PERFORMANCE AND TESTABILITY INTERACTIONS IN LOGIC SYNTHESIS

by

Alexander Saldanha

Memorandum No. UCB/ERL M91/100

28 October 1991

PERFORMANCE AND TESTABILITY INTERACTIONS IN LOGIC SYNTHESIS

by

Alexander Saldanha

Memorandum No. UCB/ERL M91/100

28 October 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

Performance and Testability Interactions in Logic Synthesis

Alexander Saldanha

University of California
Berkeley, California

Department of Electrical Engineering
and Computer Sciences
Computer Science Division

Abstract

Three primary parameters that are optimized at all levels of synthesis and design of very large scale integrated circuits are area, performance, and testability. While techniques for individually optimizing each of these parameters are well formulated, not much is known of the interactions between the parameters. There are two important reasons why this interaction is interesting. First, it is unknown whether circuits may be fully optimized simultaneously for area, performance and testability, or whether a tradeoff between the criteria necessarily exists, *i.e.* is the optimality of one parameter sacrificed when the others are optimized? Second, the impact of the tradeoff between the area, performance or testability on the resulting quality of the optimized integrated circuit is not well understood. This thesis studies the interaction between the performance, testability, and area of combinational logic circuits. The results apply to sequential circuits as well. There are three contributions; these are motivated by the example of a well-known circuit design, where testability and reliability are apparently sacrificed for performance. The first result proves that there exists a fully testable implementation for every high-performance and untestable circuit. An algorithm that transforms an untestable circuit to a fully testable circuit with no loss in performance is provided. A consequence of this result is the question of whether the testability of a circuit can be retained during performance optimization. The second part of this thesis explores several synthesis situations and provides a comprehensive summary of the testability effects of performance optimization techniques. The synthesis techniques discussed in the first two parts require the analysis of two critical problems that have emerged recently, timing analysis and delay-fault test generation. The final contribution of this thesis is the development of a novel and efficient general framework to solve the class of problems, that includes both timing analysis and delay-fault test generation, involving functional analysis on paths in a circuit.

Prof. Robert K. Brayton
Thesis Committee Chairman

Performance and Testability Interactions in Logic Synthesis

Acknowledgements

What in the dark
I had taken to be a stump of a little tree
appearing above the snow,
to which I had tied my horse,
proved to have been the weathercock of the church steeple.
Rudolf Raspe in "Travels of Baron Munchausen"

The area of performance and testability interactions in logic synthesis seemed like a little stump when I first started research work in the CAD group at Berkeley. There are several people to be acknowledged for their support, encouragement, and friendship in being a part of my "adventure".

Bob Brayton has been my research advisor for over three years. He has been a constant source of guidance, support, and enthusiasm for me. The many stimulating discussions, directions, and ideas he has always provided are directly reflected in much of the work reported in this thesis. I am especially grateful for his meticulous reading of several drafts of each paper that we have written. From him I have learned the importance of mathematically sound and precise presentation.

Alberto Sangiovanni-Vincentelli has been more than just a second member of my thesis committee. I am indebted to him for his initial support during my early years at Berkeley. I thank him for guiding and directing me into the area of testing and its relationship to logic synthesis. From him I have strived to learn the ability to relate solutions for particular problems to more general situations in creating a coherent "big picture". I am grateful for the opportunity provided by Alberto and the experience gained in teaching the two logic synthesis courses at DEC - Hudson in 1990 and 1991.

Dorit Hochbaum has been a member of both my qualifying examination and thesis committees. I am thankful to her for the time spent in serving both roles, as well as the discussions and suggestions she has made on my work.

Thanks to Randy Katz for the support during the year I spent with him completing my master's project. He also served on my qualifying examination committee.

A lot of the work reported here has been done with the assistance of others. Bob Brayton and Alberto Sangiovanni-Vincentelli have supervised or been part of all the work. The summer of 1989 was spent at AT&T Bell Laboratories with Sharad Malik and Kurt Keutzer, where the initial results on the interaction between redundancy and delay were developed. The testability invariance results were done together with Tim Cheng. The path-recursive paradigm was developed with Rick McGeer. Paul Stephan helped develop the SAT package used in many of the results reported here. Tiziano Villa worked with me on the encoding problem that is not reported here.

I have several people to thank for assisting in the process of completing my thesis. Tiziano Villa has constantly followed my work and personal life. I am grateful to him for his friendship and assistance, especially during the start of my stay in Berkeley. His wide interests have always provided a welcome break from dreary moments. Luciano Lavagno has patiently listened to many of my ideas and made helpful suggestions on drafts of papers and this thesis. Thanks to K.J. Singh, Narendra Shenoy, and Ellen Sentovich for making our office a nice place to be and for spending many times together at the coffee shop. Rick McGeer has been a constant source of ideas and I continue to enjoy our lively discussions and collaboration. The members of the CAD group at Berkeley are thanked for their interaction and companionship over the past few years. They are: Pranav Ashar, Wendell Baker, Mark Beardslee, Andrea Casotto, Srinivas Devadas, Abhijit Ghosh, Chuck Kring, Luciano Lavagno, Tony Ma, Sharad Malik, Rick McGeer, Cho Moon, Rajeev Murgai, Hamid Savoj, Ellen Sentovich, Narendra Shenoy, Kanwar Jit Singh, Rick Spickelmier, Paul Stephan, Hervé Touati, Tiziano Villa, Albert Wang, Yosinori Watanabe, and Greg Whitcomb. Many thanks to Flora Oviedo, Kathryn Crabtree, Irena Stanczyk-Ng, and Elise Mills for the all administrative assistance provided. Brad Krebs helped with many hardware and software problems over the years.

The research support for my work has been primarily from the Semiconductor Research Corporation under contract 90-DC-008. I am grateful for the support provided by Al Dunlop at AT&T Bell Laboratories - Murray Hill, during my stay in 1989. Tushar Gheewala and Susheel Chandra are acknowledged for my stay at CrossCheck Inc., San Jose, during 1990.

My life at Berkeley would not have been as happy and memorable as it has been without my friends from Communion and Liberation. In particular, words cannot express my gratitude to Roberto and Elena, Paul, Damian, Chris, Martin, Tiziano and Maria, Bruno and Kristi, Marco and Antonella, Fr. Mark, Mark and Margaret, Alessandra, Lucia, and many others. I thank them for showing me what is most important in life and for their constant and complete human friendship.

My wife Avril has patiently watched as my work has evolved over the past four years. I thank her above all for her constant love and support over these years, but especially in the past six months.

This thesis is dedicated to my parents, Eric and Doreen Saldanha. Together with my brothers and sisters, Val, Pete and Maria, Ritchie and Roopa, Joan and Ajit, Linda and Trevor, Maria, and their kids, and my "new" family of Alex and Jeannette Lobo, Nisha and Elga, they are constantly in my memory.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis explores the interaction between the three primary optimization criteria of logic synthesis during the design of *very large scale integration* (VLSI) digital circuits, namely performance, testability, and area. The interactions between these criteria are studied primarily in combinational logic circuits, yet also apply to sequential logic. This chapter is organized as follows. In Section 1.1 the role and goals of logic synthesis in the design of VLSI circuits are described. The problems that arise in combinational logic synthesis are briefly reviewed in Section 1.2, with an overview of previous work on these problems. A brief introduction to the interactions between the various optimization criteria is provided in Section 1.3. Section 1.4 describes the scope and organization of this thesis.

1.1 CAD of IC's

Computer-aided design (CAD) of digital integrated circuits (IC's) has proven to be extremely successful for a number of reasons. The most significant of these are the ability to automatically design highly complex IC's, low turnaround times, automatic verification, and better chip quality. While significant improvements have been made individually in each of these aspects of CAD of IC's over the past few years, most of the focus presently is on the quality of a chip design. The principal goals of a high quality IC are the area of the layout, the speed of the circuit and the reliability of the manufactured chip. Good chip quality almost always results in the ability to create a more complex design, with better verification capabilities and lower turnaround times. Chip quality depends to a large extent on the level of sophistication and optimization that is provided by the algorithms in the different stages of synthesis. In this thesis, the quality of a circuit is focused upon by studying the interactions between the three optimality criteria of area, performance and testability. The design flow in the complete synthesis process of an IC is briefly described below, focusing on the how each of these optimization criteria are addressed at each level.

High-level Synthesis: The initial step in the synthesis of a digital system involves the translation of an abstract specification of the system (referred to as a *behavioral specification*) to a structural description. The resulting structural description is typically specified as an interconnection of combinational logic blocks and memory elements, termed a *register transfer-level* description. The function of each combinational block may be specified as a set of logic equations or as well-known functional blocks, *e.g.* a 32-bit adder. [73] provides a tutorial of the most significant behavioral synthesis algorithms and results. The first step of translating a specification to an intermediate register transfer-level description involves optimizations akin to compiler optimizations. The target optimization criterion may be either the area or performance of the resulting description. The two core transformations used in high-level synthesis are classified as resource *scheduling* and *allocation*. Scheduling is performed to assign operations to control steps so as to minimize the delay, subject to some constraints on the amount of available hardware resources. Allocation is the problem of minimizing the amount of hardware needed. While most early synthesis systems performed these two optimization steps independently, all current systems address the area-performance tradeoff that arises at the behavioral synthesis level by relating the optimizations across the two steps. The impact on testability of design choices at this level of synthesis is still emerging as an issue.

Sequential Synthesis: The input description at this level of synthesis is typically a register transfer (or symbolic) description. Often this description is expressed as a finite state machine. Sequential synthesis techniques are used to create a logic level implementation by performing a binary assignment (or encoding) of the states of the symbolic description. There are several techniques that target the optimization of the area of the resulting implementation. These include the minimization of the number of symbolic states in the register transfer description (*state minimization*) [52], the encoding of symbolic states (*state assignment*) [34], and the decomposition and factorization of a large state machine into smaller interacting state machines (*state decomposition*) [52]. A technique for improving the testability of the resulting sequential circuit has also been proposed [39]. While each of the techniques listed above have some impact on the performance of the resulting circuit, sequential synthesis for performance optimization is not well understood as yet.

Logic Synthesis: *Logic synthesis* is the process of converting a logical description of a circuit to a technology specific interconnection of gates that realize an equivalent function. In this step of IC synthesis, the optimization goals of minimum area, minimum delay, and complete testability are most directly targeted. The design optimization performed at this level profoundly affects the quality of the final chip with respect to the three optimization goals. The interaction played out between these three axes of optimization at this level is the focus of this thesis.

Layout Synthesis: The final step of synthesis of an integrated circuit prior to manufacture is the process of arranging the network of gates onto actual silicon. This physical synthesis consists in mapping the gates into actual transistors and interconnections among them. The steps in this process include module and gate placement, and global and local routing. [62] provides a complete review of this area. A lot of the problems in this stage of synthesis have mature and widely used solutions. However, an important problem that is emerging at this level is the ability to provide the logic and behavioral level synthesis tools fast and accurate estimates of the area and delay corresponding to various choices of structures during optimization. Recent work includes algorithms for timing driven placement [54], and proposals for fast and accurate estimates of area and performance at the layout level to direct performance oriented logic synthesis operations [86]. This stage of synthesis has an impact on the testability of the IC related to defects arising due to the physical proximity of transistors and wires (*e.g. bridging faults*); however, results relating the effects of layout synthesis on testability are lacking.

A more complete description of performance-directed synthesis at all levels of the design of VLSI systems is provided in [2].

1.2 Combinational logic synthesis

Given a functional description of a system that includes memory constructs, combinational logic synthesis extracts only the combinational portion of the logic for optimization. The memory elements are connected back into the final optimized circuit at the end of the process. [16] is a complete description of the algorithms and approaches used in this level of synthesis. In this section, the three most common goals of any logic optimization problem are considered; the focus is on techniques and results relevant to the subject of this thesis.

1.2.1 Area optimization

By far the best understood aspect of combinational logic synthesis is the manipulation of logic equations to yield an implementation of minimal area. When the target technology is a two-level implementation both exact and heuristic algorithms are well established [30, 15]. In two-level implementations, the area of an implementation is proportional to the number of terms. A secondary function is the number of occurrences of the variables (referred to as the number of literals). In multilevel logic, exact minimization algorithms are much harder to achieve than for two-level logic, since the solution space is considerably larger due to increased degrees of freedom compared to two-level logic [16]. However, several techniques exist that yield sufficiently high-quality area-minimal solutions. In a multilevel implementation, area is most often estimated by the number of literals in the implementation. For multilevel implementations there are two basic approaches that are adopted. The first is a rule-based approach consisting of the application of selected transformations from a given collection of rules developed by experienced circuit designers [32, 7]. The second is algorithmic based [18, 5]. Synthesis systems based on this approach have proved substantially superior to those based on the former [16]. Many industrial synthesis systems employ the second technique followed by the first.

A successful strategy employed in the latter approach is to decompose the process into two steps: technology independent optimization and technology dependent optimization. This often simplifies the design and optimization problem to be solved, while still yielding an efficient solution. Technology independent operations, which apply to generic gates independent of technology specific information, may be further classified as algebraic and Boolean. While algebraic operations restrict the set of operations that are used to optimize circuit structure, they are popular due to the time-efficiency of the process. They can be performed in polynomial time in the number of variables of a function [111]. Boolean operations are more time-consuming, yet are essential in obtaining minimal circuits [97]. The technology dependent optimizations consist of *mapping* the generic gates to a specific library of cells, corresponding to a target technology [35].

1.2.2 Performance optimization

Performance optimization is almost always the primary optimization criteria in logic designs [2, 16] (subject to some area constraints). A typical problem is to improve the delay of an existing circuit structure. At the technology independent level this is done by incremental modifications to the structure of the network to yield a *faster* circuit. Three transformations [105, 78, 11], that have recently matured into efficient and feasible algorithms to reduce the delay of circuits are studied in detail in this thesis. At the technology dependent level, delay may be minimized instead of area during the *mapping* phase [108]. An alternate technique, that is often used in conjunction with mapping, is the insertion of buffer cells with high capacitative-drive properties to further improve delays through gates propagating signals to several different parts of the circuit [10].

While a reasonable first-order approximation of the delay of a circuit is the number of levels of gates that a signal passes through, it is known that capacitative loading effects, functional considerations, and other operating factors (*e.g.* transistor gate sizes) must also be taken into account [76]. In this thesis, performance estimates are made after the incorporation of all these factors. In fact, the core problems addressed in subsequent chapters result from the interactions of these factors during delay estimation of high-performance circuits. In particular, the interaction between the logical and timing behavior is related to the impact on the testability and area of a circuit.

1.2.3 Testability optimization

Testability refers to the ability to determine whether an IC is behaving in accordance with the given specifications. Most approaches to digital IC testing before the middle of the last decade attempted to improve the testability of the design by ad-hoc post-synthesis modifications. However, with persistently increasing VLSI densities, the increasing need for reliability in manufactured circuits has led to the evolution of testability as an important logic optimization criterion.

In order to refer to the ability to test a chip a fault model is required. Several fault models are in use today. The most common is the stuck-fault model that detects static (or DC) defects [19]. However, since the delay of a chip is often as critical as its logical behavior, circuits also have to be tested for dynamic (or AC) defects [106]. This has led to the definition of two delay fault models. Even more comprehensive detection of manufacturing defects may be achieved by checking for faults that model open and shorted connections within transistors. One goal when optimizing combinational circuits for testability is to ensure that 100% of all faults being modeled can be tested by applying a suitable test vector sequence at the primary inputs of the circuits. A defect is detected as a logical difference at some primary output. Other important considerations are the number of test vectors required to detect all the faults, the computation effort required in generating the tests, and the time required to apply them.

There are well developed synthesis techniques for all of the fault models listed above. Most are best understood for two-level circuits [6, 51, 36]. However, several multilevel optimization operations are known that may be used to retain testability in circuits [51, 36, 94, 20].

1.3 Relations between optimization criteria

The goal of logic optimization is to obtain a design that is fully optimized with respect to all three criteria; yet, all the optimization techniques mentioned in the previous section target only one of the three criteria. However, when optimizing for one of the goals, sometimes the effects on the remaining criteria are known or may be predicted. A brief description of some of these known results is presented below.

1.3.1 Area and testability

The area of a circuit is directly related to its testability. Here testability means the percentage of faults for which there exists a test vector which tests for the fault. The reason for this is as follows. Consider a connection which can be set to a constant value without affecting the functionality. In such a case, this connection may as well be replaced by the constant value, thus resulting in a smaller circuit. If the connection is retained in the circuit then a manufacturing defect that appears on the connection cannot be tested; this leads to an untestable circuit. Untestable circuits are undesirable for several reasons which are explored in detail in the next chapter. Besides a non area-minimal implementation, untestability impedes the test generation process and degrades the reliability of a chip. The circuit resulting from the replacement of the connection by a constant is more testable than the original circuit. There are several optimality criteria that can be used to relate area to testability. The first order optimality criterion for area is related to 100% *single stuck-fault testability* [6]. In such a circuit, no single connection can be removed without changing the function of the circuit. A second order form of optimality is related to the more stringent testability criterion of 100% *multiple stuck-fault testability* [51]. In such a circuit no set of connections can be simultaneously removed from a circuit without changing the function of the circuit.

1.3.2 Area and delay

Any of the approaches described in Section 1.2.2 can be used to achieve an improvement in the delay characteristic of a circuit, albeit with some penalty in area of the resulting implementation [105]. An example of this phenomenon is the several different implementations of an adder circuit. A ripple-carry adder has the least area among all adders, but it is also the slowest. A carry-skip adder [66] is faster and is derived using an additional AND gate and MUX. A carry-select adder [112], is similarly derived from a ripple-carry adder with some duplication of logic to achieve a reduction in speed. A circuit with even better delay uses a carry-lookahead structure, [112], but at substantially higher cost in area.

Synthesis techniques also show the same phenomenon. An example of the area-delay tradeoff is shown in Figure 1.1, where a 32-bit ripple carry adder is optimized for delay using path reduction techniques [105]. The experiment is performed using a parameter to control the strength of the optimizations used in restructuring the logic. Details of this procedure are discussed in the sequel. It is instructive to note that during the course of the area-delay tradeoff, each of the manual adder designs mentioned above is achieved by the automatic synthesis procedure.

1.3.3 Other relationships

Little is known of the remaining relationships that exist between the three criteria. The purpose of this thesis is to explore the interactions between area, testability and delay more completely. In particular, the focus is on the relationship between the delay and testability in area minimal circuits. This relationship is shown to be critical in the design of reliable high-performance circuits. An interesting aspect of this exploration is that it leads to similar approaches to efficiently solving problems that involve timing analysis and test generation for all types of fault models.

1.4 Thesis contributions and overview

A general question that captures the focus of this thesis is:

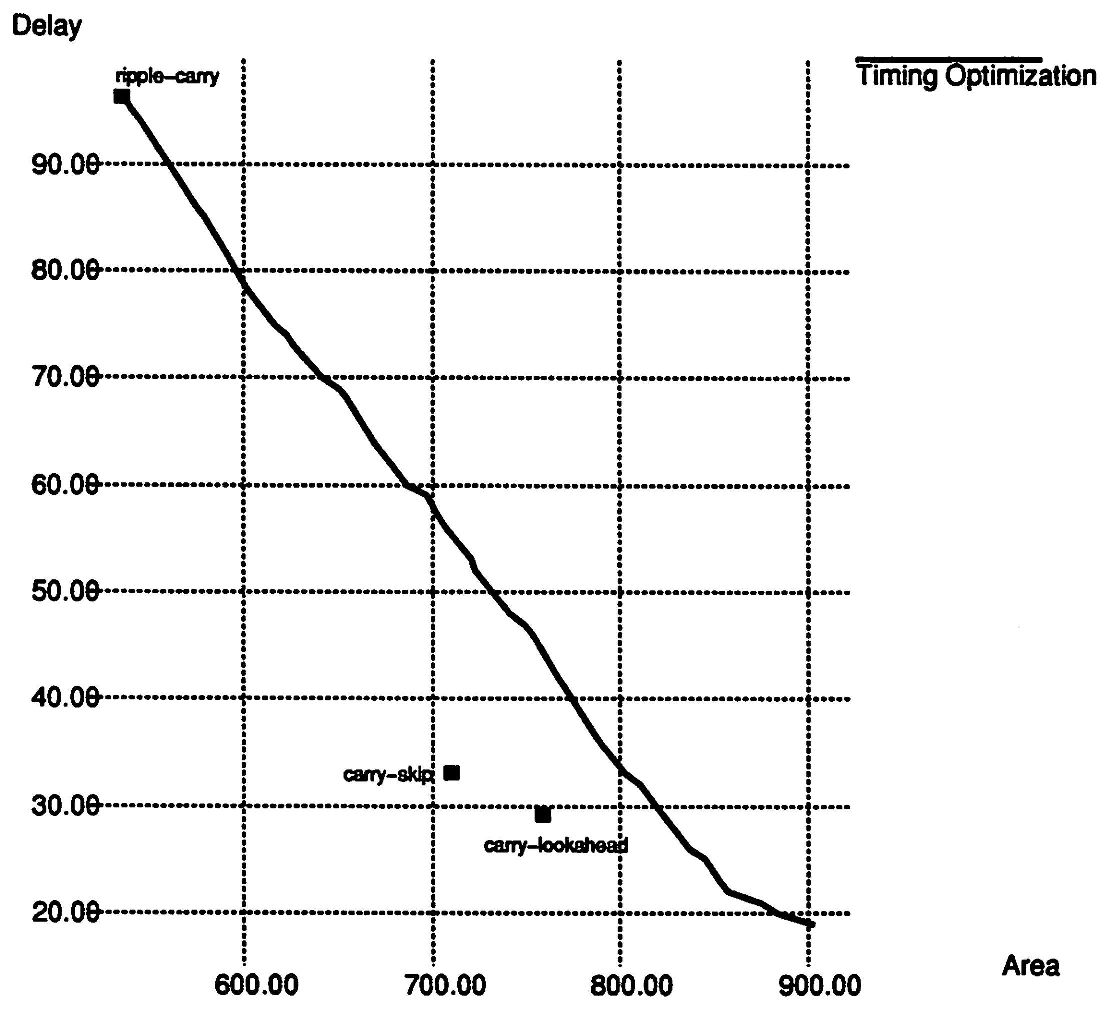


Figure 1.1: Area versus delay tradeoff on a 32-bit adder

- ***What is the relationship between the performance, testability, and area of an optimized logic circuit?***

While the principal focus will be on the interaction between the performance and testability of circuits, the area of the synthesized design is always an important consideration in the results stated in this thesis.

Chapter 2 explores the impact of redundancy in high-performance circuits. Most high-speed circuits have redundancy introduced when delay optimization is performed on a circuit. At first, it appears that this redundancy is necessary to reduce the delay of the circuit since a straightforward technique for removing redundancy slows down the circuit. The redundancy also has a profound effect on the reliability of the manufactured circuits. The illustration of a high-performance and redundant circuit that exhibits unreliable behavior motivates three problems that are addressed in the remaining chapters of this thesis.

Chapter 3 addresses the question of whether redundancy is necessary to reduce delay. An algorithm is provided that converts any high-speed redundant circuit into an irredundant one guaranteed to be no slower. A proof of the invariance of the delay estimate in the original and final circuits using the most accurate functional timing analysis technique is provided.

Even though redundancy is unnecessary to reduce delay, performance optimization transformations may reduce testability, and this is studied in Chapter 4. Different fault models are considered to examine their invariance under common delay optimization transformations. In particular, delay-fault testing of IC's , which has found increased interest in recent years, is considered.

In the area of performance optimization, delay estimation of a circuit has lacked efficient algorithms. In the area of testability, efficient algorithms for delay-fault testing have not existed until recently. Both timing analysis and delay-fault testing are the focus in Chapter 5, where a uniform and novel approach to solving general path analysis problems is presented [80].

Chapter 6 provides a synopsis of the theoretical and practical results obtained in this research.

The solutions to several problems in this thesis are obtained by formulating them as Boolean satisfiability problems. An overview of the heuristics used in solving a general Boolean satisfiability problem is provided in Appendix A.

Chapter 2

Redundancy and Delay

This chapter serves two purposes. The first part provides definitions of the two circuit properties that are of most concern in this thesis, namely *redundancy* and *delay*. Following a few basic definitions relating to combinational circuits in Section 2.1, the terms and techniques used in the testability and delay analysis of Boolean networks are discussed in Sections 2.2 and 2.3, respectively. The second part of this chapter explores the interaction between redundancy and delay. Besides the well known disadvantages of redundancy, briefly reviewed in Section 2.4, a significant reliability problem due to redundancy is discussed in Section 2.5 using the example of a well known adder circuit. This example motivates three problems, enumerated in Section 2.7, and addressed in the remainder of this thesis. Related work is reviewed in Section 2.6 to place the contributions of this thesis in context.

2.1 Boolean networks

During the process of logic synthesis, combinational logic is represented by an abstraction known as Boolean network. A few definitions related to Boolean networks required in subsequent discussions are provided below. Further details may be found in [16].

Definition 2.1.1 *A* **combinational circuit** *(or* **Boolean network***) is a directed acyclic graph composed of gates (or nodes) and connections (or edges) between gates.*

Definition 2.1.2 *A* **path** *in a combinational circuit is an alternating sequence of connections and gates, $\{c_0, f_0, c_1, ..., c_m, f_m, c_{m+1}\}$, where connection c_i, $1 \leq i \leq n$, connects the output of gate f_{i-1} to an input of gate f_i. The f_i's are referred to as gates along the path.*

Defining a path as a sequence of connections and gates, rather than simply as a sequence of gates, gives greater flexibility in modeling delay and allows the unambiguous description of circuits with more than one connection from one gate to another.

Definition 2.1.3 *A path that includes a primary input and a primary output is termed an* **IO-path**.

Definition 2.1.4 *The* **depth** *of a circuit is the maximum number of gates along any path in the circuit.*

Definition 2.1.5 *If the output of a gate f_1, is connected to an input of gate f_2, f_1 is a* **fanin** *of f_2. Gate f_2 is a* **fanout** *of gate f_1. If there is a path from f_i to f_j, then f_i is a* **transitive fanin** *of f_j, and f_j is a* **transitive fanout** *of f_i.*

Definition 2.1.6 *A* **literal** *refers to a Boolean variable that appears either in its complemented or uncomplemented form,* e.g. *x' or x.*

Definition 2.1.7 *A* **cube** *is a product of literals: e.g. $xy'z$. A* **minterm** *is a cube in which every variable appears.*

Minterms may be used to represent the values of a set of input variables: e.g. $xy'z$ is shorthand for $x = 1, y = 0$, and $z = 1$. In this way there is a natural correspondence between an input vector or input stimulus and a minterm. This correspondence may be extended to cubes where unspecified values in the function are assumed to be arbitrary or unknown values. Thus if a circuit C has inputs v, w, x, y, and z, then applying the cube $xy'z$ to C is shorthand for applying $u = X, v = X, x = 1, y = 0$, and $z = 1$. Here X denotes an unknown value.

In this thesis, the terms *logical behavior* and *temporal behavior* of a combinational circuit are used to make an important distinction. Logical behavior refers to the final value on each primary output of the circuit when a stable value is asserted on the primary inputs. Two circuits are *logically equivalent* under an input cube if they have the same stable values on the primary outputs. Temporal behavior refers to the dynamic behavior of the circuit, which considers the logical values on the primary outputs at a particular instant of time. Thus, two circuits may exhibit the same logical behavior under an input cube, yet the final values on the primary outputs of one circuit may be available at different times than the other circuit. In this case, the circuits exhibit different temporal behavior.

2.2 Redundancy in a network

Most manufactured IC's must be *tested.* The goal of testing is to determine whether a manufactured circuit behaves correctly. This is done by applying *test vectors* (at the primary inputs) that distinguish between the functional behavior of the correct and faulty circuits. Defects that may occur are abstracted using fault models which define their impact on circuit behavior. For example, a fault may change only the logical value on some gate in a circuit. Another modeled fault may impact only the timing behavior of a gate, but not its logical behavior. Although several fault models have been proposed [19], in this thesis only the two major classes of fault models are considered. The first, and most common, are *static faults* that model only the impact of defects on the logical behavior of gates or connections in circuits. This class is briefly reviewed later in this section. The second, introduced and discussed in detail in Chapter 4, are *delay-faults*, which represent changes only in the temporal behavior of gates or connections.

Having chosen a fault model, the term *testability* of a circuit refers to the ability to generate a test for each fault that may exist in the network. The term testability has many connotations in testing and synthesis [4, 27]. It often includes the time required for test pattern generation and the time required for actual application of the tests to each manufactured circuit. These two criteria are particularly significant in some methods of testing sequential circuits [47]. Since the focus here is on combinational circuits, our only concern with respect to testability is the existence of a test for each fault.

Definition 2.2.1 *A gate has a* **stuck-*1*** *(***stuck-*0*)** **fault** *on an input (output) if the logical value asserted on the input (output) is always* 1 *(0) independent of the value presented on the input (output).*

Definition 2.2.2 *A circuit has a* **single stuck-fault** *(or* **single-fault***) if there is only one stuck-1 or stuck-*0 *fault in the circuit. A circuit has a* **multiple stuck-fault** *(or* **multi-fault***) if there is one or more stuck-faults in the circuit.*

Testing for single or multiple stuck-faults attempts to ensure with high probability that manufactured circuits with defects leading to incorrect logical behavior are detected. Since the timing behavior of the circuit is of no concern, the testing of stuck-faults is referred to as *static testing.*

Definition 2.2.3 *A stuck-fault is* **testable** *(or* **irredundant***) if it causes a change in the logical behavior of the circuit for at least one input cube. Otherwise, the stuck-fault is* **untestable** *(or* **redundant***).*

In general, a circuit where each modeled fault is testable is said to be 100% (fully) testable under the chosen fault model. In particular, a circuit that is fully single stuck-fault testable is called an irredundant network. Otherwise it is called a redundant network. Like the term testability, the term *redundancy* has come to have different meanings. While most often associated with the logical behavior of circuits and with single stuck-faults, it is also widely (and incorrectly) associated with other fault models that may not even impact logical behavior, such as delay-faults[1]. Computational redundancies, such as duplication of functional units, are also used to speed up circuits, but these do not necessarily imply stuck-fault redundancies.

2.3 Delay of a network

The primary concern with respect to the temporal behavior is to determine at what speed we may clock the circuit and be certain that the logical values on the outputs at the end of the clock period are correct. Most often the clock speed is an imposed *system requirement*. There are a number of ways of determining whether a design operates correctly at a certain clock speed. The most accurate is to build the circuit and test it at the required speed in its target environment. Because of the time and expense involved, this option is rarely viable. Therefore we would like to determine as accurately as possible *before fabrication*, whether a circuit meets its timing requirement. Even more difficult is the common problem of designing a circuit such that the fabricated circuit achieves its timing requirement. This usually involves repeated analog simulations of the circuit using a simulator such as SPICE [3]. Unfortunately, simulation has two significant problems: accurate simulation is computationally expensive and its utility is limited by the vector set applied. The first problem could possibly be addressed by using less accurate but more computationally efficient algorithms [22]. Even then, simulation of all possible input stimuli is never an option. Thus, if there is one unsimulated input stimulus that could cause the

[1]It is interesting to note that by the year 1962, the term redundancy had already become hopelessly overloaded [43].

circuit to go slower, then the simulation results may lead to the manufacture of a circuit that will fail occasionally at the required speed[2].

An approach that avoids the problem of test vector dependency is to use static timing verifiers [57, 84]. In this approach the delay of a circuit is assumed to be the longest path in the circuit. One problem with this is that there may not be any input stimulus that activates the longest path. Such paths are called *false paths* [8]. Thus, static timing verifiers may be too pessimistic in estimating the delay. A potential solution is to eliminate the "statically unsensitizable" paths from consideration[3]. In this case the longest statically sensitizable path is taken as the delay of the circuit. However, it has been shown that paths which are not statically sensitizable may still contribute to the delay of the circuit [14, 76]. Thus, this approach may result in a too optimistic estimate of the delay. In this thesis, the functional timing method for logic circuits proposed in [76], is employed. While this method cannot be considered to be as accurate as simulating a complete vector set with a tool such as SPICE, it makes the most accurate assumptions of all known timing verifiers [74].

Another property of a timing method is *robustness* [76]. Any network under timing analysis actually represents a *network family*, each member of which is structurally and functionally identical, but with varying timing properties. Typically, a timing analyzer conservatively chooses the member with the maximum gate delays. The condition of robustness requires that the timing estimate made by a timing analyzer be correct for all members of a network family [76]. The method employed in this thesis meets this criterion.

Definition 2.3.1 *Each gate f has a delay $d(f)$ and each connection c has a delay $d(c)$ associated with it.*

Although a straightforward timing model is used in this thesis, the results described do not depend on this particular model. It can be shown that they hold for models with separate rise and fall delays, different delays on each pin, and slope delay models [76].

Definition 2.3.2 *The* **length** *of a path $P = \{c_0, f_0, c_1, ..., c_m, f_m, c_{m+1}\}$ is defined as $d(P) = \sum_{i=0}^{m} d(f_i) + \sum_{i=0}^{m+1} d(c_i)$. $|P|$ denotes the length of path P. $d(c_i)$ is also represented as $d(f_i, f_{i-1})$.*

[2]This same problem can occur even if the circuit is built and run in its functional environment, since highly improbable input sequences may not have been tested.

[3]Briefly stated, the static sensitization condition for a path P requires the existence of a vector under which an event propagates along P independent of delays in the circuit.

Definition 2.3.3 *An* **event** *is the transition from 0 (1) to 1 (0). Let* $\{e_0, e_1, ..., e_m\}$ *be a sequence of events occurring at gates* $\{f_0, f_1, ..., f_m\}$ *along a path* P*, such that* e_i *occurs as a result of event* e_{i-1}*. The time of event* e_i *is denoted* τ_i^P*. The event* e_0 *is said to propagate along the path. If an event can propagate along a path, then the path is said to be* **sensitizable.**

The discussion of the conditions under which a path is sensitizable is deferred until Section 2.3.1.

Definition 2.3.4 *A* **critical path** *is a longest sensitizable path in the circuit.*

The delay of a circuit is the length of a critical path.

Definition 2.3.5 *A* **controlling** *(or* **annihilator***)* **value** *for a gate* f *is the value at its input that determines the value at the output independent of the other inputs, and is denoted as* $A(f)$*. For example,* 0 *is a controlling value for an* AND *gate. A* **non-controlling** *(or* **identity***)* **value** *for a gate* f *is the value at its input which in not a controlling value for the gate, and is denoted as* $I(f)$*. For example,* 1 *is a non-controlling value for an* AND *gate.*

Definition 2.3.6 *A* **simple gate** *is any one of* AND, OR, NAND, NOR, *and* NOT.

A controlling value is only defined for simple gates. For a complex gate such as $f = ab + cd$, neither $a = 0$ nor $a = 1$ by themselves determine the value of f. For a simple gate f, $A(f) = \overline{I(f)} \in \{0, 1\}$.

Definition 2.3.7 *Let* $P = \{f_0, f_1, ..., f_m\}$ *be a path. The inputs of* f_i *other than* f_{i-1} *are called* **side-inputs** *of* f_i *along* P *and denoted as* $S(f_i, P)$*. A path that starts at a primary input and ends at a side-input of* P *is a* **side-path** *of* P*.*

Notice that the connections along the path are not explicitly enumerated in this definition. This is done for ease of exposition when a single connection exists between a pair a gates. Henceforth, except in circuits with multiple connections between pairs of gates, all paths avoid explicit reference to the connections.

Definition 2.3.8 *A path is* **statically sensitizable to** *0 (1) if there exists an input cube which sets all the side-inputs to the path to non-controlling values and causes a value of* 0 *(1) on the output of the path.*

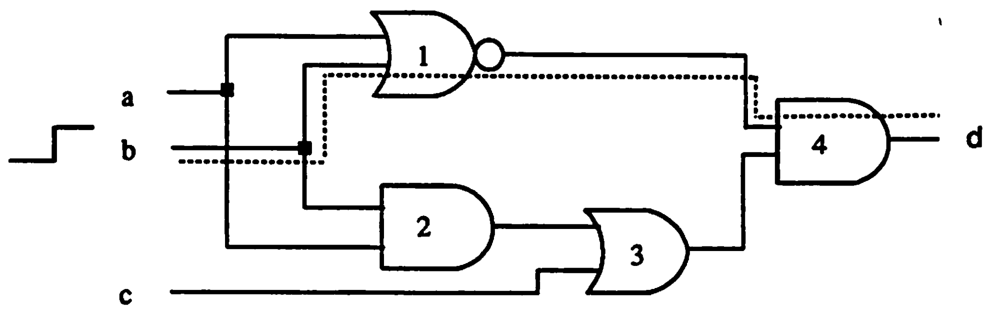


Figure 2.1: Example of definitions

Definition 2.3.9 *A path is* **statically sensitizable** *if there exists an input cube which sets all the side-inputs to the path to non-controlling values. The condition for* **static sensitization** *of a path $P = \{f_0, f_1, ..., f_m\}$ composed of simple gates is $\prod_{i=0}^{m} \prod_{g \in S(f_i,P)} (g = I(f_i))$.*

Note that a path is statically sensitizable if it is statically sensitizable to *0 or* statically sensitizable to *1* .

An illustration of these terms is provided in Figure 2.1. The path shown as a dotted line from input b to output d is statically sensitized to *0* by the cube $a'b$, and is statically sensitized to *1* by the cube $a'b'c$. The path from b to d through gates 2, 3 and 4 is not statically sensitizable since no input cube exists which causes each of the side-inputs to have non-controlling values.

2.3.1 Delay computation using viability analysis

In order to refer to the "delay" of a circuit, one must first determine how it is computed.

Given a circuit in a particular internal state, *i.e.* with some values on the wires in the circuit (*0*, *1*, or anything in between), a change in some of the inputs (an input event) can possibly result in a change in some of the outputs (an output event). The delay of this input event is the time between it and the time all the outputs of the circuit have settled to their final values. The delay of the circuit is the maximum delay over all possible input events, starting with all possible internal states.

Unfortunately, in order to measure delay as defined above, all possible input transitions must be considered, under some assumptions about the electrical behavior of the circuit components. This is a formidable but important problem for critical circuits whose delay determines the speed at which the circuit can be clocked. For correct functionality it is permissible to approximate the delay by an upper bound on the true delay. Pessimistic assumptions may be made to simplify the delay analysis since they ensure that the computed quantity is an upper bound. Any upper bound on the true delay obtained by some approximation technique is referred to as the *computed delay* to distinguish it from the true delay.

Since the computed delay can determine the clock cycle, it is important that this bound be as tight as possible. Research in computed delay methods (e.g. [14, 41, 76, 87]) has led to increasingly tighter bounds. To date, all accepted methods use only a single vector condition. In such an analysis, the values on the wires before the application of this vector are considered unknown.

In [76], delay is measured using the notion of *viability*. It is proved there that the upper bound obtained is valid, robust, and tighter than that obtained in [14]. Viability analysis provides the tightest upper bound on the delay among all approaches known so far[4]. Viability analysis is used in all the proofs and delay analyses throughout this thesis.

The approach presented in [76] is briefly overviewed. In Chapter 5, an efficient technique to determine the viability of paths is discussed.

The computed delay of network η for the primary input cube c is denoted as $delay(\eta, c)$. $delay(\eta, -)$ is the maximum value of $delay(\eta, c)$ over all input cubes. Note that if $delay(\eta, c) = l$, then there should be some path of length l from a primary input to a primary output which is sensitizable, i.e. an event propagates down this path and reaches the output after time l[5]. Viability attempts to answer the question: when is a path sensitizable? Consider a network comprised entirely of simple gates. For a path P to be sensitizable, each side-input to each gate f_i along P must have a non-controlling value on

[4]The method of Chen and Du [24] gives the same computed delay as viability analysis, but may report fewer true paths. This may be important in applications where delay analysis is used in conjunction with speed-up techniques which restructure only true critical paths. However, the method of [24] is restricted to simple gates; viability analysis applies to complex gates and is more general.

[5]Since the computed delay is only an upper bound (but a fairly tight one), it may be that no event can propagate with delay l.

it *at the time* the event reaches f_i. To see this, suppose some side-input has a controlling value on it when the event reaches f_i. The controlling value would have already set the output value of the gate thus blocking further propagation of this event.

Let τ_i^P be the time of event e_i on the connection between f_{i-1} and f_i. For a given input cube c, the side-inputs at f_i may be classified into two sets.

1. The set $\mathcal{E}$ of *early arriving* side-inputs. These inputs settle to their final value before τ_i^P.
2. The set $\mathcal{L}$ of *late arriving* side-inputs. These inputs have not settled to their final value before τ_i^P.

It is clear that at τ_i^P, the values for the side-inputs in $\mathcal{E}$ are determined exactly by the input cube, c, and not the previous internal state of the circuit. However, the values for the side-inputs in $\mathcal{L}$ are determined not only by c, but also the previous input cube that was applied, as well as the electrical characteristics of the connections. A detailed analysis for all possible cases of previous and current input cubes and electrical characteristics is, in general, considered to be too difficult. To get around this, all timing analyzers make pessimistic assumptions, assuming the worst possible internal state and electrical characteristics.

In [76], the notion of *viability* is introduced for this purpose. It is defined recursively. A path is viable under input cube c if at each gate f_i along the path, all the side-inputs in $\mathcal{E}$, as determined by c, have non-controlling values. The side-inputs in $\mathcal{L}$ are assumed (pessimistically) to be at non-controlling values, if a viable path under c of length $\geq \tau_i^P$ exists up to the side-input; otherwise, the side-input must evaluate (statically) under c to a non-controlling value. A path is viable if there exists some input cube under which it is viable. The formal definition for the viability function given in [76, 75] is stated in abstract functional terms, and requires some machinery of an operator calculus. The simple version of viability analysis, valid only for simple gates is provided here, with the remark that this formulation can be easily extended to general gates.

Definition 2.3.10 *The set of paths of length $\geq t$ which terminate at gate f is denoted $\mathcal{P}_{f,t}$.*

Thus for a gate f_i along a path P, the set of paths terminating at a side-input g_j of f_i, of length $\geq \tau_{i-1}^P$, and hence arriving no earlier than event e_{i-1}, is denoted $\mathcal{P}_{g_j,\tau_{i-1}^P}$.

Recall that $I(f)$ denotes the non-controlling value of gate f.

Definition 2.3.11 *A path $P = \{f_0, f_1, ..., f_m\}$ is* **viable** *under an input cube c if, at each node f_i for each $g_j \in S(f_i, P)$, either:*

1. $g_j(c) = I(f_i)$ *; or*
2. $\exists Q_j \in \mathcal{P}_{g_j, \tau_{i-1}^P}$ *such that Q_j is viable under c.*

The first condition in Definition 2.3.11 applies to all the side-inputs in the set $\mathcal{E}$ and some of $\mathcal{L}$; the second condition applies to the remaining side-inputs in the set $\mathcal{L}$. Viability is a weaker condition than static sensitization since it makes no demands on the static values of some side-inputs in $\mathcal{L}$. Observe that if a path is statically sensitizable then it is viable since all side-inputs have non-controlling values. The computed delay of the network is thus the length of the longest viable path. This is more pessimistic than considering the longest sensitizable path; viability analysis makes the assumption that the late viable side-inputs have non-controlling values on them at τ_i^P. [76] uses the *smoothing operator* to effectively set non-controlling values on late-arriving viable signals.

Definition 2.3.12 *The* **cofactor** *of a Boolean function f with respect to a literal x is denoted f_x and is equal to the value of f assuming the literal x is set to value* 1 *(and x' is thus set to* 0 *).*

For example, given $f = xy + x'z + w$, $f_x = y + w$, and $f_{x'} = z + w$.

Definition 2.3.13 *The* **smoothing operator** *applied to a function f with respect to a variable x is $S_x f = f_x + f_{x'}$. $S_x f$ yields the smallest Boolean function containing f which is independent of x. If X is a set of variables $\{x_0, x_1, ..., x_m\}$, $S_X f = S_{x_0} S_{x_1} ... S_{x_m} f$. This notation is meaningful since the operator is commutative,* i.e. $S_{x_1} S_{x_2} = S_{x_2} S_{x_1}$.

For a simple gate f with input x, $S_x f$ is equivalent to asserting a non-controlling value on x.

The definition of viability is for the application of a selected cube c. In general, the concern in delay estimation is to determine the existence or absence of such a sensitizing cube. Thus, a Boolean function of primary input variables is used to capture all the conditions (i.e. input vectors) under which a path is viable. This function is the viability function, and is derived in a fairly straightforward fashion from the above definition in three equations.

Definition 2.3.14 *The* **viability function** ψ_P *of a path* $P = \{f_0, f_1, ..., f_m\}$ *is defined as follows*

$$\psi_P = \prod_{i=0}^{m} \psi_P^{f_i}$$

where

$$\psi_P^{f_i} = \sum_{U \subseteq S(f_i,P)} \{\prod_{g \in U} \psi^{g,\tau_{i-1}^P} \prod_{g \in S(f_i,P)-U} (g = I(f_i))\}$$

and

$$\psi^{g,t} = \sum_{Q \in \mathcal{P}_{g,t}} \psi_Q.$$

Briefly explained, the condition under which a path P is viable, denoted ψ_P, is the condition $\psi_P^{f_i}$, under which each gate f_i satisfies the viability conditions on its side-inputs. $\psi_P^{f_i}$ is true if there is some set U of viable late arriving (with respect to event e_{i-1}) side-inputs to f_i (the term $\prod_{g \in U} \psi^{g,\tau_{i-1}^P}$), and all the other side-inputs are at non-controlling values (the term $\prod_{g \in S(f_i,P)-U}(g = I(f_i))$). ψ^{g,τ_{i-1}^P} is equivalent to the existence of at least one viable path to g of length greater than or equal to τ_{i-1}^P. A detailed explanation and proof of correctness is in [74].

