Copyright © 1999, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

TIMING ANALYSIS AND OPTIMIZATION FOR HIGH-PERFORMANCE DIGITAL CIRCUITS

by

Yuji Kukimoto

Memorandum No. UCB/ERL M99/42

8 September 1999

TIMING ANALYSIS AND OPTIMIZATION FOR HIGH-PERFORMANCE DIGITAL CIRCUITS

by

Yuji Kukimoto

Memorandum No. UCB/ERL M99/42

8 September 1999

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Timing Analysis and Optimization for High-Performance Digital Circuits

Copyright 1998
by
Yuji Kukimoto

Abstract

Timing Analysis and Optimization for High-Performance Digital Circuits

by

Yuji Kukimoto

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Robert K. Brayton, Chair

Meeting performance constraints in synchronous digital circuits is crucial since a violation may cause an unexpected value to be latched in a memory element. To guarantee the absence of timing violations the performance of a design needs to be estimated accurately and verified against given performance constraints. This delay estimation is called *timing analysis*. The main difficulty of timing analysis for gate-level digital circuits is the existence of false paths. A false path is a topological path of a given circuit along which a signal event never propagates. Since false paths do not contribute to the delay of a circuit, they need to be excluded when the performance of a circuit is estimated.

Although false path detection has been researched extensively in the last decade, it has always been studied in a specific problem, namely arrival time analysis of combinational circuits, in which the arrival times of the outputs of a circuit are estimated given the arrival times of the inputs. The main contribution of the first part of this dissertation is to introduce and present algorithms for a new problem: false-path-aware required time analysis of combinational circuits, in which the required times of the inputs of a circuit are estimated given the required times of the outputs. This problem forms the core of a rich set of novel timing analysis problems arising in hierarchical designs. The applications include false path detection and removal of combinational circuits under unknown surrounding environments and hierarchical false-path-aware timing analysis.

To meet an aggressive performance goal, timing optimization algorithms play a key role in exploring a design space systematically. The second part of the dissertation focuses on timing optimization in a logic synthesis step called *technology mapping*, where a technology-independent circuit is translated to a circuit composed only of gates in a given gate library. Although the complexity of the technology mapping problem for area minimization is well-understood, technology

mapping for delay minimization has been tackled using heuristic approaches without knowing its exact complexity. The contribution of the second part of the dissertation is a linear-time algorithm that solves the problem optimally under a load-independent delay model. There is no need to resort to heuristic approaches since the problem can be solved optimally and efficiently.

Professor Robert K. Brayton
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Acknowledgements

My graduate student life at Berkeley has been supported by many individuals. I would like to thank them all for their help and encouragement.

Bob Brayton has been my advisor since the very first semester. I learned from him how to conduct research independently: everything from the ability to carve out a potential research area to technical writing and presentation skills. He has always been there to listen to my half-baked ideas patiently and help me polish them up by asking insightful questions. The discussions we have had in the past years have always been enjoyable and intellectually stimulating.

I am grateful to Alberto Sangiovanni-Vincentelli and Dorit Hochbaum for their constructive comments to my research as the second and the third members of my dissertation committee. I would also like to thank Jan Rabaey for his feedbacks during my qualification exam.

My interest in computer-aided design started from a research collaboration with Masahiro Fujita when I was a graduate student in Japan. He introduced me to the research field and got me aware of research activity outside Japan. If it had not been for him, I would not have come to Berkeley in the first place.

Some of the research presented in this dissertation is the result of collaborations with my colleagues. I would like to thank Wilsin Gosti and Alex Saldanha for their contributions to Chapter 9. I am indebted to Alex Saldanha and Rick McGeer for various discussions on timing analysis. The result on technology mapping presented in Chapter 10 was obtained with Prashant Sawkar. I have been fortunate to be able to work closely with Eugene Goldberg for the last two years although this research is not included in the dissertation. Some of the results in Chapter 4 were inspired by comments from Daniel Brand. I would like to thank him for generously sharing his idea. I also enjoyed technical discussions with Hakan Yalcin on the topics of shared interest. The research presented in Chapter 8 started from Leon Stok's comment at an SRC meeting.

Stephen Edwards carefully read an early draft of this dissertation and gave me constructive feedbacks. Thanks to Luca Carloni, Wilsin Gosti and Tiziano Villa for comments.

