UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Functional Timing Analysis of VLSI Circuits Containing Complex Gates

by

JOSÉ LUÍS ALMADA GÜNTZEL

Thesis submitted as partial fulfillment of the requirements
for obtaining the degree of "Doutor em Ciência da Computação"

Prof. Ricardo Augusto da Luz Reis
Advisor

Porto Alegre, November 2000.

C.I.P. - Catalogação na Publicação

Güntzel, José Luís Almada

Functional Timing Analysis of VLSI Circuits Containing Complex Gates / by José Luís Almada Güntzel. – Porto Alegre : PPGC da UFRGS, 2000.

182 p. : il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR – RS, 2000. Orientador: Reis, Ricardo Augusto da Luz.

1. Microeletrônica. 2. Ferramentas de CAD para Microeletrônica. 3. Verificação de Timing. 4. Análise de Timing. 5. Portas Lógicas Complexas. I. Reis, Ricardo Augusto da Luz. II. Título.

Acknowledgments

This work is the result of more than five years of research. During this time many people contributed to its development and I will probably forget to mention somebody that helped me along the way.

First of all, I thank my family for their support and constant encouragement, mainly during the difficult periods when I could not see the light at the end of the tunnel. I also thank my fiancée, Margarida, for her friendship and patience, accepting my excuses for each working weekend.

I also thank my advisor, Ricardo Reis, for his unconditional support and friendship.

Timing analysis became the focus of my research since the time I have spent at the Laboratoire d´Informatique, de Robotique et de Micro-électronique de Montpellier - LIRMM (France). I acknowledge Prof. Daniel Auvergne who kindly accepted me in his working group. I thank Nadine Azemard, my direct supervisor, and Séverine Cremoux, my team mate, for their willingness in working with me and for their patience with my poor French skills. During 1996, the year I spent in France, I had the chance of meeting many interesting people whose friendship I really miss. Ricardo Pires, Véronique Moreda, Daniel Séverac, Éric Vanier, Jean-Michel Daga, Lionel Torres and many others, I thank you for all moments we have shared together. Still during that time, I have the support and friendship of many other people that I previously knew. I am particularly indebted with André Reis, who has hosted me during the first two weeks in France. I also thank André Reis for our discussions on logic synthesis, technology mapping, timing analysis, BDDs, test and so many other philosophical topics! I should also mention Renato Ribas, Gilson Wirth and Reginaldo Tavares for both technical and philosophical discussions.

During the other four years of research, I had the help of undergraduate and graduate students. I specially thank Ana Cristina Medina Pinto for helping me in the tediously task of testing so many versions of path enumeration algorithms. I also thank her for our endless discussions on ATPG-based timing analysis. I also thank Guilherme Dal Pizzol for helping me with the *ad hoc* timing analysis of carry skip adders and Eduardo d´Ávila for the first implementation of the extended three-valued timed calculus for complex gates.

During all the five years I could always count with the support and friendship of Fernando Moraes, including some “on-line” helps during my stay in France, for what I sincerely acknowledge.

Although timing analysis is my main topic of interest, I could not resist checking other topics. In this sense, I thank to Fernanda Lima, Luigi Carro and Marcelo Johann for having shared several discussions and some papers on Masterslices Architectures for FPGAs. I also thank to Fábio Klein Ferreira for developing the path-based power evaluation tool, as final paper of his undergraduate course.

I could not forget to mention the friendship and support of all colleagues of the UFRGS Microelectronics Group (GME) along all these years. So, thank to all of you: Marcelo Johann, Marcus Kindel, João Leonardo Fragoso, Fernanda Lima, Érika Cota, César Zeferino, Eduardo Costa, Jung Choi, Leandro Indrusiak, José Luis Gómez, Rosaldo Rossetti, Alessandro Adario, Márcio Kreutz and so many others. I also thank to all professors of the GME for their

unquestionable contribution to my research profile: Ricardo Reis, Sergio Bampi, Tiaraju Wagner, Altamiro Susin, Luigi Carro, Marcelo Lubaszewski and Flávio Wagner.

During the first four years of this work, including the year I was in France, I had the financial support of CAPES Brazilian Agency (Coordenação de Aperfeiçoamente de Pessoal de Nível Superior), for what I gratefully acknowledge.

Finally, I thank God for providing me with so many opportunities of learning!

To Margarida

Table of Contents

List of Abbreviations

ATPG	Automatic Test Pattern Generation
BDD	Binary Decision Diagram
CAD	Computer-Aided Design
BFS	Breadth-First Search
CMOS	Complementary Metal-Oxide Silicon
Csa	Carry-skip Adder
DAG	Direct Acyclic Graph
DFS	Depth-First Search
EDA	Electronic Design Automation
FTA	Functional Timing Analysis
FSM	Finite State Machine
FUCAS	FUll Custom Automatic Synthesis
iff	if and only if
mdt	Maximal Delay to Node **t**
MSF	Multiple Stuck Fault
ps	picoseconds
SCCG	Static CMOS Complex Gate
sgd	single gate delay
spgd	single pair gate delay
SAT	satisfiability
SSF	Single Stuck Fault
TTA	Topological Timing Analysis
VLSI	Very Large Scale Integration
XDB	Extended Bounded Delay Model
XBD0	Extended Bounded-Zero Delay Model

List of Figures

List of Tables

Abstract

The recent advances in CMOS technology have allowed for the fabrication of transistors with submicronic dimensions, making possible the integration of tens of millions devices in a single chip that can be used to build very complex electronic systems. Such increase in complexity of designs has originated a need for more efficient verification tools that could incorporate more appropriate physical and computational models.

Timing verification targets at determining whether the timing constraints imposed to the design may be satisfied or not. It can be performed by using circuit simulation or by timing analysis. Although simulation tends to furnish the most accurate estimates, it presents the drawback of being stimuli dependent. Hence, in order to ensure that the critical situation is taken into account, one must exercise all possible input patterns. Obviously, this is not possible to accomplish due to the high complexity of current designs. To circumvent this problem, designers must rely on timing analysis. Timing analysis is an input-independent verification approach that models each combinational block of a circuit as a direct acyclic graph, which is used to estimate the critical delay.

First timing analysis tools used only the circuit topology information to estimate circuit delay, thus being referred to as topological timing analyzers. However, such method may result in too pessimistic delay estimates, since the longest paths in the graph may not be able to propagate a transition, that is, may be false. Functional timing analysis, in turn, considers not only circuit topology, but also the temporal and functional relations between circuit elements.

Functional timing analysis tools may differ by three aspects: the set of sensitization conditions necessary to declare a path as sensitizable (i.e., the so-called path sensitization criterion), the number of paths simultaneously handled and the method used to determine whether sensitization conditions are satisfiable or not. Currently, the two most efficient approaches test the sensitizability of entire sets of paths at a time: one is based on automatic test pattern generation (ATPG) techniques and the other translates the timing analysis problem into a satisfiability (SAT) problem.

Although timing analysis has been exhaustively studied in the last fifteen years, some specific topics have not received the required attention yet. One such topic is the applicability of functional timing analysis to circuits containing complex gates. This is the basic concern of this thesis. In addition, and as a necessary step to settle the scenario, a detailed and systematic study on functional timing analysis is also presented.

Keywords: design verification of VLSI circuits, timing analysis, functional timing analysis (FTA), path sensitization problem, critical delay estimation, complex gates, automatic test pattern generation (ATPG), satisfiability (SAT).

TITLE: “ANÁLISE DE *TIMING* FUNCIONAL DE CIRCUITOS VLSI CONTENDO PORTAS COMPLEXAS.”

Resumo

Os recentes avanços experimentados pela tecnologia CMOS tem permitido a fabricação de transistores em dimensões submicrônicas, possibilitando a integração de dezenas de milhões de dispositivos numa única pastilha de silício, os quais podem ser usados na implementação de sistemas eletrônicos muito complexos. Este grande aumento na complexidade dos projetos fez surgir uma demanda por ferramentas de verificação eficientes e sobretudo que incorporassem modelos físicos e computacionais mais adequados.

A verificação de *timing* objetiva determinar se as restrições temporais impostas ao projeto podem ou não ser satisfeitas quando de sua fabricação. Ela pode ser levada a cabo por meio de simulação ou por análise de *timing*. Apesar da simulação oferecer estimativas mais precisas, ela apresenta a desvantagem de ser dependente de estímulos. Assim, para se assegurar que a situação crítica é considerada, é necessário simularem-se todas as possibilidades de padrões de entrada. Obviamente, isto não é factível para os projetos atuais, dada a alta complexidade que os mesmos apresentam. Para contornar este problema, os projetistas devem lançar mão da análise de *timing*. A análise de *timing* é uma abordagem independente de vetor de entrada que modela cada bloco combinacional do circuito como um grafo acíclico direto, o qual é utilizado para estimar o atraso do circuito.

As primeiras ferramentas de análise de *timing* utilizavam apenas a topologia do circuito para estimar o atraso, sendo assim referenciadas como analisadores de *timing* topológicos. Entretanto, tal aproximação pode resultar em estimativas demasiadamente pessimistas, uma vez que os caminhos mais longos do grafo podem não ser capazes de propagar transições, i.e., podem ser falsos. A análise de *timing* funcional, por sua vez, considera não apenas a topologia do circuito, mas também as relações temporais e funcionais entre seus elementos.

As ferramentas de análise de *timing* funcional podem diferir por três aspectos: o conjunto de condições necessárias para se declarar um caminho como sensibilizável (i.e., o chamado critério de sensibilização), o número de caminhos simultaneamente tratados e o método usado para determinar se as condições de sensibilização são solúveis ou não. Atualmente, as duas classes de soluções mais eficientes testam simultaneamente a sensibilização de conjuntos inteiros de caminhos: uma baseia-se em técnicas de geração automática de padrões de teste (ATPG) enquanto que a outra transforma o problema de análise de *timing* em um problema de solvabilidade (SAT).

Apesar da análise de *timing* ter sido exaustivamente estudada nos últimos quinze anos, alguns tópicos específicos não têm recebido a devida atenção. Um tal tópico é a aplicabilidade dos algoritmos de análise de *timing* funcional para circuitos contendo portas complexas. Este constitui o objeto básico desta tese de doutorado. Além deste objetivo, e como condição *sine qua non* para o desenvolvimento do trabalho, é apresentado um estudo sistemático e detalhado sobre análise de *timing* funcional.

Palavras-chave: verificação de projeto de circuitos VLSI, análise de *timing*, análise de *timing* funcional (FTA), sensibilização de caminhos, estimativa do atraso crítico, portas complexas, geração automática de padrões de teste (ATPG), solvabilidade (SAT).

1 Introduction

The remarkable advances achieved by CMOS fabrication technology in the last three decades have made possible the amazing expansion that consumer electronics market has been going through. Thanks to the continuously increasing transistor integration density offered by CMOS technology, ever more complex systems can be integrated on a single chip, allowing more sophisticated electronic equipment to be available at relatively low prices.

Obviously, higher transistor densities are obtained by reducing the dimensions of on-chip components. To a first approximation, reducing transistor and wiring dimensions would lead us to believe that higher clock frequencies could easily be achieved, since smaller transistors switch faster. Indeed, this used to be a well-established law for a long time. However, since CMOS technology has allowed for submicronic devices, some up till then ignored side effects have grown in importance, resulting in new phenomena that partially invalidate the previously mentioned law. One of such phenomena, probably the most cited in recent years, is the dominance of wiring delays over gate delays, as a consequence of the increase in RC factor of interconnections.

Since the 80’s the increasing complexity of electronic systems has made mandatory the use of automatic synthesis tools. Electronic design automation (EDA) tools covering all steps of circuit design, from the behavioral to the physical level, were developed. Such tools incorporated efficient algorithms, able to treat systems with hundreds of thousand gates. However, by the time submicronic technologies began to be used, the models used by these tools for estimating performance were revealed completely wrong. Since then, a lot of effort has been concentrated on developing more accurate circuit models, able to account for the side effects resulting from submicronic technologies.

Among the available EDA tools, those devoted to design verification are currently playing a key role. Since it is not practical to directly prototype the circuit in order to test it on its working environment, designers must rely on verification tools to certify, before fabrication, that the circuit will operate properly. Besides certifying proper operation, one may also desire to explore the target technology in all its extension, looking for achieving the maximal performance. Hence, we can conclude that verification tools must offer a minimum of reliability.

Timing verification targets at determining whether the timing constraints imposed to the design may be satisfied or not. More strictly, timing verification is concerned with estimating the **critical delay** of circuits and the **maximal operating frequency**, in case of clocked circuits.

As any other type of verification, the accuracy of timing verification is completely dependent on the accuracy of the adopted circuit models. By circuit models it is meant not only the physical delay model used to quantify the delay of each component, but also the models for computing circuit component delay and the circuit delay itself. Such models are strongly dependent on the circuit operation model, that is, whether the circuit is assumed to operate in a synchronous or in an asynchronous manner.

Many of the existing timing verification techniques target at synchronous sequential circuits. Thus, let us consider the issues arriving while estimating the maximal operating frequency of a sequential circuit that may be represented as a Mealy finite state machine (FSM). The Mealy FSM model, depicted by figure 1.1, divides the combinational part into

two distinct blocks: the next state logic and the output logic [GAJ99]. The next state logic computes the next state variables while the output logic is responsible for the output signals. If memory elements are edge-triggered flip-flops, then at each active clock edge the next state is loaded into the flip-flops, becoming the current state. At that time, the next state begins to be computed by the next state logic. Outputs may change as a consequence of a change in current state (stored in the flip-flops) or as a consequence of a change at the inputs or even both.

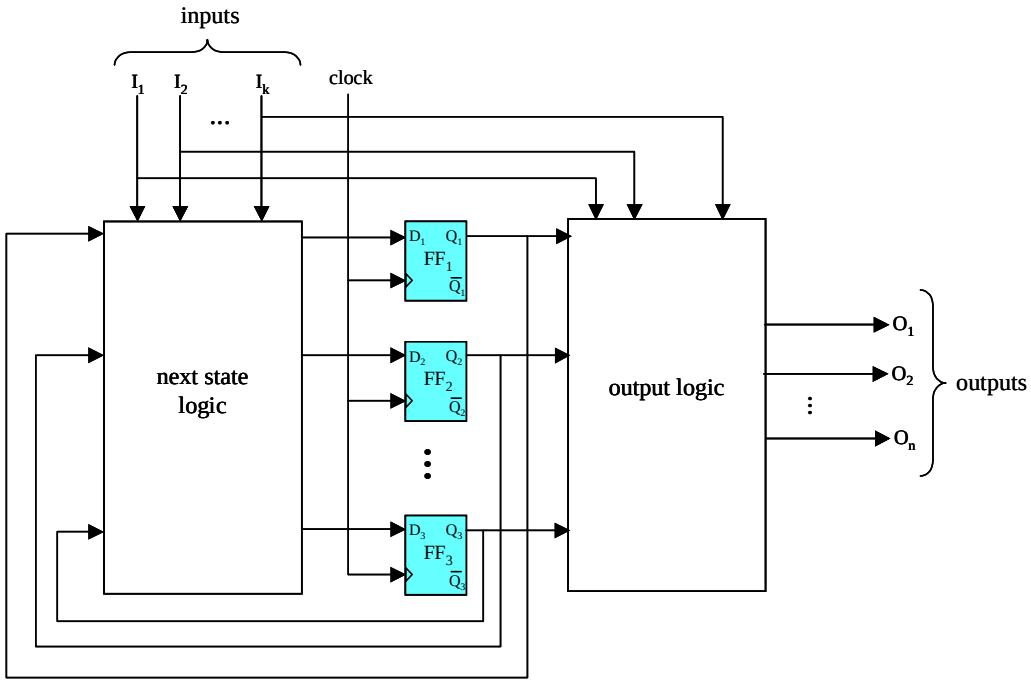


FIGURE 1.1 - Synchronous sequential circuit model.

Consider that the next state logic has *maximum* and *minimum propagation delays* $\mathbf{T_{next}}$ and $\mathbf{t_{next}}$, respectively, while the output logic has *maximum* and *minimum propagation delays* $\mathbf{T_{out}}$ and $\mathbf{t_{out}}$, respectively. Consider also that the edge-triggered flip-flops present *maximum propagation delay* $\mathbf{T_{ff}}$, *setup time* $\mathbf{t_s}$ and *hold time* $\mathbf{t_h}$. Then, in order to assure correct circuit operation, the following conditions must be observed:

- $\tau > \max\{ (\mathbf{T_{ff}} + \mathbf{T_{next}} + \mathbf{t_s}) , (\mathbf{T_{ff}} + \mathbf{T_{out}}) \}$, where τ is the clock period
- $\mathbf{t_{next}} > \mathbf{t_h}$
- circuit's inputs must be stable and valid for a period greater than $\mathbf{T_{next}} + \mathbf{t_s}$ before each active clock edge.

The derived conditions above are quite conservative but allow for a safe synchronous operation. The first condition assures that clock period is long enough to accommodate the worst case delay within the next state loop ($\mathbf{T_{ff}} + \mathbf{T_{next}} + \mathbf{t_s}$) and the worst case delay for the output logic ($\mathbf{T_{ff}} + \mathbf{T_{out}}$). The second condition avoids excessively short clock periods that could prevent flip-flops from sampling valid new states. The third condition assures that the input signals to the next state logic are computed in time, such that all outputs of this block are stable and valid for an amount of time equal or greater than $\mathbf{t_s}$ before the next clock edge.

In fact, the third condition may be disregarded if extra flip-flops are used to synchronize the inputs. Moreover, the first condition is conservative enough to allow the outputs to be sampled using the same clock phase applied to state flip-flops. In case a different phase is available, this condition could be loosen to $\tau > \max\{ (\mathbf{T_{ff}} + \mathbf{T_{next}} + \mathbf{t_s}) , \mathbf{T_{out}} \}$.

Let us go a little bit further on evaluating how circuit operation model may affect the procedure for estimating the clock frequency. Consider that the already discussed circuit operation model is to be adopted and assume that inputs are synchronized by flip-flops that are controlled by the same clock applied to the state flip-flops. If the variation in propagation delays of flip-flops is not significant, one may assume that each combinational block operates in a completely synchronous manner, in that propagation delay is a consequence of two consecutive input vectors. However, if the propagation delays of flip-flops vary significantly, combinational blocks operate in an asynchronous manner, as fast sequences of input vectors were applied to the circuit before its outputs settle to their final values.

Another important issue is the critical delay estimation of combinational blocks, which is a complex task *per se*. The most conservative approach relies on using the topological delay, that is, the delay of the longest path in the circuit. However, more accurate techniques test whether the longest path or paths are able to propagate transitions[1].

The considerations stated in the last two paragraphs are very important for developing timing verification tools that are stimuli-independent or use simplified component models, such as switch level simulators. However, in case of detailed circuit simulation, the circuit operation model is implicitly considered in the context of circuit-level detailed models, such as differential equations or signal waveforms.

Electrical simulation is the most accurate method for verifying the timing requirements of CMOS circuits. Detailed electric level simulators such as Spice [NAG75] represent the circuit as a network of passive and active elements (resistors, capacitors, inductors and controlled sources) and solve the related system of ordinary linear differential equations for each time step of the simulation run. Unfortunately, electric level simulation demands huge execution times even for moderately small circuits. To speedup electrical simulation relaxation methods have been proposed and used (equation relaxation in ELOGIC [KIM86] and waveform relaxation in RELAX [LEL82], for instance). Even though, electrical simulation cannot be used solely for determining time performance of state-of-the-art digital designs.

An alternative to electrical simulation is **timing simulation**. Timing simulation is accomplished by simplified electric level simulators, such as XPSIM [BAU88], or enhanced switch level simulators, such as Motis [CHA75], TV [JOU87] and Crystal [OUS85]. Timing simulation is faster than electrical simulation because it uses less accurate models. In case of Crystal and TV, for instance, effective resistances are used to model transistors. On the other hand, results are less accurate than those obtained through electrical simulation.

There are three serious difficulties in using the simulation approach (electrical or timing simulation) for verifying the timing requirements of circuits. The first is the execution time to accomplish all necessary computations, which was already discussed. A second problem is the effort required for preparing a set of input patterns, since the simulation approach is stimuli driven. Third one, and maybe the most stringent, is ensuring that the set of patterns exercises the critical situation that determines the circuit's critical delay. This constitutes a problem due to the high complexity of current digital designs. For instance, a combinational network with n inputs exhibits 2^n possible input vectors. Even for medium combinational blocks, where n is of the order 100, exhaustive circuit simulation would not be possible. Hence, determining a minimum set of vectors that guarantees to find the circuit delay is not trivial.

[1] This is known as **the critical path problem**, which is discussed along the next chapters of this thesis.

Due to these difficulties, the input-independent approach has replaced simulation for estimating the critical delay of VLSI circuits. This approach, known as **timing analysis**[2], represents each combinational block of the circuit as a weighted direct acyclic graph (DAG), where nodes represent gates and edges represent connections. The weights of nodes and edges represent the delays of gates and connections, respectively. The critical delay of each combinational block is determined by analyzing the length of the paths in the graph.

The most naïve solution relies on disregarding logic behavior of gates and assuming the delay of the longest path as the critical delay of the combinational block. Hence, the critical delay problem of a combinational block is reduced to finding its longest path, which can be solved in linear time by the well-known topological sort algorithm [COR90]. Such approach, referred to as **static** or **topological timing analysis** (TTA), was probably born with the IBM PERT Project [KIR66] and was used by other timing analyzers such as [HIT82].

However, there may not exist any input pattern that exercises the longest path in the circuit, or equivalently, it may never transmit any signal transition. In this case, the critical delay may be smaller than the delay of the topologically longest path. Paths that never transmit a signal transition are called **false paths** [HRA78] (or **unsensitizable paths**). A circuit may contain many false paths. In order to improve the accuracy of delay estimates, timing analysis tools must take path sensitizability into account. Unfortunately, performing false path-aware timing analysis constitutes a NP-complete problem and thus, many assumptions must be done in order to obtain safe critical delay estimates.

Although some work has been done for allowing automatic generation of false path-free circuits (e.g., [KEU91][SAL94][KUK97a][PRA00]), most of available high-level synthesis systems may generate circuits with false paths [BER91].

In late years a lot of research has focused on developing efficient timing analysis algorithms that consider the path sensitization problem. But as long as CMOS technology evolves very fast and higher clock rates are continuously being demanded, improving the accuracy of critical delay estimation is still an issue of relevant importance in design verification.

Furthermore, some specific topics have not received sufficient attention yet. One such topic is the applicability of **functional timing analysis** (FTA), i.e., false path-aware timing analysis, to circuits containing complex gates. Some works on logic synthesis have reported the possibility of area reduction and performance improvements when static CMOS complex gates (SCCGs) are used [REI98][RIE96]. However, timing analysis of circuits containing complex gates is rarely mentioned in the literature. Indeed, only some of the existing algorithms can handle such circuits. The study of functional timing analysis applied to circuits containing complex gates is the main concern of this thesis.

1.1 Thesis Organization

This thesis is organized as follows. Chapter 2 reviews the basic issues of the timing analysis approach. Topological timing analysis is discussed in detail and the reason why it may furnish too pessimistic estimates is addressed. Functional timing analysis is then introduced. The component delay models and the circuit delay computation models underlying FTA tools are presented. The basic properties to assure that FTA furnishes safe critical delay estimates are also presented.

[2] In this thesis, the term **timing verification** is used to refer to any method that can verify timing requirements of circuits. However, the term **timing analysis** will only be used for input-independent timing verification methods.

Chapter 3 presents a comprehensive collection of definitions that are necessary to understand the theory behind FTA and the related algorithms.

Chapter 4 is devoted to the path sensitization problem. The three most important sensitization criteria, the static sensitization, viability and the exact floating sensitization are presented.

The main issues that are considered in the development of a FTA tool are presented and classified in chapter 5. This includes the adopted sensitization conditions and the method used to determine whether the sensitization conditions are satisfiable. This chapter also discusses some of the most important existing algorithms.

Chapter 6 justifies the use of CMOS complex gates in the design of VLSI circuits. It also discusses the limitations of existing timing analysis methods for estimating the delay of circuits containing such type of gates. A new solution for performing functional timing analysis of circuits containing complex gates is then proposed. This solution relies on extending the three-valued timed calculus used within the timed-test generation procedure of Devadas et al. [DEV93a] in order to handle complex gates in a friendly manner (i.e., without using macro-expansion). The advantages and disadvantages of the proposed solution are discussed.

Finally, some concluding remarks are offered.

The appendices bring extra information on specific topics. Appendix 1 justifies the importance of taking into account false paths by studying false paths in carry-skip adders. Appendix 2 presents an informal evaluation of the time complexity of the best-first algorithm, used by path enumeration-based FTA tools. Appendix 3 is an extended abstract of the thesis in portuguese.

This thesis may also be used as a quick introduction to the basic concepts on timing analysis and timing models used within high level and logic level synthesis algorithms. Chapters 1 and 2 introduce most of the necessary concepts without using any formalism. For those interested in going further, the definitions presented in chapter 3 are essential, though.

2 The Timing Analysis Approach

During the 80's, as design complexity augmented quickly, the verification of timing requirements of circuits through simulation became impractical due to the increasing execution time demanded and also to the difficulty in determining a safe set of input vectors that proven to exercise the critical delay situation. Then, the attention was turned to stimuli-independent verification methods, in opposition to simulation methods.

But even before, more precisely in 1966, Kirkpatrick and Clark [KIR66] proposed the use of the management method called PERT (Program Evaluation Review Technique) to estimate the critical delay of circuits. In their work, a combinational block was represented as a weighted direct acyclic graph (DAG), with the nodes representing circuit gates and the edges representing circuit connections. The longest path in the DAG was discovered by using the topological sort algorithm [COR90] and its delay was assumed to be the critical delay of the block.

The idea of using PERT to estimate critical delay of combinational circuits was retaken by Hitchcock in the development of its "Timing Analyzer" program [HIT82], which also came up with the innovative concept of signal time slacks. Also the expression **timing analysis**, currently used by EDA community to designate any input-independent timing verification tool, seems to be borrowed from Hitchcock's work.

However, there may not exist any input pattern[3] that exercises the longest path in the circuit, in the sense that no signal transition (or event) can propagate along it. Since PERT-based timing analysis disregards the logic behavior at circuit's nodes, it may furnish a needless pessimistic critical delay estimate. In addition, the fact that PERT-based timing analysis considers only circuit topology has motivated some authors to call it **topological timing analysis** (TTA) (e.g., [DEV94]).

Paths that never transmit any signal transition are called **false paths** [HRA78]. (Some authors also use the term **unsensitizable paths**.) A circuit may contain many false paths. The problem of determining whether a path may be exercised or not is referred to as **the path sensitization problem** or as **the (general) false path problem** [DU89]. According to [MCG89], although false paths were known for some time, the first complete discussion on this topic was due to Hrapcenko, who designed a parametric circuit to show that for some circuits the true delay could differ from the topological critical delay [HRA78].

Early timing analysis tools took into account false paths by allowing **case analysis**. The Timing Analysis program (TA) from Hitchcock [HIT82], for instance, had a facility called delay modifiers that could be used to indicate paths that, in the designer's opinion, would never be activated. However, as long as manual detection of all false paths in complex circuits is impossible, since early it became clear that automatic false path detection should be pursued. The first attempts for including automatic false path detection was the work of Brand and Iyngar [BRA88] and that of Benkoski et al. [BEN87]. Since then, most of research in timing analysis has concentrated on this issue, looking for algorithms that could lead to accurate delay estimates in a reasonable amount of time. More recently, timing analysis

[3] The majority of existing timing analysis techniques do not consider explicitly all possible input patterns. Actually, the definition of which patterns are allowed depends on how inputs are assumed to be made of, what is taken into account by the adopted circuit delay computation model (see section 2.3).

techniques that consider false paths have been termed **functional timing analysis** (FTA) [ASH95][KUK97]. Such terminology is also adopted in this text.

This chapter is concerned with the basic aspects underlying the timing analysis approach, with special attention to the models used in FTA. Section 2.1 describes how the topological sort algorithm is adapted to perform TTA. In section 2.2 the false path problem is introduced by means of two circuit examples. Section 2.3 discusses delay computation models in the context of FTA, showing that circuit delay computation depends on the type of inputs considered. It also associates the possible types of inputs to circuit operation models and to the so-called modes of operation. Section 2.4 summarizes the most relevant physical component delay models, while section 2.5 presents the gate delay computation models commonly used in FTA tools. Section 2.6 discusses the correctness and robustness properties of timing analysis algorithms. Such properties allow us to evaluate whether a given FTA technique is able to furnish safe delay estimates or not and in affirmative case, how accurate such estimates are. Finally, section 2.7 gives an overview on the spectrum of model possibilities while implementing a FTA tool.

2.1 Topological Timing Analysis

Any timing analysis technique represents each combinational block as a weighted DAG. In this DAG nodes are associated with gates and edges are associated with connections (sometimes, nets). The delay of each gate (connection) may be stored at the respective node (edge). Another possibility is to concentrate (or *lump*) at each node (or edge) the sum of the delays of the gate and its output connections. This choice depends on the data structure used to store DAG information and on the accuracy of physical models used to estimate component (gates and connections) delays. Dummy nodes, i.e., nodes with zero delay, represent primary inputs and primary outputs. Frequently, *source* and *terminal* (dummy) nodes are added to the DAG to transform it into a canonical DAG. This may help the development of graph traversal functions that operate both forward and backwards. In a canonical DAG representation, any **complete path** assumes the form (**s**, e_s, v_0, e_0, v_1, e_1, …, v_n, e_n, v_{n+1}, e_t, **t**), where v_0 and v_{n+1} represent the primary input and the primary output, respectively, **s** and **t** are the source and terminal nodes and e_s and e_t are dummy edges. Figure 2.2 shows a DAG representation for the circuit of figure 2.1.

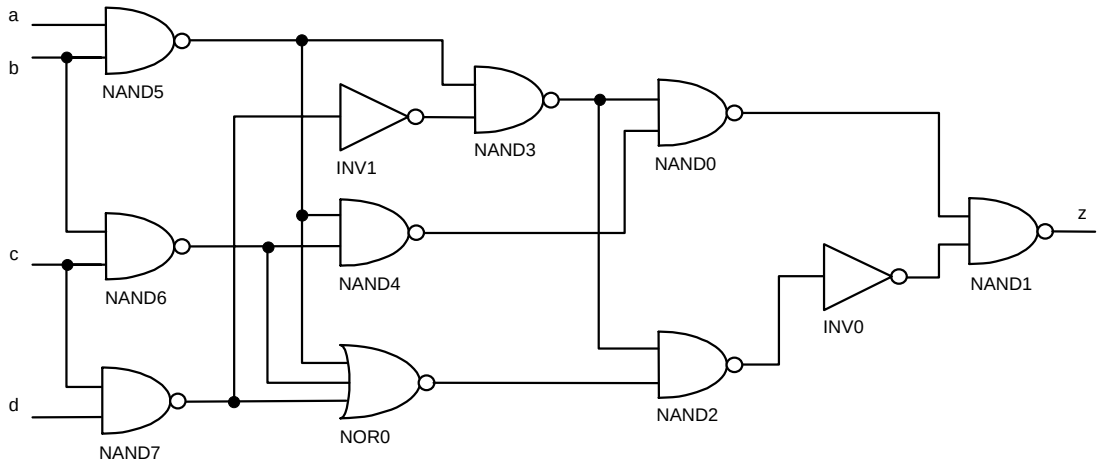


FIGURE 2.1 - Combinational circuit example.

Topological timing analysis finds the longest path in the DAG, which corresponds to the path with greatest delay, and assumes it as responsible for the critical delay of the circuit. This is accomplished by using the topological sort algorithm [COR90], which is known to execute in linear time with respect to the graph size. It also may compute the time slack of signals, which can be used in a performance optimization step.

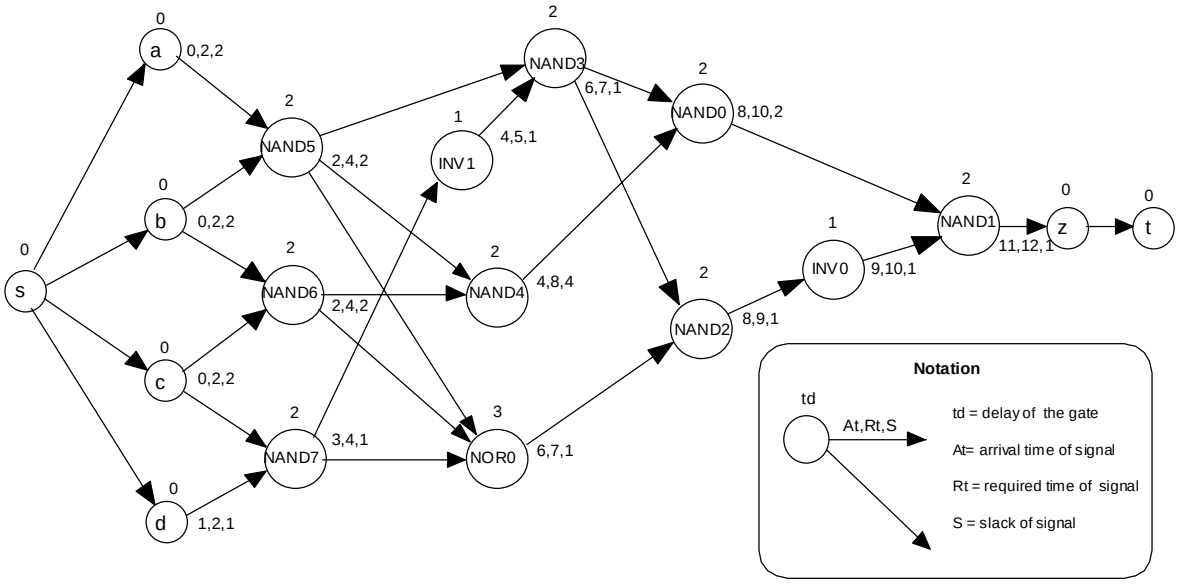


FIGURE 2.2 - Processed DAG for the circuit example of figure 2.1.

Let us illustrate the topological timing analysis procedure on the circuit of figure 2.1. (DAG of figure 2.2 holds the timing information gathered from such procedure.) For the current example, assume that graph edges represent circuit nets (instead of connections). Assume also that the delay of the gates and their output nets are lumped at graph nodes. For any circuit signal ***e*** a 3-tuple of timing values is calculated and annotated at the related graph edge: the **arrival time** of ***e***, At(***e***) which is the time that the signal at edge ***e*** settles to its final (steady state) value, the **required time** of ***e***, Rt(***e***), which is the time at which the signal at ***e*** is required to be stable and the **slack** S(***e***), calculated as the difference between the required time and the arrival time (Rt(***e***)-At(***e***)). The arrival times of the primary inputs and the required times of the primary outputs are set by the timing constraints for the block under analysis. Non-zero arrival times for the primary inputs may be necessary in case of hierarchical analysis. The topological analysis begins by computing the arrival times for each signal in a forward manner, beginning from the primary inputs. The arrival time of a signal *e*, which is the output of node *v*, is calculated by:

$$\mathrm{At}(e) = \max\nolimits_{\mathrm{i}} \{\mathrm{At}(e_{\mathrm{i}})\} + \mathrm{d}(v) \qquad (2.1)$$

where d(*v*) is the delay of the gate represented by *v* and e_{i} is the set of input signals to *v*. Once the arrival times of all primary outputs are calculated (or alternatively, node **t** is reached), the procedure performs a backward step for calculating the required times of signals, using the required times of primary outputs. The required time of a signal *e* is calculated by:

$$\mathrm{Rt}(e) = \min\nolimits_{\mathrm{j}} \{[\mathrm{Rt}(e_{\mathrm{j}}) - \mathrm{d}(v_{\mathrm{j}}))]\} \qquad (2.2)$$

where v_{j} represents the gates that have *e* as fanin and e_{j} are the output edges (nets) of such gates. Having calculated the required time of a signal, its slack can also be computed.

Sometimes this computation of arrival times, required times and slacks is called **delay trace** through the network [DEV94].

Once the graph has been completely processed, the **topologically longest path** or **topological critical path** is the path where each signal has the minimum slack and can be easily traced. In this example the topological critical path is (**s**, d, NAND7, NOR0, NAND2, INV0, NAND1, z, **t**), with delay 11 and slack 1. The slack is also a measure of the criticality of paths and may be used for identifying gates for resizing and/or buffer insertion points in case of a delay optimization procedure [JU91][JOU87][CHE93a].

Although topological timing analysis may overestimate the critical delay of circuits, it surely is an upper bound on the critical delay of a combinational circuit. Indeed, most of existing high-level and architectural-level synthesis tools still use it as a fast means of verifying the timing requirements of designs, since at these levels there is an implicit lack of accuracy in physical delay models of components. However, as operating frequency of VLSI designs enters the gigahertz range, more accurate delay estimates are demanded. Hence, considering path sensitizability is mandatory. In the next section the false path problem is introduced by means of two circuit examples.

2.2 False Paths

Currently, timing, power, area and testability are the criteria used to guide design optimization during synthesis. Unfortunately, most of optimization procedures introduce redundancies into designs [KEU91], which constitute one known source of false paths.

If input vector *v* does not activate a path P, then P is said not to be sensitizable by *v*. If P is not sensitizable by any input vector, then P is said to be false.

Consider Hrapcenko's circuit [HRA78] shown in figure 2.3. Assume that all gates have delay equal 1 and wires have zero delay. Its topologically longest paths are P_1=(i_1, G_1, G_2, G_3, G_4, G_6, G_7, G_8, h) and P_2=(i_2, G_1, G_2, G_3, G_4, G_6, G_7, G_8, h), both with delay 7. However, while i_3=0, no signal transition can propagate from a to b. Similarly, while i_3=1, no signal transition can propagate from d to f. As long as a signal cannot assume two logic values at the same time, the only possibility to sensitize P_1 and P_2 is allowing a sequence of vectors to be applied at circuit's inputs such that i_3=1 at time 1 and i_3=0 at time 4. However, if only pairs of vectors are allowed, then the previous case is not possible. Then, paths P_1 and P_2 are false and the circuit delay is less than 7.

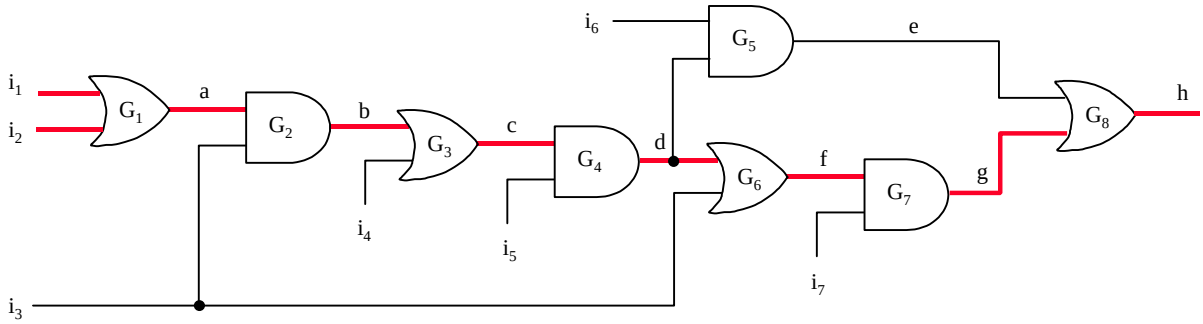


FIGURE 2.3 - First example of false path: Hrapcenko's circuit.

Some classes of circuits are designed to purposely make the topologically longest paths false, as a strategy for reducing its critical delay. One such a class is composed of carry-skip adders [KEU91][DEV94][LAM94]. Figure 2.4 shows a 2-bit carry skip adder (csa2). Higher order adders may be obtained by connecting together csa2s through the carry chain. In this case the topologically longest path will include the carry chain, which in a single csa2 as the one showed by figure 2.4 corresponds to path P=(c0, n0, n1, n2, n3, n4, c2), in bold. Thus, to determine the critical delay of a higher order csa adder, one must begin by analyzing the sensitizability of such path.

Suppose all gates of the csa2 of figure 2.4 have unit delay and wires have zero delay. Assuming only pairs of vectors at the circuit's inputs, in order to sensitize path P, p0, g0, p1, g1, ctrl_n and n5 must be 1 at times 0, 1, 2, 3, 4 and 5, respectively. However, if p0, g0, p1 and g1 are made fixed at 1, ctrl_n will be fixed at 0 and no transition will reach n4. On the other hand, no input pair of vectors satisfies p0=1 at time 0, p1=1 at time 2 and ctrl_n=1 at time 4. Thus, path P is false.

Moreover, late transitions at c0 (i.e., transitions arriving after time t=0) do not change the sensitizability of path P. This means that in any higher order adder made up from this csa2 the topologically longest path is false. Appendix 1 presents a detailed *ad hoc* analysis of csas, using fixed non-unit delay and assuming different fall and rise gate delays.

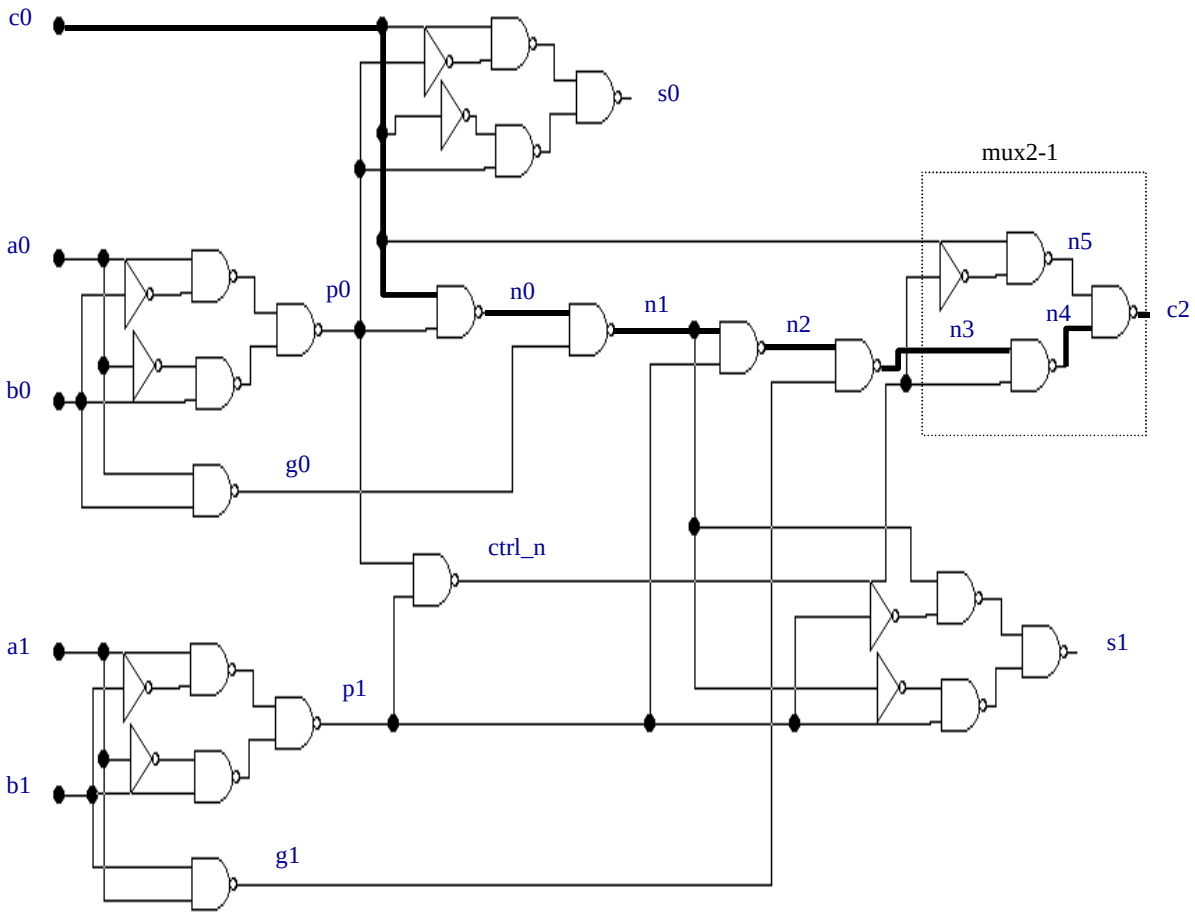


FIGURE 2.4 - Second example of false path: a 2-bit carry-skip adder.

2.3 Functional Timing Analysis and Circuit Delay Computation Models

As mentioned in chapter 1, the accuracy of any timing verification tool is completely dependent on the accuracy of the adopted circuit models. Furthermore, in the specific case of timing analysis tools, some assumptions on circuit operation are also required to improve critical delay estimation, what will become clear in the following paragraphs.

The topological timing analysis procedure presented in section 2.1 may lead to significant overestimation of critical delay because it ignores the logical behavior of circuit gates. Considering different fall and rise gate delays in that procedure is easily accomplished and may result in significant improvement of the estimation accuracy [GÜN98][GÜN98a]. In a certain manner, the use of different fall and rise gate delays may be seen as a very crude model for circuit operation, which, however, does not take false paths into account.

Due to the false path phenomenon, the critical delay of a circuit may be less than the delay of its topologically longest path. The difference between the topological delay and the actual critical delay cannot be neglected in the context of an automated design process since the designer does not have total control on the resulting generated structure. Further, even for some hand-made designs the delay estimated by a topological analysis may be too pessimistic and more accurate estimations would be highly desirable. (That is the case of carry-skip adders [KEU91][LAM94][DEV94] and some multipliers.)

In order to take false paths into account, it is necessary to revise the concept of critical delay of a (combinational) circuit. A path-based definition would state that “the **critical delay** of a circuit is the delay (or length) of its longest sensitizable path”, also mentioning that “there may exist more than one critical path” [CHE93]. Although correct, this definition is quite limited since not all of the false path-aware timing analysis algorithms work on a per-path basis.

A more general definition can be found by examining the circuit operation model and realizing that the essence of timing analysis should be the capture of the exact instant at which the slowest circuit output(s) settle to its (their) steady-state value(s). Of course, one possible solution relies on testing the sensitizability of each circuit path, beginning by the longest one. However, this is not the only possible solution. Indeed, state-of-art timing analysis techniques (or algorithms) operate on sets of paths at a time, what generally results in less computation time. Timing analysis algorithms that considers false paths fall into the **functional timing analysis** (FTA) category, and will be discussed in chapter 5.

Having posed the FTA problem from a new point of view, it is possible to redefine the delay of a circuit. The **delay of a circuit** under a given input pattern is the minimum amount of time after which all of its outputs are settled to their steady-state values. By extension, the circuit’s **critical delay** corresponds to its greatest delay, considering all possible input patterns. For the sake of simplicity, we may refer to the critical delay of a circuit just as **delay of the circuit (**or **circuit delay)**, since this is the parameter of interest in this thesis.

Note that the previously stated definitions implicitly consider path sensitizability. However, what exactly constitutes an input pattern is not clear. This point was intentionally left open in order to make the definition of critical delay independent of the circuit operation model. In reality, the definition of what constitutes a valid input pattern depends on how the circuit is assumed to operate and will be considered within the **circuit delay computation model**.

The concept of circuit delay computation model was motivated by the observation that the delay of a circuit depends on the nature of its inputs, that is, whether the inputs are

assumed to be made of **pairs of vectors** or **sequences of vectors**. To illustrate this, consider the test circuit of figure 2.5a (borrowed from [LAM94]), with gate delays as assigned inside each gate. The longest path of this circuit is (a, c, f, y), with delay 5. If the inputs are considered to be made of pairs of vectors, then the latest output transition occurs at t=1. A possible input combination that generates a latest output transition is shown in figure 2.5b. On the other hand, if the inputs are considered to be made of sequences of vectors, then the latest output transition occurs at t=3. A possible input vector sequence that leads to this late output transition is shown in figure 2.5c. These results confirm that circuit delay may differ according to the type of applied inputs.

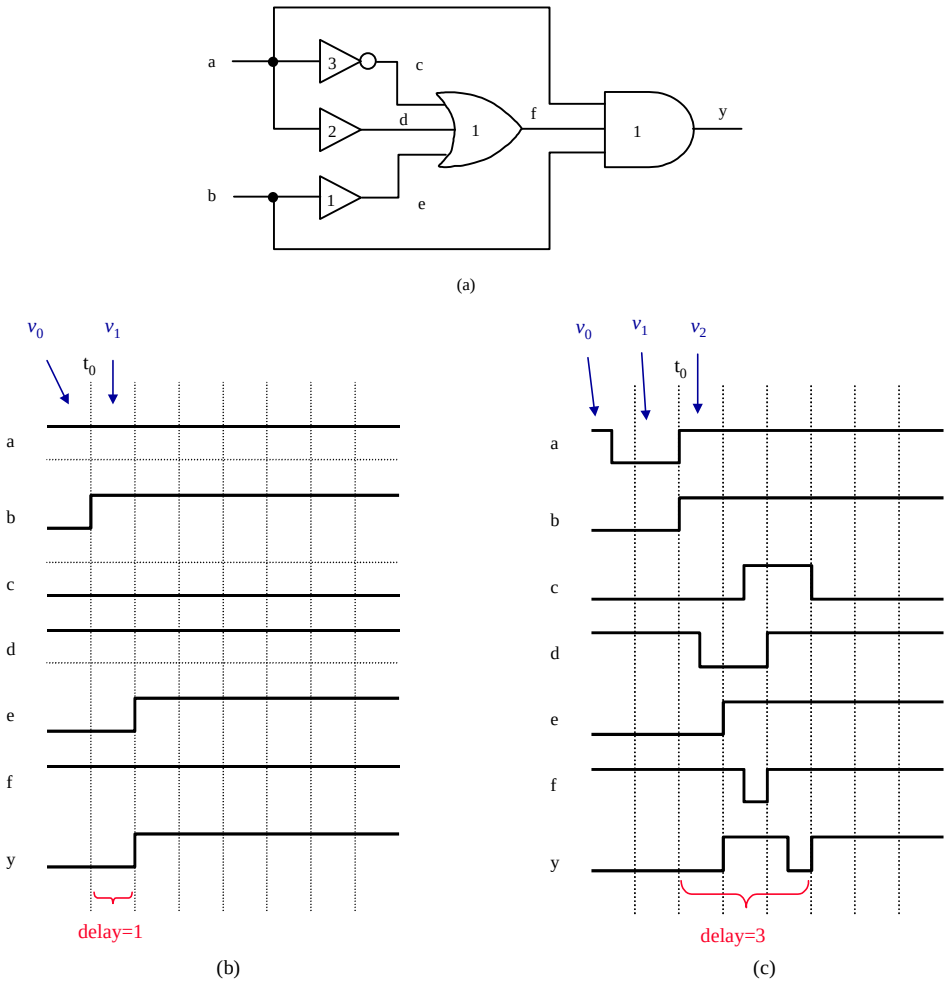


FIGURE 2.5 - The delay of circuits depends upon the type of inputs considered.

In order to compute the delay by a pair of vectors, it is assumed that a vector v_1 is applied at t=–∞ and a second vector v_2 is applied at t=0. The delay of a circuit is then defined as the maximum arrival time of the last output transition over all possible pairs of vectors. From the definition it becomes clear that all circuit nodes are assumed to be settle to their stable values with respect to v_1 by the time vector v_2 arrives.

Using the delay by pairs of vectors to estimate the circuit's critical delay is equivalent to assuming that the circuit operates in a fully synchronous manner. This type of operation has been referred to as the **transition mode** and the critical delay thus obtained is called **transition delay** of the circuit [DEV92].

It has been conjectured that the transition delay is the exact delay of a circuit [SIL99]. Indeed, this would be true if one could always guarantee a fully synchronous operation of combinational blocks within synchronous circuits. However, memory elements may present different propagation times that can lead to the misalignment of inputs. Also, a supposed benefit of the transition delay method is that, besides the delay estimation, it generates the pair of vectors responsible for this delay, thus allowing the certification of this delay through circuit simulation.

An implementation of transition delay calculation is proposed in [DEV92] and [DEV94a]. It uses symbolic simulation along with some sophisticated delay computation model. Further improvements concerning event suppression were proposed in an attempt to reduce the time complexity of this symbolic simulation based implementation [DEV94b]. The transition delay method presents severe disadvantages that have prevented its use in practical FTA tools, however. Firstly, the search space for determining the critical input vector pair situation is 2^{2n}, with n being the number of primary inputs to the circuit. Secondly, the delay computation model used, the bounded model (see section 2.4), makes the symbolic simulation extremely expensive from the computation's point of view.

The problems with delay by pairs of vectors (transition mode) have motivated the massive use of the **single vector** approach. The single vector approach is an approximation of the delay by sequences of vectors. It conservatively assumes the nodes of a circuit to be arbitrary, i.e. "floating", before they are settle by a single input vector *v*. The delay of the circuit is the latest settling time over all possible single vectors v_i. The assumption of floating nodes comes from the fact that the circuit may be still propagating the input vectors applied before *v*, what would cause the circuit nodes to be floating. In the literature, the assumption of the single vector approach is incarnated by the designation of "**floating mode of operation**" [CHE91] and the delay thus obtained is referred to as the **floating delay** of the circuit.

The only known implementation of delay by sequence of vectors is presented in [LAM93] and [LAM94]. It is based on a sophisticated formalism called **timed Boolean functions** (TBFs), which is claimed to unify the Boolean and the timing behavior of circuits. The problem with this implementation is that due to the complex formulation, it demands significant computation effort, mainly if mode detailed delay models are to be used.

Although delay by sequences of vectors would be desirable, the majority of existing FTA algorithms adopts the single-vector (floating) delay computation model due to its ease of implementation and smaller computational cost. Moreover, it has been reported in [LAM94] that for most practical circuits made of simple gates the delay by sequences of vectors and the single vector delay are coincident. And for the cases where they are not coincident, single vector delay is an upper bound on the actual circuit delay, while the delay by sequences of vectors is the exact delay. Also, single-vector techniques may use many of already existing test generation algorithms.

2.4 Component Delay Models

Component delay models refers to the **physical model** (also called circuit-level model) used to estimate the delay of each circuit component, thus obtaining individual delay information that will be used to determine the delay of the circuit as a whole. This information is generally expressed in terms of equations that use parameters derived from extensive transistor-level and/or device-level simulation of circuit components.

Logic synthesis tools as SIS [SEN92] generally use the linear delay model, which is the simplest physical model. In the linear delay model of SIS, for instance, the delay across a pin ***i***, d(***i***), of a gate is given by:

$$d(\boldsymbol{i}) = A(\boldsymbol{i}) + B(\boldsymbol{i}) \times C_L(\boldsymbol{i}) \qquad (2.3)$$

where A(***i***) is called **transport delay** for the gate at pin ***i***, B(***i***) is the inverse of the drive capability of the gate and $C_L(\boldsymbol{i})$ is the total capacitive load of the net represented by ***i***, lumped at the output of the gate. Although this simple model does not take into account some important features of CMOS technology (e.g., short channel devices, slow input ramps and body effect), it is still used by the majority of logic synthesis tools because it does not require too expensive computations.

However, the study of more sophisticated delay models appears as a major subject of the work done in the timing verification field during the first half of the 80's. Probably, this is due to the fact that these works were directly related to the development of new microprocessors. This was the case of Ousterhout's **Crystal** [OUS85] and Jouppi's **TV** [JOU87], which were used in the timing validation of the RISC II and MIPS microprocessors, respectively.

Crystal divides the circuit into *stages* to evaluate the delay. A *stage* is defined as *a chain of transistors and nodes between a signal source and a place where the signal is used*. Hence, a stage may represent not only the MOS transistor chain up to the gate output but also any pass transistor chain being driven by the gate. A natural consequence of this switch-level modeling of MOS circuits is that the delay of connections would be implicitly considered within stages. Crystal has three physical delay models: the **lumped RC** model, the **lumped slope** model and the **distributed slope** model. In the lumped RC model each transistor type is characterized by two resistance values, being one for the case when the transistor is transmitting a logic 0 and another when it is transmitting a logic 1. These values are given in Ohms per square and are multiplied by the transistor's W/L to obtain the *effective* resistance. The delay through a stage is computed by lumping all resistances and capacitances. The lumped slope model incorporates information about waveform. Each waveform is represented by its inversion time and its rise time. This tries to model the effect of the load being driven by the stage on the effective resistance, leading to a more accurate delay estimation. The lumped model is too pessimistic in estimating the delay of distributed capacitances. The distributed slope model is similar to the slope model except that, instead of assuming all RC lumped, it uses the results of Penfield and Rubinstein [RUB83] for the delay of RC trees.

TV uses also the Penfield-Rubinstein model but with two simplifications: only the waveform estimate is computed, instead of the bounds, and only one path is assumed open through a tree of pass transistors. This reduces the accuracy of the model because the influence of the input ramp slope is only roughly considered.

The formerly cited physical models use linear elements to approximate the behavior of transistors which is non-linear in its essence. Thus, it is not a surprise that a significant error may occur. Looking for more accurate delay estimations, Horowitz has proposed the use of non-linear elements for dealing with both transistors and interconnection networks [HOR84]. In his formulation the input ramp slope is modeled by a hyperbolic function while the transistor is modeled by a simple quadratic function.

A third possibility for delay modeling is to use explicit formulation. According to this approach the delay is estimated by using a closed formulation that includes parameters for technology characterization. Some of these parameters may be obtained from device extraction while others may come from exhaustive electrical simulation. But once the target technology is appropriately characterized, the delay of circuit components is estimated only by

using the formulae, without any other supporting means as, for instance, electrical simulation. Examples of explicit analytical formulations are the alpha-power model [SAK88] and the explicit formulation of [DES88].

In the explicit formulation of [DES88] the delay of an inverter with minimum W and L is used as a standard measure for a given technology. To this delay quantity various correction terms may be applied in order to generalize the delay estimation to nor and nand gates, possibly with Wn/Wp≠1. To consider the effect of slow input ramp a correction term was added latter to the formulation [AUV90]. Recently, the formulation was revised in order to account for the effects of submicronic technologies [DAG99].

Recent advances in CMOS technology have shrinked device dimensions to nanometers. Many electrical effects that were formerly disregarded in micronic technologies became very important in current submicronic (or nanometric) technologies and constitute sources of errors that may compromise the correct operation of complex designs. A practical example of such phenomenon is the effect of circuit interconnections. In submicronic technologies the delay of interconnections is considerable and frequently dominates the delay of circuit gates themselves. Hence, physical delay models for interconnections begin to play a very important role in the delay estimation. The basic models for interconnection analysis are those that evaluates RC networks as the Elmore's [ELM48], Sakurai's [SAK83], Penfield-Rubinstein [RUB83], Horowitz [HOR84] and also the models of Crystal. With the advent of submicronic technologies interconnection analysis has come into focus again and several works that simultaneously consider gate and connection delays have been published. A referential work was the interconnection analysis program called RICE [RAT94], that uses a technique known as AWE (Asymptotic Waveform Evaluation), which is based on the moments of the impulse response. Currently, several other works are being developed for modeling the delay of gates and connections for submicronic technologies (e.g., [FOR97][HIR98]).

A more detailed discussion on physical models for component delay calculation is beyond the scope of this work. A good review on this issue may be found in [UEB95], which also presents an analytical semi-empirical delay model that uses a latency time and an effective linear resistance to model the exponential region of the output curve of a stage.

2.5 Gate Delay Computation Models

The simplest gate delay computation model is known as the **fixed delay model** [DEV94]. It assumes that the delay of a gate is a fixed value d. A natural extension to this model is to assign a fixed delay value d_i to each input i of a gate. Another variation is to consider separate falling and rising delays by either assigning a single pair of delays [df,dr] to the gate or assigning a pair of delays [df_i,dr_i] to each input i. The assignment of a fixed delay or a pair of fixed delays per input is also referred to as **pin-to-pin** delay.

One important issue that arrives in timing analysis is that it is not intended to provide the critical delay of a single manufactured instance but the delay of the entire family of manufactured circuits of the same design. In this sense, the use of TTA along with maximal individual gate delays leads just to an upper bound on circuit delay. However, current submicronic designs demand more accurate delay estimations (at least tighter upper bounds), what, from the computational's point of view, can be accomplished by using the FTA approach. On the other hand, the use of maximal gate delays in FTA is not sufficient to assure that the estimated circuit delay really represents an upper bound over the entire family of

manufactured circuit instances: due to the sensitization phenomenon, maximal individual gate delays may result in an underestimation of circuit delay, what would be an unacceptable erroneous prediction. This phenomenon is known as the **monotone speedup failure** [MCG91] and will be commented in more detail in the next section.

In order to assure that FTA will not underestimate circuit delay, gate delays must be specified within a bounded interval $[d^{min},d^{max}]$, where d^{min} and d^{max} represent the minimum and the maximum delay of the gate, respectively. This is the so-called **bounded delay model** and is the model underlying the transition delay computation. The bounded delay may also be assigned in pin-to-pin format. A modification on the bounded delay model consists of considering $d^{min}=0$, which is commonly referred to as **unbounded delay model** (meaning unbounded bellow)[4]. As it will be shown in the next section, this is the model used for computing the floating delay under the monotone speed up property.

The previously described gate delay models implicitly divide signal magnitude into two values, either 0 or 1. Obviously, a more detailed analysis could be performed if signals were represented with a multi-valued logic. In the case of a ternary logic, for instance, a signal may assume one among the values (0,1,X) at a time, where X represents signal magnitudes between the 0-threshold and the 1-threshold. Ternary logic has been used for hazard detection and logic simulation [SEG89]. In [MCG93], ternary logic was formalized to be used in the FTA context, with the addition of the third state (X) to both bounded and unbounded models, originating the **extended bounded delay model** (XBD) and the **extended bounded-zero delay model** (XBD0), respectively.

2.6 Robustness and Correctness of FTA algorithms

In order to furnish a safe delay estimate any FTA algorithm must satisfy two properties: **robustness** and **correctness**. By safe estimate it is meant a delay value that reflects a tight upper bound on the delay of all design instances after fabrication. (In fact, the later statement is an informal definition of the correctness property.)

The robustness property was originally named monotone speedup property by McGeer and Brayton [MCG89][MCG91]. It basically says that the use of maximal individual gate delays does not guarantee the estimated delay to be an upper bound on the actual delay of the whole family of manufactured circuit instances. As a consequence, the use of such an estimated delay value would be useless for design validation, since it may exist one or more circuit instances exhibiting greater delays.