The formulation of viability given above appears computationally formidable. Besides the recursive nature of the formula for the viability function at a gate, a power-set computation (with size exponential in the number of fanin at the gate) is required. However, Chapter 5 describes a new and efficient approach to solving this difficult problem. Further discussion of the algorithms for timing analysis are deferred until then.

2.4 Disadvantages of redundancy

There are two well-known reasons (discussed briefly here) why redundancy is undesirable during the synthesis of logic circuits. For high speed circuits, a significant but little known problem is created by redundancy. This is illustrated in the next section.

The best known impact of redundancy in circuits is that it is difficult to detect. Redundancy often hampers the efficiency of the test pattern generation process; some test pattern generators either choke on redundant faults or abort too early on some testable faults. A simplistic explanation is that the entire Boolean space corresponding to all possible primary input values must be exhaustively enumerated, albeit implicitly, in proving a fault redundant. However, with the advent of powerful logical implication mechanisms and algorithms [100, 64], redundancy identification is no longer considered particularly difficult for combinational circuits. On the other hand, proving redundancy via deterministic test generation in sequential circuits is still difficult [47].

Another well known disadvantage of redundancy is the impact on the area of the circuit [16]. For every circuit with stuck-fault redundancy there exists an irredundant smaller circuit. Ignoring the impact on delay, this redundancy can be removed in a straightforward fashion by employing a deterministic test pattern generator, see [100] for example. Setting a redundant connection to *0* or *1* only causes the disappearance of one or more connections or gates. Experiments using test pattern generation techniques in optimization of Boolean networks are summarized in [16].

Still another reason is that some industries (*e.g.* IBM) use an internal cost method which charges a project based, in part, on a testability measure. Thus, redundancies add to the cost.

The reasons listed above for redundancy being undesirable in optimized circuits appear outdated in state-of-the-art logic design. For example, current redundancy removal algorithms can efficiently detect and remove redundant faults on most combinational circuits. One such approach is reported in Appendix A. However, two important issues remain:

- Redundancy may have been introduced to reduce the delay of the circuit; then a straightforward redundancy removal may slow down the circuit. Even worse, if the redundancy is left in the circuit (because of this), a manufacturing defect on the redundant connection would be statically untestable. Thus, the circuit contains a (statically) undetectable fault which could cause the circuit to fail by slowing it down.

- No irredundant circuit may exist with exactly the same delay and same area as the redundant circuit. The issue here is whether redundancy is really necessary to speed up the circuit.

Since performance is often the most important criterion of logic optimization, these problems must be addressed before dismissing redundancy in circuits as a solved problem. This point is emphasized with an example of a well-known high-speed circuit with a severe reliability problem. It strengthens the case for requiring irredundancy in optimized combinational logic circuits.

2.5 Redundancy in high-speed circuits

2.5.1 The carry-skip adder

Consider a 2-bit block of a carry-skip adder, shown in Figure 2.2. In terms of area and performance, the carry-skip adder is between that of a ripple-carry adder and a carry-lookahead adder. Apparently the carry-skip adder design was first proposed by Babbage in the nineteenth century [66]. See [66, 50, 83] for other studies on this type of adder.

The carry-skip adder uses a conventional ripple-carry adder (the output of gate 11 is the ripple-carry output) with an extra AND gate (gate 10), and a MUX added to each block. If all the propagate bits through a block are high (the outputs of gates 1 and 3) then the carry-out of the block ($c2$), is equal to the carry-in to the block ($c0$). Otherwise, it is equal to the output of the ripple-carry adder. The multiplexer thus allows the carry to skip the ripple-carry chain when all the propagate bits are high. A carry-skip adder of n bits can be constructed by cascading a set of individual carry-skip adder blocks, such as Figure 2.2, such that the sum of the block sizes is n. In general, the delay and area of a carry-skip adder depends on the size of the blocks and the number of bits in the adder. See [83] for a discussion of sizing blocks in a carry-skip adder to minimize delay.

2.5.2 Redundancy problems

In almost all cases the straightforward removal of redundancy does not affect the speed of a circuit. However, in the case of the carry-skip adder, in which an extra carry-chain is added to improve the speed, removing the attendant redundancy in the design (the select input of the MUX is redundant when stuck at *0*) slows the circuit down.

The extra AND gate and MUX of the carry-skip adder have a profound effect on its performance and testability. First consider the impact on the performance and refer to Figure 2.2. Assume the primary input $c0$ arrives at time $t = 5$ and all the other primary inputs arrive at time $t = 0$. Assign a gate delay of 1 for the AND and OR gates and gate delays of 2 for the XOR and MUX. By *accurate* timing analysis (such as SPICE [3]) it can be shown that the worst-case delay of the circuit is along the path from $a0$ to $c2$ through gates 1, 6, 7, 9, 11 and the MUX in Figure 2.2. This is the critical path and its output is available after 8 gate delays [6].

[6]We are concerned with the critical path through the carry-out of the circuit, even though there is a path whose output is available after 9 gate delays for the final sum bit in the block. This is because in an adder

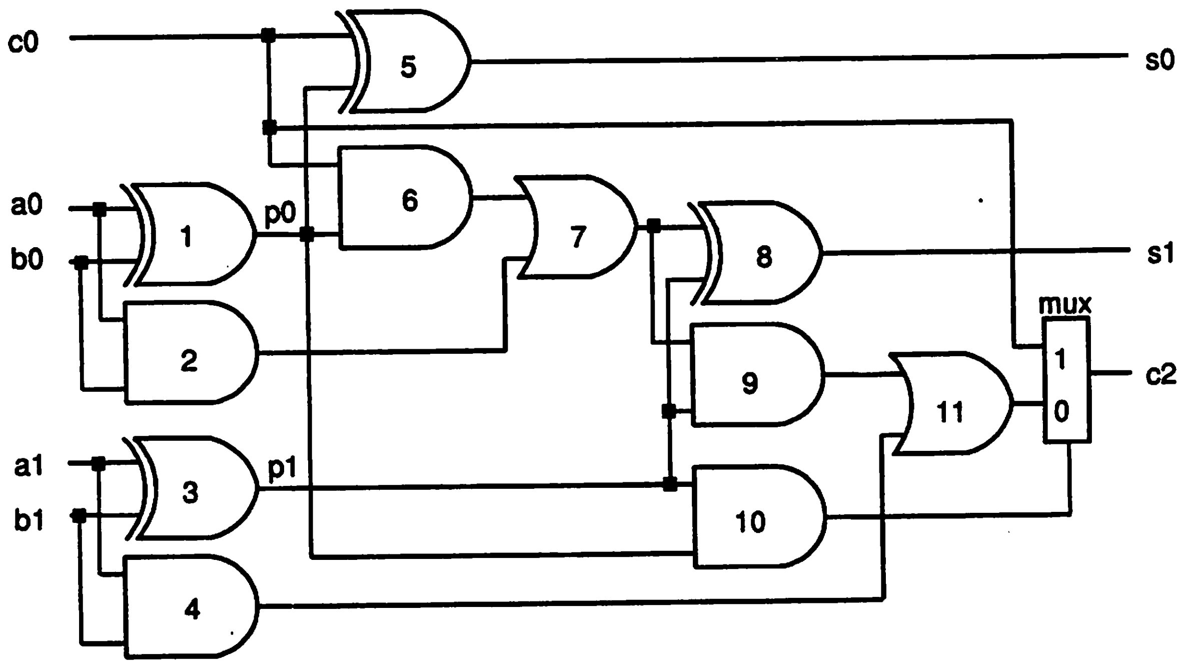


Figure 2.2: 2-bit carry-skip adder

The (statically or topologically) longest path in the circuit is the path from $c0$ to $c2$ through gates 6, 7, 9, 11 and the MUX (available after 11 gate delays). However, it is a false path in the carry-skip adder. In contrast, in a ripple-carry adder the topologically longest path determines the delay of the circuit. Thus by adding the additional circuitry, the delay of the circuit has been reduced. As regards testability, while a ripple-carry adder is fully testable, the carry-skip adder has a single redundancy in the circuit. In Figure 2.2, the single stuck-*0* fault on the output of gate 10 is not testable. This is due to the fact that the carry-skip adder becomes a logically-equivalent ripple-carry adder in the presence of the fault. Thus, in attempting to gain speed, the testability of the circuit has been compromised.

There is a further problem with the carry-skip adder. Consider the case where the output of gate 10 is stuck at *0*, effectively reducing the circuit to a ripple-carry adder. The critical path is now the longest path in the circuit and its output is available after 11 gate delays. If the clock had been set based on the length of the original critical path (in the absence of faults), then the circuit will behave incorrectly when the single stuck fault exists. This is a serious problem since the stuck-*0* fault on the output of gate 10 is not testable using standard static testing techniques.

The reason the stuck-*0* fault causes the circuit to slow down is that a non-sensitizable (*false*) path becomes sensitizable in the presence of the fault. In Figure 2.2 the longest path is a false path; however, in the presence of the stuck-*0* fault on the output of gate 10, it becomes a true path. Thus, the output of the circuit is now correctly available only after 11 gate delays rather than 8 gate delays.

2.6 Related work

This section provides a brief review of related work that addresses the impact of stuck-faults on timing behavior. In [77], the observation is made that the stuck-*0* fault on the AND gate, 10, in the example of Figure 2.2, though classified as redundant in the testing sense, changes the temporal behavior of the circuit. Hence, in a functional sense it is an irredundant fault. In [77], a stuck-fault that does not cause the outputs of the good and faulty circuit to be different at all times $t \geq \tau$ is defined as a τ-redundant fault. Otherwise a

composed of blocks similar to Figure 2.2, the critical path for the entire adder will be the path through the carry-out of each block.

fault is termed τ-irredundant, *i.e.* with respect to both timing and logical behavior, a fault is τ-irredundant if it changes the timing behavior in the good and faulty circuit, even though it may be logically redundant. The stuck-fault in question may be a single or multiple stuck-fault. The following definition is taken almost verbatim from [77].

Notation: Each multiple stuck-fault (or multi-fault) F on a network family $\mathcal{N}$ denotes a family of faulty networks, denoted $\mathcal{N}_F$. To each network $\eta \in \mathcal{N}$ there exists a corresponding faulty network $\eta_F \in \mathcal{N}_F$. If a node in η is denoted f, its counterpart in η_F is denoted f_F.

Definition 2.6.1 *A single or multiple stuck-fault F on a network family $\mathcal{N}$, inducing faulty family $\mathcal{N}_F$, is said to be* τ-**redundant** *if, for each output f of η, $f_F \oplus f \equiv 0$ at all $t \geq \tau$ in every network $\eta \in \mathcal{N}$. If a fault is not τ-redundant, then it is τ-irredundant. If a circuit is τ-irredundant, then there exists a vector v, and a time $t \geq \tau$ at which $f_F \oplus f \equiv 1$ for at least one output f of some network $\eta \in \mathcal{N}$ and we say that the fault is tested by v.*

This new concept implies that to ensure correct functionality of the circuit, the design must undergo a *speed-test* in addition to the conventional stuck-fault testing and delay-fault testing. The speed-test for a τ-irredundant fault in the circuit involves finding a vector that distinguishes between the temporal behavior in the true and faulty circuits.

Definition 2.6.2 *A vector v is a valid* **speed-test** *for a multiple stuck-fault m in a circuit η if v detects the τ-irredundant fault corresponding to m.*

Two principal observations can be made regarding the proposal in [77]. First, test generation for τ-irredundant faults requires all redundant multiple faults to be considered. This enumeration is a formidable problem. In fact, the sheer number of faults to be considered is the primary reason multiple-fault test generation is a hard problem [19]. Second, the derivation of a vector that tests each τ-irredundant fault is an open problem, so a new test generation methodology must be invented. Note that the test generation process may be restricted to only redundant faults which cannot be detected by conventional static testing. This is because a τ-irredundant fault which is logically irredundant is detected by static testing. Even so, the number of faults to be considered in this restricted case may be very high, since multiple faults are being considered.

An additional complication is the actual testing process. Unlike static testing, where the circuit under test may be clocked at any convenient rate, speed-testing requires the circuit be clocked at speed. [77] enumerates a multitude of possible problems in employing such a technique, among them hazards and the accuracy of the tests themselves. Thus far, no feasible technique is known that enables testing of τ-irredundant faults.

2.7 Redundancy and delay questions

An immediate conclusion from the discussion in the previous section, but not made in [77], is:

Theorem 2.7.1 *Speed-testing for τ-irredundant faults is not required in 100% multiple stuck-fault testable circuits.*

Proof By definition, a τ-irredundant fault is a multiple stuck-fault that causes a circuit to slow down. However, since every multiple fault is testable, conventional static testing can be used to detect the presence of the multiple fault (corresponding to a τ-irredundant fault) in each manufactured circuit. Hence, no speed-test is required. ■

Thus speed-testing can be avoided in fully multiple stuck-fault testable circuits. Even if the transformation to a fully multiple-fault testable circuit is possible via exhaustive fault enumeration and redundancy removal, there is a caveat. In order to meet performance specifications, the delay of the 100% multiple-fault testable circuit must not be any greater than the delay of the original circuit. This condition is difficult to realize by simple redundancy removal; for the example shown in Figure 2.2, a straightforward removal of the stuck-*0* redundant connection slows down the circuit.

In an irredundant circuit, the existence of each stuck-fault can be tested using conventional static testing alone. Even though a stuck-fault may cause the timing behavior of the circuit to change, it necessarily causes the logical behavior to change too. Thus, the existence of the defect can be detected by resorting to static testing alone, thus obviating the need for a speed-test.

Motivated by this observation an immediate question arises:

- *Is redundancy necessary to reduce delay?*

In other words, for every redundant circuit, does there exist an irredundant circuit that has exactly the *same delay*. Note that the delay measure (computed delay) must be at least as accurate as those used in other synthesis operations. For this reason, the computed delay technique used in resolving this question is viability analysis, described in Section 2.3.1.

To definitively answer this question there are two (obvious) cases:

1. Redundancy is necessary to reduce delay. In this case an example circuit is required where no irredundant version can be guaranteed to have the same or less computed delay.

2. Redundancy is not necessary to reduce the delay. One approach then would be to provide a transformation of the given circuit that yields an irredundant circuit with no greater computed delay.

The resolution of this question is the subject of Chapter 3.

Another approach is motivated by the fact that we are primarily interested in performance optimized circuits in this context. Since such circuits may exhibit unreliable behavior due to the presence of redundancy, rather than attempting to rectify the reliability problem after performance optimization, another option would be to maintain reliability (testability) during performance optimization. This question is posed as:

- *What, if any, testability properties are maintained invariant during performance optimization transformations?*

This answer to this question has several parts and is considered in detail in Chapter 4. The focus is on the resynthesis schemes described in [105], [78] and [11]. Most timing optimizations can be considered variations of these. The two classes of fault models mentioned in Section 2.2, namely stuck-faults and delay-faults, are considered, and the impact of performance optimization operations on the circuit testability, using each model, is considered.

Chapters 3 and 4 address the interaction between performance and testability, when optimality is of concern. A significant practical component of the discussions assume the existence of algorithms for the analysis of the timing and testability behavior of circuits. However, efficient algorithms for both timing analysis and delay-fault test generation are lacking. Both problems belong to the class of problems that require functional path analysis. Their formulations are very similar. Thus another open question is:

- *Do there exist efficient and similar techniques to perform timing analysis and delay-fault test generation?*

In particular, can the techniques used in accelerating one problem be used in improving the algorithms of the other. Chapter 5 further illustrates the interaction between performance and testability by providing a uniform framework for solving both problems efficiently. The techniques described are instrumental in obtaining more efficient algorithms and analysis tools for the problems that are solved in Chapters 3 and 4.

An important component of the efficiency of the computations discussed in Chapter 5 is the posing of the problems as ones of Boolean satisfiability [46]:

Satisfiability

Instance: A set U of variables and a collection C of clauses over U.
Each clause is a sum term of some variables from U.

Question: Is there a satisfying truth assignment for C?

As a general problem solving technique, satisfiability is related to important CAD problems such as stuck-fault testing, path sensitization analysis, and circuit equivalence checking. However, in place of adapting search strategies to each specific problem, which is the usual method, one can translate the problem into a general satisfiability question and use a generic algorithm. This allows general powerful search techniques and heuristics to be easily applied to any of these problems. Two techniques have emerged recently which have proven to be successful in solving the general satisfiability problems that arise in a class of problems [12, 64]. Appendix A discusses the most recent approach. Although this is based on the ideas proposed in [64], its superior overall quality and results, compared to those reported in the literature [65, 23] warrant a discussion of the general heuristics employed.

Chapter 3

Is Redundancy Necessary to Reduce Delay?

Experience has shown that performance optimizations can, and do in practice, introduce stuck-fault redundancies into designs. In Section 2.5, the negative impact of redundancy on reliability in a high-performance circuit is demonstrated. Are these redundancies necessary to increase performance or are they only an unnecessary byproduct of the particular performance optimizations? In this chapter a constructive resolution of this question is given in the form of an algorithm that takes as input a combinational circuit and returns an irredundant circuit that is as fast. The utility of this algorithm on the carry-skip adder is illustrated by presenting a novel irredundant design of that adder. As the algorithm may either increase or decrease circuit area, the remaining question if every circuit has an irredundant circuit that is at least as fast and is of equal or lesser area is left unresolved. However, some bounds on the area increase will be provided in terms of the initial circuit function and structure. Even for those users for whom area is of equal (or greater) importance than testability, it is worth applying the algorithm because, while it may increase area, it may decrease it as well.

In Section 3.1 an irredundant implementation of the carry-skip adder that is faster than the original redundant adder is illustrated. The algorithm is presented in detail in Section 3.2. A proof that the algorithm is guaranteed not to increase delay under the timing model is developed in Section 3.3. Section 3.4 is a description of the issues addressed in efficiently implementing the algorithm. Since the algorithm requires efficient (and accurate)

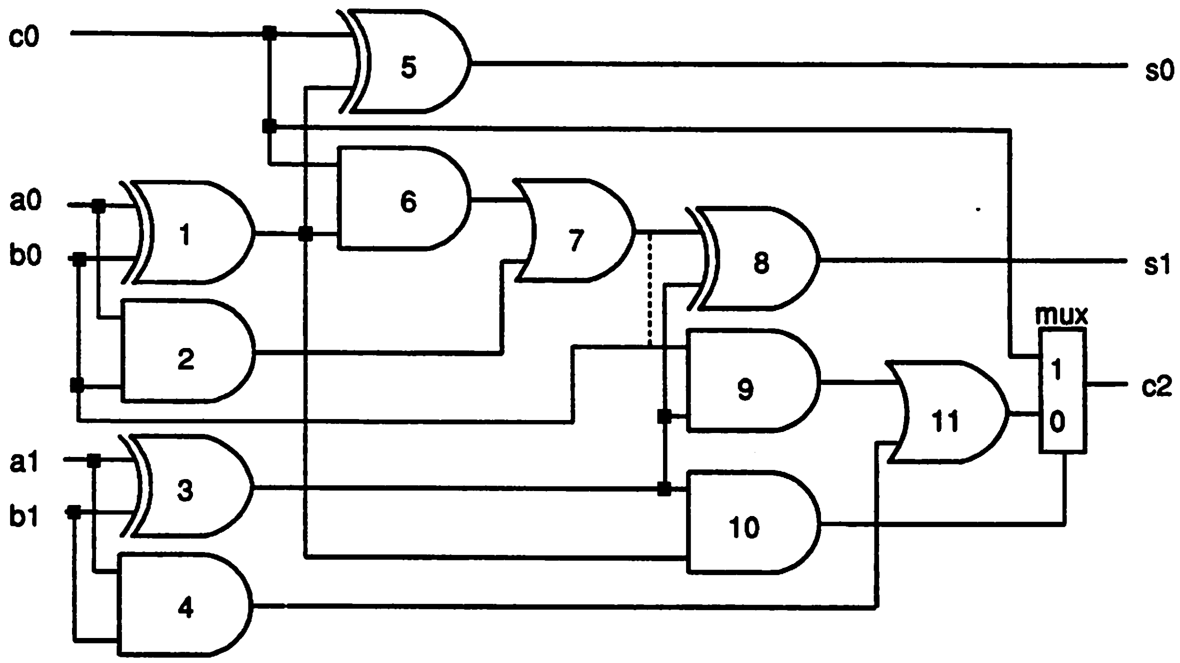


Figure 3.1: Irredundant 2-bit carry-skip adder

timing analysis, the details of this portion of the implementation are presented in a later section (*c.f.* Section 5.4). Results of applying the algorithm to benchmark examples are discussed in Sections 3.5. Due to the enormous time required on some examples, a single-pass approach is developed and proved in Section 3.6. Section 3.7 is a summary of the other applications of the algorithm. Section 3.8 concludes the chapter.

3.1 Irredundant carry-skip adder

Consider the circuit shown in Figure 3.1, derived from the original carry-skip circuit of Figure 2.2 by replacing the connection from the output of gate 7 to the input of gate 9 (shown with a dotted line) with the primary input connection *b0*. The two circuits have the same functional behavior and, as will be shown later, the new circuit is no slower than the original. However, the new circuit is fully testable for all single stuck-faults and consequently does not require a speed-test to ensure correct temporal behavior. Note that in this transformation there has been no area overhead incurred in obtaining the fully testable version of the carry-skip circuit.

The results that will be developed in the remainder of this chapter rest on the following sufficient condition that obviates the need for speed-testing.

Theorem 3.1.1 *Speed-testing is not required in a circuit with a longest path that is sensitizable.*

Proof Recall that speed-testing is only required in the presence of a τ-irredundant fault (*c.f.* Section 2.6). The occurrence of any fault cannot increase the delay of the circuit since the worst case delay is already equal to the length of the longest path. Hence there are no τ-irredundant faults that increase the delay of the circuit. ■

In the sequel it is shown that for every high performance circuit with redundancies, there always exists an equivalent irredundant circuit that is at least as fast as the original. An algorithm that realizes such an irredundant circuit is presented and proven correct. Each final single-fault irredundant circuit thus derived always has a longest path which is sensitizable. Consequently, by Theorem 3.1.1, no speed-test is required for the irredundant circuit. In a later section, this result is extended to multiple stuck-fault testability as well.

3.2 Irredundant circuits without performance penalty

In this section an algorithm is described that derives an equivalent irredundant circuit that is at least as fast as a given redundant circuit.

3.2.1 Algorithm for redundancy removal with no delay increase

Consider a circuit that has some redundancy. What can be said about the change in delay of the circuit when a constant value (*0* or *1*) is asserted on a redundant connection? While an answer to this question cannot be provided for an arbitrary circuit, there is a particular circuit structure for which the effect of the change in delay by a redundancy removal can be predicted. Three such cases are considered now. In the discussion to follow, the *first edge* of a path refers to the connection between a primary input and the first gate along the path.

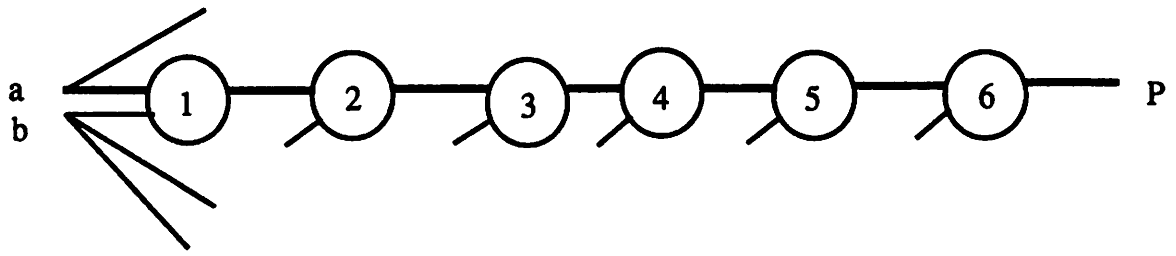


Figure 3.2: Example: Fanout-free unsensitizable longest path

Longest path statically sensitizable - Simple case: If a given circuit has a longest path that is statically sensitizable (hence sensitizable), then redundancy can be removed without any increase in delay. This is obvious since setting any connection to a constant value (*0* or *1*) cannot increase the length of any path in the circuit. Thus, the delay before and after the redundancy removal is the length of the longest path. This is the case considered in Theorem 3.1.1. The next two cases explain how this property is realized on a functionally equivalent implementation of an arbitrary circuit.

Longest path not statically sensitizable - Fanout-free case: Assume that the longest path P of a circuit is not statically sensitizable. Additionally, assume that every gate along P has a fanout of exactly one. This implies that a stuck-*0* fault and a stuck-*1* fault on the first edge of P are both redundant. Thus, if the first edge of P is set to a constant value, the logical behavior of the circuit remains unchanged. More importantly, by Theorem 3.3.2, the delay of the resulting circuit also does not increase. This case is illustrated in Figure 3.2.

Longest path not statically sensitizable - General case: Now consider the case where the longest path P is not statically sensitizable and some gates along P have fanout greater than one. As before, the fault effects of either of the two faults on the first edge do not propagate all the way along P. However, these faults may still be detected through some other path, and thus may be irredundant. With the given circuit structure, a constant value cannot be asserted on the first edge of P since this changes the logical functionality of the circuit. But a duplication of some gates can be performed to ensure that all the gates along the longest path have a fanout of exactly one. This is achieved by duplicating all the gates along P up to the last gate that has multiple fanout.

An example is shown in Figure 3.3. In the network shown at the top, path P is not sensitizable. Thus, both the stuck-*0* or stuck-*1* faults on the first edge of P, which is the connection from a to gate 1, cannot be tested through gate 6 to the output of P. However, each fault may be tested along paths through gates 7 or 8, thus causing the first edge of P to be irredundant. The network shown at the bottom is obtained by duplicating gates 1, 2, and 3, which includes all the gates between the first edge and the last gate with multiple fanout along P. As shown in the figure, all gates that are connected to gates along P in the original circuit, are now connected to the duplicate of the gate. In the example, gate 7 is fed from gate 12, which is the duplicate of gate 1 from the original circuit. Similarly, gate 8 has the fanin from gate 3 in the original circuit replaced by a fanin from gate 32 after the transformation.

This duplication retains the functionality of the circuit. As shown in Theorem 3.3.1, this duplication does not change the viability of any of the paths in the circuit and hence the delay of the circuit remains unchanged. On this new circuit, the longest path again cannot be responsible for determining the delay of the circuit. In fact, both viability and static sensitization of paths remain unchanged by duplication. Hence, the first edge on this new fanout-free path is not testable for either stuck-fault value. It can be set to either constant value without changing the logical functionality of the circuit. It is also shown that this does not increase the delay of the circuit (*c.f.* Theorems 3.3.1 and 3.3.2). This procedure is then repeated on the resulting circuit.

In summary, the procedure obtains an irredundant implementation of a given redundant circuit by an iterative loop of duplications and redundancy removals which are proven not to increase the delay of the circuit.

The algorithm, which is called the KMS algorithm[1], is presented in pseudo-code in Figure 3.4. The circuit on which the algorithm is performed must be composed of only simple gates. This avoids the problem of internal fanout within complex gates that has to be considered while determining the paths in the circuit. In converting a complex gate to an equivalent connection of simple gates (called a macro-expansion), the last gate is assigned a

[1]KMS is named after Keutzer, Malik, and Saldanha, who reported some of these ideas in [59]. Recent work is reported in [92].

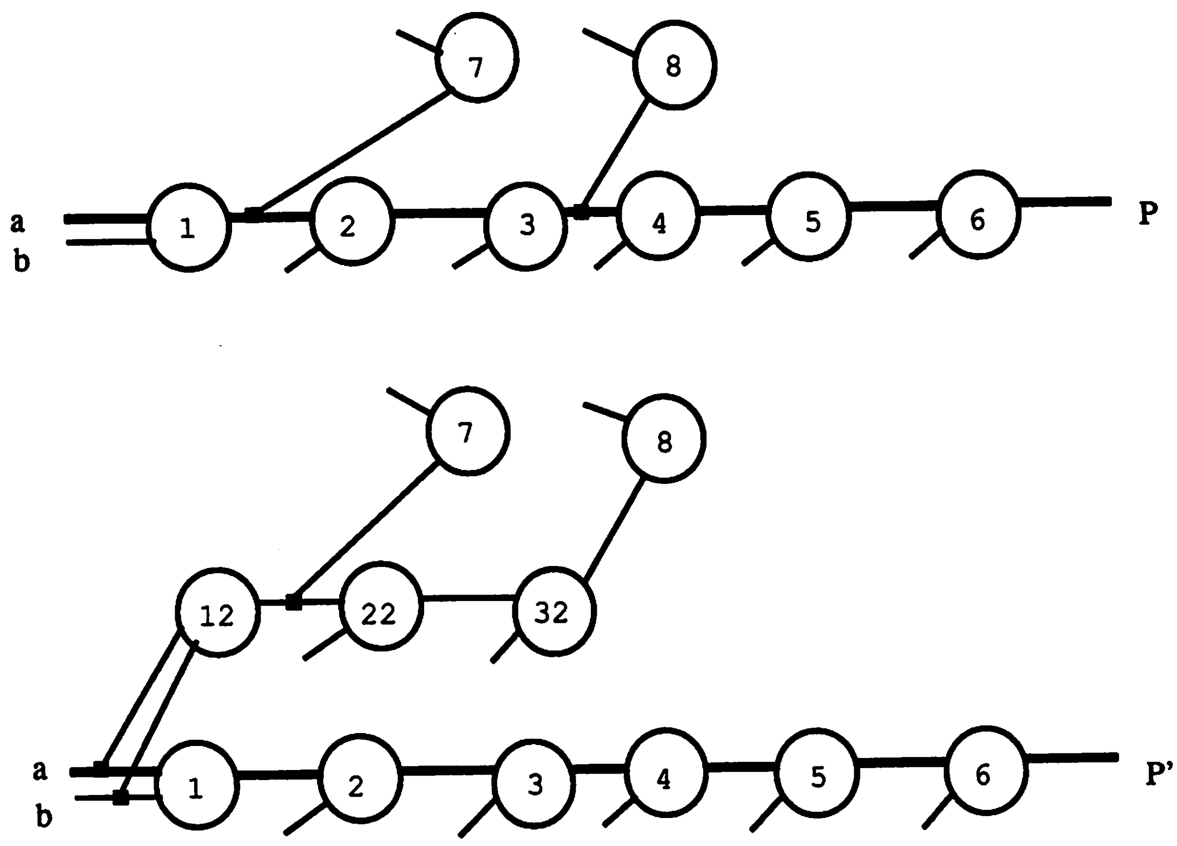


Figure 3.3: Example: Duplication to avoid fanout on longest path

delay equal to the delay of the complex gate. The other gates are assigned delays of zero[2].

When a gate n is duplicated, the duplicate n' is assigned the same delay as n and has the same fanin. The duplicate gate n' is said to *correspond* to the original gate n. The paths that include n' are said to *correspond* to the original paths through n in the original circuit before duplication. If none of the longest paths in the circuit is statically sensitizable/viable, one of the longest paths, say P, is picked for redundancy removal. If none of the gates in P has fanout greater than one, then the first edge of P is set to either *0* or *1*. In the case of multiple fanout, the gate n in P with multiple fanout closest to the output is determined. Let e be the fanout connection of n in P. All the gates between the first gate along P and n, together with their fanin connections are duplicated.

Let n' be the duplicated gate corresponding to n. Remove edge e from the output of n and reconnect it as the only fanout of n'. Now the longest path P', corresponding to P, consists only of gates with single fanout. Since path P' is not statically sensitizable/viable, setting the first edge of P' to either *0* or *1* and propagating this value as far as possible results in a logically equivalent circuit.

If at least one of the longest paths in the circuit is statically sensitizable/viable then the remaining redundancies are removed by using any redundancy removal scheme such as [100]. The redundancies are removed one at a time, and the remaining circuit redundancies must be recomputed after each removal.

Duplicating until the last node along P that has multiple fanout is only a sufficient condition. Let x be a node along P such that x is the last node along P for which there is a path P' through x such that the first edge of P is testable along P' and P' and P have the same set of gates and connections from the primary input up to x. Duplication of gates along P needs to be done only up to the gate x which may be less than duplicating all the gates up to the last multiple fanout point. However, in order to determine x, the testability of the first edge of P must be checked along all the paths that it lies on. This is a formidable task and hence the duplication done here is up to the last multiple fanout point. Although this may be more than necessary, it does not involve an expensive computation.

An implementation of the algorithm in Figure 3.4 may use either the condition of static sensitization of the longest path or the condition of viability. Note that if a path is not viable then it is not statically sensitizable, but the converse is not true [76]. Both

[2]This assumes a single delay for the gate. In the case of different pin-to-pin delays for a gate, a different assignment of delays must be made. Such an assignment is always possible, but is not considered here.

```
kms($\eta$) {
    /* Circuit $\eta$ has only simple gates.  */
    While (no longest path in $\eta$ is statically sensitizable / viable) {
        Choose a longest path $P$.
        Find $n$, the gate in $P$ closest to the output that has fanout $> 1$.
        If $n$ exists {
            Let $e$ be the fanout edge of $n$ that is in $P$.
            Let $\eta_n$ be the set of gates in $P$ and their fanin connections
                which lie between the primary input of $P$ and $e$.
            Duplicate $\eta_n$ to obtain $\eta'_n$.
            Let gate $n'$ in $\eta'_n$ correspond to $n$ in $\eta$.
            Change edge $e$ to be the single fanout of $n'$.
            Call the path in $\eta'$ corresponding to $P$ in $\eta$, $P'$.
        }
        Else {
            $P'$ is the same as $P$.
        }
        Set first edge of $P'$ to either constant $0$ or $1$.
        Propagate constant as far as possible, removing useless gates.
    }
    Remove remaining redundancies in any order.
}
```

Figure 3.4: KMS algorithm for redundancy removal with no increase in delay

these conditions involve testing the satisfiability of Boolean expressions derived from the circuit and hence their theoretical complexity is the same. In practice, as is shown later in Chapter 5, the computation of the static sensitization condition is typically faster than the viability condition. The delay analysis employed in the proofs uses viability analysis, but the proofs hold even while using the static sensitization condition in the implementation of the algorithm. In particular, the only penalty for this trade-off occurs if an unnecessary duplication is performed because a path is not statically sensitizable, but is viable.

3.2.2 Algorithm on the carry-skip circuit

The working of the KMS algorithm is demonstrated on the 2-bit carry-skip circuit. The algorithm in Figure 3.4 applies for multiple output circuits, but for ease of exposition it is shown performing here only on a single output circuit. This circuit corresponds to the sub-circuit that implements the carry bit, *c*2, of the 2-bit carry-skip adder in Figure 2.2. The initial redundant circuit with only simple gates is shown in Figure 3.5. It has a single redundancy and the output of the critical path is available after 8 gate delays through the carry-out bit, *c*2. The longest path *P* in the circuit in Figure 3.5 is from the input *c*0 along the unique path marked with an X to the output *c*2. If *P* is statically sensitizable, all the side-inputs to *P* must be at non-controlling values. This requires *p*0 and *p*1 to be at value *1* at the two AND gates along *P*, but at least one of *p*0 and *p*1 must be *0* for the MUX to sensitize *P*. Thus, *P* is not statically sensitizable. None of the edges in *P* have fanout greater than 1, hence, no duplication is required. On setting the first edge of *P* to *0* the circuit shown in Figure 3.6 is obtained. The longest path in this circuit is statically sensitizable and the remaining redundancies can be removed in any order. Thus, setting one of the two redundant connections marked with an x in Figure 3.6 to *1* (both untestable for stuck-*1* faults) results in the irredundant circuit in Figure 3.7. It is mentioned that if the algorithm is performed on the entire multiple output 2-bit adder circuit then a different version of an irredundant circuit is obtained that has the same number of gates and is also no slower than the original circuit.

3.3 Correctness of the algorithm

In this section, it is formally demonstrated that the procedure outlined in Section 3.2 is correct. First, it is shown that the duplication of gates, done so as to ensure that

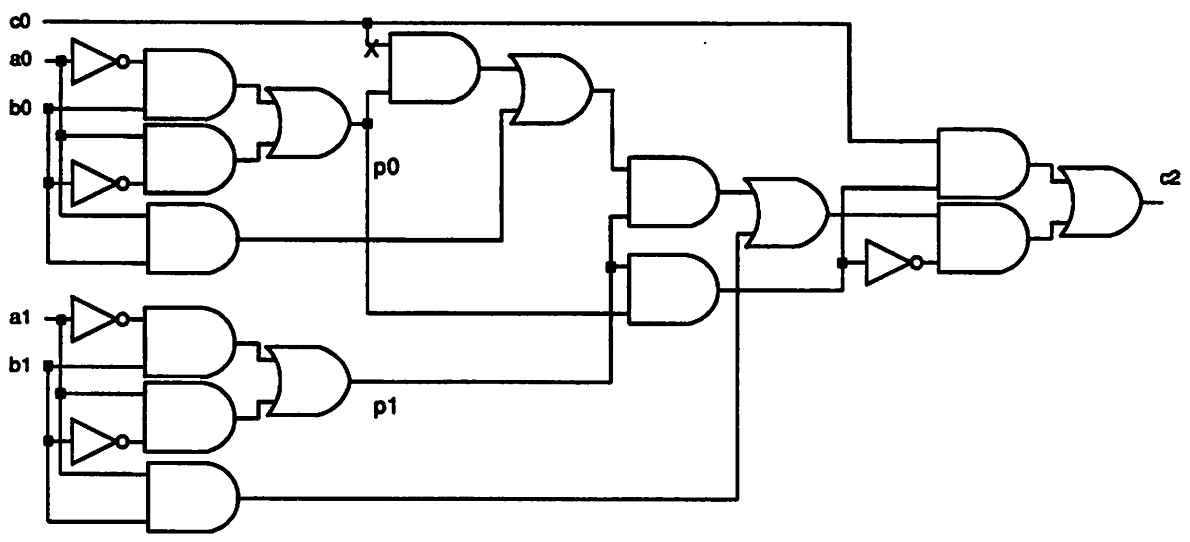


Figure 3.5: First intermediate 2-bit carry-skip circuit

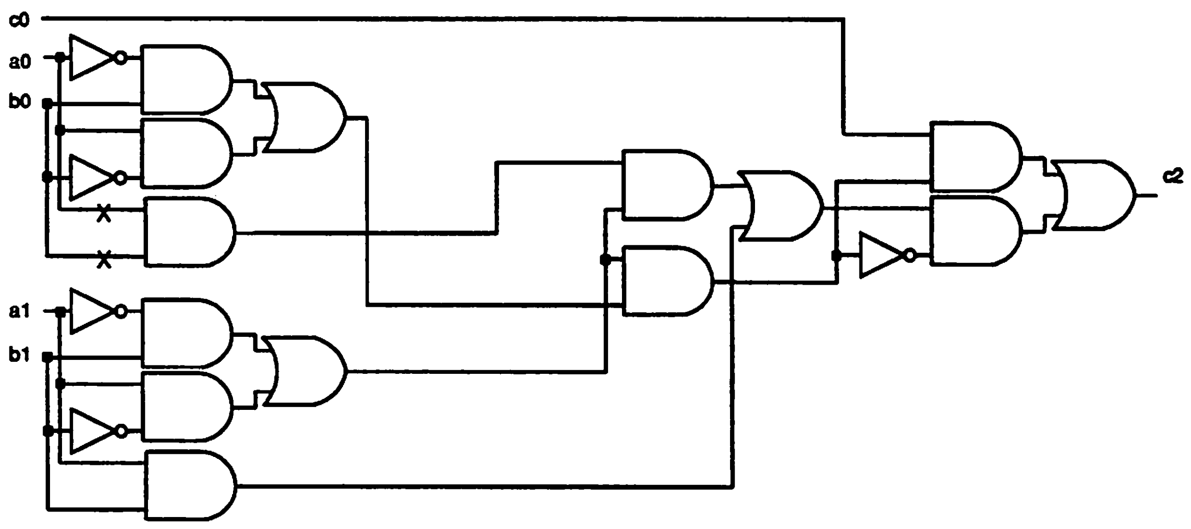


Figure 3.6: Second intermediate 2-bit carry-skip circuit

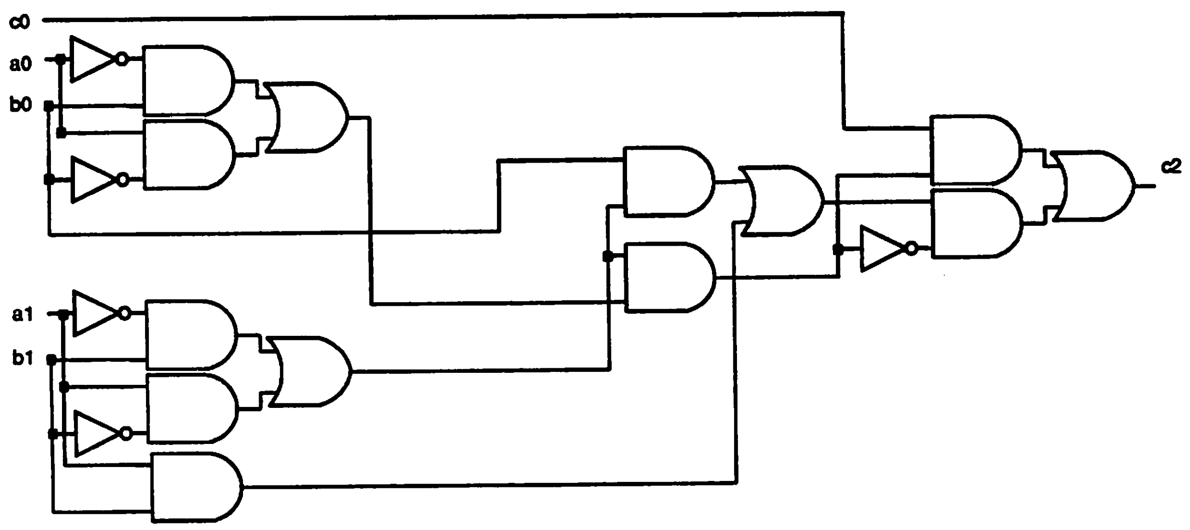


Figure 3.7: Final 2-bit carry-skip circuit

each gate along the longest path has a single fanout, does not increase the length of the longest viable path. The impact of the duplication step on the fanout, and the resulting impact on delays through the circuit, are not considered here. An explanation is provided in Section 3.6.2 that limits the fanout increase on each gate to a small amount. A variety of techniques are also described there that can easily handle this increase in fanin.

Theorem 3.3.1 *Let e be one of the fanout edges of a gate n, with fanout > 1 in circuit η, composed of simple gates. Make a duplicate n' of n and move edge e to n'. Then for every path P' in the new circuit η', there is a unique corresponding path P in η, of equal length. Moreover, $delay(\eta, c) = delay(\eta', c)$ for all cubes c.*

Proof The only paths changed in η' are the ones going through the edge e, but these are only changed by the fact that they go through n' instead of n. Since n' is a duplicate of n, it has the same delay as n. Hence, two corresponding paths P in η and P' in η' have equal length. Also, the logic function computed (in terms of the primary inputs of the circuit) at the output of each gate in η' is the same as that for the corresponding gate in η. Thus, the same delay and logical functionality exists along the corresponding paths (and side-paths) in η and η'. Viability analysis uses only path lengths and function values computed along paths and these remain unchanged by the duplication process. Hence the delay of η and η' remain equal. ■

The theorem above can be applied repeatedly for each gate that is duplicated. It assures that each duplication step does not slow down the circuit.

Next, the effect of setting the first edge of the longest path in a network to a constant value is examined. Assume that each gate along the longest path has single fanout. For the purpose of the proofs below, if a multiple-input gate becomes a single-input gate by deletion of some inputs, then this gate is not replaced by a wire. Instead, the delay on the gate and its input edge is reduced to zero to reflect the fact that it is equivalent to a wire.

Theorem 3.3.2 *Let P be a longest path in network η such that all gates along P have fanout 1. Let η' be the network obtained after setting the first edge of P to a constant value (either 0 or 1) and propagating this value as far as possible. Let v be the last value propagated, and let g be the gate where the propagation stopped. Then:*

1. *g has more than one input and v is a non-controlling value for g; and*
2. *if π is an IO-path in η', π is an IO-path in η; and*

3. *if π is viable under primary input cube c in η', it is viable under c in η. (Thus, $delay(\eta, c) \geq delay(\eta', c)$.)*

Proof

1. g has more than one input or else the constant value propagates beyond g. Also, v is the non-controlling value for g or else a constant value propagates beyond g.

2. η' is obtained from η by deleting the sub-network η'' consisting of the path Q from the first edge e_f of P up to g (and all the gates along Q). If π is an IO-path in η' then it is at least a path in η since $\eta' \subset \eta$. π is an IO-path in η because it is an IO-path in η', so its first element is an input terminal. Its last element is either g or some other element in η. Since the outputs in η and η' are identical, it is therefore an IO-path.