Harry Hsieh, Rajeev Ranjan, Serdar Taşıran and I came to Berkeley in the same time frame and prepared for a preliminary exam together. Since then we have been good friends. I would like to thank them for the friendship.

It has been great pleasure to be part of the VIS team, a joint formal verification project of Berkeley and University of Colorado. Thanks to all the members of the team.

The interactions with the current and past members of the CAD group have been always

stimulating. I would like to thank Adnan Aziz, Dominique Borrione, Premal Buch, Edoardo Charbon, Szu-Tsung Cheng, Ramin Hojati, Adrian Isles, Dirk-Jan Jongeneel, Timothy Kam, Sunil Khatri, Desmond Kirkpatrick, Sriram Krishnan, Andreas Kuehlman, William Lam, Freddy Mang, Rajeev Murgai, Amit Mehrotra, Amit Narayan, Mukul Prasad, Shaz Qadeer, Sriram Rajamani, Marco Sgroi, Tom Shiple, Subarnarekha Sinha, Gitanjali Swamy, Iasson Vassiliou, and Huey-Yih Wang. Special thanks to Yosinori Watanabe, who helped me adjust myself to the new environment here. I also enjoyed discussions we have had recently on technology mapping.

The CAD group has been supported by friendly administrative staff. I am grateful to Flora Oviedo and Brad Krebs for their assistance over the past years.

Murata Overseas Scholarship Foundation provided generous financial support during the first two years at Berkeley. Without their support I would not have thought of applying to a graduate school in the states.

The research presented in the dissertation was supported by SRC DC-324. Their support is greatly acknowledged.

Finally, I would like to thank my parents and my brother for their encouragement.

Chapter 1

Introduction

Performance-oriented design methodology for digital circuits has gained increasing importance in the last decade mainly due to the competitive microprocessor market. This trend is expected to continue in the future to push the current performance envelope to the limit. Designing such a complex digital circuit under tight time-to-market pressure is a daunting task beyond a manual design methodology and naturally requires *computer-aided design and verification* to manage the complexity in a short design cycle. The focus of this dissertation is design automation techniques for performance-oriented designs. Specifically, timing verification and optimization techniques will be studied in detail.

A typical design flow of digital circuits starts from a functional description of a design written in a hardware description language along with other specifications such as area, performance, testability and power dissipation. The initial description can be an abstract behavioral description, in which the behavior of the design is specified without an explicit hardware configuration. This is then translated into a register-transfer-level (RTL) description by mapping it to an actual hardware configuration. Operations in the behavioral description are scheduled on given hardware resources to meet given constraints. This step is called *high-level synthesis*. In the resulting RTL description a set of registers is explicitly declared and data operations between the registers are performed conditionally under a controller. *Logic synthesis* is the step which takes an RTL description and generates a gate-level circuit composed only of gates and storage elements available in an actual technology to be used. This step is followed by *physical design*, where all the components of the gate-level circuit are laid out on a two-dimensional plane to create masks, which are then used to fabricate the circuit on silicon.

Although performance-oriented design and verification methodology needs to be addressed

at every level of the design flow, we will restrict ourselves to techniques applicable to the gate level.

The first part of the dissertation deals with timing verification of gate-level circuits. Once a design is complete, we need to verify if the design meets a given performance constraint. The key step is to estimate the delay characteristics of the circuit. We assume that the delay of each gate in the circuit is pre-characterized and wire delays are extracted from the physical layout of the design. A naive way to estimate the delay of the circuit is to find the topological longest path of the circuit, where the length of a path is defined as the sum of the delays of all the components along the path. Although this gives a conservative estimate of actual delay, accuracy may suffer since the longest topological path may not be able to propagate a signal event due to the interaction of the functionality and the timing of the circuit. A path is called *false* if it is not responsible for delay. To estimate the delay of a circuit accurately we need to find the longest topological path that is not false. Although false path detection has been intensively studied in the last decade, the focus of the research has been in false-path-aware arrival time analysis (functional arrival time analysis), where arrival times at the primary outputs of a gate-level circuit are estimated given arrival times at the primary inputs by considering false paths. However, this problem is only one facet of delay analysis issues related with false paths.

The goal of the first part of the dissertation is to deepen the understanding of false paths by exploring various timing analysis problems other than functional arrival time analysis.