Let us examine the circumstance under which the robustness property would fail. In the analysis of a single design instance, the assumption of maximal individual gate delays may lead to a sensitization situation in which the longest path declared as sensitizable exhibits a delay which is smaller than that of the topologically longest path. It means that there is at least one unsensitizable path with delay greater than the circuit's topological delay. Suppose that a second instance of the same design is to be analyzed. Suppose also that one or more gates in this other instance present individual delays which are smaller than their correspondent maxima. Due to Boolean and timing relations between circuit gates, the path sensitizability analysis may lead to a situation in which the path declared as sensitizable exhibits a delay which is greater than the delay of the longest sensitizable path of the former analyzed

[4] Maybe, a more appropriate designation would be bounded-zero delay model.

instance. This failure in circuit delay estimation was originally called the **monotone speedup failure** [MCG89].

Let $\{C_1,C_2,\ldots,C_n\}$ be n fabricated instances of a combinational circuit C. The robustness property can be defined as follows. (Taken from [SIL99], with modifications.)

Definition 2.1: **robustness property**

> For each true path P in C_i, (i=1, 2 ,…, n) there must exist a true path Q in the slowest C_i such that $d(Q) \geq d(P)$, where d(P) (d(Q)) is the length or delay of path P (Q).

By true path it is meant exactly (floating-mode) sensitizable path (see subsection 4.2.4).

Assuring that a FTA algorithm follows the robustness property is not sufficient, however. As observed by several authors (e.g., [MCG89], [CHE93], [PES94]), it is imperative not to underestimate the critical delay of a circuit. Otherwise, any of the fabricated instances may present an erroneous operation.

The correctness property may be defined as follows. (Taken from [SIL99]).

Definition 2.2: **correctness property**

> For each true path P in C, there must exist a path Q such that $d(Q) \geq d(P)$, where d(P) (d(Q)) is the length or delay of path P (Q), which is sensitizable under the adopted set of sensitization conditions.

Equivalently, the property says that the set of conditions used to test paths sensitizability must not underestimate the delay of the critical path.

It is clear that while the robustness property refers to the gate delay computation model, the correctness property concerns the circuit delay computation model. The correctness property also appears as “critical path correctness” in [CHE93] and as “delay correctness” in [PES94]. Both [CHE93] and [PES94] also refer to the path sensitization correctness property, according to which a FTA algorithm (and the embedded set of sensitization conditions) is considered correct if it may not declare a true path as unsensitizable.

The previously presented properties are used in the evaluation of the existing FTA techniques, including the set of conditions used to test path sensitization.

2.7 Delay Computation Models, Path Sensitization and FTA Algorithms

Since the middle 80's a lot of work has been developed focusing on algorithms for accurately determining the delay of circuits (that is, FTA techniques). However, a systematic classification of existing methods is still very difficult because FTA techniques may differ by various aspects:

1. the circuit delay and gate delay computation models,
2. the set of conditions used to test path sensitizability, also called **sensitization criterion**,
3. and the method used to test whether the sensitization conditions are satisfied or not.

Circuit delay and gate delay computation models have already been addressed in sections 2.3 and 2.5, respectively

Several path sensitization criteria are found in the literature (e.g., [BRA88], [DU89], [MCG89], [PER89], [BEN90], [CHE91], [DEV91] and [DEV93]). The most representative ones, **static sensitization** [BEN90], **static cosensitization** [DEV91], **viability** [MCG89] and **exact floating-mode sensitization** [CHE91], are presented in chapter 4. Chapter 4 begins by investigating the robustness of transition mode and floating mode-based delay estimations.

Sensitization criteria are only a part of FTA algorithms. The other part concerns the delay computation algorithm (or technique) itself. Delay computation algorithms may work on a **per-path basis**, testing the sensitizability of one path at a time, or may work on **sets of paths** simultaneously. For testing sensitizability, they may assign logical values to circuit nodes, as done by traditional ATPG methods (**ATPG-based**), or may use satisfiability methods (**SAT-based**). Hence, there are various possibilities for computing circuit delay. Such possibilities are further detailed in chapter 5, which also describes some of the most relevant methods.

Next chapter reviews the terminology related to FTA.

3 Timing Analysis Related Terminology

As long as timing analysis uses many concepts from test generation and logic synthesis, a revision of the definitions from these areas is concentrated in this chapter. Other timing analysis related terminologies, as well as some definitions from delay testing, are also presented. The chapter is divided into three sections. The first section presents Boolean algebra definitions, used in logic synthesis. Test generation definitions are presented in the second section. The last section is devoted to the definitions that are common to delay testing and timing analysis areas.

3.1 Boolean Algebra

The binary **Boolean algebra**, considered in this thesis, is defined by the set $B=\{0,1\}$ and the operators + and · , called **disjunction** and **conjunction**, respectively, which satisfy the *commutative* and *distributive* laws. The disjunction operator, also called **sum** or **OR** operator, has 0 as identity element. The conjunction operator, also called **product** or **AND** operator, has 1 as identity element.

Any element $a \in B$ has a complement, denoted by $\bar{a}$, such that $a \cdot \bar{a} = 0$ and $a + \bar{a} = 1$. The Boolean algebra differs from an ordinary algebra in that distributivity applies to both operations and because of the presence of a complement. Table 3.1 shows the main properties of the Boolean algebra.

TABLE 3.1 - Main properties of the Boolean algebra.

$a+(b+c)=(a+b)+c$	associativity
$a\cdot(b\cdot c)=(a\cdot b)\cdot c$	associativity
$a+a=a$	idempotence
$a\cdot a=a$	idempotence
$a+(a\cdot b)=a$	absorption
$a\cdot(a+b)=a$	absorption
$\overline{(a+b)}=\bar{a}\cdot\bar{b}$	De Morgan
$\overline{(a\cdot b)}=\bar{a}+\bar{b}$	De Morgan
$\bar{\bar{a}}=a$	involution

The multi-dimensional space spanned by *n* binary-valued Boolean variables is denoted by B^n. It is often referred to as the *n*-dimensional **cube**, because it can be graphically represented as a hypercube. A vertex in B^n is represented by a binary-valued vector of dimension *n*. When binary variables are associated with the dimensions of the Boolean space, a vertex can be identified by the values of the corresponding variables.

A **literal** is an instance of a variable or of its complement. For instance, a and $\bar{a}$ are the literals associated to the variable a. A product of n literals denotes a vertex in the n-dimensional Boolean space and is considered as a zero-dimensional cube. Products of literals are frequently called cubes.

Consider the 3-dimensional Boolean space, represented in figure 3.1, with its variables a, b and c. A vertex in the Boolean space can be expressed by the product of n=3 literals, for example $\bar{a}\bar{b}\bar{c}$ (the symbol · can be omitted), or equivalently by the row vector [100]. An arbitrary subspace can be expressed by a product of literals, for example, ab, or by the row vector [11X], where X means that the third variable can take any value.

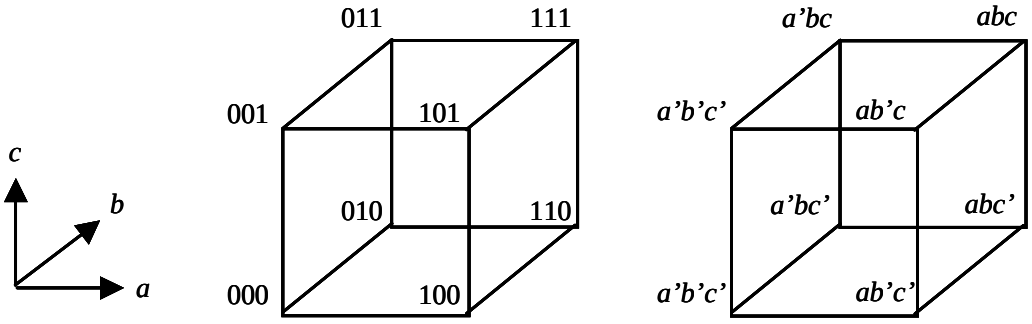


FIGURE 3.1 - Cube representation for the 3-dimensional Boolean space.

Definition 3.1: completely specified **Boolean function**

A *completely specified* **Boolean function** is a mapping between Boolean spaces. An n-input, m-output Boolean function is a mapping $\mathbf{f} : B^n \alpha B^m$, where $B^n=\{0,1\}^n$ and $B^m=\{0,1\}^m$.

An n-input, m-output Boolean function can be seen as an array of m scalar functions over the same domain, and therefore the vector notation is used.

Definition 3.2: incompletely specified **Boolean function**

An n-input, m-output *incompletely specified* **Boolean function** is a mapping $\mathbf{f} : B^n \alpha Y^m$, where $B^n=\{0,1\}^n$ and $Y^m=\{0,1,X\}^m$. The symbol X is used to denote the points where the function is not defined and is called **don't care condition**.

Definition 3.3: **on set, off set** and **dc set**

For each output of an n-input, m-output Boolean function, the subsets of the domain for which the function takes the values 1, 0 and X are called **on set**, **off set** and **dc set**, respectively.

Consider a single output Boolean function f.

Definition 3.4: **support**

Let $f(x_1,x_2,...,x_n)$ be a Boolean function of n variables. The set $\{x_1,x_2,\ldots,x_n\}$ is called **support** of f.

Definition 3.5: **cofactor**

The cofactor of $f(x_1,x_2,..., x_i,\ldots,x_n)$ with respect to variable x_i is $f_{xi}=f(x_1,x_2,...,1,\ldots,x_n)$. The cofactor of $f(x_1,x_2,..., x_i,\ldots,x_n)$ with respect to variable $\overline{x}_i$ is $f_{\overline{x}_i}=f(x_1,x_2,...,0,\ldots,x_n)$.

Definition 3.6: **Boolean expansion** (or **Shannon's expansion)**

Let $f : B^n \alpha B$. Then $f(x_1, x_2,..., x_i,..., x_n) = x_i \cdot f_{x_i} + \overline{x}_i \cdot f_{\overline{x}_i} = (x_i + f_{\overline{x}_i}) \cdot (\overline{x}_i + f_{x_i})$ $\nabla i=1,2,\ldots n$.

Definition 3.7: **minterm**

Any Boolean function $f : B^n \alpha B$ can be represented as a **sum of products** of n literals, called **minterms** of the function, by recursively applying the Shannon's expansion.

After a complete expansion, a Boolean function can be interpreted as a set of its minterms.

Frequently, set operators are used instead of Boolean operators. This is because operations and relations on Boolean functions over the same domain can be viewed as operations and relations on their minterm sets. For example, the sum and the product of two functions are the union (Y) and the intersection (I) of their minterm sets, respectively, while implication between two functions corresponds to the containment ($\subseteq$) of their minterm sets.

Definition 3.8: **unate/binate function**

A function $f(x_1,x_2,..., x_i,\ldots,x_n)$ is (positive/negative) **unate** in variable x_i if $f_{xi} \supseteq f_{\overline{x}_i}$ ($f_{xi} \subseteq f_{\overline{x}_i}$). Otherwise it is **binate** (or mixed) in that variable. A function is (positive/negative) **unate** if it is variable (positive/negative) unate in all support variables. Otherwise it is **binate** (or mixed).

Definition 3.9: **Boolean difference**

The **Boolean difference** of $f(x_1,x_2,..., x_i,\ldots,x_n)$ with respect to variable x_i is defined as $\partial f / \partial x_i = f_{x_i} \oplus f_{\overline{x}_i}$.

The Boolean difference of f with respect to x_i indicates whether f is sensitive to changes in input x_i. When it is zero, then f does not depend on x_i and x_i is said to be **unobservable**. This concept is of great interest in test generation.

Definition 3.10: **consensus**

The **consensus** of $f(x_1,x_2,..., x_i,\ldots,x_n)$ with respect to variable x_i is defined as $C_{x_i}(f) = f_{x_i} \cdot f_{\overline{x}_i}$.

The consensus of f with respect to x_i represents the component that is independent of x_i. The consensus can be extended to sets of variables: $C_{x_i} C_{x_{i+1}}(f) = C_{x_i}(f_{x_{i+1}} \cdot f_{\overline{x}_{i+1}})$

Definition 3.11: **smoothing**

The **smoothing** of $f(x_1,x_2,..., x_i,\ldots,x_n)$ with respect to variable x_i is defined as $S_{x_i}(f) = f_{x_i} + f_{\overline{x}_i}$.

The smoothing of f with respect to x_i corresponds to dropping x_i from further consideration, as all occurrences of x_i were deleted. Obviously, the smoothing may also be extended to sets of variables: $S_{x_i} S_{x_{i+1}} (f) = S_{x_i} (f_{x_{i+1}} + f_{\overline{x}_{i+1}}) = S_{x_{i+1}} (f_{x_i} + f_{\overline{x}_i}) = S_{x_{i+1}} S_{x_i} (f)$

Test generation terminology is presented in the sequel.

3.2 Test Generation Terminology

In this thesis a combinational circuit C is represented as a direct acyclic graph (DAG) C=(V,E), referred to as the **circuit graph**, where V and E are the set of nodes and the set of edges, respectively. In the circuit graph nodes represent circuit gates while edges represent circuit connections (or circuit nets, whenever explicitly indicated). Graph nodes also represent circuit primary inputs and primary outputs. The primary inputs are nodes with no incoming edges, while all the nodes with no outgoing edges are primary outputs[5]. PI(C) and PO(C) represent the set of primary inputs and the set of primary outputs of C, respectively. For such DAG representation the following definitions hold.

Definition 3.12: **path, complete path** and **partial path**

A **path** P in C is an alternating sequence of nodes and edges. In particular, a **complete** or **full path** has the form (v_0, e_0, v_1, e_1, …, v_n, e_n, v_{n+1}), where edge e_i , $1 \le i \le n$, connects the output of node v_i to an input of node v_{i+1}. Any node v_i , with $1 \le i \le n$, is a gate; node v_0 is a **primary input** and node v_{n+1} is a **primary output**. A **partial path** is any path that either does not begin in a primary input or does not end in a primary output. P may also be represented by (v_0, v_1, …, v_n, v_{n+1}) or by (e_0, e_1, …, e_n). However, as long as such notations may cause some confusion, they will be used only in the situations when the complete formalism is not needed.

Let P= (v_0, e_0, v_1, e_1, …, v_n, e_n, v_{n+1}) be a path.

Definition 3.13: **side-inputs** and **on-inputs**

The inputs of v_i other than e_{i-1} are referred to as **side-inputs** of v_i and are represented by S(v_i). The set of all side-inputs along P are referred to as the side-inputs of P, S(P). e_{i-1} is referred to as on-path input or P-input or **on-input** of v_i.

Since this thesis is concerned with combinational circuits represented at the logic level, only structural faults are of interest. Structural fault models assume that circuit interconnections may be affected by **shorts** and **opens**. A **short** is formed by connections between points that were not intended to be connected together, while **opens** concerns broken connections. In most fabrication technologies, including CMOS, a short between power or ground and a circuit line (wire) e_i causes the signal to be at a fixed voltage. The corresponding logical fault consists of signal e_i being **stuck** at a fixed logic value b ($b \in \{0,1\}$), and is denoted by $\boldsymbol{e_i}$ **s-a-**$\boldsymbol{b}$. In many technologies, the effect of an open on a unidirectional signal line with unitary fanout is to make the input that has become unconnected assume a constant logic value and hence appears as a stuck fault. Thus, a single logical fault of the type e_i s-a-b can

[5] These definitions may eventually be changed in case the circuit graph is transformed into a canonical DAG by the addition of source and terminal nodes. For the rest of this text, either this is irrelevant or will be clearly indicated.

represent many different physical faults: e_i open, e_i shorted to power or ground, and any internal fault in the component driving e_i that keeps e_i at the logic value b [ABR90].

The test terminology necessary as a background to understand the timing analysis theory is derived from the **stuck model** based test generation, which is also the most widely used model.

Definition 3.14: **single stuck fault (SSF) model and multiple stuck fault (MSF) model**

> A circuit is said **single stuck faulty** if it is assumed to contain a stuck fault in one of its lines (wires). A circuit is said **multiple stuck faulty** if it is assumed to exhibit two or more stuck faults.

Although the multiple stuck is a more realistic fault model, most of test generation algorithms assume the single stuck fault model. In the following terminology the single stuck fault is underlying model.

Definition 3.15: **test**

> A **test** for a fault $\boldsymbol{e_i}$ **s-a-*b*** is an input vector that allows for distinguishing a faulty circuit from a good circuit.

Generally, most of the points within integrated circuits cannot be accessed. Therefore, it is necessary to provide that by applying the test at the circuit's primary inputs, at least one of the primary outputs of the faulty circuit presents a logic value that is different from that expected for a fault-free (i.e., good) circuit.

Definition 3.16: **automatic test generation (ATG)** or **automatic test pattern generation (ATPG)**

> Automatic test generation or automatic test pattern generation refer to the procedure of finding tests for a given set of faults in a circuit by means of computer programs that automatically (and possibly exhaustively) generate tests by using appropriate algorithms.

Let us now consider the problem of generating a test for the fault e_i s-a-b in a combinational circuit C. Any ATPG procedure able to generate a test for this fault presents two basic steps: **fault activation** and **error propagation**. Both steps however may be seen as direct applications of a generic procedure known as **line justification**, defined as follows.

Definition 3.17: **line justification**

> Consider a combinational circuit C. Let g be a simple gate (AND, OR, NAND, NOR) in C. g has output e_0 and inputs e_1, e_2, …, e_k. **Justifying** the logic value b at e_0 means finding an assignment of logic values that applied at the primary inputs of C results in b at e_0.

Line justification is a recursive procedure in which a gate's output is justified by an appropriate assignment of logic values at its inputs and so on, until the primary inputs of the circuit are reached.

Suppose g is an AND gate. Then there is only one assignment of inputs that justifies a 1 at its output, which is setting all inputs to 1. However, there are 2^k-1 possible assignments that justify a 0 at its output. In this case, the simplest way of justifying a 0 at g's output is (arbitrarily) selecting one of its inputs to set to 0, while the others are set to 1.

Definition 3.18: **fault activation**

An input vector *w* is said to activate fault e_i **s-a-*b*** iff it sets line e_i to the logic value $\bar{b}$.

In other words, to activate the fault e_i s-a-*b* it is necessary to justify e_i with the logic value $\bar{b}$.

Before defining error propagation let us introduce two important concepts: **error signal** and **gate controllability**.

The error signal was introduced by Roth in [ROT66] and allows for confronting signal values in a faulty circuit with those in a fault-free circuit. Consider the instances N and N_f of a given circuit *C*, being N a fault-free instance and N_f a faulty instance. Consider also that both instances are submitted to the same input vector. In this case, if a given node e_i exhibits the logic value 1 in N and logic value 0 in N_f, then the composite value 1/0 indicates a potential source of **error** and is represented by the symbol **D**. Similarly, if node e_i exhibits the logic value 0 in N and 1 in N_f, then the composite value 0/1 also indicates a potential source of **error**, but is represented by the symbol $\bar{D}$. The composite values 0/0 and 1/1 do not indicate any abnormal behavior and hence are represented by 0 and 1, respectively. By using this notation, it is possible to consider both fault-free and faulty circuits at the same time, in test generation algorithms. Besides these four values, the X symbol is required in order to represent the indeterminate value.

The properties of the Boolean algebra hold for the composite values of the set $\{0,1,D,\bar{D},X\}$, since they are operated in a bitwise manner, as for instance, in the expression $D + 0 = 0/1 + 0/0 = 0 + 0/1 + 0 = 0/1 = \bar{D}$. The use of these five values within the Boolean algebra gives rise to the **D-calculus**. Tables 3.2 and 3.3 show the results for the 5-valued AND and OR operations, respectively, used in the D-calculus.

The concept of gate controllability, by its turn, holds only for simple gates and concerns the phenomenon of state absorption. The important definitions are that of controlling/noncontroling values and controlled/noncontrolled values of a gate.

Let *g* be a simple gate (AND, OR, NAND, NOR).

Definition 3.19: **controlling value** and **controlled value**

The **controlling value** of *g* is defined as being the logic value that solely determines the logic value at the output of *g*. The **controlled value** of *g* is the logic value resulted from applying the controlling value to at least one of *g*'s inputs. For instance, 0 is the controlling value for AND and NAND gates. However, the controlled value for AND gates is 0, while the controlled value for NAND gates is 1.

Definition 3.20: **noncontrolling value** and **noncontrolled value**

The **noncontrolling value** of *g* is the logic value, which is not the controlling value of *g*. The **noncontrolled value** of *g* is the logic value resulted from applying the noncontrolling value to all inputs of *g*. For instance, 1 is the noncontrolling value for AND and NAND gates. However, the noncontrolled value for AND gates is 1, while the noncontrolled value for NAND gates is 0.

Note that the correct generalization of the gate controllability concept is the Boolean difference, given in definition 3.9.

TABLE 3.2 - Truth-table for the 5-valued AND operation.

AND	0	1	D	$\bar{D}$	X
0	0	0	0	0	0
1	0	1	D	$\bar{D}$	X
D	0	D	D	0	X
$\bar{D}$	0	$\bar{D}$	0	$\bar{D}$	X
X	0	X	X	X	X

TABLE 3.3 - Truth-table for the 5-valued OR operation.

OR	0	1	D	$\bar{D}$	X
0	0	1	0	$\bar{D}$	X
1	1	1	1	1	1
D	D	1	D	1	X
$\bar{D}$	$\bar{D}$	1	1	$\bar{D}$	X
X	X	1	X	X	X

Now it is possible to define the error propagation procedure.

Definition 3.21: **error propagation**

> Consider the fault e_i s-a-*b* in circuit *C* and assume that P is a (possibly partial) path beginning at e_i and ending at a primary output of *C*. An input vector *w* is said to **propagate the error signal** D or $\bar{D}$ iff it sets all side-inputs of P to their noncontrolling values.

All noncontrolling values applied to P must be justified. If it is not possible to find such a path, then the error cannot be propagated and hence, the fault is **not testable**. Otherwise, the path(s) used to propagate the error is (are) called sensitizable. The test concept of path sensitizability is too restrictive. Therefore, it will be conveniently presented in the next section. Anyway, it is worth to mention that **multiple path sensitization** refers to the case when more than one path is needed to propagate the error signal.

As mentioned earlier, there may exist more than one assignment of input values that justifies a gate output value. On the other hand, in circuits with reconvergent fanout the implication problems are not independent, and sometimes, different logic values may be required to be assigned to the same line. This is a **conflict** (also called **inconsistency** or **contradiction**) that must be solved by changing one of the decisions made by the algorithm.

Definition 3.22: **backtracking**

Backtracking is the process of recovering from inappropriate decisions, thus restoring the state of the computation to the previous conditions.

The backtracking strategy may also allow for a systematic search of the complete space of possible solutions.

In practical terms, a decision consists of selecting a set of value assignments to some gates' inputs (even in the case of selecting a path to propagate the error signal). These values may **imply** in other values (in other circuit lines).

Definition 3.23: **implication**

Implication is the process of determining all values derived from an assignment resulted from a decision made and checking for the consistency of all derived values with respect to the previously existing ones.

Next, delay testing and specific timing analysis terminology is presented.

3.3 Delay Testing and Timing Analysis Terminology

In delay testing and timing analysis areas the circuit component delay information must be considered. Hence, the DAG representation presented in the previous section must be augmented. Firstly, the circuit graph *C* is assumed to be a weighted DAG. Each node v_i of *C* is assumed to have a delay $d(v_i)$ and each connection e_i is assumed to have a delay $d(e_i)$, which may be a fixed value or may vary within an interval.[6]. Nodes representing primary inputs and primary outputs have zero delay.

Let *C* be a combinational circuit.

Definition 3.24: **length of a path** or **delay of a path**

The **length** or **delay** of a path P, d(P), is the sum of the delays of the nodes and the edges along it. In particular, if P is a complete path, then its length is given by:

$$d(P) = \sum_{i=0}^{n+1} d(v_i) + \sum_{i=0}^{n} d(e_i) \qquad (3.1)$$

Definition 3.25: **stable value**

Let *w* be an input vector applied to *C*. The logic value settled at the end of an edge (i.e., circuit connection) *e* under *w* is referred to as the **stable value** of *e* under *w* and denoted by sv(*e*,*w*). Similarly, the logic value settled at the output of gate *v* under *w* is referred to as the **stable value** of *v* under *w* and denoted by sv(*v*,*w*).

Definition 3.26: **stable time**

Let *w* be an input vector applied to *C*. The time by which edge *e* settles to its final value under *w* is referred to as the **stable time** of *e* under *w* and denoted by st(e,w). Similarly, the time by which the output of gate *v* settles to its final value under *w* is referred to as the **stable time** of *v* under *w* and denoted by st(e,w).

The definitions specifically related to the timing analysis track are based on the concept of event propagation, defined as follows.

[6] This is in agreement with the adopted component delay model (section 2.4) and gate delay computation model (section 2.5).

Definition 3.27: **event** and **event propagation**

An **event** is a transition $0 \rightarrow 1$ ($\uparrow$) or $1 \rightarrow 0$ ($\downarrow$) at a gate. Consider a sequence of events $\{r_0, r_1, \ldots, r_n\}$ occurring at gates $\{g_0, g_1, \ldots, g_n\}$ along a path P, such that r_i occurs as a consequence of r_{i-1}. The event r_0 is said to **propagate** along path P.

Definition 3.28: **event sensitizable path**

If there exists an input vector pair such that by assuming *appropriate* delays to the circuit components (gates and connections) an event can propagate along a path P, then P is said to be **event sensitizable**.

Definition 3.29: **single event sensitizable path**

If there exists an input vector pair such that by assuming *arbitrary* delays to the circuit components (gates and connections) an event can propagate along a path P, then P is said to be **single event sensitizable**.

4 Path Sensitization Criteria and Delay Computation Models

This chapter is concerned with delay computation models that are both correct and robust and whose realizations are of practical use from the computation time's point of view.

It begins by analyzing the use of the transition mode with fixed and unbounded gate delay models. It is shown by a simple example that the transition mode results in a robust delay computation only if gate delays are assumed to be unbounded. However, unbounded symbolic simulation has a prohibitive computational cost, preventing its practical use. Then, it is argued that the floating mode is a natural alternative for computing delay because it leads to simpler implementations. Moreover, under the floating mode, the fixed and the unbounded gate delay models become equivalent.

The floating mode makes pessimistic assumptions regarding the states of circuit nodes, however. Due to this it is assured that floating mode-based delay computations provide an upper bound on the circuit delay, since it is assured that the sensitizability of circuit paths is tested by using safe conditions.

The set of conditions used for a given delay computation algorithm to decide whether a path can be responsible for the circuit delay (i.e., whether it is sensitizable) is termed **path sensitization criterion**. Declaring that a path is sensitizable is equivalent to declare that such path may be responsible for the circuit delay.

However, not all sensitization criteria are safe. Some of them may lead to an underestimation on circuit delay. Thus, one of the targets of most publications on timing analysis in last fifteen years has been the search for a sensitization criterion that, used in conjunction with a choice of delay computation models (circuit and gate), leads to the tightest possible circuit delay estimate. Such criterion must also never underestimate circuit delay. Assuming the floating mode, section 4.2 reviews four relevant sensitization criteria.

4.1 Delay Computation Models and the Robustness Property

Let us now consider the monotone speedup property in the context of circuit and gate delay computation models. Assume the **transition delay** is the circuit delay model to be used in the delay computation of the test circuit shown in figure 4.1a (borrow from [MCG89] and re-edited in [DEV94]). The delay of each gate is assigned inside it. Recalling that in transition delay the inputs are made of pairs of vectors, in order to find the delay of the circuit it will be necessary to simulate two input vector situations: (v_0,v_1) and (v_1,v_0), where v_0=(a=0) and v_1=(a=1). These situations correspond to 0→1 and 1→0 input transitions, respectively. As shown by the timing diagrams (figures 4.1b and 4.1c), applying 0→1 and 1→0 transitions at the primary input **a** does not change the output **h** from 0. Thus, the transition delay (also called the delay under the transition mode) of this circuit for the given fixed gate delays is zero.

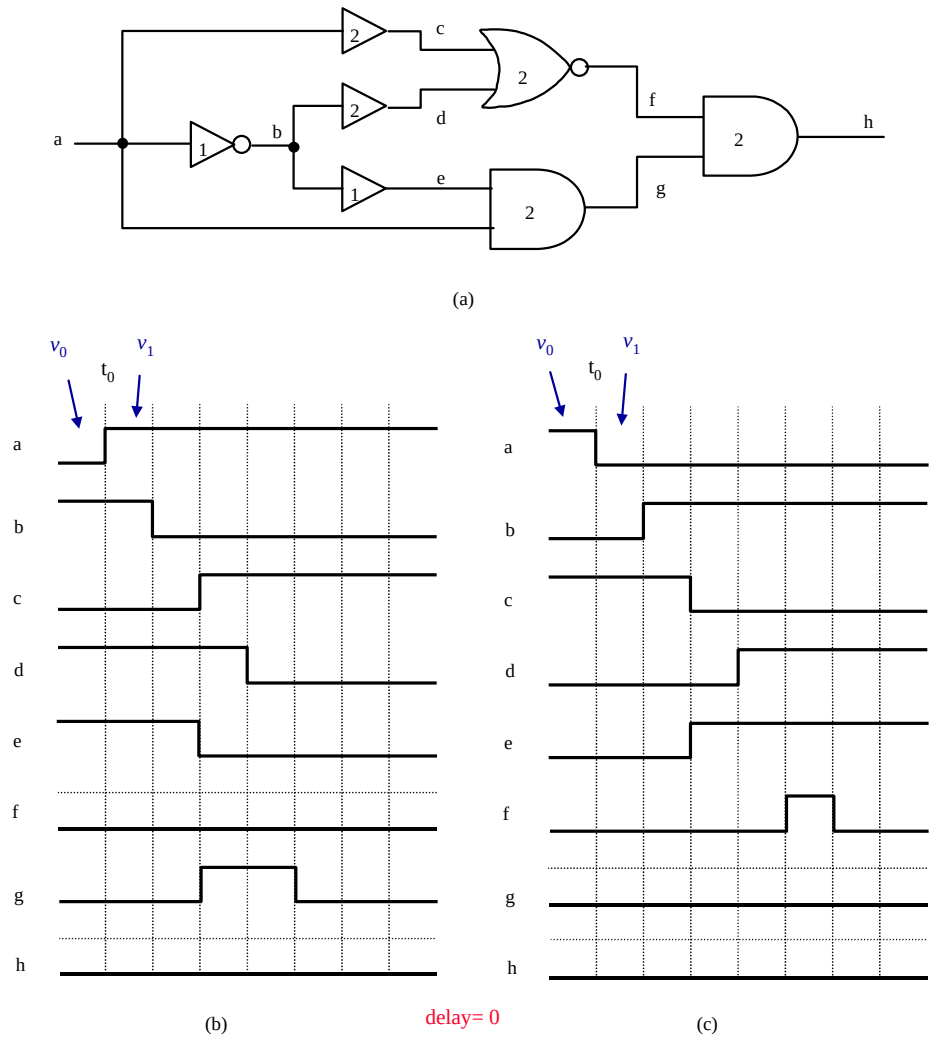


FIGURE 4.1 - Transition delay with fixed gate delays: test circuit (a) and timing diagrams (b),(c).

Now consider another instance of the circuit of figure 4.1a, shown in figure 4.2a, in which all gates have the same delays as the former instance, except one of the buffers at one of the **nor's** inputs, which has delay zero. It might be expected that speeding up one of the circuit's components would not increase the critical delay of the circuit. However, as it is shown in figure 4.2b, applying a 0→1 transition at **a** makes **h** to switch both at time 5 and 6. Although this behavior concerns a hazard, the output only settles to its steady state at time 6. Thus, 6 is the critical delay of the circuit.

The latter result shows that a FTA algorithm that assumes pairs of vectors (i.e., the transition mode) as circuit delay computation model along with fixed gate delays is not robust to the monotone speedup property and thus may underestimate the circuit delay.

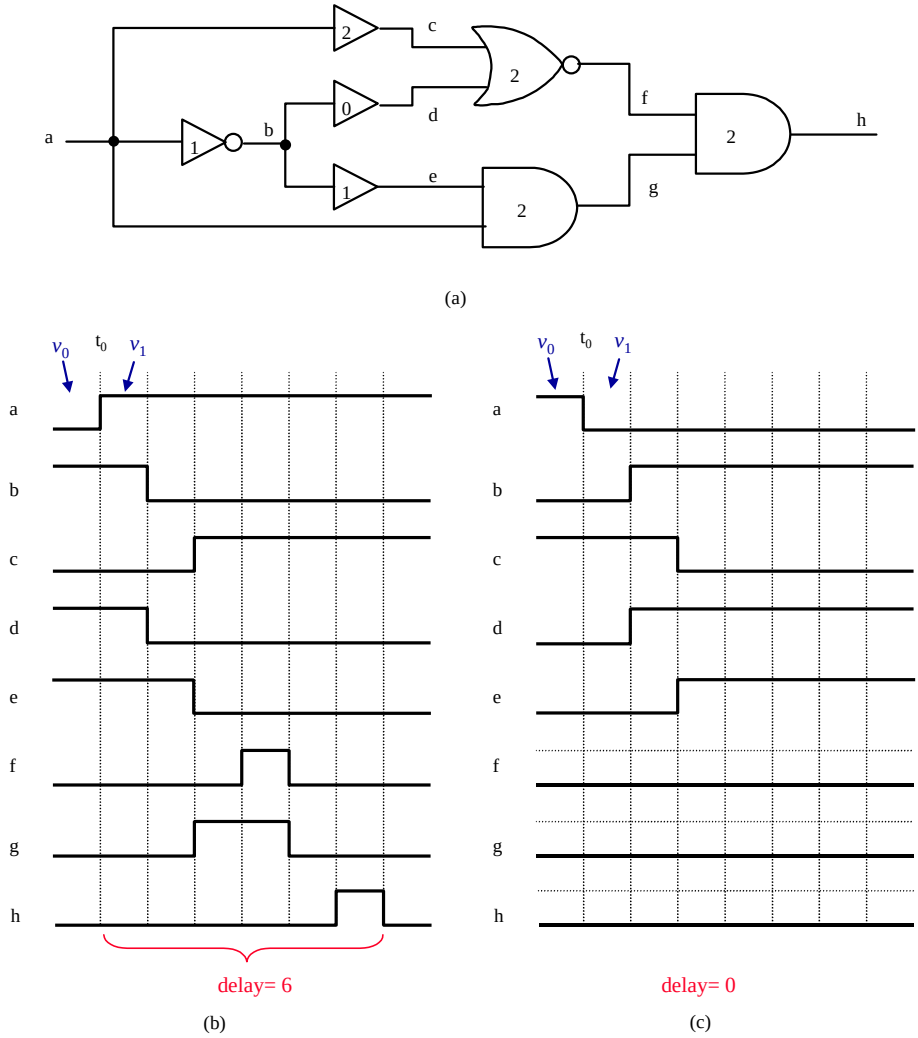


FIGURE 4.2 - Transition delay with fixed gate delays: another instance of the test circuit of figure 4.1a (a) and timing diagrams (b),(c).

It is interesting to remark that in traditional worst-case design methodology, upper bound delay values are assigned to the gates and the delay estimation is supposed to report the (worst-case) critical delay of the circuit. (Hence, in the last example, the fixed gate delays actually represent upper bounds!) This reinforces the need for adopting (monotone-speedup) robust delay computation models in the development of FTA algorithms, as stated in section 2.6. In this sense, the use of fixed gate delays along with the transition mode is a bad assumption for it is unable to accommodate possible inaccuracies of the gate delay physical model. The natural alternative is to consider that gate delays may vary within closed intervals, which is in fact the essence of the bounded and unbounded gate delay models. To investigate the effect of using such kind of gate delay model, consider the circuit of figure 4.3a, which is identical to the circuit of figure 4.2a, except that the unbounded gate delay model is used: the delay of each gate may vary within the range $[0,d^{max}(g)]$, where $d^{max}(g)$ is the upper bound on the delay of gate g. The delay values assigned to the circuit of figure 4.2a will be assumed as d^{max} for the gates of figure 4.3a.

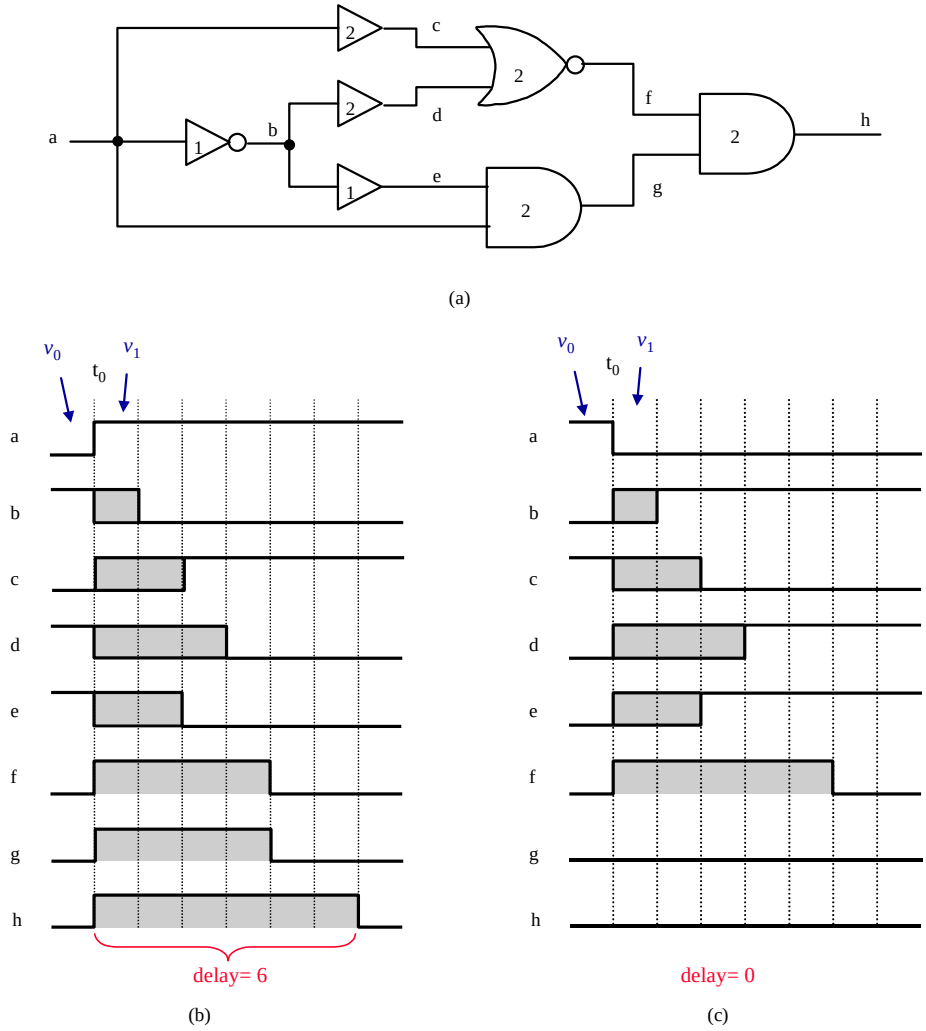


FIGURE 4.3 - Transition delay with unbounded gate delays: test circuit of figure 4.1a with unbounded delays (a) and timing diagrams (b),(c).

Again, to determine the circuit delay it is necessary to simulate the two input vector situations: (v_0,v_1) and (v_1,v_0), with v_0=(a=0) and v_1=(a=1). On the other hand, due to the unbounded gate delay model, the simulation itself is much more complicated, since the transitions at the circuit nodes' may occur at any time between t=0 and t_{max}, where t_{max} depends on both gate delays and gate functionalities. Consider that the input vector pair (v_0,v_1) is applied at the circuit input. The timing diagram for this situation is shown in figure 4.3b. Signal **c**, for instance, goes through a 0→1 transition between t=0 and t=2, while signal **d** makes a 1→0 transition between t=0 and t=3. Signal **f**, by its turn, may pass through a 0→1 transition at any time between t=0 and t<4. If this is the case, a 1→0 transition will take place between t≥2 and t=4. Otherwise, **f** stays at 0. In the waveform of signal **f** (figure 4.3b) the timing interval between t=0 and t=4 is dimmed, representing the uncertainty in the logic value of **f**. Going further in the analysis, the waveform of signal **h** shows a late possible transition may occur at t=6. Thus, the critical delay of this circuit is 6.

By using the unbounded delay simulation method [DEV94a], the critical delay of a circuit is determined as a byproduct, for it is the latest time a transition may occur. Besides this, the analysis itself takes the monotone speedup failure into account. But on the other hand, as any simulation method, the search space is 2^{2n} where n is the number of primary inputs to the circuit. Another problem is the complexity of performing simulation with unbounded delays, which is much more difficult than fixed delay simulation. Finally, it is not

clear whether such a symbolic simulation method can be considered as an input-independent timing verification method.

Due to the previously mentioned difficulties the transition mode is not of practical use for implementing timing verification tools. This also reinforces the use of the FTA technique based on the **floating mode**. As mentioned in section 2.3, this model assumes the nodes of the circuit to be at arbitrary values and determines the circuit delay by a single vector. Comparing to the transition delay, the floating delay is significantly easier to compute for both fixed and unbounded gate delay models. There is no need for storing sets of waveforms at circuit nodes, for instance.

However, the ease of implementation does not come from free. In fact, the assumption of arbitrary values at the circuit nodes is pessimistic. In particular, this assumption makes the fixed and the unbounded gate delay models equivalent, which, on the other hand, should be explored to facilitate the implementation of robust floating delay computation methods.

To understand why the fixed and the unbounded gate delay models are equivalent under the floating mode consider a circuit *C* with fixed delays assigned to its components. Let P be a path of *C* and v_2 be a vector applied to *C*. In order to determine if P is responsible for the delay of *C* on v_2, the side-inputs of P must be checked in the following sense: at each gate *g* of P the side-inputs of *g* must be at noncontrolling values when the transition that is supposed to propagate through P arrives at the on-input of *g*. (This is an informal version of the exact floating sensitization criterion, which is further formalized in section 4.4.) If the value at a side-input *i* to *g* is noncontrolling on v_2, the monotone speedup property (under transition or floating delay model) allows us to disregard the time that the noncontrolling value arrives at the respective side-inputs of P. Assume the delay of all paths from the primary inputs to *i* are greater than the delay of the subpath corresponding to P and ending at *g*. Under the unbounded delay model, the paths to *i* can always be speedup because the delay of each circuit component is assumed to be within the range $[0,d^{max}]$. Under the floating delay model with fixed component delays, it is not possible to change the delays of the paths to *i*, but one can assume that v_1, the vector applied before v_2, was providing the noncontrolling value. Thus, it is not necessary to wait for v_2 to provide the noncontrolling value. In either case, the arrival time of noncontrolling values on side-inputs does not matter.

Such considerations justify the floating mode as being an appropriate circuit delay computation model for the FTA technique, since it is “speedup robust” for both fixed and unbounded gate delay models and leads to less complex algorithmic implementations. In addition, it allows for the implementation of input-independent delay computation algorithms, which corresponds to the philosophy of timing analysis.

Once the adoption of the floating mode computation model is well justified, the next step is to investigate the necessary conditions to determine whether a given path may be responsible for the delay of the circuit, that is, whether it is sensitizable or not under the floating mode. Since 1986, when Brand and Iyngar first proposed a set of conditions to test whether a path is responsible for the delay of a circuit (an extended version of their work is presented in [BRA88]), many other sets of sensitization conditions were proposed and implemented within FTA tools. Each set of conditions is referred to as a **sensitization criterion**. Sensitization criteria are the object of the next section.

4.2 Path Sensitization Criteria

The reason for the existence of several sensitization criteria is the tradeoff between computational complexity of FTA algorithms and the accuracy of the resulted delay estimates. To a first approximation, the more complex are the sensitization tests, the more accurate are the delay estimates, obviously, implying higher computational costs.

Most of sensitization criteria were originally defined using topological parameters. To be more specific, the conditions under which a path is declared to be responsible for the delay of the circuit consider signal values at on-inputs and side-inputs of the path and, in some cases, the time these signals become stable. (Not surprisingly, many timing analysis techniques are based on ATPG algorithms.) The sensitization criteria definitions presented in the following subsections also employ topological parameters.

Sensitization criteria may be classified either as **delay-dependent** or **delay-independent**. In delay-dependent criteria not only the logic value of signals applied to the path side-inputs are considered but also the time such signals become stable. In opposition to delay-dependent, delay-independent criteria do not care about the time the signals become stable.

In the following subsections two delay-independent and two delay-dependent sensitization criteria, defined in the context of the floating mode, are presented: **static sensitization** [BEN90], **static cosensitization** [DEV91], **viability** [MCG89] and **exact floating-mode sensitization** [CHE91]. The correctness and robustness of these criteria are also discussed, using some circuit examples.

4.2.1 Static Sensitization

Static sensitization [BEN90] was one of the first sets of sentitization conditions to be formalized and used in timing analysis. If delay by pairs of vectors were assumed (i.e., the transition mode), it would correspond exactly to the same concept of sensitization used for propagating the error signal in stuck fault test generation.

Let P= (v_0, e_0, v_1, e_1, …, v_n, e_n, v_{n+1}) be a path. The sensitizability of P can be defined as follows.

Definition 4.1: **static sensitization**

Path P is said to be **statically sensitizable** if and only if there is at least one input vector w such that for each v_i, $1 \leq i \leq n$, each side-input of v_i settles to nc(v_i) under w.

Figure 4.4 illustrates this condition for any gate of a path P.

Observing that in the floating mode a path can be statically sensitized to **1** without being statically sensitized to **0** (and vice-versa), Devadas and collaborators presented another definition for static sensitization, making the two cases explicit [DEV93]. These cases are stated in definition 4.2.

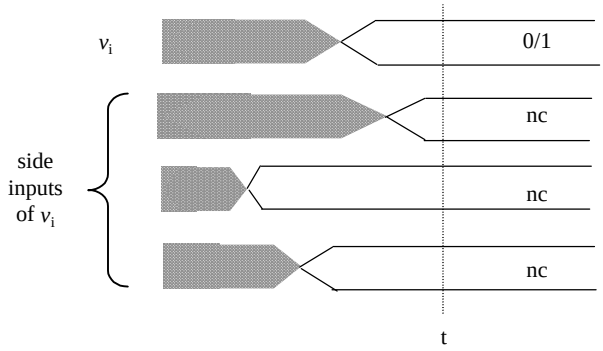


FIGURE 4.4 - Conditions for static sensitization.

Definition 4.2: **static sensitization**

An input vector *w* is said to **statically sensitize** to **1**(**0**) path P in *C* if and only if the value of v_{n+1} is **1**(**0**), and for each v_i, $1 \leq i \leq n$, if v_i has a controlled value, then the edge e_{i-1} is the only input of v_i that presents a controlling value.

Note that from the definition above, if a vector *w* statically sensitizes a path, then it either statically sensitizes the path to **1** or to **0**. Indeed, this is a direct consequence of the floating mode.

Finally, path P is said to be statically sensitizable if there exists at least one input vector *w* satisfying definition 4.2.

In order to analyze the correctness of this condition, consider the circuit of figure 4.5. The path shown in bold has been statically sensitized by the vector 100X (with X representing the undefined or the unknown value). This means that when the vector pair {110X,100X} is applied to the circuit inputs, a transition propagates along this path.

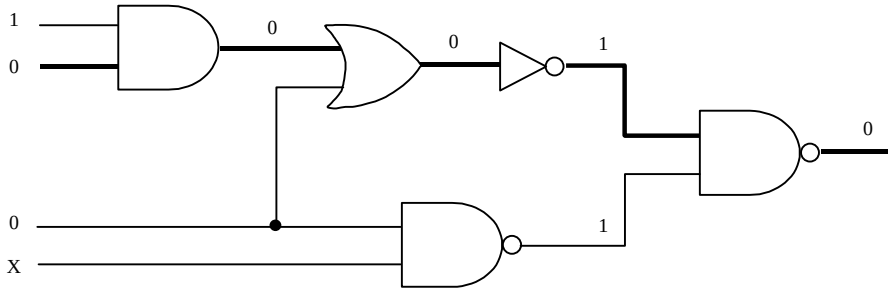


FIGURE 4.5 - Example of static sensitization of a path.

Now consider the csa2 stage of figure 2.4, reproduced in figure 4.6. It is not possible to find an input vector that statically sensitizes the path shown in bold. This is because in order to sensitize this path it is necessary to set all of its side-inputs to noncontrolling values, that is, p0=g0=p1=g1=ctrl_n=1 (with ctrl_n being the control input to the multiplexer). However, ctrl_n=1 implies that either p0 or p1 to be 0 or both, what disagrees from the former assumptions. Thus, this path is not statically sensitizable. (As demonstrated in section 2.2, we also know that it is not possible to find an input vector pair under the given fixed gate delays that propagates a transition along this path.)

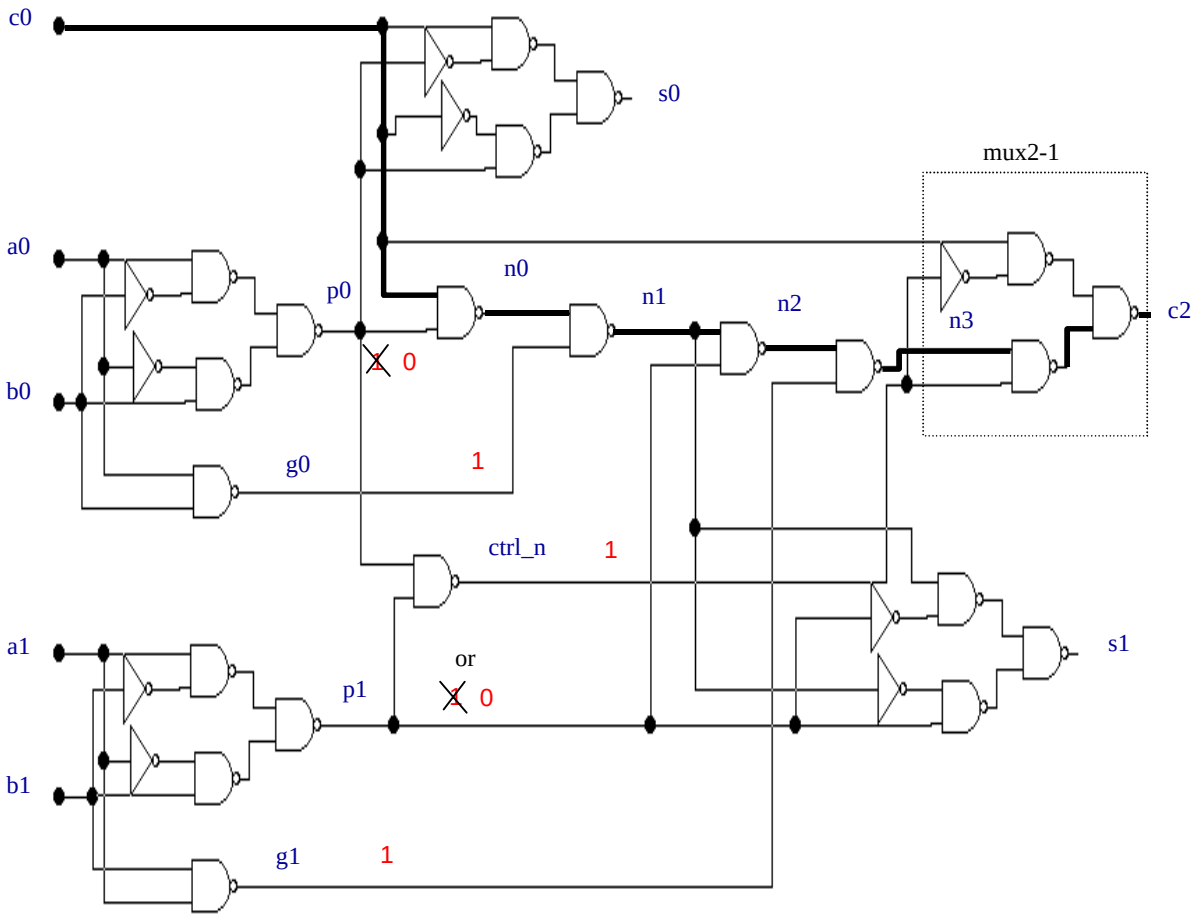


FIGURE 4.6 - Static sensitization on the csa example.

Let us now examine a third example of static sensitization. Consider the circuit of figure 4.7a (borrow from [MCG89]) and assume all its gates have delay equal 1. Paths (**a**,**d**,**f**,**g**) and (**b**,**d**,**f**,**g**) are not statically sensitizable. This is because to statically sensitize (**a**,**d**,**f**,**g**) a 1 is required on **e**, implying that both **a** and **b** have to be at 0. However, this requires that the AND gate with output **d** must have both inputs at controlling values, which disagrees from the static sensitization definition. A similar analysis can be done for the path (**b**,**d**,**f**,**g**). This analysis could induce the conclusion that the critical delay of the circuit is 2. However, keeping **c** at 0 and applying a 1→0 transition to both **a** and **b** inputs makes the circuit output to settle to 0 only at time 3, as shown in figure 4.7b. Thus, a path may not be statically sensitizable but can still be responsible for the delay of the circuit.

Indeed, it can be shown that static sensitization is a sufficient condition for a path to be responsible for the delay of a circuit under the floating mode. To understand this, consider a vector v_2 being applied to a circuit *C* where v_2 statically sensitizes path P to 1. On v_2 all side-inputs of P are at noncontrolling values. Under the floating mode, on any particular gate *g* of P, the values of the side-inputs of *g* on the previously applied vector v_1 can be assumed to be at noncontrolling values. This implies that we have steady noncontrolling values at each side-input. Thus, by the time the event propagating along P arrives at any gate *g* of P, all side-inputs of *g* are stable at noncontrolling values. Obviously, this implies that P is responsible for the delay of *C*.

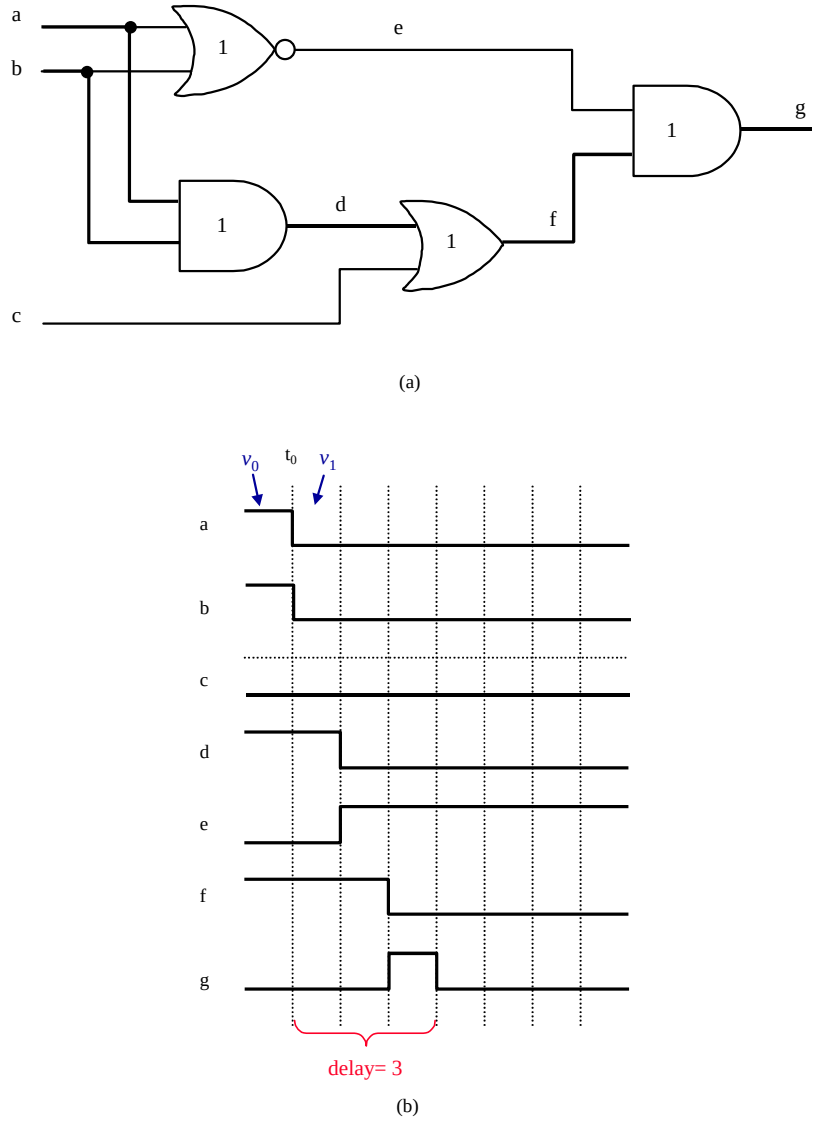


FIGURE 4.7 - Static sensitization underestimating circuit delay.

4.2.2 Static Cosensitization

Static cosensitization is another delay-independent criteria similar to but less restrictive than static sensitization. It can be defined as follows [DEV91]. Let P= (v_0, e_0, v_1, e_1, …, v_n, e_n, v_{n+1}) be a path in a circuit *C*.

Definition 4.3: **static cosensitization**

> An input vector *w* is said to **statically cosensitize** to **1**(**0**) path P in *C* if and only if the value of v_{n+1} is **1**(**0**), and for each v_i, $1 \leq i \leq n$, if v_i has a controlled value, then the edge e_{i-1} presents a controlling value.

As in the case of static sensitization, if a vector *w* statically cosensitizes a path, then it either statically cosensitizes the path to **1** or to **0**. A path P is said to be statically cosensitizable if there exists at least one input vector *w* satisfying the previous definition.

Figure 4.8 illustrates the static cosensitization conditions.

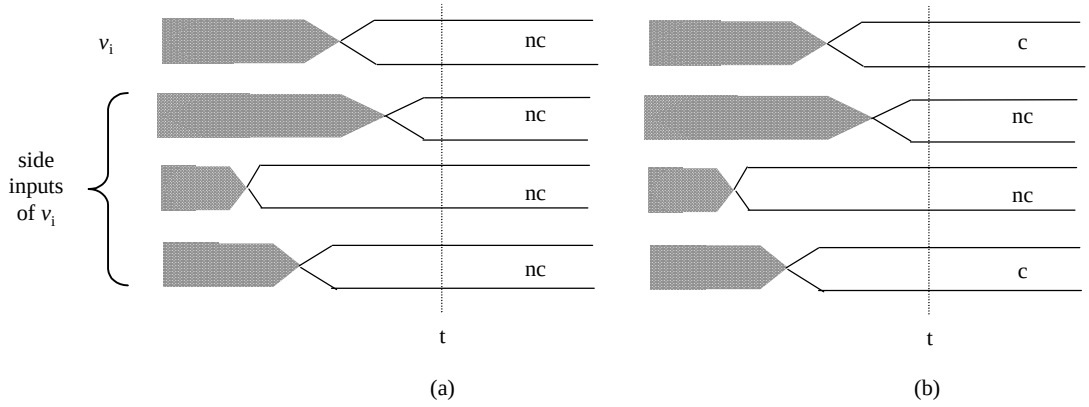


FIGURE 4.8 - Conditions for static cosensitization.

Let us discuss the accuracy of the delay estimate obtained by static cosensitization. Figure 4.9 reproduces the circuit of figure 4.7a. It can be seen that the input vector (0,0,0) statically cosensitizes paths (**a**,**d**,**f**,**g**) and (**b**,**d**,**f**,**g**). Thus, if static cosensitization were the sensitization condition used, the delay reported for this circuit would be 3.

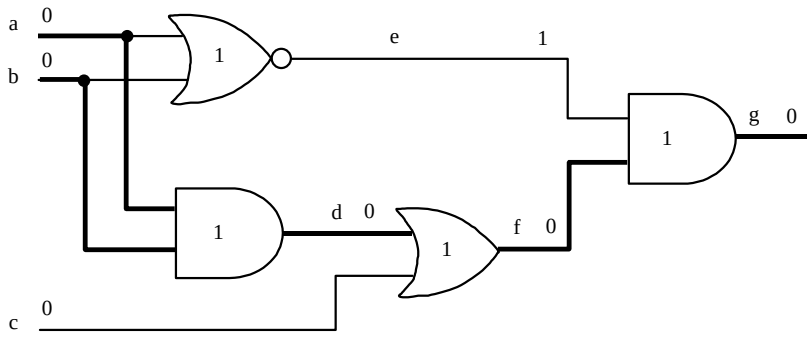


FIGURE 4.9 - Example of static cosensitization of paths.

Static cosensitization can be pessimistic. Consider the trivial circuit of figure 4.10a. The path with length 6 (in bold) is statically cosensitized by input vector a=0. However, by applying the 0→1 and 1→0 transitions at the circuit input, one can notice that the circuit output settle to its steady state at time t=5 and thus 5 is its delay. This is showed by the timing diagrams (figures 4.10b and c).

It can be shown that static cosensitization is a necessary condition for a path to be responsible for the delay of a circuit under the floating mode. To understand this, consider a path P that is not statically cosensitizable. This means that for any applied vector v_2 there is at least one gate g on P at which the on-input has a noncontrolling value and some other side-input of g has a controlling value. The controlling value at this side-input will always control the output of g. If the noncontrolling value arrives before the controlling value, the gate output will go to the controlled value after the controlling value arrives. Alternately, if the controlling value arrives before the noncontrolling value, the gate output will be at the controlled value before the noncontrolling value arrives. In either case, P is not responsible for the delay of the circuit on v_2.