3. $delay(\eta', c)$ is determined by the length of the longest viable path π in η' under c. If π is also viable under c in η, then since it is an IO-path in η, $delay(\eta, c)$ must be at least the length of this path. We just need to show that π viable under c in η' implies π is viable under c in η.

   Let e_l denote the input along Q to gate g (e_l is the output of the sub-network η''.). If e_l is neither a side-input nor in the transitive fanin of a side-input of π (this happens only in networks with multiple primary outputs), then the gates and edges along π and its side-paths are the same in both η and η'. Thus, if π is viable under c in η' then it is viable under c in η. If e_l is a side-input or a transitive fanin of a side-input to π then the logical functionality of the side inputs of π are not the same in η and η'. This case is proved by induction on the depth of η'.

   Induction Basis: If depth = 1 then η' contains a single gate, g. Figure 3.8 illustrates this case. π consists of an input edge to and the output edge from this gate. η includes η'' connected to g by edge e_l. e_l is a side-input to π. For input cube c there are two possibilities:

   Case 1: Q is viable under c.

   Since e_f is the first edge of a longest path in η, e_l is a late side-input for π in η. Therefore, it is smoothed out in the viability analysis of π in η (*c.f.* Definition 2.3.13). If π was viable under c in η', it is viable under c in η as no other side

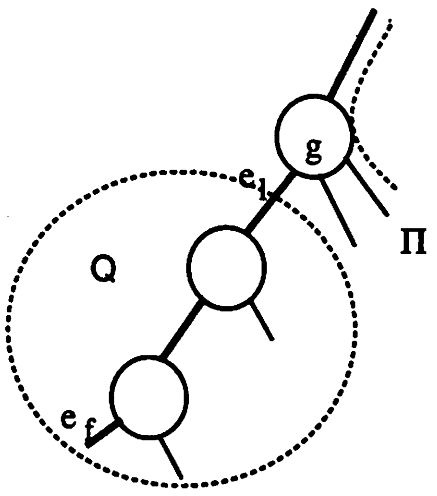


Figure 3.8: Base case for proof of KMS algorithm

inputs besides e_l have changed, and e_l is smoothed out in the viability analysis of π in η.

Case 2: Q is not viable under c.

Recall that if a path is not viable under a cube then it cannot be statically sensitized by that cube. Therefore, as far as this cube is concerned, the output of Q, e_l, is independent of the value of e_f and cannot change if e_f is changed. Thus set e_f to be the same value that was used to obtain η' from η. From (1) above, this results in a non-controlling value at input e_l of g. Thus, this cannot change the viability of π. If π is viable under c in η' it is viable under c in η.

Induction Hypothesis: Assume the theorem statement is true for all networks η' with depth $< k$.

Induction Step: Let η' be a network of depth k.

Case 1: e_l is a fanin of a gate along π in η.

Figure 3.9 illustrates this case. The proof for this case is similar to that for the basis case explained above. Either:

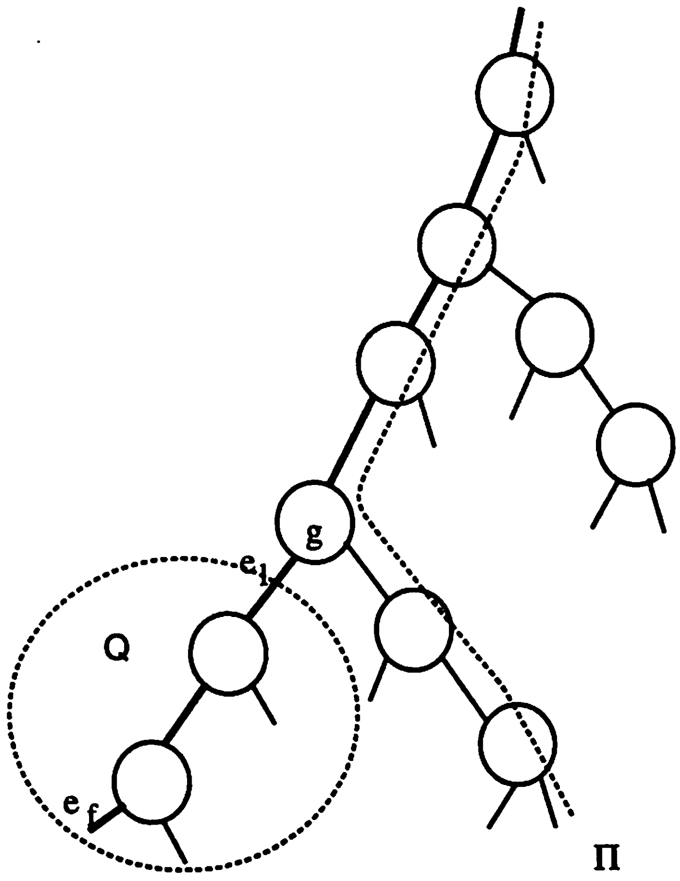


Figure 3.9: Induction case 1 for proof of KMS algorithm

- Q is viable under c in which case e_l is smoothed out in the viability analysis for π since it is a late side-input to π

  or

- Q is not sensitized by c, in which case e_l has the non-controlling value for its fanout gate, and thus cannot change the viability of π.

Case 2: e_l does not fan out to a gate along π in η. Figure 3.10 illustrates this case. There is some side-input x, to a gate h along π, such that e_l is in the transitive fanin of x. Let η_x be the network rooted at x in η and let η'_x be the network rooted at x in η'. Let P_x be the part of path P from e_f up to and including the output edge of x in η. For input cube c, one of the following must happen:

- P_x is viable under c. Since, P is a longest path in η, P_x is longer than any path that ends in h. Hence, x is smoothed out in the viability analysis of π.
- P_x is not viable under c. In this case P_x is not sensitized by c. Therefore, as far as this input cube is concerned, the first edge in P_x (i.e. e_f) may be set to any value without changing the output of η_x. (Recall that each gate in P_x has fanout of exactly one.) In particular, set the value of e_f to the value it was set in order to obtain η'. This does not change the output of η_x. This implies that $\eta_x(c) = \eta'_x(c)$. Also note that since η' has depth k, η'_x has depth $< k$. From the induction hypothesis $delay(\eta_x, c) \geq delay(\eta'_x, c)$. If x was smoothed out in the viability analysis of π in η', then in the viability analysis of π in η, it will be smoothed out too, since η_x is at least as slow as η'_x in responding to c. If x was not smoothed out in the viability analysis of π in η', then, since $\eta_x(c) = \eta'_x(c)$, the static value of x is the same for c in η and η'. Thus, the viability of π is unchanged. ■

The two theorems presented in this section guarantee that each iteration of the *while* loop in Figure 3.4 maintains the invariant that the delay of the initial circuit is not increased. This is sufficient to guarantee that at the termination of the algorithm the final circuit is no slower than the initial circuit.

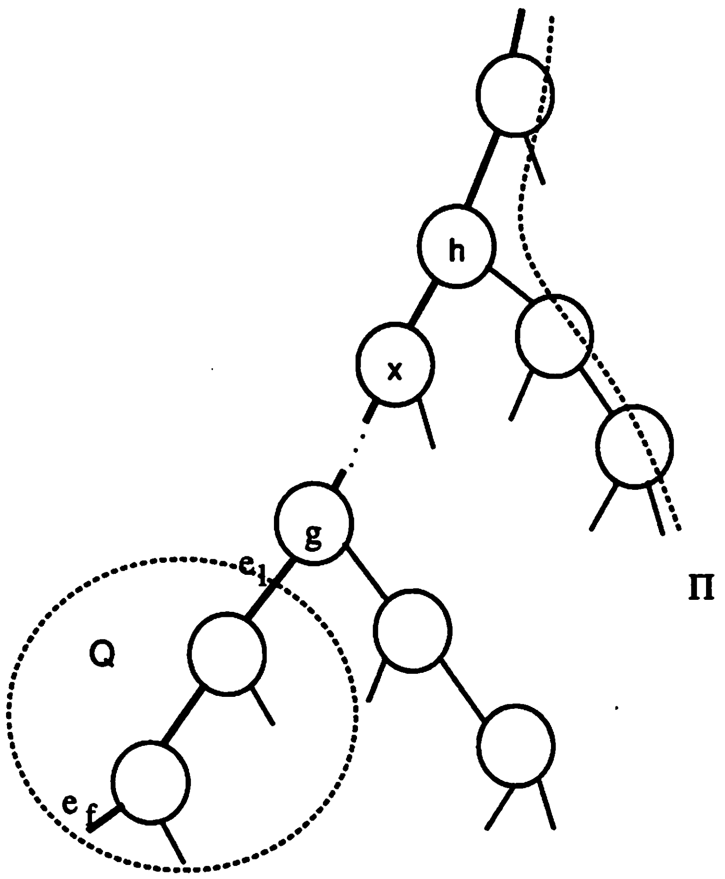


Figure 3.10: Induction case 2 for proof of KMS algorithm

3.4 Algorithm implementation

Four practical issues that must be addressed before obtaining an efficient implementation of the KMS algorithm of Section 3.2 are now discussed.

3.4.1 Timing analysis

By far the most important aspect of the KMS algorithm with regard to its efficiency is timing analysis. The condition checked in each loop of the algorithm is the existence of a sensitizable (using viability analysis or static sensitization) longest path. There are two issues of concern. First, the sensitization condition must be computed using an efficient algorithm. Second, since there may exist several paths of longest length, it would seem that the sensitization condition must be determined on each path individually to determine if any longest path is true. However, this is not the case and the existence of a sensitizable longest path can be checked in a single step. The formulation that solves both these problems is detailed in Chapter 5.

3.4.2 Path duplication

If timing analysis proves the absence of any sensitizable longest path, a longest path must be selected for duplication and subsequent redundancy removal on its first edge. Since the area increase is to be minimized, a candidate path that requires the minimum amount of duplication is selected.

The pseudo-code in Figure 3.11 shows how this is done. The algorithm has the effect of scanning each path to determine the last gate along the path that has fanout greater than one. A longest path is selected such that this gate is closest to the primary inputs.

3.4.3 Redundancy removal

Once a fanout-free unsensitizable longest path is obtained, its first edge may be set to either *0* or *1*. It is set to the controlling value of the gate it feeds, since this deletes the gate from the circuit. Since an unsensitizable (using the viability or static sensitization condition) fanout-free path implies that the first edge is redundant, the removal can be done without resorting to test pattern generation. Thus, only one application of conventional

```
choose_min_duplication($\eta$) {
    /* Circuit $\eta$ has no sensitizable longest path */
    Perform a delay trace on $\eta$.
    node_list = list of nodes in $\eta$ in topological order.
    For each node $n$ in node_list {
        If $n$ is on some critical path {
            If fanout($n$) > 1 {
                /* all critical paths through $n$ must be
                 duplicated at least up to $n$ */
                dup[$n$] = level($n$).
            }
            Else {
                /* use duplication information from its fanin */
                min_dup_fanin = fanin of $n$ with minimum dup.
                dup[$n$] = dup[min_dup_fanin].
            }
        }
    }
    min_dup_po = primary output with minimum dup.
    Return longest path to min_dup_po that corresponds to minimum dup.
}
```

Figure 3.11: Algorithm for selecting a longest path for duplication

redundancy removal, such as [100], is required after the longest path becomes statically sensitizable.

3.4.4 Area recovery

Consider the circuit shown in Figure 3.12, where the longest paths are shown in bold. Assume that both paths are unsensitizable. The KMS algorithm performs redundancy removal only on a fanout-free longest path (*c.f.* Section 3.2). On applying the KMS algorithm to the initial circuit (1) in the figure, the circuit structure evolves to circuit (2), so that a longest (and non-sensitizable) path is fanout-free. Following redundancy removal on the two fanout-free longest paths of circuit (2), assume circuit (3) is irredundant and has a longest sensitizable path. Some area may be reclaimed by merging identical gates as shown in circuit structure (4) of the figure. In fact, for this example even though duplication increases the number of gates, this increase in area is offset by the merging step. In general, no tight bound is known for the increase in area. Nonetheless, the merging step can always be attempted in a practical implementation to reclaim area. However, it must be ensured that the delay of the circuit does not increase by this step.

Theorem 3.4.1 *Let n_1 and n_2 be gates with identical fanin in a network η. Merge n_1 and n_2 to obtain gate n in a new circuit η'. Then for every path P' in η', there is a unique corresponding path P in η, of equal length. Moreover, $delay(\eta, c) = delay(\eta', c)$ for all cubes c.*

Proof Similar to Theorem 3.3.1. ∎

While Theorem 3.4.1 proves that both the static sensitization or viability remain unchanged by merging, testability may be changed on merging of gates. Even though merging may introduce redundancy (this phenomenon is illustrated and explained in Section 4.4.4), the new redundancy can be removed without recourse to the KMS algorithm since redundancy removal on a circuit with the longest path sensitizable cannot slow down the circuit (*c.f.* Theorem 3.1.1).

The effectiveness of the area recovery step will be illustrated on a number of examples in Section 3.5. The effects due to the small fanout increase are ignored in the analysis, and are addressed in Section 3.6.2.

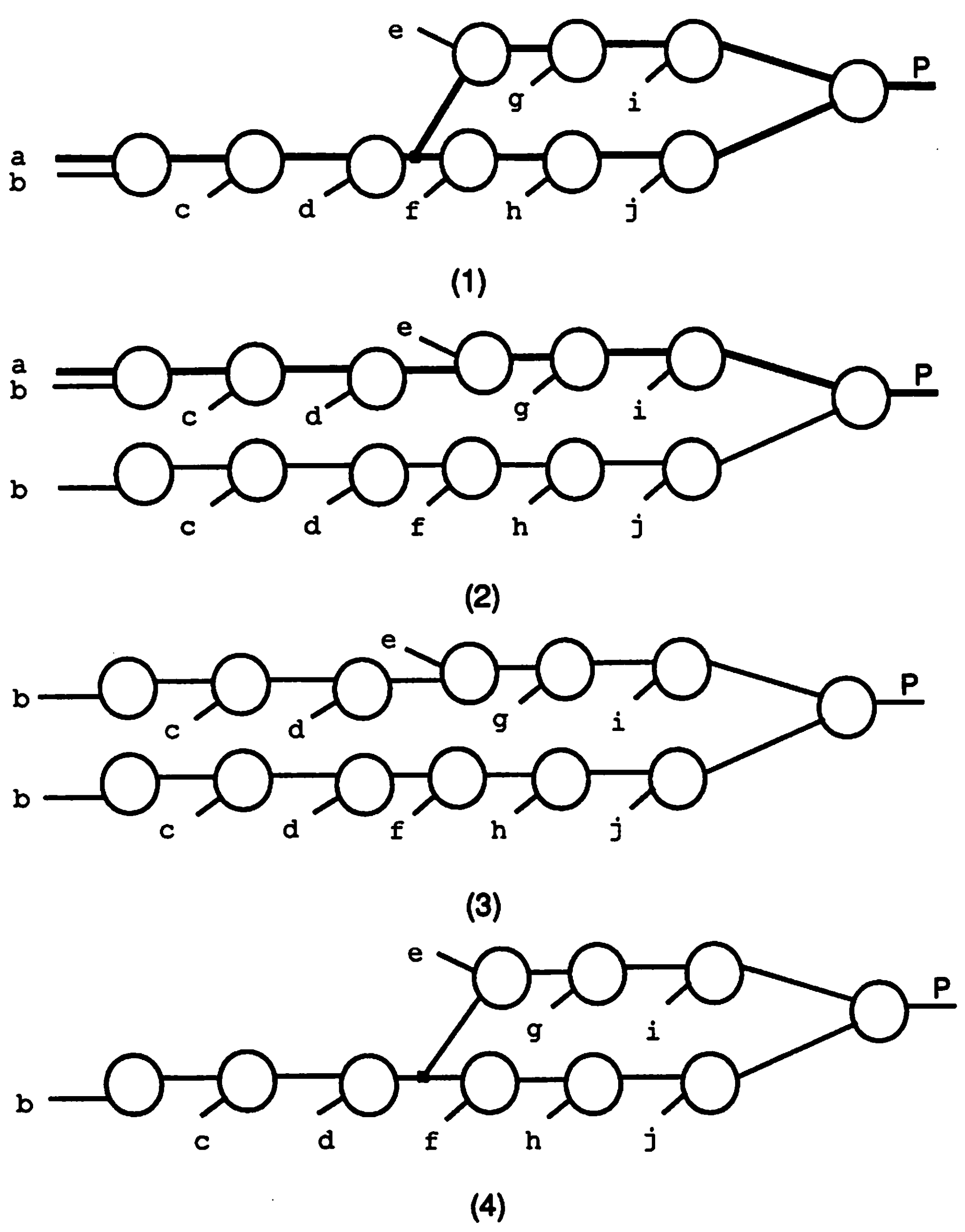


Figure 3.12: Example: Merging of gates following KMS algorithm

3.5 Results using the KMS algorithm

The algorithm described in Section 3.2 has been implemented in the SIS logic synthesis system at U.C. Berkeley [18, 102]. The results were obtained by running the algorithm on two classes of circuits: several carry-skip adders with varying block sizes, and some optimized MCNC and ISCAS benchmark examples [69]. Circuit size is measured by counting the number of simple gates. Though a unit gate delay model is used here, any delay model, including a library gate delay model, may be used. The first column in each table is the name of the circuit; the second indicates the number of redundancies in the initial circuit; the third gives the longest path (or topological) delay and the fourth the computed delay of the initial circuit. The computed delay is measured using the techniques for viability analysis described in Chapter 5. The fifth column lists the computed delay of an irredundant circuit obtained by standard redundancy removal without consideration of the circuit delay. The sixth column gives the computed delay of the irredundant circuit obtained using the KMS algorithm. In this case, the computed delay is always equal to the longest topological delay. The last three columns compare the size of the initial redundant circuit against the irredundant circuit obtained by applying redundancy removal and the KMS algorithm. The area estimate in the last column gives the final area of the irredundant circuit following the area recovery step (*c.f.* Section 3.4.4) on completion of the KMS algorithm.

3.5.1 Adders

First, the effect of the algorithm on carry-skip adders of various block sizes is described. Recall from Section 3.1 that a carry-skip adder of n bits can be constructed by cascading a set of individual carry-skip adder blocks such that the sum of the block sizes is n. Here blocks of equal size only are considered. In the *csa* examples shown in Table 3.1 the first digit refers to the number of bits in the adder and the second digit indicates the size of each block. For example, *csa 8.4* indicates a 8-bit carry-skip adder composed of two 4-bit blocks. Each block of the adder initially contains two redundancies: one on the AND gate that feeds the MUX and one within the MUX itself. Two implementations each of the ripple-carry and carry-lookahead adders are also reported in the table to compare the relative area, delay and testability of each of the adder designs. As mentioned earlier, the carry-skip adder lies between the ripple-carry and carry-lookahed adder in terms of both area and performance. However, using even fixed block sizes in the carry-skip adder along

Name	# Red.	Initial delay		Final delay		# Gates		
		Long.	True	RR	KMS	Init.	RR	KMS
ripple.16	0	31.0	31.0	31.0	31.0	139	139	139
lookahead.16	0	16.0	16.0	16.0	16.0	172	172	172
ripple.32	0	63.0	63.0	63.0	63.0	283	283	283
lookahead.32	0	28.0	28.0	28.0	28.0	361	361	361
csa 2.2	2	8.0	8.0	6.0	6.0	22	18	18
csa 2.2†	2	11.0	9.0	9.0	6.0	22	18	21
csa 4.2	4	14.0	12.0	10.0	10.0	44	36	43
csa 4.4	2	12.0	12.0	10.0	10.0	40	36	36
csa 8.4	4	22.0	20.0	18.0	18.0	80	72	87
csa 16.8	4	38.0	36.0	34.0	36.0	152	144	175
csa 8.2	8	26.0	16.0	18.0	14.0	88	72	91
csa 16.2	16	50.0	24.0	34.0	*	176	144	*
csa 16.4	8	42.0	24.0	34.0	22.0	160	144	179
csa 32.4	16	82.0	32.0	66.0	*	320	288	*
csa 32.8	8	74.0	40.0	66.0	38.0	304	288	355
csa 64.8	16	146.0	48.0	130.0	*	608	576	*

RR: redundancy removal

KMS : KMS algorithm

All primary input arrival time set at 0.0 delay units

* : Did not finish in 10 hours

†: Carry-input arrival time set at 5.0 delay units

Final circuit verified against initial circuit

Table 3.1: KMS algorithm versus redundancy removal on adders

with the crude delay model of assigning each gate a unit delay, the carry-skip adder appears comparable to the carry-lookahead adder in performance with considerably lower area. The comparison using variable block sizes and more realistic delay models is not done here and is reported in [66, 50, 83].

Consider the carry-skip adders composed of 2-bit blocks in the table. For the first example, *csa.2.2*, all primary inputs arrive at the same time while for the circuit *csa.2.2*†, the arrival time of the carry-input is set to 5.0 delay units after the other primary inputs. In both cases, naive redundancy removal does not slow down the circuit. This may appear contradictory to the discussion in Section 2.5, where it was stated that redundancy removal slows down the 2-bit carry-skip adders. However, the earlier discussion considered the carry circuit independent from the circuit for the sum outputs. All the examples considered here are multiple output circuits including the carry and all the sum bits. Thus, in the case of the 2-bit adder *csa2.2*, where all inputs arrive simultaneously, there exists at least one longest path through the sum outputs. While the longest path through the carry-output is false, a longest path through the sum outputs is true. Hence the delay is the length of the longest path. Both redundancy removal and KMS reduce the delay of the circuit to 6.0 units from 8.0 units. In *cas 2.2*†, the unique longest path is from the carry-input to the carry-output. One would expect redundancy removal to result in a slow-down of the circuit. This does not, even though the longest false path becomes true on redundancy removal. The reason is that the length of the longest path after redundancy removal happens to be reduced to 9.0 units instead of the original length of 11.0 units, using the simple delay model. The same phenomenon is observed in *csa.4.2*.

Both *csa.8.2* and *csa.16.2* slow down substantially on redundancy removal. The KMS algorithm yields faster circuits in both cases with some increase in the area. For *csa.8.2*, the irredundant circuit after KMS but no merging, has 126 gates compared to the original 88. However, as explained in Section 3.4 many of the gates are identical and can be merged. This merging does not change the delay of the circuit (*c.f.* Theorem 3.4.1). The final irredundant circuit after applying this step is 91 gates, with a total increase of only 3 gates from the original circuit. Notice that the delay even improves from 16.0 to 14.0 units.

The algorithm does not terminate within 10 hours on the examples *csa 16.2*, *csa 32.4* and *csa 64.8*. This is because the algorithm removes one false path at a time. All three examples have a huge number of false paths; for example, *csa 16.2* has 44,848 false paths equal or longer than the true delay of the circuit. This requires an inordinate number of iterations of the KMS algorithm. This problem is overcome in Section 3.6 where a single-pass approach is developed that avoids explicit enumeration of each false path.

3.5.2 Optimized circuits

The examples *5xp1*, *rot* and *des* in Table 3.2 are circuits from the MCNC benchmark set that have been optimized for area using an older version of SIS [18]. Each of these initial optimized circuit has two properties: first, there is at least one redundant single stuck-fault; second, the longest path delay is not the computed delay of the circuit.

The second set of examples, *clip* and *duke2*, are representative of examples with redundancy but with the computed delay equal to the longest path delay. Thus, for these circuits, at least one longest path is already sensitizable. Here the algorithm removes the redundancy in any order since the delay of the circuit cannot increase. Thus RR and KMS are identical.

The third set of examples, *f51m*, *misex2*, *rd73*, *sao2* and *z4ml* are circuits that are optimized for delay using the timing optimization commands in SIS on circuits that had been initially optimized for area [105]. Each is irredundant before delay optimization, but each resulting timing optimized circuit either has at least one redundant single stuck-fault, or its longest path delay is not the computed delay (*e.g. f51m*).

The final examples, *misex1*, *bw* and *z4ml* are irredundant but the computed delay is less than the topologically longest delay. Hence the KMS algorithm need not be used. However, despite being fully testable, tests for the faults on the first edge of each longest path are tested by propagating the fault effect along paths different from the longest paths. Thus, on applying the KMS algorithm, the resulting topologically longest path is reduced at little cost in area (*e.g. misex1* and *bw*). It is interesting to note that even though the algorithm need not be applied to these irredundant circuits, it may yield an even faster circuit (*e.g. bw*) than that supplied by the SIS timing optimization procedure [105], although at a further area penalty. Additionally, there are no long false paths in the resulting circuits.

The algorithm does not complete on *rot* and *des*. It also fails to complete on any of the ISCAS circuits that have redundant faults and false longest paths. The reason for this is the same as for the large adder examples; there are an enormous number of false long paths. These examples motivate the need for a single pass KMS procedure.

Name	# Red.	Initial delay		Final delay		# Gates		
		Long.	True	RR	KMS	Init.	RR	KMS
5xp1	1	11.0	9.0	9.0	9.0	58	58	57
rot	37	19.0	17.0	19.0	*	437	424	*
des	17	15.0	13.0	13.0	*	2007	2000	*
clip	4	8.0	8.0	7.0	7.0	64	61	61
duke2	2	9.0	9.0	9.0	9.0	190	190	190
f51m	39	18.0	17.0	16.0	16.0	173	139	139
misex2	6	7.0	7.0	7.0	7.0	94	90	90
rd73	10	11.0	11.0	11.0	11.0	94	83	83
sao2	9	12.0	12.0	12.0	12.0	126	119	119
z4ml	3	10.0	10.0	10.0	10.0	47	42	42
misex1	0	9.0	7.0	7.0	7.0	28	28	31
bw	0	20.0	14.0	14.0	10.0	85	85	111
z4ml	0	7.0	7.0	7.0	7.0	30	30	30

RR: redundancy removal

KMS : KMS algorithm

All primary input arrival time set at 0.0 delay units

* : Did not finish in 10 hours

Final circuit verified against initial circuit

Table 3.2: KMS algorithm versus redundancy removal on MCNC circuits

3.6 A single-pass algorithm

Due to the huge number of false long paths in many circuits, it is imperative that any efficient algorithm must not explicitly enumerate each false path while performing the KMS transformation. This section develops such an algorithm and proves its correctness.

Definition 3.6.1 *The set of all the paths beginning at connection c and terminating at a primary output is called the* **path-set** *of c, and is denoted $\mathcal{PS}_c$.*

Note that the paths in the path-set of a connection are IO-paths only if the connection is from a primary input.

Consider a connection c from a primary input in a circuit η. Additionally, assume the computed delay of η is $< L$ and let every path in $\mathcal{PS}_c$ be of length $\geq L$. Note that every path in $\mathcal{PS}_c$ is an IO-path. On completion of the KMS algorithm of Figure 3.4, all the paths in the resulting circuit, η', are of length $< L$. The KMS algorithm removes connections between primary inputs and some gates, *i.e.* only the first edge of any path is removed. Thus, c cannot exist in η', since every IO-path through c is of length $\geq L$. This notion is captured formally by the next definition and theorems.

Definition 3.6.2 *A L-***path-disjoint circuit** *is one where the paths in $\mathcal{PS}_c$, for any primary input connection c, are either all of length $\geq L$ or all of length $< L$.*

In other words, the path-set of the first edge of any path in the circuit either contains only paths of length $\geq L$ or only paths of length $< L$.

Theorem 3.6.1 *Let η be a circuit whose longest viable path is of length $< L$. Let $C = \{c\}$ be the set of all primary input connections such that each path in $\mathcal{PS}_c$ is of length $\geq L$ (and hence non-viable). Then any multiple stuck-fault composed of any combination of single stuck-0 or stuck-1 faults on each $c \in C$ is redundant.*

Proof Assume that some multiple stuck-fault F_C on C is not redundant. Let a vector v be a test for the fault. Consider the set of IO-paths P_{F_C} that propagate the fault effect to a primary output under vector v. The length of each $Q \in P_{F_C}$ is $\geq L$ by assumption. Pick a path $Q = \{f_0, f_1, ..., f_k\} \in P_{F_C}$ with the property that for each $f_i \in Q$, f_{i-1} under v is the earliest arriving input of f_i that propagates the fault effect. Such a path Q exists since the fault effect is propagated under v along some path in P_{F_C}. Consider each gate f_i along

Q in circuit η when vector v is applied. All the side-inputs to f_i that do not propagate the fault effect are at non-controlling values. The remaining side-inputs each propagate the fault effect but each of them is available only after the time input f_{i-1} arrives at f_i. Hence these side-input are smoothed out when considering the viability of the sub-path Q from f_0 up to f_i. Hence the sub-path of Q up to f_i is viable. Since this is true at f_k, Q is viable. This contradicts that no path through c is viable. ∎

Theorem 3.6.1 proves that the logical behavior of a circuit that meets the conditions of the theorem remains unchanged by a multi-fault removal. The next theorem proves that the delay, measured using viability analysis, remains unchanged under the redundancy removal.

Theorem 3.6.2 *Let η be a circuit whose longest viable path is of length $< L$. Let $C = \{c\}$ be the set of all connections from a primary input such that each path in $\mathcal{PS}_c$ is of length $\geq L$ (and hence non-viable). Let η' be the circuit resulting after asserting a multiple stuck-fault composed of any combination of the single stuck-*0 *or stuck-*1 *faults on each $c \in C$. For any viable path π' in η', the corresponding path π is viable in η.*

Proof Similar to the proof of Theorem 3.3.2. The difference is that each gate may have several late-arriving side-inputs that are set to constant non-controlling values by the redundancy removal. However, these side-inputs always get smoothed out and do not change the viability conditions of any path π' in η' from the viability of the corresponding path π in η. ∎

With Theorems 3.6.1 and 3.6.2 it is apparent that a multi-fault redundancy removal of the type specified can be removed without increasing the computed delay or changing the logical behavior of the circuit. Moreover, no duplication is performed. Of course, not all circuits may have such connections. A procedure is now described that transforms every circuit to a functionally equivalent L-path-disjoint circuit. In this case, all paths of length $\geq L$ will be removed by the application of the above theorems. The only operation used is the duplication of gates and the transfer of connections from a gate to its duplicate.

Definition 3.6.3 *The distinct paths lengths from primary inputs to a gate f is denoted by* **atimes(f)**. *The distinct path lengths from each gate f to the primary outputs is denoted by* **etimes(f)**.

If f is a primary input, *atimes(f)* is the single arrival time specified for f. If f is a primary output, *etimes(f)* is *0*. However, the algorithm could be generalized to take account of required times at the output; just take the maximum required time, R_{max}, and think of a buffer on each output f with delay $R_{max} - R_f$. The algorithm to derive an L-path-disjoint network for a given network η is described in Figures 3.13 through 3.16. The main procedure, shown in Figure 3.13, consists of three phases.

The first phase consists of computing the distinct paths lengths from primary inputs to each gate f, and the distinct path lengths from each gate f to the primary outputs. This is done by simply performing a topological traversal from the inputs to the outputs for computing *atimes*, and a reverse topological traversal for computing *etimes* (Figure 3.14).

The second phase consists of gate duplication and transfer of connections from a gate f to one or more duplicates of f, and is shown in Figure 3.15. The essential operation performed on a gate with multiple fanout is the transfer of a set of fanout connections of the gate to a duplicate gate. The gates are traversed in reverse topological order from primary outputs to primary inputs. Let f be a gate that is to be processed by the algorithm. Let P_f be any path from a primary input up to f. A duplication is only performed if there are at least two paths, $Q1_f$ and $Q2_f$ from f to the primary outputs such that[3] $|P_f|+|Q1_f| < L$ and $|P_f| + |Q2_f| \geq L$. Following the duplication one or more fanout connections are transferred from f to its duplicate f_{dup}, according to the following rule. Let Q_f represent any path from f to the primary outputs after some fanout connections are transferred to f_{dup}. Then $|P_f| + |Q_f| \geq L$ for any P_f and Q_f. Lemma 3.6.1 below ensures that this condition can always be satisfied. It is important to note that *etimes(f)* is updated whenever a fanout connection is moved to f_{dup}. Similarly, *etimes(f_{dup})* is also updated to reflect the paths through the fanout connections transferred from f. Duplication is not done on any node with only one path to any output. Hence primary outputs are not duplicated.

This transformation is repeated on f_{dup} (the assignment $g = g'$ in Figure 3.15) until no further duplication and transfer of fanout connections is required. It is now shown that the resulting network after all the gates are processed by the algorithm is a L-path-disjoint network.

Definition 3.6.4 *The* **level** *of a node is the maximum number of nodes along any path from the node to the primary outputs. The level of a primary output node with no fanout is* 0 .

[3]Recall that $|P|$ denotes the length of path P.

```
/* Derive an irredundant network no slower than $\eta$.
$\eta$ has no sensitizable paths of length $\geq L$.  */
single_pass_kms($\eta$, $L$) {
    /* Circuit $\eta$ has only simple gates.
    $atimes(f)$ are the different path lengths from primary inputs to $f$.
    $etimes(f)$ are the different path lengths from $f$ to primary outputs.  */

    /* Compute $atimes(f)$ and $etimes(f)$ for each node $f$.  */
    kms_setup_times($\eta$, $atimes$, $etimes$).

    /* Duplicate gates */
    $node\_list$ = list of gates in $\eta$ in reverse topological order.
    Foreach node $f$ in $node\_list$ {
        kms_duplicate_gate($f$, $\eta$, $L$, $atimes$, $etimes$).
    }

    /* Set constants on first edge of paths of length $\geq L$.  */
    kms_set_constant($\eta$, $L$, $atimes$, $etimes$).
    Propagate constants as far as possible.
    Remove remaining redundancy in any order.
}
```

Figure 3.13: Single-pass algorithm for redundancy removal with no increase in delay

```
kms_setup_times($\eta$, $atimes$, $etimes$) {
    Perform a delay trace on the network.
    Foreach node $f$ of $\eta$ {
        $atimes(f) = \{\}$.
        $etimes(f) = \{\}$.
    }
    /* For each node $f$, compute $atimes(f)$.
    $A_f$ is the arrival time at $f$.  */
    $node\_list$ = list of gates in $\eta$ in topological order.
    Foreach node $f$ in $node\_list$ {
        If $f$ is a primary input {
            $atimes(f)$ = $A_f$.
        }
        Else Foreach fanin $g$ of $f$ {
            $atimes(f) = \{u + d(f,g) | u \in atimes(g)\}$.
        }
    }
    /* For each node $f$, compute $etimes(f)$ */
    $node\_list$ = list of gates in $\eta$ in reverse topological order.
    Foreach node $f$ in $node\_list$ {
        If $f$ is a primary output {
            $etimes(f)$ = 0.
        }
        Else Foreach fanout $g$ of $f$ {
            $etimes(f) = \{u + d(f,g) | u \in etimes(g)\}$.
        }
    }
}
```

Figure 3.14: Path length calculations in the single-pass algorithm

```
kms_duplicate_gate($f$, $\eta$, $L$, $atimes$, $etimes$) {
    $g = f$.
    Foreach time $t \in atimes(f)$ in ascending order {
        If ($t$ + min($etimes(g)$) $< L$ && $t$ + max($etimes(g)$) $\geq L$) {
            /* some fanout of gate must be split */
            $g'$ = duplicate_gate($g$).
            $atimes(g') = atimes(g)$.
            $etimes(g') = etimes(g) - \{t_e | t_e \in etimes(g), t + t_e \geq L\}$.
            $etimes(g) = etimes(g) - \{t_e | t_e \in etimes(g), t + t_e < L\}$.
            Foreach fanout $h$ of $g$ {
                If ($t$ + min($etimes(h)$) + $d(g,h)$ $< L$) {
                    Replace connection from $g$ to $h$ by $g'$ to $h$.
                }
            }
            /* repeat on duplicate gate of $g$ */
            $g = g'$.
        }
    }
}
```

Figure 3.15: Gate duplication in the single-pass algorithm

```
kms_set_constant($\eta$, $L$, $atimes$, $etimes$) {
    /* $\eta$ is a $L$-path-disjoint network */
    Foreach primary input $f$ of $\eta$ {
        /* $atimes$(f) has exactly one entry */
        $t = atimes(f)$.
        Foreach fanout $g$ of $f$ {
            If ($t + d(f,g)$ + min($etimes(g)$) $\geq L$) {
                Replace $g$ by constant 0 or 1.
            }
        }
    }
}
```

Figure 3.16: Setting constants on false paths in the single-pass algorithm

Lemma 3.6.1 *Consider gate f, in network η, that is processed by the algorithm of Figure 3.15. Assume that the lists* atimes(f) *and* etimes(f) *are computed using the procedure of Figure 3.14. Let f' refer to gate f or any of its duplicates*[4]. *Then the following invariant is true for each f': for each $f'_{atime} \in$* atimes(f'),[5] *f'_{atime} +* etimes(f') *$< L$, or f'_{atime} +* etimes(f') *$\geq L$.*

Proof Note that the gates of network η are processed in reverse topological order. The proof is by induction on the level of a node f. Also note that the invariant holds trivially if *etimes*(f) has only one element.

Induction Basis: If level = 0, then f is a primary output with no fanout. Thus, *etimes*(f) = {*0* }. Let P_f be any path from a primary input to f. Since $|P_f| + etimes(f)$ is either $< L$ or $\geq L$, the invariant holds.

Induction Hypothesis: Assume the invariant is true for all gates of level $< k$.

Induction Step: Let f be a gate of level k.

min(*etimes*(f)) and max(*etimes*(f)) represent the minimum and maximum times, respectively, in the list *etimes*(f).

Case 1: f has single fanout.

Let h be the single fanout of f. By definition, h has level $< k$. By the induction hypothesis, either $h_{atime} + etimes(h) < L$, or, $h_{atime} + etimes(h) \geq L$, for each $h_{atime} \in atimes(h)$. But, $atimes(h) \supseteq atimes(f) + d(f,h)$. Rewriting the invariant for h, $f_{atime} + d(f,h) + etimes(h) < L$, or, $f_{atime} + d(f,h) + etimes(h) \geq L$. But, $etimes(f) = etimes(h) + d(f,h)$, since f has single fanout. On further rewriting of the invariant for h, $f_{atime} + etimes(f) < L$, or, $f_{atime} + etimes(f) \geq L$. Therefore, the invariant holds for f also.

Case 2: f has multiple fanout, $t + \max(etimes(f)) < L, \forall t \in atimes(f)$.

In this case, each path through gate f is of length $< L$ and the invariant holds.

Case 3: f has multiple fanout, $t + \min(etimes(f)) \geq L, \forall t \in atimes(f)$.

In this case, each path through gate f is of length $\geq L$ and the invariant holds.

[4]There is no loss in generality in referring to the gates by a single representative f', since, for each path from the primary inputs to f, there exists a corresponding path to some duplicate gate of f.

[5]The notation $x + S < L$, for x a scalar and S a set, means that $x + s_i < L$ for all $s_i \in S$. $x + S \geq L$ has an analogous meaning.

Case 4: f has multiple fanout, $t + \min(etimes(f)) < L$ and $t + \max(etimes(f)) \geq L$ for some $t \in atimes(f)$.

This is the case where a duplication and transfer of fanout connections is performed. Let t_{min} represent the smallest time $t \in atimes(f)$ for which this condition holds. Let f_{dup} be a duplicate of f. Each fanout h of f satisfies either $t_{min} + d(f,h) + \min(etimes(h)) < L$ or $t_{min} + d(f,h) + \min(etimes(h)) \geq L$. If the first condition holds the connection from f to h is replaced by the connection from f_{dup} to h. Nothing is done if the second condition is true.

Consider the gate f after all its original fanout connections are processed. For each fanout connection h retained on f, $t_{min} + d(f,h) + \min(etimes(h)) \geq L$.

For any $t_f \in atimes(f), t_f > t_{min}$, obviously, $t_f + d(f,h) + etimes(h) \geq L$. Rewriting, $t_f + etimes(f) \geq L$.

For any $t_f \in atimes(f), t_f < t_{min}$, $t_f + \min(etimes(f)) < L$ and $t + \max(etimes(f)) < L$. This is true due to the choice of t_{min} assumed above. Thus, $t_f + etimes(f) < L$.

Thus, f satisfies the invariance condition.

The algorithm repeats on f_{dup} which ensures that the invariant eventually holds on each duplicate of f that is created. ∎

Since the invariant stated in Lemma 3.6.1 holds for each primary input, the final network is a L-path-disjoint network. Thus on completion of the second phase, the path-set of each first edge of any path of length $\geq L$ contains only paths of length $\geq L$. Hence, all such edges are set to constant *0* or *1* in the final phase of the single-pass algorithm (Figure 3.16).

Figure 3.17 illustrates the working of the algorithm on an example network. Each gate has unit delay and all primary inputs arrive at $t = 0$. Assume that the initial network (top of the figure) has a longest viable path of length 3. Hence an L-path-disjoint network for $L = 4$ is required. Following the first phase of the algorithm, the variables *atimes* and *etimes* are as follows:

$atimes(g1) = \{1\}$	$etimes(g1) = \{2,3,4\}$
$atimes(g2) = \{1,2\}$	$etimes(g2) = \{2,3\}$
$atimes(g3) = \{1,2,3\}$	$etimes(g3) = \{1,2\}$
$atimes(g4) = \{1,2,3,4\}$	$etimes(g4) = \{1\}$
$atimes(g5) = \{1,2,3,4,5\}$	$etimes(g5) = \{0\}$

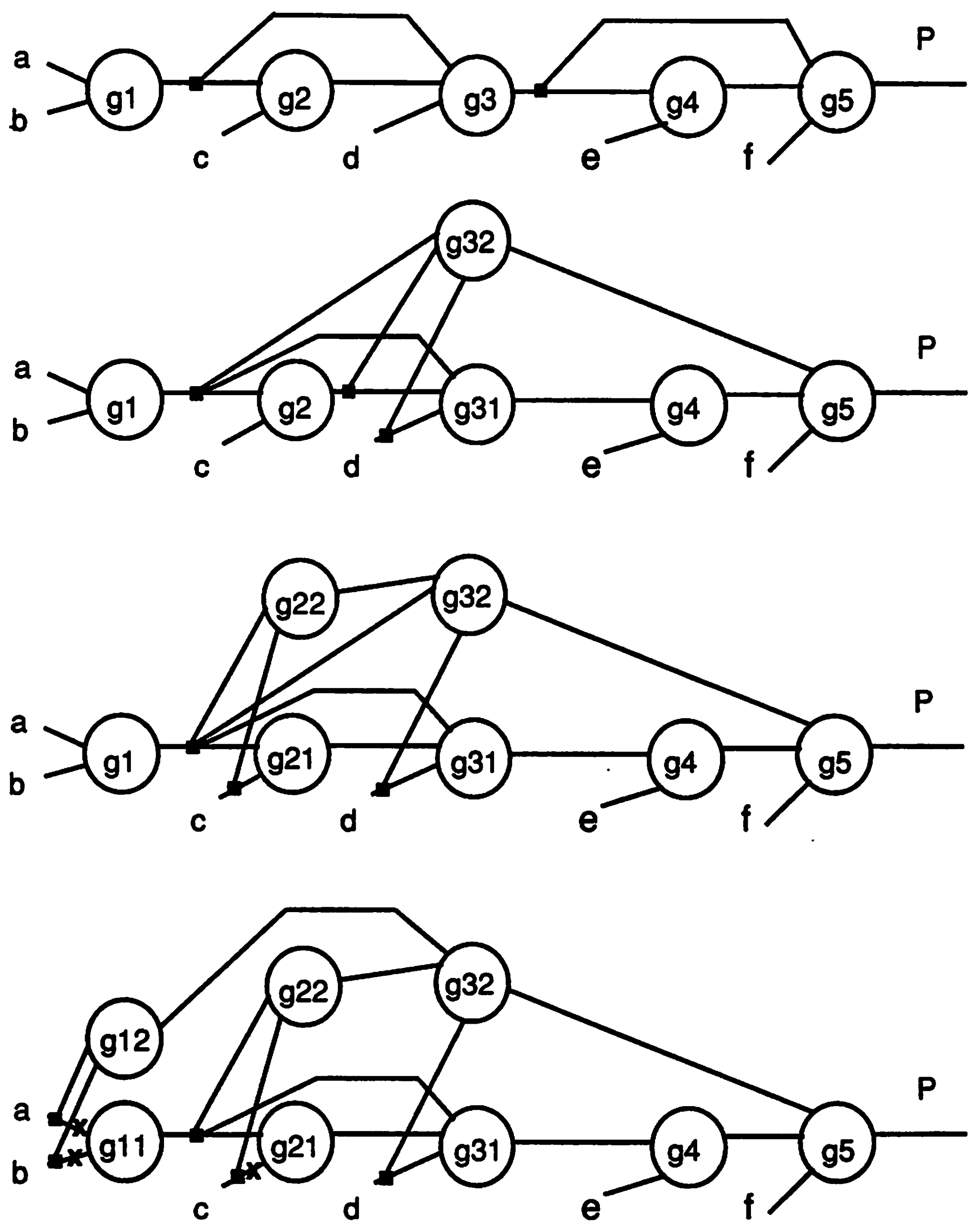


Figure 3.17: Example: Construction of an L-path-disjoint network

The L-path-disjoint network is obtained by a reverse topological traversal. Since both $g5$ and $g4$ have single fanout, no duplication is done on these gates. $g3$ is duplicated to obtain gates $g31$ and $g32$ (second network from the top of the figure). $g31$ is connected to $g4$ while $g32$ is connected to $g5$. Now, *etimes*($g31$) = $\{2\}$, and *etimes*($g32$) = $\{1\}$, and the invariant of Lemma 3.6.1 is now satisfied for $g31$ and $g32$. Similarly, on duplicating $g2$ and $g1$ as shown in the figure, an L-path-disjoint network for $L = 4$ is obtained (bottom of the figure). Any multiple stuck-fault on the inputs a and b of $g11$, and c of gate $g21$ can now be removed. By the theorems discussed earlier in this section, the computed delay and logical behavior of the resulting circuit remain unchanged.