After giving a background of the previous research on timing analysis in Chapter 2, we will introduce in Chapter 3 a new timing analysis problem, *functional required time analysis*, which is an analogue of functional arrival time analysis for required time computation. In functional required time analysis required times at the primary inputs of a circuit are estimated given required times at the primary outputs by taking false paths into account. It will be shown that an existing theory for functional arrival time analysis can be generalized to functional required time analysis. Exact and approximate algorithms for functional required time analysis will be presented, the use of which gives more relaxed required times than those estimated by topological required time analysis unaware of false paths.

Chapter 4 will discuss delay characterization of a combinational module. A combinational module is a portable combinational circuit that can be used under any surrounding environment. An intellectual property (IP) module, if combinational, is an example of such modules. False-path-aware delay characterization of a combinational module plays an important role in timing analysis for IP-based design methodology, where the accurate delay abstraction of an IP module is required without disclosing its internal proprietary details. The main difficulty of this problem is in the fact that the

surrounding environment of the module is unknown, i.e. we cannot assume specific arrival times at the primary inputs of the module. We will show that functional required time analysis studied in Chapter 3 can be used directly for the delay characterization problem. A false-path-aware delay abstraction of a combinational module can be computed without assuming specific arrival time conditions.

Chapter 5 will study hierarchical functional arrival time analysis, where a hierarchical circuit is analyzed in a bottom-up way without flattening the hierarchy. Note that existing state-of-the-art functional arrival time analysis techniques in the literature always assume that a circuit under analysis is flat. Although flattening can always be used to remove a hierarchy, this potentially forces us to analyze a huge circuit at one time thereby making functional arrival time analysis of industrial circuits difficult. Moreover, once the use of IP modules becomes prevalent, flattening will not be an option any more. The hierarchical approach presented in this chapter enables us to perform false-path-aware timing analysis by respecting a given design hierarchy. The analysis starts with the computation of a delay abstraction for each module at the lowest level in a hierarchy. The computed delay abstraction is then used for the analysis at the next higher level. There is no need to analyze the entire circuit at one time. The hierarchical analysis gives as a byproduct incremental analysis capability, which is missing in flat analysis.

Chapter 6 will introduce a new criterion, called *timing-safe replaceability*, for comparing two combinational modules in terms of performance. A module is said to be a timing-safe replacement of another if the former is no slower than the latter under any surrounding environment. This notion allows us to argue the false-path-aware performance of two combinational modules without assuming the surrounding environment in which the modules are embedded.

A new class of false paths for combinational modules will be introduced in Chapter 7. A path in a combinational module is said to be strongly false if it is false under any surrounding environment. This special class of false paths is the only false paths that can be safely assumed to be false under unknown surrounding environments. An algorithm to detect strongly false paths will be given.

Chapter 8 will then present an algorithm to remove strongly false paths from a combinational module without slowing down the module under any environment. The module after the transformation is provably a timing-safe replacement of the original. The resulting module can be analyzed more accurately than the original using topological timing analysis, which is much more efficient than functional timing analysis.

The first part of the dissertation will be concluded by approximate algorithms for flat func-

tional arrival time analysis in Chapter 9. Although flat functional arrival time analysis is well-understood by now, the analysis of industrial circuits with a large number of reconvergences is still CPU-time intensive computation. We will extend an existing functional arrival time analysis technique so that delay estimates can be obtained in less CPU time with possible overestimation. Experimental results will be provided to show that the approximation gives a dramatic reduction in CPU time for benchmark circuits with minor delay overestimation.

The second part of the dissertation is devoted to performance-oriented logic synthesis. Specifically, we will focus on the final step of logic synthesis called *technology mapping*. The goal of technology mapping is to take a technology-independent circuit and to generate a functionally equivalent circuit composed only of gates available in a given gate library so that a given criterion is optimized. The criterion of our choice in the dissertation is performance.

We will first overview the previous results of technology mapping in Chapter 10 and illustrate how the problem is solved by an approximate strategy developed for area-minimal technology mapping. The overall flow of the most popular approach to delay-minimal technology mapping is to partition a given technology-independent network into a forest of trees and to map each tree optimally using dynamic programming. This flow makes sense in area-minimal technology mapping since without the partitioning the problem becomes NP-hard. We will show that the delay-minimal technology mapping can be solved optimally for a general technology-independent network with a DAG structure in time linear in the size of the network under a load-independent delay model. The algorithm is inspired by delay-optimal technology mapping for FPGAs. This implies that the tree decomposition essential in area-optimal mapping is not necessary for delay optimization. The relationship between library-based mapping and FPGA mapping, which was not understood well, will be clarified.