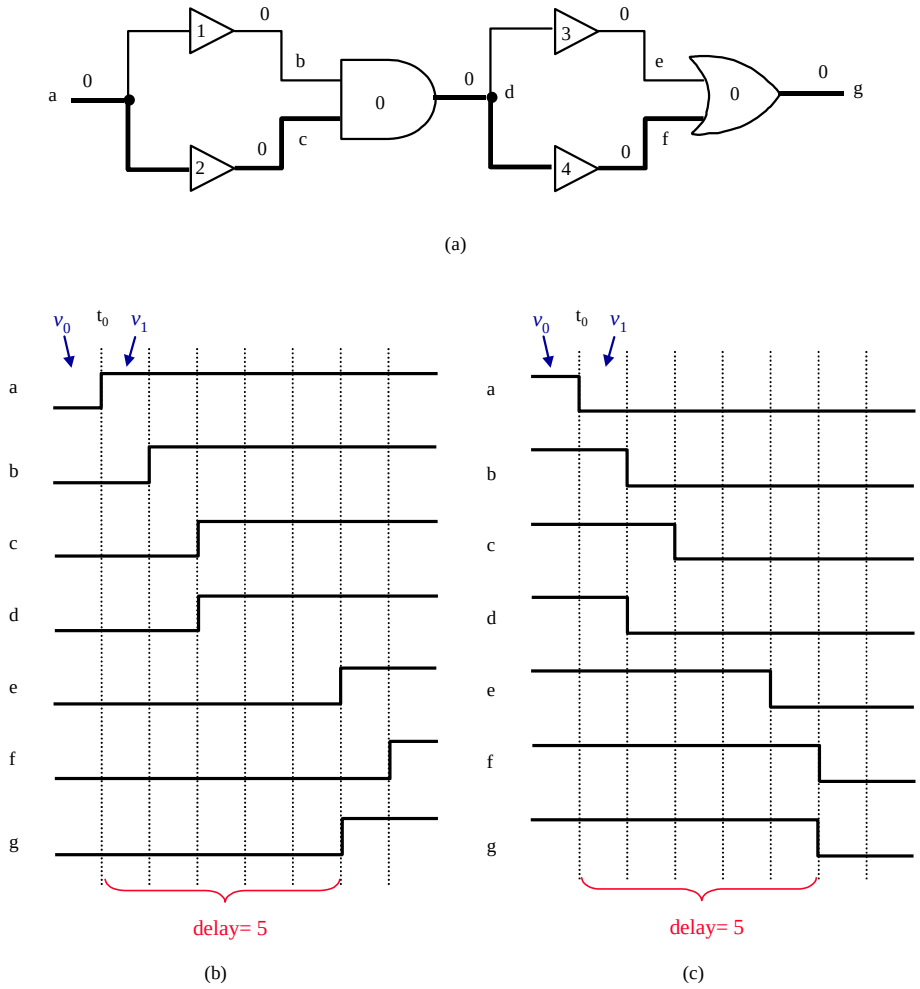


FIGURE 4.10 - Static cosensitization can be pessimistic.

4.2.3 Viability Analysis

The viability analysis, presented by McGeer and Brayton in [MCG89], is in fact a complete FTA technique based on the concept of **viable paths**. A set of delay-dependent conditions is used to test whether a path may be considered as responsible for the delay of the circuit under an input vector *w*. In case yes, such path is declared to be a **viable path**.

Let P= (v_0, e_0, v_1, e_1, …, v_n, e_n, v_{n+1}) be a path in a circuit *C*.

Definition 4.4: **viable paths**

Path P is said to be **viable** if and only if there is at least one input vector *w* such that for each gate v_i, $1 \leq i \leq n$, and for each side-input *e* of v_i, if $st(e,w) < st(e_{i-1},w)$, then $sv(e,w)$ must be equal $nc(v_i)$.

Notice that neither the value of the on-input nor the stable values at the side-inputs presenting $st(e,w) \geq st(e_{i-1},w)$ matter. Figure 4.11 illustrates viability conditions.

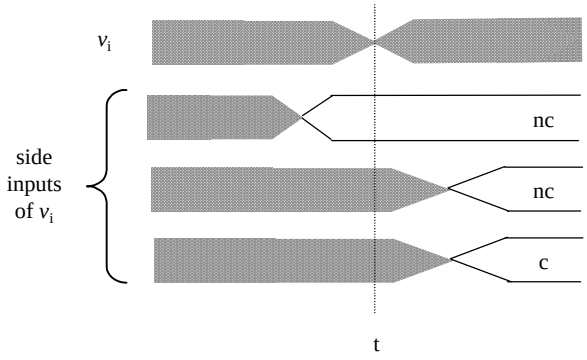
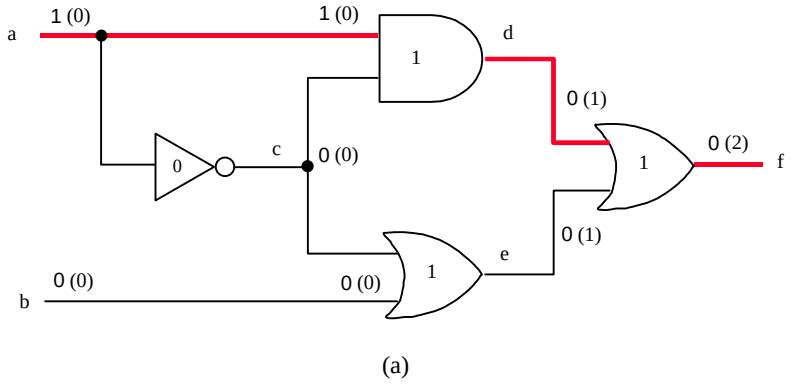


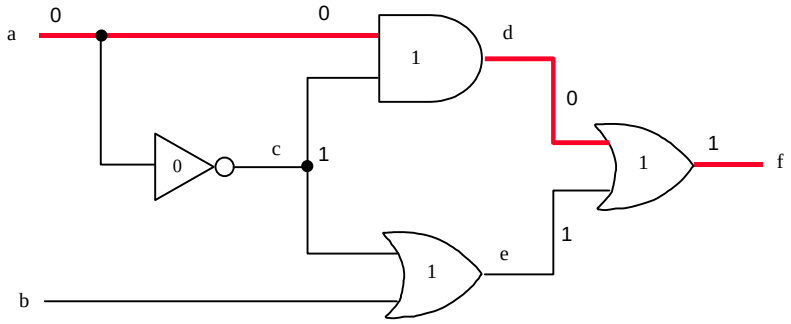
FIGURE 4.11 - Conditions for path viability.

Viability, however, does not imply static cosensitization. A path may be declared viable without being statically cosensitizable, and thus, be false.

Consider the simple circuit of figure 4.12, taken from [CHE93], where all gates have delay equal 1 except the inverter, which has zero delay. Consider also that each connection has zero delay. The path showed in bold is viable for input vector {1,0}. This is because by applying such vector the stable time at the AND gate inputs are equal, allowing us to disregard the stable value at the side-input to this gate. The same happens with respect to the OR gate (figure 4.12a). However, this path is not statically cosensitizable. For **a**=1, **c**=0 and thus **d**=0, which is the AND gate controlled value. But viability requires that **a**, being the on-input to the AND gate, must be at the AND gate noncontrolling value, i.e., **a**=0. Similarly, for **a**=0, **d**=0 and **e**=1. Again, the side-input (with respect to the OR gate) presents a controlling value while the on-input has the noncontrolling value (figure 4.12b). Hence, this path is not statically cosensitizable.



(a)



(b)

FIGURE 4.12 - Example of viable path (a) that is not statically cosensitizable (b).

As a theory, viability is remarkable because it presented the first set of conditions inherently considering the monotone speedup. Indeed, it was demonstrated in [MCG91] that viability is both robust and correct on networks composed of symmetric gates (although it is defined on networks of complex gates, as will be detailed in chapter 5). However, it was not demonstrated that viability is the exact delay estimate. Nevertheless, it is known that the delay value returned by viability is correct.

4.2.4 Exact Floating-Mode Sensitization

The two delay-independent sensitization conditions, static sensitization and static cosensitization, are sufficient and necessary, respectively, for a path to be declared as responsible for the delay of a circuit. However, the necessary and sufficient condition for a path to be responsible for the delay of a circuit under the floating mode is a delay-dependent condition that is stronger than static cosensitization but weaker than static sensitization. It has been introduced in [CHE91] and is currently referred to as the **exact floating-mode sensitization criterion**.

Let P= (v_0, e_0, v_1, e_1, …, v_n, e_n, v_{n+1}) be a path in a circuit *C*.

Definition 4.5: **exact floating-mode sensitization**

Path P is said to be **exactly sensitizable** or **true** (under the floating mode) if and only if there is at least one input vector *w* such that for each gate v_i along P, $1 \leq i \leq n$, one of the following conditions hold:

1. If $sv(e_{i-1}) = c(v_i)$, then for each side-input *e* of v_i , if $sv(e) = c(v_i)$, then $st(e) \geq st(e_{i-1})$
2. If $sv(e_{i-1}) = nc(v_i)$, then for each side-input *e* of v_i, $sv(e) = nc(v_i)$ with $st(e) \leq st(e_{i-1})$

In other words, for a path to be responsible for the delay of a circuit under the floating mode it is necessary that, for each gate *g* along the path:

1. If the output of *g* is at the controlled value, then its on-input must present the controlling value of *g* and furthermore, has to have a stable value no greater than the stable values of the side-inputs having the controlling of *g*.
2. If the output of *g* is at the noncontrolled value, then its on-input has to have a stable time no smaller than the stable times of all the side-inputs of *g*.

These conditions are depicted by figure 4.13.

Let us use these conditions to determine the delay of the previously studied circuits. Consider the circuit of figure 4.1a, reproduced in figure 4.14. Applying a vector **a**=1 sensitizes the path of length 6 shown in bold. In the figure, each connection has both a logical value and a stable time value, the later in parenthesis. This result illustrates that the sensitization condition takes into account the monotone speedup property, unlike transition delay simulation with fixed gate delays.

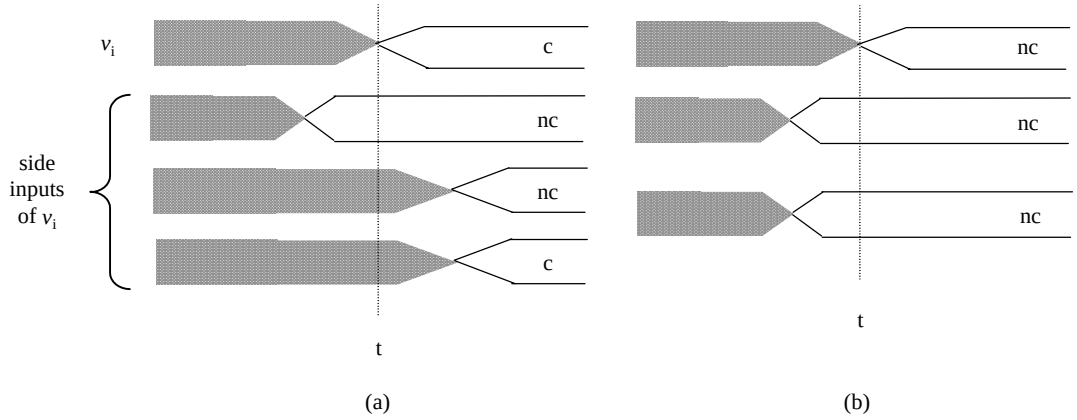


FIGURE 4.13 - Conditions for exact floating-mode sensitization.

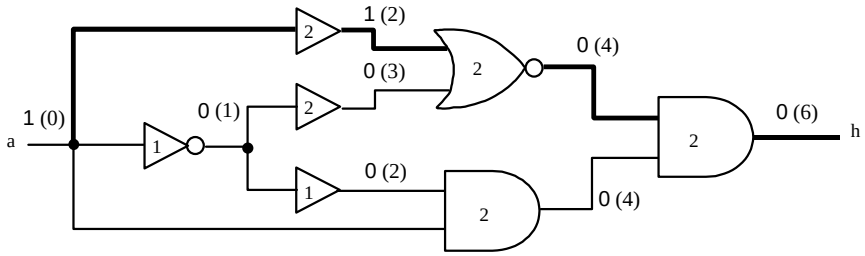


FIGURE 4.14 - First example of exact floating-mode sensitization.

Now consider the circuit of figure 4.5, reproduced in figure 4.15. Applying the vector 000 gives a floating delay of 3, illustrating that the sensitization condition is weaker than static sensitization. The paths (**a**,**d**,**f**,**g**) and (**b**,**d**,**f**,**g**) can be considered as responsible for the delay of the circuit.

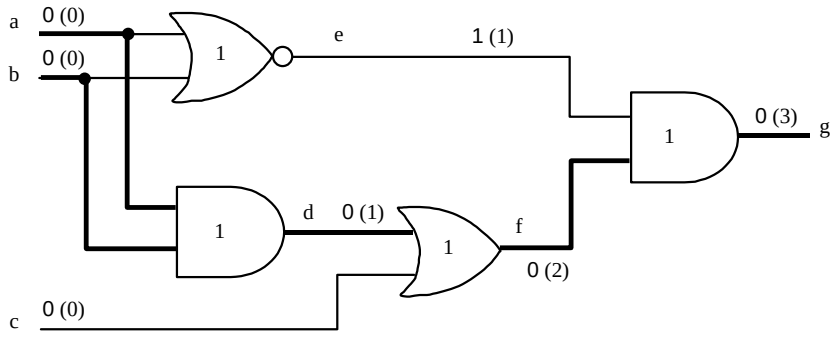


FIGURE 4.15 - Second example of exact floating-mode sensitization.

Finally, consider the circuit of figure 4.7, reproduced in figure 4.16. Applying **a**=0 sensitizes path (**a**,**b**,**d**,**f**,**g**) under the floating mode, while applying **a**=1 sensitizes path (**a**,**c**,**d**,**e**,**g**), also under the floating mode. As long as both paths have delay=5, the floating delay of this circuit is 5. This also shows that the exact floating-mode sensitization condition is stronger than static cosensitization. Recall that static cosensitization reports a delay of 6 for this circuit.

The previous considerations justify the true floating (single-vector) abstraction as a **necessary** and **sufficient** condition for a path to be responsible for the delay of a circuit under the floating model.

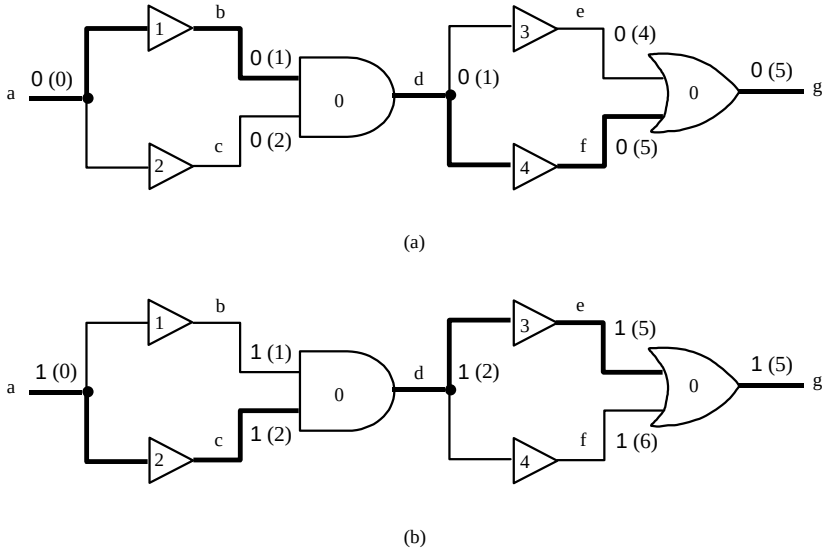


FIGURE 4.16 - Third example of exact floating-mode sensitization.

Note that the conjunction of the single-vector delay model with the sensitization conditions presented in definition 4.5 gives rise to a set of fundamental assumptions that are implicitly assumed by any floating delay computation method. Such assumptions are shown in figure 4.17. Assume the AND gate of figure 4.17a has delay d and is embedded in a larger circuit, and a vector pair (v_1,v_2) is applied to the circuit inputs, resulting in a rising transition occurring at time t_1 on the first input of the AND gate and a rising transition at time t_2 on the second input. The output of the gate rises at a time given by $\max(t_1,t_2)+d$. Under the floating mode, only the values on v_2 are considered. Thus, in this case, a 1 arrives at the first input at t_1 and a 1 arrives at second input at t_2. As a consequence, a 1 appears at the output at time $\max(t_1,t_2)+d$. Similarly, in the case of figure 4.17b, two falling transitions arrive at the AND inputs at time t_1 and t_2, respectively, resulting in a falling transition at the output at time $\min(t_1,t_2)+d$. The abstraction under the floating mode considers a 0 arriving at the first input at time t_1 and a 0 arriving at the second input at time t_2. A 0 appears at the output at time $\min(t_1,t_2)+d$.

Now consider the situation of figure 4.17c, where a rising transition occurs at time t_1 on the first input of the AND gate and a falling transition occurs at time t_2 on the second input. Depending on the arriving order of the transitions occurring at the gate inputs, the output may either stay at 0 (if $t_1 \geq t_2$) or glitch to 1 (if $t_1 < t_2$). If circuit simulation were used, the occurrence of a glitch would be easily detected. However, under the floating model only v_2 is considered, as it is a single-vector approach. The 1 at the first input of the AND gate arrives at t_1, and the 0 at the second input arrives at t_2. The output of the AND gate obviously settles to 0 on v_2. But the problem is determining at what time this occurs: if $t_1 \geq t_2$, then the output is always 0, but if $t_1 < t_2$, then the output becomes 0 only at t_2+d. In order not to underestimate the critical delay of a circuit, any sensitization condition for single-vector computation model **have** to assume that the noncontrolling value (the 1, in the case of AND gate) arrives before the controlling value (the 0, in the case of AND gate), or in other words, $t_1 < t_2$. Under the floating mode this corresponds to assuming that the values on the previous vector v_1 were noncontrolling.

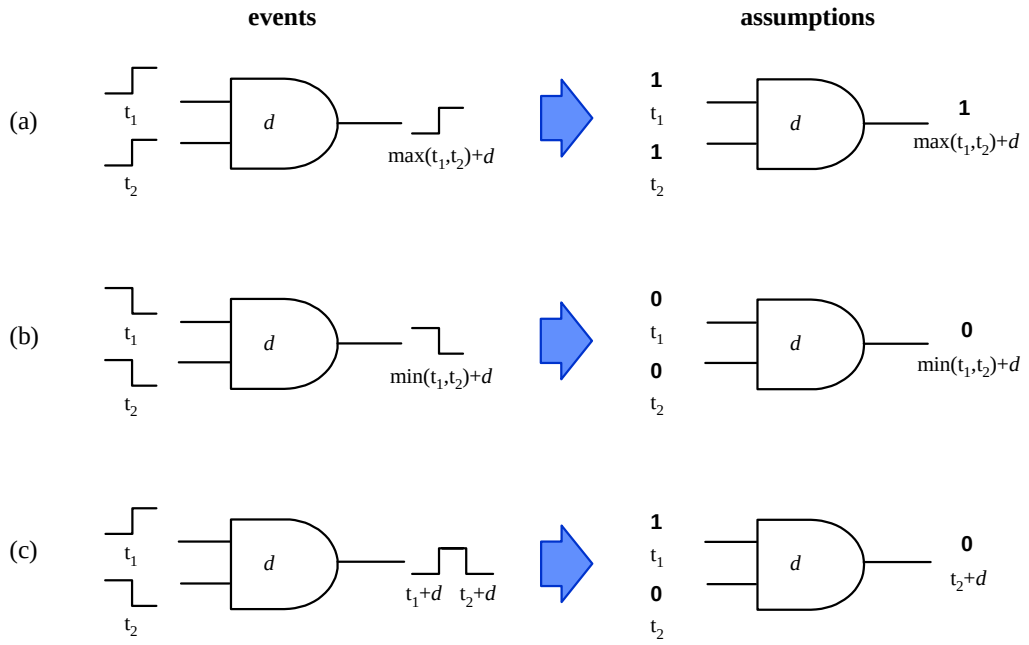


FIGURE 4.17 - Fundamental assumptions made in single-vector exact floating mode.

The rules illustrated in figure 4.17 represent a **timed calculus** for single-vector simulation with delay values that can be used to determine the exact floating mode delay under an applied vector v_2 (assuming pessimistic unknown values for v_1) and the paths that are responsible for the delay under v_2. The rules can be generalized as follows:

1. If the gate output is at a controlled value, pick the minimum among the delays of the controlling values at the gate inputs. (Of course, there has to be at least one input with a controlling value. The noncontrolling values are ignored.) The delay at the gate output is obtained by adding the gate delay to the selected input delay.
2. If the gate output is at a noncontrolled value, pick the maximum of all the delays at the gate inputs. (All the gate inputs have to settle at noncontrolling values.) The delay at the gate output is obtained by adding the gate delay to the selected input delay.

4.2.5 Other Sensitization Criteria

The former subsections presented the most important sensitization criteria. Obviously, a number of other sensitization criteria exist, but all of them are in fact approximations of the exact floating-mode criterion. Examples of other criteria are the **approximate** criterion [CHE93] and the **vigorous** criterion [CHA93].

4.3 Qualitative Comparison Between Sensitization Criteria

It is worth to notice that the concept of path sensitizability is relative. Indeed, the accuracy of delay estimates does not depend solely on the used sensitization criterion. It is important to recall that there is a set of delay computation models underlying any FTA algorithm. Thus, the exact floating-mode sensitization criterion is just an upper bound on the

circuit delay, not only because it assumes the pairs of vector condition, but also because it has to be (monotone speedup) robust.

At this point, it has to be mentioned that the customary terminology may cause some confusion, since the delay computation models underlying path sensitization criteria are rarely made clear. Let us consider the exact floating-mode criterion. Although it has been defined under the floating mode, its original name was just “exact criterion”. In order to avoid any misunderstanding that the term “exact” may cause, it has lately been referred to as exact floating-mode criterion.

From section 2.1 it became clear that the topological delay is always an upper bound on the delay of a circuit. However, it may be too pessimistic because the longest paths may be false.

Although easy to implement, the static criterion is not adequate for furnishing a safe delay estimate because it may declare false a path that is true. Thus, it constitutes a lower bound on the circuit delay. Considering the single vector delay model, the exact criterion and viability are safe delay estimates because they are guaranteed not to underestimate circuit delay. Moreover, such criteria are currently the ones providing the tightest delay estimates.

Finally, the true delay of a circuit could be ideally obtained by using a delay calculation based on delay by sequences of vectors. Unfortunately, no efficient implementation of input-independent technique using this model exists.

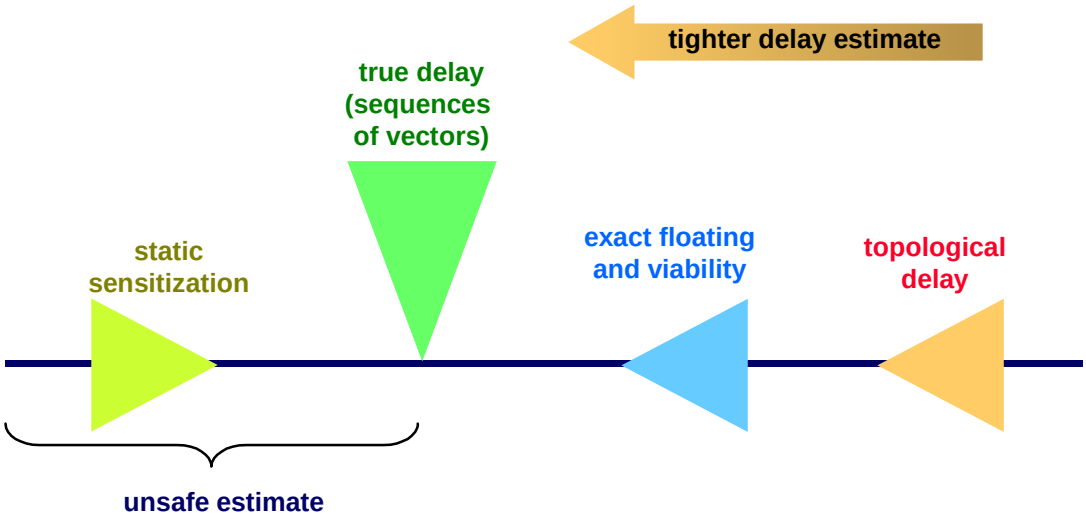


FIGURE 4.18 - Comparison between sensitization criteria.

5 Functional Timing Analysis Algorithms

The most representative sensitization criteria were presented and discussed in the previous chapter. Although the sensitization criterion is a fundamental issue of a timing analysis algorithm, it is not the only one. FTA algorithms may greatly differ by the delay computation procedure, which basically corresponds to the way path sensitizability is taken into account for determining a circuit delay estimate.

This chapter discusses some of the most relevant delay computation algorithms. It begins by proposing a taxonomy that allows for classifying the existing FTA algorithms according to two other aspects, different from the sensitization criterion:

- The number of paths simultaneously handled and
- The method used to determine whether the sensitization conditions are satisfiable or not.

Section 5.1 presents a brief historical review of FTA algorithms. The mentioned algorithms are classified following the proposed taxonomy.

Afterwards, three types of FTA algorithms are discussed in more detail: ATPG-based single path sensitization, ATPG-based concurrent path sensitization and SAT-based concurrent path sensitization. For each type, a typical example of algorithm is detailed.

5.1 Classification of FTA Algorithms and Historical Review

Although many works on path sensitization criteria and FTA algorithms and techniques have been published in the last fifteen years, the absence of a standard taxonomy on these subjects represents a serious difficulty, mainly to those researchers debuting activities in these fields. It is true, however, that some authors have been more cautious with the terminology aspects. This seems to be the case of McGeer-Brayton's works, Devadas' and collaborators' works, and more recently, Ashar-Malik's and Silva-Sakallah's works.

Given the similarities between the functional timing analysis (FTA) problem and the automatic test generation (ATPG) problem, it is obvious that many of existing solutions to the latter are of interest for the former. Hence, while the two classes of problems may share a number of basic algorithms, they also share a common terminology, which basic concepts are stated in chapter 3, and are further employed in the sequel.

Early FTA algorithms operated on a **per-path basis**, by using an adapted version of D-algorithm. This was the case of the works presented in [BEN87] (detailed in [BEN90]), [BRA88], [CHE91] and [CHE93]. In the two latter cases, not only static logic values were asserted, but also the time these signals became stable was tracked. All of these algorithms shared a common feature that was, only one path was considered at a time, concerning the sensitization tests. In practice, this corresponds to the ATPG concept of **single path sensitization**!

But since early, it has been recognized that some circuits may exhibit hundreds of thousands of false paths greater than the critical path [KEU91][DEV94][LAM94], rendering the per-path operation impractical due to its prohibitive computational cost. On the other hand, early ATPG algorithms had already the ability of dealing with multiple paths at a time, or in other words, were able to perform **concurrent** (or **multiple**) **path sensitization**. Not surprisingly, some FTA solutions tried, and actually succeeded, to catch such ability. Among them, the most remarkable are the work of Devadas et al. [DEV91] [DEV93a], that of Silva and Sakallah [SIL94] [SIL94a] and that of Ashar and Malik [ASH95]. Actually, the contribution of the latter work concerns a circuit equivalent form allowing for using unmodified ATPG algorithms to resolve the FTA problem. A **mixed** approach is also encountered in the literature [CHA93]. In this, a set of potential critical paths is identified by concurrent path sensitization. Single path sensitization is then applied to this set to identify the critical path.

Another issue, theoretically independent on the number of paths that an algorithm can handle, is the method used to determine whether the sensitization conditions are satisfiable or not. Recall all previously mentioned algorithms. The method they use to determine the sensitizability of paths (either operating on a single path or on multiple paths) relies on asserting logic values to some of the circuit nodes (on-inputs and side-inputs, in case of single path) and then, by implicating these values to other circuit nodes, watching whether any inconsistency occurs or not. Such procedure has also been borrowed from classical ATPG algorithms, thus being referred to as **ATPG-based**.

In 1989 McGeer and Brayton called the attention to the fact that the set of conditions necessary to declare a path as sensitizable (under a given criterion) could be formally stated as a path sensitization function [MCG89]. Hence, formal methods could be used to resolve the path sensitization problem, instead of the traditional ATPG-based method. Their first solutions to this alternative formulation of the sensitization problem involved dynamic programming [MCG89] and BDDs [MCG91].

Naturally, it would be expected that the same type of formulation to be straightforward applicable to ATPG. In this case, however, the formalized set of conditions tends to present higher complexity since not only the propagation conditions must be taken into account, but also the conditions for fault activation. The work incarnating this shift in terms of ATPG methods is that of Larrabee [LAR92]. Larrabee used Boolean satisfiability to solve a formulation that expressed the Boolean difference between faulty and fault-free circuits.

This shift on methods came again to the FTA field by means of the so-called path recursive functions, defined by McGeer and Brayton in [MCG91a], which provided the formalization necessary to apply Boolean satisfiability algorithms. After that, satisfiability-based (or **SAT-based**) FTA has evolved. Some other SAT-based FTA techniques deserving citation are [MCG93], [SIL93], [SIL96], [SIL98] and [SIL99].

Summarizing the taxonomy that was presented along with the historical review, the existing solutions to the FTA problem may be classified according to the following issues:

- The path sensitization criterion used,
- The number of paths simultaneously handled and
- The method used to test path sensitizability.

Path sensitization criteria were subject of chapter 4. Concerning the number of paths simultaneously handled, a FTA algorithm may use one of the following possibilities:

- Single path sensitization,
- Concurrent (or multiple) path sensitization or
- A mixed approach

Finally, the possible methods to check for path sensitizability are:

- ATPG-based,
- SAT-based or
- Other (e.g., BDDs)

A classification of some existing FTA algorithms, following the proposed taxonomy, is presented table 5.1.

TABLE 5.1 - Classification of existing FTA algorithms.

algorithm	**sensitization criterion**	**number of paths handled**	**method for testing sensitization**
SLOCOP [BEN90]	static	single path	ATPG-based
ACPA [CHE93]	approximate	single path	ATPG-based
LLLAMA [MCG89][MCG91]	viability	single path	BDDs or SAT-based
XBD0 [MCG91a][MCG93]	exact floating	concurrent	SAT-based
VIPER [CHA93]	vigorous	mixed*	ATPG-based
TrueD-F [DEV93a]	exact floating	concurrent	ATPG-based
TA-LEAP [SIL94]	“safe” static	concurrent	ATPG-based
STA [SIL93]	static	concurrent	SAT-based
GRASP [SIL96]	static or viability	concurrent	SAT-based
CGRASP [SIL99]	static or viability	concurrent	SAT-based†

* Partially concurrent path, partially single path sensitization.
† Uses circuit structure information for guiding decision backtracking.

A final note on the proposed taxonomy is required. In the literature, the term ATPG-based (or equivalently, ATPG or TG-based) is generally associated to ATPG-based concurrent path sensitization. This might be because single path sensitization does not use directly ATPG algorithms, but adapted versions of them. Similarly, in the literature the term SAT-based implicitly assumes concurrent path sensitization, since this kind of solution is powerful enough for dealing with multiple paths at a time[7]. However, for didactic reasons, in this text a more complete characterization of FTA algorithms will be used. This way, for the purpose of presenting the most representative classes of algorithms, we may use composed terms, as

[7] In other words, it would not be expectable using SAT methods for testing a path at a time, although it would be possible.

ATPG-based single path sensitization, ATPG-based concurrent path sensitization and SAT-based concurrent path sensitization. These types of algorithms are detailed in the following three sections.

Before beginning the discussions on FTA algorithms, it is worth to mention that any FTA algorithm presents three basic steps that can be presented as:

1. Circuit graph creation
2. Graph pre-processing, to compute maximal delays
3. Circuit delay computation

Thus, different types of algorithms differ from each other by the third step. This step can indeed be further divided in other substeps that may greatly differ, according to the proposed classification. These differences will be focused in the next sections.

5.2 ATPG-Based Single Path Sensitization Algorithms

ATPG-Based single path sensitization algorithms check the sensitization conditions of a given path by asserting logic values to on-inputs and side-inputs and implicating these values to other circuit nodes, in a D-algorithm-like fashion. Indeed, such technique tends to be less complex than the D-algorithm because only the propagation phase is of interest. Moreover, only single path sensitization is tried.

In order to guarantee that the critical path delay is found, the algorithm must begin by examining the topologically longest path. If the sensitization conditions are satisfied, such path is assumed to be the one responsible for the circuit critical delay and its delay is assumed to be the circuit critical delay. Otherwise, the next topologically longest path must be traced and checked. The procedure continues until a sensitizable path is found.

Single path sensitization algorithms (sometimes called per-path algorithms) share a common problem that is tracing paths according to a non-increasing order of their delays. The most efficient algorithms to accomplish this, referred to as path enumeration algorithms, are of complexity $\mathbf{O}(\boldsymbol{n} \log \boldsymbol{n})$, with $\boldsymbol{n}$ equals the number of graph nodes [PIN98]. The procedure may be speeded up slightly if the sensitizability check is accomplished while the path is being traced. In this case, the steps for testing sensitizability would be basically the same as in the D-algorithm: **implication** and **justification** [ROT66]. For each new gate appended to the path being traced, logic values are asserted to the side-inputs. Then, these values are propagated backward to the primary inputs and forward to the primary outputs. In the case of delay-dependent sensitization conditions, the stable time of these values are also taken into account. Non-evaluated nodes remain with don’t care values. Figure 5.1 depicts the execution of such a single path sensitization procedure.

While justifying a given set of propagation conditions at a gate, more than one set of possible logic values may exist for the nodes within the logic cone arriving to the considered gate. These possible sets are stored in a list for future evaluation, in case any inconsistency occurs while evaluating the propagation conditions at the next gates of the path. If there exist at least one assignment of logic values to the circuit nodes satisfying to the sensitization conditions, then the path is declared sensitizable (with respect to the applied sensitization condition and delay models). On the other hand, a path cannot be declared unsensitizable until all possible assignments of logical values have been tested.

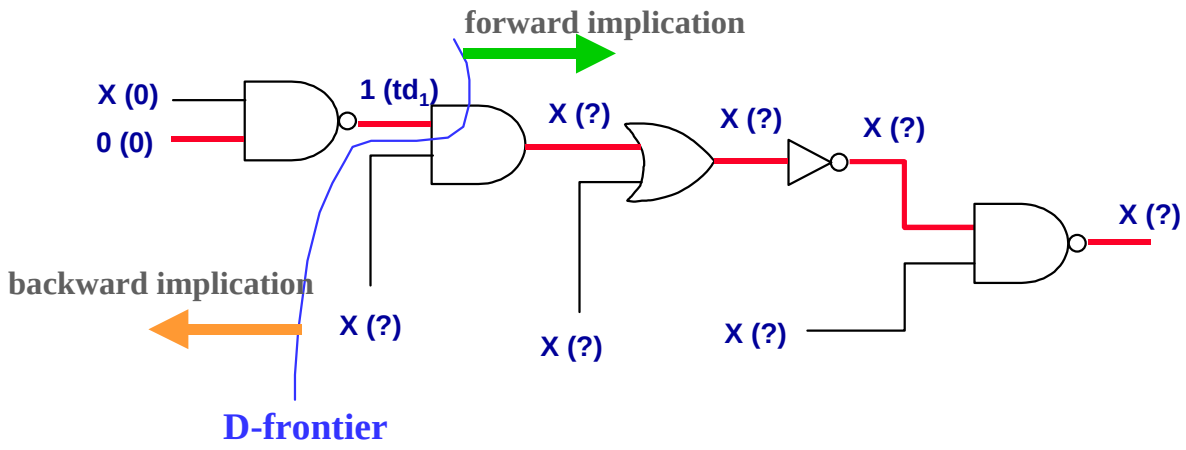


FIGURE 5.1 - Single path sensitization procedure.

In order to speedup execution time, some ATPG-based single path sensitization algorithms (as [DU89] and [CHE93]) do not perform the justification step, since most false paths may be detected in the implication step. However, the implication step is not able to identify all sensitizable paths [PES94] and thus, when justification is not performed, an unsensitizable path may be declared sensitizable, resulting in an overestimation of the circuit delay.

As mentioned earlier, one important issue in single path sensitization algorithms is the path **enumeration** algorithm itself. The problem of tracing paths of a combinational circuit has been studied since the 80's, when various path enumeration algorithms were proposed. Some of them were direct implementations of the classical Breadth-First Search (BFS) and Depth-First Search (DFS) graph traversal procedures (e.g., those presented in [YEN88]). However, as long as these algorithms do not consider pre-processed graph information, tracing paths in an arbitrary order of depth, they were not efficient for enumerating circuit paths in a non-increasing order of lengths (i.e., path delays). Other algorithms applied pruning heuristics based on designer's knowledge of the circuit, as those in [OUS85]. However, the most efficient algorithms are those based on the Best-First Search procedure [WIS84], as the ones presented in [YEN89] and [YEN91] (and generalized in [MCG91]), and thus will be further discussed in the following subsections.

5.2.1 The Best-First Search Path Enumeration Procedure of Yen et al. [YEN89]

The best-first search algorithm, also called **A*** [WIS84], is an algorithm that decides the search trajectory based on pre-computed path lengths. The main reason for using such algorithm for path enumeration of combinational circuits is that it allows for a continuous and organized exploration of the search space with time complexity **O**(***n***log***n***) [MCG91].

The best-first search procedure implemented by Yen et al. [YEN89] assumes a single delay per gate and can be divided into three main phases:

1. circuit graph creation
2. graph pre-processing phase
3. path enumeration phase.

In the first phase, a DAG is created to represent the circuit under analysis. In this DAG, each CMOS gate is represented by a node and each connection, by an edge. Dummy nodes (i.e., with zero delay) are added to represent primary inputs and primary outputs. A source node **s** and a terminal node **t** are also created; a dummy edge is added between node **s** and each primary input node and between each primary output node and node **t**. With this canonical DAG representation, any circuit path assumes the form P = (v_0, v_1, v_2, v_3, ...v_n, v_{n+1}), where v_n is a graph node. If v_0 = **s** and v_{n+1} = **t**, then P is said to be a **complete** path. If $v_0 \neq$ **s** or $v_{n+1} \neq$ **t** , then P is said to be a **partial** path. Figure 5.2 shows the DAG for the circuit of figure 2.1. For the sake of simplicity, consider that the delay of connections is lumped at circuit nodes, i.e., the delay represented by a given net is assigned at its fanin node.

An important aspect that must be considered in the implementation of path enumeration algorithms is the adopted gate delay computation model. That is, the enumeration algorithm must take into account whether a single delay or a pair of delays is assigned to each circuit gate. By pair of delays it is meant separate fall and rise delays and thus, the polarity of signals must be considered in the calculation of path delays.

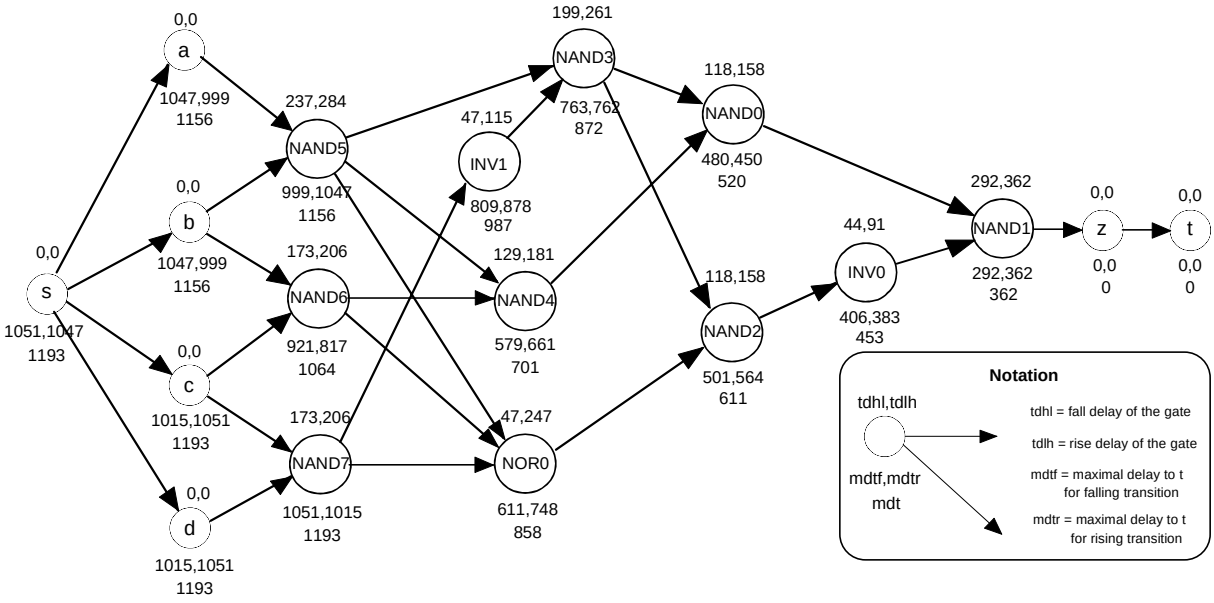


FIGURE 5.2 - DAG for circuit of figure 2.1, pre-processed according to the best-first search procedure.

In the simpler case, when a single delay is assigned to each circuit gate, the delay of a path is computed just by adding the delay of each gate of the path. Signal polarity is disregarded. For a partial path P = (Q, v_{n+1}), with Q = (**s**, v_1, v_2, v_3, ...v_n) the delay is calculated by:

$$d(P) = d(Q) + td(v_{n+1}) \qquad (5.1)$$

where $td(v_{n+1})$ is the delay of gate v_{n+1} appended to partial path Q.

For ordering path tracing, the best-first procedure of Yen et al. [YEN89] goes through a pre-processing phase in which the **maximal delay to node t** (**mdt**) is calculated for each node and stored in the data structure. The **mdt**(v) is defined as the maximum of the delays of all possible partial paths starting at node v and ending at node **t** and is calculated by the formula:

$$mdt(v) = d(v) + \max\{mdt(u_i)\} \qquad (5.2)$$

where $u_i \in$ **adj**(v) (i.e., the list of successors of node v). The procedure for calculating the mdts of all nodes in the graph starts at node **t** and goes backwards in a breadth-first order. A node v will have its mdt computed only after all of its successors' mdts have been computed. The procedure ends when node **s** has its mdt calculated. At the same time the mdt of a node v is calculated, its list of successors, **adj**(v), is sorted in a non-increasing order of each successor's mdt. In the graph of figure 5.2 the value used as gate delay for calculating mdt(v) is max{tdlh(v),tdhl(v)}.

The mdt is a valuable information, which allows to avoid searching branches that will not result in paths having greater delay than the smallest delay among the paths already discovered. It also permits to keep the partial paths candidates ordered according to their esperances. The esperance of a partial path P = (v_0, v_1, v_2, v_3, ...v_i, v_{i+1}), obtained by appending node v_{i+1} to the partial path Q = (v_0, v_1, v_2, v_3, ...v_i), is the maximum delay of all complete paths that have Q as a prefix [BEN87] and is calculated by:

$$e(P) = d(Q) + mdt(v_{n+1}) \qquad (5.3)$$

In the path enumeration phase the k most critical paths are found and stored in a linear structure named k-list, in a non-increasing order of their delays. Each position of k-list stores a given path P (initially, a partial path, but at the end of the procedure, a complete path), its delay, d(P), and its esperance, e(P). At the beginning of the procedure, the first $\mathbf{d}_s$ partial paths made up of node **s** and each of its successors will be stored in the first $\mathbf{d}_s$ positions of k-list, with $\mathbf{d}_s$ being the outdegree of node **s**. If $\mathbf{d}_s$ > k, then only the first k partial paths will be kept (k is a user provided value). Once k-list is initialized with the first partial paths, a function called **extend_and_replace** is invoked for each partial path of k-list, beginning from the first position, k-list[0]. This function takes the partial path stored in k-list[i] and extends it until node **t** has been reached. Suppose that v is the last node of partial path P = (v_0, v_1, v_2, v_3, ...v), stored at k-list[i], which is to be extended, and **adj**(v) = { u_0, u_1, u_2, u_3, ...} is its ordered list of successors. Extending P through u_0 originates a new partial path P_0 , with the same esperance of P (because u_0 is the first successor of v). Thus, P_0 will replace P in position k-list[i]. However, extending P through node u_1 creates a new partial path P_1 with esperance $e(P_1) = d(P) + mdt(u_1)$. Similarly, extending P through node u_2 creates a new partial path P_2 with esperance $e(P_2) = d(P) + mdt(u_2)$ and so on. Before extending path P by appending node u_0, each "side" partial path except P_0 has its esperance calculated in order to know whether this new partial path will be inserted in k-list or not. If the new partial path's esperance is greater than the esperance of the partial path stored in k-list[k-1] (i.e., the last position of k-list), then it is inserted in an ordered manner. Otherwise, neither this partial path nor the next side partial paths will be inserted because partial paths are examined in the same order of the elements in **adj**(v). This results in a prune of the search space. In case a new side partial path is to be inserted, the insertion position j will be within the range (i < j ≤ k-1) and the path stored in position k-list[k-1] will be automatically discarded. Each new partial path P_i that is inserted in k-list will have its delay calculated by:

$$d(P_i) = d(P) + d(u_i) \qquad (5.4)$$

Once path P, stored in k-list[i], is completely extended, d(P) = e(P) because it is a complete path. Then, function **extend_and_replace** will be called for the partial path stored in k-list[i+1]. The path enumeration phase will end when the partial path stored in k-list[k-1] is completely extended. At this point, k-list holds the k-most critical paths of the circuit. Figure 5.3 shows the contents of k-list after its initialization, for the circuit represented by the graph of figure 5.2.

It is interesting to note that while the path stored in k-list[i] is being extended it is not possible to precise which will be the next partial path to be extended. On the other hand, once path of k-list[i] has been completely extended, the next path to be extended is the one stored in k-list[i+1]. This systematic search of the space of possibilities is responsible for the greatest advantages of the best-first procedure over the depth-first procedure [YEN89][MCG91]: paths are traced in a non-increasing order of their delays with no need for costly backtracks.

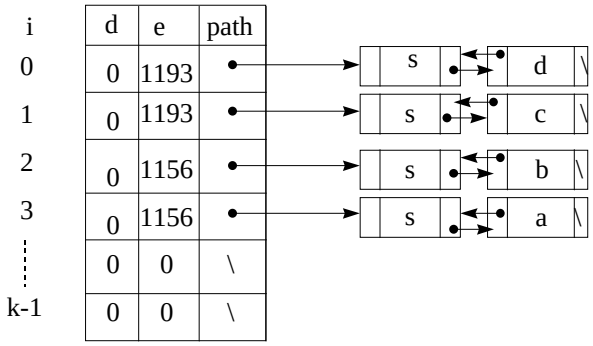


FIGURE 5.3 - k-list structure initialized with the first partial paths of the circuit of figure 5.2.

5.2.2 Best-First Search Path Enumeration Considering Different Fall and Rise Gate Delays

This section describes the modifications to the basic best-first search of Yen et al. needed for dealing with separate fall and rise gate delays. Although of maximal interest for implementing useful FTA algorithms, such modified best-first procedure is described in detail only in [GÜN98], [GÜN98b] and [GÜN99]. The basic idea is to store two lists of successors at each graph node: one for the falling transition and another for the rising transition at the node's output. This also implies in calculating two mdts, as will be explained in the sequel. Many of the ideas described here may also be applied in the context of ATPG-based concurrent path sensitization algorithms.

In order to consider different fall and rise gate delays, each topological path is decomposed into two distinct **logical paths** by fixing the type of the transition taking place at its primary input. Thus, the delay of a partial logical path P = (Q, v_{n+1}), with Q = (**s**, v_1, v_2, v_3, ...v_n) is calculated by:

$$d(P) = d(Q) + tdhl(v_{n+1}) \qquad (5.5)$$

if the input of v_{n+1} is undergoing a rising transition and

$$d(P) = d(Q) + tdlh(v_{n+1}) \qquad (5.6)$$

if the input of v_{n+1} is undergoing a falling transition. By considering signal polarity, these calculations are implicitly adding some primitive information on the relationship between the gates of a path.

In the pre-processing phase, the procedure for calculating the mdts of all nodes is essentially the same, except that for each node two values of mdts must be calculated: **mdtf** and **mdtr**. For a given node *v*, **mdtf**(*v*) and **mdtr**(*v*) give the maximum of the delays of all possible partial logical paths starting at node *v* and ending at node **t**, for a falling transition and for a rising transition at the output of *v*, respectively, and are calculated by:

$$\text{mdtf}(v) = \text{tdhl}(v) + \max\{\text{mdtr}(u_i)\} \qquad (5.7)$$

$$\text{mdtr}(v) = \text{tdlh}(v) + \max\{\text{mdtf}(w_i)\} \qquad (5.8)$$

with $u_i \in$ **adjf**(v) (the list of successors of v for a falling transition) and $w_i \in$ **adjr**(v) (the list of successors of v for a rising transition).

Hence, each node v has two lists of successors, **adjf**(v) and **adjr**(v), sorted in a non-increasing order of each successor's mdtf and mdtr, respectively.

Similarly, the esperance of a partial logical path P = (Q, v_{n+1}), obtained by appending node v_{n+1} to the partial logical path Q = (**s**, v_1, v_2, v_3, ...v_n), is calculated by:

$$e(P) = d(Q) + \text{mdtf}(v_{n+1}) \qquad (5.9)$$

if the input of v_{n+1} is undergoing a rising transition and

$$e(P) = d(Q) + \text{mdtr}(v_{n+1}) \qquad (5.10)$$

if the input of v_{n+1} is undergoing a falling transition. Figure 5.2 also shows the pre-processed mdtf and mdtr values for the DAG that represents the circuit of figure 2.1.

For determining the transition type at the end of a (partial) logical path two fields have been added to each position of structure k-list: **type**, which tells the transition type applied to the path input, and **level**, which indicates the number of nodes that compose the current stored (partial) logical path.

The enumeration phase begins by storing the first 2$\mathbf{d}_s$ partial logical paths made up from node **s** and its successors from **adjf**(s) or from **adjr**(s), with $\mathbf{d}_s$ being the outdegree of node **s**. The nodes taken from **adjf**(s) and **adjr**(s) are chosen following a unique order. If 2$\mathbf{d}_s$ > k, then only the first k partial paths will be kept. A modified version of **extend_and_replace** function is then called for each partial path, beginning from the path stored in k-list[0]. This new version is essentially the same as the original one, except that it deals with logical paths rather than topological paths. When partial logical path P = (v_0, v_1, v_2, v_3, ...v), stored in position k-list[i] is to be extended, the information of k-list[i].type and k-list[i].level is used to choose between **adjf**(v) and **adjr**(v) and to calculate the esperances and the path delays of each extended path. Figure 5.4 shows the contents of k-list when the modified procedure is extending the most critical path of circuit example.

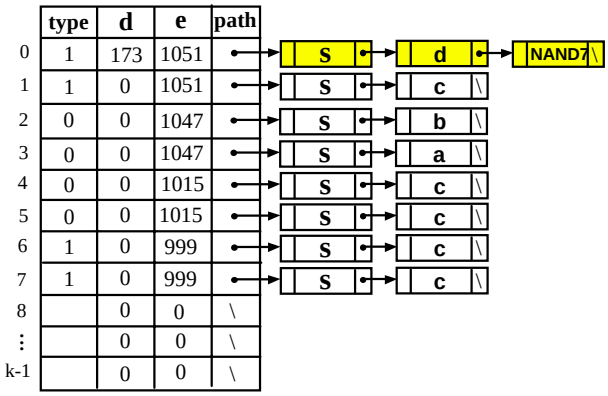


FIGURE 5.4 - k-list structure for the best-first procedure that considers separate fall and rise delays.

It is important to remark that the proposed solution for dealing with separate fall and rise delays does not imply duplication of graph nodes, as does the solution presented in [LI89]. Instead, different fall and rise delays are considered by keeping track of the transition type along paths in order to choose a node's next successor from either adjf or adjr.

Appendix 2 presents a study on the performance of best-first search-based path enumeration procedures gathered from the following publications: [GÜN98a], [GÜN98b], [PIN98], [GÜN99] and [GÜN99a]. This study considers the gate delay model (single gate delay or separate fall and rise gate delays) and also the type of data structure used for storing partial paths (linear dynamic list or binary tree).

5.3 ATPG-Based Concurrent Path Sensitization Algorithms

Searching for a vector v_2 that satisfies sensitization conditions for a particular path can take $\mathbf{0}(2^n)$ time, with n being the number of primary inputs of the circuit. If this search is to be performed on one path at a time, as single path sensitization algorithms do, then the execution time is multiplied by the number of unsensitizable long paths that will be analyzed before the first sensitizable one is found. Furthermore, the time complexity of the fastest path enumeration algorithms is $\mathbf{O}(\boldsymbol{n} \log \boldsymbol{n})$, where $\boldsymbol{n}$ is the number of nodes in the circuit graph [PIN98]. All these problems prevent single path sensitization to be used in FTA of large circuits.

An alternate strategy is to directly answer the question of what the true critical delay of the circuit is and operate on sets of paths rather than a single path at a time [DEV93a]. The question stated is: Is the delay of the circuit greater than or equal to δ? The value δ can initially be set as the delay of the topologically longest path of the circuit and then be progressively decreased while the answer to the question is "yes".

A straightforward $\mathbf{0}(2^n)$ algorithm to find the true critical delay of a circuit that does not require explicit path enumeration is to simulate each of the 2^n input vectors or minterms using the timed calculus and determine the longest delay seen at the circuit output. This process can be speed up by using cube simulation instead of using minterm simulation. This can be accomplished by using a modified version of the PODEM algorithm [GOE81], as proposed in [DEV93a].

In the next subsection the basic PODEM algorithm is described. Subsections 5.3.2, 5.3.3 and 5.3.4 describe the modifications to the PODEM algorithm proposed in [DEV93a] to perform a **timed cube calculus**.

5.3.1 The PODEM Algorithm

PODEM [GOE81] is an enumeration algorithm that implicitly and exhaustively enumerates the input space using cubes rather than minterms, while trying to satisfy a given objective. The following explanation is oriented to a timing analysis implementation and was taken from [DEV94], with modifications.

The topmost call of the procedure **`PODEM`** is described by the pseudocode in figure 5.5: it simply inserts the given primary output line `po` on a justification list `jlist` with value `lvalue`, and calls a search procedure **`SEARCH_1`**. `lvalue` is the logical value (either 0 or 1) to

be justified. Let us assume that a given logical value **out_value** (either 0 or 1) is to be justified at output **po1** of a circuit. Thus, for the first call of the **PODEM** procedure, po←**po1** and lvalue←**out_value**. This can be seen as detecting an (**out_value**) stuck-at fault at output **po1** of the circuit.

```
PODEM(po,lvalue){

  jlist = po with logical value lvalue;


  status = SEARCH_1(jlist);
  return(status);

}
```

FIGURE 5.5 - Pseudocode for the topmost call of the PODEM algorithm.

The search procedures are described by the pseudocodes of figures 5.6 and 5.7. The procedure **SEARCH_1** calls a **BACKTRACE** procedure to find a primary input whose logical value is currently unknown, beginning from the primary output po. The primary input is (initially) set to logical value 1 and the **IMPLY** procedure is called. This procedure in PODEM corresponds to standard three-valued cube simulation (i.e., without any delay information). The implication procedure may produce a logical conflict. If no conflict occurs, **SEARCH_1** is called recursively. The procedure terminates successfully in **SEARCH_1** if the justification list is empty. In the case a logical conflict occurs in **SEARCH_1** the algorithm backtracks to the most recent primary input assignment. The primary input is set to the logical value 0, and the **SEARCH_2** procedure shown in figure 5.7 is called. Failure results if either the backtrace procedure is unable to find a primary input to set or if the space has been completely enumerated without success in **SEARCH_2**. In the case of conflict or failure, the network must be restored to the state it was immediately prior to the primary input setting that caused the failure.

```
SEARCH_1(jlist)
{
   if(length of jlist is zero) return SUCCEED;

   if(BACKTRACE(po,po_value,&pi,&pi_value) == FALSE)
      return(FAILED);

   if(IMPLY(pi,pi_value,jlist) != IMPLY_CONFLICT){
      search_status = SEARCH_1(jlist);
      if(search_status) == FAILED){
         restore the state of the network to what it was
         prior to the most recent primary input assignment;
         search_status = SEARCH_2(jlist,pi,1-pi_value);
      }
   }else{
      restore the state of the network;
      search_status = SEARCH_2(jlist,pi,1-pi_value);
   }
   return(search_status);
 }
```

FIGURE 5.6 - Pseudocode for the first search procedure.

```
SEARCH_2(jlist,pi,pi_value)
 {
    backtracks = backtracks + 1 ;
    if(backtracks > BACKTRACK_LIMIT) return(ABORTED);
       if(IMPLY(pi,pi_value,jlist) != IMPLY_CONFLICT){
       search_status = SEARCH_1(jlist);
       if(search_status == FAILED)
           restore the state of the network;
    }else{
        search_status = FAILED;
        restore the state of the network;
    }
    return(search_status);
 }
```

FIGURE 5.7 - Pseudocode for the second search procedure.

In order to illustrate the PODEM algorithm, consider the following situation: we need to find an input vector that sets the output of the circuit of figure 5.8 to 1. As a first step, the PODEM algorithm sets all primary inputs to 2 (**a**=2, **b**=2 and **c**=2), which will represent the unknown value. The simulation or implication of these input values results in a 2 on all wires including **g**, the output. Backtracing sets input **a** to 1. Implication immediately sets the **g** to 0 as shown in figure 5.8a. The value of 0 at the output corresponds to a conflict since it was asked for justifying a 1 at the output. Thus, it is necessary to backtrack to the most recent primary input setting and change it. In this case **a** was set to 0. The output remains at 2 for the vector **a**=0, **b**=2 and **c**=2. Then, it is necessary to backtrace to another primary input **b** and set it to 1. Implication sets the output to 0 as illustrated in figure 5.8b, and we again have a conflict. The conflict results in backtracking to the most recently set primary input, in this case **b**. The input value is changed to 0. The vector **a**=0, **b**=0, **c**=2 results in an output of 2 as shown in figure 5.8c. Setting the remaining unknown input **c** to 1 sets the output to the required value of 1, as shown in figure 5.8d. The search procedure ends in success.

The sequence of decisions taken in the previous example may be represented in terms of a decision tree as the one shown in figure 5.9. The nodes of the tree correspond to primary inputs, and the left and right edges from each node correspond to input settings of 1 and 0, respectively. The leaves of the tree correspond to primary output settings.

5.3.2 Cube Simulation

In order to use the PODEM algorithm for delay computation, we are interested in arriving at a timed calculus over three logical values, 0, 1 and 2 (unknown), that has the following properties [DEV94]:

- It is equivalent to the timed calculus for computing the floating delay presented in subsection 4.2.4, in the case where inputs are completely specified;
- Given an incompletely specified vector, it produces an upper bound on the achievable delay over any of the minterms in the vector;
- Given an incompletely specified vector, it produces a lower bound on the achievable delay over any of the minterms in the vector;

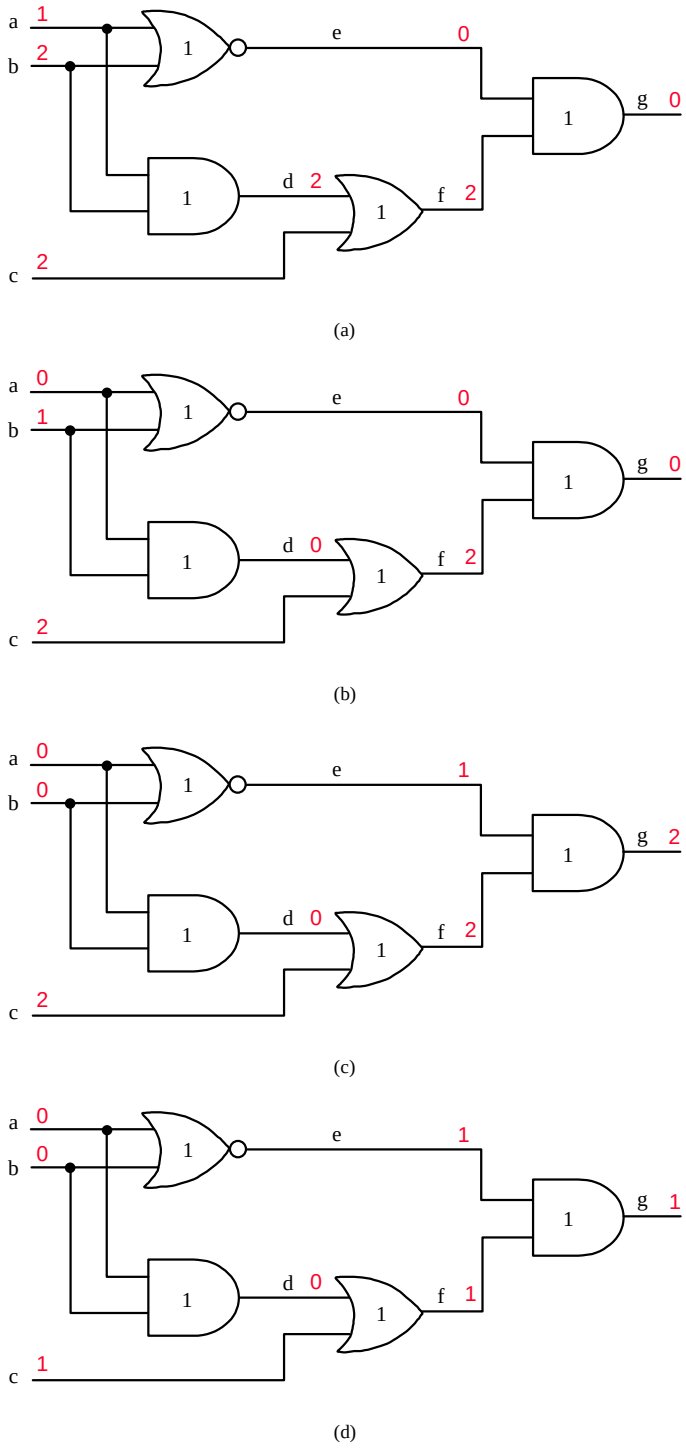


FIGURE 5.8 - PODEM algorithm example.

The timed calculus for cube simulation is given in table 5.2 for a two-input AND gate. The calculus for an OR gate would be similar, except that the role of controlling and noncontrolling values is interchanged. For each entry, we have the logical value at the AND gate output and the lower and upper bounds on the achievable delay. *d* is the delay of the AND gate.

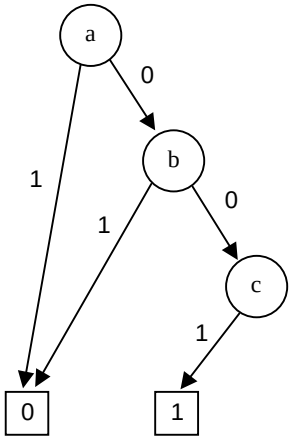


FIGURE 5.9 - Binary decision tree for PODEM algorithm.