It is also interesting to determine if a bound can be placed on the amount of duplication performed by the algorithm of this section. A loose upper bound is now stated.

Lemma 3.6.2 *Let W be the length of the (topologically) longest path, and V be the length of the longest viable path in network η. Then the number of gates in network η', resulting by the application of Figure 3.15, is no more than n times the number of gates in η, where n is the number of distinct path lengths between V and W.*

Proof Consider a gate f that is being processed by the algorithm of Figure 3.15. Consider the number of distinct values that can be generated by f_{atime} + *etimes*(f) for any $f_{atime} \in$ *atimes*(f). If the value of $f_{atime} + f_{etime} \leq V$ for any $f_{atime} \in$ *atimes*(f) and any $f_{etime} \in$ *etimes*(f), let this be represented by the value V. There is no loss in generality in doing so, because all paths of length $\leq V$ are considered viable, and are dealt with identically by the algorithm. There are n possible values between V and W which can be generated by $f_{atime} + f_{etime}$ for any $f_{atime} \in$ *atimes*(f) and any $f_{etime} \in$ *etimes*(f). For each distinct value, there is at most one duplication that is performed (in the worst case) to achieve the invariant of Lemma 3.6.1. There are n distinct path lengths of concern, so at most n duplicates of f are required. Hence, η' has no more than n times the number of gates in η. ■

The area bound stated above is very weak. For example, in the example of Figure 3.17, the lemma predicts 3 duplications per gate ($V = 3$ and $W = 5$ for the example), whereas only 3 gate duplications are required overall. A tighter bound on the area increase remains open.

3.6.1 Results using the single-pass algorithm

Results of an implementation of the single-pass algorithm for redundancy removal, guaranteeing the delay does not increase are shown in Tables 3.3 through 3.5. For each circuit the length of the longest sensitizable path, T, is first determined using an efficient timing analysis algorithm (*c.f.* Section 5.4). The smallest distinct path length $L > T$ is also known. Using the transformation of Figure 3.15, an L-path-disjoint circuit is derived from the original circuit. Finally, using Theorem 3.6.1 and 3.6.2, the first edge of every path of length $\geq L$ is set to a constant. The topologically longest path in the resulting circuit is now of length $< L$ (or $\leq T$). Finally, standard redundancy removal is used to derive an irredundant circuit that is no slower than the original circuit. The algorithm thus provides the same effect as the KMS algorithm in a single-pass following the timing analysis phase. The number of operations performed by the algorithm is linear in the number of connections of the original circuit and the number of distinct path lengths $\geq T$ in η.

The single-pass algorithm completes on all the circuits experimented with. The CPU time on a DEC 5000, not including the timing analysis phase and the final redundancy removal, is a few seconds for the largest example.

The area increase for the largest adder circuit *csa 64.8* is 22%. While this increase in area is substantial, a smaller penalty is achieved if each 8-bit block of the adder is first made irredundant using the KMS algorithm without an increase in the delay. In this case the penalty is 15%. A similar decrease in area (and delay, too) is also observed for all the other adder circuits, when each the KMS algorithm is performed independently on each block of the adder[6]. Table 3.4 shows the results of the single-pass algorithm on optimized MCNC circuits which are shown earlier in Table 3.2 for the KMS algorithm. It is observed that there is no area penalty incurred by the algorithm for the two largest circuits, *rot* and *des*. Note that straightforward redundancy removal on the initial *rot* circuit slows it down.

For the example *bw*, the final area by the single-pass algorithm is significantly smaller than that returned by the iterative KMS algorithm. However, the delay of the final circuit is larger for the single-pass algorithm. Although the single-pass algorithm achieves the delay given by the longest sensitizable path in the initial circuit, *i.e.* 14.0, the resulting circuit still has long false paths; the longest true path being of delay 10.0. The

[6]In performing the experiment, the arrival time of the carry-input to each block is set higher than the arrival times of the other inputs. This correctly captures the actual critical paths through the block when it is cascaded with the other blocks to form the complete adder.

Name	# Red.	Initial delay		Final delay		# Gates		
		Long.	True	RR	sKMS	Init.	RR	sKMS
ripple.16	0	31.0	31.0	31.0	31.0	139	139	139
lookahead.16	0	16.0	16.0	16.0	16.0	172	172	172
ripple.32	0	63.0	63.0	63.0	63.0	283	283	283
lookahead.32	0	28.0	28.0	28.0	28.0	361	361	361
csa 2.2	2	8.0	8.0	6.0	6.0	22	18	18
csa 2.2†	2	11.0	9.0	9.0	6.0	22	18	21
csa 4.2	4	14.0	12.0	10.0	10.0	44	36	43
csa 4.4	2	12.0	12.0	10.0	10.0	40	36	36
csa 8.4	4	22.0	20.0	18.0	18.0	80	72	87
csa 16.8	4	38.0	36.0	34.0	36.0	152	144	175
csa 8.2	8	26.0	16.0	18.0	14.0	88	72	83
csa 16.2	16	50.0	24.0	34.0	22.0	176	144	179
csa 16.4	8	42.0	24.0	34.0	22.0	160	144	167
csa 32.4	16	82.0	32.0	66.0	30.0	320	288	353
csa 32.8	8	74.0	40.0	66.0	38.0	304	288	335
csa 64.8	16	146.0	48.0	130.0	46.0	608	576	711
csa 8.2‡	8	26.0	16.0	18.0	13.0	88	72	84
csa 16.2‡	16	50.0	24.0	34.0	21.0	176	144	168
csa 16.4‡	8	42.0	24.0	34.0	21.0	160	144	172
csa 32.4‡	16	82.0	32.0	66.0	29.0	320	288	344
csa 32.8‡	8	74.0	40.0	66.0	37.0	304	288	348
csa 64.8‡	16	146.0	48.0	130.0	45.0	608	576	696

RR: redundancy removal

sKMS : single pass KMS algorithm

All primary input arrival time set at 0.0 delay units

†: Carry-input arrival time set at 5.0 delay units

‡: Single pass KMS algorithm performed independently on each block

Final circuit verified against initial circuit

Table 3.3: Single-pass algorithm versus redundancy removal on adders

Name	# Red.	Initial delay		Final delay		# Gates		
		Long.	True	RR	sKMS	Init.	RR	sKMS
5xp1	1	11.0	9.0	9.0	9.0	58	58	61
rot	37	19.0	17.0	19.0	17.0	437	424	423
des	17	15.0	13.0	13.0	13.0	2007	2000	1992
clip	4	8.0	8.0	7.0	7.0	64	61	61
duke2	2	9.0	9.0	9.0	9.0	190	190	190
f51m	39	18.0	17.0	16.0	16.0	173	139	138
misex2	6	7.0	7.0	7.0	7.0	94	90	90
rd73	10	11.0	11.0	11.0	11.0	94	83	83
sao2	9	12.0	12.0	12.0	12.0	126	119	119
z4ml	3	10.0	10.0	10.0	10.0	47	42	42
misex1	0	9.0	7.0	7.0	7.0	28	28	31
bw	0	20.0	14.0	14.0	14.0	85	85	99
z4ml	0	7.0	7.0	7.0	7.0	30	30	30

RR: redundancy removal

sKMS : single-pass KMS algorithm

All primary input arrival time set at 0.0 delay units

Final circuit verified against initial circuit

Table 3.4: Single-pass algorithm versus redundancy removal on MCNC circuits

Name	# Red.	Initial delay		Final delay		# Gates		
		Long.	True	RR	sKMS	Init.	RR	sKMS
C1908	3	23.0	21.0	23.0	21.0	325	322	343
C1908†	29	28.0	26.0	28.0	26.0	324	319	341
C6288	2	120.0	119.0	120.0	119.0	2353	2350	2363
s641	0	19.0	18.0	18.0	18.0	122	122	126
s713	35	27.0	24.0	19.0	18.0	148	122	126
s1238	67	20.0	19.0	17.0	17.0	429	392	396
s9234	224	29.0	28.0	24.0	24.0	1735	1526	1526
s15850	310	40.0	39.0	37.0	37.0	3268	2954	3013

RR: redundancy removal

sKMS : single-pass KMS algorithm

All primary input arrival time set at 0.0 delay units

†: optimized using the standard script in SIS

Final circuit verified against initial circuit

Table 3.5: Single-pass algorithm versus redundancy removal on ISCAS circuits

iterative algorithm picks this up and iterates until a circuit with a longest path of delay 10.0 is achieved. It is interesting to note that on repeating the single-pass algorithm the same result as the iterative KMS algorithm is obtained. The iterative and the single-pass KMS algorithms yield the same result in terms of delay on all the other circuits. On some circuits there is some insignificant variation in the final number of gates.

Table 3.5 shows the result on combinational and iscas ISCAS circuits. Inverters and buffers are removed from the initial circuit before application of timing analysis and redundancy removal.

3.6.2 Impact of transformations on fanout

In this section the impact of the transformations of the algorithm on the delay due to the possible fanout increase in the resulting circuit is evaluated. In particular, the principal concern is with the duplication of sub-circuits.

In typical static delay models the delay through a gate is a function of the fanin of the gate, the individual delay of the gate, and the fanout of the gate. The delay of a path is a function of the sum of these gate delays over the path. The algorithm presented in Figure 3.4 (or Figure 3.15 for the single-pass algorithm) may duplicate a sub-circuit η_n. For each path P' in the resulting circuit there is a corresponding path P in the original. Compare the delay of P' relative to the delay of P. For each gate g_i' in P' the individual gate delay and fanin are precisely the same for g_i' in P' and the corresponding gate g_i in P. Based on these considerations alone, the delay along P', being a function of sum of the delays of each g_i' in P', is the same as the delay along P which is a function of sum of the delays of each g_i in P.

There is one additional complication; the delay along an arbitrary path Q is also a function of the fanout of each gate g in Q. After duplication, any gate g which formerly fed a gate in η_n now feeds a gate h in η_n and a gate h' in the duplicated network η_n'. Thus, the fanout of a gate may increase. A formal upper-bound on the area or fanout increase remains an open problem, though empirical evidence indicates that this increase is never more than the size of the circuit itself.

Intuitively, consider how the fanout of any gate in a single-output circuit changes during the algorithm. When a gate is duplicated, the fanouts on each fanin of the gate increase. Let g_{high} denote a gate that is duplicated. The original fanout connections to g_{high} are now spread over g_{high} and its duplicates; this leads to a decrease in the fanout of each of these gates. However, the fanout on the gates feeding g_{high} and its duplicates is increased. Now, if the fanout on a gate g increases due to duplication of some of its fanout gates, g itself is very likely to get duplicated. This in turn tends to reduce the fanout on each duplicate gate of g, passing the fanout increase backwards to the fanin gates of g. This effect of duplication ripples backwards until it reaches the primary inputs. Thus, it is likely that the fanout on internal gates is low and often may be even lower than in the original circuit.

Observe that on *all* the benchmark examples listed in Sections 3.5 and 3.6.1, the average increase in area (relative to the initial circuit) for examples with false paths is less than 5%. The largest area increase is less than 15%. The area is often decreased in circuits with large number of redundancies. These results indicate at most a small increase on the fanout of gates. Small increases in fanout typically may not impact the delay through the gate. However, if a fanout increase does increase the delay through a gate, one of the following techniques may be applied to retain the original delay estimate through paths in the network.

Assume a CMOS technology and the use of custom design, standard cells or gate arrays. Any increase in fanout due to running the algorithm is addressed by transistor sizing in custom designs, and by cell selection in standard cell or gate-array designs. In general, if a gate g_i in P feeds k gates in η_n, then a suitable gate g_i' in P' is chosen such that g_i' can drive a fanout of $2k$ gates in η_n' at the same speed that g_i drives a fanout of k. An inspection of a typical standard cell library, such as the AT&T 1.25 μ CMOS Library, shows that "high" and "super" powered versions of such gates are available that will accomplish this even for values of k up to 30. If a transistor sizing program such as TILOS [44] is used in a custom design methodology, then an even wider variety of techniques may be employed to drive the larger fanout of g_i'. In the 2-bit carry skip adder, after removing redundancies there is an increase in fanout of at most one for any gate, and no modification of the circuit is required to accommodate the higher fanout. It would be interesting to obtain a practical circuit for which these techniques are insufficient, if such a circuit exists.

3.7 Applications

The core idea of the KMS algorithm is a transformation that results in a circuit with the longest path sensitizable yet no slower than the given initial circuit. This transformation has several applications, which are touched upon in the following sub-sections.

3.7.1 Multiple stuck-fault testability

An important clarification with respect to circuits that may exhibit an increase in circuit delay in the presence of a stuck-fault is the choice of the fault model. As mentioned in the previous chapter, redundant faults that impact the timing behavior may be multiple-stuck faults. Thus, even though a circuit may be 100% single stuck-fault testable, there maybe a redundant multiple stuck-fault that impacts the timing behavior. This requires a circuit to actually be made fully multiple stuck-fault testable to avoid a speed-test. A minor modification to the algorithm in Figure 3.4 allows a 100% multiple stuck-fault testable circuit to be derived from an irredundant circuit with no increase in delay. This is stated formally as:

Theorem 3.7.1 *Performing multiple stuck-fault redundancy removal on an irredundant circuit obtained by the* KMS *algorithm in Figure 3.4 on an initially redundant circuit does not increase the delay of the circuit.*

Proof In Theorems 3.3.1 and 3.3.2 it is proved that the KMS algorithm does not increase the delay of the circuit, while making the longest path sensitizable. Using Theorem 3.1.1, further multiple stuck-fault redundancy removal cannot slow down this circuit. ∎

Of course, this requires an algorithm for multiple-stuck fault redundancy removal.

3.7.2 Generalized bypass transformation

Recently, it is shown in [78] how the approach used to speed up a ripple-carry adder to a carry-skip adder may be generalized to improve the delay in arbitrary circuits. This *generalized bypass transform* (considered later in Section 4.9) speeds up a circuit by converting long and sensitizable paths into false paths by using an extra AND gate and MUX. Analogous to the case of the carry-skip adder, these enhanced circuits have redundancy introduced into them by the bypass transform. Additionally each of these statically redundant faults is τ-irredundant. While the KMS algorithm can be applied after the delay optimization step is completed, [78] shows how it can be interleaved with the performance enhancement process of the bypass transformation. No redundancy is introduced by this modified procedure.

3.7.3 Don't care conditions

Don't care conditions have recently become an essential factor in the optimization of Boolean networks [16]. The don't care conditions for a circuit can be easily incorporated into both the timing analysis and redundancy determination phases of the KMS algorithm. This merely requires the additional constraint of ensuring that any primary input vector that satisfies the property being determined on a path, either the sensitizability or testability, lies outside the don't care set.

3.7.4 Sequential circuits

This algorithm may be generalized to sequential circuits by extracting the combinational portion from the sequential circuit since the cycle time of a synchronous sequential circuit is determined by the delay of the combinational portion between latches. In these circuits, don't cares are generated for values that cannot appear on the memory elements and are used in the timing analysis and redundancy removal phases of the algorithm, similar to the discussion in Section 3.7.3.

3.8 Conclusions

In this chapter the relationship between the performance and redundancy of a circuit has been explored. Prior experience has shown that performance optimizations can introduce redundancies into circuit designs. In the vast majority of cases the longest path in the circuit is sensitizable in spite of these redundancies; hence the redundancies may be removed in any manner without affecting the speed of the circuit. However, the carry-skip adders are constructed in such a way that the longest path in the circuit is not sensitizable. Furthermore, for these circuits a straightforward redundancy removal approach will result in a slower, though irredundant, circuit. Thus, there is a question in general whether redundancies are in fact necessary for performance. It is shown that redundancies are not necessary for performance, and an algorithm is provided that removes redundancies from any circuit, while guaranteeing to retain or improve its speed. Applying this algorithm to the carry-skip adder produces a novel irredundant implementation. The algorithm has been applied on several other circuits as well. The area increase in most of the examples that exhibit long false paths is small. In a few such examples, the area remains unchanged or even decreases. However, the existence of a small upper-bound on the area increase of any circuit that undergoes the transformations described in this chapter remains open.

The first version of the algorithm iteratively performs redundancy removal on a path-by-path basis. Since this approach is infeasible on large circuits, a single-pass algorithm is developed. This efficient algorithm is based on the relation of circuit structure to the interaction between redundancy in delay. The construction of L-path-disjoint circuits that are used in this algorithm appears to have further ramifications, for example to timing analysis and synthesis for robust delay-fault testability. These aspects are under investigation and preliminary results are reported in [92, 93]. This approach using an L-path-disjoint network belongs to the class of techniques that transform a given circuit to a functionally equivalent circuit, whose structure realizes some significant or important relationship between the delay, testability, and area. By performing this transformation, the solution of a seemingly difficult or expensive problem, such as the iterative KMS algorithm, is converted to a simpler problem that is faster to solve.

As previously mentioned, the application of any algorithm aimed directly at the identification and removal of stuck-faults, such as [13, 100], may diminish the speed of circuits such as carry-skip adder. It is also worth noting that techniques for removing untestable path delay-faults, such as [91], are also likely to increase the delay of such circuits since they radically change their structures to bring them within the stringent condition of robust path delay-fault testability. It would be interesting to discover if the techniques described here could be generalized to the removal of path delay-fault redundancies without degrading circuit performance.

Chapter 4

Testability Effects of Performance Optimizations

Timing optimization procedures resynthesize a given circuit to meet the timing requirements with minimal area increase [60, 105, 7, 33]. The effect of the transformations used by these algorithms on the testability of the circuit has been mostly ignored and is not well understood. In Chapters 2 and 3, a high-performance circuit with poor testability is shown to be unreliable both in its logical and temporal behavior. It is proved that redundancy is unnecessary to reduce delay. Motivated by these results and the need for a further understanding of the relationship between performance optimization and testability, this chapter attempts to resolve the question: "Can performance optimization be done without introducing redundancy?" The impact of performance optimization techniques on testability under different fault models is considered. Section 4.1 is an introduction to the fault models to be considered. Background information on two multilevel operations that are needed in the discussion is provided in Section 4.2. The testability results described in this chapter will be derived for the timing optimization procedure that is described in [105], and implemented in the SIS logic synthesis system [102]. An overview of this approach is provided in Section 4.3. This approach is representative of most timing optimizations[1].

First, the possibility of maintaining single-fault testability during timing optimization is considered in Section 4.4. After identifying the conditions that cause redundancy, an attempt is made to maintain the testability of the circuit as invariant. It is demonstrated

[1]An exception is the recent work of [109], which involves the Boolean operation of collapsing.

that while a redundant connection introduced into an irredundant circuit is easy to identify and remove, its removal may lead to further redundancy (Section 4.5). These secondary redundancies are difficult to identify. This motivates an attempt to find a circuit testability property that may be easily maintained during the various timing optimization steps. Section 4.6 briefly states why multi-fault testability is not a candidate. In Section 4.7, robust path delay-fault testable circuits, constructed using known synthesis procedures, are proven to meet this criterion. Since not all circuits may have robust delay-fault testable versions, an alternative *testability-timing-invariant* is proposed in Section 4.8 and shown to have the same invariant properties as robust delay-fault testable circuits. Section 4.9 briefly considers the effect on testability of recent techniques of performance optimizations, namely the *generalized select transformation* [11] and the *generalized bypass transform* [78]. Section 4.10 concludes the chapter by describing two relevant problems that remain unsolved.

4.1 Fault models

When studying the impact of each timing optimization step on the testability of a circuit under a chosen fault model, it is assumed that the circuit before application of each optimization step is fully testable, *i.e.* for each fault site of the chosen fault model, there exists at least one test vector that tests for this fault. Before considering the change in testability during timing optimization, a brief review of the conditions and synthesis procedures that result in fully testable circuits is provided in this section.

4.1.1 Stuck-faults

Of all the known fault models, single stuck-faults (single-faults) have received the most attention over the past three decades [19]. However, it is only relatively recent that the precise relationship between the single-fault testability and optimization of a combinational circuit has been established. The details of this are in [6] and are not considered here. It is mentioned that 100% single-fault testability is equivalent to a first-order criterion for area minimality, namely that a circuit is *prime* and *irredundant* [6]. Briefly stated, if a network is prime and irredundant, no connection can be set to a constant value (*0* or *1*) without changing the logical behavior of the circuit.

Synthesis techniques for 100% single-fault testable circuits use two approaches: ATPG based redundancy removal [100, 56], and, don't care based node minimization [6, 5, 97]. In the second approach, the don't care sets are often approximated (or filtered) to reduce the computation effort [95, 82], thus loosing any guarantee of full testability. While both approaches are effective on a large collection of benchmark circuits [5, 97], the former has an advantage in this regard [97]. Due to the effectiveness of these two approaches, not much focus has been placed on maintaining the invariance of single-fault testability in Boolean networks. The only previous work is reported in [58] where it is demonstrated that single fault testability may not be retained under algebraic substitution. In Section 4.4, the question of invariance of single-fault testability under various circuit transformations is more fully explored.

Like single-fault testing, multiple stuck-fault (multi-fault) testability of circuits has been long studied [99, 61]. However, only recent results have made the synthesis of 100% multi-fault testable circuits viable. While necessary conditions for 100% multi-fault testability are still unknown, [51] relates 100% multi-fault testability to a second order criterion for area optimality of a circuit, termed *simultaneously prime* and *irredundant*. Briefly stated, if a network is simultaneously prime and irredundant, no set of connections can be set to constant values without changing the logical behavior of the circuit. [61] proves that a fully single-fault testable single output two-level circuit implies that the circuit is also 100% multi-fault testable. Sufficient conditions are given in [51] under which a two-level multi-fault testable circuit may be transformed to a multilevel circuit that retains this testability criterion. These conditions allow the unrestricted use of algebraic factorization techniques on these circuits. However, Boolean operations such as simplification [18] are shown not to preserve multi-fault testability and cannot be used. This implies a possible reduction in the area optimality of an implementation when using this testability criterion.

4.1.2 Delay-faults

Up to now, the concern with regard to the testing of an IC has only been with stuck-faults that impact the logical behavior of circuits. However, a circuit is deemed to be performing correctly only if both its logical and temporal behavior meet the design specifications. Due to the increasing need for reliability in high-performance VLSI circuits, static testing which only ensures logical correctness of the circuit is becoming insufficient. In particular, defects and random variations in process parameters often cause delays to fall outside the specifications. Static testing may not detect such defects since there may be no impact on the logical (static) behavior of the circuit. However, since most circuits operate in a clocked environment it is essential that the final logic value on the circuit be asserted before the clock arrives. This need for ensuring the temporal (dynamic) correctness of circuits, given delay specifications, has led to the development of delay-testing for chips.

Definition 4.1.1 *A* **delay-fault** *occurs when a propagation delay falls outside the specified limits.*

Two models for delay-faults have been proposed to model delay defects on gates or along paths. They are defined as follows:

Definition 4.1.2 *A* **gate delay-fault** *is said to occur when any of the delays on the gate inputs or output falls outside its specified limits. This defect is modeled by a single (lumped) delay-fault at the gate.*

Definition 4.1.3 *A* **path delay-fault** *is said to occur when the delay along the path falls outside its specified limits.*

Two notable deficiencies of the gate delay-fault are (1) the testability depends on the size of the delay-fault, and (2) all defects are not captured by the model [106]. The first problem arises due to reconvergence of paths where even a gate operating with a delay-fault may not cause any change to the timing of the circuit. The second problem arises if statistical timing rather than worst case timing is used as the basis for setting the clock period of a design. In such a situation, each gate delay may fall within specifications but some path nevertheless may be slower than the specifications. Since a delay of a path is the cumulative delay of gates along the path, even though a delay-fault exists for a path, no (gate) delay-fault may exist on any of the gates along the path. For these reasons, the path delay-fault model is considered more comprehensive. The testing protocol used for path delay-faults applies to gate delay-faults only if the size of the delay-fault is large [106]. However, it applies to all sizes of delay-faults along paths. Another difference is that the gate delay-fault model assumes only a single occurrence of a delay-fault, thus modeling isolated failures, whereas the path delay-fault models multiple delay-faults, due to more distributed failures[2].

Since delay-faults model defects that impact the delay on gates or paths, testing for them requires the application of a pair of vectors such that a transition is propagated

[2]Under the condition of robustness, however, a gate delay-fault test remains valid even in the presence of multiple gate delay-faults.

along the gate or path under test. The test for a path delay-fault P_f on a path P (assume its delay is specified to be P_t), involves the application of vectors $< v_0, v_1 >$ to the primary inputs of the circuit. It is assumed that there are no stuck-faults in the circuit under test. After applying v_0 and allowing the circuit to stabilize, v_1 is applied. This vector must cause a transition along P. The circuit output of P is observed at time P_t after v_1 is applied. If the transition is observed, no delay-fault along P is said to exist. If the transition is not observed, P is said to have a delay-fault. There are two important observations.

First, the circuit is clocked *at-speed*. This means that unlike static testing, the circuit operates at the normal clocking speed. This requires test vectors to be loaded and unloaded into the memory elements under normal operating conditions, which implies substantially more testing overhead than static testing [4]. In this thesis, the presence of enhanced scan structures that facilitate delay-fault testing is assumed [106].

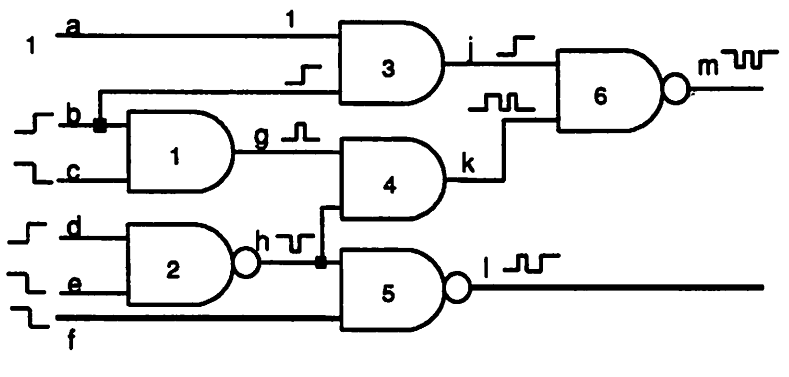
Second, while the test pair $< v_0, v_1 >$ propagates some transition along P, it may also propagate transitions along other paths. Due to the occurrence of hazards, even if P has a delay-fault, the output of P may have a spurious transition (due to a hazard) *at the time the output is observed*. Thus, the test may be invalidated in the presence of hazards caused by varying delays along other paths in the circuit. This phenomenon is illustrated in Figure 4.1.

In circuit (1) of the figure, assume that the path shown in bold is to be tested. Let the vectors $< v_0, v_1 >$ be applied where $v_0 = ab'cd'ef$ and $v_1 = abc'de'f'$. The vector pair causes a falling transition on input f of the path under test (in addition to transitions on b, c, d, and e). A transition from *0* to *1* is expected at the output l. Assume that some delay assignment within gate 2 causes a hazard (or glitch) on its output that propagates to gate 5 as shown. The output l may be as shown in the figure. Even if a delay-fault exists along the bold path, the correct value of *1* may be available at l, at the moment it is observed. This invalidates the test $< v_0, v_1 >$.

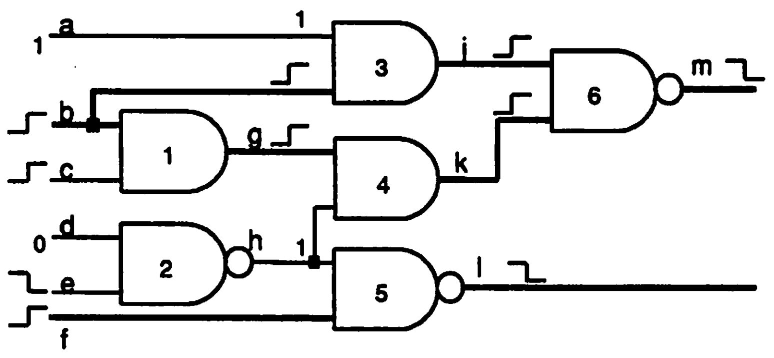
This invalidation of a path delay-fault test is avoided by imposing the condition of robustness, described in the next section.

4.1.3 Robust path delay-fault testing

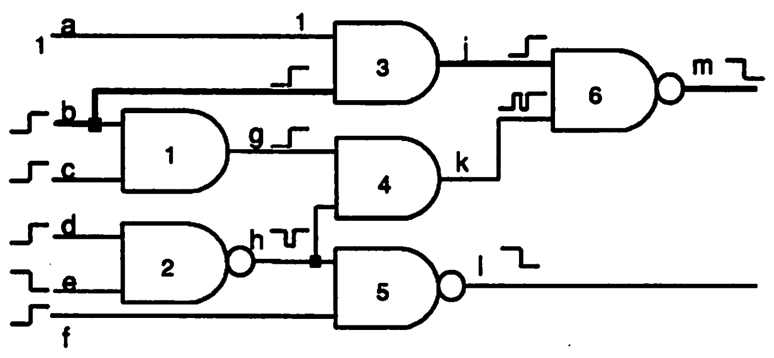
A path is robust delay-fault testable if there exists a test for the path delay-fault associated with the given path that is valid under arbitrary delays along other paths. A



(1)



(2)



(3)

Figure 4.1: Examples: Delay-fault testing

circuit is called 100% robust path delay-fault testable if every path from a primary input to a primary output in the circuit is robust delay-fault testable. Over the past few years, several definitions have been proposed for robust delay-fault testability [106, 96, 36, 88]. The classification may be described along three axes:

1. single or multiple path propagation,

2. presence or absence of hazards,

3. single or multiple input change for test vector pairs.

Of these, the last parameter is not considered here since every delay-fault that is tested by multiple input changes can be tested by a single input change. See [106] for a complete explanation of this. The main reason for choosing multiple input change test vectors is to reduce the size of the test set; however, not much work has been reported on this aspect.

The first two parameters are illustrated through the example of Figure 4.1. Consider the bold paths starting from input b in circuit (2) of the figure. The given transitions at the inputs (applied using the appropriate vector pair) are robust tests for both paths. In this case, the test is called a multiple path propagation delay-test. The same vectors, however, are a single-path test for the delay-fault on the bold path from input f.

The circuit (3) in the figure illustrates a robust path delay-fault test even in the presence of hazards. The hazard shown at k may be caused under some combination of delays in gates 2 and 4. However, the output m transits from *1* to *0* only after the input along the path being tested arrives. Thus, if the bold path has a delay-fault, the output is delayed and a transition is not observed at the required time. The test is said to be robust in the presence of hazards.

In [36] necessary and sufficient conditions for single path propagation hazard-free robust path delay-fault testability (RPDFT) are given. More recently, a complete classification of robust path delay-faults has been provided [88]. While all the results in this chapter are stated for the case of single path propagation hazard-free robust path delay-fault testability [36], all results can be generalized to the conditions stated for *general* (multiple path propagation with hazards) robust path delay-faults. Necessary and sufficient conditions for hazard-free robust path delay-fault testing of a path P are provided in [36] using an equivalent two-level expression of a given multilevel circuit, called the ENF expression. Instead of resorting to the ENF of a circuit, the conditions are stated here directly in terms of the structure of the multilevel circuit. The proof of this condition is not provided and is detailed in [36].

Definition 4.1.4 (RPDFT)*: A path $P = \{f_0, f_1, ..., f_m\}$ is said to be* **single path robust delay-fault testable without hazards** *by the vector pair $< v_1, v_2 >$ if at each node f_i, $f_i(v_1) \neq f_i(v_2)$, and for each $g_j \in S(f_i, P)$:*

1. *$g_j(v_1) = g_j(v_2) = I(f_i)$; and*
2. *there is no transition on g_j.*

The vector v_2 is assumed to be applied after v_1, delayed by an amount greater than the delay of the circuit.

Besides requiring each side input to the single path under test to be at a non-controlling value, the definition requires that no transition (or hazard) appear on any of these side-inputs as well. Thus, if the vector pair $< v_0, v_1 >$ tests a given path for a rising (*0* to *1*) transition, $< v_1, v_0 >$ tests the same path for the falling transition.

In contrast to the above definition, the most general form of robust delay-fault testing requires less stringent conditions on the side inputs:

Definition 4.1.5 (GRPDFT)*: A path $P = \{f_0, f_1, ..., f_m\}$ is said to be* **robust delay-fault testable** *for the rising (falling) transition at f_m by the vector pair $< v_1, v_2 >$ if at each node f_i, $f_i(v_1) \neq f_i(v_2)$ yields the desired transition being tested, and for each $g_j \in S(f_i, P)$:*

1. *$g_j(v_2) = I(f_i)$; and*
2. *if $f_{i-1}(v_1) = I(f_i)$, then there is no transition on g_j.*

The vector v_2 is assumed to be applied after v_1, delayed by an amount greater than the delay of the circuit.

This general robust test (GRPDFT) may involve multiple path propagation with hazards. This definition does not specify the value on side-inputs for v_1 when a transition from a controlling to a non-controlling value is being tested along the path. This is because even in the presence of hazards on side inputs, a delay on this input transition will appear as a delay on the transition at the output of the gate. However, for the opposite transition (from non-controlling to controlling value) on the gate input, no transitions are allowed on the side-inputs to this gate along the path. This difference in definitions between the two categories of robust delay-faults appears minor. Nonetheless, circuits are known for which no robust test exists using the first definition, while a robust test does exist in the presence of hazards [88]. One advantage of using delay-tests with hazards is a possible reduction in the size of the test set. This is because more paths can potentially be tested in the case when hazards are allowed compared to the hazard-free case. This issue is not addressed in this thesis.

[36] describes an approach to realize a 100% RPDFT two-level circuit by modifying one step of a standard two-level logic minimization program [15]. Fully testable multilevel circuits are then obtained by using algebraic factorization. Similar to the case of multi-fault testability the RPDFT test set remains invariant under algebraic factorization. Techniques for creating 100% robust delay-fault testable implementations of common data-path designs, such as adders and parity trees, which cannot be synthesized using two-level logic due to the size of the circuit, are described in [37].

It is interesting to examine the relationship among the various fault models detailed in this chapter. A multi-fault testable circuit implies the circuit is single-fault testable. A robust path delay-fault testable circuit also implies single-fault testability. This is because the ability of propagating a falling (rising) transition at some connection c using the vector pair $< v_0, v_1 >$ along a path implies that the stuck-*1* (stuck-*0*) fault on c is detected [63]. It is unknown whether full RPDFT testability implies full multi-fault testability. This result is conjectured in [36], but the proof provided there is incorrect.

4.2 Algebraic resynthesis and collapsing

Two of the principal operations in multilevel logic optimization are algebraic factorization and decomposition (together termed *algebraic resynthesis*) and collapsing [16]. Briefly explained, a factored form is a parenthesized logic expression, *e.g.* $(a + b(c + d) + (d + e)(a + c'))$. Defined recursively, a factored form is either a product of single literals and factored forms, or a sum of single literals and factored forms. Decomposition of a function is the process of re-expressing a single function as a collection of new functions. For example, $F = abc + abd + a'c'd' + b'c'd'$ may be decomposed to $F = XY + X'Y', X = ab, Y = c + d$. Typically, the factored form is used to guide the decomposition of a function [16]. Collapsing of a function G into F is the process of re-expressing F without including G. The result of collapsing is an expression for F that uses its original inputs and the inputs of G.

Results pertaining to the effects of algebraic resynthesis and collapsing on the network structure are required in proving testability results in subsequent sections. Let n be a two-level (AND-OR gates) complex node in a multilevel network η. Let algebraic factorization and decomposition replace n by a sub-network η_n to yield a network η'. Let P_n be the set of paths in η passing through an AND gate and the OR gate in n. Let P_{η_n} be the set of paths passing through gates in η_n in η'. Obviously, the logical functionality of η and η' are identical. Moreover, η' is *syntactically equivalent* to η [51], meaning:

- for each path $p \in P_n$, there is a corresponding path $q \in P_{\eta_n}$, and
- each path $q \in P_{\eta_n}$ may correspond to more than one path $p \in P_n$.

This means that for each path through an AND gate and the OR gate in n there is a path through some gates in η_n. This many-to-one mapping of paths in η to paths in η' is referred to as a *merging* of paths in the sequel.

The effect of collapsing on the paths in a network is analogous to algebraic factorization. Let η_n be a sub-network in a multilevel network η. Let collapsing replace η_n by a single complex node n to yield a network η'. Let P_{η_n} be the set of paths in η passing through gates in η_n. Let P_n be the set of paths in η' passing through an AND gate and the OR gate in n. Obviously, the logical functionality of η and η' are identical. Moreover, η' is *syntactically equivalent* to η, which implies:

- for each path $q \in P_{\eta_n}$, there are one or more corresponding paths $p \in P_n$.

This means that for each path through η_n there are one or more corresponding paths through an AND gate and the OR gate in n. This one-to-many mapping of paths in η to paths in η' is referred to as a *splitting* of paths.

4.3 Timing optimization procedure

The testability results described in the subsequent part of this chapter are derived for the timing optimization procedure described in [105], which as mentioned previously, is representative of most timing optimizations using restructuring.

Timing optimization is viewed as a three-phase process [16]. In the first phase. global restructuring is performed on the circuit to reduce the maximum level or the longest path in the circuit. Typically, this is accomplished in a technology independent fashion. For example, changing from a ripple-carry-adder to a carry-look-ahead adder or something in between is accomplished in this phase. The second phase is considerably more dependent on the target technology and is referred to as *technology mapping*. Techniques for reducing the delay of mapped circuits, with minimal area increases, have been studied recently in [108]. The final phase is to speed up the circuit during the physical design process. Transistor sizing [44] or timing driven placement of modules [85] are examples of such optimizations. In this phase, when an actual design exists, a more accurate timing analyzer is used to fine-tune the circuit parameters. In this chapter, only the first phase of timing optimization is considered, *viz.* technology independent logic resynthesis. It is shown elsewhere, [81], that the most common approach to technology mapping, namely *tree-mapping* [35], preserves testability. The other technology dependent optimizations of buffering [104] and transistor sizing [44] do not change the testability of a design. This result is obvious since these operations neither change the function nor the structure of the circuit.

A brief argument is provided that demonstrates that the testability results described here apply to most other approaches to achieving speed-up in networks. Two transformations that are not covered by the resynthesis procedure being discussed here are studied at the end of the chapter. Early attempts at speed-up make local changes only in the topology. [53, 32] reduce the delay by adding buffers and decomposing an *existing gate* into gates containing early and late arriving signals, with the latter being placed closer to the output. SOCRATES [7] uses a rule based system to improve the timing by local transformations. A significant drawback of these techniques is that they may not achieve the global restructuring desired. However, the strength of the approaches is that features of the library and technology being used are fully exploited. [105] states that both these approaches are subsumed by the resynthesis technique described herein.

The problem of restructuring the logic is solved using a global view. The approach in [105] is similar to the *circuit re-synthesis* step of the Yorktown Silicon Compiler [33]. In both approaches, a sub-network is defined and a critical section to be transformed is identified. The significant contrasts are that the former algorithm focuses only on logic resynthesis and operates on a technology independent representation of the circuit. The algorithms in [33] combine device sizing with logic manipulation and are somewhat specialized to domino CMOS designs. [33] uses transformations only between neighboring gates, whereas in [105] a parameter is available to control the extent of the transformations.

The algorithm uses a timing driven decomposition of the network into 2-input gates. This is important since the manner in which a complex gate is implemented changes its delay characteristics. Various models are used for computing delays. One is a fast technology mapping [35] of the two input gates into a standard cell library. This provides more accuracy to the timing estimates. An algorithm, based on timing constraints, for decomposing a complex function into two input gates is also used. This is done recursively from the bottom up, so that at each stage the input arrival times are fairly accurate, and thus subsequent decomposition of the upper nodes can be based on these updated arrival times.

The resynthesis algorithm takes as input a network of 2-input NAND gates and inverters. Timing constraints are specified as the *arrival times* at the primary inputs and *required times* at the primary outputs. The algorithm manipulates the network until the timing constraints are satisfied or no further decrease in the delay is possible. The output of the algorithm is also in terms of 2-input NAND gates and inverters.

An outline of the timing optimization algorithm is given in Figure 4.2. Details of each step of the algorithm and its parameters are in [105]. Given the primary input arrival times, the arrival times for each of the signals is computed. Using the required times at the outputs, the required times for all signals are computed. The *slack* at a node s, is defined as $R_s - A_s$ where A_s is its arrival time and R_s its required time. An *ϵ-network* is defined as the sub-network containing all signals with slack within ϵ of the most negative slack. The procedure *node_cutset(ϵ-network)* determines a cut-set [46] of nodes, each of which must be sped up in order to realize some global delay reduction. The procedure *partial-collapse(n, distance)* collapses all the nodes in the transitive fanin of the node n which are themselves in the ϵ-network and at most distance *distance* from n. All such internal nodes that fanout elsewhere will be duplicated in this process. The *speedup_node(n)* procedure performs timing driven decomposition on the complex node n. The overall strategy attempts to place late-arriving signals closer to the output. Thus, in the first phase, divisors (or common sub-expressions [17]) of the node with early arriving inputs are factored out. After all such divisors are exhausted, a timing based decomposition into a 2-input NAND gate tree structure is done. This routine again places late-arriving signals closer to the output and also ensures that the final timing-optimized network is composed of 2-input gates only. A final step of area recovery may be performed to merge identical gates.

```
/* $\eta$ is the Boolean network to be speeded up.
$distance$ is the number of levels up to which the critical fanins are
collapsed.  */
speed_up($\eta$, $distance$) {
   do {
      delay_trace($\eta$).
      $\epsilon$-$network$ = select_critical_network($\eta$, $distance$).
      $node\_list$ = node_cutset($\epsilon$-$network$).
      Foreach node $n \in node\_list$ {
         partial_collapse($n$, $distance$).
      }
      Foreach node $n \in$ node_list {
         speedup_node($n$).
      }
   } while (delay($\eta$) decreases && timing constraints not satisfied).
}
```

Figure 4.2: Outline of the timing optimization process

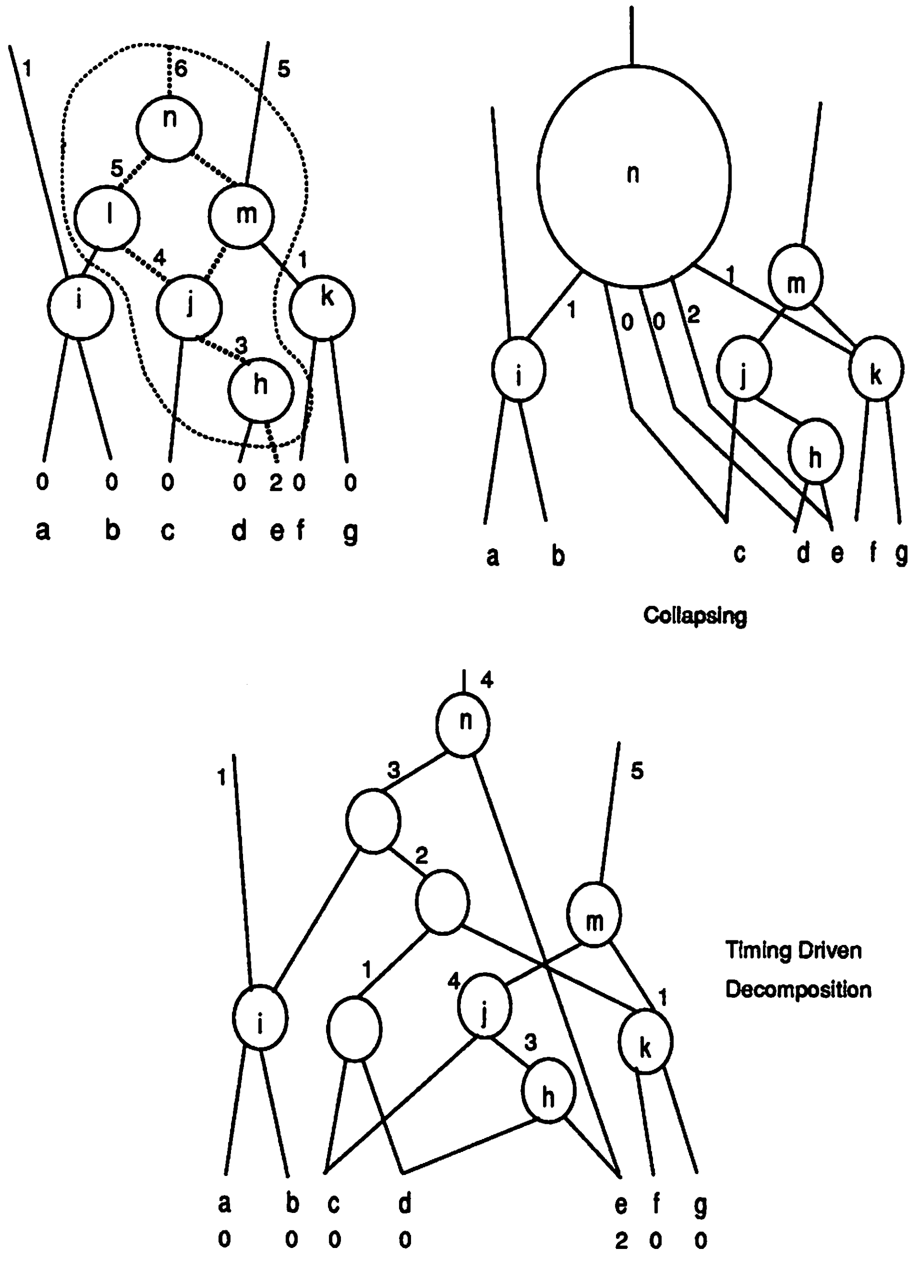


Figure 4.3: Example: Timing optimization process

Figure 4.3 illustrates the procedure on an example circuit. Primary input e is a late-arriving signal which causes output n to be available after 6 units of time. Since this signal is critical the timing optimization procedure attempts to reduce the delay to n. In the first phase the ϵ-critical network (encircled in the first schematic of the figure) is identified. In the second step, these nodes are collapsed to yield a complex node at n. Note that nodes h, j and m are duplicated since they are required at other places in the network. Finally, the large node n is decomposed based on arrival times of its inputs. Notice that e passes through fewer gates since it arrives after the other signals. The final optimized delay for n is 4 units. While a simple unit-delay model is used in this example, several different delay models can be employed to improve the correspondence of the delay estimate on the 2-input NAND gate network with its final mapped implementation [105].