The dissertation will be concluded in Chapter 11. The summary of the dissertation and future directions will be given.

Chapter 2

Preliminaries

This chapter overviews timing issues arising in designs of synchronous digital circuits. After introducing basic terminology on Boolean networks in Section 2.1, we will discuss in Section 2.2 how to realize a synchronization mechanism using memory elements, and show that the underlying assumption of this synchronization mechanism requires that certain timing constraints be met in designs. We will then illustrate in Section 2.3 how these timing constraints are passed to synthesis so that a synthesized circuit is free from a timing violation. Finally, given an actual implementation of a design, one needs to verify whether the implementation meets all given timing constraints. This task is called *timing analysis*, which will be covered in Section 2.4. We will start with classic topological timing analysis and then introduce more accurate analysis called *functional timing analysis*. We will give an overview of functional timing analysis and then summarize the XBD0 analysis, one of the most accurate functional timing analysis techniques known so far, which forms the basis of the first part of the dissertation.

2.1 Boolean Networks

A *Boolean network* [BRSVW87] is an abstract representation of multi-level combinational circuits. It is a directed acyclic graph, where a node without any incoming edge represents a primary input and a node without any outgoing edge represents a primary output. All the other nodes represent intermediate gates. A Boolean function is associated with each intermediate node. There is an edge from a node n_i to a node n_j if the function associated with n_j explicitly depends on n_i.

If there is an edge from a node n_1 to a node n_2, n_1 is a *fanin* of a node n_2 and n_2 is a *fanout* of a node n_1. If there is a directed path from n_1 to n_2 in the Boolean network, n_1 is a *transitive fanin*

of n_2 and n_2 is a *transitive fanout* of n_1.

2.2 Synchronization and Memory Elements

In this dissertation we only consider synchronous circuits, those controlled by a global clock[1].

Any interesting digital system has memory elements that keep track of previous states. These memory elements are synchronously updated by clock signals. There are two main types of memory elements:

1. edge-triggered flip-flops and

2. level-sensitive latches (transparent latches).

An edge-triggered flip-flop has a data input and a data output. It also has a control input connected to an external clock signal. Whenever the control input has a transition from high to low (for falling-edge-triggered flip-flops), the data input value at the time is captured and will be available at the data output until the next falling transition. A level-sensitive latch has the same three terminals as an edge-triggered flip-flop, but its behavior is different. When the control input is high, the data output value tracks the data input value (for active-high level-sensitive latches). Once the control input transitions from high to low, the data input value at the moment is sampled and the data output will be held at the value until the next time the control gets high.

In both types of memory elements, one needs to pay attention to relative timing of the signals so that the correct data input is sampled. Consider a high-to-low transition of the control input. Since the sampling is done by a physical device, the data input value must have been stable long enough before the sampling takes place. This is called a *set-up time constraint*. The minimum amount of the set-up time required depends on the circuit structure of a memory element. Given this constraint, we need to make sure that the data input of the memory element becomes stable early so that there is enough stable time before the actual sampling transition. Furthermore, it is also necessary to hold the data input value even after the sampling transition. This is called a *hold time constraint*. There is a minimum amount of time for the data input to be held after the transition edge so that the correct value is sampled.

[1] It is possible to create a local clock from external clocks and internal signals. This is called *gated clocking* and is used frequently in low-power designs. We will only consider pure external clocking in this dissertation.

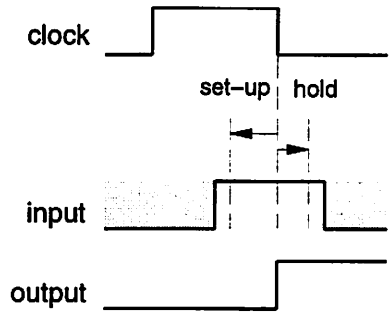

Figure 2.1: Timing Diagram of a Falling-Edge-Triggered Flip-flop

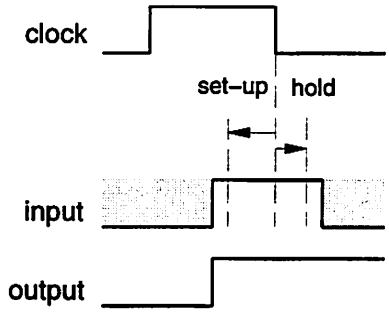

Figure 2.2: Timing Diagram of an Active-High Level-Sensitive Latch

Figures 2.1 and 2.2 illustrate the behaviors of a falling-edge-triggered flip-flop and an active-high level-sensitive latch by timing diagrams respectively.