TABLE 5.2 - Timed calculus with unknown values.

i_2 \ i_1	0	1	2
0	0 MIN(l_1,l_2)+d MIN(u_1,u_2)+d	0 l_2+d u_2+d	0 MIN(l_1,l_2)+d u_2+d
1	0 l_1+d u_1+d	1 MAX(l_1,l_2)+d MAX(u_1,u_2)+d	2 l_1+d MAX(u_1,u_2)+d
2	0 MIN(l_1,l_2)+d u_1+d	2 l_2+d MAX(u_1,u_2)+d	2 MIN(l_1,l_2)+d MAX(u_1,u_2)+d

Each input of the AND gate has an associated logical value in {0,1,2} and a lower and upper bounds on the delay corresponding to the value. For input i_1, for instance, the lower bound is l_1 and the upper bound is u_1. The logical values in the table correspond to standard three-valued simulation with 0, 1 and 2 values. To illustrate the process, it will be explained the calculation of the lower and upper bounds.

When the inputs are at {0,1}, the rules of figure 4.17 are obeyed. When we have two 0s at the AND gate inputs, we choose the minimum of the lower (upper) bounds to calculate the lower (upper) bound at the output. When we have two 1s, we choose the maximum of lower (upper) bounds to calculate the lower (upper) bound at the output. When we have a 0 and a 1, we simply use the lower and the upper bounds of the 0 input.

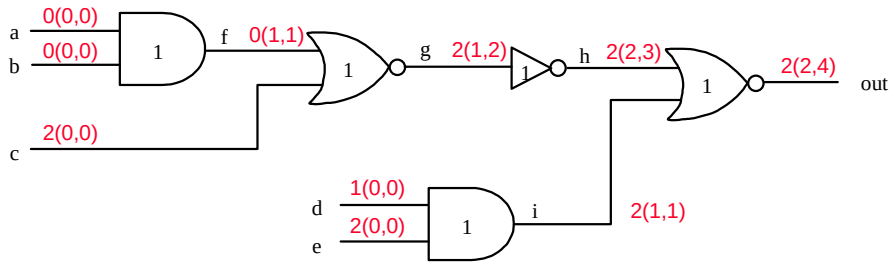
When a 2 is at the input of an AND gate, the calculus gives the lower and upper bounds on the achievable delay with either (0 or 1) setting of the 2 value. For example, consider the entry in the table when i_1 is 1 and i_2 is 2. The logical value at the AND output is 2. If i_2 is set to 0, the lower (upper) bound on the delay will be l_2+d (u_2+d). However, if i_2 is set to 1, the lower (upper) bound on the delay will be MAX(l_1,l_2)+d (MAX(u_1,u_2)+d). Since $l_2 \leq$MAX(l_1,l_2), l_2 represents the lower bound on the achievable delay, and since MAX(u_1,u_2)$\geq u_2$, MAX(u_1,u_2) represents the upper bound on the achievable delay.

Now consider the situation where i_1 is 0 and i_2 is 2. In this case the value and the upper bound of i_1 are passed through to the output. If i_2 is set to 1, its delay will not matter. If i_1 is set to 0, then the delay at the output will be MIN(u_1,u_2)+d. However $u_1 \geq$MIN(u_1,u_2), and thus we take u_1+d. By the same reasoning, the lower bound will be MIN(l_1,l_2)+d.

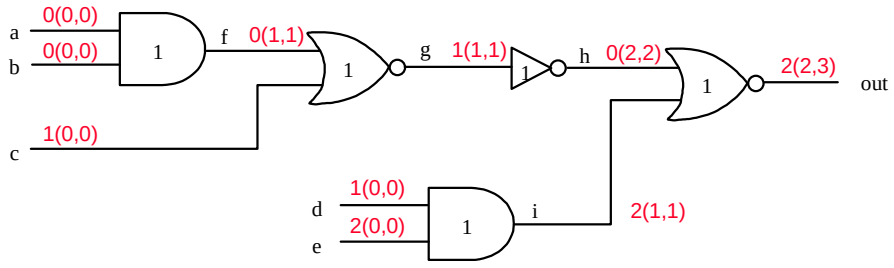
When both inputs of the AND gate are 2, the lower bound on the achievable delay at the output is clearly MIN(l_1,l_2)+d. The upper bound on the achievable delay at the output is MAX(u_1,u_2)+d. Similarly, for the other entries.

Figure 5.10 illustrates the cube simulation using timed calculus. Each wire has a logical value and two delay values. For example, 0(1,2) implies that the logical value of the wire is 0, the lower bound on the achievable delay is 1 and the upper bound on the achievable delay is 2. In figure 5.10a we have a vector with two 2 entries being simulated on the circuit. The output of the circuit is 2, and for the minterms that are contained in 00210, the maximum achievable delay is 4. When c is set to 1, the maximum achievable delay at the output is reduced to 3.

It is important to emphasize that the computed delays using the timed calculus merely give the range of the achievable delays over all the minterms contained in the partial input setting (input cube). Even if a wire is at a known value under a partial input setting v, the delay of the wire may be a range rather than a constant. For instance, consider the case where the first input of a two-input AND gate with zero delay is at 0 with l_1= u_1=4. If the second input is at 2 with l_2= u_2=3, then the lower bound on the output is achieved when the second input is set to 0; we obtain a delay of 3 at the output. The upper bound of 4 is achieved when the second input is set to 1. The output of the AND gate is 0 for both cases.



(a)



(b)

FIGURE 5.10 - Cube simulation using timed calculus.

5.3.3 Timed Test Generation

The application of the PODEM algorithm to compute the floating delay is described in the sequel. In order to use the PODEM algorithm for computing the floating delay, the timed

calculus of table 5.2 must be used. It will be necessary to justify both logical values on wires and delay values, and therefore the procedure is called *timed test generation* [DEV93].

The procedure is given a logical value $L \in \{0,1\}$ and a number δ, and asked to find an input vector, if such a vector exists, which sets the output of the circuit to L and which results in a floating mode delay $\geq\delta$.

The procedure follows the same steps as the PODEM algorithm. However, there are some important differences. The implication procedure uses the timed calculus of table 5.2 rather than a purely logical calculus. Conflicts occurring during implication may be logical conflicts or time conflicts. Logical conflicts as before correspond to the case when the output is set to the value $\bar{L}$. A time conflict occurs when the output is set to L but the upper bound on the delay at the output is strictly less than δ. In this case too we have to backtrack since we cannot find an input vector within the cube corresponding to the current settings of the primary inputs that has a delay $\geq\delta$. The procedure ends successfully if the output has been set to L and the lower bound on the computed delay at the output is$\geq\delta$.

Consider the example of figure 5.11. The delay of each gate is assigned inside the gate. We wish to justify a 1(3) at the output of the circuit. Initially, all inputs are set to 2 with delay zero. Implication results in a 2 logical value at the output with a delay range (2,3) as shown in figure 5.11a. Backtracking sets a to 1. Implication immediately sets the output to 0, which is a logical conflict. We backtrack to the most recently set primary input and change its value. Hence, a is set to 0. Implication results in a 2 at the output with a delay range of (2,3). Backtrack sets b to 1, resulting in a 0 at the output, a logical conflict, as shown in figure 5.11b. We backtrack to set b to 0. The output is unknown with the input setting corresponding to a=0, b=0 and c=2. Setting c to 1 results in the output being set to 1, but with a delay of (2,2), as shown in figure 5.11c. This is a time conflict since the upper bound on the delay is strictly less than the required delay. Finally, setting c to 0 results in a 0 at the output, a logical conflict, as shown in figure 5.11d. We have failed to find an input vector that results in a 1 at the output of the given circuit with floating mode delay 3. Note that we did not have to simulate all of the 2^3 different input minterms.

It is possible to find an input vector that results in a delay of 3 that produces a 0 at the output, as shown in figure 5.11d.

5.3.4 Backtrace

In PODEM the backtrace procedure is called when the primary output of the circuit is at an unknown value for the current primary input settings. It begins from the primary output of the circuit and traces back through the gates in the circuit to a primary input that is still at the unknown value. All gates traversed exhibit the property that their outputs are at unknown values. The required value at the output may dictate the required value at these gates, but in general there will be choices as to which path the backtrace follows. The backtrace procedure uses heuristics in following paths that begin from a primary input with an unknown value which when set will likely set the output to the desired value. The particular value that the primary input is set to, either 0 or 1, is also decided heuristically.

The backtrace procedure in timed test generation is similar to the purely logical backtrace of PODEM except that it uses both the logical and the desired delay value at the output to choose what path to follow. The backtrace procedure is called when the primary output is at the value 2, or if the lower and upper bounds on the computed delay at the output

are not equal. This is illustrated in figure 5.12, where we have a fragment of a circuit with a partial setting of its primary inputs. The logical value and the upper bound on the delay value for each wire is shown in the figure. The desired value at the primary output is 1(10) and is shown in bold. The current value of the output is unknown with an upper bound on the delay being 11.

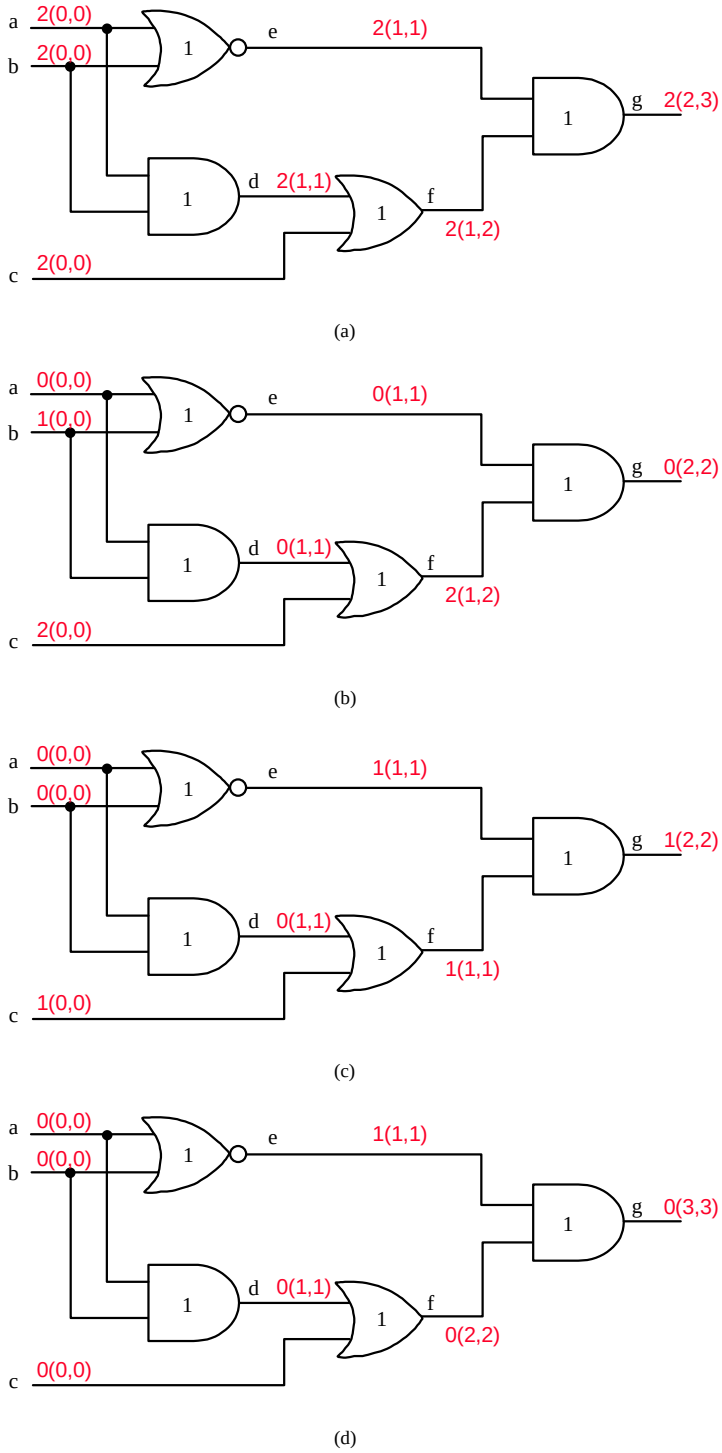


FIGURE 5.11 - Timed test generation example.

Backtracing begins at the OR gate connected to the primary output. Since we require a 1 at the primary output and the other two inputs to the OR gate are 0, we can infer that we require a 1 at the first input to the OR gate. Furthermore, the delay value required is 9. We now move to the AND gate and note that we require a 1(9) at this output (in bold in the

figure). This means that all of its inputs have to be at 1. The first and second inputs to the AND gate are at 2. We will choose to follow the path corresponding to the first input because that is the only path that can satisfy the delay value of 9 at the AND gate output. Next, we move to the NOR gate with a require output value of 1(8). Both its inputs are at unknown values, and it is possible for either input to provide a 0(6) value. The backtrace procedure randomly selects one of the inputs and continues until it reaches a primary input.

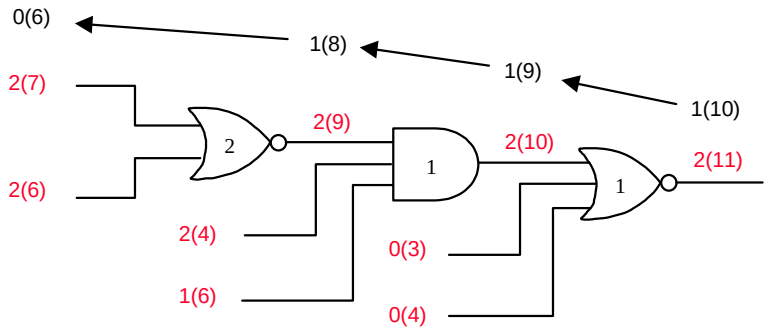


FIGURE 5.12 - Backtrace example.

5.4 SAT-Based Concurrent Path Sensitization Algorithms

As discussed in previous two sections, in the ATPG-based algorithms path sensitization is either considered explicitly, by tracing a path at a time and checking its sensitizability, or implicitly, by justifying logical values at each output while decreasing the minimal output stable time T. Both cases use modified ATPG algorithms derived from the classical D and PODEM algorithms.

In the SAT-based algorithms however, path sensitizability is implicitly tested by using the satisfiability approach (SAT). Indeed, this is a broader solution, since sensitization testing (and also any stuck-at fault detection problem) may be formulated as a general satisfiability problem.

In subsection 5.4.1 it is described a SAT-based FTA technique proposed by McGeer et al. [MCG93], to illustrate a possible implementation solution for the floating delay computation. This currently appears as the most accepted SAT-based algorithm for performing FTA of combinational circuits. Moreover, it claims to allow the analysis of circuits composed of general complex possibly asymmetric gates without using macro-expansion.

5.4.1 Philosophy of the SAT-Based Method of [MCG93]

The SAT-Based method proposed by McGeer et al. [MCG93] uses an extension of the unbounded gate delay model called XBD0 (**Extended Bounded Delay-0**), in which a third value 2 is added to the set of values {0,1} of the ordinary Boolean algebra. The third value models both the indeterminate state and the unknown state. It is also adopted a formalism called **waveform algebra** introduced by Augustin in [AUG89]. The resulting waveform-based algebraic model is analogous in the static domain to the switching algebra over Boolean operators. Within this framework, the gate delay computation model is redefined, in order to merge the logical functionality with the gate delay. Hence, a gate may be seen as an operator

that takes the waveforms applied to its inputs and generates an output waveform. It is claimed that this theory may be used to a variety of delay models.[8]

The main idea of the exact analysis method is to characterize recursively the set of all input vectors that make the signal value of a primary output stable at a logic value (either 0 or 1) by a given required time T. Once these sets are identified both for constants 0 and 1, it is necessary to compare these against the on-set and the offset of the primary output respectively, to see if the output is actually stable for all input vectors by the required time T. Circuit delay computation starts with T being set to the topological critical delay minus $\delta>0$. Then, T is gradually decreased (or equivalently, δ is increased) until some input vector cannot make the output stable by the required time T under evaluation. Guessing the next value for T can be speeded up by using a binary search. Figure 5.13 illustrates the exact analysis method.

In the following subsections the waveform calculus is presented along with the concept of characteristic functions. The implementation itself is also discussed.

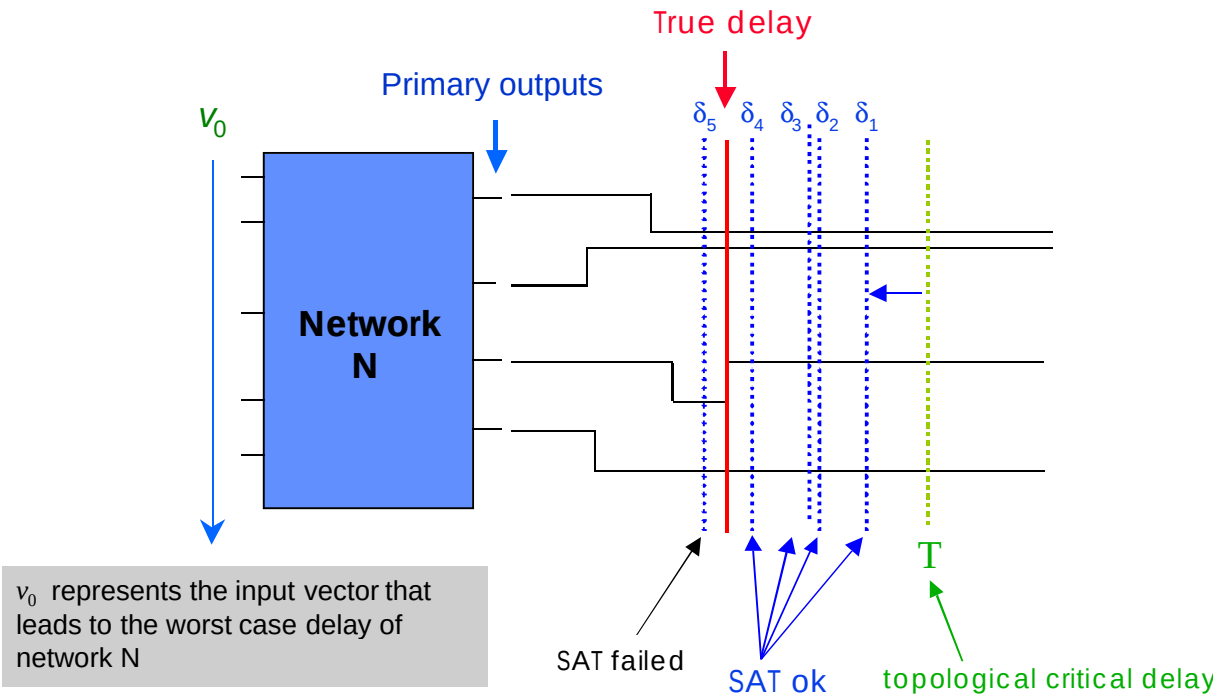


FIGURE 5.13 - Basic operation of SAT-based FTA algorithms.

5.4.2 Ternary Delay Simulation and Waveform Calculus

The ternary algebra used in this analysis is formed by adding a third value, denoted 2, to the Boolean algebra. The third value is concerned with two phenomena:

- The indeterminate state, which occurs while a gate is switching from logic 0 to logic 1 or vice-versa. In other words, it models the analog behavior of a signal.
- The unknown state, when signal is either 0 or 1, but it is not possible to precise its value. Thus, it serves to model the uncertainty in each of the variables that determine the delay of circuit components, as process variation, crosstalk, slope of the input waveform etc.

[8] Indeed, an extension to the bounded delay model, called XBD model, is also presented. As in the XBD0 model, the XBD model uses a ternary algebra.

Summarizing, 2 represents every case where the value at a gate's output cannot be assured to be a Boolean value. This leads to a straightforward extension of the binary algebra to the ternary behavior given in the following truth tables.

TABLE 5.3 - Truth table for the AND function in ternary algebra.

a.b	b = 0	b = 1	b = 2
a = 0	0	0	0
a = 1	0	1	2
a = 2	0	2	2

TABLE 5.4 - Truth table for the OR function in ternary algebra.

a+b	b = 0	b = 1	b = 2
a = 0	0	1	2
a = 1	1	1	1
a = 2	2	1	2

A ternary variable ranges over the set $\mathcal{T}^n=\{0,1,2\}$ and a ternary function g is a mapping:

$$g: \mathcal{T}^n \alpha \mathcal{T} \qquad (5.11)$$

The addition of the third value 2 to the Boolean algebra introduces a partial order $\subseteq$ over $\mathcal{T}$, defined as follows:

$$t \subseteq t \text{ for each } t \in \mathcal{T}, \text{ and further, } 0 \subseteq 2,\ 1 \subseteq 2 \qquad (5.12)$$

Indeed, the partial order operator $\subseteq$ also represents increasing instability. Thus $a \subseteq b$ means b is more unstable than a. The operator $\subseteq$ extends also to vectors:

$$\{x_1,\ldots, x_n\} \subseteq \{y_1,\ldots, y_n\} \text{ if and only if } x_i \subseteq y_i \text{ for each i} \qquad (5.13)$$

The ternary space $\mathcal{T}^n$ is related to the underlying binary space B^n through the following. A vector $x=\{x_1,\ldots, x_n\}$ over $\mathcal{T}^n$ is said to be a **vertex** if each $x_i \in \{0,1\}$. Hence, if vector x is a vertex, then $g(x_1,\ldots, x_n) \in \{0,1\}$. The evaluation rule for g over an arbitrary vector $\{x_1,\ldots, x_n\} \in \mathcal{T}$ is defined as follows:

$$g(x_1,\ldots, x_n) = \begin{cases} 1 & g(y_1,\ldots, y_n) = 1\ \forall\ \{y_1,\ldots, y_n\} \subseteq \{x_1,\ldots, x_n\} \\ 0 & g(y_1,\ldots, y_n) = 0\ \forall\ \{y_1,\ldots, y_n\} \subseteq \{x_1,\ldots, x_n\} \\ 2 & \text{otherwise} \end{cases}$$

The correspondence between vectors of the ternary space and cubes is evident. For this reason, if $g(x_1,\ldots, x_n)=1$, $\{ x_1,\ldots, x_n \}$ is said to be an implicant of g. And a maximal such implicant is said a prime of g.

The evaluation rule gives rise to the following lemma:

Lemma 5.1:

Let g be an arbitrary function of $\{x_1,\ldots, x_m\}$, where each x_j ranges over the set $\{0,1,2\}$. Let $\{p_1,\ldots, p_n\}$ be the primes of g and $\{q_1,\ldots, q_r\}$ be the primes of g'. Then, $g(x_1,\ldots, x_m)=2$ if and only if there is no prime p_i such that $p_i(x_1,\ldots, x_m)=1$ and no prime q_i such that $q_i(x_1,\ldots, x_m)=1$.

In other words, the output of a gate is constant if the values applied to its inputs are contained in a prime in the on-set or offset of the ternary function that represents it. And this evaluation is independent of the gate structure.

A delay model augments this algebra by associating a time t with each value of a gate or wire; the value of a gate at time t, $g(t)$ is a function of the values of the gate and its inputs over some interval (t_0,t_1), where $t_1 \leq t$.

Definition 5.1: **waveform**

Given a gate g, an associated waveform for g, Ω^g, is a map:

$$\Omega^g : \Re \; \alpha \; \{0,1,2\} \qquad (5.14)$$

such that, for every t, every $\varepsilon>0$, if $\Omega^g(t+\varepsilon)\neq\Omega^g(t)$, $\Omega^g(t+\varepsilon)$ and $\Omega^g(t)$ both in $\{0,1\}$, then there is some $t<t_1<t+\varepsilon$ such that $\Omega^g(t_1)=2$.

The definition of a waveform models a logic signal varying over time. The restriction that any changes in signal forces a transition through 2 models the continuity of the physical waveform and that 0 and 1 are physically separate values. Figure 5.14 shows an example of a ternary waveform.

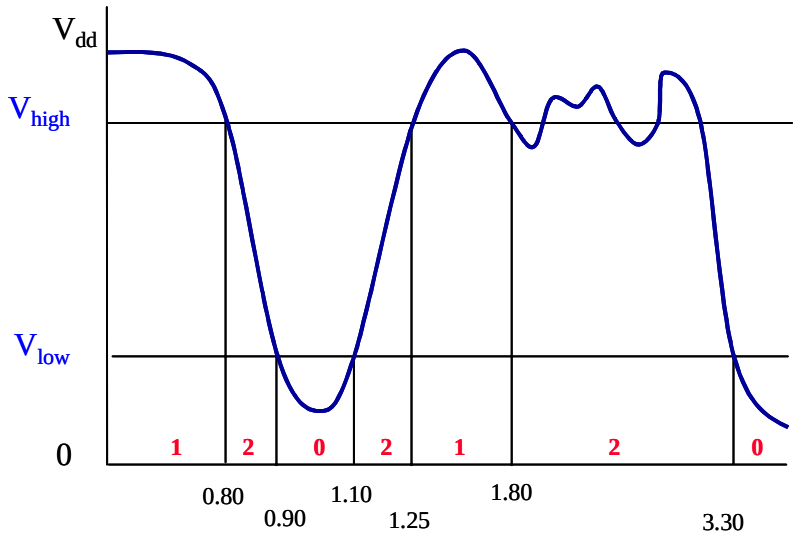


FIGURE 5.14 - Example of ternary waveform.

Given a waveform Ω and a real interval **I**, the **partial waveform of interval I**, Ω_I, is the waveform Ω restricted to the domain **I**.

Definition 5.2: **map for a gate**

Map *M* for a gate *G*, with inputs $f_1,\ldots, f_r$:

$$M : \Omega^{f1}_{(0,t)} \times \Omega^{f2}_{(0,t)} \times\ldots\times \Omega^{fr}_{(0,t)} \times \Omega^{G}_{(0,t)} \;\alpha\; \Omega^{G}(t) \qquad (5.15)$$

is a **delay model** if for any subset *S* of inputs where $\Omega^{fs}_{(0,t)}$ is a constant function for each $s \in S$, and cube $c=\prod_{s\in S} (f_s=\Omega^{fs}_{(0,t)})$, G(c) is a constant, then $\Omega^{G}(t)=G(c)$.

The definition treats transitions on a gate. The output waveform of a gate at t is determined by the input waveform as well as gate waveform occurring between 0 and some time t' preceding t. By convention, 0 is chosen as the base time; choosing a fixed base time for all model mappings enforces the intuition that the value given by the delay model should be independent of any time shift.

A Boolean algebra consists of a set of variables, each of which can assume a value within {0,1}; a combinational network, evaluated statically, is the realization of a function over its input variables. A delay model, a network of gates and wires, a set of input variables, and a set of possible waveforms for each input variable yield a **waveform algebra**. An assignment of one waveform to each input corresponds to an input **waveform vector**. The set of all waveform vectors forms a **wave space**, which (for n inputs) is denoted W_n. A waveform vector is the equivalent, in wave space, to an input vertex in Boolean space. Further, a pair (gate, delay model) is the equivalent, in wave space, to the gate in Boolean space. A pair (gate, delay model) takes an input wave and produces an output wave. This is illustrated in figure 5.15.

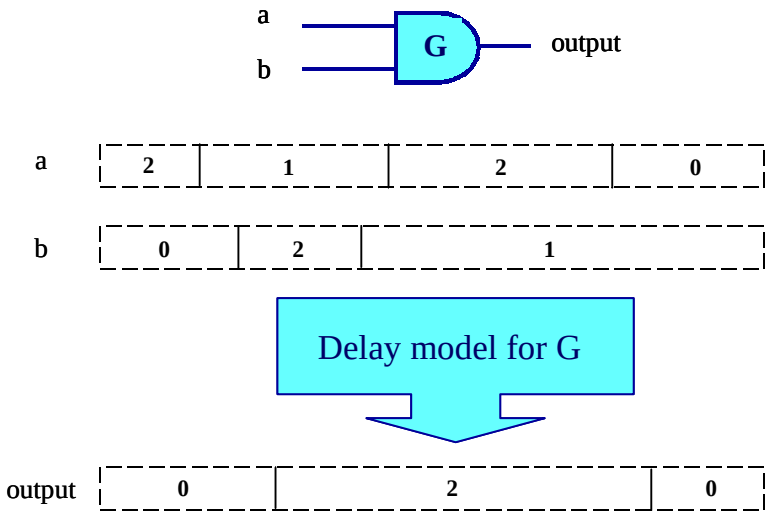


FIGURE 5.15 - Delay model for a gate in the wave space.

The concept of characteristic functions also holds for the wave space.

Definition 5.3: **characteristic function**

A **characteristic function** over a wave space is a mapping:

$$\chi : W^n \ \alpha \ \{0,1\} \quad (5.16)$$

conventionally, χ is associated with some set $S \subseteq W^n$: $\chi(w)=1$ if and only if $w \in S$.

The exact analysis method uses extensively characteristic functions, particularly those of the form:

$$\chi^{\Omega^g} = \{w | w \text{ is a waveform producing waveform } \Omega^g \text{ on signal } g\} \quad (5.17)$$

In current case, characteristic functions represent sets of waveform vectors. A waveform is an uncountable sequence of symbols from the set {0,1,2}, representing the values of the wave at each real point in time t. However inputs are not toggled infinitely often. Thus, as a result, there are relatively few waveform vectors of interest and these are easily encoded. For example, in most timing analysis problems, the inputs switch only once, at t=0. In this case, the waveform space $\mathbf{W}^n$ may be represented as the space $B^n \times B^n$, where (v_1,v_2) represents the binary input vectors applied at t=–∞ and t=0, respectively. Under these circumstances, the wave characteristic function is:

$$\chi : B^n \times B^n \ \alpha \ \{0,1\} \quad (5.18)$$

and is conveniently represented in the standard ways.

Specific delay computation models typically avoid an enumeration by giving rules for computing the output waveform given the input waveforms. In [MCG93] the various existing delay computation models are discussed in the context of the ternary algebra. Here, the discussion will be restricted to the bounded and unbounded delay models, renamed **Extended Bounded Delay** model (XBD) and **Extended Bounded Delay-0** model (XBD0), respectively

Under the XBD model, the delay associated to input *i* of gate g is within the range $[d_i^{min}, d_i^{max}]$ and represents a transition region of uncertain width. As a result, pure translation in time of the input waveform to the output is not allowed. The computation of $\Omega^g(t)$ is given as a two-step process:

$$F_i(t)= \begin{cases} \Omega^{fi}_{(t-d_i^{max},\, t-d_i^{min})} & \Omega^{fi}_{(t-d_i^{max},\, t-d_i^{min})} \text{ is a constant} \\ 2 & \text{otherwise} \end{cases}$$

$$\Omega^g(t) = g(F_i(t), \ldots, F_n(t)) \quad (5.19)$$

The values $F_i(t)$ form "effective values" of the input waves, as presented to the output, at time t. If $\Omega^{fi}_{(t-d_i^{max},\, t-d_i^{min})}$ is a constant, then input f_i has not changed over the interval $(t-d_i^{max}, t-d_i^{min})$; since any change in the state of f_i can only propagate to the output g at t if that change in state occurred between $(t-d_i^{max}, t-d_i^{min})$, it follows that the present state of input f_i is simply the constant state of the interval $(t-d_i^{max}, t-d_i^{min})$. If, on the other hand, f_i changed state between $(t-d_i^{max}, t-d_i^{min})$, then the presented value of the input might be any state of f_i between the intervals, or a transient; the only reasonable value to choose in such circumstances is 2. The value of $\Omega^g(t)$ is then easily obtained as the static ternary evaluation of g on the $F_i(t)$.

The XBD0 model is equivalent to the XBD model, except that $d_i^{min}=0$ for any input *i* of a gate g. The XBD0 model is the model underlying both viability [MCG89] and floating [CHEN93][DEV93] delay calculations.

5.4.3 Computing the Floating Delay under the XBD0 Model

Based on the waveform space concept and the associated definitions, the combinational timing verification problem can be defined.

Definition 5.4: **combinational timing verification problem**

Given a circuit *C*, a delay model *M* and a family of possible waveforms on the combinational inputs of *C* such that each such waveform is a constant binary value on the intervals (–∞,0) and (t,∞) (i.e., each input changes state only in the interval (0,t)), the **combinational timing verification problem** is to find the least positive *d* such that, for any possible combination of input waveforms, $\Omega^g_{(d,\infty)}$ is a binary constant for each circuit output g.

Under the XBD0 model, an input waveform for input *a* is one of two forms:

$$\Omega^a = \begin{cases} x_{(-\infty,\infty)} & \text{or} \\ x_{(-\infty,0)} 2_{[0,t_a]} \bar{x}_{(t_a,\infty)} & \end{cases}$$

where $x \in \{0,1\}$ and t_a is a positive constant associated with input *a*. This leads to the following result concerning properties of circuit waveforms.

Lemma 5.2:

Let g be any gate in a logic circuit. Under the XBD0 model, under any waveform vector, $\Omega^g(t) \in \{1,0\}$ for t>0 implies $\Omega^g(t_1) = \Omega^g(t)$ for all $t_1 \geq t$.

The proof of lemma 5.2 is by induction. By definition, the result holds for the primary inputs. Suppose it is true for all gates of level<*N* and consider a gate g at level *N*, and an arbitrary waveform *w*. Let Ω^g be induced by *w* with $\Omega^g(t)=1$. We have $g(f_1, \ldots, f_n)$, and by the XBD0 evaluation model, $1 = \Omega^g(t) = g(F_1(t), \ldots, F_n(t))$, where

$$F_i(t) = \begin{cases} \Omega^{fi}_{(t-d_i^{max},\, t)} & \Omega^{fi}_{(t-d_i^{max},\, t)} \text{ is a constant} \\ 2 & \text{otherwise} \end{cases}$$

Since $g(F_1(t), \ldots, F_n(t))=1$, by lemma 5.1 there is some prime *p* of g, such that $p(F_1(t), \ldots, F_n(t))=1$. Consider an arbitrary $t_1>t$. Since each input to g is of level<*N*, if $\Omega^{fi}(t) \in \{0,1\}$, then by induction $\Omega^{fi}(t_1) \in \{0,1\}$, and hence $F_i(t) \in \{0,1\} \Rightarrow F_i(t_1) = F_i(t)$; hence, since *p* is a positive unate function of its literals, $p(F_1(t_1), \ldots, F_n(t_1))=1$, and by lemma 5.1 $\Omega^g(t_1)=1$.

This lemma permits a characterization of the waves given by the XBD0 model.

Theorem 5.1:

Let g be any gate in a logic circuit. Under the XBD0 model and any allowed input waveform vector:

$$\Omega^g = \begin{cases} x_{(-\infty,\infty)} & \text{or} \\ x_{(-\infty,0)} 2_{[0,t_g]} \bar{x}_{(t_g,\infty)} & \text{or} \\ x_{(-\infty,0)} 2_{[0,t_g]} x_{(t_g,\infty)} & \end{cases}$$

for some $x \in \{0,1\}$.

This theorem is an immediate consequence of the preceding lemma.

By this theorem, any waveform of a gate g is fully characterized. The next step is to use the derived theory to resolve the timing analysis problem. Equation 5.17 states the following:

$$\chi^{\Omega^g} = \{w | w \text{ is a waveform producing waveform } \Omega^g \text{ on signal } g\}$$

Now, consider the set $\chi^{\Omega^g_{(t,\infty)} \in \{0,1\}}$.

This is the set of all input waveforms such that g is a binary constant on the interval (t,∞). Under the XBD0 model, the delay of a circuit with primary outputs $o_1,\ldots,o_n$, under input waveform vector *w*, is:

$$d_w = \max_i \min \left\{ t \perp w \in \chi^{\Omega^{o_i}_{(t,\infty)} \in \{0,1\}} \right\} \qquad (5.20)$$

Hence the delay over all waveform vectors may be written:

$$d_w = \max_i \min \left\{ t \perp \chi^{\Omega^{o_i}_{(t,\infty)} \in \{0,1\}} = 1 \right\} \qquad (5.21)$$

It is important to note that d is the exact delay of the circuit. It is the exact minimum time after which all outputs have stabilized. Thus, for any $d_1 < d$ there is an input waveform vector and some output o_i such that $\Omega^q(d_i) = 2$. Now:

$$\chi^{\Omega^{o_i}_{(t,\infty)} \in \{0,1\}} = \chi^{\Omega^q_{(t,\infty)} = 0} + \chi^{\Omega^{o_i}_{(t,\infty)} = 1} \qquad (5.22)$$

From lemma 5.2.

$$\chi^{\Omega^g_{(t,\infty)} = 0} = \chi^{\Omega^g(t) = 0} \qquad (5.23)$$

$$\chi^{\Omega^g_{(t,\infty)} = 1} = \chi^{\Omega^g(t) = 1} \qquad (5.24)$$

Thus, it is necessary to calculate $\chi^{\Omega^g(t)=0}$ and $\chi^{\Omega^g(t)=1}$ to complete the formulation for the exact delay computation.

Lemma 5.3:

Let g be a gate with inputs $f_1,\ldots, f_r$. Let $p_1,\ldots, p_n$ be all the primes of g, and $q_1,\ldots, q_n$ all the primes of g'. Then:

$$\chi^{\Omega^g(t)=1} = \sum_{i=1}^{n}(p_i(F_1,\ldots,F_r)=1)\prod_{k=1}^{r}\sum_{v\subseteq F_k}\chi^{\Omega^{f_k}(t-d_k^{\max})=v} \qquad (5.25)$$

$$\chi^{\Omega^g(t)=0} = \sum_{j=1}^{m}(q_j(F_1,\ldots,F_r)=1)\prod_{k=1}^{r}\sum_{v\subseteq F_k}\chi^{\Omega^{f_k}(t-d_k^{\max})=v} \qquad (5.26)$$

For the proof of this lemma consider the following. If $w\in \chi^{\Omega^g(t)=1}$, then $\Omega^g(t)=1$ when w is applied as the input waveform vector. Hence there is some prime p_i such that $p_i(F_1,\ldots,F_r)=1$, and, further, $F_k \supseteq \Omega^{f_k}(t-d_k^{\max})$ i.e.,

$$w\in \sum_{v\subseteq F_k}\chi^{\Omega^{f_k}(t-d_k^{\max})=v} \qquad (5.27)$$

for all k. Conversely, let $w\in \sum_{v\subseteq F_k}\chi^{\Omega^{f_k}(t-d_k^{\max})=v}$ for all k, and, further, let $p_i(F_1,\ldots,F_r)=1$. Then $\Omega^g(t)=1$ by the evaluation rule, and hence $w\in \chi^{\Omega^g(t)=1}$.

The expression for both $\chi^{\Omega^g(t)=0}$ and $\chi^{\Omega^g(t)=1}$ may be rewritten so that they depend only on the sensitization functions of the fanin of g.

Lemma 5.4:

Let g be a gate with inputs $f_1,\ldots, f_r$. Let $p_1,\ldots, p_n$ be all the primes of g, and $q_1,\ldots, q_n$ all the primes of g'. Let $F_k(p)$ denote the value of input f_k in a prime p. Then:

$$\chi^{\Omega^g(t)=1} = \sum_{i=1}^{n}\prod_{k=1}^{r}\left[\left\{\left(F_k(p_i)=1\right)\Rightarrow \chi^{\Omega^{f_k}(t-d_k^{\max})=1}\right\}\left\{\left(F_k(p_i)=0\right)\Rightarrow \chi^{\Omega^{f_k}(t-d_k^{\max})=0}\right\}\right] \quad (5.28)$$

$$\chi^{\Omega^g(t)=0} = \sum_{j=1}^{m}\prod_{k=1}^{r}\left[\left\{\left(F_k(q_j)=1\right)\Rightarrow \chi^{\Omega^{f_k}(t-d_k^{\max})=1}\right\}\left\{\left(F_k(q_j)=0\right)\Rightarrow \chi^{\Omega^{f_k}(t-d_k^{\max})=0}\right\}\right] \quad (5.29)$$

These results follow from equations 5.25 and 5.26 because if $F_k(p)=2$, then

$$\sum_{v\subseteq F_k(p)}\chi^{\Omega^{f_k}(t-d)_k^{\max}=v}=1$$

And this follows since for any gate f_k,

$$\chi^{\Omega^{f_k}(t-d_k^{\max})=0}+\chi^{\Omega^{f_k}(t-d_k^{\max})=1}+\chi^{\Omega^{f_k}(t-d_k^{\max})=2}=1$$

The previous equations suggest a recursive scheme for the computation of exact path delay under the XBD0 model. This can be accomplished by using the path recursive function technique described in [MCG91a]. The sensitization functions $\chi^{\Omega^g(t)=0}$ and $\chi^{\Omega^g(t)=1}$ at a gate g are computed using only the sensitization functions of its immediate fanin.

The overall process has two steps. In the first step the times for which sensitization functions are required at each gate are determined by a reverse topological traversal: given a list of times at a gate g, the times required at each fanin f_k are determined by subtracting the

delay from f_k to g from each time in the list. In the second step, distinct path delays (from primary inputs) are determined at each gate; this is done by a forward propagation of path lengths using a topological traversal. At the end of this step, for each gate g, there is a list of times required in computing the sensitization functions at g and a list of actual path lengths up to g from any primary input. Suppose the sensitization function $\chi^{\Omega^g(t_r)=1}$ is to be computed. Let t_a be the greatest path length to g such that $t_a \leq t_r$. Since no event occurs between t_a and t_r, $\chi^{\Omega^g(t_r)=1} = \chi^{\Omega^g(t_a)=1}$. This matching between required times and path lengths is performed at each gate for each required time.

Finally, the characteristic functions of the sensitization functions $\Omega^g(t)=1$ and $\Omega^g(t)=0$ are built up in topological order. A node representing a characteristic function is created for each path length, which is matched by some required time. The function of each such node is linear in the number of primes of the gate (or its complement), and the number of fanin of the gate. The existence of a sensitizable path is determined by calling a satisfiability program, as described in [MCG91a].

The complexity of the method deserves some comments. As mentioned by the authors themselves in [MCG93], in very large circuits and/or circuits with distinct delays on nearly all the connections, the number of characteristic functions become very large. Indeed, if t is the least time for which $\chi^{\Omega^g(t)}$ is to be computed at a gate g, the number of functions required at g is bounded above by the distinct path lengths between t and the longest path length terminating at g. This potential explosion may be alleviated by using some pruning rules.

6 Functional Timing Analysis of Circuits Containing Complex Gates

Functional timing analysis has been massively investigated from the beginning of current decade. First FTA techniques performed single path sensitization by using a modified D algorithm and a variety of sensitization criteria. Indeed, in most of proposed criteria the sensitization conditions were kindly simplified attempting to reduce the time taken for testing conditions, while providing conservative (but still safe) delay estimates.

Since path-by-path identification of false paths has proved to be impractical, concurrent path sensitization has come into focus. With concurrent path sensitization, FTA techniques shifted from a “path enumeration-based false path identification” approach to a “delay enumeration-based delay computation” approach. Besides discarding the costly enumeration phase, concurrent path sensitization techniques have also allowed the use of the exact floating-mode sensitization criterion, which uniquely furnishes the exact circuit delay under the floating mode of operation.

However, all theory underlying the methods used to test path sensitizability and also the sensitization conditions themselves were developed assuming combinational circuits to be made of simple gates, i.e., inverters, AND/NAND and OR/NOR gates. Consequently, if a combinational circuit containing more complex gates is to be analyzed, the FTA tool must be able to recognize such gates and treat the circuit properly under the adopted delay computation model. This may be accomplished either by adding a pre-processing phase or by extending the circuit delay computation method/algorithm and the sensitization conditions of the chosen criterion.

The availability of efficient CMOS macrocell generators, as the ones presented in [CAD99] and [MOR97], and efficient library-free technology mapping tools [REI97] has made possible the extensive use of complex gates (mainly static CMOS ones) in the physical design of large combinational blocks. Hence, the ability of handling circuits containing complex gates became highly desirable for current FTA tools.

This chapter investigates FTA of circuits containing complex gates, which is intended to be the main contribution of this thesis. Section 6.1 discusses some basic issues associated to the technology mapping and layout generation making use of static CMOS complex gates (SCCGs). Section 6.2 discusses possible solutions for performing FTA of circuits containing complex gates. Section 6.3 presents an extended ATPG-based algorithm able to perform FTA of circuits containing complex gates.

6.1 Technology Mapping and Layout Generation for Circuits Containing Complex Gates

In traditional VLSI physical design methods random logic blocks are generated using pre-characterized (standard) cells from a library. The reasons for adopting this library-based approach were the ease of developing tools to automate layout generation and the good electrical performance predictability of the resulting design. However, the amazing evolution of CMOS technology has shortened the life cycle of libraries: re-design and re-

characterization is becoming more frequent, which in turn increases the cost associated to their maintenance. Another drawback of the library-based approach is the limited number of primitives: the higher is the number of cell options, concerning both function availability and driving capability, the higher is the flexibility offered to the technology mapping tool. Obviously, this last tradeoff establishes a conflict, since libraries with many elements are more expensive to maintain. A third issue is concerned with electrical performance predictability. In current submicronic technologies, the delay introduced by the routing dominates over the delays of the gates. Hence, the linear physical model (see section 2.4), which is commonly used in delay estimation of cell-based layouts, is no longer able to offer the required accuracy for assuring the correct operation of the fabricated design.

An alternative to the cell-based layout generation is the macrocell generation approach. A macrocell generator does not use cells from a library. Instead, it generates each element (transistors and connections) according to a layout pattern that is intrinsically programmed within its algorithms. The pioneer works on automatic cell generation were those of Lopez and Law [LOP80] and Uehara and Cleemput [UEH81]. The formers' introduced a layout style known as **gate matrix**, while the latters' presented the so-called **linear matrix** layout style. Both works were originally developed having in mind the automatic generation of cell libraries. Current macrocell generators are mainly based on the linear matrix style and are able to generate combinational modules with up to some tenths of thousands of transistors [MOR94][MOR97][CAD99][EIJ94][GUR97].

A very important feature of macrocell generators is that they are potentially able to generate any type of static CMOS gate. This introduces a new paradigm in the technology mapping problem, since the major part of existing technology mapping tools use the **Boolean matching** technique, which is not an efficient solution for libraries with a large number of elements [REI98]. This new problem is being referred to as the "library-free" or "virtual library" technology mapping problem [REI95], due to the absence of a physical library of cells. Recently proposed solutions for performing technology mapping over a virtual library rely on associating the arcs of the BDD that represents a given logic function to the transistors of static CMOS gates [REI98][RIE96].

The shift from the traditional cell-based to the library-free layout generation strategy is unavoidable, not only because cell libraries became too expensive, but also because physical models used to estimate performance (up till now) do not work for current CMOS technologies.

On the other hand, the increase of flexibility provided by the library-free strategy may only be explored if appropriate tools are available. For instance, it is reported in [REI98] a reduction in the total number of transistors when technology mapping considers not only simple CMOS gates (inverters, NAND and NOR gates), but also more complex static CMOS gates, generally referred to as SCCGs [REI95] (and formerly as supergates). This result indicates a possible increase in the performance of designed circuits that must be carefully investigated by the use of appropriate CAD tools.

Figure 6.1 shows a physical design flow using the FUCAS (FUll Custom Automatic Synthesis) layout generation strategy [REI99]. Among the basic features of FUCAS, it is proposed the extensive use of SCCGs as a means of improving both logic density and electrical performance. Within FUCAS strategy, the current macrocell generator to be used is TROPIC, which has been developed as part of a doctoral thesis [MOR94]. Concerning technology mapping for SCCGs, there is a specific tool under development called TRABUCO, which is the evolution of a technology mapping prototype called TABA [REI98].

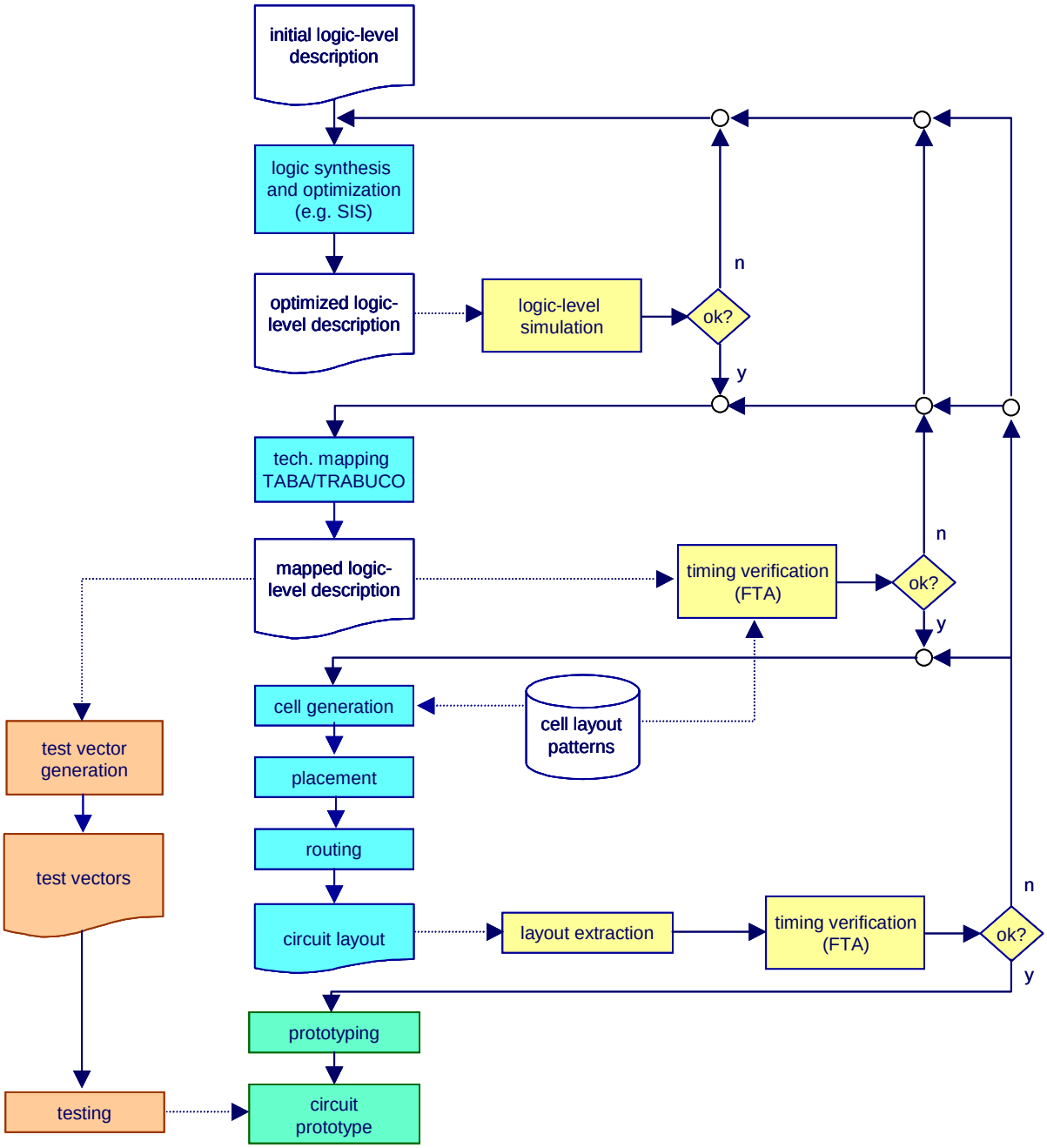


FIGURE 6.1 - Physical design flow using the FUCAS layout generation strategy.

The design flow with the FUCAS strategy obviously requires a timing verification tool able to validate the timing requirements of designs. Hopefully, such tool must be able to provide fast, safe and accurate delay estimates of large combinational blocks containing SCCGs that will be used to guide the physical design as a whole. Doubtlessly, the timing analysis technique is the natural choice for performing timing verification because it is input-stimuli independent. Furthermore, it presents a lower computation cost when comparing to circuit simulation, thus being much faster. Within timing analysis techniques, functional timing analysis is the one able to provide the most accurate estimates. Hence, the physical design with FUCAS represents a real need for a FTA tool able to handle complex gates.

6.1.1 Simple Gates, General Complex Gates and Static CMOS Complex Gates

The expressions “simple gates” and “complex gates” are intensively used in the fields of logic synthesis, technology mapping, test generation and timing analysis. As long as such expressions may be employed with slightly different meanings in each of the mentioned fields, it seems to be advisable beginning by redefining them within the context of the current work.

In the context of this thesis, a **simple gate** corresponds to a (possible) physical implementation of the basic operators of the Boolean algebra, complemented or not. Thus, the whole set of simple gates may be enumerated as follows: *n*-input AND, OR, NAND, NOR gates and the inverter (sometimes called NOT gate), with $n = \{2, 3, 4,\ldots\}$. Particularly, the set of **simple CMOS gates** is a subset of the simple gates set, since the basic Boolean operators that may be directly implemented with the CMOS technology are: *n*-input NAND, NOR gates and inverters.

A **general complex gate,** or simply **complex gate,** corresponds to a physical implementation of any single output Boolean function of higher complexity than the basic Boolean operators. Strictly speaking, a **static CMOS complex gate,** or **SCCG** for shortly**,** corresponds to a complex gate implemented using the CMOS technology. Consequently, any SCCG implements an inverting Boolean function.

Although this work is intended to investigate FTA algorithms that are able to operate on general complex gate networks, let us first turn our attention to the SCCG case, as a first step for extending FTA theory for complex gates and at the same time, taking advantage on the real-case design opportunity provided by the FUCAS design strategy.

A static CMOS gate may also be classified as a “full restoring” network of CMOS transistors because it is composed of two distinct networks of transistors: a PMOS transistor network connected between the power supply (Vdd) and the gate output and a NMOS transistor network connected between the ground (Gnd) and the gate output. The networks have equal number of transistors and are commonly referred to as NMOS-network and PMOS-network. For the sake of simplicity, we are going to assume that a SCCG is a static CMOS gate in which both networks are made up from serial/parallel only associations and also that the NMOS-network is the dual of the PMOS-network, in terms of transistor association. Figure 6.2 shows an example of SCCG. Note that, for an *n*-input static CMOS gate, there are *n* pairs of NMOS-PMOS transistors connected together through the gate terminal to form each of the *n* inputs to the gate.

It is interesting to notice that in a simple CMOS gate the NMOS and PMOS networks are either parallel only or series-only. On the other hand, SCCGs present series/parallel transistor NMOS and PMOS transistor networks with at least 3 inputs.

Static CMOS gates (including SCCGs) may be classified according to the number of serial/parallel transistors encountered in their NMOS/PMOS networks. The set of static CMOS gates formed by no more than *n* (*p*) serial NMOS (PMOS) transistors is referred to as virtual “library” [REI98] and may be designated by SCG(*n*,*p*). We may also use the notation SCCG(2,2) to designate the subset of SCG(2,2) formed by SCCGs only. Figure 6.3 (borrowed from [REI98]) shows the elements of the set SCG(2,2), while table 6.1 (also borrowed from [REI98]) shows the number of elements for various virtual libraries.

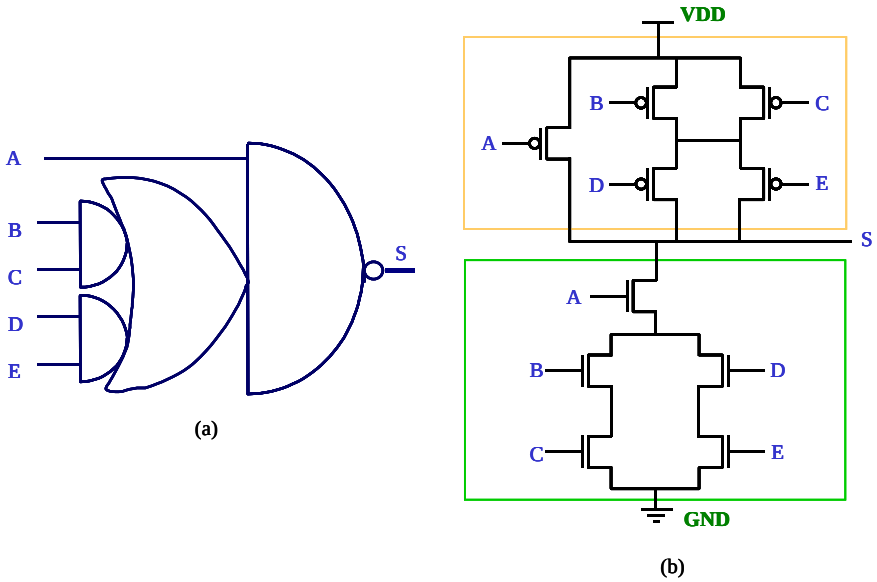


FIGURE 6.2 - Example of SCCG.

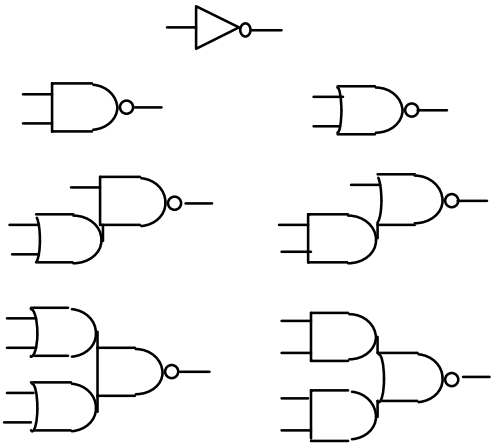


FIGURE 6.3 - Elements of the virtual library SCG(2,2) [REI98].

TABLE 6.1 - Number of elements for various virtual libraries [DET87].

		number of serial PMOS transistors				
		1	**2**	**3**	**4**	**5**
number of serial NMOS transistors	**1**	1	2	3	4	5
	2	2	7	18	42	90
	3	3	18	87	396	1677
	4	4	42	396	3503	28435
	5	5	90	1677	28435	425803

6.2 The Applicability of Existing Functional Timing Analysis Techniques for Circuits Containing Complex Gates

Although many FTA algorithms were developed in the past decade, most of them are not able to work directly on circuits containing complex gates. In the case of ATPG-based algorithms, such limitation is due to the fact that all sensitization theory has been developed for simple gates only. Of course, the extension of sensitization criteria to complex gates results in more complicated rules, making traditional single path sensitizability testing even more complicated. At a first glance, SAT-based algorithms seem to be more promising since characteristic equations are potentially able to represent any Boolean function. However, in order to reduce the complexity in solving SAT instances, some SAT-based algorithms assume simple gates-only combinational circuits.

The simplest solution for treating circuits with complex gates relies on replacing each complex gate by an equivalent sub-circuit composed of simple gates. This technique is known as macro-expansion and offers the advantage of allowing the use of gate-level FTA tools originally developed for simple gates [MCG91][HSU98]. Macro-expansion presents two severe drawbacks, however [HSU98]. Firstly, it is very difficult to accurately model the delay of macro-expanded complex gates. Consequently, the resulted circuit delay estimates may not be sufficiently accurate. The use of sophisticated delay models for macro-expanded gates can reduce this loss in accuracy, but may overly increase the complexity of the macro-expansion step, also increasing the overall execution time. Secondly, macro-expansion procedures increase the number of nodes in the circuit graph. In an ATPG-based algorithm, extra nodes represent potential extra circuit lines to be justified. In the case of SAT-based algorithms, each extra node originates an extra characteristic function. In any case, the overhead in execution time will depend on the complexity of equivalent sub-circuits used to model complex gates.

The second possible solution relies on modifying the sensitization tests in order to handle complex gates. Such modifications include not only the sensitization criterion but also the sensitization testing algorithm itself. Since path sensitizability testing may be accomplished by different algorithms, this solution represents indeed a group of possible solutions.

In [HSU98], for instance, a FTA algorithm able to operate directly on circuits containing complex gates is presented. In order to avoid macro-expansion, the exact floating-mode sensitization conditions are extended to consider complex gates. The extended sensitization conditions are then used within a single path sensitization procedure derived from the classical D-algorithm. The practical results presented in [HSU98] allow for a comparison between some macro-expansion procedures, featuring different delay models for complex gates, and the direct application of the extended sensitization criterion. However, all results were obtained through the use of a single path sensitization algorithm. Although the advantage of directly treating sensitization of complex gates becomes clear, single path sensitization procedures suffer from the path explosion problem and thus do not represent the state-of-the-art in FTA.

As presented in section 5.3, the timed-test generation procedure of Devadas et al. [DEV93a] is an ATPG-based multiple path sensitization algorithm derived from the PODEM [GOE81] test generation algorithm. In the timed-test generation procedure the delay computation problem is transformed into a set of single stuck-at fault test generations in which the algorithm tries to justify “timed” stuck-at faults at the primary outputs of the circuit. This is accomplished by using an appropriate subset of PODEM functions along with a set of

three-valued timed calculus rules that allows the application of “timed” logic values through the circuit lines. The three-valued timed calculus assumes the exact floating-mode sensitization criterion. Although the possibility of extending this procedure to circuits containing complex gates is mentioned in [DEV93a], this issue is not further detailed.

SAT-based algorithms, and especially the one presented in section 5.4 [MCG93], implicitly consider path sensitizability by applying the concept of sensitization characteristic functions. The characteristic functions are computed recursively for each circuit node, assuming a given required time T at the circuit outputs. The elegance of this approach relies on the fact that characteristic functions are potentially able to represent any type of Boolean functionality, including the general complex gate case. However, SAT solver algorithms are heavily CPU intensive and the only way to make the approach feasible for timing analysis is to control the size of the SAT problem, or equivalently, the size of the characteristic functions network. Kukimoto et al. [KUK97] observes that using the two basic pruning rules presented in [MCG93] is not sufficient to assure that the method can be used in the analysis of more complex designs, and proposes two approximation schemes. One of them is based on designer’s knowledge of circuits (the control/data dichotomy proposed in [YAL95]). The other one concerns a simplification on calculating arrival times at circuit gates. Unfortunately, both approximations may result in a significant loss of accuracy, which may partially cancel the gain obtained in execution time.

Another problem with the SAT-based approach relies on the fact that the use of more accurate delay models increases the number of SAT instances that must be solved due to wrong decisions taken during the solving steps. In order to alleviate this problem, Silva proposed in [SIL99] a mixed approach that maintains circuit structure information while using a SAT solver. By doing so, it is possible to apply acceleration techniques commonly used in test generation, as non-chronological backtracking and recursive learning.