The five principal steps in this timing resynthesis system and most others may be summarized as being one or more of the following steps, executed iteratively and in any order:

- Selection of a subset of gates (*critical sub-network*), to resynthesize.
- Duplication of gates in the critical sub-network that have fanout.
- Collapsing of the gates in the critical sub-network.
- Algebraic resynthesis (factoring and decomposition) of the collapsed node.
- Area reclamation by merging of identical gates.

4.4 Single stuck-fault testability effects

Table 4.1 shows a profile of the area and delay of a typical circuit as the algorithm of [105] progresses. This illustrates the ability of the algorithm to trade off area for speed. The first experiment with the example performs timing optimization on an area optimized irredundant circuit. The delay and area on each pass of the algorithm are given in the first two columns. The third column indicates the number of redundancies in the circuit. In this example, speedup always led to redundancy. While the number of redundancies typically increases, note that it may also decrease from one iteration to the next. The second experiment performs redundancy removal during each pass of the timing optimization algorithm.

While this procedure is computationally expensive, it results in a significantly more optimized circuit. In fact, the final delay of the circuit in the second experiment is 23% less than that of the first experiment (21.80 versus 26.80). On performing redundancy removal on the final circuit of the first experiment, a delay of 21.80 is also obtained, but at an increase of almost 7% in the area of the circuit. However, in general, no definite conclusion can be made about the impact of redundancy during timing optimization on the delay or area of the final circuit. It is conceivable that a redundant circuit may allow some operations that lead to smaller delays and areas, while such an operation may not be possible on an irredundant version of the circuit. The merits of allowing redundancy in timing optimization is not considered further, since few theoretical results appear to exist on this aspect. However, recall the results of Chapter 3; redundancy is undesirable in optimized circuits and every redundant circuit has an irredundant version at least as fast. The interest in this chapter is in studying the conditions that create redundancy. Besides being of theoretical interest, the invariance of testability properties also has several important practical benefits. First, test vector sets are retained if testability remains invariant. This allows testability preserving optimizations to be applied without the need for repeating the test generation phase on the new circuit. Second, maintaining the invariance of some testability criterion may be the only practical method for generating a fully testable multilevel circuit. An example is multiple stuck-fault testability [51].

Table 4.2 shows the final number of redundant faults for various circuits after performing timing optimization on an initially irredundant area optimized circuit. The initial circuits for timing optimization were obtained by using a standard area optimization script in SIS [102], followed by redundancy removal to ensure full testability. Only those circuits in the ISCAS and MCNC benchmark suite that have redundant faults introduced by timing optimization are reported in the table. While most of the circuits have a few redundancies introduced into them during the delay optimization process, a few circuits have a large number of redundant faults. In every case, redundancy removal interleaved with the speed up operations results in smaller and faster circuits. However, recall that timing optimization is performed using static analysis only. In the case when timing optimization uses more accurate delay estimates, such as viability analysis (*c.f.* Section 2.3.1), the redundancy removal will have to be performed more carefully. This is to ensure that circuit does not slow down under the more accurate delay analysis. In fact, a version of the KMS algorithm, described in the previous chapter, provides this form of careful redundancy removal.

without redundancy removal			with redundancy removal	
Delay	Area	# Red.	Delay	Area
38.80	1229.0	0	38.80	1229.0
37.40	1233.0	7	37.40	1233.0
35.80	1241.0	6	35.60	1227.0
34.60	1261.0	10	34.40	1237.0
33.20	1273.0	10	32.80	1249.0
32.60	1292.0	15	31.60	1269.0
31.40	1318.0	20	30.60	1285.0
31.20	1328.0	23	30.00	1297.0
30.60	1338.0	23	29.80	1313.0
30.20	1350.0	23	28.80	1309.0
29.40	1358.0	26	28.00	1321.0
29.00	1376.0	26	26.80	1327.0
28.80	1394.0	28	26.20	1343.0
28.20	1400.0	33	25.80	1367.0
27.80	1414.0	37	25.80	1369.0
27.40	1439.0	40	25.60	1379.0
27.60	1443.0	41	24.40	1367.0
27.40	1471.0	42	23.80	1373.0
†26.80	1483.0	45	23.60	1395.0
			23.40	1425.0
			22.60	1437.0
			22.00	1467.0
			21.80	1473.0

Example circuit is rot.blif

Each primary input arrival time set at 0.0 delay units

† On performing timing optimization after redundancy removal on this circuit, area of circuit with delay 21.80 is 1577.0

Table 4.1: Example: Impact of redundancy on timing optimization

Name	Initial circuit		Final Circuit		
	Delay	Area	Delay	Area	# Red.
C1355	33.20	820.0	30.80	868.0	4
C1908	46.80	849.0	41.80	1079.0	4
C5315	48.20	2904.0	45.40	3037.0	14
5xp1	22.60	188.0	20.00	203.0	3
alu4	27.40	687.0	26.60	691.0	2
apex2	19.20	416.0	18.00	493.0	18
apex4	212.20	4986.0	149.80	5457.0	173
apex7	17.00	431.0	16.00	471.0	4
b9	13.60	266.0	12.40	290.0	6
clip	17.00	201.0	14.60	209.0	2
des	102.40	6109.0	95.40	6146.0	9
duke2	18.60	637.0	18.00	659.0	6
f51m	30.00	238.0	26.20	301.0	13
misex1	20.00	95.0	17.40	146.0	14
misex2	10.60	171.0	9.40	185.0	1
rd73	23.40	111.0	17.80	175.0	4
rd84	19.20	229.0	17.60	353.0	28
rot	38.00	1173.0	27.20	1381.0	40
sao2	21.40	233.0	16.80	276.0	16
z4ml	15.60	76.0	13.60	104.0	5

Each primary input arrival time set at 0.0 delay units

All circuits initially irredundant

Table 4.2: Redundancy in timing optimization

Four of the five steps listed in Section 2.3 alter the network structure during the timing resynthesis algorithm. Each is considered below to determine its effect on the testability. Assume that the circuit before timing resynthesis is fully testable for all single stuck-faults. This may be achieved either by synthesis [6] or redundancy removal via automatic test pattern generation [100].

4.4.1 Node duplication

Node duplication may introduce redundancies. If a network is irredundant before the duplication of a node then the check for an untestable fault in the new network has only to be made on the fanin and fanout connections of the gate being duplicated and its duplicate. In fact, the existence of redundancy in the network after the node duplication *can be predicted by an analysis of the initial irredundant network itself.*

Consider the irredundant circuit shown on the top in Figure 4.4. The stuck-*0* fault on input $i5$ of the gate 2, cannot be tested by a ***single-path sensitization*** test. A single-path sensitization test means that the fault effect (D or $\overline{D}$ in the terminology of [90]) propagates along exactly one path to the output of the circuit. However, it is tested by ***multiple-path sensitization***, where the fault effect propagates along several reconvergent paths to the output. Assume that gates 2 and 5 are to be collapsed for timing optimization, leading to the duplication of gate 2. This new circuit, shown on the bottom in the figure, has redundant stuck-*0* faults on input $i5$ of gate $2'$ and input $i5$ of gate $2''$. The only test vector for the fault in the original circuit is $i_1i_2i_3i_4i_5 = 00001$. This causes a D value on the output of gates 5 and 6. On duplication, one of the D values becomes a 1 at gate 8, thus, blocking the fault effect that is to propagate along the other path.

Now consider the irredundant circuit on the top in Figure 4.5. The stuck-*1* fault on the primary input $i2$ of the inverter, 2, can be detected only by multiple-path sensitization via the vector $i_1i_2i_3i_4 = 0000$, which causes a $\overline{D}$ value (*1* in the good circuit, *0* in the faulty circuit) on the outputs of gates 5 and 6. On duplication of gate 2, however, no redundancy is introduced into the circuit (bottom of Figure 4.5). This is because while testing for the stuck-*1* fault on the $i2$ input of gate $2'$ ($2''$), gate 6 (5) has a 0 value, which is the non-controlling value for gate 8.

Definition 4.4.1 *Let g be a gate with multiple fanouts and let c be one of its fanouts. The* **reconvergent gate set** *of c, $R_g(c)$, is defined as,*

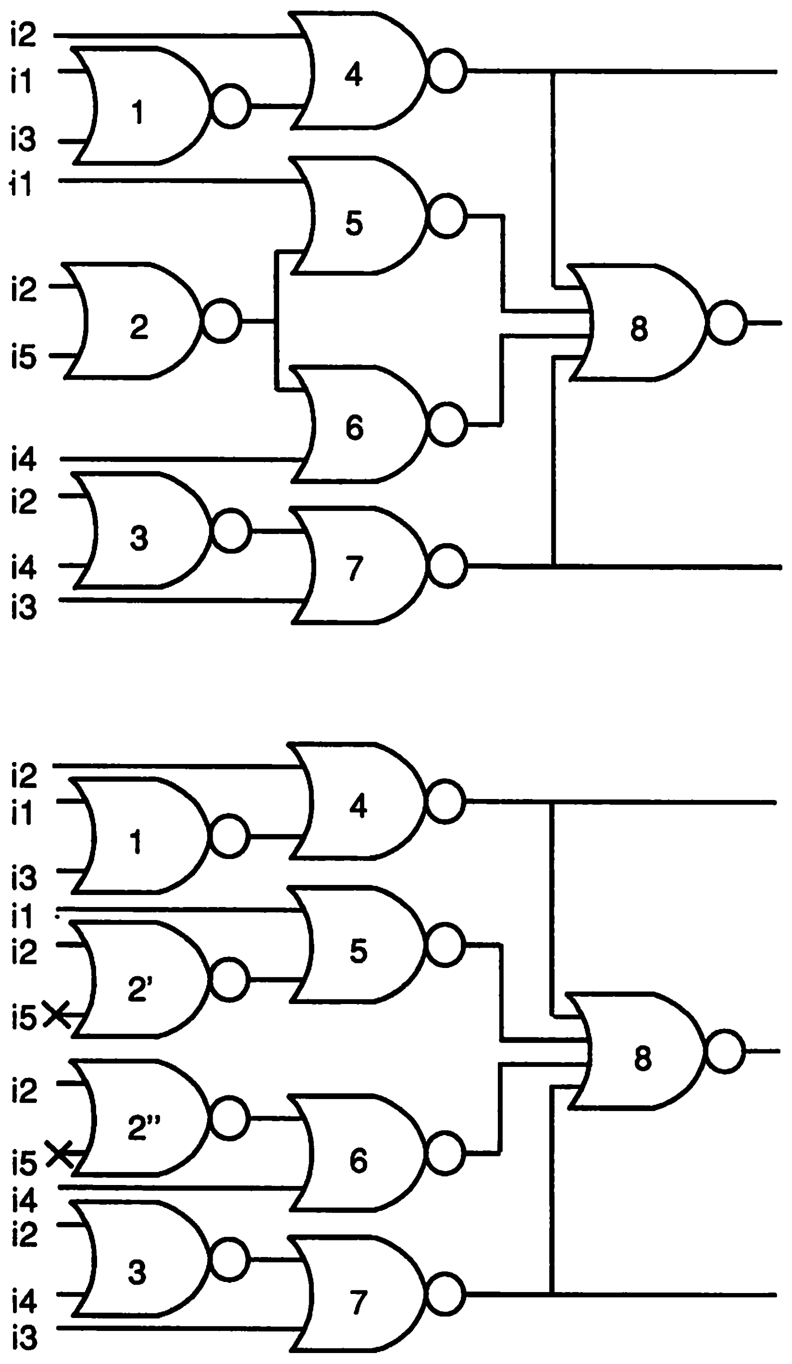
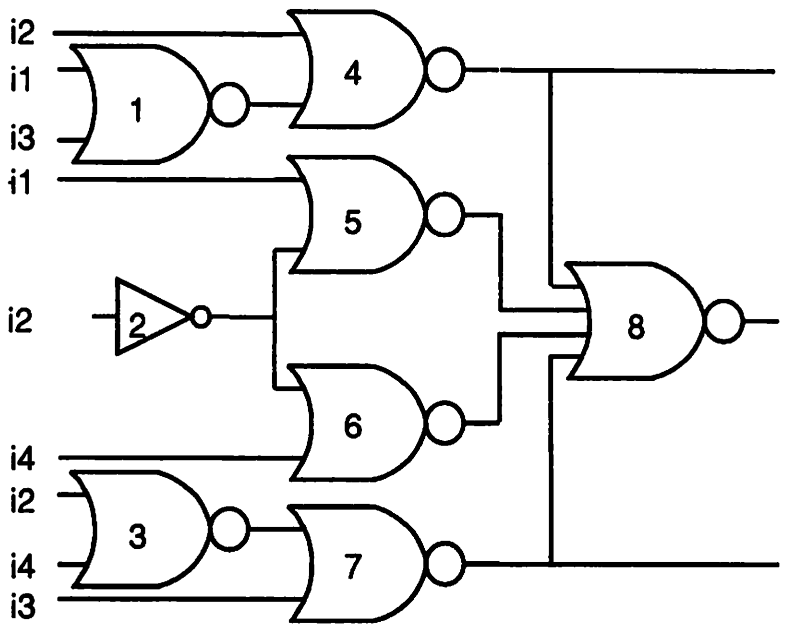


Figure 4.4: Example: Redundant fault on duplication



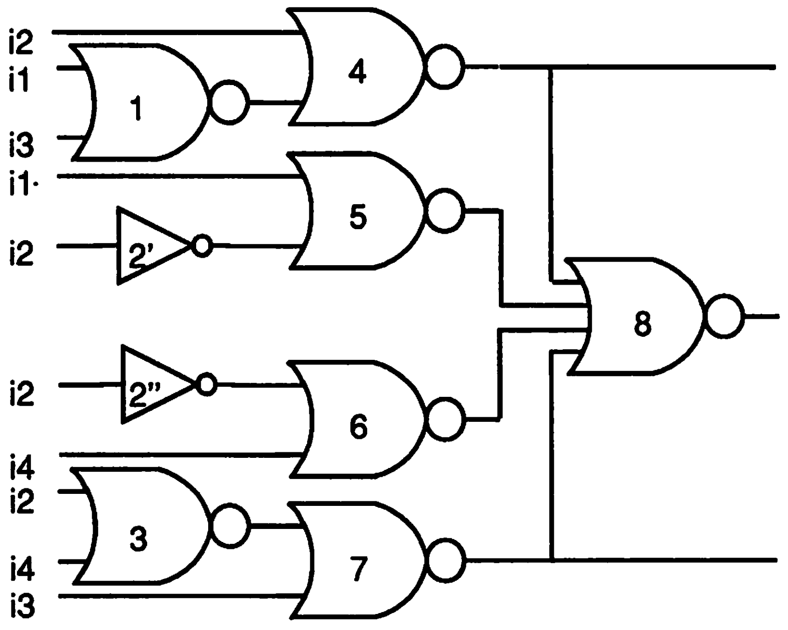


Figure 4.5: Example: No redundant fault on duplication

$R_g(c) = \{r$: paths from g to r include one path through c and one path not through $c\}$. Each path to a gate in $R_g(c)$ that includes c is called a $c \rightarrow R_g(c)$ path. Each path from g to a gate in $R_g(c)$ that does not include c is called a $\overline{c} \rightarrow R_g(c)$.

For example, in the circuit on the top in Figure 4.4, if c is the connection between gate 2 and gate 5, then $R_2(c) = \{8\}$. The path through gates 2, 5 and 8 is a $c \rightarrow R_2(c)$ path and the path through gates 2, 6 and 8 is a $\overline{c} \rightarrow R_2(c)$ path of c. Note that $R_g(c)$ contains only reconvergent points of paths from g.

Theorem 4.4.1 *Let n be a simple gate in a fully testable circuit η, with fanout connections $c_0, c_1, ..., c_k$. Assume that c_0 is within the critical sub-circuit and $c_1, ..., c_k$ are outside. Let η' be the circuit obtained from η by replacing n by two identical gates n_1 and n_2. Let the fanout connection of n_1 be c_0 and let n_2 have fanout connections $c_1, ..., c_k$. Let C denote the set of fanin and fanout connections of n in η. η' remains fully testable if and only if in the original network η both of the following conditions are satisfied for each fault f which may occur on any $c \in C$.*

- *There exists a test vector v_c and at least one fault-propagating path P with $c_0 \in P$ such that all the $\overline{c_0} \rightarrow R_n(c_0)$ paths of c_0 have non-controlling values at the gates in $R_n(c_0)$.*
- *There exists a test vector v'_c and at least one fault-propagating path P' with $c_0 \notin P'$ such that all the $c_0 \rightarrow R_n(c_0)$ paths have non-controlling values at the gates in $R_n(c_0)$.*

Proof Let f_1 on n_1 and f_2 on n_2 in η' correspond to the fault f on n in η.
If part: Assume that both conditions are true. Then v_c detects f_1 in η' and v'_c detects f_2 in η'. The testability of all other connections in η' which do not correspond to some connection $c \in C$ in η remains unchanged. Hence, η' is fully testable.
Only if part: Assume that the first condition does not hold. This implies that any test vector v_c with a fault propagating path through c_0, some path in $\overline{c_0} \rightarrow R_n(c_0)$ causes a controlling value at some gate in $R_n(c_0)$ in η. (This however does not affect the testability of f in η since this value changes to non-controlling in the faulty circuit. Otherwise v_c would not be a test for c). On duplication, this controlling value blocks the propagation of the fault f_1 in η'. Thus, no test for c in η is a test for c in η'. A vector that is not a test for f in η cannot test f_1 in η'. This is because the condition for testing f is weaker than testing for f_1. Consider any vector v_c which is not a test for f. This occurs due to at least one controlling value on a side-input to a gate along each path that f can be tested along.

The same holds when testing for f_1, implying f_1 cannot be tested by v_c if f is not. Thus, f_1 is redundant.

Assume that the second condition does not hold. Then any test vector v_c' for c in η with a fault propagating path not through c_0, some path in $c_0 \rightarrow R_n(c_0)$ has a controlling value on some gate in $R_n(c_0)$ in η. On duplication, this controlling value blocks the propagation of the fault in η'. A vector that is not a test for f in η cannot test f_1 in η'. This is because the condition for testing f is weaker than testing for f_1. Consider any vector v_c which is not a test for f. This occurs due to at least one controlling value on a side-input to a gate along each path that f can be tested along. The same holds when testing for f_1, implying f_1 cannot be tested by v_c if f is not. Thus, f_1 is redundant. ■

4.4.2 Collapsing of critical sub-network

In this step of timing resynthesis the gates in the critical sub-network are collapsed into a single node expressed as a sum-of-products. This sum-of-products is also made *locally prime and irredundant*, in the sense that it is prime and irredundant considering its inputs and output as primary inputs and output respectively.

The effect of collapsing is demonstrated by an example. The circuit on the top in Figure 4.6 is irredundant. Gates 2, 3 and 4 are to be collapsed into a two-level NOR-NOR form. The circuit on the bottom is obtained after making the collapsed region locally prime and irredundant. However, there is a single stuck-*0* redundant fault on the input of gate 5 that is marked with an X in the figure. This fault has been caused by reconvergence that originates outside the collapsed region. Such a redundancy may only be created when the fanout of an input to the collapsed region changes due to the collapse operation. In the example, the fanout of the output of gate 1 increases to two and a redundant fault due to reconvergence of c is created.

Theorem 4.4.2 *Let C be a critical fanout-free sub-network of a fully testable network η and let $n_0, ..., n_k$ be all the gates in C that are collapsed to obtain a two-level (AND-OR gates) node n_k'. A redundant connection can only be created on an input to a first-level (AND) gate in the two-level representation of n_k'. The rest of the new network η' outside n_k' remains fully testable.*

Proof Since the output of the collapsed sub-network for each input vector is unchanged by the collapse operation, the testability of faults not on n_k' in η' remain unchanged. The

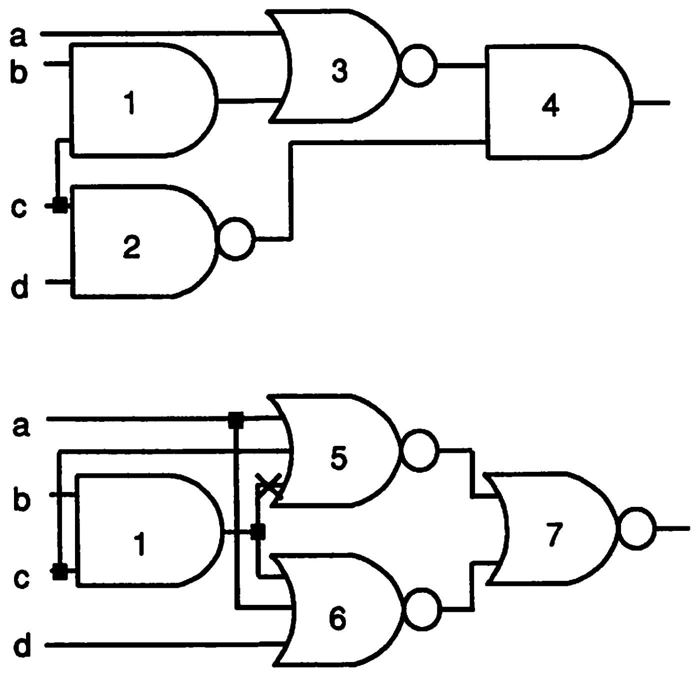


Figure 4.6: Example: Redundant fault on collapsing

input connections and the output connection of n'_k are the same as those of C. Since each connection is irredundant in η it remains irredundant in η'. (As pointed out earlier, the testability of some of the fanout connections of this input may change, but the faults located before the fanout branches on these inputs remain testable in η'.) The testability of the single output connection of n'_k remains unchanged since the function at n'_k is unchanged (although its representation changes from multilevel to two-level logic). An internal single fault in a single output two-level circuit (composed of AND-OR gates) is equivalent to some single fault either on the output of the two-level circuit or the inputs to the AND gates [19]. Thus, the only possible redundant connections are on the inputs to the first-level gates within the two-level representation of n'_k. ■

4.4.3 Resynthesis of a collapsed node

The decomposition of a single collapsed node that is performed in the timing optimization procedure described in [105] uses only algebraic factorization techniques. In particular, the best common algebraic factor which is also suitable with respect to the desired timing property is selected. When no further divisor can be found an algebraic NAND-NAND decomposition into 2-input NAND gates is performed.

Theorem 4.4.3 *100% single-fault testability is preserved on algebraic synthesis of a node.*

Proof Let n be a node in a fully testable network η that is replaced by an equivalent algebraically synthesized sub-network η_n in η'. The testability of any fault on connections to gates which are outside η_n remains unaffected. Any single fault in η_n corresponds to a multi-fault in the two-level representation of n [51]. Since the function at n is prime and irredundant, any multi-fault in n is covered by some single fault on the inputs of n which is testable. Hence, η' is fully single-fault testable. ■

Single fault testability invariance when performing algebraic factoring that allows a restricted use of complementation is shown in [20]. A similar result in [111] shows the invariance of single-fault testability when factoring is carried out using sub-expressions each with either two literals or two cubes.

4.4.4 Merging of identical nodes

After the timing optimization loop is completed, there is a final step that attempts to reclaim some of the area of the circuit. If there exists two or more nodes with identical inputs that compute the same function, these nodes are replaced by a single node. This step may introduce redundancy into the circuit. An example was first provided in [51] and is shown in Figure 4.7. On merging gates 1 and 2 in the irredundant circuit at the top of the figure, a single stuck-0 redundancy is produced on the output of the merged gate. It is easy to see that this is caused by a redundant multi-fault in the first circuit.

Necessary conditions that prove the absence of redundancy on merging of identical gates are not known. However, a sufficient condition for no redundancy to be introduced is as follows.

Theorem 4.4.4 *Let η be a single-fault testable network with identical gates $n_0, ..., n_k$. Let η' be obtained by replacing $n_0, ..., n_k$ by the single gate n. η' remains fully testable if for each single fault f on n the corresponding multi-fault on the gates $n_0, ..., n_k$ is testable in η.*

Proof The testability of faults not on $n_0, ..., n_k$, or n remain unchanged. Each single fault on n in η' is equivalent to a testable multi-fault on $n_0, ..., n_k$ in η. Thus, η' is fully testable. ■

The redundant connections may only be created on the merged gate (either on the inputs ot the outputs). Other sufficient conditions can be constructed but they are fairly weak and appear to be far from necessary. However, if the area of the circuit is not of primary concern in the case where a redundant connection may be created, merging may be omitted to retain testability.

4.5 Redundancy removal problem

Can all redundancies be identified and removed at the moment they are being introduced? If so, it would be possible to efficiently maintain testability without resorting to a complete and general redundancy removal such as that performed by automatic test generation schemes (e.g. Socrates [100]). In particular, are local changes alone sufficient to maintain single-fault testability? Unfortunately, the answer is no.

First consider node duplication. Assume we start with a given fully testable circuit, η, with a critical sub-network, C, where nodes $n_0, n_1, ..., n_k$ must be duplicated before C can be collapsed and resynthesized for delay. Assume that nodes $n_0, n_1, ..., n_k$ are in reverse topological order. Using the conditions mentioned in Section 4.4.1, determine whether the duplication of a node n_0 creates a redundancy. If not, duplicate n_0 in the new circuit η_0.

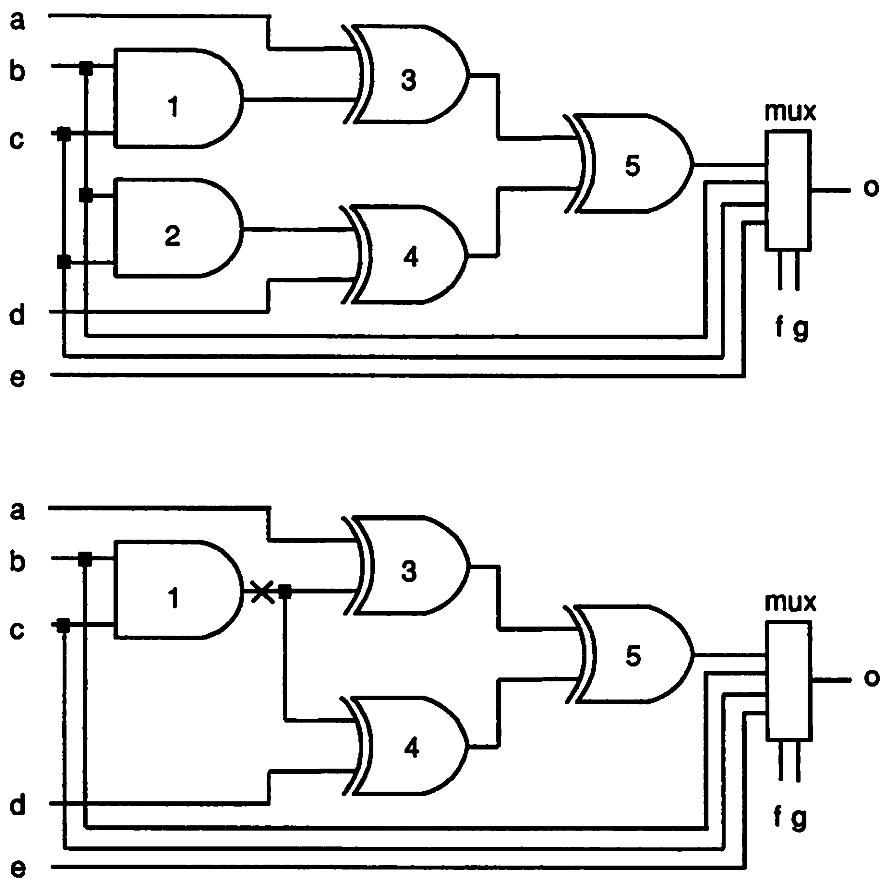


Figure 4.7: Example: Redundant fault on merging

If duplication of n_0 causes a redundant connection, remove this connection from η_0. Then continue to check for further redundancy on duplication of the other nodes on this new circuit η_0.

Consider the irredundant circuit shown on the top left in Figure 4.8. On duplication of gate 2 the stuck-*0* fault on input b of gate $2'$ is untestable in the circuit at the top right. On removal of this fault new redundant stuck-*0* faults are created on input c of gate 5, input a of gate 3 and input d' of gate 4, as shown in the circuit at the bottom of the figure. These secondary redundancies may occur on other gates in the circuit. More precisely, the secondary redundancies can only be created on gates in the transitive fanin cone of the gates in the transitive fanout of the gate on which the redundant connection is being removed. Other than this rough location, currently, there is no way of predicting the effect of a redundancy removal. Analogously, the removal of a redundant connection created by collapsing and gate merging may also introduce such secondary redundancies which are difficult to locate precisely.

Thus, it is not possible to maintain complete single-fault testability of a network during timing optimization with local changes alone.

4.6 Multiple stuck-fault testability effects

Since single stuck-fault testability cannot be maintained, is there a stronger testability property that can be easily maintained as an invariant during the various steps of resynthesis? Two candidates are multi-fault testability [51] and robust path delay-fault testability [36]. Similar to single stuck-fault testability, it can be easily shown that duplication and collapsing do not preserve multi-fault testability, while algebraic synthesis of a (single-output) node in a multi-fault testable network preserves testability. However, unlike single-fault testability, multi-fault testability is retained on merging of identical gates in a fully testable network.

Theorem 4.6.1 *100% multi-fault testability is preserved on gate merging.*

Proof Let gates n_1 and n_2 in η be merged into a single gate n in circuit η'. Consider any multi-fault m in η' corresponding to an array of single stuck-faults. Any single stuck-fault in η' corresponds to either a single or multiple stuck-fault in η. If the fault is on n in η' it corresponds to a multi-fault on n_1 and n_2 in η, otherwise it corresponds to a single-fault in

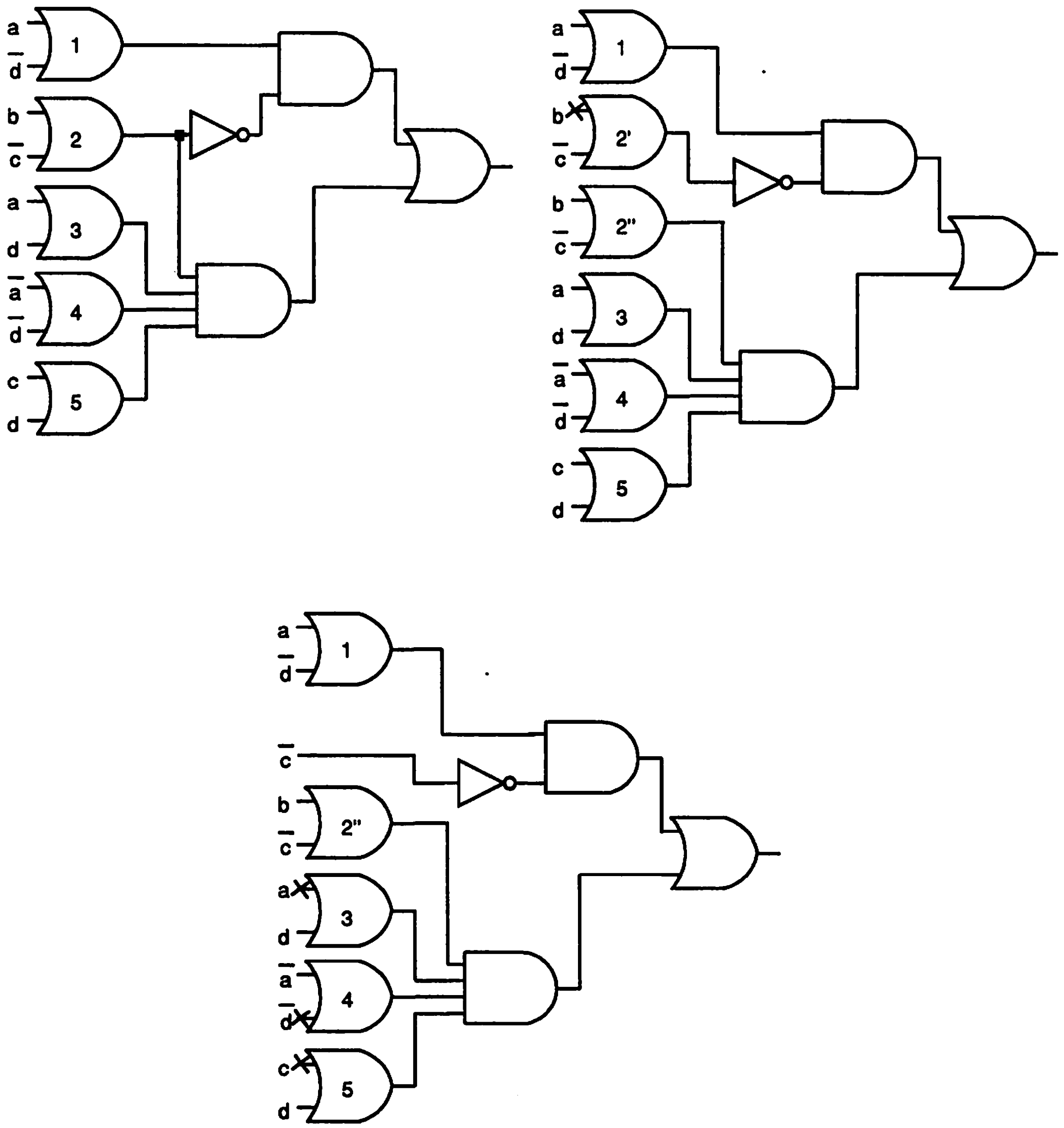


Figure 4.8: Example: Redundancy removal problem

η. Thus, each component single-fault of a multi-fault in η' corresponds to a multi-fault in η, implying that for every multi-fault in η' there is a corresponding multi-fault in η. Since η is multi-fault testable, every multi-fault in η' is testable. ■

Unlike the case of single faults, a redundant multi-fault cannot be localized as occurring only on the duplicated or collapsed gates. This happens because a redundant multi-fault may have its component single faults anywhere in the circuit, although it appears that at least one stuck-fault component must occur on the duplicated or collapsed gate. This causes the redundancy identification phase itself to be nearly hopeless.

In summary, the case of multi-fault testability is similar to single-fault testability. Unlike single-fault testability, multi-fault testability is retained on gate merging. However, it is not retained during duplication and collapsing, and redundancy removal of multi-faults is not practical.

4.7 Robust delay-fault testability effects

Here it is first shown that 100% robust path delay-fault testability[3] is retained during timing optimization if the initial circuit is synthesized using algebraic factorization on an initial two-level RPDFT implementation. Besides providing high levels of testability these circuits have properties that make them attractive in timing optimization, as will be discussed in Section 4.7.5. Each step of timing optimization is again considered as being separately performed on a fully RPDFT circuit to determine the effects on the testability of the resultant circuit.

4.7.1 Node duplication

Theorem 4.7.1 *100%* RPDFT *is preserved on gate duplication and the set of tests is preserved.*

Proof Let η be the initial network and η' be the circuit after duplication. The number of paths remains unchanged by duplication and each path P' in η' corresponds to a unique path P in η. The paths in η and η' are identical except for a relabeling of gates in the path. This means that for each input vector, for any path P in η and the corresponding path P'

[3]As mentioned earlier, all the results stated here are for the hazard-free single-path propagating delay-fault model. However, the results extend to the general model. Wherever possible, the proofs will avoid discriminating between the two models.

in η', the side-paths and side-inputs to the gates have the same value. Thus, if P is robust path delay-fault testable, so is P'. This also implies that any test vector pair for P in η is a test for P' in η'. Hence, the set of tests is preserved. ■

4.7.2 Resynthesis of a collapsed node

Theorem 4.7.2 *100%* RPDFT *is preserved on algebraic synthesis of a node and the set of tests is preserved.*

Proof Let n be a node in a fully RPDFT network η that is replaced by an equivalent algebraically synthesized sub-network η_n in η'. Let P be the paths in η and P' be the paths in η'. Let P_n be the paths passing through n in η and P'_{η_n} be the paths passing through η_n in η'. Clearly, the paths $P - P_n$ in η remain unchanged in η', since they are unaffected by the factoring on node n. Since the logical functionality of each side-input to every path in $P - P_n$ also remains unchanged, these paths remain RPDFT. Now consider the paths in P_n. Each path $p_n \in P_n$ passes through an AND gate and the OR gate in the complex node representation of n. On factoring, two or more of these paths are merged into a single path $p'_n \in P'_{\eta_n}$ (*c.f.* Section 4.2). The concern is with the RPDFT of p'_n. Since each path in P_n that is merged into p'_n is RPDFT, p'_n itself is tested for RPDFT by any one of the tests of a component path. Hence, η' is 100% RPDFT and the test set is preserved. ■

4.7.3 Merging of identical nodes

Theorem 4.7.3 *100%* RPDFT *is preserved on gate merging and the set of tests is preserved.*

Proof Similar to that of Theorem 4.7.1. Let η be the initial network and η' be the circuit after merging. The number of paths remains unchanged by merging and each path P' in η' corresponds to a unique path P in η. The paths in η and η' are identical except for a relabeling of gates in the path. This means that for each input vector, for any path P in η and the corresponding path P' in η', the side-paths and side-inputs to the gates have the same value. Thus, if P is robust path delay-fault testable, so is P'. This also implies that any test vector pair for P in η is a test for P' in η'. Hence, the set of tests is preserved. ■

4.7.4 Collapsing of critical sub-network

Collapsing of a sub-network in a RPDFT circuit may create untestable robust delay-fault paths. This redundancy can be related to the splitting of paths that occurs on collapsing (*c.f.* Section 4.2). For example, if a single path is split due to collapsing, then some of the paths resulting from this split may become untestable. An example is shown in Figure 4.9. In the fully RPDFT circuit shown on the top in the figure, consider the path shown in bold, $\{d, 5, 7, o\}$. On collapsing the encircled region; consider the bold paths $\{d, 8, 11, o\}$, $\{d, 9, 11, o\}$, $\{d, 10, 11, o\}$ in the new circuit. These three paths correspond to the splitting of path $\{d, 5, 7, o\}$ in the initial circuit. The path $\{d, 10, 11, o\}$ is RPDFT whereas the other two paths are (robust) untestable for delay-faults. Now consider the path $\{b, 3, 5, 7, o\}$ in the original circuit. No splitting occurs and the corresponding path is $\{b, 10, 11, 7, o\}$. Hence, if the original path is RPDFT, so is the corresponding path after collapsing.

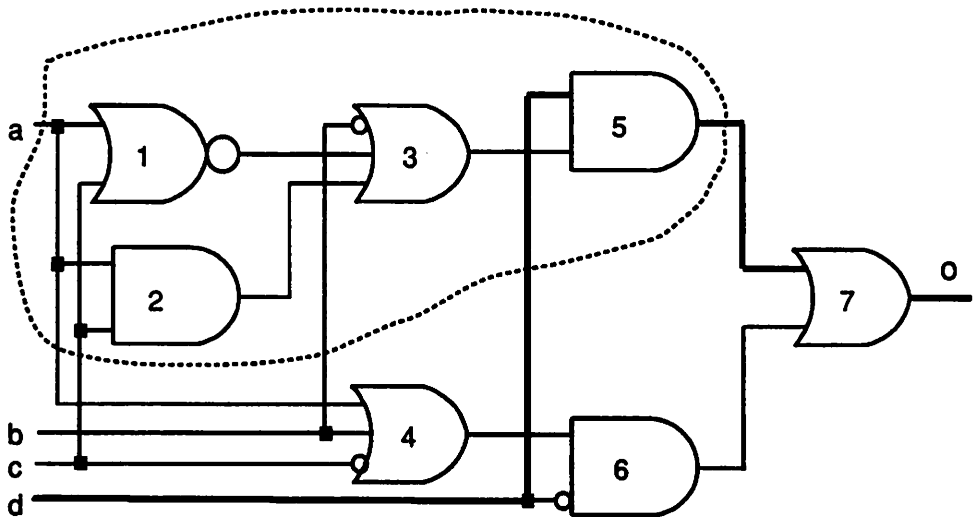
Functions also exist that are not fully RPDFT in any two-level implementation, but may be realized with 100% RPDFT in a multilevel implementation [36, 63]. Therefore, on collapsing these circuits to two-levels it is impossible to retain the full RPDFT property. However, we can prove a result for a smaller class of RPDFT circuits, denoted ARPDFT, namely those multilevel circuits that are obtained from two-level 100% RPDFT circuits using algebraic factorization only.

Theorem 4.7.4 *Let $n_0, ..., n_k$ be all the nodes in a critical sub-network C of an* ARPDFT *network η that are collapsed into node n_k. The new network η' is an* ARPDFT *network.*

Proof The number of paths in the circuit may change due to collapsing. Each path in η is a merge of some paths from the original two-level function F. Since the paths in η' correspond to some splitting of paths in η, η' must also be some merge (possibly different from that of η) of paths from F. Thus, η' remains RPDFT since each path in the two-level implementation of F is RPDFT. ■

It can be easily seen that performing duplication and merging on an ARPDFT circuit results in an ARPDFT circuit. This follows simply because any ARPDFT circuit is also a RPDFT circuit; applying Theorems 4.7.1 through 4.7.3 yields the result.

Thus, for the synthesis approach of obtaining multilevel RPDFT circuits starting from fully testable circuits via algebraic factorization only, we have shown this testability criterion is retained during timing optimization.



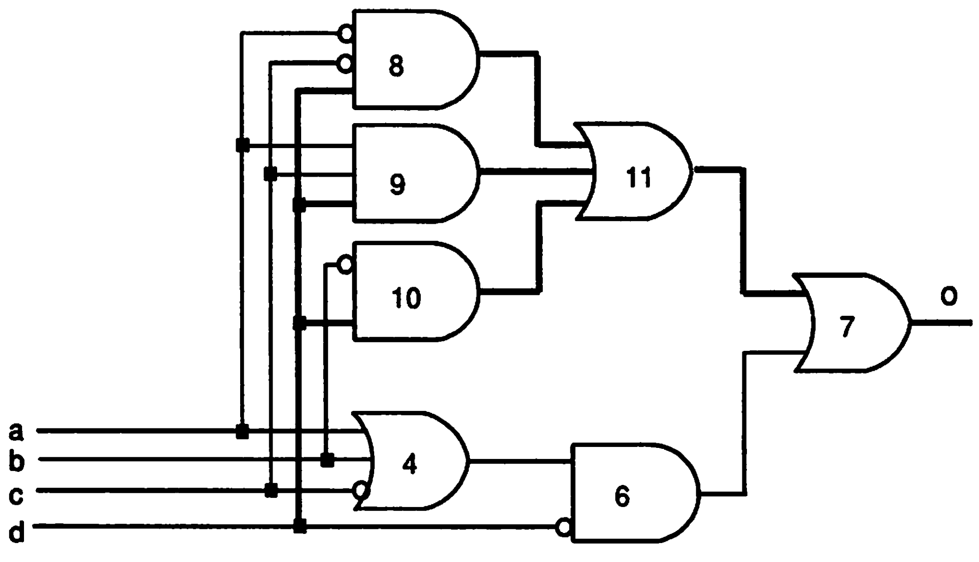


Figure 4.9: Example: RPDFT untestability on collapsing

4.7.5 Delay-fault testability in delay optimization

An important effect that arises out of using robust path delay-fault circuits in delay optimization is discussed here. This is done by an analysis of the computed delay of a circuit before and after timing optimization using three different techniques.

Static analysis: Consider a delay optimization algorithm that selects the critical path based on the longest path delay in the circuit, which may be false. After timing optimization, even though the path is topologically shorter, it may become sensitizable and possibly cause an increase in the delay of the circuit.