2.3 Timing Constraints in Synthesis

2.3.1 Impacts of Set-up/Hold Time Constraints on Synthesis

The clock frequency of a digital system is one of the major performance goals set at the beginning of a design project. The reciprocal of the clock frequency is called a *cycle time*, which is the length of one cycle of the clock. The cycle time has a deep impact on the circuit design. As in Section 2.2, we need to meet a set-up time constraint and a hold time constraint for each memory element in order to guarantee correct synchronization. Therefore any signal feeding the data input of a memory element is required to be stable early enough and to maintain its value long enough before and after every clocking edge respectively. We can then assure that every memory element captures a correct value at any point in time. Since the data input signal of a memory element is typically generated by a combinational logic block, the timing constraint is posed directly on the logic block

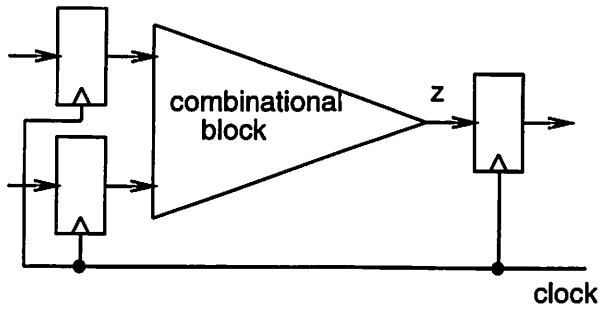

Figure 2.3: Combinational Block between Flip-Flops

feeding the data input.

Consider a simple case where all the memory elements are edge-triggered flip-flops controlled by the same clock signal. Figure 2.3 shows that the data input of an edge-triggered flip-flop is connected to the output of a combinational logic block whose inputs come from the outputs of other flip-flops. Let c be the cycle time of the clock. Suppose that the clock falls from high to low at $t = 0$. The next falling edge takes place at $t = c$. Let $d_{\max}$ be the maximum time for the combinational block to take to get stabilized given all the inputs at $t = 0$. The output of the combinational block, z, gets stable no later than $t = d_{\max}$. Let s be the minimum set-up time required. The set-up time constraint is satisfied if it is satisfied for the worst case, which gives the following inequality.

$$c - d_{\max} \geq s$$

Therefore, the combinational block must be designed so that

$$d_{\max} \leq c - s. \tag{2.1}$$

Note that c and s are constants.

Consider the hold time constraint. Assume that the set-up time constraint is met already. We can then assume that the signal value z for the previous clock cycle is stable by $t = -s$. At $t = 0$, all the flip-flops sample and hold new values. These new values cause the combinational block to re-evaluate its functionality thereby eventually giving a new final value at z. Let $d_{\min}$ be the minimum time for z to take to get destabilized given all the new input values at $t = 0$. $d_{\min}$ is the earliest possible time for z to start changing from its old value. In order to meet the hold time constraint,

$$d_{\min} \geq h \tag{2.2}$$

should hold, where h is the minimum hold time required for the flip-flop and is a constant given the type of the flip-flop.

The two inequalities (2.1) and (2.2) pose constraints on d_{max} and d_{min} of the combinational block. The satisfaction of the inequalities guarantees the absence of a set-up time or a hold time violation[2].

2.3.2 Synthesis under Timing Constraints

Various techniques have been studied in the literature to address synthesis of combinational blocks under timing constraints. Although the constraints (2.1) and (2.2) for the set-up and the hold time constraints respectively are equally important for correct synchronization, the hold time constraint is relatively easier to meet than the set-up time constraint in reality[3]. Therefore, we will focus on the satisfaction of set-up time constraints in the rest of the dissertation.

Timing-driven logic synthesis of a combinational circuit takes as input a timing specification along with a functional specification to generate a circuit satisfactory both in terms of functionality and timing. The timing specification is given as

1. *arrival time* at each primary input and

2. *required time* at each primary output.

Arrival times at primary inputs specify the timing environment under which the circuit is to be used while required times at primary outputs put a constraint on the speed of the circuit.