Due to the mentioned difficulties presented by SAT-based implementations and considering that many acceleration techniques exist for ATPG algorithms, the use of the timed-test generation procedure to work directly on combinational blocks containing complex gates seems to be the most promising possibility, in terms of both CPU execution time and delay estimate accuracy. Hence, the rest of this chapter is devoted to further investigate this option.

6.3 ATPG-Based FTA of Circuits Containing Complex Gates

Although the timed-test generation procedure has already been addressed in subsection 5.3.3, it will be necessary to go deeper into the implementation details in order to detect which modifications are needed to allow it operating directly on circuits containing complex gates.

The timed-test generation procedure serves as the core of the ATPG-based concurrent path sensitization FTA algorithm called TrueD-F [DEV93a]. Therefore, before proceed the analysis of the timed-test generation procedure itself it is convenient to examine how TrueD-F computes the floating delay of a combinational block.

Consider a single-output combinational block. TrueD-F determines the maximum delay at the block’s output by answering the question “is this delay greater than or equal δ?” δ is initially set to $T-\varepsilon_0$, where T is the circuit topological delay and ε_0 is a small quantity greater than 0. To answer the question, input cube simulation is accomplished by using a modified version of the PODEM algorithm [GOE81], the so-called “timed-test generation procedure”

[DEV93a]. While the answer to the question is “no”, the timed-test generation procedure is successively called for $\delta_i = T-\varepsilon_i$, with i=0,1,2,...., $\varepsilon_{i+1} > \varepsilon_i$. A “yes” answer means that the output’s maximum delay is between the current value of δ (say $T-\varepsilon_k$) and the previous one ($T-\varepsilon_{k-1}$). Hence, $T-\varepsilon_{k-1}$ represents a safe upper bound on the maximum delay of current output. In case a more accurate delay value is needed a binary search may be performed on the interval [$T-\varepsilon_k$, $T-\varepsilon_{k-1}$].

Since the block’s output may stabilize either with logic value 0 or with logic value 1, the timed-test procedure must be performed twice for each delay value δ_i (except in case a “yes” answer occurs for the first logic value tested). Figure 6.4 depicts the timed-test generation procedure.

In the case of a multi-output combinational block, the timed-test generation procedure must be called for each output for both 0 and 1 logic values and for each δ_i. A first approximation to the lower bound on the circuit delay (i.e., $T-\varepsilon_k$) is determined by the first “yes” answer arriving.

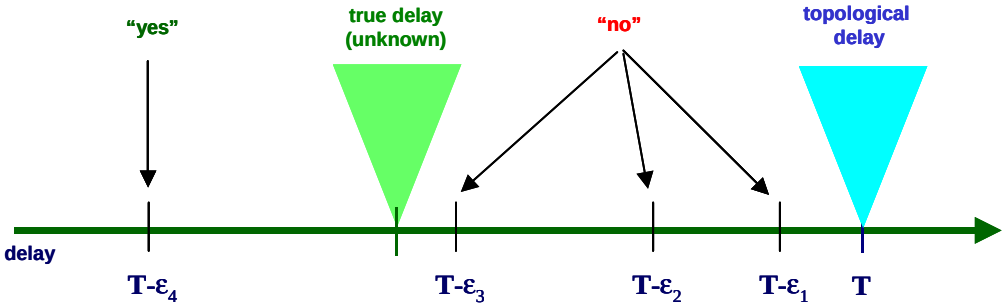


FIGURE 6.4 - Timed-test generation procedure applied to a single-output circuit.

In order to determine whether the delay of a circuit’s output is greater than or equal to δ_i for a logic value `lvalue`, the timed-test procedure tries to justify `lvalue` at the considered output with delay greater than or equal δ_i. This is done by using cube simulation, in a PODEM-like fashion. Since PODEM allows for a systematic and exhaustive exploration of the input space, once it fails in finding an input cube that justifies `lvalue` at a circuit’s output under the considered delay δ_i, then the answer to the question is “no” for that output (and for logic value `lvalue`). In other words, the problem of determining the maximum delay is transformed into a test generation problem for a single stuck-at fault at the circuit’s output.

The topmost call of the procedure `timed_test` is described by the pseudo-code in figure 6.5. As in the purely logical PODEM, there is a justification list `jlist` holding circuit lines that must be justified. The procedure begins by inserting a given primary output line `po` into `jlist` with logic value `lvalue` (either 0 or 1) and assuming `delay` as circuit delay lower bound to be tested (δ_i). Then, procedure `SEARCH_1` is called.

```
timed_test(po,delay,lvalue)
{
  v.value = lvalue;
  v.lower = delay;
  v.upper = INFINITY;

  modify_jlist (po,v,jlist);
  backward schedule po;

  status = SEARCH_1(jlist);
  return(status);
}
```

FIGURE 6.5 - Pseudo-code for the topmost call of the timed-test generation procedure.

The search procedures are described by the pseudo-codes of figures 6.6 and 6.7. They are similar to the search procedures of PODEM. The procedure SEARCH_1 calls a BACKTRACE procedure to find a primary input which logical value is currently unknown, beginning from the primary output po. The primary input is (initially) set to logic value 1 and the IMPLY procedure is called. This procedure performs a timed-test cube simulation that considers exact floating-mode sensitization conditions (in PODEM, it corresponds to three-valued cube simulation without any delay information). The implication procedure may produce a logical or a temporal conflict. If no conflict occurs, SEARCH_1 is called recursively. The procedure terminates successfully in SEARCH_1 if the justification list is empty. In case a conflict occurs in SEARCH_1 the algorithm backtracks to the most recent primary input assignment. The primary input is set to logic value 0, and SEARCH_2 procedure shown in figure 6.7 is called.

Failure results if either the BACKTRACE procedure is unable to find a primary input to set or if the space has been completely explored without success in SEARCH_2. A conflict occurs when it is not possible to set a given circuit line to the required logic value at the required time. In the case of conflict or failure, the network must be restored to the state it was immediately prior to the primary input setting that caused the conflict or the failure.

```
SEARCH_1(jlist)
{
  if(length of jlist is zero) return SUCCEED;

  if(BACKTRACE(gate,value,delay,&pi,&pi_value)==FALSE)
    return(FAILED);

  if(IMPLY(pi,pi_value,jlist)!=IMPLY_CONFLICT)
  {
    search_status = SEARCH_1(jlist);
    if(search_status == FAILED)
    {
      restore the state of the network;
      search_status = SEARCH_2(jlist,pi,1-pi_value);
    }
  }
  else
  {
     restore the state of the network;
     search_status = SEARCH_2(jlist,pi,1-pi_value);
  }
  return(search_status);
}
```

FIGURE 6.6 - Pseudo-code for the first search procedure.

The timed-test generation procedure assumes that each circuit gate (and additionally, each circuit edge) has a “timed-value” variable, composed of three fields: a lower and an upper bound on the delay of the gate (actually, stable times) and a logic value. A pre-processing step initializes all logic values to the unsigned value, i.e. 2, and lower and upper bounds to the minimal and maximal topological delays from any primary input. In the test generation phase, while primary inputs are set to known logic values, the delay bounds are tightened during the forward simulation due to the sensitization or blocking of paths. The lower and upper bounds are also modified by backward implication, by the time new values are inferred at particular gates, given logic and delay values requirements at the primary outputs.

```
SEARCH_2(jlist,pi,pi_value)
{
   backtracks = backtracks + 1 ;
   if(backtracks > BACKTRACK_LIMIT) return(ABORTED);

   if(IMPLY(pi,pi_value,jlist)!=IMPLY_CONFLICT)
   {
     search_status = SEARCH_1(jlist);
     if(search_status == FAILED)
       restore the state of the network;
   }
   else
   {
     search_status = FAILED;
     restore the state of the network;
   }
   return(search_status);
}
```

FIGURE 6.7 - Pseudo-code for the second search procedure.

Another important component of the procedure is the justification list. Gates are added to the justification list during backward implication and removed from it during both forward and backward implication. At any time, the justification list holds the gates whose logic or delay values must be justified by setting more primary inputs. By the time the justification list becomes empty the search is successfully completed. Hence, the answer to the question “is the delay of the circuit (when a logic value lv is assigned to the output) greater than or equal δ?” is “yes”. On the other hand, if the search space has been completely enumerated without the justification list becoming empty, the search has failed: the answer to the question is “no”. As a third situation, the search may be abandoned due to excessive backtracking, with the question remaining unanswered.

It is important to notice that the same “timed-value” variable is used for storing actual logic and delay values at a gate due to primary input settings, and the required values inferred by backward implication. The difference is that in the latter case, the gate will be on the justification list. Indeed, all gates whose inputs do not produce the values in the gate “timed_value” variable should be on the justification list. On the other hand, any gate whose inputs produce the values in the gate “timed_value” variable should not be on the justification list. These two assertions form the best definition for the justification list.

Having presented the topmost procedure and the two search procedures, let us examine the `imply` procedure, which is called by both `search1` and `search2` procedures. The `imply` procedure is showed by the pseudo-code of figure 6.8. The primary input is set to the given logic value and the effect is propagated into the circuit using the `forward_set` procedure. The `forward_set` procedure schedules the fanout of the changed primary input, and event-driven timed simulation is performed by `forward_imply`. `forward_imply` is followed by backward implication (procedure `backward_imply`). These two procedures are iterated over till the circuit values do not change. In general, setting a gate in the circuit to a particular value during forward or backward implication requires the forward scheduling of all its fanouts, and the backward scheduling of all fanouts that are on the justification list.

Conflicts are detected during both implication procedures. These conflicts may be time conflicts or logical conflicts. It is worth to remind that gates may be removed from the justification list during both forward implication and backward implication. However, gates may be added to the justification list only during backward implication.

```
imply(pi,pi_value,jlist)
{
  v = pi.timed_value;
  v.value = pi_value;
  status = forward_set(pi,v,jlist);

  while(status==IMPLY_NORMAL)
  {
       status=forward_imply(jlist);
       if(status!=IMPLY_CONFLICT)
            status=backward_imply(jlist);
  }
return status;
}
```

FIGURE 6.8 - Pseudo-code for the imply procedure.

The key for determining whether it is possible to justify a logic value at a circuit's output within the required time or not relies on using a three-valued timed calculus that takes into account the conditions stated by the exact floating-mode sensitization criterion. Consider a 2-input AND gate with delay d. Each input i_i of the gate may present a logic value in {0,1,2} with lower and upper bounds on its delay given by l_i and u_i, respectively. The term "delay of a signal" will be used rather than "stable time" [CHE93] because even gates presenting the value 2 at the output have lower and upper delays. The timed calculus for cube simulation for 2-input AND gates and 2-input OR gates are given in tables 6.2 and 6.3, respectively. In these tables lv is the logic value at the gate's output, while l_o and u_o represent the lower and the upper bounds on the delay at the gate's output, respectively.

TABLE 6.2 - Three-valued timed calculus for a 2-input AND gate.

i_2	i_1	0	1	2
0	lv	0	0	0
	l_o	$\min(l_1,l_2)+d$	l_2+d	$\min(l_1,l_2)+d$
	u_o	$\min(u_1,u_2)+d$	u_2+d	u_2+d
1	lv	0	1	2
	l_o	l_1+d	$\max(l_1,l_2)+d$	l_1+d
	u_o	u_1+d	$\max(u_1,u_2)+d$	$\max(u_1,u_2)+d$
2	lv	0	2	2
	l_o	$\min(l_1,l_2)+d$	l_2+d	$\min(l_1,l_2)+d$
	u_o	u_1+d	$\max(u_1,u_2)+d$	$\max(u_1,u_2)+d$

TABLE 6.3 - Three-valued timed calculus for a 2-input OR gate.

i_2	i_1	0	1	2
0	lv	0	1	2
	l_o	$\max(l_1,l_2)+d$	l_1+d	l_1+d
	u_o	$\max(u_1,u_2)+d$	u_1+d	$\max(u_1,u_2)+d$
1	lv	1	1	1
	l_o	l_2+d	$\min(l_1,l_2)+d$	$\min(l_1,l_2)+d$
	u_o	u_2+d	$\min(u_1,u_2)+d$	u_2+d
2	lv	2	1	2
	l_o	l_2+d	$\min(l_1,l_2)+d$	$\min(l_1,l_2)+d$
	u_o	$\max(u_1,u_2)+d$	u_1+d	$\max(u_1,u_2)+d$

6.3.1 Extending the Timed-Calculus to Complex Gates

The rules presented in tables 6.2 and 6.3 are easily extended to *n*-input AND and OR gates. Furthermore, they may also be extended to NAND and NOR gates if gate function polarity is accounted for. Therefore, before considering their use for complex gates, it is convenient to generalize the three-valued timed calculus to any type of *n*-input simple gates. In order to do that, let us first recall the concepts of controlling/non-controlling and controlled/non-controlled values, as they are defined within ATPG theory. The controlling/controlled value of an AND (OR) gate *g*, c(*g*)/cd(*g*) is logic 0 (logic 1), while the non-controlling/non-controlled value of an AND (OR) gate *g*, nc(*g*)/ncd(*g*), is logic 1 (0). Then, given an *n*-input AND/OR gate *g*, we may classify the cases listed in tables 6.2 and 6.3 according to the following groups:

1. Cases in which at least one of the inputs of *g* presents a controlling value (c(*g*)). The others may present either non-controlling values (nc(*g*)) or the 2 value;
2. The case in which all inputs of *g* present non-controlling values (nc(*g*));
3. Cases in which at least one of the inputs of *g* presents a 2 value, but none presents a controlling value (c(*g*)). The others may present non-controlling values (nc(*g*)).

Now, we may generalize the timed-calculus for simple gates, and at the same time include NAND and NOR gates, by the rules listed in table 6.4.

TABLE 6.4 - Generalized three-valued timed calculus for *n*-input simple gates.

group		rules	*lv*
1	l_o	$\min\{ l_i \mid i = c(g) \text{ or } i=2 \} + d$	$pol(g) \oplus c(g)$
	u_o	$\min\{ u_j \mid j = c(g) \} + d$	
2	l_o	$\max\{ l_i \} + d$	$pol(g) \oplus nc(g)$
	u_o	$\max\{ u_j \} + d$	
3	l_o	$\min\{ l_i \mid i=2 \} + d$	2
	u_o	$\max\{ u_j \} + d$	

Now, in order to apply the rules of table 6.4 to AND gates, for instance, one has to assume c(*g*)=0, nc(*g*)=1 and pol(*g*)=0. In the case of a NOR gate, c(*g*)=1, nc(*g*)=0 and pol(*g*)=1. Each of the three groups of rules are further illustrated by figures 6.9, 6.10 and 6.11, respectively. Note that for the ease of explanation, we have assumed a single delay per gate. More sophisticated gate delay computation models may also be used, however. Subsection 6.3.2 elaborates more on the gate delay computation model issue.

The extension of the generalized three-valued timed calculus to complex gates is straightforward since we assume each gate function to be represented as a factored form, using a convenient data structure. For instance, consider the SCCG of figure 6.2, re-edited in figure 6.12. The logic function implemented by this gate, $S = \overline{A \cdot ((B \cdot C) + (D \cdot E))}$, is easily retrieved from its transistor-level description and may be represented as a tree. By examining this "function tree" one notice that, given an assignment of input timed-values (logic values with lower and upper bounds on delays), the corresponding output timed-values may be computed by successively applying the rules showed in table 6.4 to each subtree, beginning from the bottom most, since the following is assumed:

1. All subtrees, except the topmost one, have no delay and polarity equals zero
2. The gate delay is applied to the gate output lower and upper bounds, i.e., only to the timed-values resulted from the evaluation of the topmost subtree.
3. gate polarity is treated when the topmost subtree is processed.

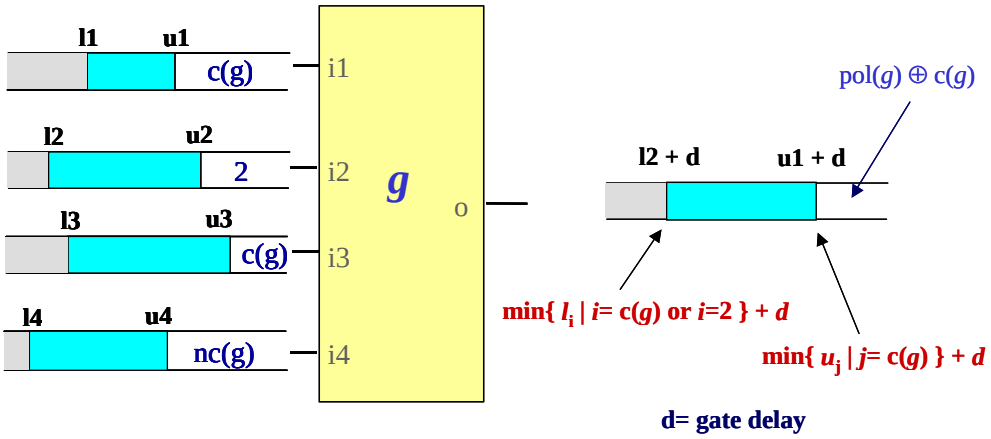


FIGURE 6.9 - Three-valued timed calculus for group 1.

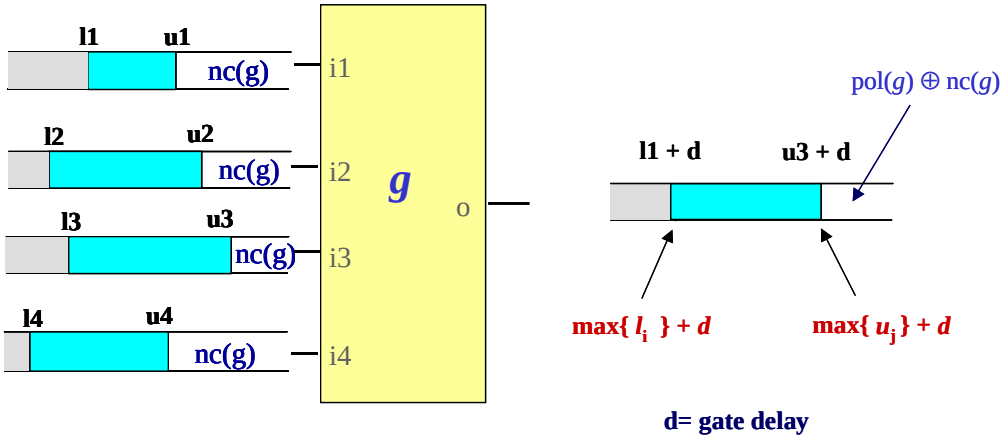


FIGURE 6.10 - Three-valued timed calculus for group 2.

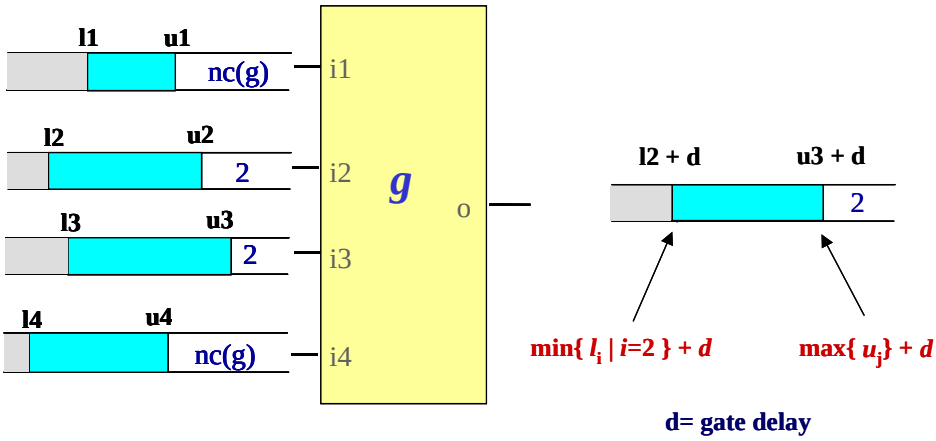


FIGURE 6.11 - Three-valued timed calculus for group 3.

The first and second assumptions allow us to split the timed evaluation of a SCCG into two independent steps. In the first step the output logic value and the lower and upper bounds with zero gate delay are computed. We may call such delay interval as “first-order delay

bounds". In the second step actual lower and upper bounds are computed by adding the gate delay to the first order delay bounds.

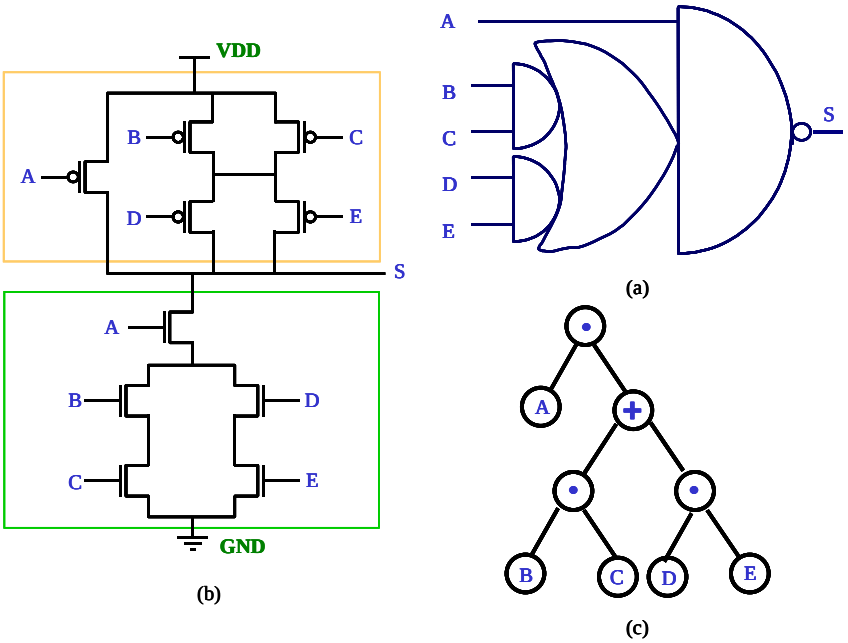


FIGURE 6.12 - Example of SCCG: logic-level symbol (a), transistor schematics (b) and function tree (c).

In order to facilitate its comprehension and also its use, we may rewrite the three-valued timed calculus rules to be used in the first step of SCCG evaluation as shown in table 6.5.

TABLE 6.5 - Three-valued timed calculus for evaluating SCCGs.

group		lower subtrees		topmost subtree
		first-order delay bounds	*lv*	output *lv*
1	l_o	min{ l_i | i= c(g) or i=2 }	c(g)	pol(g) ⊕ c(g)
	u_o	min{ u_j | j= c(g) }		
2	l_o	max{ l_i }	nc(g)	pol(g) ⊕ nc(g)
	u_o	max{ u_j}		
3	l_o	min{ l_i | i=2 }	2	2
	u_o	max{ u_j }		

Figure 6.13 illustrates the three-valued timed-test evaluation procedure on the SCCG of figure 6.12. The intermediate dummy nodes shown in this figure are actually dispensable in case of a recursive implementation.

It is interesting to notice that in case of SCCGs, all subtrees belonging to the same level corresponds to either the same logic operation or to a leaf. In the (original) function tree a leaf represents a gate input. Another property of SCCG function trees concerns the fact that any two subsequent levels present different logic operations, either AND or OR.

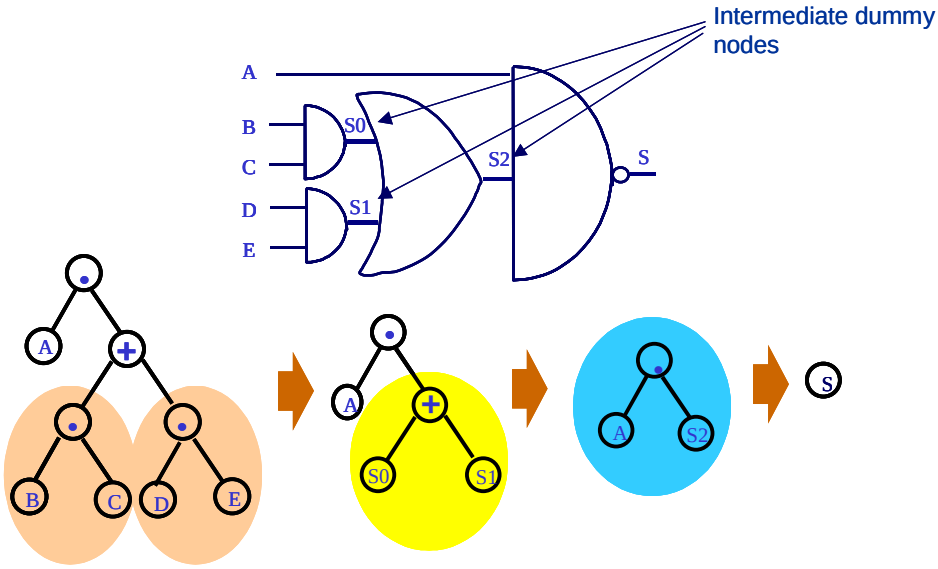


FIGURE 6.13 - Using the three-valued timed calculus for evaluating a SCCG.

The procedure for evaluating SCCGs presented above may be directly applied to any complex gate in which the only possible polarity inversion concerns the gate output. In case of existing an inversion in any input or intermediate subtree, this may be accounted for by a polarity-checking step occurring at the end of subtree evaluation.

Having proposed a procedure able to evaluate any complex gate, one may imagine its natural extension to characterize macrocells, looking for performing “hierarchical floating delay computation”. Of course, the accuracy of such hierarchical approach depends on both gate and circuit delay computation models. Delay computation models for complex gates are studied in depth in the following subsection.

6.3.2 The Floating Delay of a Gate

Once the procedure for evaluating the output logic value and the first-order delay bounds of a gate has been detailed, it is necessary to investigate gate delay computation models that may be adopted to determine the gate delay value to be used in the computation of the lower and upper delay bounds of the gate.

As showed in section 2.5, the basic types of gate delay computational models fall into three main categories: fixed delay models, bounded delay models and unbounded delay models. Particularly, in section 4.1 it was proved, through theoretical arguments, that the fixed and the unbounded models are equivalent under the floating mode, since we assume maximal gate delay values for the former case.

On the other hand, concerning verification tools, it is clear that the accuracy of any estimation depends on the accuracy of the models underlying the tools used. In this sense, (physical) component delay models, gate delay computation models and circuit delay computation models form the core of any FTA tool. Further, the accuracy of a FTA tool depends not only on the accuracy of each model individually, but also on the compatibility of such models.

Concerning the circuit delay computation model, the timed-test procedure assumes the floating mode. However, it makes no restriction with respect to component delay or to gate delay computation models. Hence, it is advisable to further investigate the relationships

between these models, in order to identify the existing options, as well as their respective accuracies.

As mentioned in the previous paragraph, the timed-test procedure computes the floating delay of a circuit. Thus, the study starts by assuming the floating mode circuit delay computation model. In addition, this study targets at investigating the circumstances under which the most accurate estimates are achieved and which is the computational effort involved.

In order to establish relationships between the circuit delay computation model and both gate delay computation and physical models, consider the inverter of figure 6.14(a). Consider also the application of the logic value 1 at the inverter's input, which will result in the logic value 0 at its output. As long as the floating mode does not make assumptions on the previous state of circuit nodes, under such delay model, the 1 at the inverter's input may represent either a rising transition or a steady 1. Given this, it results that the 0 at the inverter's output may be either a falling transition or a steady 0 (figure 6.14(b)). However, in order not to underestimate the circuit delay, we have to consider the worst case, that is, a falling output transition.

Note that, although the floating mode does not make any assumption on the previous state of a node, the whole history needed for analyzing the inverter of figure 6.14(a) corresponds to the two input situations shown in figure 6.14(b). Thus, in the inverter case, the falling delay under the floating mode corresponds to the falling delay under the transition mode. By equivalent arguments, it is possible to affirm that the rising floating delay of the inverter corresponds to its rising transition delay.

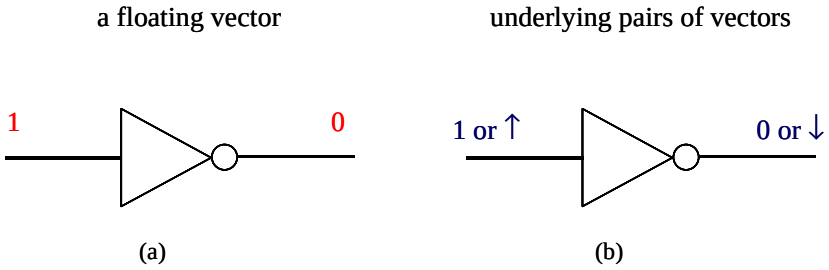


FIGURE 6.14 - The relationship between floating mode (a) and transition mode (b).

Consider now the 3-input NAND gate of figure 6.15(a). Consider also the input vector *v*=(a=1;b=1;c=0) is applied to this NAND gate. In order to determine the floating delay of the NAND gate under *v*, it is necessary to examine all possibilities represented by such vector, finding the worst-case delay. To begin with, it is necessary to recall that under the floating mode, a 0 represents either a falling transition or a steady 0, while a 1 represents either a rising transition or a steady 1. Thus, by replacing 0 by ↓ or by 0 and replacing 1 by ↑ or by 1, we get a total of 8 possibilities. Not surprisingly, each of these cases represents a pair of vectors (figure 6.15(b)). This, of course, establishes a relationship between the floating mode and the transition mode, concerning delay modeling of logic gates.

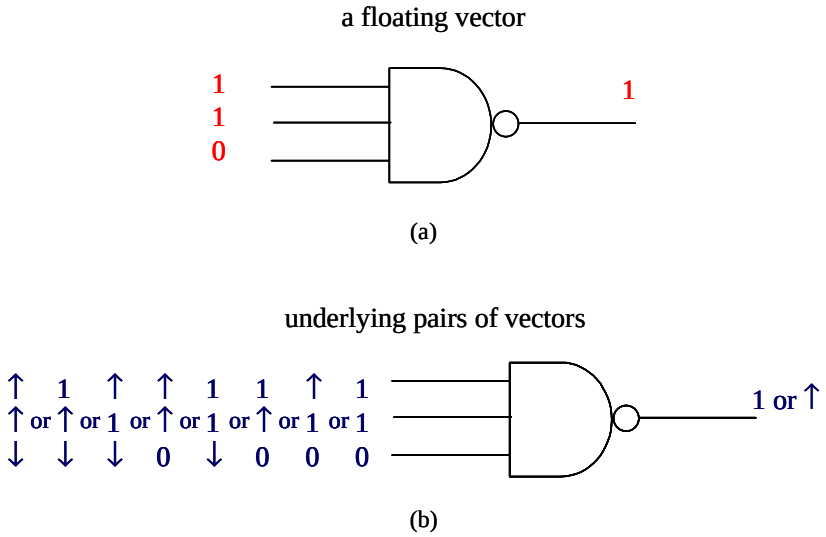


FIGURE 6.15 - The relationship between floating mode and transition mode: a floating vector applied to a 3-input NAND gate (a) and the 8 underlying pairs of vectors (b).

Before proceeding with the discussion, let us introduce the definitions of floating vector and floating delay:

Definition 6.1: **floating vector** and **floating delay**

> A **floating vector** is an assignment of logic values applied to the gate inputs or to the circuit inputs, when the circuit is assumed to operate under the floating mode. The computed gate (circuit) delay by assuming such mode is then called **floating delay** of the gate (circuit).

Given the definition of floating vector, it is worth to highlight that an *n*-variable floating vector contains exactly 2^n pairs of vectors, among them one is made up from two equal vectors. Table 6.6 shows all existing floating vectors and respective pairs of vectors for a 3-input NAND gate. The light-hashed pairs produce a single transition at the gate's output while the dark-hashed ones may either produce a glitch or do not affect the gate's output. The bottom row shows the output logic values resulted from the application of each of the 8 possible input floating vectors.

Going a little further, it is possible to build a "floating mode/transition mode equivalence table" like table 6.6 for each of the 256 existing 3-input logic functions. Since the association between floating vectors and pairs of vectors is fixed, 3-input logic functions will be distinguished by the output logic values and also by the vector pairs responsible for transitions or glitches. Naturally, it is also possible to build such equivalence table for any *n*-input logic function.

A fundamental contribution of the floating mode/transition mode equivalence table is that it allows for identifying which are the possible gate delay computation models to be used within the computation of the floating delay of a circuit. More specifically, the table says that a gate may present at least one (possibly) different delay value for each input floating vector. In a conservative assumption, we may consider that the most detailed gate delay computation model under the floating mode corresponds to assuming a gate delay per input floating vector. We will refer to this model as the **vector delay model**.

TABLE 6.6 - Relationship between floating model vectors and transition model vectors.

input vector	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
underlying pairs of vectors	000	001	010	011	100	101	110	111
	↓00	↓01	↓10	↓11	↑00	↑01	↑10	↑11
	0↓0	0↓1	0↑0	0↑1	1↓0	1↓1	1↑0	1↑1
	00↓	00↑	01↓	01↑	10↓	10↑	11↓	11↑
	↓↓0	↓↓1	↓↑0	↓↑1	↑↓0	↑↓1	↑↑0	↑↑1
	↓0↓	↓0↑	↓1↓	↓1↑	↑0↓	↑0↑	↑1↓	↑1↑
	0↓↓	0↓↑	0↑↓	0↑↑	1↓↓	1↓↑	1↑↓	1↑↑
	↓↓↓	↓↓↑	↓↑↓	↓↑↑	↑↓↓	↑↓↑	↑↑↓	↑↑↑
output logic value	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**

A less accurate, but still correct gate delay computation model corresponds to grouping the input floating vector situations according to the transition type taking place at the gates' output. Thus, the gate will be assigned with a falling delay, taken from the delays of those vectors that cause a falling output transition, and a rising delay, whose value comes from the vectors producing a rising output transition. This corresponds to a pair of **separate fall/rise delay model**.

A third gate delay computation model corresponds to assigning a single delay per gate, thus being called **single delay model**. Naturally, this is the most crude delay model, although correct from the floating delay computation's point of view.

It is important to remark that, in order to get a robust floating delay computation, the gate delay values considered in each of the three models must represent an upper bound on the delay of that gate for all instances of a whole circuit family. This means that the adopted physical delay model or component delay estimation method must furnish worst-case delay values for each circuit component.

A more careful analysis of table 6.6 allows us to conclude that the pin-to-pin delay format normally used in *standard-cell* characterization is not appropriate for floating delay computation since this kind of model takes into account only a part of all input pairs of vectors that may cause a transition at the gate's output. For instance, in the case of a 3-input gate, the pin-to-pin delay format would cover only six pairs of vectors: ↓11, 1↓1, 11↓, ↑11, 1↑1, 11↑. That is, only single input transition situations.

A second contribution of the floating mode/transition mode equivalence table is that it lists all pairs of vectors underlying a given floating vector. Such information is useful for carrying out the delay characterization of a gate through simulation. In this case, only those pairs of vectors able to produce a transition at the gate output need be simulated. This may reduce significantly the number of cases to be simulated: in the case of a 3-input NAND gate, only 26 cases from a total of 64. On the other hand, if gate delays are obtained through any analytical formulation, this information may be used to calibrate or to adapt the delay modeling formulae in order to cover only the floating vector pairs of interest.

6.3.3 Gate Delay Computational Models and Timed Forward Implication for SCCGs

Table 6.6 allows us identifying all pairs of vectors that influence on the delay of a given input-floating vector, in the case of a 3-input NAND gate. However, thus far it was not mentioned the criterion to determine the gate delay value itself under the considered input-floating vector. Of course, whatever this criterion is, it is necessary to guarantee the correctness of the delay computation procedure (see section 2.6), that is, it is necessary to guarantee that the delay computation procedure provides a safe upper bound on the circuit delay. Therefore, let us examine what are the circumstances leading to the safe upper bound floating delay for the timed-test generation procedure.

As already explained, the timed-test generation procedure operates on a single primary output at a time. Given a delay value δ and a logic value lv (0 or 1), it will result in success if it is possible to justify lv at the primary output with lower delay greater than or equal δ. Thus, a success means that the delay at the considered primary output is never less than δ for lv. On the other hand, if it is not possible to justify lv at the primary output, or if it is possible to justify it but the upper bound on the delay is strictly less than δ, then the procedure results in fail. Figure 6.16 illustrates these two situations.

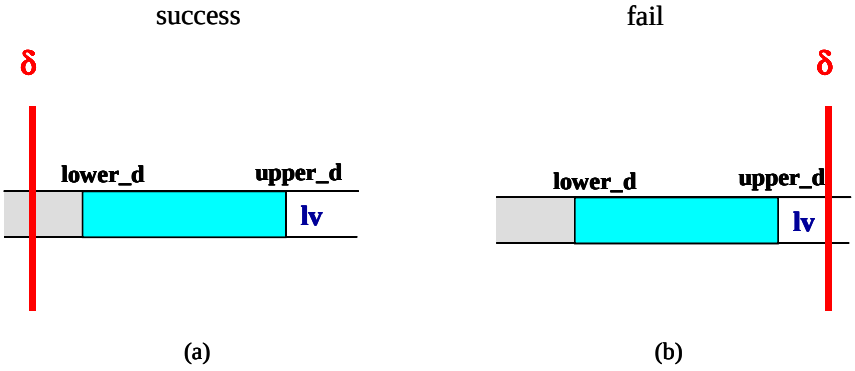


FIGURE 6.16 - Success (a) and fail (b) conditions for the timed-test generation procedure.

Given a multi output circuit and a delay value δ, if the timed-test procedure results in fail for both 0 and 1 logic values for each primary output, then the next step for searching the circuit floating delay is to guess a new δ, smaller than the previous one and restart to test each primary output. At this point, it is assumed with 100% of certitude that the circuit floating delay is less than the already tested δ. This certitude must be assured, whatever the models used are.

This way, the criterion for determining the delay of a gate under a given floating vector must reinforce the previous premise. Considering the whole process of determining the floating delay of a circuit with the timed-test generation procedure, a safe upper bound estimate is associated to the action of avoiding decreasing the value of δ whenever the accuracies of the underlain models are questionable. Indeed, this is exactly the case: we are looking for a criterion that allows determining the floating delay of a gate from a set of possible transition delays. Hence, it seems to be logical that for determining the floating delay of a gate one should take the greatest value among all transition delays associated to a given floating input vector because this may contribute to shift right both lower and upper delays at circuit nodes with respect to the time axis. Shifting right delay bounds, by its turn, is a

conservative action because it increases the chances of the timed-test generation to return a success. Due to the already posed arguments, the delay of a gate under a given input-floating vector must correspond to the maximum among the delays caused by the pairs of vectors underlying such floating vector.

On the other hand, the separate fall/rise delay model corresponds to an approximation of the vector delay model in which all floating vectors causing the same logic value at the gate output are grouped together. As a consequence, a gate may exhibit as many different delays for a given (output) logic value as the number of floating vectors that cause this logic value. Since we have already been conservative in determining floating vector delays, one may improve the gate delay computational model by considering a pair of delays for each logic value instead of a single value per logic value. To do that, each logic value 0 and 1 is assumed to have a maximum and a minimum delay value, taken from the slowest and fastest floating vectors that cause the logic value, respectively. The minimum value is added to the lower delay bound, while the maximum value is added to the upper delay bound. Using the same arguments, the single delay model may be transformed into a pair of delays model in which a maximum and a minimum delay values are assumed.

Once we have derived safe rules for determining the delay of gates under any of the three gate delay computation models, it is necessary to consider the application of such rules while computing the lower and upper bounds of a gate within a forward implication procedure. Recalling the rules of the three-valued timed calculus described in tables 6.4 and 6.5, the input floating vector situations that may occur can be classified according to the following three groups:

1. Cases in which at least one of the inputs of g presents a controlling value (c(g)). The others may present either non-controlling values (nc(g)) or the 2 value;
2. The case in which all inputs of g present non-controlling values (nc(g));
3. Cases in which at least one of the inputs of g presents a 2 value, but none presents a controlling value (c(g)). The others may present non-controlling values (nc(g)).

Groups 1 and 3 refer to cases where one of the gate inputs may present a 2 value. Within the three-valued calculus, a 2 represents the existence of either logic value 0 or logic value 1. Thus, we may introduce the definition of a floating cube as follows.

Definition 6.2: **floating cube**

> Given the three-valued calculus, a **floating cube** is an assignment of logic values, being at least one of such values equal to 2. The number of floating vectors underlying a floating cube is 2^m, where m is the number of positions presenting the value 2.

Indeed, a floating cube contains at least two floating vectors and thus can be viewed as a variation of the separate fall/rise delay model. In order to guarantee maximum accuracy, it is necessary to consider the maximum and the minimum delay values for a cube. By doing this, the delay of a cube has an invariant form, independently on the gate delay computation model used. To understand this, consider that the floating cube 022 is applied to the SCCG of figure 6.17, which logic function is $S = \overline{A + B \cdot C}$. Table 6.7 shows the equivalence between floating and transition modes for such gate. Note that this gate evaluates to 1 for floating vectors {000, 001, 010} and evaluates to 0 for floating vectors {011, 100, 101, 110, 111}. If the delay computation procedure has assumed the vector delay model, then the delay of this SCCG under cube 022 will present maximum and minimum values given by

max{d(000),d(001),d(010),d(011)} and min{d(000),d(001),d(010),d(011)}, respectively, where d(000), d(001), d(010) and d(011) are the delays of floating vectors 000, 001, 010 and 011. On the other hand, if the delay computation procedure has assumed the separate fall/rise delay model, the maximum and minimum values are given by max{tp_{LHmax} , tp_{HLmax}} and min{tp_{LHmin} , tp_{HLmin}}. However, for this SCCG, tp_{LHmax}=max{d(000),d(001),d(010)} and tp_{LHmin}=min{d(000),d(001),d(010)}, while tp_{HLmax}=tp_{HLmin}=d(011), which results the same maximum and minimum delay values as in the floating vector delay model case. By equivalent arguments, we may claim that the floating delay of this cube under the pair of delays model will result in the same maximum and minimum delay values computed for the other two gate delay models.

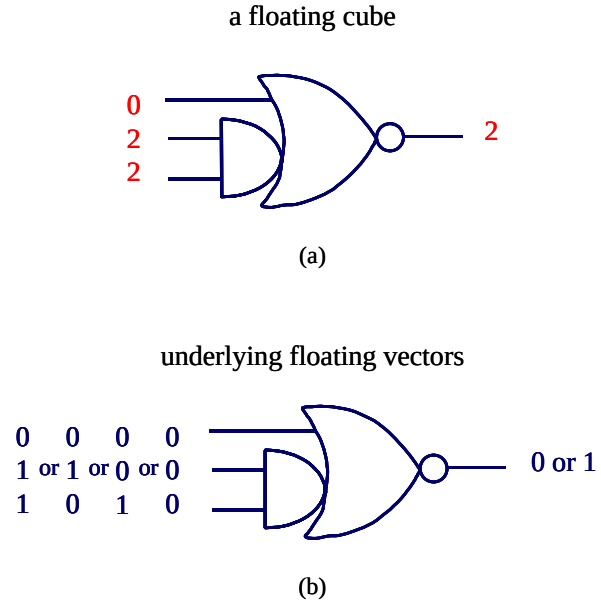


FIGURE 6.17 - Delay of a SCCG under a floating cube.

TABLE 6.7 - Equivalence between floating and transition modes for the SCCG of figure 6.17.

input vector	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
underlying pairs of vectors	000	001	010	011	100	101	110	111
	↓00	↓01	↓10	↓11	↑00	↑01	↑10	↑11
	0↓0	0↓1	0↑0	0↑1	1↓0	1↓1	1↑0	1↑1
	00↓	00↑	01↓	01↑	10↓	10↑	11↓	11↑
	↓↓0	↓↓1	↓↑0	↓↑1	↑↓0	↑↓1	↑↑0	↑↑1
	↓0↓	↓0↑	↓1↓	↓1↑	↑0↓	↑0↑	↑1↓	↑1↑
	0↓↓	0↓↑	0↑↓	0↑↑	1↓↓	1↓↑	1↑↓	1↑↑
	↓↓↓	↓↓↑	↓↑↓	↓↑↑	↑↓↓	↑↓↑	↑↑↓	↑↑↑
output logic value	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**

Analyzing again any equivalence table, it is possible to affirm that the delay of any gate depends only on the cube (or vector) applied to its inputs. Particularly in the case of the

separate fall/rise delay model, we can affirm that the delay of a gate may be perfectly defined by the group(s) to which the input floating vector (cube) belongs. This conclusion ratifies the assumption of splitting the timed-value computation of a gate into two independent subtasks:

- The computation of the gate output logic value and the first-order delay bounds
- The identification of the floating delay of the gate, used to calculate the actual delay bounds.

From this verification, it becomes clear that the modifications needed to allow computing the timed-value of arbitrary complex gates, including SCCGs, were already covered. These involve:

1. The function tree and the appropriate procedures for computing the gate output logic value and the first-order delay bounds;
2. The use of one of the three derived gate delay computation models (vector delay, pair of delays and single delay) to compute the lower and upper bounds on gate output delay.

6.3.4 Timed Backward Implication for SCCGs

The extended three-valued timed calculus detailed in the previous subsection allows for performing forward implication on complex gates. However, the timed-test generation procedure also has to perform backward implication during its execution. Thus, it is necessary to investigate how to deal with complex gates while performing backward implication.

Given a gate, logical backward implication consists of finding an input assignment with respect to this gate that sets its output to the desired logic value. Obviously, the desired output logic value is a known value, i.e., either a 0 or a 1. A conflict occurs when a given gate output that presents logic value lv is required to have logic value $\overline{lv}$.

In the case of timed backward implication, it is not sufficient to check for logical conflicts. It is also necessary to check whether the lower and upper bounds on the delay required to each input are within the already set lower and upper bounds for the given input. Given a simple gate and a desired logic value at the gate's output, the original timed-test generation procedure searches for the cube able to set the gate output to the desired logic value by exploiting the concept of controlling and noncontrolling values. Consider that a logic 0 is to be set at an AND gate. Since 0 is the controlling value of an AND gate, there may exist various possible vectors able to set the gate output to 0. Thus, the procedure searches for the fastest gate input able to be set to logic 0. If such input does not exist, then it is not possible to backward the logic 0 through the considered gate.

In the case of complex gates, the concept of controlling and noncontrolling values cannot be directly applied. However, we may realize that backward implying a logic value through a gate is in fact a kind of "guessing procedure", in which we are asked to find an assignment of input values that cause the desired output values. In this sense, it is obvious that any backward implication procedure able to treat complex gates is indeed a generalization of the simple gate case.

The basic heuristics used by the timed-test generation procedure for performing backward implication on simple gates relies on setting the smallest number of inputs because

such procedure results in a potentially smaller number of circuit lines that will have to be justified. Thus, it is necessary to perform a systematic exploration of the local Boolean subspace (i.e., the Boolean space associated to the logic function implemented by the gate under consideration), and at the same time, setting one input variable at a time.

The reasoning developed in the two last paragraphs suggests the possibility of using a systematic input space enumeration procedure along with the three-valued timed calculus presented in table 6.6 to accomplish the logical backward implication on complex gates. Figure 6.18 exemplifies the backward implication procedure when a logic 0 is desired at the output of the SCCG of figure 6.17. In order to do that, we begin by setting the topmost input to 0. Forward implication of this assignment results in a 2 at the SCCG output (figure 6.18a). This means that it is necessary to set one or more inputs. Assigning a 0 value to the middle input results in logic 1 at the SCCG's output (figure 6.18b), which is a conflict. Thus, we backtrack this assignment and replace de 0 by 1. The forward implication of this new assignment results in a 2 at the SCCG's output (figure 6.18c). Setting the bottom most input to 0 results in logic 1 at the output (figure 6.18d), which is also a conflict. Finally, by changing the bottom input to logic 1 results the desired logic 0 at the gate output.

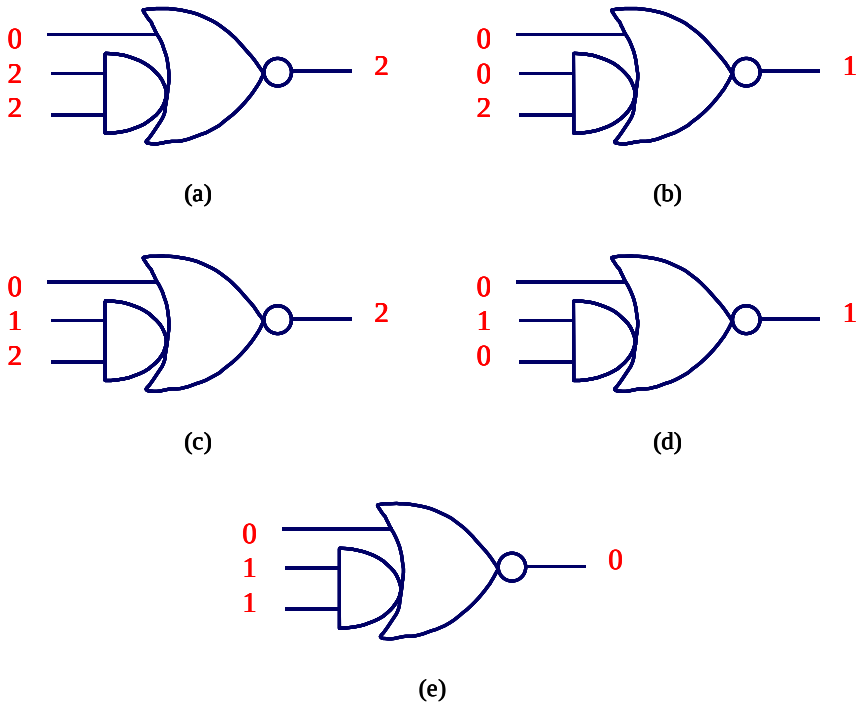


FIGURE 6.18 - Backward implication in a SCCG by using forward implication rules.

The previously described solution for backward implication of complex gates is equivalent to the PODEM algorithm applied to a single gate, but with significantly smaller time complexity, since we restrict the number of inputs for complex gates. However, the most important aspect on such procedure is that it may use the three-valued timed rules and function tree already developed. The only extra resource needed is an enumeration mechanism, for controlling the subspace exploration.

Having found a procedure for performing logical backward implication on complex gates, let us now move to the timed backward implication. Similarly to the timed forward implication, we will split the procedure into two steps:

1. Logical backward implication
2. First-order delay bounds computation and conflict detection.

The first step has already been described. The second one is composed of the following sub-steps, by the order:

1. Determine the gate delay under the selected input cube *k* (i.e., gate input assignment)
2. From the output lower and upper bounds, computes the first-order lower and upper bounds for each gate input by using the formulae: $l_o' = l_o - d(k_{max})$ and $u_o' = u_o - d(k_{min})$, where $d(k_{max})$ and $d(k_{min})$ are the maximum and minimum delays for cube *k* assigned to the gate inputs, respectively.
3. For each gate input i, compute the intersection between the input lower and upper delay bounds and the first-order lower and upper delay bounds computed in the previous step.

In step 3, a conflict is detected if $l_o' < l_i$ or if $u_o' < u_i$. Note that a gate input may provoke a conflict even if its logic value is 2.

Once timed forward and backward implication procedures able to handle complex gates have been developed, the timed-test generation procedure has been made sufficiently general to treat circuits containing complex gates.

It is important to notice that the proposed method to perform the three-value timed calculus on complex gates does not represent a significant increase in the overall execution time because macrocell generators are generally not allowed to generate complex gates with more than five serial transistors, due to technological reasons.

7 Conclusions

This thesis was concerned with the functional timing analysis (FTA) of combinational circuits, with emphasis to the applicability of FTA algorithms and models to circuits containing complex gates. Contributions were given to both FTA general theory and to FTA of circuits containing complex gates.

As a fundamental contribution, it was provided probably the deepest systematic study on timing analysis models and algorithms ever found in the literature. In order to facilitate the comprehension of the whole scenario, a new taxonomy for classifying FTA algorithms was proposed. According to this taxonomy, FTA algorithms may be classified with respect to the number of paths simultaneously handled in sensitization tests (single path sensitization, concurrent path sensitization or mixed approach), with respect to the method used to determine whether sensitization conditions are satisfiable or not (ATPG-based, SAT-based or other) and with respect to the sensitization criterion itself.

Still within the FTA theory, it was claimed that synchronous circuits implemented with current fabrication technologies might present an asynchronous behavior that cannot be properly captured by the transition delay model. It means that, in order to avoid underestimating the circuit delay, one would rather compute the delay under sequences of vectors than the delay under pairs of vectors. On the other hand, the floating delay model is the only one able to provide a safe upper bound on the circuit delay under sequences of vectors. As long as the floating delay model is the delay model underlying FTA, this is a strong argument to justify the use of FTA as the timing verification method.

Experimental results on path enumeration procedures presented in appendix 2 confirmed the severity of the so-called “path explosion problem” faced by the single path sensitization method. Due to this, concurrent path sensitization methods (either ATPG-based or SAT-based) represent the state-of-art in FTA algorithms.

Concerning FTA with complex gates, the TrueD-F algorithm proposed by Devadas et al. was chosen for serving as basis of an ATPG-based concurrent path sensitization technique that deals with circuits containing complex gates without requiring macro-expansion. The choice of the ATPG-based approach was motivated by the problems exhibited by the SAT-based approach when more realistic delay models are used. Another argument for choosing the ATPG-based approach is the existence of several acceleration techniques for ATPG algorithms.

The most important contribution of this thesis to FTA with complex gates concerns the extension of the timed-calculus to complex gates. Within the TrueD-F algorithm, the timed calculus consists of a set of rules used to compute logic values at gates and the related lower and upper bounds on delay. According to the proposed method, the computation of the timed-value of a complex gate may be accomplished by a recursive procedure that evaluates each sub-tree, beginning from the leaves. For a maximum of 4 serial transistors in both PMOS and NMOS networks, the maximal number of sub-trees that any SCCG may exhibit is 11, including the topmost one. Considering that the evaluation of a single sub-tree consists of a simple computation of delay intervals and logic values over the involved inputs, one can expect that operating directly on complex gates can lead to execution times that are at most equivalent to those necessary to perform the timed-value computation on an equivalent

macro-expanded network. However, it is worth to mention that applying an ATPG-based algorithm to an equivalent macro-expanded network may lead to significant overhead in execution time due to the potential increase in the number of circuit lines to be justified.

Other important contributions to the FTA with complex gates resulted from the investigation of the computational delay models applicable to the complex gates' case. The floating mode/transition mode equivalence table identifies the pairs of vectors underlying each floating vector. It also leads to the identification of the three valid gate delay computation models under the floating circuit delay model. These are the vector delay, the separate fall/rise delay and the single delay. Although being the most accurate, the vector delay model requires a delay value (or a pair of values) for each 2^n floating vectors, where n is the number of inputs of the gate. Considering SCCGs with no more than 4 serial transistors, the maximal n is 16, which gives a total of 2^{16}= 65536 floating vectors. On the other hand, the separate fall/rise delay model groups all floating vectors into two cases, thus reducing the delay values of a gate to two (or two pairs), while still assuring an upper bound on the gate delay.

The identification of the pairs of vectors underlying a given floating vector is of great interest when gate delay characterization is to be performed. In case of electrical simulation, only the vector pairs that may cause an output transition must be considered. In case of delay characterization by analytical formulation, this information may be used to adapt the delay modeling formulae in order to cover only the vector pairs of interest. However, identifying all pairs of vectors of interest for gates with many inputs constitutes a very hard task that must be further investigated. On the other hand, a very important conclusion that can be stated from the equivalence table is that the pin-to-pin delay format normally used in *standard-cell* characterization may result in an underestimation of gate delay since it takes into account only a part of all input floating vectors that may cause a transition at gate's output.

Finally, from the floating mode/transition mode equivalence table it was possible to introduce the concept of floating delay of a cube, as a generalization of the vector delay model.

Timed forward implication for complex gates is accomplished by using the procedure that computes the timed-value of gates. As already argued, the resulted time complexity is expected to be at most equivalent to that resulted from working on an equivalent macro-expanded network. Timed backward implication with complex gates, by its turn, cannot be directly derived from backward implication with simple gates because the concepts of controlling/non-controlling values are not applicable to complex gates. Thus, the proposed method relies on using the same procedure that computes the timed-value in forward implication, but in a PODEM-like fashion, by using input cube simulation on each gate. Input cube simulation presents a worst-case execution time that is proportional to the input space of the gate. For instance, the input space of a SCCG with 4 serial transistors in both networks has at most 2^{16}= 65536 floating vectors. Of course, this affects the execution time of the overall FTA algorithm.

This thesis showed that it is possible to perform ATPG-based concurrent path sensitization FTA of circuits containing complex gates without macro-expanding complex gates. The main advantage of such approach is the potential reduction in the number of circuit lines that must be justified, with comparison to the approach that uses macro-expansion. Nevertheless, the complexity of backward implication is significantly increased mainly when complex gates with many inputs are present in the circuit. This problem may be alleviated by the use of testability measures to guide the gate input space exploration.

Finally, the proposed extended timed calculus along with the complex gate delay valid models form a set of macro-modeling rules that make possible to perform “hierarchical floating delay computation”. This is a consequence of the fact that complex gates are the correct generalization of the smallest portion of any CMOS network.

7.1 Future Work

The performance of the proposed approach is greatly dependent on the ability of making right decisions. Such quality can be attained by making use of testability measures, as done in PODEM and FAN algorithms. Thus, the computation of testability in circuits with complex gates is a point to be addressed in short term.

The studies on valid gate delay computation models helped to fill the gap existing between physical and computation delay models. There are several points deserving attention, however. One of them is the (automatic) identification of the pairs of vectors responsible for transitions at gates’ outputs. Another one is the gate delay characterization itself. This latter one involves either electrical simulation or analytical characterization, but both respecting the valid gate delay floating models. A third one concerns the maximal number of inputs of complex gates, which directly influences the performance of the proposed ATPG-based FTA method. The number of inputs in complex gates is also responsible for preventing the use of the floating vector delay model, which is the most accurate. Limiting the number of serial transistors to 4 is a commonly used design strategy that assures reasonable electrical performance for SCCGs, but still leads to a total of 3503 possibilities of SCCGs. A very useful work consists on investigating the actual capabilities of state-of-art technology mapping algorithms in order to determine a practical limit on the number of inputs of SCCGs.

The proposed approach must be compared to the macro-expansion approach both in terms of performance and in accuracy, to certify that the reduction in the number of lines to be justified overcomes the increase in complexity of timed forward and backward implication procedures.

Investigating the accuracy and the performance of hierarchical floating delay computation is also a very important work. It seems to be clear that the acceleration obtained by the hierarchical mode may allow FTA of circuits with at least one or two orders of magnitude. However, it is necessary to investigate the impact of the gate delay computation model on the resulted accuracy.

Last, but also quite important, is a broad comparison between the SAT-based approach and the proposed one, which is an ATPG-based approach. One basic difficulty relies on selecting a set of delay computation models that are valid on both cases and at the same time are able to provide delay estimates with acceptable accuracy. Another difficulty relies on choosing an existing SAT-based tool (or algorithm) able to deal with complex gates.

Appendix 1 The Need for Functional Timing Analysis: a Case Study

Concerning delay computation techniques and algorithms, a question that frequently arises is whether it is worth to perform FTA, since the computational effort is significantly greater than in TTA case. Obviously, this would be quite simple to answer if one could always assure to have total control over the design under development. However, due to the increasing complexity of current VLSI designs and to the use of EDA tools, it is not possible for the designer to know all details of each design. Thus, since TTA and FTA may provide different delay estimates for the same circuit, it is important to measure how severe this difference can be and what would be its impact in the maximal operating frequency estimation.

There are some classes of combinational circuits that have been extensively used to illustrate the effect of disregarding the influence of path sensitization in delay estimation. **Carry-skip adders** (csas) constitute one such class for it is well known they may contain hundreds of thousands of false paths [LAM94][DEV94]. Although in the case of csas the difference between the delay estimated by TTA and the delay estimated by FTA may represent an extreme case, it is still necessary to consider that for automatically generated designs this difference may also be significant.

In order to evaluate the difference between TTA and FTA estimations originated from considering or not path sensitizability, a discussion on delay estimation of csas is carry out in the sequel. To be more realistic, a non-unitary component delay model is used [DAG96].

Figure A1.1 reproduces the 2-bit csa showed in figure 2.4. This csa2 has been mapped using simple CMOS gates (in this case, 2-input nands and inverters). Assume a fixed gate delay computation model with separate falling and rising delays. In order to construct higher order adders, csa2 stages are connected together through the carry in/carry out pins, as shown in figure A1.2. Thus, a practical approach for estimating the delay of higher order csas is first characterize a single csa2 stage and then use the results of this characterization to infer the whole circuit delay.

If TTA is the chosen method, then the circuit delay is found by identifying the longest topological path, which is clearly the carry chain.

However, if FTA is the computation method to be used, then it will be necessary to identify the longest path able to propagate a transition. This means that path sensitizability must be considered. The fact that the only connection between two adjacent stages in higher order csas is through the carry chain suggests the classification of stages into three groups [GÜN99a]:

- first stage
- intermediate stages
- last stage

For each of these three cases an *ad hoc* sensitization analysis is then performed. It is also possible to take advantage on the regularity exhibited by csas: all analyses may be performed over the same csa2 instance. In order to facilitate the analyses, the transition mode will be

assumed (see section 2.3). The error of such assumptions may not be significant in the specific case of csas.