Viability analysis: The problem due to speed up of false paths may be avoided by resorting to a more accurate delay analysis algorithm, such as viability analysis [76] (Section 2.3.1). However, there are at least two disadvantages of this. First, viability analysis is computationally expensive. Second, while the longest sensitizable path is selected to be sped up, the resynthesis may cause a previously non-sensitizable path to become sensitizable. Though this new sensitizable path is not longer than the critical path of the original circuit, it may well become the longest sensitizable path in the new circuit. Thus, in the worst case, the timing optimization operations may not have provided any speed up in the circuit.

ARPDFT **circuits:** On the other hand, if a 100% ARPDFT testable circuit is used in delay optimization, every path remains statically sensitizable throughout the process and the longest path is always critical. Thus, once the longest path is selected for speed up, the circuit delay after timing optimization is always predictable. However, while ARPDFT is an attractive testability criterion it is not known if all circuits may be implemented as ARPDFT circuits with the same performance, albeit with some (possibly significant) area penalty.

4.8 0-1 static sensitization

In the previous section, the most stringent condition known for testability, namely robust path delay-fault testability (synthesized using algebraic factorization on a fully testable two-level circuit), is shown to be invariant during timing optimization. This criterion also has a desirable property in the speed-up of circuits, since the problem of false paths does not arise when using this criterion. However, it is not known whether there exists an equivalent RPDFT implementation for every circuit. Additionally, due to the (possibly severe) area overhead when implementing this criterion, it is pertinent to ask if there exists a weaker testability criterion which can always be achieved on an arbitrary circuit and preserved during timing optimization. A *testability-timing invariant* is now considered by defining a new testability model which includes the advantages of delay-fault testability for timing considerations, but reduces the stringent constraints on synthesis. It combines the least common denominators for timing and testability properties of a circuit. Recall the definition of a path being statically sensitizable to *0* or *1* .

Definition 4.8.1 *A path is said to be* **statically sensitizable to** *0 (***statically sensitizable to 1***) if there exists an input cube which sets all the side-inputs to the path to non-controlling values and causes a value of* 0 *(1) on the output of the path.*

Definition 4.8.2 *A path is* **0-1 statically sensitizable** *if it is statically sensitizable to a* 0 *and statically sensitizable to a* 1 *. A Boolean network is* **100% 0-1 statically sensitizable** *if each path is 0-1 statically sensitizable.*

From the definition above it is seen directly that a 100% 0-1 statically sensitizable circuit implies:

1. Each path is statically sensitizable, and hence there exist no false paths.
2. The network is fully single-fault testable. In fact, each single-fault is testable along each path to any primary output since all the paths are statically sensitizable.

The weakest known invariant for predictable timing analysis is the static sensitization of each path. If static analysis is used, then the occurrence of long false paths causes pessimism in the results. If viability analysis is used, then any increase in the delay of a gate (or path) during speed-up (this may easily happen due to an increase in fanout of the gate) may result in a change in viability of other paths. This results in expensive recomputation of the sensitization of paths throughout the circuit, and the possibility that a long false path may become viable. (Note that viability analysis on paths remain unchanged if gates are only sped up [76].) Single-fault testability is the weakest circuit testability property. Thus, 0-1 static sensitization possesses the weakest properties that exist for accurate timing analysis (using static analysis) and testability of Boolean networks.

The relationship to the other known fault models is illustrated in Figure 4.10. Complete 0-1 static sensitization is implied by a 100% RPDFT circuit and in turn implies full

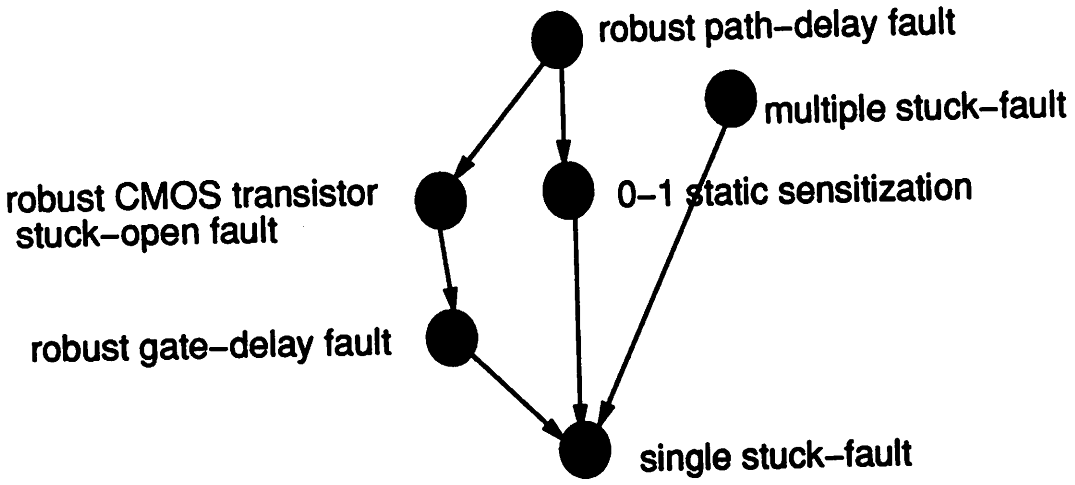


Figure 4.10: Relationship of 0-1-static-sensitization to fault models

single-fault testability. There appears to be no relationship between multi-fault testability and 0-1 static sensitization. 100% 0-1 static-sensitizable circuits exist which are not fully multi-fault testable [31]. However, it is not known if full multi-fault testability in a single output circuit implies full 0-1 static sensitizability, although it appears that multiple output multi-fault testability does not require 100% 0-1 static sensitizability.

4.8.1 Synthesis of 0-1 statically sensitizable circuits

By far the most significant difference between the 0-1 statically sensitizable property and RPDFT is that every circuit can be made 0-1 statically sensitizable with no increase in the delay of the circuit; the same is not true of RPDFT.

Conceptually, a fully 0-1 statically-sensitizable circuit may be achieved by converting any arbitrary circuit, η, into a logically equivalent circuit, η_{ff} with fanout greater than one only on the primary input leads. This is achieved by performing a reverse topological traversal of the circuit and performing a duplication of a gate which has fanout greater than one. On performing redundancy removal on η_{ff} a fully 0-1 statically sensitizable circuit is obtained. By using the KMS algorithm of Chapter 3, the delay of the final circuit can be ensured to be no greater than the initial circuit (*c.f.* Section 3.3).

In the case of RPDFT, as mentioned previously, circuits exist for which fully RPDFT versions can only be achieved via resynthesis [63], which may result in an increase in the delay of the circuit.

4.8.2 Node duplication and merging of identical nodes

Theorem 4.8.1 *100% 0-1 static sensitization is preserved on gate duplication and gate merging. In addition the test vectors are also preserved.*

Proof Similar to that of Theorems 4.7.1. Let η be the initial network and η' be the circuit after duplication or merging. The number of paths remains unchanged by duplication or merging and each path P' in η' corresponds to a unique path P in η. The paths in η and η' are identical except for a relabeling of gates in the path. This means that for each input vector, for any path P in η and the corresponding path P' in η', the side-paths and side-inputs to the gates have the same value. Thus, if P is 0-1 statically sensitizable, so is P'. This also implies that any test vector pair for P in η is a test for P' in η'. Hence, the set of tests is preserved. ■

4.8.3 Resynthesis of a collapsed node

Theorem 4.8.2 *100% 0-1 static sensitization is preserved on algebraic synthesis of a node and the set of tests is preserved.*

Proof Similar to that of Theorem 4.7.2. Let n be a node in a fully 0-1 statically sensitizable network η that is replaced by an equivalent algebraically synthesized sub-network η_n in η'. Let P be the paths in η and P' be the paths in η'. Let P_n be the paths passing through n in η and P'_{η_n} be the paths passing through η_n in η'. Clearly, the paths $P - P_n$ in η remain unchanged in η', since they are unaffected by the factoring on node n. Since the logical functionality of each side-input to every path in $P - P_n$ also remains unchanged, these paths remain 0-1 statically sensitizable. Now consider the paths in P_n. Each path $p_n \in P_n$ passes through an AND gate and the OR gate in the complex node representation of n. On factoring, two or more of these paths are merged into a single path $p'_n \in P'_{\eta_n}$ (*c.f.* Section 4.2). The concern is with the 0-1 static sensitizability of p'_n. Since each path in

P_n that is merged into p'_n is 0-1 statically sensitizable, p'_n itself is 0-1 statically sensitizable by any one of the tests of a component path. Hence, η' is 100% 0-1 statically sensitizable and the test set is preserved. ∎

4.8.4 Collapsing of critical sub-network

Collapsing of a sub-network in a 0-1 static sensitization circuit may create paths that are not statically sensitizable. While the 0-1 static sensitization property can be recovered for these classes of circuits, it may be computationally expensive. However, we can prove a result for a smaller class of 0-1 static sensitization circuits, namely those multilevel circuits that are obtained from two-level 100% 0-1 static sensitization circuits using algebraic factorization only. Let the notation *0-1 algebraic static sensitization* denote these circuits.

Theorem 4.8.3 *Let $n_0, ..., n_k$ be all the nodes in a critical sub-network C of a 0-1 algebraic static sensitization network η that are collapsed into node n_k. The new network η' is a 0-1 algebraic static sensitization network.*

Proof Similar to that of Theorem 4.7.4. The number of paths in the circuit may change due to collapsing. Each path in η is a merge of some paths from the original two-level function F. Since the paths in η' correspond to some splitting of paths in η, η' must also be some merge (possibly different from that of η) of paths from F. Thus, η' remains 0-1 statically sensitizable since each path in the two-level implementation of F is 0-1 statically sensitizable. ∎

It can be easily seen that performing duplication and merging on an 0-1 algebraic static sensitization circuit results in a 0-1 algebraic static sensitization circuit.

4.9 Other performance optimizations

Generalized select transform: A generalization of the Shannon cofactor [103] is effective in delay optimization of circuits with only a few long critical paths. This technique is reported in [11] and is illustrated using Figure 4.11, where the bold path is critical. The transformation moves the late arriving signal a nearer to the output.

Generalized bypass transform: Chapter 2 describes the example of the carry-skip adder which is obtained from a ripple-carry adder by adding bypass logic consisting of an AND gate and a MUX. A recent technique that generalizes this technique to speed

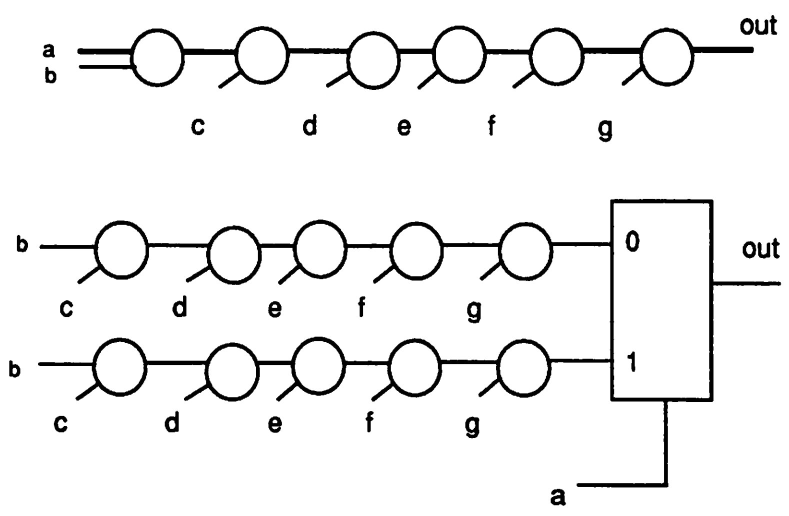


Figure 4.11: Generalized select transform

up arbitrary circuits by making critical paths false is described in [78]. This approach is illustrated using Figure 4.12. The speed-up is accomplished by making the long path from f_m a false path.

Boolean simplification: Don't cares arising from the structure of a Boolean network are most often used to minimize the area of the circuit. [26] describes a method of optimizing the delay of the circuit by reducing the level of the critical paths while using don't cares during simplification. The global flow technique of using implications between connections to move late signals forward may be considered similar to this technique [11].

Each of these three techniques uses Boolean identities in the application of the timing optimization transform. Thus, it is more difficult to say much about the testability invariance of a circuit when employing these techniques. Similar to the reasoning used earlier in this chapter, the locality of possible redundancies is known. However, removal of these redundancies may create secondary redundancy elsewhere in the circuit.

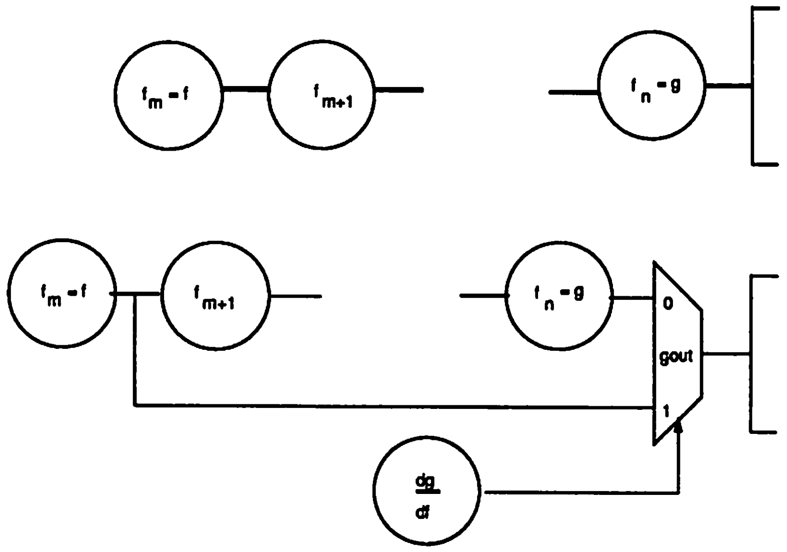


Figure 4.12: Generalized bypass transform

4.10 Conclusions

Motivated by the results of Chapter 3 where it is proved that redundancy is not necessary to reduce the delay of a circuit, this chapter explores whether performance optimization may be performed without introducing redundancy. It is demonstrated how timing optimization may introduce single-fault redundancies. While identification of these redundancies is straightforward, removal of these redundancies on-the-fly during timing optimization involves substantial computation, equivalent to complete (conventional) test generation and redundancy removal [100].

These results motivate the need for an alternate testability criterion that can be easily maintained during timing optimization. It is shown that the stringent condition of robust path delay-fault testability can be easily maintained with the current known synthesis procedures. This requires that only algebraic factorizations be used along with duplication, collapsing and merging of gates on an initially two-level robust path delay-fault testable circuit. At the present time it is unclear whether the use of robust path delay-fault testable circuits may preclude high performance circuit implementations. The impact on the resulting area is also an open question.

A new testability property, termed 0-1 static sensitization, is developed that has the same attractive properties of ARPDFT circuits, *viz.* invariance during timing optimization and absence of false paths. Unlike ARPDFT however, fully 0-1 statically sensitizable circuits can always be realized with no increase in the delay of the circuit.

The results of this chapter restrict the speed-up operations to RPDFT circuits synthesized used constrained two-level minimization [36, 37], followed by algebraic factorization, *i.e.* ARPDFT circuits. Even though this is presently the only effective method of synthesizing RPDFT circuits, it is interesting to explore if transformations exist which can derive a fully RPDFT circuit from a general Boolean network. While one such technique has been proposed [63, 89], it is based on repeated application of the select transformation described in Section 4.9. This procedure involves a significant area penalty and less expensive techniques are still desired. Future work will aim at developing an algorithm that applies a series of transformations to achieve full robust delay-fault testability. Currently, a few results are known in this direction, but further work is required before an efficient algorithm is realized. For a given circuit, the goal of such an algorithm can be listed as:

- Make the given circuit RPDFT.

- Ensure the final circuit is as fast as the initial circuit.
- Minimize the area overhead in the transformation.

As shown earlier, RPDFT is not retained during collapsing. However, it is plausible that robust untestable delay-faults introduced during timing optimization may be removed by local redundancy removal and resynthesis only. Further exploration of this aspect is required.

A second problem that remains unexplored is the effect of performance optimizations on the timing behavior of a circuit. As mentioned qualitatively in Section 4.7.5, while the timing characteristic of paths through the resynthesized region may improve, the timing behavior of paths outside this region may deteriorate. This appears especially true when functional analysis is used to determine the delay of a network. Preliminary results of the effects of transistor sizing on timing characteristic of paths has been recently reported in [25]. However, the effects of structural changes in the network is not that well understood as yet.

Chapter 5

Path-Recursive Functions and Applications

In Chapters 2 through 4, the interaction between the delay and testability of circuits is studied. Two properties of circuits considered there are the computed delay of a circuit using viability analysis and robust path delay-fault testability. The efficient computation of these two properties is the subject of this chapter. Both problems fall into a class of problems that involve the functional analysis on paths or sets of paths. A general approach that efficiently solves this class of problems is developed. The chapter is organized as follows. Section 5.1 is an introduction and motivation to the difficulty of the problems being solved. An overview of the proposed paradigm that can be used to solve any functional path analysis problem is presented in Section 5.2. The first application of the technique is to viability analysis using a well-known path-tracing algorithm (Section 5.3). However, due to a drawback arising from the amount of path-tracing required in large circuits, a timing analysis technique without path-tracing is introduced and developed in Section 5.4. The functions that provide necessary and sufficient conditions for the most stringent model of delay-fault test generation are developed in Section 5.5. The application of the technique to the other delay-fault models is indicated in Section 5.6. Section 5.7 concludes the chapter.

5.1 Functional path analysis problems

Chapters 3 and 4 primarily deal with the synthesis of fully testable circuits during or after the performance optimization process. However, the synthesis techniques discussed there only apply following an analysis of the requisite property being synthesized for. For example, the results of Chapters 2 and 3 require the computation of the delay of a circuit using viability analysis. Chapter 4 provides invariance results of the robust path delay-fault testability of circuits undergoing performance optimizations. Thus, a necessary requirement before the synthesis procedures become efficient is the functional analysis of paths for delay estimation or delay-fault test generation. In fact, the functional analysis of paths through combinational logic circuits has emerged as a critical problem in timing analysis, delay-fault and τ-irredundant test generation, and hazard analysis. These areas have been thoroughly studied and are well understood [14, 9, 76, 75, 41, 106, 68, 36, 88, 110, 42]. The algorithms devised, however, have not yet been able to analyze large combinational circuits. In this chapter this problem is solved by introducing a general framework for algorithmic development and applying this framework to timing analysis and delay-fault test generation problems.

Timing analysis and delay-fault test generation are examples of problems which require functional analysis on the paths of the circuit. In both problems the question of interest is whether a path or a set of paths is capable of propagating an event; the problems differ in the subtleties of event propagation, and, more precisely, the delay assumptions underlying event propagation. For example, in timing analysis the interest is in determining the longest path in the circuit along which a signal can propagate. In robust path delay-fault test generation, for each path in the circuit, a vector pair that sensitizes the path under test is required. Moreover, the test must not be invalidated by hazards.

The first, and to date the only, approach taken for both problems is a modification of existing stuck-fault test generation programs [14, 41, 68, 101]. Paths are traced, input to output, asserting values on signal wires during the trace. The programs and problems differ in the values which are asserted on signal wires, but in no other essential manner. In [74], it is observed that asserting values on signal wires is equivalent to the implicit computation of a Boolean function. It is concluded there that the various programs can be analyzed through the explicit computation of these functions. In [74], the suite of programs presented in the literature for the timing analysis problem are analyzed and systematized in precisely this fashion. It is shown that the various programs in fact differ only in the functions they compute and the manner in which these functions are determined to be satisfiable. However, this explicit computation of the sensitization functions is viewed largely as a theoretical tool in [74], though a program is devised which performs timing analysis through explicit computation of these functions [75]. Up to now it appears that path-tracing algorithms outperformed the approach of computing a single Boolean function that captured a desired property for a path or a set of paths.

Recently, two factors have emerged which make explicit computation of these functions attractive as an implementation strategy for timing analysis and delay-fault test generation:

- Analysis of the form of the sensitization functions reveals that the explicit form of these functions yields an efficient data structure for their storage and manipulation; and
- Recent work in test generation [64] demonstrates that, while the asymptotic complexity of finding a satisfying assignment of an arbitrary Boolean function is likely exponential, the practical performance of well-chosen algorithms is very good.

The contribution of this chapter is to demonstrate the first aspect. The second factor is described in Appendix A. Here timing analysis and delay-fault test generation are performed through the explicit computation of path sensitization functions. A recursive form of the formulation of these functions is provided, from which a multilevel logic network is derived to represent these functions. It is shown that the size of such a network is linear in the size of the network under analysis.

5.2 Path-recursive functions

Any path analysis problem that involves the computation of Boolean functions may be naively formulated as follows. Let the property that is being determined for a path P in a network η be denoted by the function $\mathcal{F}_P$. $\mathcal{F}_P$ is usually a function of the primary inputs plus some intermediate variables in η depending on the computation involved. The property is said to exist for P if a satisfying assignment is found for $\mathcal{F}_P$. Each satisfying assignment is specified as a set of values on the primary inputs. Clearly, by exhaustively computing $\mathcal{F}_P$ for each path P in η, it can be determined whether the desired property holds for any path. The first drawback of this method is the number of computations required to compute each $\mathcal{F}_P$. A second drawback arises if the size of the representation of $\mathcal{F}_P$ happens to be large. In general, a bound on the size of the representation cannot be guaranteed by this method. Besides the impact on the memory usage, the size of the function is of concern for the satisfiability phase, where a BDD or SAT program is invoked. While there is no rigorous proof that a compact representation always requires less effort for processing than a larger equivalent representation, this observation holds empirically for both the SAT program (*c.f.* Appendix A) and BDD computations [12].

The most common approach to alleviate the first problem is to use a path-tracing algorithm to compute the functions incrementally. Typically, a function of a given path P may be expressed recursively as some Boolean combination of other functions on partial paths that *interact* with P. These functions are built up while paths are traced. Thus, it is a small step from the realization that path-tracing algorithms implicitly compute Boolean functions to the realization that the reason algorithms trace paths is simply to compute these functions efficiently. However, it is possible that there are more efficient methods than path-tracing to compute these functions; hence it is worthwhile to write the functions out explicitly so that algebraic machinery may be used to simplify the functions and suggest new methods of computation. The set of functions which are computed by path-tracing algorithms are collectively called *path-recursive functions.* A path-recursive function γ on a path P is denoted γ_P, and is a function from the space of primary input variables onto $\{0,1\}$.

Definition 5.2.1 *A function γ is* **path-recursive** *over a network N if and only if there is a partial order $\prec$ on the paths of N, such that for each path $P = \{f_0, ..., f_m\} = \{P', f_m\}$, where $f_{m-1}, g_1, ..., g_n$ are the fanins of f_m, there exist functions $\gamma_P^1, ..., \gamma_P^n$ (one for each fanin of f_m), such that:*

1. $\gamma_P = \mathcal{F}(f_{m-1}, g_1, ..., g_n, \gamma_{P'}, \gamma_P^1, ..., \gamma_P^n)$; *and*
2. *Each γ_P^i is a function of path-recursive functions γ_Q for $Q \prec P$.*

If there is an expression for γ_P linear in the size of the set $g_1, ..., g_n$, γ is said to be **linear path-recursive.**

Path-recursive functions over a given network η may be represented in a single multilevel network η_{paths}. Since each path-recursive function γ_P is dependent only upon the functions γ_Q for Q preceding P in the partial order, the construction of η_{paths} may be efficiently interleaved with path-tracing by choosing a path-tracing order consistent with the partial order $\prec$. In particular the best-first order, preferred for efficiency reasons for timing analysis on Boolean networks [74], is consistent with the partial order under which the viability function is path-recursive.

There are two scenarios that arise when tracing paths. In the first category of problems, such as the best-first path-tracing algorithm for viability analysis, the size of η_{paths} is linear in both the size of η and the number of paths traced before a true path is found. In the second case, such as robust delay-fault test generation, the size of the function computed is independent of the number of paths traced before the current path. In this case, any topological order is consistent with the partial order $\prec$, and the size of η_{paths} is linear *only* in the size of η. Both cases permit compact representations, and rapid satisfiability tests in the average case. This is because the efficacy of the satisfiability procedure is dependent upon the size of the function being tested for satisfiability. However, as shown in Section 2.3.1, the functions computed for problems in the first category may sometimes get very large if the number of false paths is huge. Thus, it is advantageous to determine formulations that fall into the second category. One such formulation for viability analysis is developed in Section 5.4 and is empirically shown to be superior to the original formulation.

Figure 5.1 illustrates the construction of the multilevel network that is used to compactly represent linear path-recursive functions. The network on the left is to be analyzed for some path-recursive function. For each node i in this network, the corresponding path-recursive function Ri is created in the network on the right (shown in bold). This is done by traversing the nodes in topological order. Thus, when creating the function Ri, the functions Rj for each fanin node j of i are already created and available for use in creating the node function. In addition the node functions of the fanins of i may also be used in creating Ri (shown by the dotted lines in the figure). To determine the existence of the property for node i, either a BDD is built, or the SAT program (Appendix A) is called for the function Ri.

The importance of path-recursive functions in general, and linear recursive functions in particular, becomes clear when we consider the complexity of the general path analysis algorithms. If the function computation is linear in the number of fanins to a node, then the amortized overhead of function computation is linear in the maximum fanin of a network node. This overhead compares favorably with the common practice of tracing implications of assertions, which (depending upon the algorithm used) is no better than linear in network size and may be as much as quartic in network size [74]. Further details

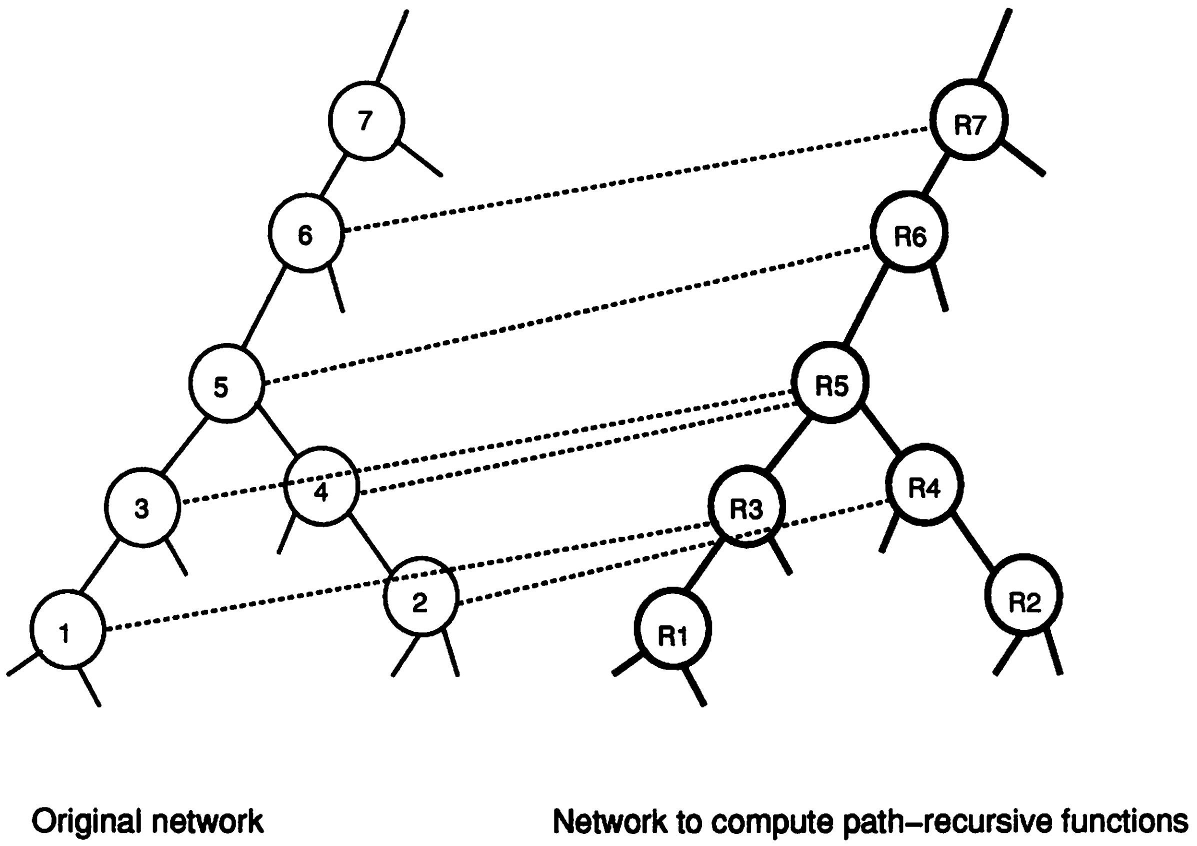


Figure 5.1: Example: Representing path-recursive functions by multilevel network

on the amount of computation required when employing the path-recursive paradigm is provided when each problem is considered individually.

5.2.1 Linearizing Boolean functions

A result is now stated that guarantees a linear sized representation for Boolean functions that exhibit a specific property. In particular, this holds for any Boolean function that is true when any subset of variables satisfy some Boolean function and the remaining variables satisfy some other property.

Theorem 5.2.1 *If $\mathcal{F}_f$ is a Boolean function expressed as*

$$\mathcal{F}_f = \sum_{U \subseteq \mathcal{E}} \{ \prod_{g \in U} \alpha_g \prod_{g \in \mathcal{E}-U} \beta_g \} \tag{5.1}$$

where α_g and β_g are arbitrary Boolean functions, and $\mathcal{E}$ is an arbitrary set, then an equivalent representation of $\mathcal{F}_f$ is

$$\mathcal{F}_f = \prod_{g \in \mathcal{E}} (\alpha_g + \beta_g) \tag{5.2}$$

Proof Assume that $\mathcal{E}$ is the set of $k+1$ elements, $g_0, g_1, ..., g_k$. Let λ_j for $j = 1, 2, ..., 2^{k+1}$ denote index functions from the domain $0, 1, ..., k$ to the range $0, 1, ..., k$.

The j^{th} product term in the sum-of-products expression in the right hand side of Equation (5.1) can be written as

$$\prod_{i=0}^{m} \alpha_{g_{\lambda_j(i)}} \prod_{i=m+1}^{k} \beta_{g_{\lambda_j(i)}} \tag{5.3}$$

Note that there are 2^{k+1} product terms in Equation (5.1).

The right hand side of Equation (5.2) can be written as

$$(\alpha_{g_0} + \beta_{g_0})(\alpha_{g_1} + \beta_{g_1}) ... (\alpha_{g_k} + \beta_{g_k}) \tag{5.4}$$

Thus, the i^{th} term in Equation (5.4) is $(\alpha_{g_i} + \beta_{g_i})$.

$\sum_{U \subseteq \mathcal{E}} \{ \prod_{g \in U} \alpha_g \prod_{g \in \mathcal{E}-U} \beta_g \} \subseteq \prod_{g \in \mathcal{E}} (\alpha_g + \beta_g)$:

For each term of the right-hand side of Equation (5.1) an equivalent term is be generated when the right-hand side of Equation (5.2) is expanded. In particular, the general term in Equation (5.3) is obtained as the product of $\alpha_{g_{\lambda_j(i)}}$ for $0 \leq i \leq m$ and $\beta_{g_{\lambda_j(i)}}$ for $m+1 \leq i \leq k$

from the $\lambda_j(i)^{th}$ term in Equation (5.4).
$\prod_{g\in\mathcal{E}}(\alpha_g+\beta_g)\subseteq\sum_{U\subseteq\mathcal{E}}\{\prod_{g\in U}\alpha_g\prod_{g\in\mathcal{E}-U}\beta_g\}$:
In the sum-of-products representation derived from the right-hand side of Equation (5.2), each term is of the form given by the right-hand side of Equation (5.3). Hence right hand side of Equation (5.2) $\subseteq$ right hand side of Equation (5.1). ■

This result can be considered analogous to the well known result for equivalence between a sum-of-product of minterms and a product-of-sums of minterms for a given set of Boolean variables. The significance of this theorem is that it allows an arbitrary Boolean function that is defined in the form of (5.1) to be realized by an equivalent linear sized representation. The remainder of this chapter demonstrates three applications of this theorem in performing fast timing analysis and delay-fault test generation.

The statement of the theorem can be easily generalized to more than two subsets[1].

Theorem 5.2.2 *If $\mathcal{F}_f$ is a Boolean function expressed as*

$$\mathcal{F}_f=\sum_{\mathcal{V}\subseteq\mathcal{E}}\prod_{i=1}^{n}\prod_{g\in U_i}\alpha_g^i\prod_{g\in\mathcal{V}}\beta_g$$

where $\mathcal{E}$ is an arbitrary set, α_g^i, $i=1,\dots,n$ and β_g are arbitrary Boolean functions, $U_i\subseteq\mathcal{E}$ for $i=1,\dots,n$, $\sum_{i=1}^{n}U_i=\mathcal{E}-\mathcal{V}$, $U_i\cap U_j=\emptyset$ for all $i\neq j$,
then an equivalent representation of $\mathcal{F}_f$ is

$$\mathcal{F}_f=\prod_{g\in\mathcal{E}}(\sum_{i=1}^{n}\alpha_g^i+\beta_g)$$

Proof Similar to Theorem 5.2.1. ■

5.3 Viability analysis

The contributions of the work presented here is discussed following a brief review of the research in the area of timing analysis. The obstacles that these methods face in realizing effective practical algorithms are highlighted.

Functional timing analysis is important during synthesis since circuit simulation is typically too slow to be used for an entire circuit. It is well known that using static analysis with the longest path delay as an estimator of the true delay is sub-optimal. Often both

[1]This general form is not used in this thesis.

designers and synthesis algorithms [66, 78] actually exploit the fact that the longest path does not contribute to the delay of the circuit to come up with high-performance circuits.

There are two requirements for a functional timing analyzer. The delay estimation must be accurate and the analysis must be robust [76], that is, it must provide a correct delay estimate over a range of delays. The simple condition of static sensitization for a path does not guarantee accuracy, since it may sometimes underestimate the true delay of a circuit [14]. The condition of dynamic sensitization is not robust. Viability analysis has been shown to be the most accurate and robust of all published timing analysis algorithms. The algorithms of [14] and [41] have been shown to sometimes provide more pessimistic delay estimates than viability analysis. The method of [24] gives the same result as viability analysis, but reports fewer true paths.

As mentioned earlier, the most common approach to timing analysis has been modifications to existing test generation algorithms. The first approach to timing analysis used the condition of static sensitization to determine if a path is true or false [9]. This condition easily fits into the test generation paradigm; it requires the justification of non-controlling values on the side-inputs to the path being analyzed. However, static sensitization is known to underestimate the actual delay of a circuit [14, 76]. A way to correct this optimistic, and erroneous, estimation by static sensitization is provided in [14]. The algorithm there again naturally extends to the test generation paradigm. This approach, however, does not guarantee as tight a delay estimate as may be possible. Hence, recourse is taken to viability analysis theory for delay estimation, first presented in [76]. More recently, the method of [41] gives the same delay estimate as viability analysis. The approach there uses test generation approach to determine the sensitization condition of each path being traced.

A test generation based approach to determine whether a path is sensitizable, performed on a path by path basis, is undesirable for two reasons. First, numerous paths may be traced before a true path is found, thus requiring several calls to the modified test generation program. Second, justification on a complete path is often overkill. There may be a short partial path which is not sensitizable under any input vector. Hence all paths that include this partial path segment are not sensitizable, and several individual calls to the modified test generation program for each complete path can be replaced by a single call on the partial path segment. This drawback is alleviated by the best-first algorithm that is suggested in [9] and also employed in viability analysis [75]. Recently, [38] provides a new algorithm for delay estimation based on a modified multi-fault test generation algorithm that also considers timing information. The delay is estimated using the same conditions as [24], but path-tracing is not performed.

5.3.1 Viability equations

In viability analysis, the longest path along which an event can propagate from the inputs to the outputs, given an input vector c and an upper bound on the delay of each input to a gate, is to be determined. This determination of whether an input could propagate through gate f_i is performed as follows. For an input f_{i-1} to f_i, some of the other side-inputs to f_i have a lesser upper bound on their delays; these inputs must be set to their sensitizing (non-controlling) values. The inputs which have an equal or greater upper bound on their delays are of indeterminate value when an event on f_{i-1} is to propagate through f_i. To be conservative, their values must be assumed to be the sensitizing values. Thus, to determine an upper bound on the delay of a gate input one finds a set of paths which contain all the paths down which an event travels under input vector c.

An algorithm for viability analysis is given in [75]. The first step in computing the longest viable path through a network is to formulate the functions associated with the viability conditions. The second step is the interleaving of the computation of the viability function with the path-tracing procedure in such a way that the overhead of function computation is minimized to a large extent. In particular, it is shown that computation of the viability function can be interleaved with the most efficient known path-tracing algorithm in such a way that exactly one function computation is required for each partial path that is traced. This procedure, while as efficient as possible from a path-tracing viewpoint, nevertheless results in very large functions being generated for analysis [74]. Even though the best known path-tracing algorithm to perform viability analysis is provided in [75], an efficient implementation has been lacking because the Boolean function that is derived is formidable to compute and represent.

The version of viability analysis restricted to networks of simple gates is provided in Section 2.3.1. The formal definition for the viability function given in [76, 75] is stated in abstract functional terms, which requires the machinery of an operator calculus. It is provided here for the sake of completeness.

Recall that $\mathcal{P}_{g,t}$ denotes the set of paths of length $\geq$ t which terminate at gate g.

Definition 5.3.1 *The* **Boolean difference** *of f with respect to variable g is $f_g \oplus f_{g'}$ and* *is denoted* $\frac{\partial f}{\partial g}$.

The Boolean difference $\frac{\partial f}{\partial g}$ represents all the conditions under which f is sensitive to a change in value of g. This is the generalization of the static sensitization condition of f with respect to g (*c.f.* Definition 2.3.9).

Definition 5.3.2 *The* **viability function** ψ_P *of a path* $P = \{f_0, f_1, ..., f_m\}$ *is defined as follows*

$$\psi_P = \prod_{i=0}^{m} \psi_P^{f_i} \tag{5.5}$$

$$\psi_P^{f_i} = \sum_{U \subseteq S(f_i,P)} \{S_U \frac{\partial f_i}{\partial f_{i-1}} \prod_{g \in U} \psi^{g,\tau_{i-1}^P}\} \tag{5.6}$$

$$\psi^{g,t} = \sum_{Q \in \mathcal{P}_{g,t}} \psi_Q. \tag{5.7}$$

5.3.2 Dynamic programming algorithm for viability analysis

All the timing analysis algorithms in the literature are based on constructing IO-paths by enumeration of partial paths [9, 14, 41, 75]. While in the case of the procedures described in [9, 14, 41], the sensitization condition depends only on the path being extended, in the case of viability analysis, the viability function of a path is not only a function of the path itself, but also of the viability function of all adjoining paths at least as long. Correct computation of the viability function requires that this function be computed for each side input for each node on the candidate path as that node is encountered. This recursive path-tracing is too expensive and a more efficient dynamic programming algorithm is suggested in [75] to avoid recursive computations of path viability function. Refer to [75, 74] for a complete explanation of the algorithm.

In order to properly maintain the correct value of $\psi^{g,t}$ it is necessary that all paths longer than t be traced before any path of length t is traced through f_i. It is shown in [76] that any path through f_i is of length $> t$ if and only if it has *esperance* (The esperance of a path is the length of its longest remaining extension.) greater than a path of length t through f_{i-1}, and hence is traced first by the best-first procedure. This fact allows us to drop the path length superscript (t and τ_{i-1}^P in Definition 5.3.2) from the viability functions. Henceforth, ψ^g and $\mathcal{P}_{g,-}$ are used instead of ψ^{g,τ_{i-1}^P} and $\mathcal{P}_{g,t}$ respectively.

Viability computations were extremely expensive in the implementation reported in [75], due to two principal difficulties.

- There was no good procedure implemented for determining whether a Boolean function is satisfiable; and
- In general, the representation of $\psi_P^{f_i}$ is very large. Indeed, one can see from examination of the equation for $\psi_P^{f_i}$, that computation of a power-set sum is required.

In the next subsection, the path-recursive function for viability analysis is manipulated to ensure that the function $\psi_P^{f_i}$ is of size equal to the size of f_i, and that ψ^g is of size at most equal to the number of true paths in $\mathcal{P}_{g,-}$.

5.3.3 Linearizing the viability equations

Careless computation of the viability functions can lead to exceptional running times since the functions $\psi_P^{f_i}$ and ψ^g produce very large functions. Therefore, efficient computation and representation of these functions is a necessity. It is immediate that the function $\psi_P^{f_i}$ in Definition 5.3.2 is path-recursive. Unfortunately, this function is not linear path recursive, due to the power-set sum $\sum_{U \subseteq S(f_i,P)} \{S_U \frac{\partial f_i}{\partial f_{i-1}} \prod_{g \in U} \psi^{g,\tau_{i-1}^P}\}$. However, the function is made linear path-recursive through the following two steps:

1. Restrict the network to be composed of simple gates only; in this case, the viability definition becomes:

   Definition 5.3.3 *A path $P = \{f_0, f_1, ..., f_m\}$ is* **viable** *under an input cube c if, at each node f_i for each $g_j \in S(f_i, P)$, either:*

   - $g_j(c) = I(f_i)$ *; or*
   - $\exists Q_j \in \mathcal{P}_{g_j,\tau_{i-1}^P}$ *such that Q_j is viable under c*

   (5.6) is now rewritten as:

$$\psi_P^{f_i} = \sum_{U \subseteq S(f_i,P)} \{\prod_{g \in U} \psi^{g,\tau_{i-1}^P} \prod_{h \notin U} (h = I(f_i))\}. \tag{5.8}$$

2. Apply Theorem 5.2.1 to the revised equation to get,

$$\psi_P^{f_i} = \prod_{g \in S(f_i,P)} \{\psi^{g,\tau_{i-1}^P} + (g = I(f_i))\} \tag{5.9}$$

which is plainly linear.

The computation of the sensitization functions using a multilevel representation is now explained.

Substitute a new variable U^{g,f_i} for $(g = I(f_i)) + \psi^g$, and write

$$\psi_P^{f_i} = \prod_{g \in S(f_i,P)} U^{g,f_i} \tag{5.10}$$

where

$$U^{g,f_i} = (g = I(f_i)) + \psi^g \tag{5.11}$$

and (as in Definition 5.3.2)

$$\psi^g = \sum_{Q \in \mathcal{P}_{g,-}} \psi_Q. \tag{5.12}$$

Note that each $\psi_P^{f_i}$ is now at most the size of f_i, and that each new node U^{g,f_i} is of constant size. This has been obtained using two new functions at each connection of the network. However, the number of new functions is linear in the size of the network.

The second step in the transformation is the compact representation of ψ^g. Maintain a separate node for each ψ_P and add that as a *symbol* to ψ^g as path P is traced through g. This transformation has the effect of ensuring that ψ^g has at most one cube (each cube of size 1) for each true path in $\mathcal{P}_{g,-}$.

The third step in the transformation is the efficient representation of ψ_P. Since a separate ψ_P is to be maintained for each partial path P, let $P = \{Q, f\}$ for Q a partial path. Then ψ_Q must have already been calculated and available as a separate node before ψ_P is computed. Hence:

$$\psi_P \equiv \psi_Q \psi_P^f \tag{5.13}$$

i.e., each path sensitization function is a two-input AND gate.

This completes the transformation of an exponentially-sized function into a linear number gates. Note that this has been accomplished by an increase in the number of nodes in the path analysis network, which means a larger number of nodes to form BDD's for in the case of BDD justification, and a much larger problem space to explore for the SAT program. Experimental results indicate that this tradeoff is a good one on the benchmark circuits.

The complete algorithm is shown in Figures 5.2 and 5.3. In the main procedure shown in Figure 5.2, a *priority queue* data structure is used to retrieve a partial path (among those on the queue) that has the maximum esperance [76]. Initially the primary inputs are placed on the priority queue; each has its viability function equal to 1. When a partial path P is retrieved from the queue, if it is satisfiable the path is extended through its fanout connection with the longest unexplored path length (or esperance) to obtain a new path Q. The function U^{f_{i-1},f_i} is updated according to Equation 5.11. The viability function of Q is computed according to Equations 5.10 and 5.13. Finally the esperance of P is updated before placing it back on the queue. This allows the other unexplored (and shorter or equal) extensions from P to be considered at a later time, if necessary. Note that the functions U^{f_{i-1},f_i} and ψ_P are nodes that are created in the multilevel network used to store the path-sensitization conditions (*c.f.* Figure 5.1).

5.3.4 Complex gates

One assumption made throughout this discussion is that the gates dealt with are simple gates. Only these gates have controlling vs. non-controlling values for each input. In [75], complex gates are dealt with through the use of functional generalizations of controlling vs. non-controlling values, namely the Boolean difference and associated operators [76]. However, the data structure representations introduced have relied explicitly on the simple gate model of networks. In order to handle complex gates properly, reference is made to the use of the macro-expansion operation reported in [76]. A macro-expansion of a gate g is its realization as a network of gates, with one distinguished fanout gate. Each internal gate and wire in the network realizing g has delay 0, and the output gate of the macro-expanded network has delay equal to the delay of g.