Suppose that the combinational block in Figure 2.3 needs to be synthesized. Let us see how the set-up time constraint (2.1) is translated into this framework. Since a primary input of the combinational block is fed by the output of a flip-flop, the time when sampling takes place gives arrival time of the primary input. This sampling time is typically known since it is determined by an external clock. In our case we can set arrival time of all the primary inputs to $t = 0$. From (2.1) the signal stable time at the primary output z must be no later than $c - s$, which directly gives the required time at the output.

[2]The extension of this constraint extraction for the case where different edge-triggered flip-flops are controlled by different clock signals is trivial. The use of level-sensitive latches makes this constraint extraction much more complex due to cycle stealing, and is beyond the scope of the dissertation. For details refer to [She93].

[3]A hold time violation can be resolved by inserting active delays. Shenoy *et al.* propose one such technique in [SBSV93].

2.4 Timing Analysis

We have seen so far that combinational logic blocks must be designed so that a set-up time constraint and a hold time constraint are met for each memory element. Since the satisfaction of the timing constraints is crucial in correct functionality, we need to verify whether a combinational block meets the timing constraints once it is synthesized. This section overviews the basics of timing verification.

Consider the problem of estimating d_{max} and d_{min} of a given combinational circuit as accurately as possible. This problem is called *timing analysis* or *timing estimation*. The estimation is essential in verifying whether the circuit is free from set-up time and hold time violations. The exact values of d_{max} and d_{min} are hard to estimate since the timing characteristic of a circuit is determined by complex analog behaviors. Therefore our goal is to estimate the values conservatively so that any timing violation is detected by using the estimated values.

Let d'_{max} be an estimate of d_{max} such that $d_{max} \leq d'_{max}$ is guaranteed, i.e. d'_{max} never underestimates d_{max}. If the set-up time constraint is satisfied using the estimate d'_{max},

$$d'_{max} \leq c - s.$$

Since $d_{max} \leq d'_{max}$,

$$d_{max} \leq d'_{max} \leq c - s.$$

This shows that if there is no set-up time violation using an overestimate d'_{max}, it implies the absence of set-up time violation under the actual d_{max}. Notice that the condition that d_{max} never gets underestimated is the key in establishing this result. Any timing analysis technique must meet this condition to be useful in the context of timing verification. Otherwise timing violation could be overlooked, which defeats the whole purpose of timing verification.

We can obtain a similar result for the estimation of d_{min}. This time we require that d_{min} never get overestimated. With this condition any hold time violation can be detected using an estimate of d_{min}.

Conservative estimation of d_{max} and d_{min} is crucial in guaranteeing that no false positive result is obtained in timing verification. However, if the estimation is far off from the actual value, a timing violation may be detected using the estimate although there is no violation in the real circuit. This is a false negative result and should be avoided. To minimize such false alarms conservative yet accurate estimation is desired.

Although the estimation of minimum delay $d_{\min}$ is important for the timing correctness of a digital system, $d_{\max}$ plays a key role in timing-driven designs. Therefore, we will focus on the estimation of maximum delay $d_{\max}$ in this dissertation.

2.4.1 Gate-level Timing Analysis

The timing characteristic of digital circuits is determined as a result of complex analog behaviors. Circuit simulation such as SPICE directly solves a set of differential equations to compute an output waveform given an input waveform. Although accurate, this approach is computationally expensive and the applicability is limited to small circuits. Furthermore, since the analysis is only valid for a given input waveform, one needs to repeat simulations for all input waveforms to make sure that no timing violation occurs. The exhaustive simulations of all possible input waveforms are not feasible even for moderate-size circuits[4].

This difficulty can be overcome by performing timing analysis at a more abstract level, i.e. the gate level. A circuit under analysis is assumed to be given at the gate level; it is composed of gates connected by wires. We assume that the delay of each gate in the circuit is pre-characterized by circuit simulation and that the maximum delay of each gate is known. Gate-level timing analysis then takes these gate delay values and estimates the maximum delay of the circuit. We also assume that each gate can take its maximum delay simultaneously. Although this may not be possible in reality, the assumption is conservative [5].

2.4.2 Problem Formulation

We formulate the problem of estimating $d_{\max}$ of gate-level combinational circuits.

Let $G = (V, E)$ be a Boolean network. V can be partitioned into three disjoint sets V_{PI}, V_{PO} and V_I. V_{PI} and V_{PO} are the sets of all primary input nodes and primary output nodes respectively. V_I is the set of all intermediate nodes. Recall that each intermediate node corresponds to a gate.