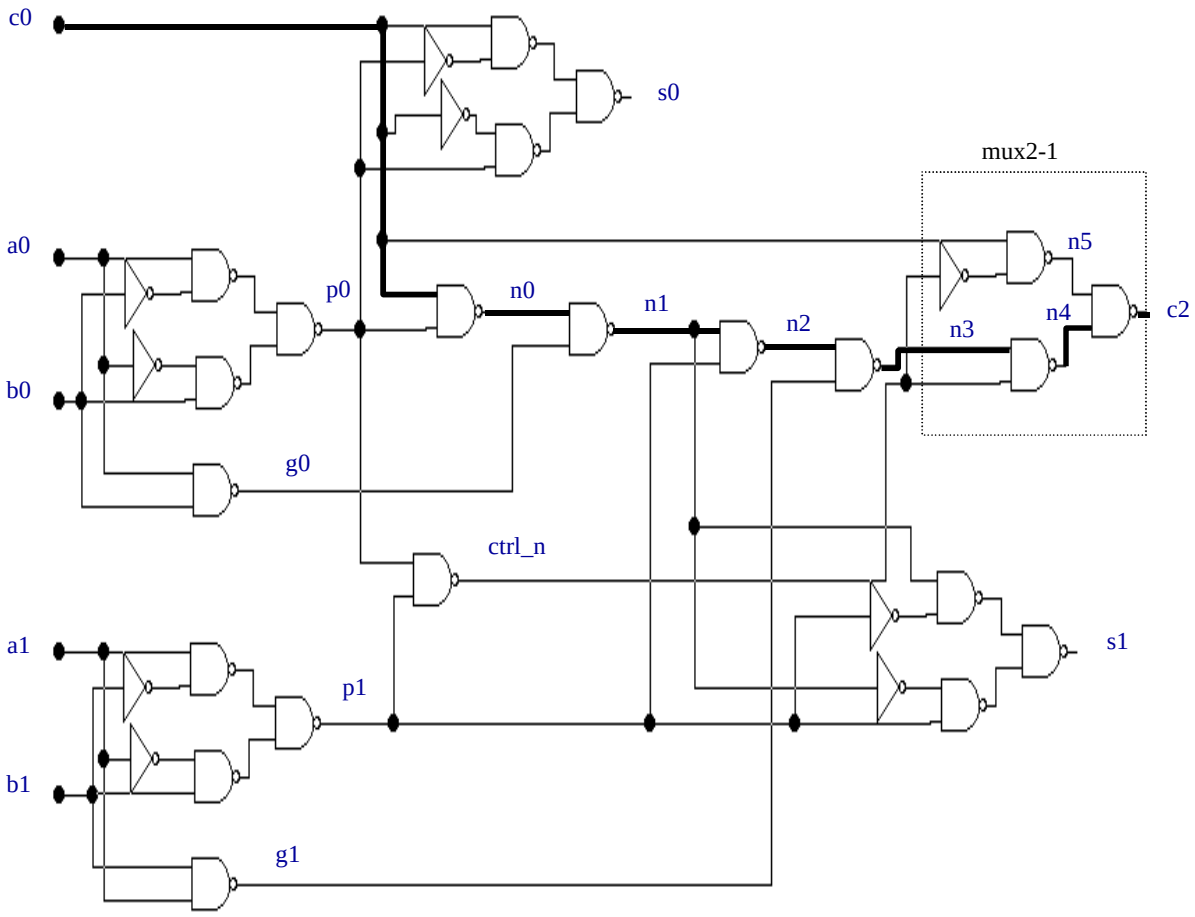


FIGURE A1.1 - A 2-bit carry-skip adder (csa2) mapped using 2-input nands and inverters.

The target of the sensitization analysis of the first stage is to find the pair of longest sensitizable paths beginning at any primary input and ending at output c2, being one responsible for a falling transition at c2 and the other responsible for the rising transition at c2.

For an intermediate stage, the target is to find the pair of longest sensitizable paths beginning at c0 and ending at c2, with one of the paths beginning by a falling transition (at c0) and the other beginning by a rising transition at (c0).

Finally, the analysis of the last stage is to find the pair of longest sensitizable paths beginning at c0 and ending at any output, where one of the paths begins with a falling transition (at c0) and the other begins with a rising transition at (c0).

The longest sensitizable path is then determined by summing the delay of all stages, taking into account signal polarity. Note that the described method allows for performing an approximate hierarchical FTA without enumerating circuit paths.

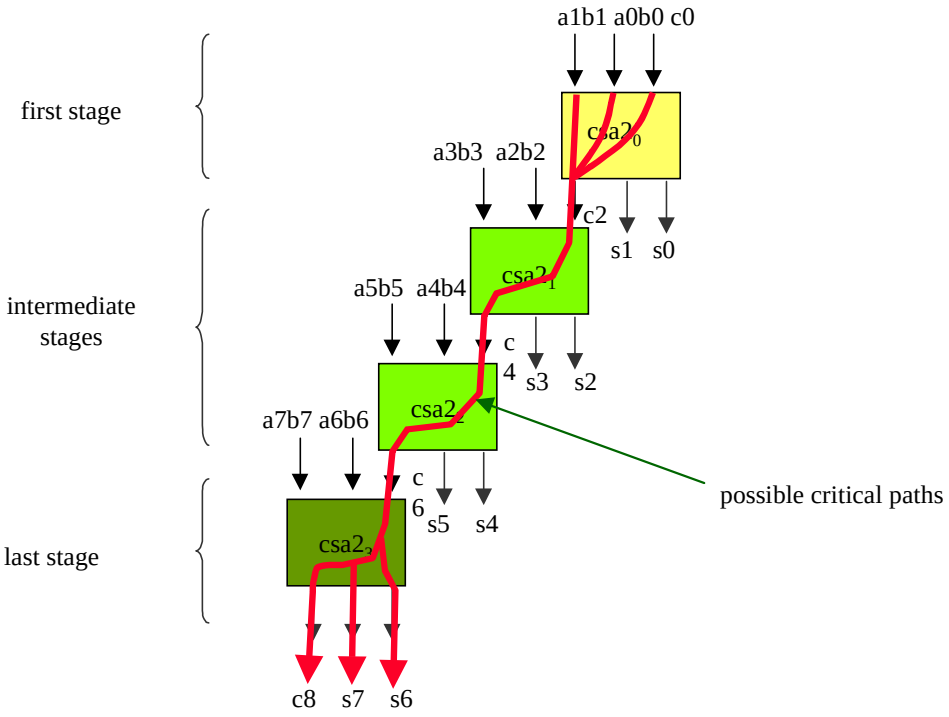


FIGURE A1.2- An 8-bit carry-skip adder made of csa2 stages.

Consider an intermediate csa2 stage. There are only two topological paths between c0 and c2. Each topological path may be divided into two **logical path**s, being one associated to the falling transition (at its input) and the other associated to the rising transition (at its input). Thus, for an intermediate stage, the logical paths of interest are P0↑=(↑c0, n0, n1, n2, n3, n4, c2), P0↓=(↓c0, n0, n1, n2, n3, n4, c2), P1↑=(↑c0, n5, c2) and P1↓=(↓c0, n5, c2), which delays are 1204ps, 1134ps, 546ps and 482ps, respectively.

The *ad hoc* analysis will try to determine whether P0↑ is sensitizable or not by checking all possible conditions that may activate it. If it is not possible to find a set of signal conditions that activates P0↑, then P1↑ will be analyzed. The search of a sensitizable logical path beginning with a falling transition will also follow the path delay order: P0↓ and then P1↓. To reduce the search space, the check begins by setting stable logic values (either 0 or 1) to circuit nodes whenever possible. Otherwise, 0→1 and 1→0 transitions will have to be considered, along with the required stable times. Of course, the second possibility increases the analysis complexity, and thus should be avoided.

Figure A1.3 shows a set of signal conditions for trying to sensitize P0↑: inputs a0 and b0 have been assigned to static values in such a manner that both p0 and g0 have a static 1. This allows a transition to propagate from c0 to n0 and from n0 to n1. However, given the previous conditions, it is not possible to assign static values to g1, p1, ctrl_n and n5 in order to allow a transition to propagate from n2 to c2. Thus, it will be necessary to consider signal transitions at g1, p1, ctrl_n and n5, along with the required stable times.[9] In figure A1.3, these delay-dependent conditions are expressed in terms of inequations, annotated at the respective signals. By assembling appropriate inequation systems it is possible to determine whether a sensitization condition exists or not for the path P0↑:

[9] Note that the analysis of any path in the intermediate stages (and also for paths in the last stage) must consider a possibly non-zero arrival time of a transition applied to c0, to model the effect of previous csa2 stages.

$$\left.\begin{array}{l} \downarrow p1 > 399 + \Delta \\ \downarrow p1 + 285 < 662 + \Delta \end{array}\right\} \quad \text{(A1.1)}$$

and

$$\left.\begin{array}{l} \uparrow p1 < 399 + \Delta \\ \uparrow p1 + 212 > 662 + \Delta \end{array}\right\} \quad \text{(A1.2)}$$

By evaluating inequations A1.1 and inequations A1.2 one conclude that it is not possible to find an arriving time value at input c0 (Δ) that combined with either a falling or a rising transition at node p1 (↓p1 or ↑p1) could result in a valid sensitization condition. None of the other possible signal assignments resulted in valid sensitization conditions. Thus, the conclusion is that path P0↑ is not sensitizable.

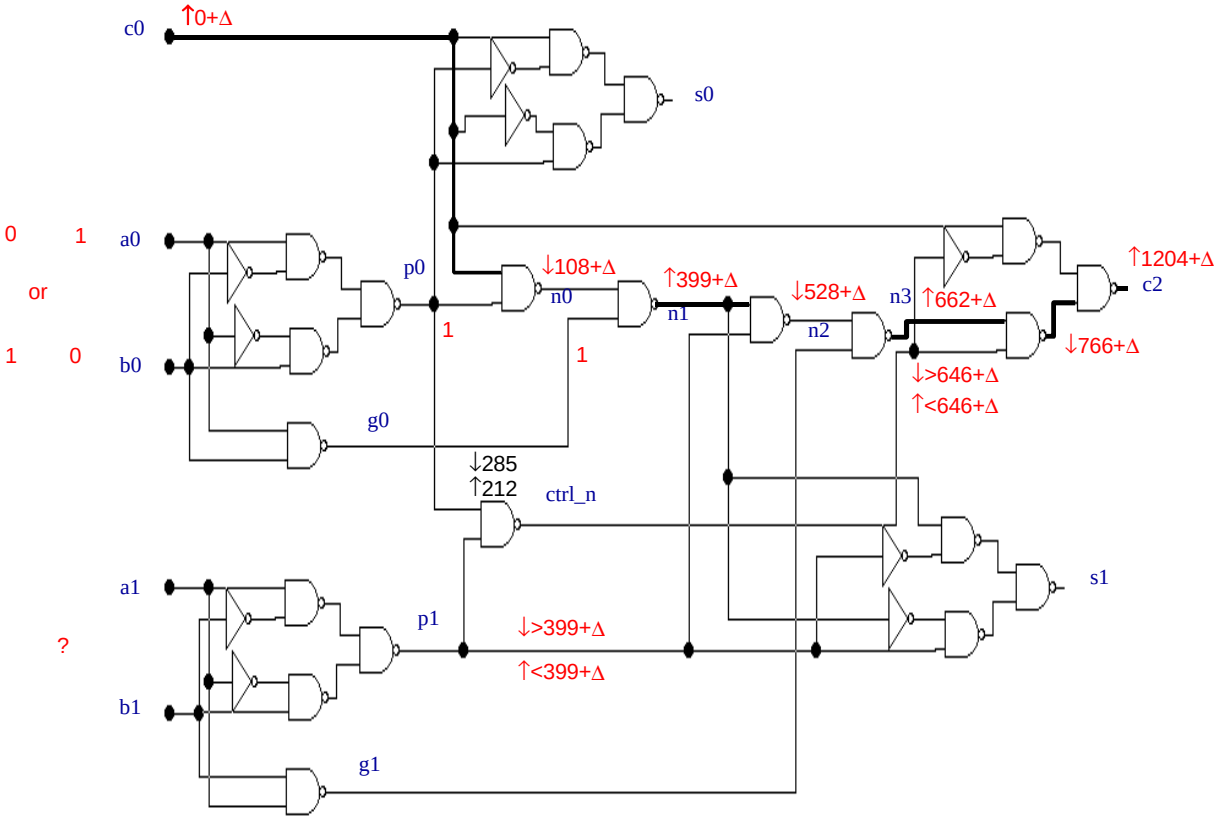


FIGURE A1.3 - A set of signal conditions for trying to sensitize P0↑.

Similar analyses on the other paths lead us to the conclusion that P1↑ and P1↓ are the sensitizable paths **for any** intermediate csa2 stage.

Applying the same procedure to the first csa2 stage we find that the longest sensitizable paths generating a falling transition at c2 are P2↓=(↓a0, 4-X0, p0, n0, n1, n2, n3, n4, c2) and P3↓=(↓b0, 3-X0, p0, n0, n1, n2, n3, n4, c2), both with delay 1559ps. And the longest sensitizable paths generating a rising transition at c2 are P4↓=(↓a0, 1-X10, 3-X0, p0, n0, n1, n2, n3, n4, c2), P5↓=(↓b0, 2-X0, 4-X0, p0, n0, n1, n2, n3, n4, c2), P6↓=(↓a1, 1-X3, 3-X3, p1, ctrl_n, sel-X11, n5, c2) and P7↓=(↓b1, 2-X3, 4-X3, p1, ctrl_n, sel-X11, n5, c2), all of them with delay 1435ps.

Finally, for the last csa2 stage, the longest sensitizable paths are P8↓=(↓c0, n0, n1, 2-X5, 4-X5, s1) and P9↑=(↑c0, n0, n1, 2-X5, 4-X5, s1), with delays of 1074ps and 998ps, respectively.

The delay values obtained from the sensitization analysis should be combined to derive general formulae to approximate the delay of csa adders made up from the analyzed csa2 stage. The formulae will give approximate values because they do not account for the modifications on gate delays arriving from the new fanin/fanout relations when csa2 stages are connected together. The equations that approximate the (output) falling and the rising critical delays of csa adders are:

$$td_{max}\downarrow = 546\times\left(\frac{m}{2}-2\right)+2433 \quad [ps] \qquad (A1.3)$$

$$td_{max}\uparrow = 482\times\left(\frac{m}{2}-2\right)+2633 \quad [ps] \qquad (A1.4)$$

where $m \in (4,8,16,..)$, corresponding to 4-bit, 8-bit, 16-bit csa adders. To estimate the critical delay of a given adder (csa4, csa8 etc), equations A1.3 and A1.4 must be evaluated. The critical delay will be given by $\max(td_{max}\downarrow, td_{max}\uparrow)$.

And the equations that approximate the (output) falling and the rising topological delays are:

$$td_{max}\downarrow = 1134\times\left(\frac{m}{2}-1\right)+1605 \quad [ps] \qquad (A1.5)$$

$$td_{max}\uparrow = 1204\times\left(\frac{m}{2}-1\right)+1766 \quad [ps] \qquad (A1.6)$$

Similarly, the topological delay of a given adder is given by $\max(td_{max}\downarrow, td_{max}\uparrow)$, where $td_{max}\downarrow$ and $td_{max}\uparrow$ are obtained through equations A1.5 and A1.6. The estimations for 2, 4, 8, 16, 32 and 64-bit adders are shown in table A1.1.

TABLE A1.1 - Critical delay estimations for csa adders.

adder	**topological delay (ps)** (1)	**critical delay (ps)** (2)	(1)/(2)
csa2	1766	1560	1.13
csa4.2	2970	2633	1.12
csa8.2	5378	3597	1.49
csa16.2	10194	5709	1.78
csa32.2	19826	10077	1.96
csa64.2	39090	18813	2.07

The results of table A1.1 show that in case of adders built up from the analyzed csa2 the difference between the topological delay and the critical delay (i.e., the delay obtained by testing path sensitizability) may be significant. In the case of csa64.2, for instance, the topological analysis has overestimated the delay by 100%. This means that the clock frequency obtained through a topological analysis could still be speeded up by a factor of 2!

The previous results illustrate the importance of considering path sensitization in timing analysis. And although the carry-skip adder case would represent a very particular class of circuits, there are other important classes of circuits exhibiting significant difference between the topological delay and the (actual) critical delay.

Appendix 2 Gate Delay Computation Models and the Complexity of Best-First Search Procedures

The crudest gate delay computation model that can be used in a FTA tool is the single fixed delay that assigns one delay value per gate. The target of the following discussions is investigating the computational effort necessary to use a pair of delays per gate within best-first search path enumeration procedures. Throughout this discussion, the referred gate delay models were renamed to *single gate delay* (**sgd**), which assumes only one delay per gate, and *single pair gate delay* (**spgd**), which considers a pair of delays per gate, being one for the falling (output) transition and another for the rising (output) transition [GÜN98a][GÜN98b].

In order to investigate the relevance of using the **spgd** method in path enumeration (instead of the **sgd** method), it was initially considered the problem of finding the longest (topological) path of the CMOS gate chains listed in table A2.1. These chains are either single-type (i.e., containing only one type of CMOS gate) or multiple-type (i.e., containing more than one type of CMOS gate). All chains have 20 gates. Note that in the sgd case, the delay of a path is computed simply by adding the delay of each gate along the path, according to equation 5.1. However, in the spgd, each topological path originates two logical paths, possibly with distinct delays calculated by equations 5.5 and 5.6.

TABLE A2.1 - List of CMOS gate chains used to evaluate sgd and spgd path delay calculations.

chain	description
p_nand2	2-input nand chain
p_nand3	3-input nand chain
p_nand4	4-input nand chain
p_nor2	2-input nor chain
p_nor3	3-input nor chain
p_nor4	4-input nor chain
p_nand3-4	chain with 3 and 4-input nands
p_nor3-4	chain with 3 and 4-input nors
p_n3n4i	chain with inverters, 3 and 4-input nands and nors
p_n3n4	3 and 4-input nands and nors

For estimating individual gate delays, a linear delay model was used. This model is associated to the cells of a sea-of-gates library, developed in standard 0.8μm CMOS technology. Although the linear delay model is not very accurate, it serves for the purpose of comparing the two methods under evaluation.

Considering that the adopted component delay model is able to furnish a pair of delays per gate, the **spgd** method was used as reference to evaluate the loss of accuracy resulted from using the **sgd** method. To derive the delay of each gate v for the **sgd** case, three possibilities were considered [GÜN98]: $td(v)=\max\{tdlh(v),tdhl(v)\}$, $td(v)=(tdlh(v)+tdhl(v))/2$ and

td(v)=min{tdlh(*v*),tdhl(*v*)}, which will be referred as **max**, **typical** and **min** cases, respectively. Figure A2.1 shows the percent error resulted from using the **sgd** method (instead of the spgd method) for the CMOS gate chains.

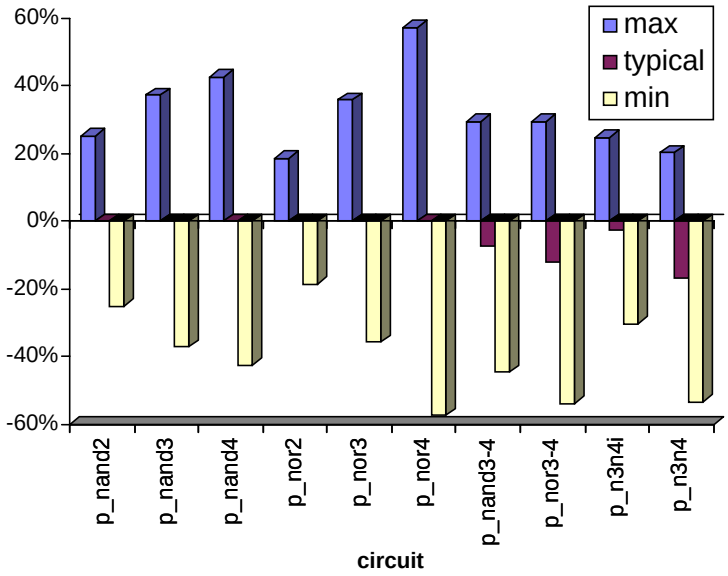


FIGURE A2.1 - Percent error resulted from using the sgd method.

For all chains, **max** case overestimated the critical path delay up to 57%, while **min** case underestimated the critical path delay up to 57%. **Typ** case has presented no error for the first 6 cases (from p_nand2 to p_nor4), which are single-type gate chains. For the remaining 4 circuits, which are multiple-type gate paths, the error of **typ** case was within the range 3%-17%. Since in real-life digital designs it is not practical to avoid multiple-type gate paths, we cannot disregard the fact that typ case may underestimate the critical path delay.

Another aspect of relevance in the performance of best-first search path enumeration is the type of data structures used to store partial paths. In the best-first procedure of Yen et al. [YEN89] k-list was originally described to be a static linear list used to store both partial and complete paths. At the beginning of the procedure it was initialized with partial paths, sorted by their esperances. After all partial paths had been extended k-list holds the k-most critical paths of the circuit. Thus, the procedure controls the total amount of memory to be allocated by allowing only the first k paths to be traced and by using a static data structure. However, in path-based FTA, i.e. single path sensitization algorithms, it is not possible to precise beforehand how many paths must be traced until the first sensitizable one is found. In case the original procedure is used, the absence of a sensitizable path within the k-most critical paths will demand another run with a greater k. For circuits containing hundreds of thousands of long unsensitizable paths, the need for re-running the enumeration procedure many times may render FTA impracticable.

In order to allow the procedure to stop by other criteria (e.g., when the first sensitizable path is found), the data structure used in the modified best-first search procedure of subsection 5.2.2 was mostly modified [PIN98][GÜN99]. For storing partial paths, two options of data structures were made available: a **dynamic linear list** and a **binary tree**. During the whole process of path enumeration partial paths are kept ordered by using the esperance value, as in the original version of [YEN89]. In the binary tree case, a heapsort procedure [COR90] is used to maintain the order. Hence, the main feature of this “heapified” binary tree is that the partial path having the greatest esperance is stored on its root. From now on, the two versions of partial path lists are going to be referred to as **linear** and **tree**.

A second (linear) list called **list_of_paths** is used to store complete paths. This list does not require any special insertion scheme since the best-first procedure assures that paths are traced in an ordered manner. Figure A2.2 depicts the possible data structures for storing partial paths, along with the structure that stores complete paths.

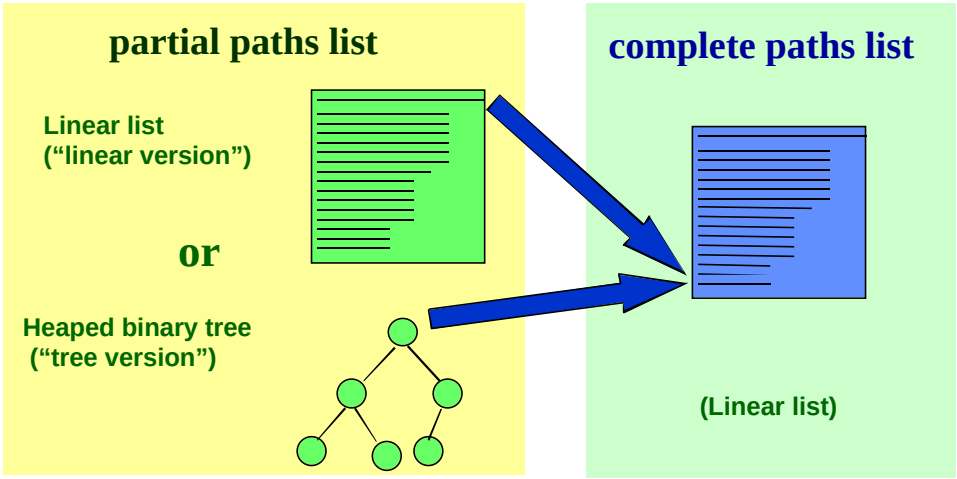


FIGURE A2.2 - Data structures of the best-first procedure for path enumeration.

These later data structures gave rise to two versions of the best-first procedure, which, by they turn, could use either the sgd or the spgd path delay calculations [GÜN99]. Thus, four versions of best-first enumeration procedure were available, as summarized in table A2.2.

TABLE A2.2 - Versions of the best-first procedure for path enumeration.

version	**path delay calculation method**	**type of list for storing**	
		complete paths	**partial paths**
spg-linear	spd	dynamic linear	dynamic linear
spgd-linear	spgd	dynamic linear	dynamic linear
spg-tree	spd	dynamic linear	binary tree
spgd-tree	spgd	dynamic linear	binary tree

The described versions of the best-first search based path enumeration algorithm were implemented using C language.

To evaluate the performance of the best-first search based path enumeration algorithms the circuits of the ISCAS'85 suite were used. These circuits were mapped using only simple CMOS static gates (neither transmission gates nor CMOS complex gates were used). Table A2.3 shows the complexity of the DAGs that represent these circuits.

TABLE A2.3 - Complexity of the ISCAS'85 benchmark circuits.

circuit	# of gates	# of nets	# of graph nodes	# of graph edges
c432	182	222	231	448
c499	364	405	439	883
c880	529	589	617	987
c1355	604	645	679	1227
c1908	955	988	1015	1656
c2670	1605	1762	1828	2773
c3540	2307	2357	2381	3671
c5315	3249	3427	3552	5752
c6288	2672	2704	2738	5152
c7552	4556	4762	4871	7608

The evaluation of the execution time was accomplished by running the algorithms on a Sun Ultra2™ workstation with 256 Mbytes of RAM memory and 586 Mbytes of swap memory for several values of **k** (number of paths traced). Execution times for linear version with k=10000 are showed in figure A2.3. From these results one can conclude that the execution time of the linear version is greater than that of the tree version. Furthermore, this difference is significant for the most complex circuits. It is also possible to observe that for the majority of the circuits the execution time of the spgd case is greater than the execution time of the sgd case (for both linear and tree versions). This result was expected since the number of partial path insertions tends to be greater in the spgd case. However, the difference is significant only for circuit c880 and c1355 (3.2 seconds and 7 seconds, respectively). For all other cases, the spgd-based procedure is at most 1 second slower.

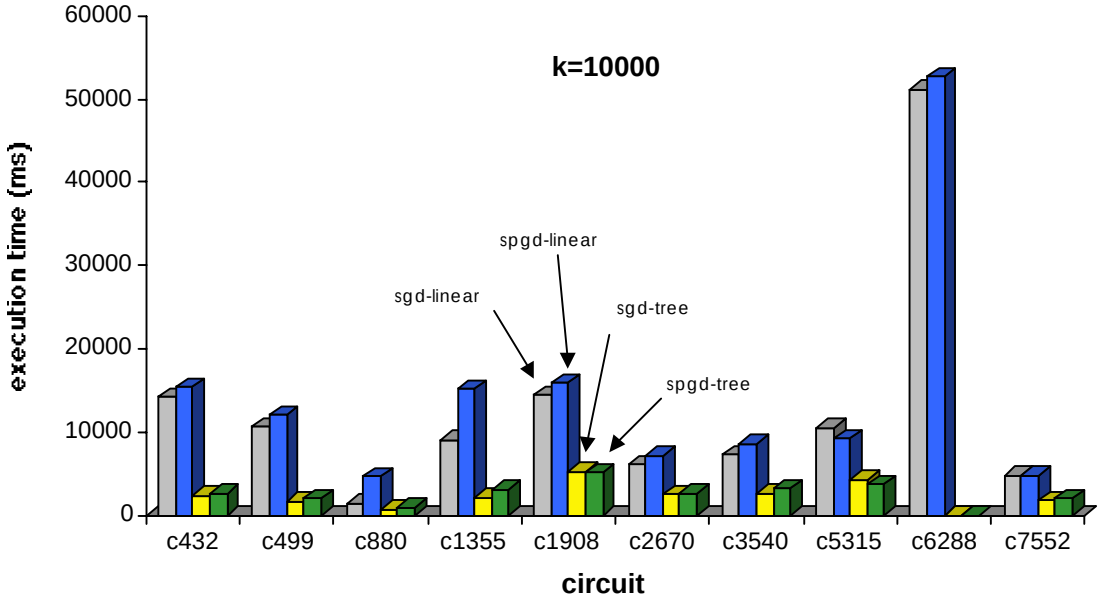


FIGURE A2.3 - Execution times for k=10000 paths.

The time complexity of the best-first procedure is basically dominated by inserting partial paths in the partial path list. Thus, reducing the partial path insertion time may significantly reduce the overall execution time. Indeed, the curves *execution time* x *number of traced paths* showed by figure A2.4 confirm this supposition [PIN98][GÜN99]. It can be seen

that the time complexity of linear version is exponential with respect to **k**, while the time complexity of tree version is almost linear. As a result, the execution time overhead of the spgd method is greatly reduced by the use of the "heaped" binary tree.

Figure A2.5 shows the curves *memory used* x *number of traced paths* for the same circuits of figure A2.3. It can be seen that in terms of memory use the behavior is exactly the opposite: the linear version presents linear dependency on the number of traced paths **k**, while the tree version presents exponential dependency.

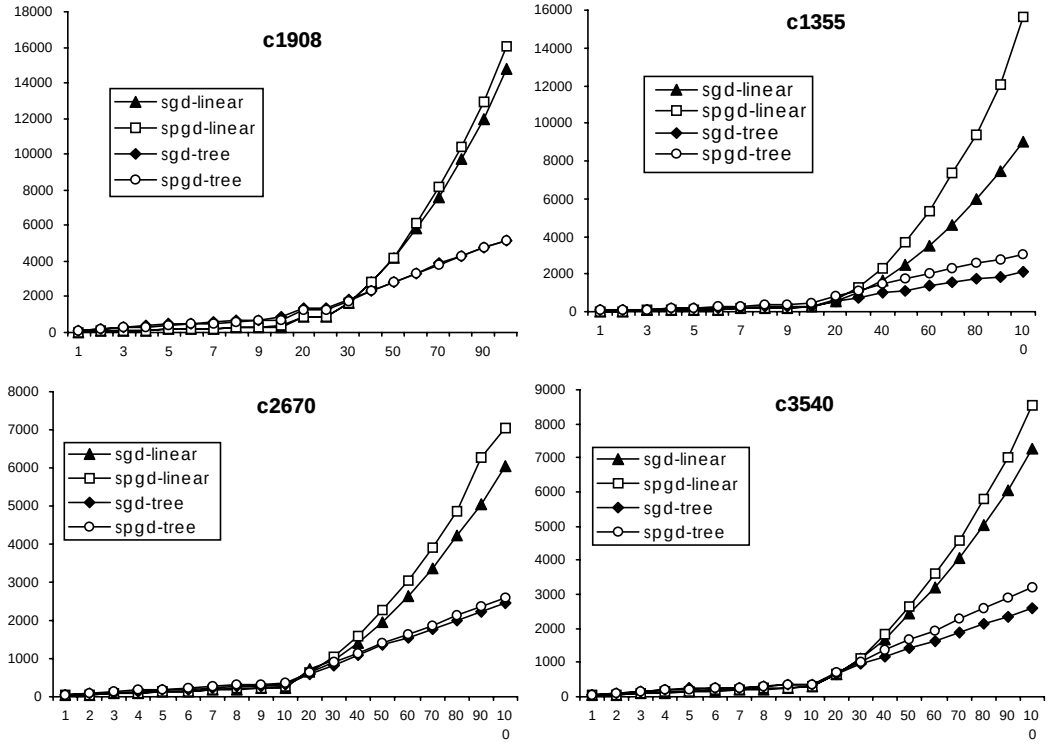


FIGURE A2.4 - Curves *execution time(ms) x number of traced paths* for some ISCAS85 circuits.

In a timing analysis tool that checks sensitizability, the tree version would be preferred because it is not possible to know *a priori* how many paths will be traced to find a sensitizable one. Thus, the use of a linear version (solely) would result in excessively long execution times. On the other hand, the tree version would not finish executing for circuits with too many false paths, as the carry-skip adders studied in Appendix 1. The solution for this can be the integration of both algorithms in a timing analysis tool, such that the user can choose the algorithm she/he wants to use. Another possibility is to perform hierarchical timing analysis. But even in this case, enumeration-based timing analysis tools would have embedded one or both discussed algorithms.

Let us now turn our attention to the applicability of the explicit enumeration method for performing FTA of a real-life case. Consider again the carry-skip adders discussed in Appendix 1. As it was mentioned, the only connection between two adjacent csa stages is through logical paths P0↑, P0↓, P1↑, or P1↓. It was shown that both logical paths P0↑ and P0↓ are not sensitizable for the assumed gate delay assignment. Therefore, one could think of logical paths P0↑ and P0↓ as being *blocking trunks* for any signal propagation through adjacent csa stages and use this information to detect false paths in an explicit enumeration loop [GÜN99a]. In order to do this, the best-first search procedures previously described (spgd-linear and spgd-tree) were adapted: each time a new path is completely extended, a

checking procedure searches for the presence of any *blocking logical trunk*, which are previously furnished by the user, through an ASCII file. As long as paths are enumerated in a non-increasing order of delays, the first path not containing any *blocking trunk* is assumed to be the critical path of the circuit.

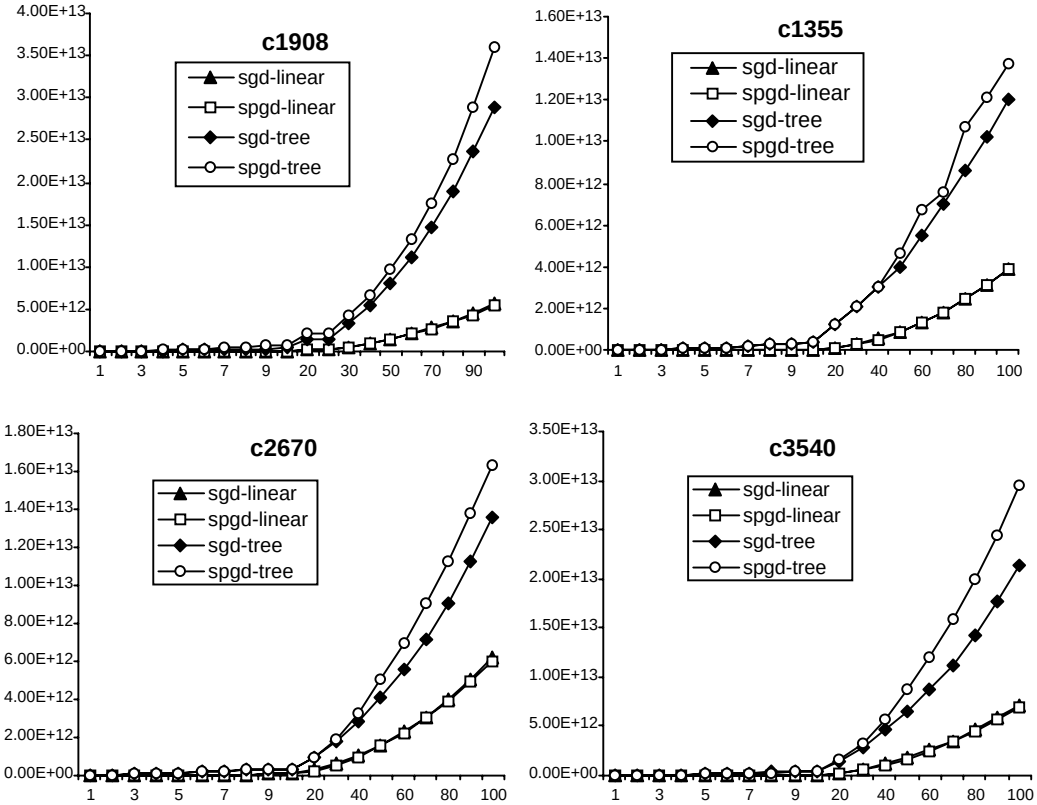


FIGURE A3.5 - Curves *memory used (bytes) x number of traced paths* for some ISCAS85 circuits.

Table A2.4 shows the approximate execution times for the semi-automatic unsensitizable path detection procedure for the carry-skip adders of Appendix 1. It has run on an Ultra10 SparcStation with 512Mbytes of physical memory. It is also showed the computed critical delays and the number of traced paths until the first sensitizable path was found and the approximate running time. The computed critical delays may differ from the calculated ones (also in Appendix 1) because in the semi-automatic procedure the fanout loads resulted from connecting two adjacent csa stages are considered in detail by the physical delay modeler.

The execution times confirm the limitation of explicit path enumeration procedures running in flat mode: the two most complex circuits did not complete due to the large number of unsensitizable long paths. In the case of csa32.2, the enumeration tool has traced more than 300 thousand paths without finding any sensitizable one!

TABLE A2.4 - Execution time for the semi-automatic unsensitizable path detection procedure.

adder	**topological delay (ps)** (1)	**computed delay (ps)** (2)	**# of traced paths**	(1) /(2)	**execution time (ms)****
csa2	1766	1560	7	1.13	30
csa4.2	2970	2652	16	1.12	70
csa8.2	5378	3654	786	1.47	680
csa16.2	10194	5658	40190	1.80	157000
csa32.2	19826	*	-	-	-
csa64.2	39090	*	-	-	-

* Not finished after 24 hours.
** Execution times for the spgd-linear procedure.

One advantage of the presented semi-automatic method is that it reduces the effort needed to perform the *ad hoc* sensitization analysis, since only unsensitizable trunks must be found. Another advantage is the better accuracy obtained in delay calculation, when compared to the calculation method proposed in section 3.4, since actual fanin and fanout are considered in the delay calculation phase. As a counterpart, the method suffers from the *path explosion problem* [MCG93], since the explicit path enumeration tool runs only on flat circuit descriptions. It is interesting to remark that the spgd-tree version has presented memory allocation problems that prevented its use for the largest circuits. On the other hand, the spgd-linear version has not presented memory problems, but execution time problems, since it could not finish executing after 24 hours.

Appendix 3 Análise de *Timing* Funcional de Circuitos VLSI Contendo Portas Complexas

Este anexo apresenta um resumo estendido em português desta tese.

A **verificação de *timing*** tem como objetivo determinar se as restrições temporais impostas ao projeto podem ser satisfeitas ou não. De modo mais objetivo, a verificação de *timing* está relacionada com a estimativa do **atraso crítico** dos circuitos e com a **máxima freqüência de operação**. A precisão da verificação de *timing* depende da precisão dos modelos adotados. Por modelos entende-se não apenas os modelos físicos de atraso usados para quantificar o atraso de cada componente do circuito, mas também os modelos computacionais para os atrasos dos componentes e do circuito como um todo. Estes últimos estão relacionados ao modelo de operação do circuito, isto é, se o circuito opera no modo síncrono ou no modo assíncrono.

A maioria das técnicas de verificação de *timing* são orientadas a circuitos seqüenciais síncronos. Desta forma, consideremos os aspectos mais importantes ao se estimar a máxima freqüência de operação de um circuito seqüencial que pode ser representado pelo modelo de Mealy. O modelo de Mealy, mostrado na figura A3.1, divide o bloco combinacional em dois sub-blocos distintos: a lógica de próximo estado e a lógica de saída [GAJ99]. Considerando-se que os elementos de armazenamento são *flip-flops* disparados pela borda, então, a cada borda ativa do relógio, o próximo estado é carregado nos *flip-flops*, tornando-se estado atual. Neste momento, o novo próximo estado começa a ser computado pela lógica de próximo estado. As saídas podem mudar como conseqüência de uma mudança no estado atual ou como conseqüência de uma mudança nas entradas ou como conseqüência de ambos.

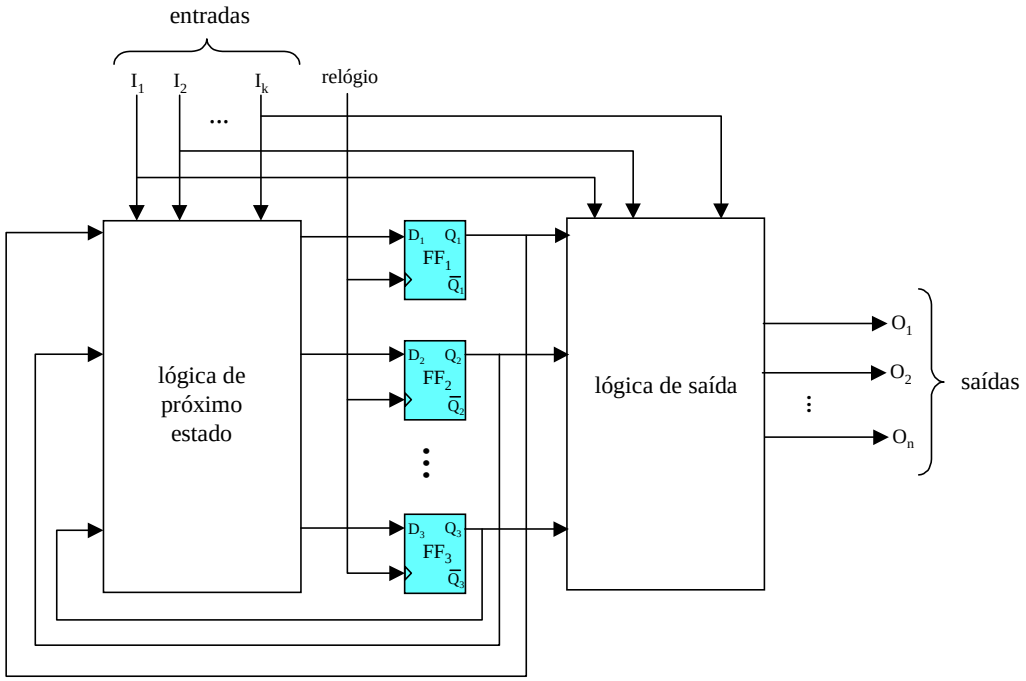


FIGURA A3.1 - Modelo de circuito seqüencial síncrono.

Considere que a lógica de próximo estado possui atraso de propagação máximo e mínimo $\mathbf{T_{next}}$ e $\mathbf{t_{next}}$, respectivamente, e que a lógica de saída possui atraso de propagação

máximo e mínimo $\mathbf{T_{out}}$ e $\mathbf{t_{out}}$, respectivamente. Considere também que os *flip-flops* apresentam máximo atraso de propagação igual a $\mathbf{T_{ff}}$, tempo de preparação (*setup time*) igual a $\mathbf{t_s}$ e tempo de manutenção (*hold time*) igual a $\mathbf{t_h}$. Então, para que o circuito opere corretamente, deve-se assegurar as seguintes condições:

- $\tau > \max\{ (\mathbf{T_{ff}} + \mathbf{T_{next}} + \mathbf{t_s}) , (\mathbf{T_{ff}} + \mathbf{T_{out}}) \}$, onde τ é o período do relógio
- $\mathbf{t_{next}} > \mathbf{t_h}$
- as saídas do circuito devem estar estáveis e válidas durante um tempo maior do que $\mathbf{T_{next}} + \mathbf{t_s}$ antes de cada borda ativa do relógio.

As condições anteriores são bastante conservadoras, porém asseguram uma operação síncrona correta. A primeira condição assegura que o período do relógio seja longo o suficiente para acomodar o pior caso em termos de atraso do laço de realimentação do próximo estado ($\mathbf{T_{ff}} + \mathbf{T_{next}} + \mathbf{t_s}$) e o pior caso de atraso para a lógica de saída ($\mathbf{T_{ff}} + \mathbf{T_{out}}$). A segunda condição assegura que o período do relógio é longo o suficiente para que os *flip-flops* possam amostrar um novo estado. A terceira condição assegura que os sinais de entrada da lógica de próximo estado sejam computados a tempo, de modo que todas as saídas deste bloco estejam estáveis e válidas por um tempo maior ou igual a $\mathbf{t_s}$ antes da próxima borda ativa do relógio.

Considere que se adote o modelo de operação descrito anteriormente e que as entradas sejam sincronizadas por *flip-flops*. Se não houver variação significativa nos tempos de propagação dos *flip-flops*, pode-se assumir que cada bloco combinacional opera de maneira completamente síncrona, de modo que o atraso de propagação é conseqüência da aplicação de dois vetores de entrada consecutivos. Entretanto, caso os tempos de propagação dos *flip-flops* sejam significativamente diferentes, os blocos combinacionais operarão de maneira assíncrona, como se seqüências rápidas de vetores fossem aplicadas às suas entradas.

Dentre as técnicas existentes para a estimativa do atraso crítico de blocos combinacionais, a **simulação elétrica** é a que fornece os resultados mais precisos. Por outro lado, o esforço computacional requerido por esta técnica limita seu uso prático para circuitos de tamanho moderado, segundo padrões atuais de complexidade.

A **simulação de *timing*** é uma técnica similar à simulação elétrica, mas que utiliza modelos de atraso simplificados. Em função disso, a simulação de *timing* pode ser até duas ordens de grandeza mais rápida que a simulação elétrica, ao custo de uma redução de precisão da estimativa de atraso.

Existem, entretanto, três dificuldades sérias com relação ao uso de simulação para se determinar o atraso crítico de circuitos complexos. A primeira é o longo tempo de execução. A segunda diz respeito ao esforço necessário para a preparação do conjunto de estímulos a serem usados na simulação. A terceira, e provavelmente a mais séria, reside em garantir que o conjunto de estímulos propostos exercita a situação na qual o atraso crítico do circuito se manifesta. Sem dúvida, somente a simulação exaustiva nos ofereceria tal garantia. No entanto, tomando-se, por exemplo, um bloco combinacional com 100 entradas, existem 2^{100} possíveis vetores. Restringindo-se a simulação exaustiva a apenas pares de vetores (modo síncrono) seria necessário simularem-se 2^{100x99} situações diferentes.

Em função das dificuldades citadas anteriormente, a técnica denominada **análise de *timing*** tem sido extensivamente utilizada pelos projetistas na estimativa do atraso crítico de circuitos VLSI estado-da-arte.

A3.1 A Análise de *Timing*

A análise de *timing* é uma técnica para estimar-se o atraso crítico de circuitos que dispensa a aplicação de estímulos às entradas do circuito. Para que isso seja possível, cada bloco combinacional é representado como um grafo acíclico direto (DAG) no qual os nodos representam as portas e as arestas representam as conexões. Associado às arestas e às conexões existem pesos que representam os atrasos de portas e conexões, respectivamente. As entradas e as saídas primárias do circuito são representadas por nodos de entrada e de saída (i.e., com peso zero), respectivamente. Freqüentemente, o grafo é polarizado pela inclusão de um nodo fonte e de um nodo terminal. Ao nodo fonte estão ligados todos os nodos de entrada, ao passo que os nodos de saída se ligam ao nodo terminal. Num DAG polarizado, um **caminho completo** assume a forma (**s**, e_s, v_0, e_0, v_1, e_1, …, v_n, e_n, v_{n+1}, e_t, **t**), onde v_i é um nodo e e_i é uma aresta. Em especial, v_0 e v_{n+1} representam a entrada primária e a saída primária, respectivamente, **s** e **t** são os nodos fonte e terminal e e_s e e_t são arestas sem atraso. Um **caminho parcial** é um caminho que ou não termina com **t** ou não começa com **s**.

A **análise de *timing* topológica** (*topological timing analysis* - TTA) assume que o atraso crítico de um bloco combinacional corresponde ao atraso do caminho mais longo, o qual é encontrado com o uso do algoritmo topological sort [COR90]. Este algoritmo apresenta custo computacional muito reduzido e que aumenta de maneira linear com relação ao tamanho do grafo. Entretanto, o caminho topologicamente mais longo ou **caminho crítico topológico** pode não permitir a propagação de transições. Um caminho que não permite a propagação de transições é denominado **caminho falso** ou **não-sensibilizável**.

O primeiro estudo sistematizado sobre falsos caminhos se deve a Hrapcenko. Considere o circuito da figura A3.2, retirado de [HRA78]. Suponha que neste circuito cada porta tem atraso unitário e as conexões possuem atraso igual a zero. Este circuito possui dois caminhos críticos topológicos: P_1=(i_1, G_1, G_2, G_3, G_4, G_6, G_7, G_8, h) e P_2=(i_2, G_1, G_2, G_3, G_4, G_6, G_7, G_8, h), ambos com atraso 7. Entretanto, enquanto i_3=0, nenhuma transição consegue se propagar de a para b. Similarmente, enquanto i_3=1, nenhuma transição consegue se propagar de d para f. Desde que um sinal não pode assumir ambos valores lógicos ao mesmo tempo, a única maneira de se sensibilizar P_1 e P_2 é aplicar uma seqüência de vetores nas entradas do circuito de modo que i_3=1 no tempo 1 e i_3=0 no tempo 4. Entretanto, se apenas pares de vetores são permitidos, então P_1 e P_2 são declarados falsos e o atraso do circuito é menor do que 7.

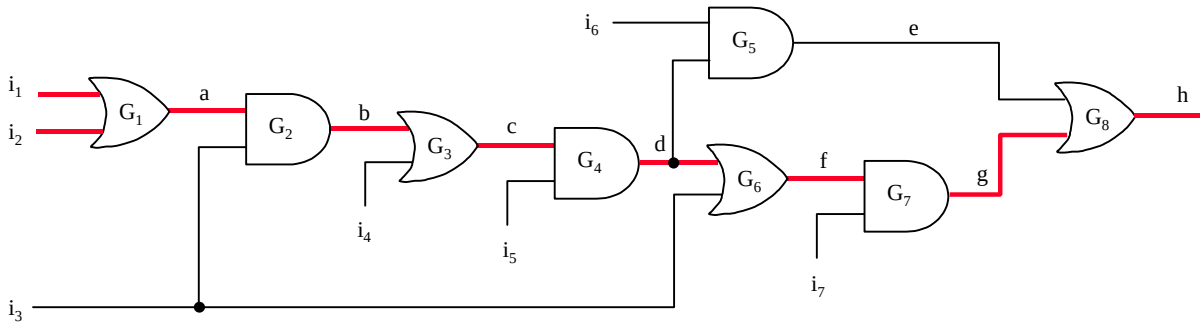


FIGURA A3.2 – Exemplo de caminho falso: circuito de Hrapcenko.

Algumas técnicas de síntese lógica são responsáveis pela introdução de redundâncias nos circuitos, as quais são sabidamente uma fonte de falsos caminhos [KEU91]. Além disso, algumas classes de circuitos são projetados de maneira que os caminhos topologicamente

mais longos são propositadamente tornados falsos. Este é o caso, por exemplo, dos somadores tipo carry skip [DEV94][LAM94]. Conclui-se, então, que a ocorrência de falsos caminhos é um fato bastante comum, principalmente devido ao uso extensivo de ferramentas de síntese automática no fluxo de projeto de circuitos VLSI. Por outro lado, caso os caminhos topológicos mais longos sejam falsos, a atraso topológico poderá representar uma estimativa deveras pessimista para o atraso crítico do circuito. Assim sendo, é altamente desejável que um analisador de *timing* seja capaz de considerar o fenômeno dos falsos caminhos.

A fim de se considerar o fenômeno dos falsos caminhos, faz-se mister que se revise o conceito de atraso crítico de blocos combinacionais. A definição mais tradicional é a **baseada em caminhos**: "o **atraso crítico** de um bloco combinacional corresponde ao atraso do caminho sensibilizável mais longo", acrescentando ainda que "pode existir mais de um caminho crítico" [CHE93]. Apesar de correta, tal definição foi derivada considerando apenas as técnicas baseadas em caminhos, as quais determinam o atraso crítico avaliando a sensibilização de cada caminho, iniciando pelo mais longo. Uma definição mais geral pode ser derivada notando-se que a essência da análise de *timing* deve ser a captura do exato instante em que a(s) saída(s) mais lenta(s) do circuito estabiliza-se(zam-se) com seu(s) valor(es) final(is). Assim sendo, pode-se redefinir o atraso de um circuito combinacional: o **atraso de um circuito** sob um dado padrão de entradas corresponde à mínima quantidade de tempo após a qual todas as saídas são garantidas estar estáveis. Por extensão, o **atraso crítico do circuito** corresponde ao maior atraso, considerando todos os padrões possíveis de entradas. Por uma questão de simplicidade, o atraso crítico será referenciado apenas por **atraso do circuito**, uma vez que este é o parâmetro de maior interesse neste trabalho. No contexto da nova definição, a técnica baseada em caminhos é uma solução possível, porém já há um certo tempo não corresponde ao estado-da-arte. Na atualidade, as técnicas de análise de *timing* ("estado-da-arte") operam sobre conjuntos de caminhos, o que resulta em significativa redução do tempo de execução. De um modo geral, qualquer técnica (algoritmo) de análise de *timing* que leve em consideração os falsos caminhos recai na categoria da **análise de *timing* funcional** (*functional timing analysis* - FTA).

Note-se que na redefinição de atraso e atraso crítico, a sensibilização de caminhos está implicitamente considerada. Porém, não é detalhado como são constituídos os padrões de entrada. Esta questão foi intencionalmente deixada em aberto para que a definição de atraso crítico ficasse independente do modo de operação assumido para o circuito. Na realidade, o que constitui um padrão de entradas válido depende do modo de operação do circuito, e é considerado no contexto do **modelo de computação de atraso de circuitos**.

O conceito de modelo de computação de atraso de circuitos foi motivado pela observação de que o atraso de um circuito depende da natureza dos sinais aplicados às suas entradas, isto é, se as entradas são assumidas como sendo **pares de vetores** ou **seqüências de vetores** [LAM94]. Para ilustrar isso, considere o circuito teste da figura A3.3a, onde o atraso de cada porta está assinalado no interior da mesma. O caminho mais longo deste circuito é (a, c, f, y), com atraso 5. Se considerarmos pares de vetores sendo aplicados às entradas do circuito, então a última transição de saída ocorre em t=1. A figura A3.3b mostra um dos possíveis pares de vetores capazes de gerar uma transição em t=1. Por Outro lado, se forem aplicadas seqüências de vetores às entradas do circuito, então a última transição de saída ocorre em t=3. A figura A3.3c mostra uma possível combinação de entradas do tipo seqüência de vetores capaz de provocar uma transição em t=3. Estes resultados revelam que o atraso de um circuito combinacional depende da maneira como vetores são aplicados às entradas.

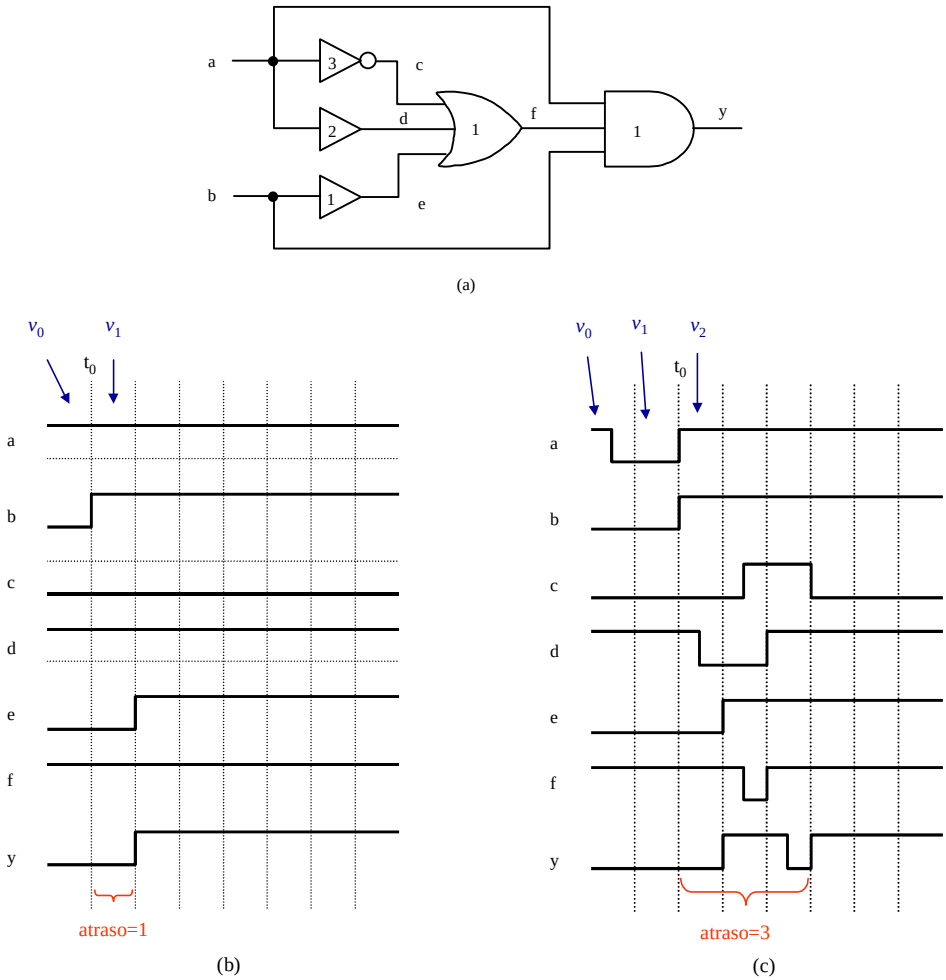


FIGURA A3.3 – O atraso dos circuitos depende da maneira como vetores são aplicados às entradas.

O cálculo do atraso para pares de vetores assume que um vetor v_1 é aplicado em t=–∞ e um segundo vetor v_2 é aplicado em t=0. O atraso do circuito é então definido como sendo o pior caso dentre os tempos de estabilização das saídas, considerando-se todos os pares de vetores possíveis. Por essa definição fica claro que todos os nós do circuito são assumidos como estando estáveis sob v_1 quando v_2 é aplicado.

O atraso para pares de vetores é equivalente a assumir-se que o circuito opera de modo totalmente síncrono. Este tipo de operação é referenciado como **modo de transição** (*transition mode*) e o atraso por ele obtido é chamado **atraso de transição** (*transition delay*) [DEV92].

Alguns autores, dentre os quais Silva et al. [SIL99], sustentam que o atraso de transição de um circuito corresponderia ao seu atraso exato. De fato, tal conjectura seria correta caso se pudesse sempre garantir a operação totalmente síncrona. Entretanto, os elementos de memória, tais como *latches* e *flip-flops*, apresentam tempos de propagação cuja discrepância de valores pode causar um desalinhamento de sinais nas entradas dos blocos combinacionais. Tal desalinhamento tem o efeito de uma seqüência rápida de vetores, caracterizando assim um funcionamento assíncrono. Outro suposto benefício de se estimar o atraso usando o modo de transição seria a possibilidade de identificar o par de vetores responsável pelo atraso crítico, o qual poderia ser usado numa simulação para a certificação da estimativa. Entretanto, os resultados mostrados pelos algoritmos que computam o atraso de transição (e.g., [DEV92],

[DEV94a], [DEV94b]) revelam o alto custo computacional, uma vez que todos se baseiam em simulação de pares de vetores.

A dificuldade encontrada na implementação de algoritmos eficientes para a computação do atraso de transição motivou a adoção da abordagem de **um vetor único**. Esta abordagem é uma aproximação do atraso para seqüência de vetores. A abordagem do vetor único assume valores lógicos arbitrários para os nós do circuito, como se estes estivessem "flutuando", até que a aplicação de um único vetor de entrada *v* determine os valores estáveis finais. O atraso do circuito é definido como sendo o maior tempo de estabilização das saídas, considerando-se todos os possíveis vetores únicos v_i. A assertiva referente aos nós estarem flutuando decorre do fato de que o circuito pode ainda estar propagando vetores de entrada aplicados antes de *v*. Na literatura, a abordagem do vetor único é mais conhecida por **modo de operação flutuante** (*floating mode)* [CHE91] e o atraso computado segundo esta é denominado **atraso flutuante** (*floating delay*).

Apesar do atraso para seqüências de vetores ser teoricamente o método exato, a inexistência de algoritmos eficientes resultou na adoção generalizada por parte dos algoritmos de FTA do método do vetor único. Esta escolha vem a ser corroborada pelos resultados apresentados em [LAM94], os quais demonstraram que na prática, o atraso por um vetor único (atraso flutuante) ou é coincidente com o atraso para seqüências de vetores, ou representa um limite superior para este. Outro fator que contribuiu para a ampla utilização do método do vetor único reside na sua semelhança com o problema da geração automática de vetores de teste (ATPG), semelhança esta que permite o aproveitamento das diversas técnicas de aceleração existentes.

Os problemas relacionados à computação do atraso de blocos combinacionais constituem, a grosso modo, apenas metade do problema da estimativa de atraso. A outra metade está relacionada aos chamados **modelos físicos de atraso** (*physical models*) ou modelos no nível de circuito (*circuit-level models*), os quais são usados para estimar o atraso individual dos componentes do circuito, tais como portas lógicas e conexões. Tipicamente, os modelos físicos de atraso são constituídos por conjuntos de equações com parâmetros que caracterizam a tecnologia a ser usada na fabricação do circuito. Alguns destes parâmetros são obtidos por meio de medições em *chips* de teste, enquanto que outros são provenientes de simulações elétricas exaustivas.

Dentre os modelos físicos, o mais simples é o chamado modelo linear, dado pela equação:

$$d(\boldsymbol{i}) = A(\boldsymbol{i}) + B(\boldsymbol{i}) \times C_L(\boldsymbol{i}) \qquad (A3.1)$$

onde A(***i***) é o **atraso de transporte** com relação à saída ***i***, B(***i***) é o inverso da capacidade de carga da porta e $C_L(\boldsymbol{i})$ é a carga capacitiva total da rede ***i***, concentrada na saída da porta. Apesar deste modelo não levar em consideração características importantes da tecnologia CMOS corrente (e.g., efeito do canal curto, rampas de entrada lentas), ele ainda é bastante utilizado pela maioria das ferramentas de síntese lógica. Por outro lado, sua simplicidade acarreta uma baixa precisão que o torna pouco útil no contexto de uma ferramentas de análise de *timing*.

Diversos modelos físicos mais sofisticados têm sido desenvolvidos desde o início dos anos 80. Dentre estes, os pioneiros foram os modelos apresentados no contexto dos simuladores de *timing* **Crystal** [OUS85] e **TV** [JOU87], os quais levam em consideração atraso de portas e de conexões de maneira simultânea. Após, diversos outros trabalhos surgiram, porém, concentrando-se ou no atraso das portas, ou no das conexões. Como exemplos dos primeiros citam-se [SAK88], [DES88], [AUV90] e [DAG99]. Exemplos do

segundo caso são [SAK83], [RUB83], [HOR84] e [RAT94]. Porém, alguns trabalhos ainda procuram tratar portas e conexões de maneira simultânea. Como exemplos, citam-se [UEB95], [FOR97] e [HIR98].

Dada a existência de diversas possibilidades de modelos físicos de atraso, outro aspecto importante diz respeito a como os atrasos individuais dos componentes serão considerados no contexto da computação do atraso do circuito. Este aspecto é considerado pelo **modelo de computação de atraso de porta**. O modelo mais simples é o **modelo de atraso fixo**, o qual assume um valor fixo *d* por porta. Uma extensão natural do modelo fixo reside em se considerar um atraso d_i para cada entrada i da porta. Outra possibilidade é considerar-se atraso de descida e de subida em separado, podendo ser um par descida/subida por porta [*df*,*dr*] ou um par descida/subida por entrada i [df_i,dr_i]. O assinalamento de um valor (fixo) de atraso ou de um par de valores (fixos) de atraso por entrada é referenciado por atraso **pino-a-pino**.

