc. International Conference on Computer-Aided Design*, pp. 282-285, November 1991.
- [108] A. Slobodova and C. Meinel. Sample method for minimization of OBDDs. In *International Workshop on Logic Synthesis*, May 2000.
- [109] L. H. Soicher. GRAPE: a system for computing with graphs and groups. In L. Finkelstein and W. M. Kantor, editors. *Groups and Computation*, vol. 11 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pp. 287-201. ACM, 1993.
- [110] F. Somenzi. *CUDD: CU Decision Diagram Package. ver. 2.3.0*. University of Colorado, Boulder, September, 1998. (<http://vlsi.colorado.edu/~fabio/CUDD>)
- [111] L. Stok, D. S. Kung, D. Brand, A. D. Drumm, A. J. Sullivan, L. N. Reddy, N. Hieter, D. J. Gieger, H. H. Chao, and P. J. Osler. BooleDozer: Logic synthesis for ASICs. *IBM Journal of Research and Development*, 40(4):407-430, July 1986.
- [112] M. H. Stone. Subsumption of Boolean algebras under the theory of rings. In *Proc. Nat. Acad. Sci., USA*, vol. 21, pp. 103-105, 1935.
- [113] M. H. Stone. The theory of representations for Boolean algebras. *Trans. Amer. Math. Soc.*, vol. 40, pp.37-111, 1936.
- [114] H. J. Touati, H. Savoj, and R. K. Brayton. Delay Optimization of Combinational Logic Circuits and Partial Collapsing. In *Proc. 28th Design Automation Conference* pp. 188-191, June 1991.
- [115] C.-C. Tsai and M. Marek-Sadowska. Boolean matching using generalized Reed-Muller forms. In *Proc. 31rd Design Automation Conference*, pp. 339-344, June 1994.
- [116] C.-C. Tsai and M. Marek-Sadowska. Multilevel logic synthesis for arithmetic functions. In *Proc. 33rd Design Automation Conference*, pp. 242-247, June 1996.
- [117] C.-C. Tsai and M. Marek-Sadowska. Generalized Reed-Muller forms as a tool to detect symmetries. *IEEE Transactions on Computers*, C-45(1):772-781, August 1996.
- [118] H. Vaishnav and M. Pedram. Minimizing the routing cost during logic extraction. In *32nd Design Automation Conference*, pp. 70-75, June 1995.
- [119] A. Wang. *Algorithms for Multi-Level Logic Optimization*. PhD thesis, University of Califor-

nia, Berkeley, 1989.

- [120] I. Wegener. *The Complexity of Boolean Functions*. Wiley-Teubner, 1987.
- [121] B. Wurth, K. Eckl, and K. Antreich. Functional multiple-output decomposition: theory and an implicit algorithm. In *Proc. 32nd Design Automation Conference*, pp. 54-59, June 1995.
- [122] C. Yang, M. Ciesielski, and V. Singhal. BDS: A BDD-based logic optimization system. In *Proc. 37th Design Automation Conference*, pp. 92-97, June 2000.
- [123] S. Yang. *Logic synthesis and optimization benchmarks user guide – version 3.0*. Microelectronics Center of North Carolina, Research Triangle Park, NC, January 1991.
- [124] Y. Ye and K. Roy. A graph-based synthesis algorithm for AND/XOR networks. In *Proc. 34th Design Automation Conference*, pp. 107-112, June 1997.
- [125] K. Yoshikawa, H. Ichiryu, H. Tanishita, S. Suzuki, N. Nomizu, and A. Kondoh. A depth-decreasing heuristic for combinational logic; or how to convert a ripple-carry adder into a carry-lookahead adder or anything in-between. In *Proc. 28th Design Automation Conference*, pp. 112-117, June 1991.