---

[4] If exhaustive simulations are not performed, $d_{\max}$ may be underestimated.

[5] For example, it may not be possible for two gates to take their worst case delays simultaneously since the condition that one gate takes its maximum conflicts with the condition that the other takes its maximum. Recent work by Sivaraman and Strojwas [SS97] addressed this issue and showed that more accurate timing analysis is possible by considering this interdependency of worst-case gate delays explicitly.

The gate-level arrival time analysis problem: Given

1. a Boolean network $G = (V, E)$ representing a combinational circuit,
2. a delay range $[0, d(n)]$ for each intermediate node $n \in V_I$, and
3. arrival time $arr(x_i)$ for each primary input $x_i \in V_{PI}$,

estimate arrival time $arr(z_i)$ for each primary output $z_i \in V_{PO}$.

Notice that the minimum delay for each internal node is assumed to be zero. Since the delay value of a node is not a constant, but a range, the Boolean network under analysis implicitly represents a family of networks, each of which is functionally and structurally identical with different gate delays within the given upper-bounds. Therefore, any analysis must guarantee that a computed arrival time at a primary output is a conservative estimate for all the members of the network family. This property is called *monotone speedup* or *robustness* [MB91].

Although a wire has no delay in this formulation, wire delay can be taken into account by assuming on the wire a buffer whose maximum delay is set to the delay of the wire.

In this formulation, each node has a common delay value from any fanin. In reality delays are different for different fanins. Also, even for the same fanin, the delay can be different between rising and falling transitions. Although it is easy to generalize the subsequent arguments to capture these aspects, we will follow the original formulation for ease of exposition.

2.4.3 Topological Arrival Time Analysis

A Boolean network represents signal dependency between nodes in the network. It is conservative to assume that each node becomes stable only after all its fanin nodes do. Under this assumption one can determine the signal arrival time at an output by detecting the longest path from a primary input node to a primary output node, where the length of a path is defined as the sum of the arrival time at the input and all node delays along the path [Hit82, HSC82]. This is a well-known *longest path problem* on directed acyclic graphs. We can visit nodes in the network in a topological order and determine the signal stable time at node n, $arr(n)$, by

$$arr(n) = d(n) + \max_{m_i \in FANIN(n)} arr(m_i).$$

Since the algorithm runs in time linear in the size of the network, it is applicable to large industrial designs.

2.4.4 False Paths

Although topological analysis gives an efficient solution to the arrival time analysis problem, the quality of the result may suffer. The basic assumption in topological analysis is that each intermediate node becomes stable only after all its fanin nodes become stable. The functionality associated with the node is ignored. This is the same as conservatively assuming that all the paths in the circuit can propagate signal events, and are responsible for delays. This is not necessarily true in real circuits.

For example, consider a two-input AND gate n in a network. Assume that both of its fanins get value zero under an input vector. Since the functionality of the gate is AND, once either fanin becomes stable to zero, the output of the gate will be stabilized to zero after $d(n)$ even if the other fanin has not yet been stabilized. Therefore, any path leading to the late stabilizing fanin or its extension is not responsible for delay under this input vector.

As in this example, it may not be possible to propagate a signal event along a path if any signal event along the path is blocked on its way by other paths. We call such a path *false*. The existence of false paths in a circuit makes accurate estimation of arrival time difficult for topological analysis. For example, if the topological longest path is false under all input vectors, the arrival time estimated by topological analysis overestimates an actual arrival time. The detection of false paths is only possible by taking into account the Boolean nature of the circuit. Since there are practical circuits known to have long false paths (e.g. carry-skip adders), arrival time analysis with the capability of false path detection, termed *functional arrival time analysis*, is critical for accurate delay estimation.

2.4.5 Functional Arrival Time Analysis

Although the existence of false paths has been known for a long time, it was not until the late 80's that research on systematic detection of false paths started. Initially the research focus was to identify a correct condition for determining whether a given path is responsible for delay. Various path sensitization conditions have been proposed in the literature, and the relative accuracy between the conditions has been analyzed. Well-accepted path sensitization conditions such as viability [MB91] and the floating-mode condition [CD93] were introduced.

One of the drawbacks of the first generation algorithms on functional arrival time analysis was that false path detection was performed for each path separately. Since a huge number of paths can exist in a circuit, explicit path enumeration is never practical for the analysis of large circuits. Devadas *et al.* [DKM93] overcame this difficulty by showing that the falsity of a set of paths can be determined by a modified automatic test pattern generation (ATPG), termed timed ATPG. This breakthrough enlarged the applicability of functional arrival time analysis to industrial circuits.