Os modelos citados no parágrafo anterior assumem valores fixos que normalmente correspondem ao máximo atraso individual dos componentes. Tais valores, quando usados no contexto da análise de *timing* topológica (TTA), jamais fornecerão uma subestimativa do atraso do circuito. Porém, é bastante comum que forneçam uma sobrestimativa, cuja gravidade dependerá da diferença de atraso entre o caminho crítico topológico e o caminho crítico real (sensibilizável). Por outro lado, em função do fenômeno dos falsos caminhos, o uso de valores (fixos) máximos de atraso no contexto da análise de *timing* funcional (FTA) pode conduzir a uma subestimativa do atraso do circuito, constituindo assim uma estimativa errônea inaceitável. Este último fenômeno foi chamado de **falha da aceleração monótona** (monotone speedup failure) por McGeer e Brayton [MCG91].

A fim de se garantir que a estimativa de atraso fornecida pela FTA seja segura (i.e., não seja uma subestimativa), os atrasos das portas devem ser especificados por intervalos limitados do tipo [d^{min},d^{max}], onde d^{min} e d^{max} representam o mínimo e o máximo atrasos que uma dada porta pode apresentar nos vários exemplares do circuito fabricado. Este é o chamado **modelo de atraso limitado** (*bounded delay model***)** e é o modelo implicitamente assumido pelo modo de transição. O modelo limitado também pode ser usado no formato pino-a-pino. Uma modificação do modelo limitado consiste em assumir d^{min} =0, dando origem ao chamado **modelo não-limitado** (*unbounded delay model*). Conforme será comentado mais adiante, o uso do modelo limitado permite que a computação do atraso flutuante do circuito forneça uma estimativa segura e mais precisa, pois leva em consideração a falha da aceleração monótona.

Uma vez detalhados os modelos de computação de atraso, tem-se os subsídios básicos para considerar-se a análise dos algoritmos de FTA existentes, bem como para determinar os requisitos básicos no desenvolvimento de novos algoritmos. Neste sentido, é importante ressaltar que o objetivo da FTA é fornecer uma estimativa segura de máxima precisão do atraso crítico do circuito. Por estimativa segura de máxima precisão entende-se um valor que represente uma sobrestimativa mais justa possível para o atraso de todos os exemplares fabricados do circuito. Para cumprir este objetivo, um algoritmo de FTA deve satisfazer a duas propriedades. São elas, a **robustez** e a **corretude**. Diz-se que um algoritmo de FTA é robusto se ele assegura uma estimativa de atraso válida para qualquer um dos exemplares fabricados do circuito. Em outras palavras, um algoritmo robusto é capaz de levar em conta a aceleração monótona e por isso, também é denominada propriedade da aceleração monótona [MCG89][MCG91]. Já a propriedade da corretude diz que o conjunto de testes de sensibilização não devem subestimar o atraso crítico do circuito. As definições formais destas duas propriedades encontram-se na seção 2.6 desta tese.

As diversas possibilidades para o desenvolvimento de algoritmos de FTA se diferenciam pelos seguintes aspectos:

- os modelos de computação de atraso para as portas e para o circuito;
- conjunto de condições usadas para testar a sensibilização dos caminhos. Este conjunto de condições é normalmente referenciado por **critério de sensibilização** e
- método usado para testar se as condições de sensibilização são satisfeitas ou não.

A seção A3.2 trata dos critérios de sensibilização, ao passo que a seção A3.3 apresenta um estudo sistemático das diversas possibilidades de algoritmos de FTA, explicitando os métodos possíveis para se testar a sensibilização de caminhos.

A3.2 Critérios de Sensibilização

À medida em que eram desenvolvidos os primeiros trabalhos em FTA, diversos critérios de sensibilização foram propostos. A razão para isso era a busca de um bom compromisso entre a precisão da estimativa de atraso e o tempo de execução do algoritmo usado no teste de sensibilização. Com efeito, este compromisso foi extremamente importante para as primeiras ferramentas de FTA, as quais testavam a sensibilização caminho por caminho, a partir dos caminhos topologicamente mais longos.

Os critérios de sensibilização podem ser divididos em **critérios independentes de tempo** e **critérios dependentes de tempo**. Enquanto que os primeiros consideram apenas os valores lógicos dos sinais aplicados às entradas laterais das portas, ou seja, as entradas que não fazem parte do caminho considerado, os segundos levam em consideração também o tempo em que tais sinais estabilizam. A seguir, são apresentados dois critérios independentes de tempo e dois dependentes de tempo, todos assumindo o modo flutuante. São eles: **sensibilização estática** [BEN90], **co-sensibilização estática** [DEV91], **viabilidade** [MCG89] e **sensibilização exata do modo flutuante** [CHE91]. Exemplos e considerações sobre a corretude de cada um dos métodos citados encontram-se na seção 4.2.

A sensibilização estática constitui, provavelmente, o primeiro critério de sensibilização que se tem notícia. Ela é herdeira direta do conceito de sensibilização empregado em geração de teste, sendo inclusive coincidente, caso se adote pares de vetores ao invés de um vetor único.

Seja um caminho P= (v_0, e_0, v_1, e_1, …, v_n, e_n, v_{n+1}). Diz-se que o vetor de entrada *w* **sensibiliza estaticamente** P para **1**(**0**) no circuito *C* se e somente se o valor de v_{n+1} é **1**(**0**), e para cada v_i, $1 \leq i \leq n$, se v_i possui o valor controlante, então a aresta e_{i-1} é a única entrada de v_i que apresenta o valor controlante. Esta definição foi apresentada por Devadas e colaboradores em [DEV93]. Cabe ainda observar que, se *w* sensibiliza estaticamente um caminho, então ele sensibiliza estaticamente ou para **1** ou para **0**. Esta propriedade é uma conseqüência direta do modo flutuante.

A co-sensibilização estática é outro critério independente de tempo, porém um pouco menos severa que a sensibilização estática, no sentido de que as condições necessárias para se declarar um caminho como co-sensibilizável estaticamente são menos restritivas. Diz-se que o vetor de entrada *w* **co-sensibiliza estaticamente** P para **1**(**0**) no circuito *C* se e somente se o valor de v_{n+1} é **1**(**0**), e para cada v_i, $1 \leq i \leq n$, se v_i possui o valor controlante, então a aresta e_{i-1}

apresenta o valor controlante. Como no caso da sensibilização estática, se *w* co-sensibiliza estaticamente um caminho, então ele co-sensibiliza estaticamente ou para **1** ou para **0**.

A análise de viabilidade, apresentada por McGeer e Brayton em [MCG89], constitui, na realidade, um algorimto completo de FTA baseado no conceito de **caminhos viáveis**. Um conjunto de condições dependentes de tempo é usado para testar se um caminho pode ser considerado como responsável pelo atraso do circuito sob um vetor de entrada *w*. Em caso afirmativo, tal caminho é declarado ser **viável**. Dado um caminho P= (v_0, e_0, v_1, e_1, …, v_n, e_n, v_{n+1}) em um circuito *C*, P é dito viável se e somente se existe ao menos um vetor de entrada *w* tal que para cada porta v_i, $1 \leq i \leq n$, e para cada entrada lateral *e* de v_i, se *e* estabiliza antes de e_{i-1}, então *e* deve apresentar o valor não-controlante de v_i. Note-se que nem o valor lógico apresentado por e_{i-1}, nem o tempo de estabilização das entradas laterais que são mais lentas que e_{i-1} interessam.

As condições para a sensibilização estática são suficientes para se concluir que um caminho pode ser responsável pelo atraso do circuito. Por outro lado, as condições para a co-sensibilização estática são apenas necessárias, mas não suficientes. As condições necessárias e suficientes para um caminho ser responsável pelo atraso do circuito no modo flutuante foram introduzidas em [CHE91] e referenciadas por **critério de sensibilização exato do modo flutuante**, ou simplesmente **critério exato**.

Seja um caminho P= (v_0, e_0, v_1, e_1, …, v_n, e_n, v_{n+1}) em um circuito *C*. P é dito **exatamente sensibilizável** ou **verdadeiro** (sob o modo flutuante) se e somente se existe ao menos um vetor de entrada *w* tal que, para cada porta v_i ao longo de P, com $1 \leq i \leq n$, uma das condições que seguem é satisfeita:

1. se e_{i-1} apresenta o valor controlante de v_i, então cada entrada lateral *e* de v_i que apresentar o valor controlante de v_i deve ser, no máximo, tão rápida quanto e_{i-1}.

2. se e_{i-1} apresenta o valor não-controlante de v_i, então cada entrada lateral *e* de v_i deve ser, no mínimo, tão rápida quanto e_{i-1}, e deve apresentar o valor não-controlante de v_i.

Assim sendo, para um caminho ser responsável pelo atraso do circuito sob o modo flutuante é necessário que, para cada porta *g* ao longo do caminho:

1. se a saída de *g* apresentar o valor controlado, então sua entrada principal deve apresentar o valor controlante e as entradas laterais que também apresentarem o valor controlante não devem ser mais rápidas que a entrada principal.

2. se a saída de *g* apresentar o valor não-controlado, então todas as entradas devem apresentar o valor não-controlante de *g* e sua entrada principal não deve ser mais rápida que as entradas laterais.

Uma análise qualitativa dos critérios de sensibilização discutidos é apresentada na seção 4.3.

A3.3 Algoritmos de Análise de *Timing* Funcional

Apesar do grande número de trabalhos sobre critérios de sensibilização e algoritmos de FTA publicados desde o fim da década dos 80, a ausência de uma terminologia padrão é uma

séria dificuldade para o desenvolvimento de atividades de pesquisa nestes temas, sobretudo para principiantes.

Esta seção discute algoritmos de computação de atraso usados em FTA. A fim de sistematizar a discussão, propõe-se uma nova taxonomia para a classificação dos algoritmos de computação de atraso.

Os primeiros algoritmos de FTA testavam a sensibilização dos caminhos, um após o outro, usando adaptações do algoritmo D [ROT66]. Este era o caso dos trabalhos apresentados em [BEN90], [BRA88], [CHE91] e [CHE93]. Nos dois últimos, os critérios de sensibilização usados eram dependentes de atraso. Esta característica de testar um caminho por vez corresponde ao conceito de **sensibilização individual de caminhos** existente em ATPG.

Porém, logo alguns autores reconheceram ser bastante comum a ocorrência de circuitos com centenas de milhares de falsos caminhos com atraso maior do que o atraso do caminho crítico (verdadeiro) [KEU91][DEV94][LAM94], de modo que o teste de sensibilização caminho por caminho foi reconhecido ser de uso limitado, na prática. Por outro lado, surgiram diversas propostas de algoritmos de FTA capazes de tratar simultaneamente a sensibilização de conjuntos de caminhos, habilidade esta que em ATPG é conhecida por **sensibilização concorrente de caminhos**. Dentre tais algoritmos merecem destaque o **procedimento de geração de teste temporal** (*timed-test generation procedure*), de Devadas et al. [DEV91] [DEV93a] e o trabalho de Silva e Sakallah [SIL94] [SIL94a]. Existe ainda uma abordagem **mista**, proposta em [CHA93], na qual um conjunto de potenciais caminhos críticos é identidicado usando sensibilização concorrente de caminhos. Após, sensibilização individual é aplicada ao conjunto.

Outra questão, a princípio independente do número de caminhos tratados, diz respeito ao método usado para determinar se as condições de sensibilização podem ser satisfeitas ou não. Nos algoritmos citados nos dois parágrados anteriores, valores lógicos são assinalados a alguns dos nós do circuito e então, implicados para os demais nós. Tal procedimento é derivado dos algorimos tradicionais de ATPG, sendo portanto referenciados por **baseados em ATPG**.

Em 1989 McGeer e Brayton chamaram a atenção para o fato de que as condições de sensibilização poderiam ser formalmente expressas por uma função de sensibilização de caminho [MCG89]. Assim, o problema de sensibilização de caminhos poderia ser resolvido usando algum método mais formal, baseado em programação dinâmica ou em BDDs, tal como o próprio McGeer propôs em [MCG89] e [MCG91], respectivamente.

Naturalmente, é de se imaginar que tal fomulação possa também ser aplicada aos algoritmos que realizam sensibilização concorrente de caminhos, obviamente com um aumento significativo na complexidade. Neste sentido, dois trabalhos representam marcas importantes. O primeiro foi o de Larrabee [LAR92], que utilizou solvabilidade (*satisfiability*) Booleana para resolver uma formulação que expressava a diferença Booleana entre circuitos com falhas e sem falhas. O segundo, apresentado por McGeer e Brayton [MCG91a], propôs as chamadas funções recursivas, as quais proveram o formalismo necessário para a aplicação de solvabilidade Booleana para FTA. Nasciam assim os algoritmos de FTA **baseados em solvabilidade** (SAT). Outros algoritmos de FTA baseados em SAT que merecem citação são apresentados em [MCG93], [SIL93], [SIL96] and [SIL99].

Sumarizando a taxonomia apresentada juntamente com a revisão histórica anterior, as soluções possíveis para FTA podem ser classificadas conforme as seguintes questões:

1. critério de sensibilização usado;
2. número de caminhos simultaneamente tratados e
3. método usado para determinar se as condições de sensibilização são satisfeitas ou não.

Com relação ao número de caminhos simultaneamente tratados, um algoritmo de FTA pode utilizar uma das seguintes estratégias:

- sensibilização individual de caminhos;
- sensibilização concorrente de caminhos;
- abordagem mista

Finalmente, os métodos possíveis para o teste de sensibilização são:

- baseado em ATPG,
- baseado em solvabilidade (baseado em SAT)
- outros (e.g., BDDs)

A tabela A3.1 classifica alguns dos algoritmos de FTA mais importantes encontrados na literatura, fazendo uso da taxonomia aqui proposta.

TABELA A3.1 – Classificação dos principais algoritmos de FTA encontrados na literatura.

algoritmo	**1**	**2**	**3**
SLOCOP [BEN90]	estática	individual	ATPG
ACPA [CHE93]	aproximada	individual	ATPG
LLLAMA [MCG89][MCG91]	viabilidade	individual	SAT ou BDDs
XBD0 [MCG91a][MCG93]	exata	concorrente	SAT
VIPER [CHA93]	vigorosa	mista	ATPG
TrueD-F [DEV93a]	exata	concorrente	ATPG
TA-LEAP [SIL94]	estática “segura”	concorrente	ATPG
STA [SIL93]	estática	concorrente	SAT
GRASP [SIL96]	estática ou viabilidade	concorrente	SAT
CGRASP [SIL99]	estática ou viabilidade	concorrente	SAT

Faz-se necessária, ainda, uma observação final sobre terminologia. Na literatura, a designação “baseada em ATPG” está associada aos algoritmos baseados em ATPG que usam sensibilização concorrente. Isto ocorre por razões históricas, pois que na realidade a sensibilização individual, que é derivada do algoritmo D, surgiu antes da sensibilização concorrente. De modo similar, a designação “baseado em SAT” assume implicitamente a sensibilização concorrente. Embora tal técnica pudesse ser aplicada à sensibilização individual, seu poder reside justamente em tratar os caminhos de maneira concorrente. Por razões didáticas, este texto utiliza a taxonomia proposta anteriormente.

A seguir, serão discutidas as seguintes classes de algoritmos: baseados em ATPG com sensibilização individual, baseados em ATPG com sensibilização concorrente e baseados em SAT com sensibilização concorrente. A discussão se utilizará de um exemplo típico para cada um dos três tipos de algoritmos.

Antes de se inciar a discussão, faz-se mister mencionar que a quase totalidade dos algoritmos de FTA apresentam três passos básicos:

1. criação do grafo que representa o circuito,
2. pré-processamento do grafo para computar máximos atrasos e
3. computação do atraso (crítico) do circuito

As três classes de algoritmos citadas se diferenciam basicamente pelo terceiro passo. Particularmente, o terceiro passo pode ser dividido em sub-passos, os quais variam significativamente, de acordo com a classificação proposta.

A3.3.1 Algoritmos Baseados em ATPG com Sensibilização Individual

Os algoritmos baseados em ATPG com sensibilização individual testam as condições de sensibilização de um caminho assinalando valores lógicos às entradas principais e às entradas laterais das portas ao longo do caminho e implicando os valores assinalados para os demais nós do circuito. Isto normalmente é feito por um algoritmo derivado do algoritmo D [ROT66], mas que apresenta menor complexidade do que aquele, pois apenas a fase de propagação é implementada. Além disso, os testes de sensibilização são aplicados a um único caminho por vez.

A fim de garantir que o caminho crítico seja encontrado, o algoritmo deve iniciar examinando o caminho topologicamente mais longo. Se as condições de sensibilização são satisfeitas, tal caminho é declarado como sendo responsável pelo atraso do circuito e seu atraso é assumido como sendo o atraso do circuito. Caso contrário, o próximo caminho topologicamente mais longo deve ser traçado e sua sensibilização deve ser testada. Este procedimento continua até que um caminho sensibilizável seja encontrado.

Os algoritmos de sensibilização individual necessitam utilizar um procedimento capaz de enumerar caminhos segundo a ordem não decrescente dos atrasos. Os algoritmos de enumeração de caminhos mais eficientes apresentam complexidade de execução $\mathbf{O}(\boldsymbol{n} \log \boldsymbol{n})$, com $\boldsymbol{n}$ igual ao número de nodos do grafo [PIN98]. O procedimento pode ser ligeiramente acelerado se o teste de sensibilização for sendo realizado à medida que o caminho for sendo traçado. Neste caso, os passos para o teste da sensibilização são os mesmos do algoritmo D: **implicação** e **justificação** [ROT66]. Para cada nova porta que é acrescida ao caminho que está sendo traçado, valores lógicos são assinalados às entradas laterais. Então, tais valores são propagados para trás, em direção às entradas primárias, e para frente, em direção às saídas primárias. No caso de condições de sensibilização dependentes de atraso, os tempos de estabilização dos valores lógicos são também considerados. Os nós não avaliados permanecem com valores desconhecidos (don’t cares). A figura A3.4 ilustra este procedimento.

Como pode existir mais de um conjunto de valores lógicos capazes de justificar um conjunto de condições de propagação para uma porta, os conjuntos de valores possíveis são armazenados numa lista. No caso de ocorrer alguma inconsistência quando da propagação das condições para as demais portas do caminho, os últimos valores assinalamentos devem ser desfeitos. Então, um novo conjunto de valores deve ser escolhido da lista, e propagado para o

resto do circuito. Se, ao final do processo, for encontrado um assinalamento de valores lógicos que satisfaça às condições de sensibilização, então o caminho será declarado sensibilizável (com respeito ao critério de sensibilização e aos modelos de atraso adotados). Por outro lado, um caminho não pode ser declarado não-sensibilizável até que todas as possibilidades de assinalamentos de valores lógicos tenham sido testadas.

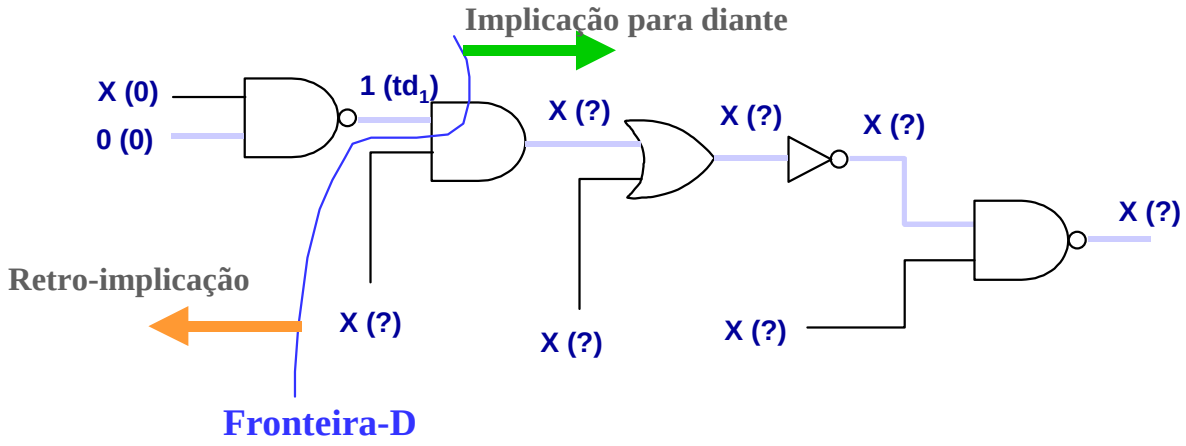


FIGURA A3.4 – Procedimento de sensibilização individual de caminho.

A fim de reduzir o tempo de execução, alguns algoritmos baseados em ATPG com sensibilização individual (e.g., [DU89] e [CHE93]) não realizam o passo de justificação, uma vez que seus autores alegam que a maioria dos caminhos falsos podem ser detectados no passo de implicação. Entretanto, de acordo com Peset Llopis, o passo de implicação não é capaz de detectar todos os caminhos falsos [PES94] e assim, quando a justificação não é realizada, um caminho não-sensibilizável pode ser declarado sensibilizável, resultando numa sobrestimativa do atraso do circuito.

Conforme já mencionado, a fase de **enumeração de caminhos** é extremamente importante para qualquer algoritmo de sensibilização individual. O problema de traçar caminhos em circuitos combinacionais tem sido estudado desde o início dos anos 80, quando vários algoritmos de enumeração foram propostos. Alguns deles eram implementações diretas dos procedimentos clássicos para percorrer grafos, a busca "primeiro em largura" (*Breadth-First Search BFS*) e a busca "primeiro em profundidade" (*Depth-First Search DFS*). Exemplos são encontrados em [YEN88] e [OUS85]. Entretanto, tais procedimentos não eram capazes de traçar os caminhos de forma ordenada, pois não consideravam informações pré-anotadas no grafo. Assim, para poder ser aplicada, havia a necessidade de armazenar os caminhos numa lista, que deveria ser posteriormente ordenada. Tal procedimento logo se mostrou infactível em função do grande número de caminhos que um circuito pode apresentar. Para solucionar este problema, Yen e Du propuseram em [YEN89] (e com mais detalhes em [YEN91]) o uso do procedimento de busca "primeiro o melhor" (*Best-First Search*), também conhecido por A* (A-star) [WIS84], o qual foi extensivamente utilizado pelos algoritmos de sensibilização individual que sucederam.

O procedimento de busca "primeiro o melhor" e seu uso nos algoritmos baseados em ATPG com sensibilização individual são detalhadamente discutidos nos itens 5.2.1 e 5.2.2 desta tese. Uma análise prática sobre sua complexidade aparece no anexo 2.

A3.3.2 Algoritmos Baseados em ATPG com Sensibilização Concorrente

A busca de um vetor de entrada *v* que satisfaça às condições de sensibilização para um dado caminho apresenta uma complexidade de execução de ordem $\mathbf{0}(2^n)$, onde n é o número de entradas primárias do circuito. Se esta busca deve ser realizada para cada caminho, como ocorre na sensibilização individual, então o tempo de execução total é proporcional ao número de caminhos longos não-sensibilizáveis que precisam ser analisados até que o caminho crítico seja encontrado. Estas dificuldades relacionadas à sensibilização individual de caminhos motivaram o surgimento de uma outra estratégia baseada não na enumeração de caminhos, mas na **enumeração de atrasos**. Esta estratégia reside em testar se as saídas primárias do circuito estão estáveis (ou em 0 ou em 1) para um dado tempo δ. Caso positivo, δ é reduzido e todas as saídas são testadas novamente. Caso negativo, i.e., se alguma saída primária não estiver estável, então o atraso do circuito está entre o valor δ testado e o valor anterior de δ. Ao testar uma saída, o algoritmo estará considerando implicitamente um conjunto de caminhos que podem influenciar a saída em questão e estará, portanto, realizando sensibilização concorrente de caminhos. No caso dos algoritmos de FTA baseados em ATPG, os procedimentos utilizados para verificar se uma saída está estável em 0 ou em 1 no tempo δ são derivados de algoritmos de ATPG.

O exemplo mais significativo de algoritmo baseado em ATPG com sensibilização concorrente é o **algoritmo TrueD-F**, desenvolvido por Devadas e colaboradores [DEV93a]. Este algoritmo responde à seguinte pergunta: **o atraso do circuito é maior ou igual a δ?** O valor δ é inicializado com $T-\varepsilon_0$, onde T é o atraso topológico do circuito e ε_0 é um pequeno valor maior que zero. Para responder à pergunta, é empregado o método de simulação de cubos de entrada (*input cube simulation*) por meio de uma versão modificada de algoritmo PODEM [GOE81], denominada de **procedimento de geração de teste temporal** (*timed-test generation procedure*) [DEV93a]. Enquanto a resposta à pergunta for "não" para cada uma das saídas primárias, o procedimento de geração de teste temporal é sucessivamente invocado para $\delta_i = T-\varepsilon_i$, com i=0,1,2,...., $\varepsilon_{i+1} > \varepsilon_i$. Se a resposta for "sim" para uma saída, então esta saída apresenta um atraso entre o valor corrente de δ (digamos $T-\varepsilon_k$) e o valor anterior ($T-\varepsilon_{k-1}$). Assim, $T-\varepsilon_{k-1}$ representa um limite superior seguro para o máximo atraso do circuito. Caso se deseje um valor mais preciso para o atraso, pode-se aplicar pesquisa binária sobre o intervalo $[T-\varepsilon_k , T-\varepsilon_{k-1}]$.

Como o circuito pode estabilizar com o valor lógico 0 ou com o valor lógico 1, para cada valor δ_i, o procedimento de geração de teste temporal deve ser aplicado duas vezes a cada saída primária (exceto no caso em que a resposta for "sim" para o primeiro valor lógico testado). A figura A3.5 ilustra o procedimento básico do algoritmo TrueD-F.

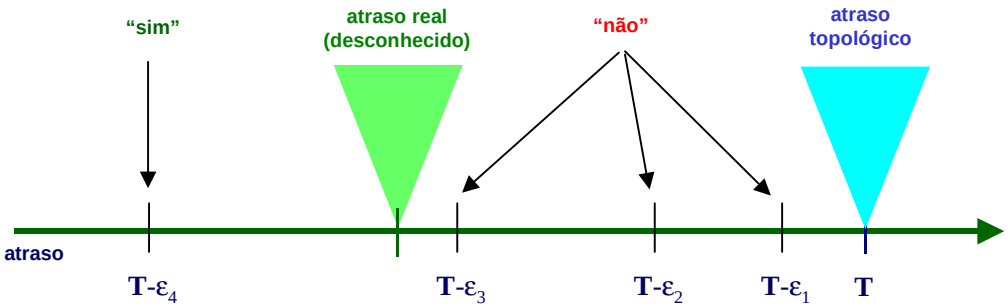


FIGURA A3.5 - Procedimento de geração de teste temporal aplicado a um circuito de uma única saída.

A fim de determinar se o atraso máximo (atraso crítico) numa saída primária do circuito é maior ou igual a δ_i para o valor lógico `lvalue`, o procedimento de geração de teste tenta justificar `lvalue` na saída considerada com atraso maior ou igual a δ_i. Isto é feito por meio de simulação de cubos, de maneira similar à realizada pelo algoritmo PODEM. Como o PODEM permite uma exploração sistemática e exaustiva do espaço das entradas, caso ele falhe em encontrar um cubo de entrada capaz de justificar `lvalue` na saída do circuito para o valor de atraso δ_i considerado, então a resposta à pergunta será "não" (para a saída testada e para o valor lógico `lvalue` considerado). Em outras palavras, o problema de determinar o atraso do circuito é transformado num problema de geração de teste para uma falha de colagem simples localizada na saída primária do circuito.

A chamada de mais alta hierarquia do procedimento de geração de teste temporal (`timed_test`) é descrita pelo pseudocódigo que segue (figura A3.6). Tal qual no algoritmo PODEM puramente lógico, há uma lista de justificação `jlist` armazenando as linhas do circuito que precisam ser justificadas. O procedimento inicia pela inserção de uma dada saída primária `po` em `jlist` com valor lógico `lvalue` (0 ou 1) e assumindo `delay` como sendo o limite inferior do atraso do circuito a ser testado (δ_i). Então, o procedimento `SEARCH_1` é chamado.

```
timed_test(po,delay,lvalue)
{
  v.value = lvalue;
  v.lower = delay;
  v.upper = INFINITY;

  modify_jlist (po,v,jlist);
  backward schedule po;

  status = SEARCH_1(jlist);
  return(status);
}
```

FIGURA A3.6 – Pseudocódigo para a chamada de mais alta hierarquia do procedimento de geração de teste temporal.

As funções de busca são similares àquelas encontradas no algoritmo PODEM e estão descritas pelos pseudocódigos das figuras A3.7 e A3.8. A função `SEARCH_1` chama a função `BACKTRACE` para, partindo da saída primária `po`, encontrar uma entrada primária cujo valor lógico ainda é desconhecido. A entrada primária identificada é inicialmente ajustada ao valor lógico 1 e então a função `IMPLY` é chamada. Esta, por sua vez, leva a cabo uma rodada de simulação de cubos para teste temporal considerando as condições de sensibilização do critério exato do modo flutuante (no PODEM original, que a partir daqui será referido por PODEM lógico, a função `IMPLY` corresponde à simulação de cubos com três valores lógicos e sem informação de atrasos). A função de implicação pode levar a uma situação de conflito, o qual pode ser de duas naturezas distintas: lógica ou temporal. Se não ocorrer conflito, `SEARCH_1` será chamada recursivamente. O procedimento termina com sucesso em `SEARCH_1` se a lista de justificação tornar-se vazia. No caso de ocorrência de conflito em `SEARCH_1`, o algoritmo retrocede ao assinalamento de entrada primária mais recente, assinalando-lhe o valor lógico 0. Então, a função `SEARCH_2`, mostrada na figura A3.8, é chamada.

O procedimento `BACKTRACE` falha se suas funções forem incapazes de encontrar uma entrada primária livre de assinalamento ou se o espaço das entradas tiver sido completamente explorado sem sucesso em `SEARCH_2`. Ocorre conflito se não for possível atribuir a uma linha do circuito o valor lógico que lhe é requerido com o período de tempo necessário. Em caso de

conflito ou falha, é necessário desfazer todos os assinalamentos originados do assinalamento de entradas que causou o conflito ou falha.

```
SEARCH_1(jlist)
{
  if(length of jlist is zero) return SUCCEED;

  if(BACKTRACE(gate,value,delay,&pi,&pi_value)==FALSE)
    return(FAILED);

  if(IMPLY(pi,pi_value,jlist)!=IMPLY_CONFLICT)
  {
    search_status = SEARCH_1(jlist);
    if(search_status == FAILED)
    {
      restore the state of the network;
      search_status = SEARCH_2(jlist,pi,1-pi_value);
    }
  }
  else
  {
     restore the state of the network;
     search_status = SEARCH_2(jlist,pi,1-pi_value);
  }
  return(search_status);
}
```

FIGURA A3.7 – Pseudocódigo para o primeiro procedimento de busca.

O procedimento de geração de teste temporal assume que cada porta do circuito (e também cada aresta) possui uma “variável lógico-temporal” formada por três campos: um limite inferior e um limite superior de atraso da porta (atrasos no contexto do circuito como um todo) e um valor lógico. Um passo de pré-processamento inicializa os campos de valores lógicos com o valor 2 (o qual indica que o valor lógico ainda não está determinado) e os limites inferior e superior de atraso com os mínimos e máximos atrasos topológicos computados a partir das entradas primárias. Na fase de geração de teste, ao serem assinalados valores lógicos conhecidos às entradas primárias, os limites inferior e superior de atraso vão sendo aproximados durante a simulação para diante, devido à sensibilização ou bloqueio dos caminhos. Os limites inferior e superior são também modificados pela retro-implicação, no momento em que novos valores são inferidos em algumas portas, como decorrência dos valores lógicos e respectivos tempos requeridos para as saídas primárias.

```
SEARCH_2(jlist,pi,pi_value)
{
   backtracks = backtracks + 1 ;
   if(backtracks > BACKTRACK_LIMIT) return(ABORTED);

   if(IMPLY(pi,pi_value,jlist)!=IMPLY_CONFLICT)
   {
     search_status = SEARCH_1(jlist);
     if(search_status == FAILED)
       restore the state of the network;
   }
   else
   {
     search_status = FAILED;
     restore the state of the network;
   }
   return(search_status);
}
```

FIGURE A3.8 - Pseudocódigo para o segundo procedimento de busca.

Outro elemento importante do procedimento de geração de teste temporal é a lista de justificação. Portas são incluídas a esta lista durante a retro-implicação e retiradas tanto durante implicação para diante quanto durante a retro-implicação. A qualquer tempo, a lista de justificação contém as portas cujos valores lógicos ou de atraso precisam ser justificados, o que só se concretiza pelo assinalamento de novas entradas primárias. O fato da lista de justificação ficar vazia significa que a busca foi concluída com sucesso. Então, a resposta para a questão “o atraso do circuito é maior ou igual a δ (quando o valor lógico lv é assinalado à saída considerada)?” é “sim”. Por outro lado, caso o espaço de busca tiver sido completamente enumerado sem que a lista de justificação tenha sido esvaziada, a busca falhou: a resposta para a pergunta é “não”. Uma terceira situação seria o caso em que a busca é abandonada devido ao número excessivo de retrocessos, quando então a pegunta permanece sem resposta.

É importante ressaltar que a variável lógico-temporal é usada para armazenar tanto os valores reais (lógicos e de atraso) quanto aqueles valores inferidos por meio da retro-implicação. A diferença entre esse dois casos reside no fato de que, no segundo caso a porta relacionada estará na lista de justificação. De fato, todas as portas cujos valores de entradas não produzem os valores constantes em sua variável lógico-temporal devem estar na lista de justificação. Por outro lado, qualquer porta cujas entradas produzem os valores constantes em sua variável lógico-temporal não deve estar na lista de justificação. Estas duas assertivas anteriores caracterizam de maneira precisa a lista de justificação.

Tendo apresentado as duas funções de mais alta hierarquia do procedimento, passemos a examinar a função `imply`, a qual é chamada dentro das funções `search1` e `search2`. A função `imply` é detalhada pelo pseudocódigo da figura A3.9. A entrada primária é assinalada com o valor lógico devido e o efeito deste assinalamento é propagado pelo circuito usando a função `forward_set`. Esta função, por sua vez, arrola o tratamento do fanout da entrada primária que foi modificada e então a função `forward_imply` realiza uma simulação temporal digirida por eventos. `forward_imply` é seguida de uma retro-implicação (função `backward_imply`). Estas duas funções são chamadas de maneira iterativa até que os valores no circuito não mudem mais. Geralmente, o assinalamento de um valor lógico particular a uma porta durante a implicação para diante ou durante a retro-implicação exige a previsão de tratamento diante de todos os seus fanouts e a previsão de tratamento para trás de todos os seus fanins que estão na lista de justificação (forward e backward scheduling).

Os eventuais conflitos são detectados por ambas funções de implicação. Estes conflitos podem ser de natureza lógica (conflitos lógicos) ou de natureza temporal (conflitos temporais).

```
imply(pi,pi_value,jlist)
{
  v = pi.timed_value;
  v.value = pi_value;
  status = forward_set(pi,v,jlist);

  while(status==IMPLY_NORMAL)
  {
        status=forward_imply(jlist);
        if(status!=IMPLY_CONFLICT)
              status=backward_imply(jlist);
  }
return status;
}
```

FIGURA A3.9 – Pseudocódigo para o procedimento de implicação.

A chave para determinar se é possível ou não justificar um valor lógico numa saída do circuito para um dado tempo reside na adoção de um cálculo temporal de três valores que leva em consideração as condições de sensibilização do critério exato do modo flutuante. Considere uma porta E de duas entradas com atraso d. Cada uma das entradas desta porta i_i pode apresentar um valor lógico pertencente ao conjunto {0,1,2} com limites inferior e superior de atraso dados por l_i e u_i, respectivamente. O termo "atraso do sinal" será utilizado, ao invés de "tempo de estabilização" [CHE93], pois mesmo portas que apresentam valor lógico não assinalado (i.e., 2) em suas saídas, apresentam valores coerentes para os limites inferior e superior do atraso. Os resultados para uma simulação de cubos usando o cálculo temporal para uma porta E de duas entradas e para uma porta OU de duas entradas são mostrados nas tabelas A3.2 e A3.3. Nestas tabelas, *lv* é o valor lógico na saída da porta, enquanto l_o e u_o representam os limites inferior e superior do atraso na saída da porta, respectivamente.

TABELA A3.2 – Cálculo temporal de três valores para uma porta E de duas entradas.

i_2	i_1	0	1	2
0	*lv*	0	0	0
	l_o	$\min(l_1,l_2)+d$	l_2+d	$\min(l_1,l_2)+d$
	u_o	$\min(u_1,u_2)+d$	u_2+d	u_2+d
1	*lv*	0	1	2
	l_o	l_1+d	$\max(l_1,l_2)+d$	l_1+d
	u_o	u_1+d	$\max(u_1,u_2)+d$	$\max(u_1,u_2)+d$
2	*lv*	0	2	2
	l_o	$\min(l_1,l_2)+d$	l_2+d	$\min(l_1,l_2)+d$
	u_o	u_1+d	$\max(u_1,u_2)+d$	$\max(u_1,u_2)+d$

TABELA A3.3 - Cálculo temporal de três valores para uma porta OU de duas entradas.

i_2	i_1	0	1	2
0	*lv*	0	1	2
	l_o	$\max(l_1,l_2)+d$	l_1+d	l_1+d
	u_o	$\max(u_1,u_2)+d$	u_1+d	$\max(u_1,u_2)+d$
1	*lv*	1	1	1
	l_o	l_2+d	$\min(l_1,l_2)+d$	$\min(l_1,l_2)+d$
	u_o	u_2+d	$\min(u_1,u_2)+d$	u_2+d
2	*lv*	2	1	2
	l_o	l_2+d	$\min(l_1,l_2)+d$	$\min(l_1,l_2)+d$
	u_o	$\max(u_1,u_2)+d$	u_1+d	$\max(u_1,u_2)+d$

O fato dos algoritmos de FTA baseados em ATPG (com sensibilização concorrente) serem derivados dos próprios algoritmos de ATPG os torna bastante interessantes, pois que as inúmeras técnicas de aceleração já desenvolvidas para os últimos podem ser aplicadas quase que diretamente aos primeiros.

A3.3.1 Algoritmos Baseados em SAT com Sensibilização Concorrente

Conforme discutido nas duas subseções anteriores, nos algoritmos baseados em ATPG, a sensibilização é considerada ou de forma explícita, traçando e testando a sensibilização dos caminhos um a um, ou de forma implícita, justificando valores lógicos nas saídas do circuito para um dado valor limite de atraso T. Estes procedimentos baseiam-se no algoritmo D e no algoritmo PODEM, respectivamente.

Nos algoritmos baseados em SAT, entretanto, a sensibilização é testada de forma implícita, usando solvabilidade Booleana (SAT). Trata-se de uma solução mais ampla, já que o teste de sensibilização (e também a detecção de falhas de colagem) pode ser formulada como um problema de solvabilidade.

No contexto dos algoritmos de FTA baseados em SAT, o método proposto por McGeer et al. [MCG93], chamado **método exato**, merece destaque em função de sua ampla aceitação junto à comunidade de análise de *timing*. Este método utiliza uma extensão do modelo de atraso não-limitado denominado XBD0 (*Extended Bounded Delay-0*), no qual um terceiro valor é acrescido ao conjunto de valores Booleanos {0,1}. Este terceiro valor, normalmente simbolizado por 2, é utilizado para modelar o estado indeterminado e também o estado desconhecido. O método também utiliza um formalismo denominado **álgebra de formas de onda**, introduzido por Augustin em [AUG89]. O modelo algébrico baseado em formas de onda seria equivalente à álgebra de chaveamento com os operadores Booleanos, tomada no domínio estático. Dentro desta nova plataforma, o modelo de computação de atraso de porta é redefinido, com o intuito de unir funcionalidade lógica e atraso de porta. Assim, pela nova definição de atraso de porta, uma porta é vista como um operador que toma as formas de onda aplicadas às suas entradas e encontra a forma de onda da saída.

Tal como os algoritmos baseados em ATPG com sensibilização concorrente, o método exato também baseia-se em enumeração de atrasos, ao invés de enumeração de caminhos. Para enumerar atrasos, o método exato se utiliza de um procedimento análogo àquele discutido na seção anterior, apenas mudando o método utilizado para testar se todas as saídas primárias estão estabilizadas para o tempo δ testado. Para este teste, o procedimento empregado caracteriza recursivamente o conjunto de todos os vetores de entrada capazes de fazer com que uma determinada saída primária estabilize com um valor lógico (0 ou 1), para o que é utilizada uma álgebra ternária e o **cálculo de forma de onda** (*waveform calculus*).

A álgebra ternária é formada pela adição de um terceiro valor ao conjunto de valores da álgebra Booleana, denotado por 2, e tem o intuito de representar os seguintes fenômenos:

- estado indeterminado, que ocorre enquanto a saída de uma porta está transicionando de 0 para 1 ou vice-versa. Em outras palavras, modela o comportamente analógico de um sinal;
- estado desconhecido, quando não é possível determinar o valor lógico de um sinal, apesar de se saber que o mesmo vale 0 ou 1. Assim, o valor 2 também serve para modelar a incertitude de cada variável que determina o atraso dos componentes do circuito, tais como variação do processo de fabricação, *crosstalk*, inclinação da rampa do sinal de entrada etc.

Sumarizando, o valor 2 representa os casos em que o valor na saída de uma porta não pode ser assegurado como sendo um valor Booleano.

Dado um tempo δ em que se deseja testar a estabilização de uma saída, o método exato encontra a função característica de cada nó. Por meio da função característica, é possível

determinar se as condições (Booleanas e temporais) necessárias para que o nó esteja estabilizado no tempo desejado podem ser satisfeitas ou não, o que é feito utilizando solvabilidade Booleana. Assim, o método exato determina, de maneira recursiva, as equações características dos nós do circuito que podem influenciar a saída que se deseja testar. Após, é aplicada solvabilidade Booleana ao conjunto de funções características. Se for possível resolver o sistema de função características, então a saída está estável no tempo δ. Por outro lado, caso não seja possível encontrar uma solução, então a saída não está estável e seu atraso é maior do que δ. É importante ressaltar que, dado um tempo δ, este procedimento deve ser realizado para cada saída. Além disso, para cada novo valor de δ será necessário testar-se cada uma das saídas. Ora, sabendo-se que os métodos de testes de solvabilidade exigem grande esforço computacional, conclui-se que a complexidade (em termos de tempo) dos algoritmos de FTA baseados em SAT é proporcional ao passo de tempo utilizado. Além disso, conforme mencionado em [MCG93], em circuitos muito grandes e/ou em circuitos que apresentam um grande número de caminhos com atrasos distintos, o número de funções características necessárias torna-se muito grande, aumentando ainda mais o tempo de execução. Em função disso, foram propostas algumas regras de poda no procedimento de geração das funções características que, no entanto, conseguem minorar o problema apenas de forma parcial. Em função desta dificuldade, os algoritmos baseados em SAT com sensibilização concorrente tem se mostrado de aplicação restrita quando modelos físicos de atraso mais realistas são utilizados. Isto porque, da adoção de tais modelos resulta uma individualização dos caminhos, no sentido de que poucos caminhos compartilham um mesmo valor de atraso.

A formulação teórica do método exato está descrita em maiores detalhes nas subseções 5.4.2 e 5.4.3 desta tese.

A próxima seção apresenta um algoritmo de FTA baseado em ATPG, com sensibilização concorrente, capaz de operar sobre circuitos que contenham portas complexas.

A3.4 Análise de *Timing* Funcional de Circuitos Contendo Portas Complexas

As primeiras técnicas de FTA realizavam sensibilização individual de caminhos utilizando variações do algoritmo D e critérios de sensibilização simplificados, buscando reduzir o tempo execução. Entretanto, logo a sensibilização individual se mostrou impraticável mesmo para circuitos de complexidade moderada, e a sensibilização concorrente tomou-lhe o lugar. Com a sensibilização concorrente as técnicas de FTA passaram a utilizar uma abordagem baseada em “enumeração de atraso”, ao invés de uma abordagem baseada em “enumeração de caminhos”. Além de não necessitar da fase de enumeração, a sensibilização concorrente também permite a adoção do critério de sensibilização exato do modo flutuante, o qual é o único capaz de fornecer o atraso exato sob o modo flutuante.

Entretanto, toda a teoria que embasa os métodos de teste de sensibilização de caminhos e também os próprios critérios de sensibilização foram desenvolvidos assumindo circuitos compostos por portas simples, isto é, portas E/NÃO-E, OU/NÃO-OU e inversores. Conseqüentemente, se um circuito combinacional que contenha portas mais complexas deve ser analisado, a ferramenta de FTA a ser utilizada deve estar apta não somente a reconhecer tais portas mas também de tratar coerentemente o circuito de acordo com o modelo computacional de atraso adotado. Isto pode ser realizado ou pela inclusão de uma fase de pré-processamento ou pela extensão do algoritmo/método de computação do atraso do circuito e das condições de sensibilização do critério adotado.

A disponibilização de geradores de macrocélulas CMOS eficientes tais como os apresentados em [CAD99] e [MOR97], e de ferramentas de mapeamento tecnológico independentes de bibliotecas [REI97] tornou possível o uso extensivo de portas complexas (sobretudo portas CMOS estáticas) no projeto físico de grandes blocos combinacionais. Deste modo, a capacidade de tratar circuitos que contenham portas complexas passa a ser altamente desejável para novas ferramentas de FTA.

Antes de iniciar uma discussão sobre a análise de *timing* funcional de circuitos que contenham portas complexas, é importante prover definições para portas simples e portas complexas. No contexto desta tese, uma **porta simples** corresponde a uma implementação física de um operador básico da álgebra Booleana. Assim, são portas simples as portas E, OU, NÃO-E, NÃO-OU (com qualquer número de entradas) e o inversor. Particularmente, **portas simples CMOS** são portas simples que podem ser implementadas diretamente em tecnologia CMOS, ou seja, portas NÃO-E, NÃO-OU e inversor. Uma porta complexa corresponde a uma implementação física de qualquer função Booleana de complexidade maior do que os operadores Booleanos básicos. Especificamente, uma porta complexa CMOS estática (*Static CMOS Complex Gate* - SCCG) corresponde a uma porta complexa implementada em tecnologia CMOS.

Inicialmente, será investigado o caso particular das SCCGs, em função de sua importância. Após, serão feitas considerações sobre a extensão dos modelos e algoritmos propostos para o caso de portas complexas quaisquer.

Uma porta CMOS estática é implementada por uma rede de transistores CMOS conectados segundo uma topologia de “restauração completa”, composta por uma rede PMOS e uma rede NMOS. A rede PMOS é capaz de prover um caminho entre a saída da porta e a massa (Vdd), enquanto que a rede NMOS é capaz de prover um caminho entre a saída da porta e a terra (Gnd). As redes NMOS e PMOS possuem mesmo número de transistores. Por uma questão de simplicidade, iremos assumir que uma SCCG é uma porta CMOS estática na qual ambas redes de transistores apresentam apenas associações série/paralelo, e sendo rede PMOS o dual da rede NMOS, em termos de associação de transistores. A figura A3.10 mostra um exemplo de SCCG. Note-se que, para uma porta CMOS estática de *n* entradas, há *n* pares NMOS/PMOS conectados pela grade, formando cada par uma entrada da porta.

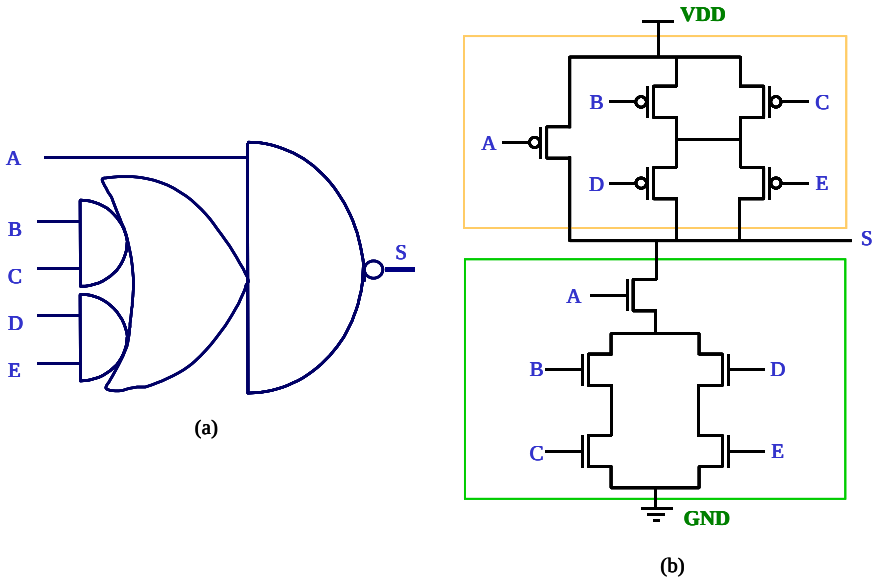


FIGURA A3.10 - Exemplo de SCCG.

As portas CMOS estáticas, incluindo as SCCGs, podem ser classificadas de acordo com o número de transistores série/paralelo existentes nas redes NMOS e PMOS. O conjunto das portas CMOS estáticas que apresentam não mais do que *n* (*p*) transistores NMOS (PMOS) em série é definido como sendo uma "biblioteca virtual" [REI98] que pode ser designada por SCG(*n*,*p*). Pode-se também utilizar a notação SCCG(*n*,*p*) para designar o subconjunto de SCG(*n*,*p*) composto apenas por SCCGs. A tabela A3.4, retirada de [DET87], detalha o número de SCGs existentes para bibliotecas virtuais de até 5 transistores série.

TABELA A3.4 – Número de elementos para várias bibliotecas virtuais [DET87].

		número de transistores PMOS em série				
		1	**2**	**3**	**4**	**5**
número de transistores NMOS em série	**1**	1	2	3	4	5
	2	2	7	18	42	90
	3	3	18	87	396	1677
	4	4	42	396	3503	28435
	5	5	90	1677	28435	425803

Muitos algoritmos de FTA foram desenvolvidos na década dos 90. Entretanto, a maioria destes não considera a possibilidade de tratar circuitos que contenham portas complexas. No caso dos algoritmos baseados em ATPG, tal limitação deve-se ao fato da teoria de sensibilização de caminhos ter sido desenvolvida unicamente para portas simples. Obviamente, a extensão de um critério de sensibilização para considerar portas complexas resulta em regras mais complicadas. O uso de tais regras por um algoritmo de sensibilização individual de caminhos tende a piorar significativamente o desempenho deste. Em uma primeira análise, os algoritmos baseados em SAT parecem ser mais promissores, uma vez que as equações características são potencialmente capazes de representar qualquer função Booleana. Entretanto, a fim de reduzir a complexidade das instâncias de SAT a serem resolvidas, alguns algoritmos baseados em SAT assumem que os circuitos combinacionais são compostos unicamente de portas simples.

A solução mais simples para realizar-se FTA de circuitos contendo portas complexas consiste em substituir-se cada porta complexa por um subcircuito equivalente composto de portas simples. Esta técnica é conhecida como macroexpansão [MCG91][HSU98] e, quando utilizada a título de pré-processamento, viabiliza o uso de qualquer ferramenta de FTA que tenha sido desenvolvida para tratar circuitos constituídos unicamente por portas simples. Entretanto, a macroexpansão apresenta dois inconvenientes [HSU98]. Em primeiro lugar, é muito difícil modelar com precisão o atraso das portas complexas macroexpandidas. Em segundo lugar, a macroexpansão cria novos nós no circuito. Estes novos nós representam um potencial aumento no número de linhas que devem ser justificadas, no caso de FTA baseada em ATPG, ou num aumento do número de equação características, no caso de FTA baseada em SAT. Em qualquer um destes casos, o aumento no tempo de execução dependerá da complexidade das portas complexas e dos modelos de atraso utilizados para estas.

Uma segunda solução reside em modificar os testes de sensibilização de modo a torná-los aptos a tratar de circuitos que contenham portas complexas. Tal modificação refere-se não apenas ao critério de sensibilização, mas também ao algoritmo que testa a sensibilização.

Desde que há mais de um algoritmo para testar a sensibilização de caminhos, esta solução pode ser desdobrada em várias soluções.

Em [HSU98], por exemplo, é apresentado um algoritmo de FTA capaz de operar diretamente sobre circuitos com portas complexas. Com o intuito de evitar a macroexpansão, as condições para sensibilização exata no modo flutuante são estendidas, de modo a considerar portas complexas. Estas condições de sensibilização estendidas são utilizadas por um algoritmo baseado em sensibilização individual derivado do algoritmo D. Os resultados mostrados em [HSU98] permitem comparar modelos de atraso para macroexpansão, bem como comparar o uso da macroexpansão com o algoritmo que testa diretamente portas complexas. Por outro lado, todos os resultados foram obtidos pelo uso de algoritmos baseados em sensibilização individual de caminhos, a qual, sabidamente, não representa o estado-da-arte por sofrer de um problema conhecido por “explosão de caminhos”.

Conforme já apresentado no item A3.3, o procedimento de geração de teste temporal de Devadas et al. [DEV93a] é um algoritmo de sensibilização concorrente de caminhos baseado em ATPG derivado do algoritmo PODEM [GOE81]. No procedimento de geração de teste temporal, o problema de se calcular o atraso de um circuito é transformado num conjunto de geração de testes para falhas de colagem “temporais” imaginadas como ocorrendo nas saídas primárias do circuito. O procedimento tem então o objetivo de justificar tais falhas de colagem.

A fim de se poder estender o procedimento de geração de teste temporal para circuitos que contenham portas complexas, é necessário generalizar o cálculo temporal para portas E/OU de *n* entradas. Para tanto, lança-se mão dos conceitos de valor controlante e valor não-controlante, conforme definidos no escopo de teste. Assim, dada uma porta *g* tipo E/OU de *n* entradas, podemos classificar seu estado lógico conforme os seguintes casos:

1. Casos em que ao menos uma das entradas de *g* apresenta o valor controlante (c(*g*)). As demais entradas podem apresentar ou o valor não-controlante (nc(*g*)) ou o valor 2;
2. Caso em que todas as entradas de *g* apresentam o valor não-controlante (nc(*g*));
3. Casos em que ao menos uma das entradas de *g* apresenta o valor 2, mas nenhuma entrada apresenta o valor controlante (c(*g*)). As demais entradas podem apresentar o valor não-controlante (nc(*g*)).

A partir da identificação dos casos possíveis, chega-se à generalização das regras originalmente expressas nas tabelas A3.2 e A3.3. Também é possível expandir tais regras de modo a se considerar a polaridade de saída das portas, o que permite tratar portas NÃO-E e NÃO-OU. As regras generalizadas são mostradas na tabela A3.5.

TABELA A3.5 – Regras generalizadas para o cálculo temporal de três valores para portas simples de *n* entradas.

grupo		regras	*lv*
1	l_o	$\min\{ l_i \mid i = c(g) \text{ or } i=2 \} + d$	$pol(g) \oplus c(g)$
	u_o	$\min\{ u_j \mid j = c(g) \} + d$	
2	l_o	$\max\{ l_i \} + d$	$pol(g) \oplus nc(g)$
	u_o	$\max\{ u_j \} + d$	
3	l_o	$\min\{ l_i \mid i=2 \} + d$	2
	u_o	$\max\{ u_j \} + d$	

As regras resumidas na tabela A3.5 podem ainda ser representadas de maneira gráfica, conforme mostram as figuras A3.11, A3.12 e A3.13. Por uma questão de simplicidade, tanto na tabela A3.5 como nas figuras A3.11, A3.12 e A3.13 assumiu-se um atraso único por porta. Modelos computacionais de atraso mais sofisticados serão discutidos mais adiante.

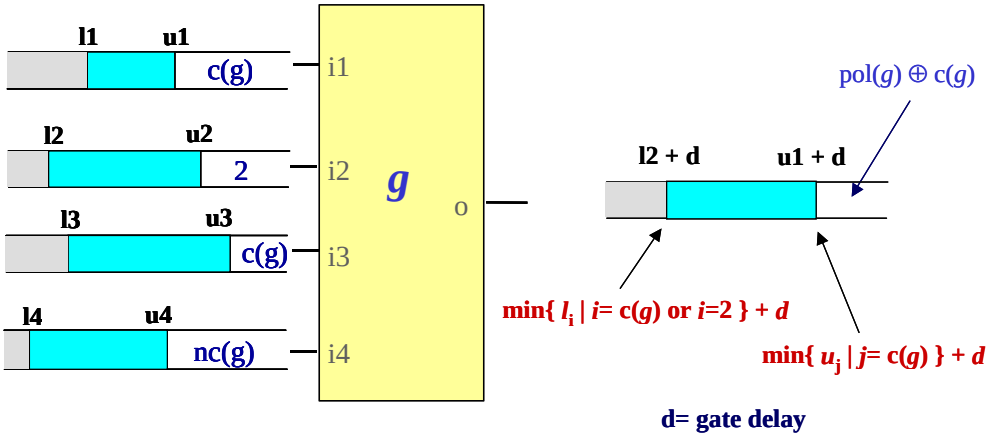


FIGURA A3.11 – Cálculo temporal de três valores para o grupo 1.

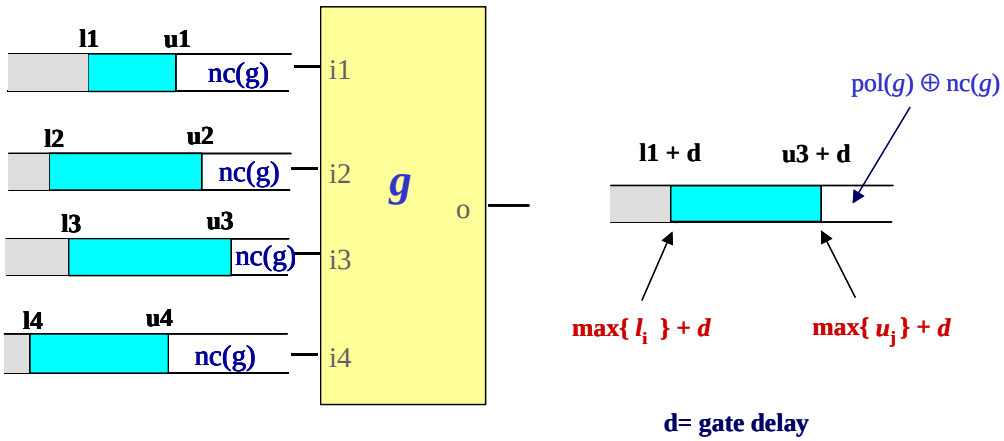


FIGURA A3.12 - Cálculo temporal de três valores para o grupo 2.

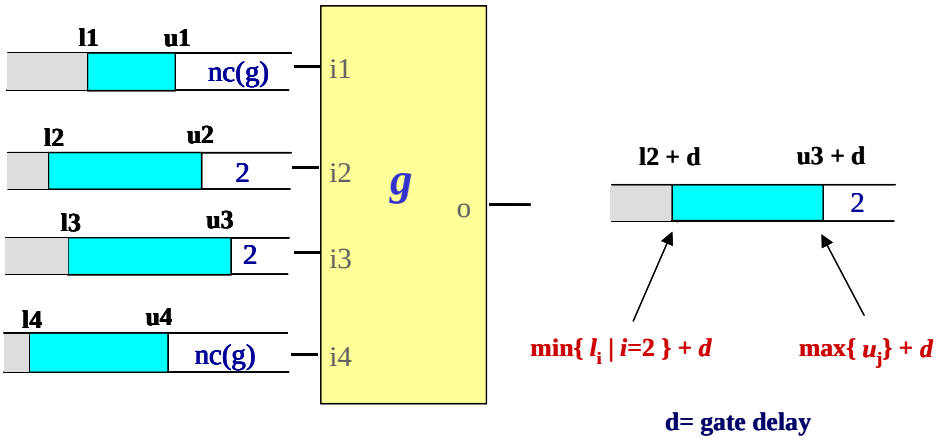


FIGURA A3.13 - Cálculo temporal de três valores para o grupo 3.

Assumindo-se que a função lógica de uma porta qualquer *g* esteja na forma fatorada e utilizando-se uma estrutura de dados apropriada para representá-la, a aplicação das regras generalizadas é direta. Considere, por exemplo, a SCCG mostrada na figura A3.14a. A função

lógica desta porta é dada por $S = \overline{A \cdot ((B \cdot C) + (D \cdot E))}$ e pode ser representada por uma "árvore da função" (figura A3.14c). Examinando-se esta árvore da função nota-se que, dado um assinalamento de valores temporais de entrada (valores lógicos e respectivos limites inferior e superior de atraso), os valores temporais na saída podem ser obtidos mediante a aplicação sucessiva das regras da tabela A3.5 para cada subárvore, iniciando-se pela subárvore mais próxima da base, desde que sejam feitas as seguintes assertivas:

1. Todas as subárvores, exceto a raiz, apresentam atraso de propagação zero e polaridade igual a zero
2. atraso de propagação da porta é aplicado somente sobre os limites inferior e superior de atraso da saída da porta, i.e., sobre os valores temporais resultantes da avaliação da subárvore de maior hierarquia.
3. A polaridade da porta é tratada somente quando a subárvore de maior hierarquia é processada.

A primeira e a segunda assertiva permitem que se divida a avaliação temporal da SCCG em dois passos independentes. No primeiro passo, o valor lógico da saída e os limites inferior e superior de atraso são computados para um atraso de propagação da porta igual a zero. Chamaremos este intervalo de atrasos de "limites de atraso de primeira ordem". Num segundo passo, os limites inferior e superior reais são computados por meio da adição do atraso de propagação da porta aos limites de atraso de primeira ordem. Este segundo passo leva em conta o modelo computacional de atraso adotado para as portas.