An example of the macro-expansion operation is shown in Figure 5.4 [74]. Two macro-expansion networks of a complex gate f computing the function $b(a+c)$ with a delay of 2 are shown; in circuit (1) the AND gates are assigned delays of *0* and the OR gate is assigned delay 2 (in circuit (2) the OR gate has delay *0* and the AND gate has delay 2). As pointed out in [74], for every viable path through the complex gate, there exists a viable path with the same delay in the macro-expansion network of the complex gate. However, the converse does not hold. For example, if b is late-arriving, the path $\{a, x, f\}$ is viable when $c = 1$ in circuit (1) of Figure 5.4. For the complex gate however, the path from a through f is not viable, using the computation of Equations 5.5 through 5.7.

The macro-expansion theorem [76] guarantees that viability analysis on any chosen macro-expansion of the network η will not underestimate the length of the longest

```
find_longest_true_path($\eta$) {
    /* $U^{g,f}$ represents sensitizing condition for connection from $g$ to $f$ */
    $U^{g,f} = I(f)$ for each fanin $g$ of each gate $f$ in $\eta$.
    initialize queue to primary inputs of circuit.
    Foreach path $P$ on queue
        $\psi_P = 1$.
    while (path $P$ = pop(queue)) {
        /* satisfiability check */
        if ($\psi_P \neq 0$) {
            $f_{i-1}$ is last gate of $P$.
            if ($f_{i-1}$ is a primary output)
                return "True path is" $P$.
            /* $f_i$ is a fanout of $f_{i-1}$ with longest unexplored path.
            sensitizing condition for connection from $f_{i-1}$ to $f_i$ is
            increased (c.f. Equations 5.11 and 5.12) */
            $U^{f_{i-1},f_i} = U^{f_{i-1},f_i} + \psi_P$.
            $Q$ = {$P$, $f_i$}.
            /* Store $\psi_Q$ for later use */
            $\psi_Q$ = viability_function($P$, $f_i$).
            if there are more fanouts of $f_{i-1}$ {
                update esperance of $P$.
                insert $P$ on queue.
            }
        }
    }
Return "No true paths".
}
```

Figure 5.2: Dynamic programming procedure to find longest viable path

```
/* path-recursive computation of viability condition */
viability_function($P$, $f_i$) {
    $f_{i-1}$ is the last gate of $P$.
    $\psi_P^{f_i}$ = 1.
    Foreach fanin $g$ of $f_i$ {
        if ($g \neq f_{i-1}$) {
            /* Computation of Equation 5.10 */
            $\psi_P^{f_i} = \psi_P^{f_i} * U^{g,f_i}$.
        }
    }
    return $\psi_P * \psi_P^{f_i}$.
}
```

Figure 5.3: Computation of the viability function

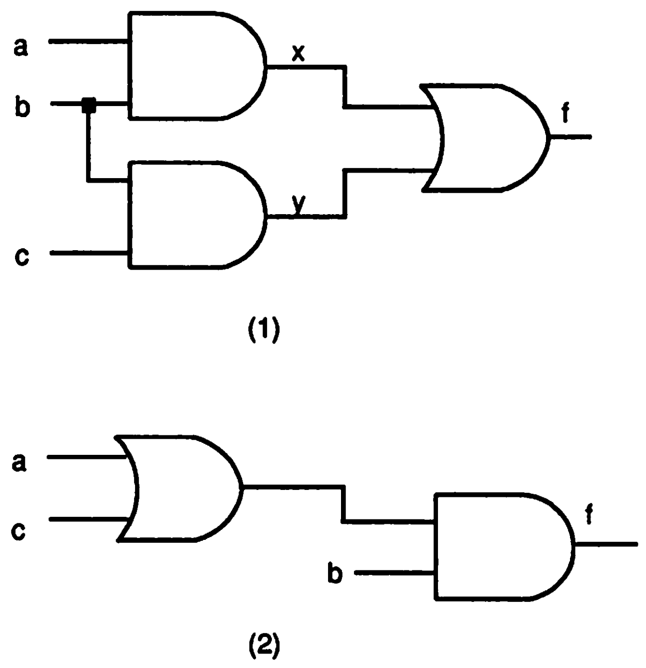


Figure 5.4: Macro-expansion operation for the function $f = b(a + c)$

viable path. It may, however, possibly overestimate the length, depending upon the extent to which internal glitches can occur inside the implementation of gates. However, *one* macro-expansion is guaranteed to report the correct longest viable path. In the example of Figure 5.4, viability analysis on circuit (2) yields the same result as on the complex gate.

Consider an arbitrary static CMOS gate: in general, it consists of an arbitrary series/parallel pull-down structure of n-transistors and a dual pull-up structure of p-transistors. It has long been known that this structure maps onto a *factored-form* model of gates, where the gate itself is considered the inverted form of a small network of AND and OR gates. Now, if this equivalent factored form of a gate is considered as its macro-expanded form, then it is clear that this network will glitch only when the gate it represents physically glitches; it is trivial to show (from induction on the inputs of the original network) that the delay under any cube c to the macro-expanded network for g is equal to the delay under c to g in η. Thus, in the example of Figure 5.4, circuit (2) has exactly the same delay as that of the complex gate $f = b(a + c)$; the same cannot be asserted for the circuit (1) realization of f.

5.3.5 Timing analysis results with path-tracing

A timing analysis system based on the ideas of the preceding discussion is implemented in the SIS system [18, 102], and results on some examples are shown in Tables 5.1 and 5.2. Delay estimation is performed using three methods: static sensitization (column labeled *Static*), the Brand and Iyengar approach [14] (column labeled *Brand*) and viability analysis (column labeled *Viable*). The sensitization functions of connections used by the first two methods remain unchanged during the path-tracing algorithm and each function is represented as a multilevel connection of simple gates as described in Section 2.3.1. The sensitization function for viability analysis is similarly represented by a multilevel function, which is updated as the path tracing is being performed (*c.f.* Figures 5.2 and 5.3).

Table 5.1 shows the results of using BDD's to solve the Boolean satisfaction problem required during timing analysis. The examples are from the ISCAS benchmark suite and the run times are for a DEC 5000. Equal arrival times at all inputs and a unit delay for each gate in the circuit are assumed. *C6288* can not be analyzed since the BDD's for this multiplier circuit cannot be constructed [21]. Similarly, *C3540* does not complete since the BDD's for some of the intermediate functions required during timing analysis are too large. In the case of *C2670*, the Brand and Iyengar approach overestimates the delay of the circuit

Name	Delay Estimate				CPU secs.		
	Longest	Static	Brand	Viable	Static	Brand	Viable
C1908	40.0	37.0	37.0	T	406	364	-
C2670	32.0	30.0	31.0	30.0	134	91	1025
C3540	47.0	M	M	M	-	-	-
C5315	49.0	47.0	47.0	47.0	169	25	205
C6288	124.0	M	M	M	-	-	-
C7552	43.0	42.0	42.0	42.0	22	18	23

T : Did not finish in 10 hours

M : out of memory

Table 5.1: BDD based path-tracing timing analysis on ISCAS circuits

Name	Delay Estimate				CPU secs.		
	Longest	Static	Brand	Viable	Static	Brand	Viable
C1908	40.0	37.0	37.0	T	97	1223	-
C2670	32.0	30.0	31.0	T	415	74	-
C3540	47.0	46.0	46.0	46.0	18	15	24
C5315	49.0	47.0	47.0	47.0	12	11	20
C6288	124.0	T	T	T	-	-	-
C7552	43.0	42.0	42.0	42.0	11	14	11

T : Did not finish in 10 hours

Table 5.2: SAT based path-tracing timing analysis on ISCAS circuits

while viability analysis takes significantly longer to get a better estimate.

In Table 5.2, timing analysis on the ISCAS circuits is performed using the SAT algorithm, described in Appendix A, to check the sensitization condition. The static sensitization and the Brand and Iyengar approaches for timing analysis complete on all but the circuit *C6288*. The analysis does not complete in 10 hours due to the number of false paths in this circuit. The SAT program checks at least 300,000 sensitization functions for this circuit.

For the circuit *C3540*, while the BDD based approach fails due to the amount of memory required, the SAT approach yields an answer in very short time.

On the other hand, for circuits *C1908* and *C2670*, the SAT approach fails to complete in 10 hours. The BDD approach is significantly faster in these cases. Note that the both the SAT and the BDD programs see identical sensitization functions. The discrepancy in running times is explained as follows. The BDD based approach constructs a BDD for each new function and it does not recompute the BDD's of existing functions. In contrast, each satisfiability call in the SAT approach is independent of the other calls. In the case of *C1908* and *C2670*, the processing time for each of the large number of satisfiability calls is very large, leading to the long running time. This mainly arises due to the number of implications and backtrackings that are made in each satisfiability call. From these examples, it may be summarized that the BDD approach is very effective if the BDD's for all the functions can be compactly represented. Another reason BDD's are more effective than the SAT approach on the path-tracing approach is that many intermediate functions are the same from one satisfiability call to the next. Thus, creating the BDD once for each of these functions is advantageous compared to determining a single satisfying assignment (possibly different each time) on each SAT call. In the cases where BDD's cannot be stored in the allocated memory, resort must be made to the SAT approach. Neither approach completes when the number of false paths is very large.

5.3.6 Performance optimization and false paths

Tables 5.3 and 5.4 show timing analysis results for three carry-skip adder circuits and five optimized circuits, all of which exhibit false paths. All other unoptimized and optimized circuits experimented with have at least one longest path that is true, though there may be some false paths.

The adder circuits used in timing analysis are not optimized. The three examples illustrate the large discrepancy between the length of the topologically longest path and the longest sensitizable path. The BDD approach significantly outperforms the SAT approach on the adder examples for viability analysis. Besides the large number of SAT calls required, the SAT program also performs poorly with the current heuristics (Appendix A) on each SAT call.

The MCNC circuits are optimized using the Boolean script in SIS [18, 102]. The SAT based approach does not complete on the adder examples for viability analysis. This occurs because the large number of false paths leads to a huge number of intermediate sensitization functions. Thus, each successive SAT call sees an increase in the size of the satisfiability problem, which leads to a dramatic slowdown. Since the sensitization func-

Name	Delay Estimate				CPU secs.		
	Longest	Static	Brand	Viable	Static	Brand	Viable
csa 16.2	50.0	24.0	24.0	24.0	11	17	65
csa 16.4	42.0	24.0	24.0	24.0	8	10	35
csa 32.4	82.0	32.0	32.0	32.0	90	145	656
5xp1	11.0	9.0	9.0	9.0	1	1	1
bw	20.0	14.0	14.0	14.0	2	4	14
des	15.0	13.0	13.0	13.0	20	35	40
misex1	9.0	7.0	7.0	7.0	1	1	1
rot	19.0	17.0	18.0	17.0	10	7	10
bw.sp.in	29.0	22.0	22.0	22.0	3	7	22
bw.sp.out	27.0	22.0	22.0	22.0	3	3	11
bw.kms	18.0	18.0	18.0	18.0	1	1	1
bw.kms.sp	15.0	15.0	15.0	15.0	2	1	2

bw.sp.in : Circuit before timing optimization (2-input NAND gates)

bw.sp.out : Circuit after timing optimization (2-input NAND gates)

bw.kms : Circuit after KMS algorithm and before timing optimization

bw.kms.sp : Circuit after KMS algorithm and timing optimization

Table 5.3: BDD based path-tracing timing analysis on MCNC circuits

tions do not change for the static sensitization or the Brand and Iyengar approach, the SAT program yields similar results to the BDD program in these cases.

The most interesting circuit among the optimized MCNC circuits is *bw*, which highlights the need for accurate timing analysis. The true delay of the initial area optimized circuit before timing optimization is 22.0 instead of 29.0. Due to the nature of the technology independent timing optimization program, note that this circuit is composed of only 2-input NAND gates. The *speed_up* command in SIS is used to improve the delay of the circuit. While the depth of the circuit decreases to 27.0, the true delay remains 22.0. In effect, *speed_up* provides no improvement in circuit performance since it enhances only false paths. This example highlights the importance of false path analysis for combinational performance optimization techniques. By using the KMS algorithm of Chapter 3, the initial optimized *bw* circuit is first converted to an equivalent circuit with the same (or less) computed delay (circuit *bw.kms*). For this circuit, *speed_up* guarantees a performance enhancement since there are no false paths. This is illustrated by the circuit *bw.kms.sp* in Tables 5.3 or 5.4. In fact, without the use of the KMS algorithm, a circuit with a delay of 15.0 cannot be achieved

Name	Delay Estimate				CPU secs.		
	Longest	Static	Brand	Viable	Static	Brand	Viable
csa 16.2	50.0	24.0	24.0	24.0	31	30	T
csa 16.4	42.0	24.0	24.0	24.0	20	19	T
csa 32.4	82.0	32.0	32.0	32.0	232	202	T
5xp1	11.0	9.0	9.0	9.0	1	1	2
bw	29.0	25.0	26.0	25.0	6	12	3027
des	15.0	13.0	13.0	13.0	24	21	1140
misex1	9.0	7.0	7.0	7.0	1	1	9
rot	19.0	17.0	18.0	17.0	6	5	18
bw.sp.in	29.0	22.0	22.0	22.0	3	7	2551
bw.sp.out	27.0	22.0	22.0	22.0	5	6	735
bw.kms	18.0	18.0	18.0	18.0	1	1	1
bw.kms.sp	15.0	15.0	15.0	15.0	1	1	3

T : Did not finish in 10 hours

bw.sp.in : Circuit before timing optimization (2-input NAND gates)

bw.sp.out : Circuit after timing optimization (2-input NAND gates)

bw.kms : Circuit after KMS algorithm and before timing optimization

bw.kms.sp : Circuit after KMS algorithm and timing optimization

Table 5.4: SAT based path-tracing timing analysis on MCNC circuits

even after applying the strongest options available in the *speed_up* command on the circuit *bw.sp.in*.

Note the significantly larger run time consumed by viability analysis compared to static sensitization for the example *bw*. The satisfiability is checked on each partial path that is enumerated during path-tracing. Static sensitization incorrectly estimates a partial path (call it P) to be false when it is in fact true. It then determines that another path of the length 22.0 is true, which is returned as the delay estimate. However, viability analysis correctly determines that the partial path P is true and explores several extensions of paths through P before returning a correct delay estimate of 22.0. The Brand and Iyengar algorithm yields the true delay estimate for this example. However, it overestimates the delay in the case of the circuit *rot*. Of course, if the path sensitization conditions are only checked for satisfiability only on IO-paths (paths terminating at primary outputs) rather than on partial paths, viability analysis explores fewer or equal number of paths than static sensitization, since the latter implies the former. In this case, the run time for viability is comparable to that of static sensitization or the Brand and Iyengar procedure.

5.4 Timing analysis without path-tracing

Since the optimal path-tracing procedure for timing analysis requires tracing each false path longer than the longest true path, these algorithms break down when there are a great many false paths as long or longer than the longest true path. In vanilla best-first-search path-tracing, this problem is alleviated by justifying path sensitization functions at each fanout point along the path; in this manner, the branches of a false trunk are never explored. Nevertheless, this approach has its limits. As first proposed in [38], ideally, one would not like to trace paths at all. The objective in timing analysis is principally to determine when a primary output f achieves its final value, not to determine down which paths the final event travels. Nonetheless, clearly in performance optimization one wishes for precisely this latter information, and for that application path-tracing is still the only appropriate approach. In the following, the path recursive paradigm is applied to delay estimation using viability analysis but without performing explicit path-tracing.

Define the input vectors under which a node f settles to a final value no earlier than t as the Boolean function $\chi^{f,t}$, i.e.

$$\chi^{f,t}(c) = 1 \Leftrightarrow f \text{ settles no earlier than } t \text{ when } c \text{ is applied to the primary inputs.}$$

Clearly, the delay of a circuit is equal to the greatest t such that $\chi^{F,t} \neq 0$ for some primary output F. From the viability theory, it is clear that:

Lemma 5.4.1

$$\chi^{f,t} = \sum_{P \in \mathcal{P}_{f,t}} \psi_P$$

Proof Let $c \subseteq \chi^{f,t}$. By the viability definition (*c.f.* Definition 5.3.2), there is some path P of length $\geq t$, terminating in f, viable under c. By the definition of $\mathcal{P}_{f,t}$, $P \in \mathcal{P}_{f,t}$, and since P is viable under c, $c \subseteq \psi_P$. Conversely, if $c \subseteq \sum_{P \in \mathcal{P}_{f,t}} \psi_P$, then $c \subseteq \psi_P$ for some $P \in \mathcal{P}_{f,t}$. P is viable under c, of length $\geq t$, and terminates in f, and hence f must settle to a final value no earlier than t when c is applied. Consequently, $c \subseteq \chi^{f,t}$. ∎

This lemma is not in and of itself extremely useful in the correct derivation of $\chi^{f,t}$; direct application leads back to path-tracing. Fortunately, $\chi^{f,t}$ can be expressed directly in terms of the χ functions of the inputs to f[2].

Theorem 5.4.1

$$\chi^{f,t} = \sum_{g \in FI(f)} \chi^{g,t-d(f,g)} \sum_{U \subseteq S(f,g)} S_U \frac{\partial f}{\partial g} \prod_{h \in U} \chi^{h,t-d(f,g)} \tag{5.14}$$

Proof Follows directly from the definition of viability (*c.f.* Definition 5.3.2). A path terminating in f of length $\geq t$ is viable under cube c if and only if:

1. there exists a viable path of length $\geq t - d(f,g)$ to some input g of f under c; and,
2. each of the remaining inputs of f either is at a statically sensitized value (non-controlling value for a simple gate), or terminates a viable path of length $\geq t - d(f,g)$ under c.

The first condition is equivalent to asserting $\chi^{g,t-d(f,g)}$ and the second condition is equivalent to $\sum_{U \subseteq S(f,g)} S_U \frac{\partial f}{\partial g} \prod_{h \in U} \chi^{h,t-d(f,g)}$. ∎

[2]In subsequent discussion $S(f,g)$ denotes the inputs of gate f other than fanin g. While this is slightly different from previous usage, there is no ambiguity introduced.

In Theorem 5.4.1, for each $g \in FI(f)$ for which $\chi^{g,t-d(f,g)}$ holds, a different time $t-d(f,g)$ appears in the term $\sum_{U \subseteq S(f,g)} S_U \frac{\partial f}{\partial g} \prod_{h \in U} \chi^{h,t-d(f,g)}$. This may lead to a potential inefficiency in the representation of $\chi^{f,t}$, as illustrated by the following example.

Given inputs g_1, g_2, and g_3 for gate f,

$$\begin{aligned}\chi^{f,t} = \; & \chi^{g_1,t-d(f,g_1)} \sum_{U \subseteq \{g_2,g_3\}} S_U \frac{\partial f}{\partial g} \prod_{h \in U} \chi^{h,t-d(f,g_1)} + \\ & \chi^{g_1,t-d(f,g_2)} \sum_{U \subseteq \{g_1,g_3\}} S_U \frac{\partial f}{\partial g} \prod_{h \in U} \chi^{h,t-d(f,g_2)} + \\ & \chi^{g_1,t-d(f,g_3)} \sum_{U \subseteq \{g_1,g_2\}} S_U \frac{\partial f}{\partial g} \prod_{h \in U} \chi^{h,t-d(f,g_3)}.\end{aligned}$$

Clearly, if each $d(f,g_i)$ for $i = 1,2,3$ is distinct, then no further factorization of the above expression is possible. In fact, three different functions $\chi^{g_1,\tau}$ for $\tau = \{t-d(f,g_1), t-d(f,g_2), t-d(f,g_3)\}$ are required. Similarly, three different functions are also required at g_2 and g_3. Thus, for a gate f with i inputs, $\chi^{g_j,t}$ for each input gate g_j of f may potentially have to be computed for i different times. This implies that the number of functions required during the computation may easily become exponential in number. Fortunately, a more efficient form of the computation exists when symmetric complex gates are being considered[3].

The next theorem requires a technical lemma stated in [76]. It is reproduced here without proof for ease of reference.

Definition 5.4.1 *A function f is said to be* **symmetric** *in some set of variables U if, for every permutation of U, there exists a phase assignment to the variables in U such that f is invariant.*

Lemma 5.4.2 *If f is symmetric in a set of variables U then for every $V \subseteq U$ where $|V| \geq 2$ and $x, y \in V$:*

$$S_{V-\{y\}} \frac{\partial f}{\partial y} = S_{V-\{x\}} \frac{\partial f}{\partial x}$$

Note the subtle difference in the term $\prod_{h \in U} \chi^{h,t-d(f,h)}$ in the next theorem from the term $\prod_{h \in U} \chi^{h,t-d(f,g)}$ in Theorem 5.4.1.

Theorem 5.4.2

$$\chi^{f,t} = \sum_{g \in FI(f)} \chi^{g,t-d(f,g)} \sum_{U \subseteq S(f,g)} S_U \frac{\partial f}{\partial g} \prod_{h \in U} \chi^{h,t-d(f,h)} \tag{5.15}$$

[3]The case for asymmetric complex gates is handled by using the macro-expansion operator discussed in Section 5.3.4 to obtain an equivalent network of symmetric gates. There may be some penalty due to an increase in the number of paths, but the estimate on the true delay remains a correct upper bound.

Proof Let $c \subseteq \chi^{f,t}$.

By Lemma 5.4.1, there exists some $P \in \mathcal{P}_{f,t}$ such that P is viable under c. Let V be the subset of $FI(f)$, such that:

$$V = \{h | h \in FI(f) \text{ and } \exists P_h = \{P'_h, h, f\}, P_h \in \mathcal{P}_{f,t}, c \subseteq \psi_{P_h}\}$$

Since $c \subseteq \chi^{f,t}$, the set V is non-empty. Choose $h_0 \in V$ for which $t - d(f, h_0)$ is maximum. By the definition of viability,

$$c \subseteq \mathcal{S}_U \tfrac{\partial f}{\partial h_0} \text{ for } U = \{h | h \in V - \{h_0\}, t - d(f,h) = t - d(f,h_0)\}.$$

Since there is a viable path under c of length $\geq t - d(f, h_0)$ terminating at h_0, using Theorem 5.4.1:

$$c \subseteq \chi^{h_0, t-d(f,h_0)} \sum_{U \subseteq S(f,h_0)} \mathcal{S}_U \tfrac{\partial f}{\partial h_0} \prod_{h \in U} \chi^{h, t-d(f,h_0)}.$$

By definition, for a node f and any $t \geq 0$ and $\epsilon \geq 0$, $\chi^{f,t} \subseteq \chi^{f,t-\epsilon}$. For each $h \in V - \{h_0\}$, $t - d(f,h_0) \geq t - d(f,h)$. Hence $\chi^{h,t-d(f,h_0)} \subseteq \chi^{h,t-d(f,h)}$. Consider the term

$$\chi^{h_0, t-d(f,h_0)} \sum_{V-\{h_0\}} \mathcal{S}_{V-\{h_0\}} \tfrac{\partial f}{\partial h_0} \prod_{h \in V-\{h_0\}} \chi^{h, t-d(f,h)}.$$

Since $c \subseteq \chi^{h,t-d(f,h_0)}$, it follows that $c \subseteq \chi^{h,t-d(f,h)}$ $\forall h \in V - \{h_0\}$. For $U = \{h | h \in V - \{h_0\}, t - d(f,h) = t - d(f,h_0)\}$,

$$c \subseteq \chi^{h_0, t-d(f,h_0)} \sum_{U \subseteq S(f,h_0)} \mathcal{S}_U \tfrac{\partial f}{\partial h_0} \prod_{h \in U} \chi^{h, t-d(f,h)}.$$

Thus, c is contained in at least one term in the right hand side of 5.15. Hence,

$$\chi^{f,t} \subseteq \sum_{g \in FI(f)} \chi^{g, t-d(f,g)} \sum_{U \subseteq S(f,g)} \mathcal{S}_U \tfrac{\partial f}{\partial g} \prod_{h \in U} \chi^{h, t-d(f,h)}.$$

Now, let

$$c \subseteq \sum_{g \in FI(f)} \chi^{g, t-d(f,g)} \sum_{U \subseteq S(f,g)} \mathcal{S}_U \tfrac{\partial f}{\partial g} \prod_{h \in U} \chi^{h, t-d(f,h)}.$$

Choose any term of the first sum containing c, i.e.,

$$c \subseteq \chi^{g, t-d(f,g)} \sum_{U \subseteq S(f,g)} \mathcal{S}_U \tfrac{\partial f}{\partial g} \prod_{h \in U} \chi^{h, t-d(f,h)}.$$

For each $h \in U$, there exists a path P_h, viable under c, of length $\geq t - d(f,h)$ terminating in h. Similarly, there exists a path P_g, viable under c, of length $\geq t - d(f,g)$ terminating in g. Let $V = U + \{g\}$. For each element h of V, let P_h be the path of least length $\geq t - d(f,h)$ viable under c, terminating in h. Let P_k be any path of minimum length among these paths, and let P_k terminate in $k \in V$. By the symmetry of f (*c.f.* Lemma 5.4.2), $c \subseteq S_{V-\{g\}}\frac{\partial f}{\partial g}$ implies that $c \subseteq S_{V-\{k\}}\frac{\partial f}{\partial k}$, and each $h \in V - \{k\}$ terminates a path of length at least $|P_k|$, viable under c. Hence the path $\{P_k, f\}$ is viable under c and of length at least t, and $c \subseteq \chi^{f,t}$. Hence,

$$\chi^{f,t} \supseteq \sum_{g \in FI(f)} \chi^{g,t-d(f,g)} \sum_{U \subseteq S(f,g)} S_U \frac{\partial f}{\partial g} \prod_{h \in U} \chi^{h,t-d(f,h)}.$$

■

Though this function is expressed in terms of the smoothed Boolean difference to properly handle the case of symmetric, complex gates, it may be linearized in the case of simple gates using Theorem 5.2.1.

Equation (5.15) for simple gates is written as:

$$\chi^{f,t} = \sum_{g \in FI(f)} \chi^{g,t-d(f,g)} \sum_{U \subseteq S(f,g)} \{\prod_{h \in U} \chi^{h,t-d(f,h)} \prod_{k \in S(f,g)-U} (k = I(f))\} \tag{5.16}$$

Using the transformation of Theorem 5.2.1 this becomes:

$$\chi^{f,t} = \sum_{g \in FI(f)} \{\chi^{g,t-d(f,g)} \prod_{h \in S(f,g)} (\chi^{h,t-d(f,h)} + (h = I(f))\} \tag{5.17}$$

Since $\chi^{g,t-d(f,g)}(\chi^{g,t-d(f,g)} + (g = I(f)))$ is equal to $\chi^{g,t-d(f,g)}$, the product term on the right hand side can be made to include the term $(\chi^{g,t-d(f,g)} + (g = I(f)))$ without changing the function. This leads to the further simplification:

$$\chi^{f,t} = \sum_{g \in FI(f)} \chi^{g,t-d(f,g)} \prod_{h \in FI(f)} (\chi^{h,t-d(f,h)} + (h = I(f)) \tag{5.18}$$

Note that while each function computed at any gate f is linear in the size of the gate, a function is created for each different time t for which $\chi^{f,t}$ is required. This may be a problem if the number of functions computed for each different time is large. Figure 5.5 is an illustration of this phenomenon. Assume that each gate in the figure has unit delay. To determine if a viable path of length greater than or equal to 5 exists, gate $g7$ in the figure requires the computation of χ^{g7}, t at two different times, namely $t = 2$ and $t = 4$. This is

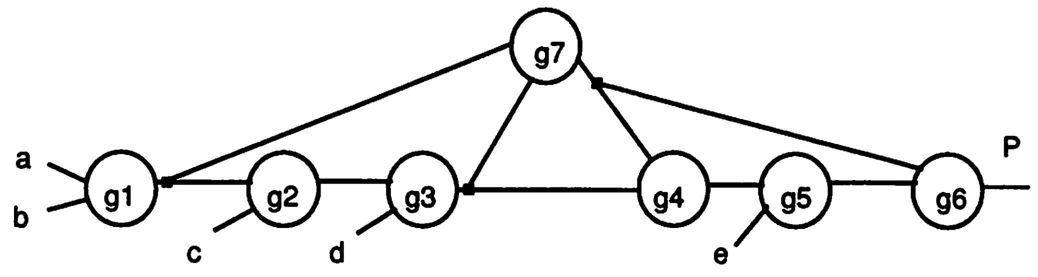


Figure 5.5: Example: Multiple function computations at a node during timing analysis

because there is a path of length 2 from primary inputs a and b up to the output of $g7$, and a path from $g7$ to P of length 3; there is also a path of length 4 from a and b to the output of $g7$ and there exists a path of length 1 to P from $g7$. This occurs because of reconvergent paths of different lengths. If viability is being checked for the longest path length L, then only a single function is computed at each node [38]. In the remaining cases, a tight upper bound is not known. However, a loose upper bound on the number of functions required at a node when determining viability for delay T, is the number of distinct path lengths that exist between T and L [38]. The detailed algorithm is given in the next section.

5.4.1 Timing analysis algorithm without path-tracing

The algorithm for performing timing analysis based on the computation shown in Equation (5.18) is described in pseudo-code in Figures 5.6 through 5.8. There are two phases in the main algorithm shown in Figure 5.6.

The first step, shown in Figure 5.7 consists of computing the distinct times required at each node that result in path lengths $\geq T$ to the primary outputs (variable *rtimes* in the figure). This is done by performing a delay trace followed by traversals of the network in topological and reverse topological order, respectively.

The second step is the computation of $\chi^{f,t}$ at each gate f for different times t. This is done by traversing the network from primary inputs to primary outputs in topological order. The computations of Figure 5.8 directly reflect the form of Equation (5.18).

```
/* determine if there exists a viable path of length $\geq T$ in network $\eta$ */
is_path_true($\eta$, $T$) {
    /* Circuit $\eta$ has only simple gates.
    $A_f$, $R_f$, and $S_f$ are respectively the arrival time,
    required time, and slack time at gate $f$.
    $rtimes(f)$ are the different times $t$ at which $\chi^{f,t}$ is required.  */

    /* Compute $rtimes(f)$ for each node $f$.  */
    setup_times($\eta$, $rtimes$).

    /* Compute $\chi^{f,t}$ for each node $f$ at different times.  */
    $node\_list$ = list of gates in $\eta$ in topological order.
    Foreach node $f$ in $node\_list$ {
        Foreach time $t \in rtimes(f)$ {
            $\chi^{f,t}$ = chi_fn($f$, $t$).
        }
    }
    /* Check if $\chi^{po,T}$ is 1 for any primary output $po$ of $\eta$ */
    Foreach primary output $po$ of $\eta$ {
        If $\chi^{po,t} \neq 0$ {
            Return TRUE.
        }
    }
    Return FALSE.
}
```

Figure 5.6: Delay estimation using viability analysis without path-tracing

```
setup_times($\eta$, $rtimes$) {
    Foreach primary output $F$ of $\eta$ {
        $R_F = T$.
    }
    Perform a delay trace on $\eta$ to obtain $\{R_f, A_f\}$ for all nodes $f$.
    Foreach node $f$ of $\eta$ {
        $S_f = R_f - A_f$.
        $rtimes(f) = \{\}$.
    }
    /* For each node $f$, compute times $t$ at which $\chi^{f,t}$ is required */
    $node\_list$ = list of gates in $\eta$ in reverse topological order.
    Foreach node $f$ in $node\_list$ {
        If $S_f \leq 0$ {
            If $f$ is a primary output {
                $rtimes(f) = T$.
            }
            Else Foreach fanout $g$ of $f$ {
                If $(A_f + d(f,g) \geq T)$ {
                    $rtimes(f) = \{u - d(f,g) | u \in rtimes(g)\}$.
                }
            }
        }
    }
}
```

Figure 5.7: Path length calculations for timing analysis without path-tracing

```
chi_fn($f$, $t$) {
    If $S_f \leq 0$ {
        If $f$ is a primary input {
                return $1$ .
        }
        Else {
            $sum$ = 0.
            $prod$ = 1.
            Foreach fanin $g$ of $f$ {
                If $(A_f + d(f,g) \geq t)$ {
                    $sum = sum + \chi^{g,t-d(f,g)}$.
                    $cond = (g = I(f)) + \chi^{g,t-d(f,g)}$.
                    $prod = prod * cond$.
                }
                Else {
                    /* $\chi^{g,t-d(f,g)}$ is 0 */
                    $prod = prod * (g = I(f))$.
                }
            }
            return $sum$ * $prod$.
        }
    }
    Else {
        return $0$.
    }
}
```

Figure 5.8: Computation of $\chi^{f,t}$

Name	Delay Estimate		CPU
	Longest	Viability	secs.
C1908	40.0	37.0	9
C2670	32.0	30.0	22
C3540	47.0	46.0	7
C5315	49.0	47.0	10
C6288	124.0	123.0	31
C7552	43.0	42.0	7
s641	74.0	71.0	2
s713	74.0	70.0	4
s1238†	20.0	19.0	1
s9234†	29.0	28.0	4
s15850	82.0	81.0	27
s35932	29.0	26.0	308
s38417	47.0	40.0	136

†: Inverters and buffers removed from initial circuit

Table 5.5: Timing analysis without path-tracing on ISCAS circuits

The final step in Figure 5.6 is to determine if $\chi^{po,T}$ is satisfiable for any primary output po of η. This is done using either BDD's or SAT .

5.4.2 Timing analysis results without path-tracing

Tables 5.5 and 5.6 show the results of timing analysis on benchmark circuits. A unit delay through each gate in the network is assumed. The topologically longest path is indicated in the second column of the table. Viability analysis is performed without path tracing. The time required on a DEC 5000 for viability analysis is shown in the last column.

From Tables 5.5 and 5.6 it is clear that the non path-tracing algorithm outperforms the path-tracing version on every example. The primary reason for this is that the huge number of satisfiability calls in the latter approach is replaced by a single satisfiability call for each distinct path length in the former approach. The fact that each function is linear in the size of the network when no path-tracing is employed is also a significant factor for the very low running times. In the case of path-tracing, the size of the sensitization function at a node n is linear in the network size and the number of paths already explored through n.

All the results shown here use only the SAT package to determine if a function is

Name	Delay Estimate		CPU
	Longest	Viability	secs.
csa 16.2	50.0	24.0	47
csa 16.4	42.0	24.0	19
csa 32.4	82.0	32.0	260
5xp1	11.0	9.0	1
bw	20.0	14.0	4
des	15.0	13.0	7
misex1	9.0	7.0	1
rot	19.0	17.0	2

Table 5.6: Timing analysis without path-tracing on MCNC circuits

non-zero. The BDD package performs worse on this problem for two reasons. First, the time consumed in building up the BDD's is considerably longer than the time used by SAT to find a single satisfying assignment or prove the absence of any satisfying assignment. Since each satisfiability call involves completely different functions, unlike the case of the path-tracing approach, there is no advantage to building the BDD at each node that is created during the computations. Second, the BDD's cannot be built for a few of the examples. Thus, for this problem the SAT approach is preferred to the BDD approach since only a single satisfiability check is being determined. A similar speed-up in the computation times is also observed when the static sensitization and Brand and Iyengar [14] criteria are implemented without path-tracing.

5.5 Delay-fault test generation

This section describes the computations required by an algorithm to perform robust path delay-fault (RPDFT) test generation using the path-recursive function technique.

5.5.1 Hazard-free robust delay-fault testability

There have been a variety of definitions given for RPDFT over the past few years [106, 68, 36, 88]. The definition of robust path delay-fault testability considered in subsequent discussion is the strongest of all the criteria proposed up to now. A later section indicates how the conditions developed for the most stringent form of robust delay-fault testability are easily extended to the more general forms of robust delay-fault testing in the presence

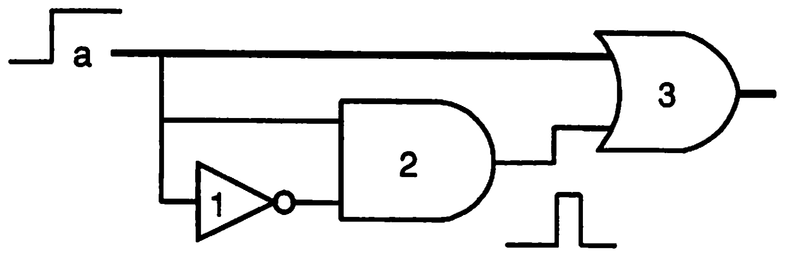


Figure 5.9: Example: Hazard problem in robust delay-fault test generation

of hazards and multiple path propagation. Definition 4.1.4 from Chapter 4 is repeated here for ease of reference.

Definition 5.5.1 *A path $P = \{f_0, f_1, ..., f_m\}$ is said to be* **single path robust delay-fault testable without hazards** *by the vector pair $< v_1, v_2 >$ if at each node f_i, $f_i(v_1) \neq f_i(v_2)$, and for each $g_j \in S(f_i, P)$:*

1. *$g_j(v_1) = g_j(v_2) = I(f_i)$; and*
2. *there is no transition on g_j.*

The vector v_2 is assumed to be applied after v_1, delayed by an amount greater than the delay of the circuit.

A necessary requirement of the above definition is that the side-inputs to gates along the path being tested must be at non-controlling values. As shown in [36], this condition is not sufficient to guarantee a valid test under arbitrary delays in the circuit. Consider the circuit in Figure 5.9 where the output of gate 2 is statically always *0*. Under a feasible combination of delays, while delay-fault testing the bold path by applying a 0-1 transition at *a*, a static hazard (shown in the figure) may be created at the output of gate 2 which invalidates the test. [36] handles this case by providing necessary and sufficient conditions for robust hazard-free delay-fault test vectors in the two-level representation (called the ENF expression) of any multilevel circuit being tested. As pointed out in [36], due to the enormous size of the two-level expression representing most multilevel circuits, using the ENF expression is not feasible for actual test generation. Thus, it is impossible to perform hazard-free delay-fault test generation by simply modifying a combinational test generator to ensure non-controlling values on side inputs.

The only deterministic and complete approaches to hazard-free delay-fault test generation published up to now have employed multiple-valued test generation [106, 68]. This approach suffers from the amount of backtracking possibly required, since each side-input to a path under test may have up to two valid values assigned that allow for a test. A detailed look at this aspect is provided in the following sub-section.

There are two assumptions made in the discussion of delay-fault test generation here. First, the description of the delay-fault testing conditions assumes an enhanced scan structure for performing the actual manufacture test. Several approaches have been suggested for delay-fault test generation in sequential circuits when reduced requirements on the scan structure are desired. These approaches are heuristic techniques that use standard stuck-fault scan structures [28], rearrange enhanced or standard scan structures to improve fault coverage [72], or use no scan structures at all [1]. These situations are beyond the scope of this thesis. It should be noted that most of the techniques described here can be extended to situations that reduce or eliminate the requirement for the enhanced scan structures.

The huge number of paths in large circuits implies that complete test generation for each path delay-fault requires an inordinate amount of time. Most approaches resolve this bottleneck by performing test generation on a selection of paths in the circuit. There are two parameters of concern; first, all connections should be included in at least one test, and secondly, critical paths must be covered. In the discussion here, path selection is not an issue. The interest in this chapter is to illustrate a technique for test generation on a given path. The results apply to any selection of paths made by the different approaches proposed in the literature. For the experimental results, when the number of paths is over 25,000, the paths are restricted to 500 longest paths from each primary input. In even larger circuits, this limit is lowered to 100 longest paths from each input.

5.5.2 Previous work

Compared to the problem of test generation for static faults, test generation for delay-faults is a relatively recent problem. A brief overview of previous work in delay-fault test generation is provided here. All the previous approaches are modifications to the algorithms for test generation for static faults. A huge volume of literature is available on the subject of stuck-fault testing, for example [90, 49, 45, 100, 64].

The concept of a robust delay-fault test that is valid under arbitrary delays on side-inputs to a path under test is introduced in [106]. An accompanying six-valued calculus is also introduced there to perform test generation for a delay-fault on a selected path in the circuit. [106] does not provide any indication of the actual algorithms used to justify the valid values on the side-inputs to the path under test. However, [68] provides a test generation algorithm based on the calculus proposed in [106]. In fact, [68] demonstrates that a five-valued calculus is sufficient to perform robust delay-fault testing. The test generation algorithm is a direct modification of the PODEM [49] algorithm, which has proved successful in stuck-fault test generation. Briefly summarized, PODEM generates a test vector by implicitly enumerating the Boolean space on the primary inputs until a test vector is found. Thus cubes instead of minterms of the primary input Boolean space are explored in determining a test vector. For the case of delay-faults, a pair of vectors is determined to be a test by similarly exploring the space on the primary inputs. However, there are up to five valid values on each primary input compared to two in the case of stuck-fault testing. This implies that a significant increase in the amount of backtracking may be required in generating a test. [68] indicates that the complexity is less than 4^n, where n is the number of primary inputs. In fact, it is demonstrated there that the complexity is $2^{(n+m)}$ where m is the number of internal connections in the circuit. This follows since each internal connection may only have up to two valid assignments [68]. However, this is still much larger than the 2^n worst-case complexity for stuck-fault test generation.

The only other published and exact approach to robust delay-fault test generation is the extension of the SOCRATES test generation program to delay-fault test generation [101]. SOCRATES is the state-of-the-art test generation program for deterministic stuck-fault test generation and redundancy removal. [100] reports that two procedures provide a remarkable reduction in the test generation effort, namely static implications and dynamic implications. The former is a pre-processing routine whereas the latter is invoked during the implicit enumeration of the search space. [101] uses a 10-valued calculus and extends the binary-value implication mechanism of [100] to this new calculus. However, no indication, other than empirical evidence, is made that this provides a reduction in the search for a delay-fault test. No bounds are provided in [101] that restrict the worst-case size of the search space to less than 10^n, for n primary inputs.

There are several proposals made for approximate delay-fault test generation in the literature. The first approach in this area is that of pseudo-random delay-fault test generation [96]. In [48], a heuristic approach to deriving robust tests from single stuck-fault tests is employed along with eight-valued delay-fault simulation to check the validity of a candidate test vector pair. This approach was shown to be better than pseudo-random delay-fault test generation but the algorithm may generate invalid (non-robust) tests which have to be detected using fault simulation. There is also no guarantee of the completeness of this technique.

Another recent proposal is made in [28], where the problem of delay-fault test generation is reduced to sequential test generation over two time frames. This is done by performing test generation over two copies of the combinational logic to generate the required transitions along the path under test, in addition to other side-inputs where transitions are permitted [28]. The vector corresponding to the first time frame is used as the first vector of the test pair, followed by the vector for the second time frame. The size of the worst-case search space in this case is 2^{2n}, for n primary inputs. While this technique appears attractive, since sequential test generation techniques can be exploited, it suffers from the drawback that it may generate non-robust delay-fault tests, since hazards are not considered at all.

In the next section a function is derived that ensures the proper dynamic behavior (no static and dynamic hazards) during test generation. The formulation allows us to restrict the test generation algorithm to the binary valued case and the Boolean satisfiability program, described in Appendix A, can be invoked to determine whether a test exists.

5.5.3 Delay-fault test generation equations

There are two properties that a hazard-free robust delay-fault test for a path P must satisfy. First, the side-inputs to P must be at non-controlling values. This can be easily ensured by the static sensitization (or Boolean difference) condition. Second, there should be no transitions or hazards on any of the side-inputs to P when the input to P is toggled during the application of the test-vector pair. The next definition describes this path-recursive function.

Definition 5.5.2 *Let T^i_j be the Boolean function (of primary inputs) which is 1 if there is no transition at the output of gate j under all possible delays when primary input i is changed from 0-1 or 1-0 and all the other primary inputs maintain their constant values.*

For primary input nodes i and j, thus, $T^i_i = 0$ and $T^i_j = 1$ for all $i \neq j$, assuming independence of the inputs.

Definition 5.5.3 *The* **consensus operator** *applied to a function f with respect to a variable x is $C_x f = f_x f_{\overline{x}}$. $C_x f$ yields the largest Boolean function contained in f which is independent of x. If X is a set of variables $\{x_0, x_1, ..., x_m\}$, $C_X f = C_{x_0} C_{x_1} ... C_{x_m} f$.*

For a simple gate f, the transition function with respect to an arbitrary primary input i is computed recursively as follows:

Proposition 5.5.1

$$T^i_f = \sum_{U \subseteq FI(f)} \{(C_U f + C_U \overline{f}) \prod_{g \in U} \overline{T^i_g} \prod_{h \notin U} T^i_h\}$$

Proof Let U be the set of fanins of f that make a transition when i is toggled. The term $C_U f$ gives the conditions under which the value of f is set to *1* independent of the values (and hence transitions) on the inputs in U. Similarly, the term $C_U \overline{f}$ specifies the conditions under which the value of f is set to *0* independent of the values (and hence transitions) on the inputs in U. Hence, the condition that there will be no transition on the output of the gate f, under all possible delays, is given by $(C_U f + C_U \overline{f})$. The term $\prod_{g \in U} \overline{T^i_g} \prod_{h \notin U} T^i_h$ indicates that the fanins in U make a transition when i is toggled and the other fanins of f remain constant. Summing over all such subsets (in general, an exponential number in the number of fanin of f), results in the only conditions under which f does not make a transition under arbitrary delays when input i is toggled. ■

Having described both conditions for a delay-fault test, the Boolean function, for which each satisfying vector yields a robust delay-fault test, is now defined.