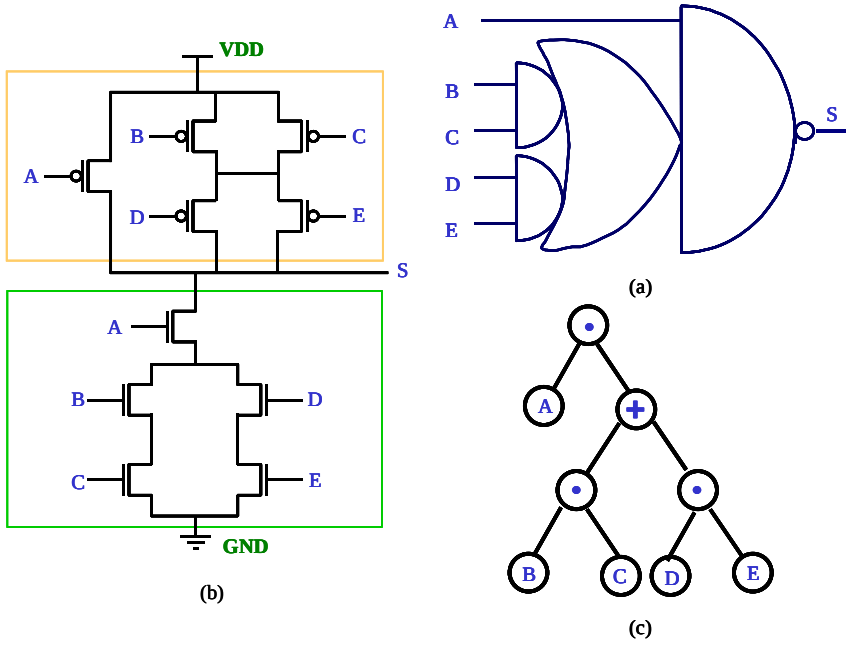


FIGURA A3.14 - Exemplo de SCCG: símbolo para o nível lógico (a), esquemático de transistores (b) árvore da função (c).

A figura A3.15 ilustra a aplicação das regras do cálculo temporal aplicadas à SCCG da figura A3.14.

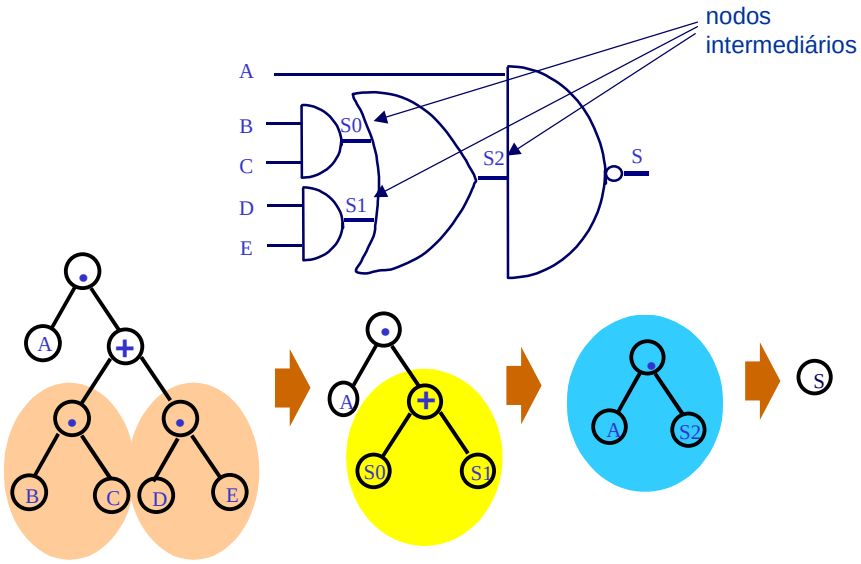


FIGURA A3.15 – Uso do cálculo temporal de três valores para avaliar uma SCCG.

Uma vez proposto um procedimento para determinar os limites de atraso de primeira ordem na saída de uma porta complexa, faz-se necessário investigar modelos computacionais de atraso para portas que sejam válidos para o cálculo do atraso flutuante de um circuito. A fim de revelar a relação que existe entre o modelo computacional de atraso de circuitos e o modelo computacional de atraso de portas (e também, o modelo físico de atraso), consideremos a porta NÃO-E de 3 entradas mostrada na figura A3.16. Considere também que o vetor *v*=(a=1;b=1;c=0) seja aplicado a suas entradas. A fim de se determinar o atraso flutuante desta porta NÃO-E sob o vetor *v*, é necessário examinar todas as combinações de entrada contidas em tal vetor, encontrando o pior atraso. Para iniciar, é necessário lembrar que no modo flutuante, um valor 0 representa tanto a transição de descida quanto o valor 0 estático, enquanto que um valor 1 representa tanto a transição de subida quanto o valor 1 estático. Então, substituindo-se 0 por ↓ ou por 0 e substituindo-se 1 por ↑ ou por 1, obtém-se um total de 8 possibilidades de pares de vetores (figura A3.16b) o que permite estabelece-se uma relação entre o modo flutuante e o modo de transição, no que se refere ao modelo de atraso de portas lógicas.

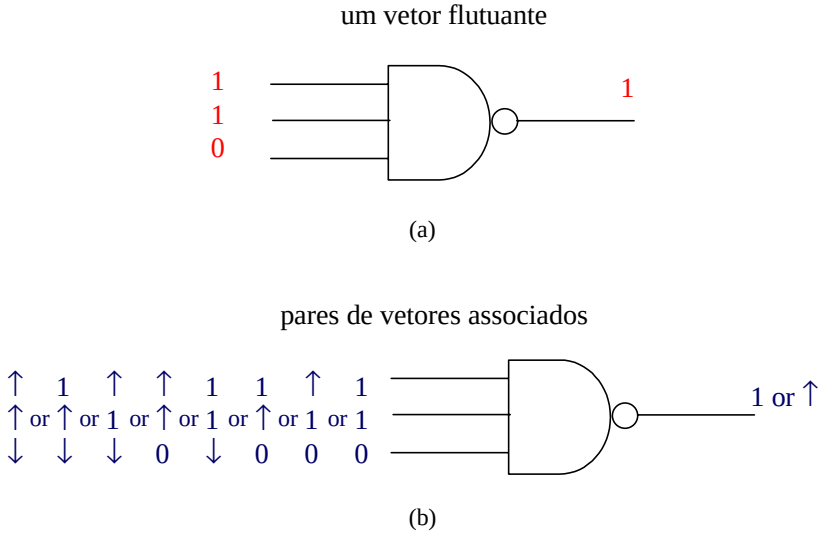


FIGURA A3.16 – Relação entre modo flutuante e modo de transição: um vetor flutuante aplicado a uma porta NÃO-E de 3 entradas (a) e os 8 pares de vetores associados (b).

Antes de continuar a discussão, é conveniente definir-se vetor flutuante e atraso flutuante.

Definição A3.1: **vetor flutuante** e **atraso flutuante**

> Um **vetor flutuante** é um assinalamento de valores lógicos aplicado às entradas de uma porta ou às entradas do circuito, quando assume-se que o circuito está operando no modo flutuante. O atraso computado para a porta (para o circuito) assumindo-se tal modo de operação é chamado **atraso flutuante** da porta (do circuito).

Dada a definição para vetor flutuante, vale lembrar que um vetor flutuante de *n* variáveis contém 2^n pares de vetores, dentre os quais um é composto por dois vetores iguais. A tabela A3.6 mostra todos os vetores flutuantes possíveis e os pares de vetores associados para uma porta NÃO-E de 3 entradas. Os pares hachureados em cinza claro são responsáveis por uma única transição de saída, enquanto que os pares hachureados em cinza escuro podem provocar uma transição espúria (*glicth*) na saída da porta.

TABELA A3.6 – Relação entre vetores do modo flutuante e vetores do modo de transição.

vetor de entrada	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
pares de vetores associados	000	001	010	011	100	101	110	111
	↓00	↓01	↓10	↓11	↑00	↑01	↑10	↑11
	0↓0	0↓1	0↑0	0↑1	1↓0	1↓1	1↑0	1↑1
	00↓	00↑	01↓	01↑	10↓	10↑	11↓	11↑
	↓↓0	↓↓1	↓↑0	↓↑1	↑↓0	↑↓1	↑↑0	↑↑1
	↓0↓	↓0↑	↓1↓	↓1↑	↑0↓	↑0↑	↑1↓	↑1↑
	0↓↓	0↓↑	0↑↓	0↑↑	1↓↓	1↓↑	1↑↓	1↑↑
	↓↓↓	↓↓↑	↓↑↓	↓↑↑	↑↓↓	↑↓↑	↑↑↓	↑↑↑
valor lógico da saída	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**

Indo um pouco além, é possível criar-se uma tabela de equivalência entre modo flutuante e modo de transição como a tabela A3.6 para cada uma das 256 funções Booleanas de 3 variáveis. Desde que a associação entre vetores flutuantes e pares de vetores é fixa, as funções lógicas de 3 entradas se distinguiriam pelos valores lógicos resultantes na saída e também pelos pares de vetores que influenciam a computação do atraso da porta. Potencialmente, pode-se construir tabelas de equivalência para qualquer função lógica de *n* entradas.

Uma contribuição da tabela de equivalência reside no fato de que ela permite a identificação dos modelos computacionais de atraso para portas a serem usados para calcular o atraso flutuante de um circuito. Mais especificamente, a tabela diz que uma porta pode apresentar ao menos um atraso distinto por vetor flutuante de entrada. De maneira mais conservadora, pode-se considerar que o modelo computacional de atraso de porta mais detalhado para o modo flutuante corresponde a assumir-se um valor de atraso por vetor flutuante. Tal modelo será referenciado por **modelo de atraso de vetor**.

Um modelo menos preciso, porém ainda correto do ponto de vista da análise de *timing*, corresponde a agrupar os vetores flutuantes de entrada de acordo com o tipo de transição de saída da porta. Assim, a cada porta é assinalado um atraso de descida, correspondendo ao máximo atraso dentre os vetores flutuantes que provocam uma transição de descida, e um atraso de subida, correspondendo ao máximo atraso dentre os vetores flutuantes que provocam uma transição de subida. Tal modelo será chamado **modelo de atraso de subida/descida**.

Um terceiro modelo corresponde a assinalar um único valor de atraso por porta, chamado de **modelo de atraso único**. Naturalmente, este é o modelo menos preciso, embora ainda correto.

É importante ressaltar que, para obter-se um cálculo robusto de atraso flutuante do circuito, os valores de atraso de portas a serem usados em cada um dos três modelos deve corresponder ao limite superior para o atraso da porta, considerando-se todas as instâncias do circuito fabricado. Isto significa que o modelo físico de atraso usado ou o método de estimativa de atraso deve ser capaz de fornecer o valor correspondente ao pior caso individualmente para cada componente.

Uma análise mais cuidadosa da tabela A3.6 permite ainda concluir que o formato de atraso pino-a-pino, normalmente utilizado na caracterização de standard-cells, não é apropriado para a computação do atraso flutuante desde que considera somente parte de todos os vetores flutuantes de entrada. Por exemplo, no caso de uma porta de 3 entradas, o formato pino-a-pino cobre apenas seis pares de vetores: $\downarrow$11, 1$\downarrow$1, 11$\downarrow$, $\uparrow$11, 1$\uparrow$1, 11$\uparrow$. Ou seja, apenas as situações em que ocorre uma única transição de entrada.

Uma segunda contribuição da tabela de equivalência entre modo flutuante e modo de transição é a própria associação entre um dado vetor flutuante e os pares de vetores subscritos. Tal informação é útil quando se deseja caracterizar o atraso de uma porta, sobretudo se o método de caraterização for simulação elétrica. Neste caso, apenas os pares de vetores que podem produzir uma transição de saída devem ser simulados, o que pode reduzir significativamente o número de casos. Por outro lado, no caso de caracterização por meio de formulação analítica, a informação da tabela de equivalência pode ser usada para ajustar ou adaptar o conjunto de equações de modo a cobrir apenas os pares de vetores de interesse.

Tendo derivado regras seguras para determinar o atraso das portas sob qualquer um dos três modelos computacionais de atraso de porta, faz-se necessário considerar a aplicação de tais regras no cálculo dos limites inferior e superior de atraso de uma porta usando o procedimento de implicação para diante. Retomando as regras do cálculo temporal de três valores descritas na tabela A3.5, as situações possíveis de vetores flutuantes podem ser classificadas nos três grupos que seguem:

1. Casos em que ao menos uma das entradas de g apresenta o valor controlante (c(g)). As demais entradas podem apresentar ou o valor não-controlante (nc(g)) ou o valor 2;
2. Caso em que todas as entradas de g apresentam o valor não-controlante (nc(g));
3. Casos em que ao menos uma das entradas de g apresenta o valor 2, mas nenhuma entrada apresenta o valor controlante (c(g)). As demais entradas podem apresentar o valor não-controlante (nc(g)).

Os grupos 1 e 3 referem-se aos casos onde uma das entradas da porta apresenta o valor 2. No contexto do cálculo de três valores, um 2 representa a existência ou do valor lógico 0 ou do valor lógico 1. Em decorrência disto, podemos introduzir a definição de cubo flutuante:

Definição A3.2: **cubo flutuante**

> Dado o cálculo de três valores, um **cubo flutuante** é um assinalamento de valores lógicos tal que ao menos um destes corresponde ao valor 2. O número de vetores flutuantes contidos em um cubo flutuante é igual a 2^m, onde m é o número de posições que apresentam o valor 2.

Com efeito, um cubo flutuante contém ao menos dois vetores flutuantes e desta forma, pode ser visto como uma variação do modelo de atraso de subida/descida. A fim de se garantir máxima precisão, é necessário considerar o máximo e o mínimo atrasos da porta sob um dado cubo. A partir de tal assertiva, o atraso de uma porta sob um cubo terá uma forma invariante, independentemente do modelo computacional de atraso de porta. Para efeitos ilustrativos, considere a aplicação do cubo flutuante 022 à SCCG da figura A3.17, a qual implementa a função lógica $S = A + B \cdot C$. A tabela A3.7 mostra a equivalência entre o modo flutuante e o modo de transição para tal porta. Note que a saída desta SCCG será 1 quando o vetor flutuante aplicado a suas entradas pertencer a {000, 001, 010} e será 0 quando o vetor flutuante aplicado a suas entradas pertencer a {011, 100, 101, 110, 111}. Se o procedimento de cálculo do atraso assumir o modelo de atraso de vetor, então o atraso desta SCCG para o cubo 022 terá máximo e mínimo limites de atraso dados por max{d(000),d(001),d(010),d(011)} e min{d(000),d(001),d(010),d(011)}, respectivamente, onde d(000), d(001), d(010) e d(011) são os atrasos para os vetores flutuantes 000, 001, 010 e 011. Por outro lado, se o procedimento de cálculo do atraso assumir o modelo de atraso de subida/descida, então o máximo e o mínimo limites de atraso serão dados por max{tp_{LHmax} , tp_{HLmax}} e min{tp_{LHmin} , tp_{HLmin}}. Entretanto, para esta SCCG, tp_{LHmax}=max{d(000),d(001),d(010)} e tp_{LHmin}=min{d(000),d(001),d(010)}, enquanto que tp_{HLmax}=tp_{HLmin}=d(011), resultando nos mesmos valores de máximo e mínimo atrasos fornecidos pelo modelo de atraso de vetor. De forma análoga, pode-se declarar que o atraso flutuante deste cubo para o modelo de atraso subida/descida irá resultar nos mesmos valores de máximo e mínimo atrasos calculados pelos outros dois modelos de atraso de porta.

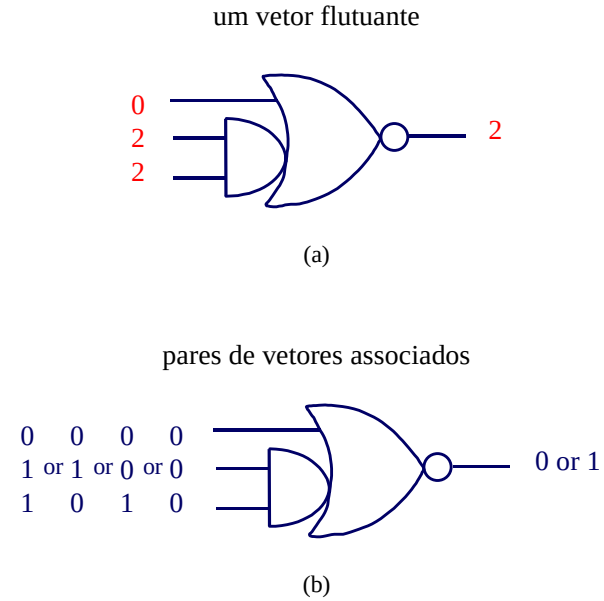


FIGURA A3.17 – Atraso da SCCG da figura A3.17 para o cubo flutuante 022.

TABELA A3.7 – Equivalência entre modo flutuante e modo de transição para a SCCG da figura A3.17.

vetor de entrada	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
pares de vetores associados	000	001	010	011	100	101	110	111
	↓00	↓01	↓10	↓11	↑00	↑01	↑10	↑11
	0↓0	0↓1	0↑0	0↑1	1↓0	1↓1	1↑0	1↑1
	00↓	00↑	01↓	01↑	10↓	10↑	11↓	11↑
	↓↓0	↓↓1	↓↑0	↓↑1	↑↓0	↑↓1	↑↑0	↑↑1
	↓0↓	↓0↑	↓1↓	↓1↑	↑0↓	↑0↑	↑1↓	↑1↑
	0↓↓	0↓↑	0↑↓	0↑↑	1↓↓	1↓↑	1↑↓	1↑↑
	↓↓↓	↓↓↑	↓↑↓	↓↑↑	↑↓↓	↑↓↑	↑↑↓	↑↑↑
valor lógico da saída	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**

Analisando novamente as tabelas de equivalência, é possível afirmar que o atraso de qualquer porta depende apenas do cubo (ou vetor) aplicado às suas entradas. Particularmente no caso do modelo subida/descida, pode-se afirmar que o atraso de uma porta fica perfeitamente definido pelos grupos aos quais o vetor (ou cubo) flutuante de entrada pertence. Tal conclusão ratifica a decisão de se dividir o cálculo do valor temporal de uma porta em duas subtarefas independentes:

1. Cálculo do valor lógico da saída e dos limites de primeira ordem;
2. Identificação do atraso flutuante da porta e aplicação deste no cálculo dos limites de atraso reais.

A partir desta verificação, torna-se claro que os recursos necessários para realizar-se a computação do valor temporal para portas complexas arbitrárias já foram propostos. Estes recursos são:

1. A árvore da função e os procedimentos necessários para computar o valor lógico na saída das portas e os limites de atraso de primeira ordem;
2. O uso de um dentre os três modelos de cálculo de atraso de porta (atraso de vetor, atraso de subida/descida ou atraso único) para calcular os limites inferior e superior do atraso das portas.

Uma vez detalhado o procedimento para realizar-se a implicação para diante, resta-nos investigar o procedimento para a retro-implicação.

Dada uma porta e um valor lógico desejado para a sua saída, a retro-implicação consiste em encontrar um assinalamento de entradas capaz de produzir tal valor. Obviamente, o valor lógico desejado na saída da porta é um valor conhecido, ou seja, 0 ou 1. Ocorre conflito quando a saída de alguma porta apresenta valor lógico oposto ao que se deseja.

No caso da retro-implicação temporal, não basta verificar a ocorrência de conflitos lógicos. Para cada entrada de cada porta, é necessário verificar se os limites inferior e superior de atraso requeridos estão dentro dos limites inferior e superior de atraso pré-existentes. Dada uma porta simples e um valor lógico desejada para sua saída, o procedimento original de

geração de teste temporal procura pelo cubo capaz de produzir tal valor lógico explorando os conceitos de valor controlante/valor não-controlante. Considere que se deseje colocar a saída de uma porta E no valor lógico 0. Como 0 é o valor controlante da porta E, existe mais de uma possibilidade de vetores capazes de satisfazer a condição desejada. Assim, o procedimento procura pela entrada capaz de produzir um 0 na saída da porta, e sendo esta enrada a mais rápida. Se tal entrada não existe, então não é possível implicar para trás o valor lógico 0 pela porta considerada.

No caso das portas complexas, os conceitos de valor controlante/valor não-controlante não podem ser aplicados diretamente. Entretanto, percebe-se que implicar para trás um valor lógico por uma porta pode corresponder a um “processo de adivinhação”, no qual se tenta descobrir um assinalamento de entradas capaz de produzir o valor lógico de saída desejado. Neste sentido, o procedimento de retro-implicação para portas complexas vem a ser uma generalização do caso das portas simples.

A heurística usada pelo procedimento de geração de teste temporal na retro-implicação com portas simples reside em definir o valor lógico do menor número possível de entradas da porta, pois isto resultará num menor número de linhas do circuito necessitando justificação. Tal procedimento pode ser visto como uma exploração sistemática do subespaço Booleano local (i.e., espaço Booleano associado à função realizada pela porta considerada), ao mesmo tempo em que uma variável por vez é “setada”.

O raciocínio desenvolvido nos últimos dois parágrafos sugere o uso de um procedimento de enumeração sistemática do espaço de entradas, juntamente com o cálculo temporal de três valores apresentado na tabela A3.5, para realizar a retro-implicação lógica em portas complexas. A figura A3.18 exemplifica o procedimento de retro-implicação quando se deseja o valor lógico 0 na saída da SCCG da figura A3.17. Para tanto, inicia-se colocando o valor lógico 0 na entrada superior. Implicando-se para diante, descobre-se que a saída da SCCG mantém-se com valor 2 (figura A3.18a). Isto significa que é necessário “setar” uma ou mais entradas. Assinalando-se o valor 0 à entrada do centro resulta que a saída da SCCG terá valor lógico 1 (figura A3.18b), o que corresponde a um conflito. Então, retrocede-se ao estado anterior, substituindo-se o valor da entrada central pelo seu oposto, i.e., 1. A implicação desta nova situação de entradas resulta no valor 2 na saída da SCCG (figura A3.18c). Aplicando-se o valor 0 na entrada inferior resulta no valor lógico 1 na saída (figura A3.18d), o que corresponde a um conflito. Finalmente, trocando-se o valor da entrada inferior para 1 obtém-se o valor lógico 0 desejada para a saída.

A solução para retro-implicação com portas complexas descrita acima equivale a aplicar o algoritmo PODEM a uma única porta, o que exige tempo de execução reduzido, desde que se restrinja o número de entradas da porta. Entretanto, o aspecto mais importante de tal solução reside no fato dela permitir o uso das regras do cálculo temporal de três valores e da árvore da função propostos neste trabalho. O único recurso extra de que se necessita é um mecanismo de enumeração que permita o controle do subespaço referente à porta cuja saída se deseja retro-implicar.

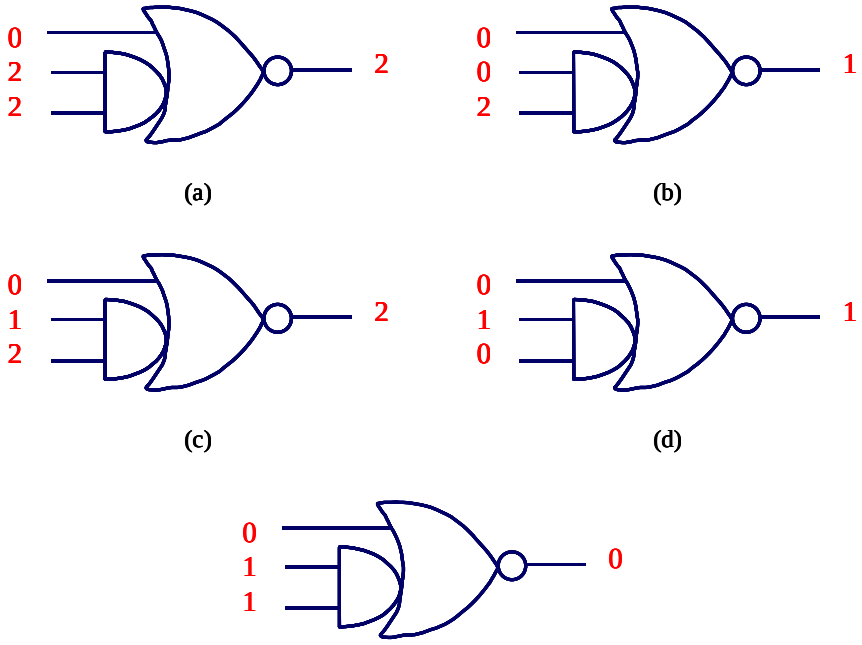


FIGURA A3.18 – Retro-implicação aplicada à SCCG da figura A3.17.

Uma vez encontrado um procedimento para realizar a retro-implicação lógica em portas complexas, deve-se pensar em como estendê-lo à retro-implicação temporal. Similarmente à implicação temporal para diante, pode-se compor tal procedimento assumindo-se os seguintes passos:

1. Retro-implicação lógica;
2. Cálculo dos limites de atraso de primeira ordem e deteção de conflito.

O primeiro passo já foi descrito nos parágrafos anteriores. O segundo passo pode ainda ser subdividido em:

1. Determinar o atraso da porta sob o cubo de entrada *k* selecionado (i.e., sob o assinalamento escolhido para as entradas da porta)
2. A partir dos limites inferior e superior da saída, calcular os limites inferior e superior de primeira ordem para cada entrada da porta usando as seguintes fórmulas: $l_o' = l_o - d(k_{max})$ e $u_o' = u_o - d(k_{min})$, onde $d(k_{max})$ e $d(k_{min})$ correspondem ao máximo e mínimo atrasos para o cubo *k* assinalado à entrada da porta, respectivamente.
3. Para cada entrada i da porta, calcular a interseção entre os limites inferior e superior de atraso da entrada e os limites inferior e superior de primeira ordem calculados no passo anterior.

No passo 3, um conflito é detetado se $l_o' < l_i$ ou se $u_o' < u_i$. Note que pode ocorrer conflito ao se verificar a entrada de uma porta (passo 3 anterior) ainda que a mesma apresente valor lógico 2. Neste caso, seria um conflito temporal.

Uma vez propostos procedimentos que realizam a implicação para diante e a retro-implicação sobre portas complexas, o procedimento de geração de teste temporal foi suficientemente generalizado de modo a permitir a análise de circuitos que contenham portas complexas.

É importante notar que o método proposto para realizar o cálculo temporal de três valores sobre portas complexas não irá representar significativo aumento no tempo total de

execução uma vez que, por razões tecnológicas, os projetistas costumam limitar a 4 o número de transistores em série nas portas CMOS.

A3.5 Conclusão e Perspectivas Futuras

O tema central desta tese foi a análise de *timing* funcional (*functional timing analysis* - FTA) de circuitos combinacionais, com ênfase na aplicabilidade dos modelos e algoritmos de FTA a circuitos que contenham portas complexas. As contribuições desta tese dizem respeito tanto à teoria geral de FTA quanto à FTA com portas complexas.

Como contribuição fundamental, foi apresentada um ampla e sistemática revisão sobre modelos e algoritmos de análise de *timing*. Com o intuito de facilitar a compreensão dos diversos aspectos da FTA, foi proposta uma nova taxonomia para a classificação dos algoritmos de FTA. De acordo com esta taxonomia, os algoritmos de FTA podem ser classificados pelo número de caminhos tratados simultaneamente, quando dos testes de sensibilização (sensibilização individual de caminhos, sensibilização concorrente de caminhos ou abordagem mista), pelo método usado para determinar se as condições de sensibilização são satisfeitas ou não (baseado em ATPG, baseado em solvabilidade – SAT, outros) e pelo critério de sensibilização usado.

Ainda com relação à teoria de FTA, foi especulado que os circuitos síncronos implementados com o uso da tecnologia CMOS estado-da-arte podem apresentar um comportamento assíncrono impossível de ser apropriadamente capturado pelo modelo de atraso de transição. Isto significa que, a fim de evitar uma subestimativa do atraso do circuito, seria preferível calcular o atraso para seqüências de vetores do que para pares de vetores. Por outro lado, na ausência de métodos eficientes capazes de utilizar o modelo de atraso para seqüências de vetores, o modelo de atraso flutuante seria o único capaz de fornecer uma estimativa segura para o atraso dos circuitos. Esta argumentação também justifica o uso da análise de *timing* funcional, uma vez que esta baseia-se essencialmente no modelo de atraso flutuante.

Considerando-se os algoritmos de FTA, foram realizados experimentos com métodos de enumeração de caminhos que confirmaram a seriedade do chamado “problema de explosão de caminho”. Em função deste problema, os métodos que usam sensibilização individual de caminhos foram abandonados, cedendo lugar para os métodos que usam sensibilização concorrente.

No que se refere à FTA com portas complexas, escolheu-se o algoritmo TrueD-F, proposto por Devadas et al., para servir de base para um algoritmo de sensibilização concorrente baseado em ATPG e que seja capaz de tratar circuitos que contenham portas complexas, sem requerer macroexpansão. Esta escolha foi motivada pela existência de inúmeras técnicas de aceleração de algoritmos de ATPG. Outro argumento que sustenta a escolha de um algoritmo baseado em ATPG advém dos problemas de desempenho apresentados pelos algoritmos baseados em SAT, quando modelos de atraso mais realistas são adotados.

A contribuição mais importante desta tese para a análise de *timing* funcional concerne a extensão do cálculo temporal para o caso de portas complexas. No contexto do algoritmo TrueD-F, o cálculo temporal consiste em um conjunto de regras que são usadas para calcular os valores lógicos e os limites inferior e superior dos atrasos nas saídas das portas. De acordo com o método proposto, o cálculo dos valores temporais de uma porta complexa é levado a

cabo por um procedimento recursivo que avalia cada subárvore, iniciando pelas folhas. Para um máximo de 4 transistores em série na rede PMOS e na rede NMOS, o número máximo de subárvores que qualquer SCCG pode apresentar é 11, incluindo a subárvore raiz. Considerando-se que a avaliação de uma única subárvore consiste no cálculo do valor lógico e dos intervalos de atraso, pode-se esperar que, operando-se diretamente sobre portas complexas, os tempos de execução serão, na pior das hipótese, equivalentes aos tempos de execução para um método que faça uso de macroexpansão. Entretanto, vale mencionar que a aplicação de um algoritmo baseado em ATPG a uma rede macroexpandida pode levar a um aumento significativo no tempo de execução, uma vez que haverá forte tendência de aumento no número de linhas do circuito, e portanto, aumento no número de linhas que devem ser justificadas.

Outras contribuições importantes para a FTA com portas complexas resultam da investigação de modelos computacionais de atraso aplicáveis ao caso das portas complexas. A tabela de equivalência modo flutuante/modo de transição permite a identificação dos pares de vetores contidos num dado vetor flutuante. Este tipo de tabela também conduziu à identificação dos três modelos computacionais de atraso de portas válidos para o modo flutuante, quais sejam: modelo de atraso de vetor, modelo de atraso de subida/descida e modelo de atraso único. Apesar de ser o mais preciso, o modelo de atraso de vetor requer 2^n valores de atraso (ou pares de atrasos) por porta, onde n é o número de entradas da porta. Considerando-se SCCGs com não mais do que 4 transistores em série, o valor máximo para n é 16, produzindo um total de 2^{16}= 65536 vetores flutuantes. Por outro lado, o modelo de subida/descida separa os vetores flutuantes em dois grupos, o que reduz para dois valores de atraso (ou dois pares) por porta, sem deixar de garantir um limite superior para o atraso da porta.

A identificação dos pares de vetores contidos num dado vetor flutuante é de grande interesse quando se deseja caracterizar o atraso das portas. No caso de se utilizar simulação elétrica, apenas os pares de vetores capazes de provocar uma transição na saída da porta precisam ser simulados. Para o caso de caracterização por meio de formulação analítica, a mesma informação pode ser usada para se adaptar as fórmulas de modo a cobrir apenas os pares de vetores que são de interesse. Entretanto, a tarefa de identificação dos pares de vetores capazes de provocar transição na saída de uma porta é, *per se*, uma tarefa difícil que demanda mais investigação. Por outro lado, uma conclusão muito importante que pode ser derivada da tabela de equivalência é que o formato de atraso pino-a-pino, normalmente usado na caracterização de células de bibliotecas standard-cells, conduz a uma subestimativa do atraso das portas, uma vez que é capaz de considerar somente uma parte dos vetores flutuantes que podem causar transição na saída das portas.

Finalmente, a partir da tabela de equivalência modo flutuante/modo de transição, foi possível introduzir o conceito de atraso flutuante de um cubo, generalizando assim o modelo de atraso de vetor.

A implicação temporal para diante para portas complexas faz uso de um procedimento que calcula o valor temporal das portas. Conforme já argumentado, espera-se que a complexidade de execução de tal procedimetno seja, no máximo, equivalente àquela resultante de se operar sobre circuitos macroexpandidos. A retro-implicação temporal com portas complexas, por seu turno, não pode ser derivada diretamente da retro-implicação com portas simples porque os conceitos de valor controlante/valor não-controlante não pode ser aplicado diretamente sobre as portas complexas. Assim, o método proposto consiste em usar o mesmo procedimento da implicação temporal para diante, porém seguindo a filosofia do algoritmo PODEM, ou seja, usando simulação de cubos sobre cada porta. O tempo de

execução para a simulação de cubos é, no pior caso, proporcional ao espaço das entradas da porta. Por exemplo, o espaço das entradas de uma SCCG com 4 transistores em série em cada uma das redes é de, no máximo, 2^{16}= 65536 vetores flutuantes. Obviamente, isto afeta o tempo de execução do algoritmo de FTA como um todo.

Esta tese mostrou que é possível realizar FTA de circuitos que contenham portas complexas usando a abordagem baseada em ATPG, com sensibilização concorrente de caminhos, sem o uso de macroexpansão. A principal vantagem de tal abordagem é a potencial redução no número de linhas do circuito que precisam ser justificadas, em comparação com a abordagem que usa macroexpansão. Apesar disso, a retro-implicação tende a ser significativamente mais complexa sobretudo nos casos em que houver a presença de portas complexas com significativo número de entradas. Este efeito pode ser minimizado pelo uso de métricas de testabilidade para guiar a exploração do espaço de entradas.

Finalmente, a extensão proposta para o cálculo temporal, juntamente com os modelos de atraso válidos para o modo flutuante, formam um conjunto de regras de macromodelamento que abrem a possibilidade para se realizar uma “computação hierárquica do atraso flutuante” de circuitos. Esta é a conseqüência do conceito de porta complexa constituir uma representação genérica de qualquer porção possível de circuito CMOS.

O desempenho da abordagem proposta depende grandemente da habilidade do algoritmo em tomar decisões acertadas. Tal qualidade pode ser incorporada através do uso de medidas de testabilidade, conforme feito pelos algoritmos PODEM e FAN. Assim, cálculos de testabilidade para portas complexas constituem um ponto a ser abordado no futuro próximo.

Os estudos de modelos computacionais de atraso válidos ajudou a preencher a lacuna que existe entre modelos físicos e modelos computacionais de atraso. Porém, há ainda diversos pontos que necessitam de maior investigação. Um destes corresponde à identificação automática dos pares de vetores responsáveis por transições nas saídas de uma dada porta. Outro ponto importante é o próprio método de caracterização do atraso, seja pelo uso de simulação elétrica, seja pelo uso/desenvolvimento de modelos analíticos. Um terceiro ponto diz respeito ao máximo número de entradas para as portas complexas, fator este que influencia diretamente o desempenho do método de FTA proposto: um número elevado de entradas por porta pode inviabilizar o uso do modelo de atraso de vetor, o qual tende a ser o mais preciso. A limitação do número de transistores em série para 4 é uma estratégia normalmente adotada pelos projetistas para garantir o desempenho elétrico do circuito. Mas ainda com tal limitação, há 3503 possibilidades de SCCGs. Um trabalho útil consiste em investigar um limite prático no número de componentes das bibliotecas (virtual) de SCCGs, com base no desempenho das ferramentas de mapeamento tecnológico do estado-da-arte.

A abordagem proposta deve ser comparada com a abordagem baseada em macroexpansão, em termos de tempo de execução e precisão, para certificar-se que a redução no número de linhas a serem justificadas se sobrepõe ao aumento de complexidade associado aos procedimentos de implicação aplicados a portas complexas.

Outro aspecto a ser investigado é o desempenho e a precisão de uma computação hierárquica do atraso flutuante. Em uma primeira aproximação, pode-se esperar que o uso do modo hierárquico permita a análise de circuitos com complexidade de uma a duas ordens de grandeza maior do que no modo plano. Porém, é necessário investigar a influência dos modelos computacionais de atraso de porta sobre a estimativa de atraso resultante no modo hierárquico.

Por fim, um trabalho futuro de grande envergadura consiste em estabelecer uma ampla comparação entre a abordagem proposta, que é baseada em ATPG, e alguma abordagem baseada em SAT. Neste trabalho, uma dificuldade básica reside em se escolher um conjunto de modelos computacionais de atraso que sejam válidos para ambas abordagens e ao mesmo tempo propiciem estimativas de atraso com precisão aceitável. Outra dificuldade reside em escolher algum algoritmo de FTA (ou ferramenta) baseada em SAT que seja capaz de tratar portas complexas.

References

[ABR90] ABRAMOVICI, M.; BREUER, M.; FRIEDMAN, A. **Digital Systems Testing and Testable Design**. Piscataway, NJ: IEEE Press, 1990. 652p.

[ASH95] ASHAR, Pranav; MALIK, Sharad. Functional Timing Analysis Using ATPG. **IEEE Transactions on Computed-Aided Design of Integrated Circuits and Systems**, Los Alamitos, California, v.14, n.8, p.1025-1030, August 1995.

[AUG89] AUGUSTIN, L. **An Algebra of Waveforms**. Stanford, California: Computer Systems Laboratory, University of Stanford, Stanford, 1989. Technical Report.

[AUV90] AUVERGNE, D.; AZEMARD, N.; DESCHACHT, D.; ROBERT, M. Input Waveform Slope Effects in CMOS Delays. **IEEE Journal of Solid-State Circuits**, Piscataway, NJ, CA, v.25, n.6, p.1588-1590, Dec. 1990.

[BAU88] BAUER, R. et al. XPSim: A MOS VLSI Simulator. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1988. Santa Clara, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1988. p.66-69.

[BEN87] BENKOSKI, J. et al. Efficient Algorithms for Solving the False path Problem in Timing Verification In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1987. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1987. p.44-47.

[BEN90] BENKOSKI, J.; VAN DEN MEERSCH, E.; CLAESEN, L.; DE MAN, H. Timing Verification Using Statically Sensitizable Paths. **IEEE Transactions on CAD of Integrated Circuits and Systems**, Los Alamitos, California, v.9, n.10, p.1073-1084, Oct. 1990.

[BEN91] BENKOSKI, J.; STEWART, R. TATOO: An industrial Timing Analyzer with False Path Elimination and Test Pattern Generation. In: EUROPEAN DESIGN AUTOMATION CONFERENCE, 1991. Los Alamitos, California: IEEE Computer Society, 1991. p.256-260.

[BER91] BERGAMASCHI, R. The Effects of False Paths in High-Level Synthesis. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1991, Santa Clara, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1991. p.80-83.

[BRA88] BRAND, D.; IYENGAR, V. Timing Analysis Using Functional Analysis. **IEEE Transactions on Computers**, Piscataway, NJ, v.37, n.10, p.1309-1314, October 1988

[CAD99] CADABRA. **CLASSIC-SC Automated Transistor Layout Tool**. Available at: <http://www.cadabradesign.com>. Visited on September 24, 1999.

[CHA75] CHAWLA, B. R.; GUMMEL, H. K.; KOZAK, P. MOTIS – An MOS Timing Simulator. **IEEE Transactions on Circuits and Systems**, Los Alamitos, California, v.CAS-22, n.12, p.901-909, December 1975.

[CHA93] CHANG, Hoon; ABRAHAM, Jacob A. VIPER: An Efficient Vigorously Sensitizable Path Extractor. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 30., 1993, Dallas, Texas. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1993. p.112-117.

[CHA96] CHANDRAMOULI, V.; SAKALLAH, Karem A. Modeling the Effects of Temporal Proximity of Input Transitions on Gate Propagation Delay and Transition Time. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 33., 1996, Las Vegas, Nevada. **Proceedings...** Los Alamitos, California: IEEE Computer Press, 1996. p.617-622.

[CHE91] CHEN, H.-C.; DU, D. Path Sensitization in Critical Path Problem. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1991, Santa Clara, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1991. p.208-211.

[CHE93] CHEN, H.-C.; DU, D. Path Sensitization in Critical Path Problem **IEEE Transactions on CAD of Integrated Circuits and Systems**, Los Alamitos, California, v.12, n.2, p.196-207, February 1993.

[CHE93a] CHEN, H.-C.; DU, D. Critical Path Selection for Performance Optimization **IEEE Transactions on CAD of Integrated Circuits and Systems**, Los Alamitos, California, v.12, n.2, p.185-195, February 1993.

[CHE94] CHENG, S.; CHEN, H.-C.; DU, D.; LIM, A. The Role of Long and Short Paths in Circuit Performance Optimization **IEEE Transactions on CAD of Integrated Circuits and Systems**, Los Alamitos, California, v.13, n.7, p.857-864, July 1994.

[CHS93] CHENG, S.; CHEN, H.-C.; HSU, Y.-C.; DU, D. A Path Sensitization Approach to area Reduction In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN: VLSI IN COMPUTERS AND PROCESSORS, 1993, Cambridge (Mas.). **Proceedings...** Los Alamitos, California: IEEE, 1993. p.73-76.

[COR90] CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L. **Introduction to Algorithms.** Cambridge: MIT, 1990.

[DAG96] DAGA, J.-M.; TURGIS, S.; AUVERGNE, D. Design Oriented Standard Cell Delay Modeling. In: INTERNATIONAL WORKSHOP ON POWER AND TIMING MODELING, OPTIMIZATION AND SIMULATION, PATMOS, 1996. Bologna, Italy. **Proceedings...** Bologna, Italy: Pitagora Editrice Bologna, 1996. p.265-274.

[DAG99] DAGA, J.-M.; AUVERGNE, D. A Comprehensive Delay Macro Modeling for Submicrometer CMOS Logics. **IEEE Journal of Solid-State Circuits**, Piscataway, NJ, v.34, n.1, p.42-55, Jan. 1999.

[DES88] DESCHACHT, D.; ROBERT, M.; AUVERGNE, D. Explicit Formulation of Delays in CMOS Data Paths. **IEEE Journal of Solid-State Circuits**, Piscataway, NJ, v.23, n.5, p.1257-1264, October 1988.

[DET87] DETJENS, E.; GANNOT, G.; RUDELL, R. L. Technology Mapping in MIS. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1987. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1987. p.116-119.

[DEV91] DEVADAS, S.; KEUTZER, K.; MALIK, S. Delay Computation in Combinational Logic Circuits: Theory and Algorithms. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1991, Santa Clara, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1991. p.176-179.

[DEV92] DEVADAS, S.; KEUTZER, K.; MALIK, S.; WANG, A. Certified Timing Verification and the Transition Delay of a Logic Circuit. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 29., 1992, Anaheim, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1992. p.549-555.

[DEV93] DEVADAS, S.; KEUTZER, K.; MALIK, S. Computation of Floating Mode Delay in Combinational Circuits: Theory and Algorithms. **IEEE Transactions on Computed-Aided Design of Integrated Circuits and Systems**, Los Alamitos, California, v.12, n.12, p.1913-1923, Dec. 1993.

[DEV93a] DEVADAS, S.; KEUTZER, K.; MALIK, S.; WANG, A. Computation of Floating Mode Delay in Combinational Circuits: Practice and Implementation. **IEEE Transactions on Computed-Aided Design of Integrated Circuits and Systems**, Los Alamitos, California, v.12, n.12, p.1924-1936, December 1993.

[DEV94] DEVADAS, S.; GHOSH, A.; KEUTZER, K. **Logic Synthesis.** New York: McGraw-Hill, 1994. 404p.

[DEV94a] DEVADAS, S.; KEUTZER, K.; MALIK, S.; WANG, A. Certified Timing Verification and the Transition Delay of a Logic Circuit. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, Los Alamitos, California, v.2, n.3, p.333-342, September 1994.

[DEV94b] DEVADAS, S.; KEUTZER, K.; MALIK, S.; WANG, A Event Suppression: Improving the Efficiency of Timing Simulation for Synchronous Digital Circuits. **IEEE Transactions on Computed-Aided Design of Integrated Circuits and Systems**, Los Alamitos, California, v.13, n.6, p.814-822, June 1994.

[DU89] DU, David H. C.; YEN, Steve H. C.; GHANTA, S. On the General False Path Problem in Timing Analysis In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 26., 1989, Las Vegas, Nevada. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1989. p. 555-560.

[EIJ94] EIJNDHOVEN, J.T.J. van. CMOS cell generation for Logic Synthesis. In: INTERNATIONAL CONFERENCE ON ASICS, ASICON, 1994, Beijing, China. **Proceedings...** Beijing, China: Electr. Ind. Publishing House, 1994. p. 75-78.

[ELM48] ELMORE, W. C. The Transient Response of Damped Linear Networks with Particular Regard to Wide-Band Amplifiers. **Journal of Applied Physics**, New York, v.19, n.1, p.55-63, Jan. 1948.

[FOR97] FORZAN, C.; FRANZINI, B.; GUARDIANI, C. Accurate and Efficient Macromodel of Submicron Digital Standard Cells. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 34., 1997, Anaheim, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1997. p.633-637.

[GOE81] GOEL, Prabhakar. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. **IEEE Transactions on Computers**, Piscataway, NJ, v.C-30, n.3, p.215-222, Mar. 1981.

[GAJ99] GAJSKI, D. **Principles of Logic Design.** New Jersey: Prentice Hall, 1999. 447p.

[GÜN98] GÜNTZEL, José L.; PINTO, Ana Cristina M.; REIS, Ricardo A. L. Improving Path Enumeration Accuracy by Considering Different Fall and Rise Gate Delays. In: WORKSHOP IBERCHIP 4., 1998, Mar del Plata, Argentina. **Memorias...** Buenos Aires: IBERCHIP/Universidad Nacional de La Plata, 1998. p.91-100.

[GÜN98a] GÜNTZEL, José Luís; PINTO, Ana Cristina M.; REIS, Ricardo. Considering Different Fall and Rise Gate Delays in Path Enumeration. In: UFRGS MICROELECTRONICS SEMINAR, 13., 1998, Bento Gonçalves, Brazil. **Proceedings...** Porto Alegre, Brazil: UFRGS, 1998. 196p. p.59-65.

[GÜN98b] GÜNTZEL, José L. et al. An Improved Path Enumeration Method Considering Different Fall and Rise Gate Delays. In: BRAZILIAN SYMPOSIUM ON INTEGRATED CIRCUIT DESIGN, SBCCI, 11., 1998, Búzios, Brazil. **Proceedings...** Los Alamitos, California: IEEE Computer Society, 1998. p.208-211.

[GÜN99] GÜNTZEL, José L. et al. Path Enumeration Algorithms for Timing Analysis of Digital Circuits. In: WORKSHOP IBERCHIP, 5., 1999, Lima, Peru. **Memorias...** Lima, Peru: IBERCHIP/Pontificia Universidad Católica del Perú, 1999. p.334-341.

[GÜN99a] GÜNTZEL, José Luís; DALL PIZZOL, Guilherme M.; REIS, Ricardo. Exploiting Circuit Regularity in Functional Timing Analysis. In: UFRGS MICROELECTRONICS SEMINAR, 14., 1999, Pelotas, Brazil. **Proceedings...** Porto Alegre, Brazil: UFRGS, 1999. 175p. p.77-82.

[GUR97] GURUSWAMY, M. et al**.** Cellerity: A Fully Automatic Layout Synthesis System for Standard Cell Libraries. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 34., 1997, Anaheim, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1997. p.327

[HIR98] HIRATA, A.; ONODERA, H.; TAMARU, K. Proposal of a Timing Model for CMOS Logic Gates Driving a CRC π Load. In: ACM/IEEE INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1998, Santa Clara, California. **Proceedings…** Los Alamitos, California: IEEE Computer Society Press, 1998. p.537-544.

[HIT82] HITCHCOCK, R. B. Timing Verification and the Timing Analysis Program. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 19., 1982, Las Vegas, Nevada. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1982. p.594-604.

[HOR84] HOROWITZ, M. **Timing Models for MOS Circuits**. Stanford, California: Stanford University, 1984. 120p. Ph.D. Thesis.

[HRA78] HRAPCENKO, V. Depth and Delay in a Network. **Soviet Math. Dokl.**,[S.l.], v.19, n.4, p.1006-1009, 1978.

[HUA94] HUANG, Shiang-Tang; PARNG, Tai-Ming; SHYU; Jyou-Min Timed Boolean Calculus and Its Applications in Timing Analysis. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, Los Alamitos, California, v.13, n.7, p.875-883, July 1994.

[HSU98] HSU, Y.-C.; CHEN, H.-C.; SUN, S.; DU, D. Timing Analysis of Combinational Circuits Containing Complex Gates. In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN: VLSI IN COMPUTERS AND PROCESSORS, 1998, Austin, Texas. **Proceedings...** Los Alamitos, California: IEEE, 1998. p.407-412

[JOU87] JOUPPI, Norman P. Timing Analysis and Performance Improvement of MOS VLSI Designs. **IEEE Transactions on Computer-Aided Design**, Los Alamitos, California, v.CAD-6, n.4, p.650-665, July 1987.

[JU91] JU, Y.-C.; SALEH, R. Incremental Techniques for the Identification of Statically Sensitizable Critical Paths. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 28., 1991, San Francisco, California. **Proceedings...** New York: IEEE, 1991. p.541-546.

[KEU91] KEUTZER, K.; MALIK, S.; SALDANHA, A. Is Redundancy Necessary to Reduce Delay? **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, Los Alamitos, California, v.10, n.4, p.427-435, April 1991.

[KIM86] KIM, Y. H. **ELOGIC: A Relaxation-Based Switch-Level Simulation Technique**. Berkeley, California: University of California, Department of Electrical Engineering and Computer Sciences, 1986. 39p. (UCB/ERL M86/2).

[KIM98] KIM, Juho; DU, David H.C. Performance Optimization by Gate Sizing and Path Sensitization. **IEEE Transactions on CAD of Integrated Circuits and Systems**, Los Alamitos, California, v.17, n.5, p.459-462, May 1998.

[KIR66] KIRKPATRICK, T. W.; CLARCK, N. Pert as an Aid to Logic Design. **IBM Journal of Research and Development**, Armonk, v.10, n.2, p.135-141, March 1966.

[KUK97] KUKIMOTO, Y. et al. Approximate Timing Analysis of Combinational Circuits under the XBD0 Model. In: ACM/IEEE INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1997, San Jose, California **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1997. p.176-181.

[KUK97a] KUKIMOTO, Y.; BRAYTON, R. Removing False Paths from Combinational Modules. In: ACM/IEEE INTERNATIONAL WORKSHOP ON TIMING ISSUES IN SPECIFICATION AND SYNTHESIS OF DIGITAL SYSTEMS, TAU, 1997. **Proceedings...** [S.l.:s.n.], 1997.

[LAM93] LAM, W.; BRAYTON, R.; SANGIOVANNI-VINCENTELLI, A. Circuit Delay Models and Their Exact Computation Using Timed Boolean Functions. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 30., 1993, Dallas, Texas. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1993. p.128-134.

[LAM94] LAM, W.; BRAYTON, R. **Timed Boolean Functions:** A Unified Formalism for Exact Timing Analysis. Norwell, MA: Kluwer Academic Publishers, 1994. 273p.

[LAR92] LARRABEE, T. Test Pattern Generation Using Boolean Satisfiability. **IEEE Transactions on CAD of Integrated Circuits and Systems**, Los Alamitos, California, v.11, n.1, p.4-15, Jan. 1992.

[LEL82] LELARASMEE, E.; SANGIOVANNI-VICENTELLI, A. RELAX: A New Circuit Simulator for Large Scale MOS Integrated Circuits. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 19., 1982, Las Vegas, Nevada. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1982. p.682-690.

[LIM99] LIMA, Fernanda Gusmão de. **Projeto com Matrizes de Células Lógicas Programáveis.** Porto Alegre: Instituto de Informática da UFRGS, 1999. 123p. Dissertação de Mestrado.

[LI89] LI, W.; REDDY, S.; SAHNI, S. On Path Selection in Combinational Logic Circuits. **IEEE Transactions on CAD of Integrated Circuits and Systems**, Los Alamitos, California, v.8, n.1, p.56-63, Jan. 1989.

[LIN94] LIN, H.-R.; HWANG, T-T. Dynamical Identification of Critical Paths for Iterative Gate Sizing. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1994, Santa Clara , California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1994. p.481-484.

[LIN95] LIN, H.-R.; HWANG, T-T. Power Reduction by Gate Sizing with Path-Oriented Slack Calculation. In: ASIAN AND SOUTHERN PACIFIC DESIGN AUTOMATION CONFERENCE, 1995, August Chiba, Japan. **Proceedings...** [S.l.]: SIGDA, 1995. p.7-12.

[LOP80] LOPEZ, A.D.; LAW, H.S. A Dense Gate Matrix Layout Method for MOS VLSI. **IEEE Transactions on Electron Devices**, New York, v.ED-27, n.8 p.1671-1675, Aug. 1980.

[MCG89] MCGEER, P.; BRAYTON, R. Efficient Algorithms for Computing the Longest Viable Path in a Combinational Circuit In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 26., 1989, Las Vegas, Nevada. **Proceedings...** Los Alamitos, California: IEEE Computer Press, 1989. p.561-567.

[MCG91] MCGEER, P.; BRAYTON, R. **Integrating Functional and Temporal Domains in Logic Design**: The False Path Problem and its Implications. Norwell, MA: Kluwer Academic Publishers, 1991. 212p.

[MCG91a] MCGEER, P. et al. Timing Analysis and Path Delay-Fault Test Generation using Path-Recursive Functions. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1991, Santa Clara, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1991. p.180-183.

[MCG93] MCGEER, P. et al. Delay Models and Exact Timing Analysis. In: SASAO, T. (Ed.). **Logic Synthesis and Optimization**. Norwell, MA: Kluwer Academic Publishers, 1993. p.167-189.

[MOR94] MORAES, Fernando. **Synthèse Topologique de Macro-Cellules en Technologie CMOS.** Montpellier, France: Université Montpellier II, 1994. Thèse de Doctorat.

[MOR97] MORAES, F.; REIS, R.; LIMA F. An Efficient Layout Style for Three-Metal CMOS Macro-Cells. In: REIS, R.; CLAESEN, L. (Ed.). **VLSI**: Integrated Systems on Silicon. London: Chapman & Hall, 1997. p.415-426.

[NAG75] NAGEL, W. **SPICE2, A Computer Program to Simulate Semiconductor Circuits**. Berkeley, California: University of California, Department of Electrical Engineering and Computer Sciences, 1975. 63p. (UCB/ERL M75/520).

[NEM98] NEMANI, M.; NAJM, F. Delay Estimation of VLSI Circuits from a High-Level View In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 35., 1998, San Francisco, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1998. p.591-594.

[OUS85] OUSTERHOUT, John K. A Switch-Level Timing Verifier for Digital MOS VLSI, **IEEE Transactions on Computer-Aided Design**, Los Alamitos, California, v. CAD-4, n. 3, p.336-349, July 1985.

[PER89] PERREMANS, S.; CLAESEN, L.; DE MAN, H. Static Timing Analysis of Dynamically Sensitizable Paths. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 26., 1989, Las Vegas, Nevada. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1989. p.568-573.

[PES94] PESET LLOPIS, R. Exact Path Sensitization in Timing Analysis In: EUROPEAN DESIGN AUTOMATION CONFERENCE, 1994, Grenoble. **Proceedings...** Los Alamitos: IEEE, 1994. p.380-385.

[PIN98] PINTO, Ana Cristina M.; GÜNTZEL, José Luís; REIS, Ricardo. Performance Evaluation of the sgd and spgd Path Enumeration Methods. In: UFRGS MICROELECTRONICS SEMINAR, 13., 1998, Bento Gonçalves, Brazil. **Proceedings...** Porto Alegre, Brazil: UFRGS, 1998. 196p. p.67-72.

[PRA00] PRADO, A.; LUBASZEWSKI, M.; REIS, A. Identifying Stuck-at Faults with Vertex Precedent BDDs. In: WORKSHOP IBERCHIP, 6., 2000, São Paulo, Brazil. **Anais...** São Paulo, Brazil: CTI/CYTED, 2000. 196p. p.235-244.

[RAB96] RABAEY, Jan. M. **Digital Integrated Circuits:** a Design Perspective. Upper Saddle River, New Jersey: Prentice Hall, 1996. chapter 4, p.210-222.

[RAT94] RATZLAFF, C. L.; PILLAGE, L. T. RICE: Rapid Interconnect Circuit Evaluation Using AWE. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, Los Alamitos, California, v.13, n.6, p.763-776, June 1994.

[RIE96] RIERA i BABURÉS, Jordi. **Mapatge Tecnològic Orientat a Generadors de Mòduls**. Bellaterra, Espanya: Universitat Autònoma de Barcelona, 1996. Tesi Doctoral.

[REI95] REIS, André I. et al. Associating CMOS Transistors with BDD Arcs for Technology Mapping. **IEE Electronic Letters**, London, v.31, n.14, p.1118-1120, July 1995.

[REI97] REIS, André I. et al. Library Free Technology Mapping. In: REIS, R.; CLAESEN, L. (Ed.). **VLSI**: Integrated Systems on Silicon. London: Chapman & Hall, 1997. p.303-314.

[REI98] REIS, André I. **Assignation Technologique sur Biblioteques Virtuelles de Portes Complexes CMOS**. Montpellier: Université Montpellier II, 1998. 122p. Thèse de Doctorat.

[REI99] REIS, Ricardo A. L. **FUCAS: FUll Custom Layout Synthesis.** Available at: <http://www.inf.ufrgs.br/~reis/rese99.html>. Visited on September 23, 1999.

[ROT66] ROTH, J.P. Diagnosis of Automata Failures: A Calculus and a New Method In: **IBM Journal of Research and Development**, Armonk, v.10, n.6, p.278-291, July1966.

[RUB83] RUBINSTEIN, J.; PENFIELD, P. Jr.; HOROWITZ, M. A. Signal Delay in RC Tree Networks. **IEEE Transactions on Computer-Aided Design**, Los Alamitos, California, v.CAD-2, n.3, p.202-210, July 1985.

[SAK83] SAKURAI, Takayasu Approximation of Wiring Delay in MOSFET LSI. **IEEE Journal of Solid-State Circuits**, Piscataway, NJ, v.SC-18, n.4, p.418-426, August 1983.

[SAK88] SAKURAI, Takayasu. CMOS Inverter Delay and Other Formulas Using α-Power Law MOS Model. In: ACM/IEEE INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1988, Santa Clara, California. **Proceedings…** Los Alamitos, California: IEEE Computer Society Press, 1988. p.74-76.

[SAL94] SALDANHA, A.; BRAYTON, R.; SANGIOVANNI-VICENTELLI, A. Circuit Structure Relations to Redundancy and Delay. **IEEE Transactions on CAD of Integrated Circuits and Systems**, Los Alamitos, California, v.13, n.7, p.875-883, July 1994.

[SEG89] SEGER, Carl-Johan. A Bounded Delay Race Model. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1989, Santa Clara, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1989. p.130-133.

[SEN92] SENTOVICH, E. M. et al. **SIS**: A System for Sequential Circuit Synthesis. Berkeley, California: Electronics Research Laboratory, University of California. May 1992. 45p. (Memorandum No UCB/ERL N92/41)

[SIL93] SILVA, João P.M.; SAKALLAH, Karem. Concurrent Path Sensitization in Timing Analysis. In: THE EUROPEAN CONFERENCE ON DESIGN AUTOMATION WITH THE EUROPEAN EVENT IN ASIC DESIGN, 1993, Paris, France. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1993.

[SIL93a] SILVA, João P.M.; SAKALLAH, Karem. An Analysis of Path Sensitization Criteria. In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN: VLSI IN COMPUTERS AND PROCESSORS, 1993, Cambridge, MA. **Proceedings...** Los Alamitos, California: IEEE, 1993. p.68-72.

[SIL94] SILVA, João P.M.; SAKALLAH, Karem. Efficient and Robust Test Generation-Based Timing Analysis. In: IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, 1994, London. **Proceedings...** Piscataway: IEEE 1994. v.1, p.303-306.

[SIL94a] SILVA, João P.M.; SAKALLAH, Karem. Dynamic Search-Space Pruning Techniques in Path Sensitization. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 31., 1994, San Diego, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1994.

[SIL96] SILVA, João P.M.; SAKALLAH, Karem. GRASP – A New Search Algorithm for Satisfiability. In: IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, 1996, San Jose, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1996. p.220-227.

[SIL98] SILVA, Luís Guerra et al. Realistic Delay Modeling in Satisfiability-Based Timing Analysis. In: IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, 1998. **Proceedings...** Piscataway: IEEE, 1998. v.6, p.215-218.

[SIL99] SILVA, Luís Jorge B. M. G. e. **Models and Algorithms for Timing Analysis of Combinational Circuits.** Lisboa: Instituto Superior Técnico (IST), Universidade Técnica de Lisboa, 1999. 84p. Master Dissertation.

[SUN94] SUN, Shang-Zhi; DU, David H.C.; CHEN, Hsi-Chuan. Efficient Timing Analysis of CMOS Circuits Considering Data Dependent Delays. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1994, Santa Clara, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1994. p.156-159.

[UEB95] UEBEL, Luis Felipe. **Verificação de *Timing* em Circuitos VLSI CMOS Digitais**. Porto Alegre: CPGCC da UFRGS, 1995. 222p. Dissertação de Mestrado.

[UEH81] UEHARA, T.; CLEEMPUT, W.M. Optimal Layout of CMOS Functional Arrays. **IEEE Transactions on Computers**, Piscataway, NJ, v.C-30, n.5, p.305-312, May 1981

[YAL95] YALCIN, H.; HAYES, J. Hierarchical Timing Analysis Using Conditional Delays. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1995, Santa Clara, California. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1995.

[YEN88] YEN, S.; GHANTA, S.; DU, D. A Path Selection Algorithm for Timing Analysis In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 25., 1988. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1988. p.720-723.

[YEN89] YEN, S.; DU, D.; GHANTA, S. Efficient Algorithms for Extracting the K Most Critical Paths in Timing Analysis In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 26., 1989, Las Vegas, Nevada. **Proceedings...** Los Alamitos, California: IEEE Computer Society Press, 1989. p.649-652.

[YEN91] YEN, S.; DU, D.; GHANTA, S. Efficient Algorithms for Extracting the K Most Critical Paths in Timing Analysis. **International Journal of Computer Aided VLSI Design**,[S.l.],v.3, n.2, p.193-215, 1991.

[WIS84] WISTON, Patrick H. **Altificial Intelligence**. 2nd ed. Reading, Massachusetts: Addison-Wesley Publishing Company, 1984. 524p.