Theorem 5.5.1 *The path $P = \{f_0, f_1, ..., f_m\}$ is hazard-free robust delay-fault testable (for both transitions along the path) if and only if there is a input assignment that satisfies $\mathcal{R}_P$ where*

$$\mathcal{R}_P = \prod_{i=0}^{m} \{ \prod_{g \in S(f_i, P)} (g = I(f_i)) \prod_{h \in S(f_i, P)} T^{f_0}_h \}$$

Proof Follows directly from the Definition 5.5.1 and Proposition 5.5.1. Consider a connection from f_{i-1} to f_i along P. The Boolean difference condition ensures the non-controlling values along each side-input s to f_i and the term $\prod_{h \in SI(f_i,P)} \overline{T_h^{f_0}}$ ensures that h does not have any transition when the input f_0 is toggled. ■

It follows that the test vector pairs for delay-faults in the 1-0 and 0-1 transitions along a path P are [4]$< v \cap f_0, v \cap \overline{f_0} >$ and $< v \cap \overline{f}_0, v \cap f_0 >$ respectively, for any $v \in \mathcal{R}_P$.

Similar to the original viability function definition [74], the function T defined by Proposition 5.5.1 is difficult to compute directly. As before, a more direct computation is shown for the case when dealing with simple gates.

Proposition 5.5.2

$$T^i_f = \prod_{g \in FI(f)} T^i_g + \sum_{h \in FI(f)} (T^i_h(h \neq I(f)))$$

Proof Recourse is made to Theorem 5.2.1 in proving the transformation. First derive an expression for $\overline{T^i_f}$, which gives the conditions under which there may be a transition at f when primary input i is toggled. When no input makes a transition, the output cannot make a transition. If some set of inputs, U, makes a transition, then the consensus terms $C_U f + C_U \overline{f}$ simplifies to the assertion of a controlling value on at least one input not in U. The complement of this term is the assertion of non-controlling values on all these inputs. That is, when at least one input makes a transition, there will be a transition at the output if all the inputs that do not have a transition on them are at non-controlling values. From this we obtain:

$$\overline{T^i_f} = \overline{\prod_{g \in FI(f)} T^i_g} \sum_{U \subseteq FI(f)} \{\prod_{k \in U} \overline{T^i_k} \prod_{h \notin U} (h = I(f)) T^i_h\}$$

The transformation of Theorem 5.2.1 yields

$$\overline{T^i_f} = \overline{\prod_{g \in FI(f)} T^i_g} \prod_{h \in FI(f)} \{\overline{T^i_h} + (h = I(f)) T^i_h\}$$

On simplifying the second term and complementing the entire expression, we get

$$T^i_f = \prod_{g \in FI(f)} T^i_g + \sum_{h \in FI(f)} (h \neq I(f)) T^i_h$$

■

[4]Note that $\mathcal{R}_P$ is a function independent of f_0.

The Boolean function T described by Proposition 5.5.2 is convenient for computing the hazard-free robust delay-fault test condition for any given path. The robust path delay-fault testability condition of Theorem 5.5.1 can now be expressed as a path-recursive function. Let $P = \{Q, f_i\}$, with Q a partial path. Then $\mathcal{R}_P = \mathcal{R}_Q \prod_{h \in S(f_i,P)}(h = I(f_i))T_h^{f_0}$. As is the case in viability analysis, the number of simple gates in the multilevel network needed to represent the robust delay-fault test condition is linear in the number of connections and gates in the original circuit.

Note that for each path, we derive a test vector pair $< v_1, v_2 >$, where v_1 and v_2 differ in the literal at the input to the path. It has been shown that when a path is delay-fault testable then such a vector pair always exists [106]. Thus, the conditions formulated above are necessary and sufficient to guarantee a test is generated if and only if a path is hazard-free robust delay-fault testable with single-path propagation. Note that allowing multiple input changes allows test set compaction, but is not considered here.

5.5.4 Delay-fault test generation results

Path delay-fault test generation is performed by replacing the viability sensitization functions of the path-tracing based timing analysis algorithm described in Section 5.3.5 with the conditions specified by Theorem 5.5.1 and Proposition 5.5.2.

Table 5.7 shows results obtained on a DEC 5000 for some large circuits These results are significantly better than those reported in [68] and [101]. A direct comparison is difficult because each program uses a different selection of paths on the large circuits. Nonetheless, the times by the proposed technique are superior to those in reported in [68] and are comparable to those of [101]. While both previous approaches report some aborted faults, there are no aborted faults by this approach. The times reported here are conservative estimates because much of the overhead is often consumed by the path-tracing procedures. This contribution is especially significant when there is a small number of paths from each primary input.

All the results reported here are obtained using the SAT package. BDD's perform comparably for those examples on which they can be built in the allocated memory. However, the BDD's for the delay-fault test functions cannot be built for several of the examples. Interestingly, this occurs even on circuits which can themselves be compactly represented by BDD's , *e.g. C880.*

It must also be noted that the paths are enumerated using the best-first algorithm employed for tracing paths in timing analysis. This approach incurs some penalty due to the processing required for this selection of paths. This penalty is particularly significant in examples where there are very few paths from a primary input; surprisingly, this happens in almost all the examples reported here.

5.6 Extensions of the path-recursive paradigm

5.6.1 General robust delay-fault models

The treatment of robust delay-fault test generation in Section 5.5 is confined to the most stringent of the different robust delay-fault models proposed in the literature. The extension of the path-recursive paradigm to the general robust delay-fault model is briefly indicated here. A full description requires an understanding of the hazard phenomenon in circuits, and is beyond the scope of this thesis.

Consider the definition of the general robust delay-fault testability of Chapter 4, which is repeated here:

Definition 5.6.1 *A path $P = \{f_0, f_1, ..., f_m\}$ is said to be* **robust delay-fault testable** *for the rising (falling) transition at f_m by the vector pair $< v_1, v_2 >$ if at each node f_i, $f_i(v_1) \neq f_i(v_2)$ yields the desired transition being tested, and for each $g_j \in S(f_i, P)$:*

1. *$g_j(v_2) = I(f_i)$; and*
2. *If $f_{i-1}(v_1) = I(f_i)$, then there is no transition on g_j.*

The vector v_2 is assumed to be applied after v_1, delayed by an amount greater than the delay of the circuit.

There are a few notable differences from the hazard-free single-path propagation RPDFT definition of Section 5.5. Using this delay-fault model, the rising and falling transition at each gate is treated separately. While each side-input to a gate along the path being tested must be at a non-controlling value on v_2, it may or may not be required to be at the non-controlling value on v_1. This condition depends on the transition that is propagated along the path under test. When the side-input is constrained to have no transition (when condition (3) is satisfied in the definition), the conditions of hazard-free delay-fault testing apply unchanged. When there is no condition on the value of the side-input under v_1, there

Name	# PI	# PO	# Gates	# Paths	# Testable	CPU secs.
5xp1	7	10	58	1072	146	59
9sym	9	1	88	560	270	114
9symml	9	1	73	485	212	90
apex6	135	99	483	4045	2401	998
apex7	49	37	151	951	833	92
b9	41	21	89	684	475	70
clip	9	5	64	506	285	33
con1	7	2	11	23	23	1
duke2	22	29	190	1366	1300	109
e64	65	65	95	2145	2145	115
misex1	8	7	28	860	68	22
misex2	25	18	38	196	179	14
o64	130	1	66	130	130	196
rd53	5	3	22	187	46	5
rd73	7	3	44	2290	177	117
rd84	8	4	68	793	277	50
vg2	25	8	130	438	392	26
xor5	5	1	12	46	46	1
z4ml	7	4	30	342	112	12
alu4†	14	8	157	7000	10	2486
bw†	5	28	85	2500	0	233
des†	256	245	2007	44537	5714	12240
f51m†	8	8	56	2869	35	253
rot†	135	107	437	20193	2553	3294
sao2†	10	4	64	4790	32	536

Satisfiability check using SAT program

† Limited to 500 longest paths from each primary input

Table 5.7: Hazard-free RPDFT test generation on MCNC circuits

Name	# PI	# PO	# Gates	# Paths	# Testable	CPU secs.
C432†	36	7	160	13849	82	1981
C880	60	26	357	8642	7551	2310
C1355†	41	32	514	20500	0	4483
C1908†	33	25	880	15700	1160	4591
C3540†	50	22	1667	20616	94	20505
C5315†	178	123	2290	29643	7366	29149
C6288‡	32	32	2416	3200	0	6047
C2670‡	233	140	1161	9709	1909	14662
C7552‡	207	108	3466	14894	4132	20249
s1196	32	32	529	3098	1544	1060
s1238	32	32	508	3559	1426	1279
s1423†	91	79	657	16178	6686	2747
s1488	14	25	653	962	916	177
s1494	14	25	647	976	913	187
s5378†	199	213	2779	13377	8480	3262

Satisfiability check using SAT program

† Limited to 500 longest paths from each primary input

‡ Limited to 100 longest paths from each primary input

Table 5.8: Hazard-free RPDFT test generation on ISCAS circuits

are several scenarios that arise. First, the side-input may have the same non-controlling value under v_1 and v_2, yet there may be transitions on the side-inputs. This is termed a ***static hazard*** [42]. Second, the side-input may have a controlling value under v_1 and makes exactly one transition to the non-controlling value when v_2 is applied. Third, the side-input may make several transitions before changing from controlling to non-controlling value when v_1 and v_2 are applied in sequence. This situation on the side-input is termed a ***dynamic hazard*** [42].

Similar to the creation of the transition function T used for the hazard-free case, functions can be defined at each gate representing the static, dynamic and hazard-free conditions with respect to a transition on some primary input. Each of these functions at a gate is described recursively in terms of functions on the gate inputs. Intuitively, each such function is of the form of Theorem 5.2.1 since some set of inputs exhibit a single property (captured by a function) while the remaining inputs exhibit another property.

5.7 Conclusions

In this chapter a new approach to solving functional path sensitization problems that have previously lacked efficient algorithms is demonstrated. First the Boolean function representing a desired condition is formulated recursively for a given path in terms of similar Boolean functions on partial or side paths (or other paths that may interact with the given path). The Boolean functions for conditions on all the paths in the original network is then represented by a multilevel network that is linear in the size of the original network. This guarantees that the condition that is being determined can always be represented for a given circuit. The second step is to determine whether the multilevel network has a satisfying assignment. We have shown that, given the multilevel network built up by the first step, one or the other of the two recent techniques of BDD's and SAT successfully perform timing analysis and delay-fault test generation on all large benchmark examples. The BDD approach is more effective than SAT when timing analysis is performed with path-tracing, and the BDD's for the multilevel functions can be built in the allocated memory. However, for timing analysis without path-tracing and delay-fault test generation, the SAT approach outperforms the BDD approach.

This paradigm is quite general and has recently been applied to determine the formulation of path characteristic functions for hazard conditions under simultaneous multiple input changes [79]. This allows reduced test size in the case of delay-fault test generation by allowing a single test vector pair to test several paths simultaneously. A potential benefit of the technique should also be in the synthesis of hazard-free asynchronous circuits.

Chapter 6

Conclusions

Logic synthesis has three principal optimization criteria, area, performance, and testability. While optimization techniques for each of these parameters is well developed, not much is understood of the interaction among the three criteria. This thesis has attempted to deepen the understanding of the interaction between the area, delay, and testability of an optimized combinational logic circuit. The focus has been on the interaction between performance and testability in both the synthesis and analysis process in the design of circuits. The contributions of the preceding chapters are summarized below.

Motivated by the example of a high-performance circuit that exhibits unreliable functional behavior in the presence of redundancy, it is shown that redundancy is not necessary to reduce the delay of a circuit. That is, for every redundant circuit there exists a logically equivalent (single or multiple-stuck fault) irredundant circuit guaranteed to have the same or less delay. The most accurate functional analysis technique for estimating the delay, namely viability analysis, is employed in proving this result. An efficient single-pass implementation of the algorithm has been developed. While the increase in area and fanout on the gates due to duplication during the algorithm is empirically small, an open question is whether a theoretical bound exists on the area of the final irredundant circuit relative to the area of the initial circuit.

The result of the first part of the thesis leads to the question of whether performance optimization can be done without creating redundancy in a circuit. The resolution of this question constitutes the second contribution of the thesis. The conditions under which common performance optimization operations introduce redundancy into an initially single stuck-fault irredundant circuit are derived. It is shown that recovering the testability of the circuit is difficult and alternate fault models are considered to determine invariance under performance optimization. The testability of 100% robust path delay-fault testable circuits is shown to remain invariant during timing optimization. These circuits also have another property desirable during timing optimization, namely that there are no false paths. Thus, the need for delay estimation using techniques such as viability analysis is obviated by static analysis (longest path estimate). Two open questions remain before testability can be completely considered while optimizing the delay of a circuit. While synthesis procedures for some classes of robust delay-fault testable circuits are known, an unanswered question is whether a robust delay-fault testable implementation exists for any circuit with at most the same delay and minimal area increase.

The first two problems addressed in this thesis are concerned with the synthesis aspects for optimizing the performance and testability of a circuit. However, the synthesis procedures require the analysis of two properties of a circuit: the computed delay of a circuit using viability analysis and robust delay-fault testability. Both properties are well understood but efficient algorithms have been lacking. Most previous approaches attempt to solve the two similar problems using modifications to a standard stuck-fault test generation program. The final contribution of this thesis is the development of a general framework for solving a class of problems, which includes both viability analysis and delay-fault test generation, that requires functional analysis on paths in the circuit. It is shown that an efficient linear sized formulation exists for these two problems by expressing the Boolean function capturing the desired property recursively.

The techniques used in Chapter 3 in implementing both versions of the KMS algorithm consist of structurally modifying a given circuit (such as gate duplications and connection movement) to obtain a functionally identical circuit which realizes an important (topological) relationship between the delay, testability, and area of the circuit. While the property being studied is difficult to analyze or synthesize for on the given circuit structure, it is simplified to some direct (topological) relationship between the delay, testability, and area on the transformed (but equivalent) circuit. For example, if all paths of length $\geq L$ are non-viable in a given circuit, this implies a redundant multiple-stuck fault in the corresponding L-path-disjoint circuit (*c.f.* Chapter 3). However, nothing is implied about testability in the original circuit. This class of techniques has further ramifications both in the synthesis and analysis of problems concerned with functional timing analysis [92] and delay-fault test generation [93]. The transformations are easy to apply (mostly linear time in the size of the initial circuit), and often, the increase in area can be bounded. For example, in the case of redundancy removal to ensure no increase in the delay, only a single redundant multiple-stuck fault is removed; in the case of robust delay-fault test generation [93], single stuck-fault test generation is required. It is expected that similar techniques can be applied to other circuit properties (such as hazards) that appear to be difficult to analyze and synthesize.

While all of the results have been proved for combinational logic circuits they potentially apply to synchronous sequential circuits. An additional dimension arises in sequential circuits due to the presence of memory elements. Thus, the impact of state assignment on the resulting performance and testability of the circuit is an extremely interesting and important problem to be explored. Some understanding of the relationship between state assignment and sequential single stuck-fault testability is reported in [40, 39]. Circuit restructuring and resynthesis techniques have also been proposed that can optimize logic across latch boundaries [67, 71]. However, little is understood of the remaining interactions that occur among the memory elements, performance, and testability in sequential circuits.

Both timing analysis using accurate functional analysis techniques such as viability analysis and robust delay-fault testability are emerging criteria in logic synthesis. The work reported in this thesis has deepened the theoretical understanding and interactions of these two properties while simultaneously providing a novel framework for solving practical instances of these and other problems. It is hoped that these contributions will become part of the state-of-the-art techniques in digital circuit synthesis in the near future.

[illegible]

[illegible]

[illegible]

Appendix A

Boolean Network Satisfiability

A.1 The satisfiability problem

The problem *satisfiability* is considered the first NP-complete problem since it was used to introduce this class of problems [46]. The satisfiability problem is [46]:

Satisfiability

Instance:	A set U of variables and a collection C of clauses over U. Each clause is a sum term of some variables from U.
Question:	Is there a satisfying truth assignment for C?

As a general problem solving technique, satisfiability is related to important CAD problems. Some of these and their previous solutions are briefly described below:

- Two-level logic minimization: A fundamental question asked in several two-level logic minimization programs [15, 30, 70, 98] is the tautology question. This is the complement of the satisfiability problem and asks whether a Boolean function is identically one, or equivalently, is there a satisfying truth assignment to the complement of the satisfiability problem, a two-level sum-of-products expression. The tautology algorithm is solved in [15] using an algorithm based on the *unate recursive paradigm*. Given a Boolean cover, a set of rules is used to detect if the cover is not a tautology. If this step does not succeed, the Boolean cover is split into two covers corresponding to the Shannon cofactors with respect to a selected variable. The rules are then applied to each cofactored cover independently. This divide and conquer approach has proved successful on almost all two-level minimization examples [15].

- Test-generation: The goal in stuck-fault test generation is to determine an assignment of values to the primary input of a circuit such that the true and faulty circuits exhibit different logical behavior. Almost every test generation program up to now has solved this problem directly at the circuit level by developing a suite of heuristics that attempt to speed up the search for a satisfying test vector. Among these are deterministic improvements which always guarantee an improvement in the search process, such as the use of implications [45, 100]. Other heuristics, although not guaranteed to always yield fast solutions yet perform exceptionally well on most benchmark examples, are also widely used. One such technique is implicit enumeration of the Boolean space restricted to the primary inputs [49].

- Logic verification: The equivalence of two circuits is checked by ensuring that a satisfying assignment exists for an output in one circuit if and only if the assignment satisfies the same output in the other circuit. This assumes the assignment is not a don't care for the output [55]. The pervasive approach of logic simulation has given way to techniques using BDD's [21, 12]. The latter is successful in solving most combinational circuit verification problems and has recently been extended to sequential circuit verification [29]. A few comments on the BDD ordering problem are given in Section A.2.

- Path sensitization conditions: Timing analysis and delay-fault test generation are examples of path-sensitization problems considered in depth in Chapter 5. Other examples include τ-irredundant test generation [77] and hazard analysis [42]. All of the approaches to these problems, with the exception of the techniques discussed in this thesis, involve modifications of the test generation paradigm used for stuck-faults [14, 41, 38, 68].

Each problem has been solved individually by tailoring the algorithms and heuristics to the particular application at hand. Two factors have emerged that warrant a second look at solutions to these problems. First, the interaction and similarity among the problems is better understood now. For example, in this thesis the close-knit similarity between timing analysis and delay-fault test generation is exploited. Second, two recent proposals for an efficient solution to the satisfiability problem represented by a multilevel network [12, 64], has enabled a large class of existing and emerging problems to be solved efficiently. Thus, in place of adapting search strategies to each specific problem, which is the usual method, one can translate the problem into a general satisfiability question and use a generic algorithm. This allows the powerful search techniques and heuristics that are developed for the single general problem to be easily applied to related problems.

The general satisfiability problem on Boolean networks is stated as:

Boolean Network Satisfiability
Instance: A Boolean network with primary inputs I and a single primary output O.
Question: Does there exists an assignment to I that causes O to be 1 ?

Clearly, every satisfiability problem can be converted into a Boolean network satisfiability problem and vice versa. In the sequel, the discussion is restricted to the Boolean network satisfiability problem.

Two techniques have emerged recently which have proven to be successful in solving the general satisfiability problems that arise in a class of problems.

A.2 BDD's

A reduced ordered binary decision diagram (BDD) is a canonical representation of a function that allows very fast Boolean function manipulations, such as cofactoring and equivalence checking. The size of the BDD is very sensitive to the ordering used for the variables in constructing the BDD. Since the problem of determining an optimum order is NP-complete, this technique is limited by the size of the BDD when a good ordering is not obtained heuristically. There are also functions known for which every order yields an exponentially sized BDD [21]. The details of the implementation of the BDD package used in the work reported in this thesis can be found in [12].

A.3 The SAT package

Larrabee has shown that for a certain class of problems, the satisfiability problem can be effectively solved by a heuristic technique [64]. This approach is adopted here and augmented with new search strategies that provide a significant improvement over earlier results [65, 23]. The interface to the SAT package is quite simple. It accepts a set of clauses and determines if they can be satisfied; if a satisfying solution is found, the variable assignment is returned.

A.3.1 Forming clauses

In order to use SAT, the specific problem must be converted into a set of clauses. It is assumed that the circuit represents the set of all solutions for the property being determined. For example, the circuit description for testing a stuck-fault must include the condition that at least one output is different in the good and faulty circuit. This is done by using an EXOR gate between each output in the true and faulty circuit [64].

Often additional information can be passed to the satisfiability program that enhances the efficiency of the search problem. For example, [64] illustrates the use of *active* clauses for improving the speed of test generation. Active clauses capture the close interaction between gates in the true and faulty circuits. More precisely, active clauses ensure the existence of at least one path in the transitive fanout of the fault site such that the true and faulty values on the gates along the path are at different values. While [64] generates these active clauses via an independent analysis of the circuit, these clauses are just as easily captured as part of the multilevel circuit being checked for satisfiability. This allows uniform treatment of all the problems being solved using the Boolean network satisfiability approach.

The overhead of generating the clauses is generally negligible compared to the complexity of solving the problem.

The set of clauses which describe a combinational logic element are found by expressing its *characteristic function* in product of sums form. The characteristic function can be formed by rewriting the conventional representation of $f = g$, where f is a gate and g is an expression, as $fg + f'g'$. For example, an AND gate, $f = ab$ has the characteristic function $f(ab) + f'(a' + b')$. Expressing this in product of sums form gives $(a + f')(b + f')(a' + b' + f)$. The characteristic function of a Boolean network is the conjunction of the characteristic functions of the circuit elements.

A.3.2 The search problem

Given the clauses, searching for a solution can be done using standard branch and bound techniques. In SAT, branching consists of selecting an unassigned variable, setting it to either *1* or *0* (true or false, respectively), and reversing the assignment (backtracking) if the first choice does not lead to a solution.

The search is bounded in three ways. First, if an unassigned clause has no more unassigned literals (*i.e.* the clause evaluates to false), the current partial assignment is contradictory, and the search can be bounded. Second, if an unsatisfied clause has only one unassigned literal, that literal can be immediately implied. Third, if the previous step implies x, but x' is already asserted, the assignment is contradictory and the search can be bounded. Despite these rules for bounding the search, a contradictory partial assignment may still not be detected unless it is completed to all the variables. This often leads to exponential worst case performance. However, there are several deterministic improvements that are made to the basic brand and bound algorithm that help improve this situation.

Rejecting a partial assignment is termed backtracking, Typically, the goal of an efficient algorithm is to minimize the number of backtracks needed to find a satisfying assignment. Nevertheless, the amount of computation required for determining each assignment must also be accounted for. Thus, a heuristic that reduces the backtrack limit by a factor of two but increases the amount of computation in choosing each assignment by a factor of four is not effective.

The set of heuristics used to guide the branch and bound search is called the search strategy. There are three parameters in the Boolean network satisfiability problem:

- Variable (and clause) order for branching.
- Processing performed at each branch point.
- Amount of backtracking performed.

Each of these aspects is touched upon now.

Variable ordering

Three heuristics for variable orderings are reported in [64]. All of them are static orderings which use a single order for the variables throughout the search process. The problem with static ordering is that it often does many unnecessary variable assignments. This exacerbates the exponential behavior of the search. For example, if one input of an AND gate is assigned *0*, this fixes the output to *0*. Since the other inputs remain unassigned, a static ordering may assign values to each of these inputs, before realizing that the actual conflict is by the gate output being *0*.

In the SAT package developed here, three simple greedy dynamic orderings are employed. In the first ordering, at each branch point, the first unsatisfied variable in the first unassigned clause is selected for assignment. In the second ordering, at each branch point, the last unsatisfied variable in the first unassigned clause is selected for assignment. These two strategies perform approximately equally well. In the third strategy, at each branch point, the variable that occurs most frequently in each of the remaining clauses is selected for assignment.

Clause ordering

Since the variable orderings described above depend on the ordering of the clauses, this is also a factor.

The strategy followed for ordering clauses is based upon the ideas proposed in the PODEM algorithm. PODEM performs test generation by performing assignments only on the primary inputs of the circuit. This heuristic often provides an order of magnitude improvement over the D-algorithm of [90]. A similar strategy is realized in SAT by ordering the clauses from inputs to outputs; even though the SAT program makes no distinction between primary inputs and intermediate variables, ties are broken in favor of primary inputs and variables closest to primary inputs. The first two variable ordering strategies branch only on primary inputs if the clauses are presented in topological order to the SAT program.

Implications

The greedy strategies alone do not generate 100% fault coverage on most circuits. Increasing the backtrack limit beyond a small number does not prove to be cost effective. Thus, additional heuristics are used. These are based on the notion of implications, first described in [45, 100] and extended to the SAT framework in [64].

Referring to a clause with n literals as a *n-clause*, the clauses are partitioned into 1-clauses, 2-clauses, and (3+)-clauses. All 1-clauses force the corresponding literal to be asserted. The 2-clauses are converted into an implication graph which makes it easy to process implications and compute nonlocal implications. The remaining clauses are put in an efficient data structure for detecting contradictions during the branch and bound. A contradiction occurs when assigning a variable a value implies the opposite value, or if a (3+)-clause has no satisfied literals.

There are two kinds of implications: local implications and non-local implications. Local implications correspond to 2-clauses which mostly relate values on inputs and outputs of single nodes and values on branch points of connections. Non-local implications are derived by following implications through (3+)-clauses. When an implication involves a (3+)-clause, the contra-positive implication is also added to the implication graph. For example, if $A \Rightarrow F$ is derived through a (3+)-clause, $F' \Rightarrow A'$ is added to the implication graph.

A further implication may be derived by determining if a variable must be forced to a unique value for a satisfying solution. To detect such an implication for a variable A, assign a value to A and apply the bounding steps described above. If this leads to a contradiction, then the opposite value is asserted on A. If this leads to a contradiction, the partial assignment is contradictory. This technique finds more implications since it also uses the (3+)-clauses.

Of course, all the heuristics are not required on each SAT call. In fact, using all of them often leads to a very slow implementation. [23] provides an approach where each of these heuristics can be incorporated into the search problem. However, that approach involves the computation of the transitive closure of the implications among all the variables, which leads to excessive pre-processing times. The approach sketched above and reported in [107] invokes the increasingly powerful heuristics only after the weaker and faster options fail to determine a solution within a fixed number of backtracks.

A.3.3 Network structure and satisfiability

It is empirically observed that the SAT program behaves differently on alternate representations of a given network structure. For example, a network composed of complex gates is typically harder to solve for a satisfying assignment compared to an equivalent network of simple gates. [65] also observes this behavior for XOR gates, and replaces each XOR gate by an equivalent connection of AND-OR gates before performing test generation.

The SAT framework also has another significant advantage over structure based test generation that is performed directly on an existing network. By formulating the

SAT problem as a multilevel function, network restructuring techniques can be employed to realize alternate structures that are more amenable to Boolean satisfiability. At this time, while SAT successfully completes on all circuits experimented with for test generation, timing analysis and delay-fault test generation, it is expected that as larger circuits are encountered, the SAT problem will become proportionately more difficult to solve. The added dimension of network synthesis before SAT remains an interesting open problem where more experimentation is required.

Bibliography

[1] P. Agrawal and V. Agrawal. Personal communication. June 1991.

[2] J. Allen. Performance-directed synthesis of VLSI systems. *The Proceedings of the IEEE*, 78(2):336–355, February 1990.

[3] P. Antognetti and G. Massobrio. *Semiconductor Device Modeling with SPICE.* McGraw Hill, 1988.

[4] P. Bardell, W. McAnney, and J. Savir. *Built-in test for VLSI : Pseudorandom techniques.* John Wiley and Sons, 1987.

[5] K. Bartlett, D. Bostick, G. Hachtel, R. Jacoby, M. Lightner, P. Moceyunas, C. Morrison, and D. Ravenscroft. Bold: The Boulder optimal logic design system. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 62–65, 1987.

[6] K. Bartlett, R. Brayton, G. Hachtel, R. Jacoby, C. Morrison, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic minimization using implicit don't cares. *IEEE Transactions on Computer-Aided Design*, C-7(6):723–740, June 1988.

[7] K. Bartlett, W. Cohen, A. deGeus, and G. Hachtel. Synthesis and optimization of multilevel logic under timing constraints. *IEEE Transactions on Computer-Aided Design*, C-5(4):582–595, October 1986.

[8] J. Benkoski, E. Meersch, L. Claesen, and H. De Man. Efficient algorithms for solving the false path problem in timing verification. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 44–47, November 1987.

[9] J. Benkoski, E. Meersch, L. Claesen, and H. De Man. Timing verification using statically sensitizable paths. *IEEE Transactions on Computer-Aided Design*, C-9(10):1073–1084, October 1990.

[10] C. Berman, J. Carter, and K. day. The fanout problem: From theory to practice. In *The Proceedings of the Decennial Caltech VLSI Conference*, pages 69–99, March 1989.

[11] L. Berman, D. Hathaway, A. LaPaugh, and L. Trevillyan. Efficient techniques for timing correction. In *The Proceedings of the International Conference on Computer Design*, pages 415–419, August 1990.

[12] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *The Proceedings of the Design Automation Conference*, pages 40–45, June 1989.

[13] D. Brand. Redundancy and don't cares in logic synthesis. *IEEE Transactions on Computers*, C-32(10):947–952, October 1983.

[14] D. Brand and V. Iyengar. Timing analysis using functional analysis. *IEEE Transactions on Computers*, C-37(10):1309–1314, October 1988.

[15] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[16] R. Brayton, G. Hachtel, and A. Sangiovanni-Vincentelli. Multilevel logic synthesis. *The Proceedings of the IEEE*, 78(2):264–300, February 1990.

[17] R. Brayton and C. McMullen. The decomposition and factorization of Boolean expressions. In *The Proceedings of the Internation Symposium on Circuits and Systems*, pages 49–54, May 1982.

[18] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, C-6(6):1062–1081, November 1987.

[19] M. Breuer and A. Friedman. *Diagnosis and reliable design of digital systems.* Computer Science Press, 1976.

[20] M. Bryan, S. Devadas, and K. Keutzer. Testability-preserving circuit transformations. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 456–459, November 1990.

[21] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[22] R. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS: A compiled simulator for MOS circuits. In *The Proceedings of the Design Automation Conference*, pages 9–16, 1987.

[23] S. Chakradhar and V. Agrawal. A transitive closure based algorithm for test generation. In *The Proceedings of the Design Automation Conference*, pages 353–358, June 1991.

[24] H-C. Chen and D. Du. Path sensitization in critical path problem. In *The Proceedings of the 1990 ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, August 1990.

[25] H-C. Chen, D. Du, and L-R. Liu. Critical path selection for performance optimization. In *The Proceedings of the Design Automation Conference*, pages 547–550, June 1991.

[26] K-C. Chen and S. Muroga. Timing optimization for multi-level combinational circuits. In *The Proceedings of the Design Automation Conference*, pages 339–344, June 1990.

[27] K-T. Cheng and V. Agrawal. *Unified methods for VLSI simulation and test generation*. Kluwer Academic Publishers, 1989.

[28] K-T. Cheng, S. Devadas, and K. Keutzer. Robust delay-fault test generation and synthesis for testability under a standard scan design methodology. In *The Proceedings of the Design Automation Conference*, pages 80–86, June 1991.

[29] O. Coudert, C. Berthet, and J. Madre. Verification of synchronous sequential machines based on symbolic execution. In *The Proceeedings of the Workshop on Automatic Verification Methods for Finite State Systems, Grenoble - France*, June 1989.

[30] M. Dagenais, V. Agrawal, and N. Rumin. McBoole: A new procedure for exact logic minimization. *IEEE Transactions on Computers*, C-33(1):229–238, January 1986.

[31] R. Dandapani and S. Reddy. On the design of logic networks with redundancy and testability considerations. *IEEE Transactions on Computers*, C-23(11):1139–1149, November 1974.

[32] J. Darringer, D. Brand, J. Gerbi, W. Joyner, and L.Trevillyan. LSS: A system for production logic synthesis. *IBM Journal of Research and Development*, 28(5):326–328, September 1984.

[33] G. DeMicheli. Performance-oriented synthesis of large-scale domino CMOS circuits. *IEEE Transactions on Computer-Aided Design*, C-6(5):751–765, September 1987.

[34] G. DeMicheli, R. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on Computer-Aided Design*, C-4(3):269–285, July 1985.

[35] E. Detjens, G. Gannot, R. Rudell, and A. Sangiovanni-Vincentelli. Technology mapping in MIS. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 116–119, November 1987.

[36] S. Devadas and K. Keutzer. Necessary and sufficient conditions for robust delay-fault testability of combinational logic circuits. In *The Proceedings of the 6th MIT Conference on Advanced Research in VLSI*, pages 221–238, April 1990.

[37] S. Devadas and K. Keutzer. Synthesis and optimization procedures for robustly delay-fault testable combinational logic circuits. In *The Proceedings of the Design Automation Conference*, pages 221–227, June 1990.

[38] S. Devadas, K. Keutzer, and S. Malik. Path sensitization conditions and delay computation in combinational logic circuits. In *The Proceedings of the International Workshop on Logic Synthesis*, May 1991.

[39] S. Devadas, H-K. Ma, A. Newton, and A. Sangiovanni-Vincentelli. A synthesis and optimization procedure for fully and easily testable sequential machines. *IEEE Transactions on Computer-Aided Design*, C-8(10):1100–1107, October 1989.

[40] S. Devadas, H-K. Ma, A. Newton, and A. Sangiovanni-Vincentelli. Irredundant sequential machines via optimal logic synthesis. *IEEE Transactions on Computer-Aided Design*, C-9(1):8–18, January 1990.

[41] D. Du, S. Yen, and S. Ghanta. On the general false path problem in timing analysis. In *The Proceedings of the Design Automation Conference*, pages 555–560, 1989.

[42] E. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM Journal of Research and Development*, 9:90–99, March 1965.

[43] L. Fein. Redundancy - a misleading misnomer. In *Redundancy Techniques for Computing Systems*, pages 1–8. Spartan Books, 1962.

[44] J. Fishburn and A. Dunlop. TILOS: A posynomial programming approach to transistor sizing. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 326–328, 1985.

[45] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Transactions on Computers*, C-32(12):1137–1144, December 1983.

[46] M. Garey and D. Johnson. *Computers and intractability: A guide to the theory of NP-completeness.* W.H. Freeman and Company, 1979.

[47] A. Ghosh. Techniques for test generation and verification in VLSI sequential circuits. *Ph.D. Thesis, University of California - Berkeley*, August 1991.

[48] C. Glover and M. Mercer. A method for delay fault test generation. In *The Proceedings of the Design Automation Conference*, pages 90–95, June 1988.

[49] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Transactions on Computers*, C-30(3):215–222, March 1981.

[50] A. Guyot, B. Hochet, and J-M. Muller. A way to build efficient carry-skip adders. *IEEE Transactions on Computers*, C-36(10):1144–1152, October 1987.

[51] G. Hachtel, R. Jacoby, K. Keutzer, and C. Morrison. On properties of algebraic transformations and the multifault testability of multilevel logic. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 422–425, November 1989.

[52] J. Hartmanis and R. Stearns. *Algebraic Structure Theory of Sequential Machines.* Prentice-Hall, Englewood Cliff, N.J., 1966.

[53] M. Hoffman and J. Lim. Delay optimization of combinational static CMOS logic. In *The Proceedings of the Design Automation Conference*, pages 125–131, June 1987.

[54] M. Jackson and E. Kuh. Performance-driven placement of cell based IC's. In *The Proceedings of the Design Automation Conference*, pages 370–375, June 1989.

[55] R. Jacoby. On the comparison of Boolean functions. *Ph.D. Thesis, University of Colorado - Boulder*, May 1989.

[56] R. Jacoby, P. Moceyunas, H. Cho, and G. Hachtel. New ATPG techniques for logic optimization. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 548–551, November 1990.

[57] N. Jouppi. Timing analysis and performance improvement of MOS VLSI designs. *IEEE Transactions on Computer-Aided Design*, C-6(4):650–665, July 1987.

[58] K. Keutzer and G. Hachtel. Logic optimization can diminish the testability of combinational logic circuits. *AT&T Bell Laboratories Technical Memorandum, 11253-881123-24TM*, November 1988.

[59] K. Keutzer, S. Malik, and A. Saldanha. Is redundancy necessary to reduce delay? *IEEE Transactions on Computer-Aided Design*, C-10(4):427–435, April 1991.

[60] K. Keutzer and M. Vancura. Timing optimization in a logic synthesis system. In *The Proceedings of the International Workshop on Logic Synthesis, Amsterdam*. North-Holland, May 1988.

[61] I. Kohavi and Z. Kohavi. Detection of multiple faults in combinational logic networks. *IEEE Transactions on Computers*, C-21(6):556–568, June 1972.

[62] E. Kuh and T. Ohtsuki. Recent advances in VLSI layout. *The Proceedings of the IEEE*, 78(2):237–263, February 1990.

[63] S. Kundu and S. Reddy. On the design of robust testable CMOS combinational logic circuits. In *The Proceedings of the International Fault Tolerant Computing Symposium*, pages 220–225, June 1988.

[64] T. Larrabee. Efficient generation of test patterns using Boolean difference. In *The Proceedings of the International Test Conference*, pages 795–801, August 1989.

[65] T. Larrabee. Efficient generation of test patterns using Boolean satisfiability. *Ph.D. Thesis, Stanford University*, December 1989.

[66] M. Lehman and N. Burla. Skip techniques for high-speed carry-propagation in binary arithmetic units. *IRE Transactions on Electronic Computers*, EC-10:691–698, December 1961.

[67] C. Leiserson and J. Saxe. Optimizing synchronous circuitry. *Journal of VLSI and Computer Systems*, 1:41–67, January 1983.

[68] C. Lin and S. Reddy. On delay fault testing in logic circuits. *IEEE Transactions on Computer-Aided Design*, C-6(5):694–703, September 1987.

[69] R. Lisanke. Logic synthesis benchmark circuits for the International Workshop on Logic Synthesis, May 1989.

[70] A. Malik, R. Brayton, A. Newton, and A. Sangiovanni-Vincentelli. A modified approach to two-level minimization. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 106–109, 1988.

[71] S. Malik, E. Sentovich, R. Brayton, and A. Sangiovanni-Vincentelli. Retiming and resynthesis: Optimization of sequential networks with combinational techniques. *IEEE Transactions on Computer-Aided Design*, C-10(1):74–84, January 1991.

[72] W. Mao and M. Ciletti. Arrangement of latches in scan-path design to improve delay fault coverage. In *The Proceedings of the International Test Conference*, pages 387–393, September 1990.

[73] M. McFarland, A. Parker, and R. Camposano. The high-level synthesis of digital systems. *The Proceedings of the IEEE*, 78(2):301–318, February 1990.

[74] P. McGeer. On the interaction of functional and timing behavior of combinational logic circuits. *Ph.D. Thesis, University of California - Berkeley*, November 1989.

[75] P. McGeer and R. Brayton. Efficient algorithms for computing the longest viable path in a combinational network. In *The Proceedings of the Design Automation Conference*, pages 561–567, June 1989.

[76] P. McGeer and R. Brayton. Provably correct critical paths. In *The Proceedings of the Decennial Caltech VLSI Conference*, 1989.

[77] P. McGeer, R. Brayton, R. Rudell, and A. Sangiovanni-Vincentelli. Extended stuck-fault testability for combinational networks. In *The Proceedings of the 6th MIT Conference on Advanced Research in VLSI*, pages 239–259. MIT Press, April 1990.

[78] P. McGeer, R. Brayton, A. Sangiovanni-Vincentelli, and S. Sahni. Performance enhancement through the generalized bypass transform. In *The Proceedings of the International Workshop on Logic Synthesis*, May 1991.

[79] P. McGeer, A. Saldanha, R. Brayton, and A. Sangiovanni-Vincentelli. The hazard equation and its application to delay-fault test generation. *Unpublished manuscript, University of California - Berkeley*, September 1991.

[80] P. McGeer, A. Saldanha, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Timing analysis and delay-fault test generation using path recursive functions. In *The Proceedings of the International Workshop on Logic Synthesis*, May 1991.

[81] C. Morrison. Multilevel logic minimization. *Ph.D. Thesis, University of Colorado - Boulder*, August 1989.

[82] S. Muroga, Y. Kambayashi, H. Lai, and J. Culliney. The transduction method - design of logic networks based on permissible functions. *IEEE Transactions on Computers*, C-38(10):1404–1424, October 1989.

[83] V. Oklobdzija and E. Barnes. Some optimal schemes for ALU implementation in VLSI technology. In *The Proceedings of the Seventh Symposium on Computer Arithmetic*, pages 2–8, June 1985.

[84] J. Ousterhout. A switch-level timing verifier for digital MOS circuits. *IEEE Transactions on Computer-Aided Design*, C-4(3):336–349, July 1985.

[85] M. Pedram and N. Bhat. Layout driven technology mapping. In *The Proceedings of the Design Automation Conference*, pages 99–105, 1991.

[86] M. Pedram, N. Bhat, K. Chaudhary, S. Mayrhofer, and E. Kuh. Layout considerations in combinational logic synthesis. In *The Proceedings of the International Workshop on Logic Synthesis*, May 1991.

[87] S. Perremans, L. Claesen, and H. De Man. Static timing analysis of dynamically sensitizable paths. In *The Proceedings of the Design Automation Conference*, pages 568–573, 1989.

[88] A. Pramanick and S. Reddy. On the design of path delay fault testable combinational circuits. In *The Proceedings of the International Fault Tolerant Computing Symposium*, pages 374–381, June 1990.

[89] A. Pramanick, S. Reddy, and S. Senjupta. Synthesis of combinational logic circuits for path delay fault testability. In *The Proceedings of the Internation Symposium on Circuits and Systems*, pages 3105–3108, May 1990.

[90] J. Roth. Diagnosis of automata failures: A calculus and a method. *IBM Journal of Research and Development*, 10:278–291, July 1966.

[91] K. Roy, J. Abraham, K. De, and S. Lusky. Synthesis of delay fault testable combinational logic. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 418–421, November 1989.

[92] A. Saldanha, R. Brayton, and A. Sangiovanni-Vincentelli. Circuit structure relations to redundancy and delay: The KMS algorithm revisited. *Submitted for publication*, October 1991.

[93] A. Saldanha, R. Brayton, and A. Sangiovanni-Vincentelli. Equivalence of robust delay-fault and single stuck-fault test generation. *Submitted for publication*, October 1991.

[94] A. Saldanha, R. Brayton, A. Sangiovanni-Vincentelli, and K-T. Cheng. Timing optimization with testability considerations. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 460–463, November 1990.

[95] A. Saldanha, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Multi-level logic simplification using don't cares and filters. In *The Proceedings of the Design Automation Conference*, pages 277–282, June 1989.

[96] J. Savir and W. McAnney. Random pattern testability of delay faults. In *The Proceedings of the International Test Conference*, pages 263–273, September 1986.

[97] H. Savoj, R. Brayton, and H. Touati. The use of image computation techniques in extracting local don't cares and network optimization. In *The Proceedings of the International Workshop on Logic Synthesis*, May 1991.

[98] H. Savoj, A. Malik, and R. Brayton. Fast two-level logic minimizers for multi-level logic synthesis. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 544–547, 1989.

[99] D. Schertz. On the representation of digital faults. *Ph.D. Thesis, University of Illinois - Urbana*, 1969.

[100] M. Schulz and E. Auth. Advanced automatic test pattern generation and redundancy identification techniques. In *The Proceedings of the International Fault Tolerant Computing Symposium*, pages 30–35, June 1988.

[101] M. Schulz, K. Fuchs, and F. Fink. Advanced automatic test pattern generation techniques for path delay faults. In *The Proceedings of the International Fault Tolerant Computing Symposium*, pages 44–51, June 1989.

[102] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli. SIS: A sequential synthesis and optimization system. *Submitted for publication*, October 1991.

[103] C. Shannon. The synthesis of two-terminal switching circuits. *Bell System Technical Journal*, 28:59–98, 1949.

[104] K. Singh and A. Sangiovanni-Vincentelli. A heuristic algorithm for the fanout problem. In *The Proceedings of the Design Automation Conference*, pages 357–360, June 1990.

[105] K. Singh, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 282–285, November 1988.

[106] G. Smith. Model for delay faults based upon paths. In *The Proceedings of the International Test Conference*, pages 342–349, August 1985.

[107] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. TEGUS: Test generation using satisfiability. *Unpublished manuscript, University of California - Berkeley*, September 1991.

[108] H. Touati. Performance-oriented technology mapping. *Ph.D. Thesis, University of California - Berkeley*, November 1991.

[109] H. Touati, H. Savoj, and R. Brayton. Delay optimization of combinational logic circuits through clustering and partial collapsing. In *The Proceedings of the International Workshop on Logic Synthesis*, May 1991.

[110] S. H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley Interscience, 1969.

[111] J. Vasudevamurthy and J. Rajski. A method for concurrent decomposition and factorization of Boolean expressions. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 510–513, November 1990.

[112] N. Weste and K. Eshraghian. *Principles of CMOS VLSI design: A systems perspective.* Addison-Wesley Publishing Company, 1985.

Index of Definitions