In the rest of this subsection we will introduce various path sensitization conditions[6] to be referred to in the subsequent chapters after giving basic definitions on paths and controlling/non-controlling values. To simplify the argument we assume that a Boolean network under analysis is comprised of simple gates although some sensitization conditions (e.g. viability) can be defined over general networks. Simple gates are NOT, AND, OR, NAND and NOR.

Definition 2.1 *A* path *in a Boolean network is a sequence of nodes (gates)* $(g_0, g_1, \ldots, g_m)$*, where* g_{i+1} *is a fanout of* g_i*.* $(i = 0, \ldots, m-1)$[7].

Definition 2.2 *A path that starts from a primary input node and ends at a primary output node is an* input/output path *or an* I/O path *for short.*

Definition 2.3 *Let* $P = (g_0, g_1, \ldots, g_m)$ *be a path. The fanins of* $g_i (i = 1, \ldots, m)$ *other than* g_{i-1} *are called* side inputs *of* g_i *along P. A path from a primary input to a side input along P is called a* side path *of P.*

Definition 2.4 *A* controlling value *for a gate is the input value that determines the output of the gate independent of the values of the other inputs. The output of the gate determined by a controlling value is called a* controlled value *for the gate. For example, the controlling value for an AND gate is 0 while the controlling value for an OR gate is 1. The controlled value for an AND gate is 0 while the controlled value for an OR gate is 1. A* non-controlling value *for a gate is the input value that is not a controlling value for the gate. A* non-controlled value *for a gate is the output value that is not a controlled value for the gate. For example, the non-controlling value for an AND gate is 1 while the non-controlling value for an OR gate is 0. The non-controlled value for an AND gate is 1 while the non-controlled value for an OR gate is 0.*

[6] We restrict ourselves to single-vector timing analyses, those in which a path sensitization is defined under a single input vector. Devadas *et al.*[DKMW94] argued the truth and the falsity of a path given two input vectors. The first vector is used to settle down a circuit, and then the second vector is applied to see if the path is responsible for delay. This can be generalized to a multi-vector analysis proposed by Lam *et al.* [LBSV93, LB94]. Although multi-vector analysis is more accurate than single-vector analysis in theory, the latter is as accurate as the former in most cases in practice.

[7] We assume that fanins of a node are distinct. A path is uniquely identified under this condition.

Early approaches to the false path problem were based on case analysis [Ous83, Jou83]. A specific value is applied to a primary input and propagated forward as much as possible. If a path has a gate whose side input is set to a controlling value of the gate, the path is categorized as false. The underlying path sensitization condition is called *static sensitization*, and was formalized by Benkoski *et al.* [BMCM87, BMCM90]. It is the first sensitization condition proposed in the literature to tackle the false path problem.

Definition 2.5 *A path* $P = (g_0, g_1, \ldots, g_m)$ *is* statically *sensitizable under an input vector* **x** *if all the side inputs of* $g_i (i = 1, \ldots, m)$ *are set to the non-controlling value of* g_i.

It turned out that this sensitization condition is too optimistic, i.e. it can classify a true path as false thereby underestimating true delay[8]. As discussed earlier, this flaw is fatal and makes the condition unusable in timing verification.

Once it became clear that static sensitization is incorrect, several new conditions were proposed to fix the flaw. Brand and Iyengar [BI88] were the first group that gave a sensitization condition proven to be correct.

Definition 2.6 *Assume that the fanins of each gate are linearly ordered. A path* $P = (g_0, g_1, \ldots, g_m)$ *is sensitizable in the* Brand-Iyengar *condition under an input vector* **x** *if all the side inputs of* $g_i (i = 1, \ldots, m)$ *that are after* g_{i-1} *in the fanin ordering of* g_i *have the non-controlling value of* g_i.

Depending on the fanin orderings used, the accuracy of delay estimates under the Brand-Iyengar condition varies. However, it is guaranteed [BI88] that no underestimation occurs under any fanin ordering and any arrival time condition. Notice that the Brand-Iyengar condition is a purely Boolean condition independent of gate delays and arrival times at the inputs.

McGeer and Brayton [MB91] introduced a sensitization condition termed *viability*.

Definition 2.7 *A path* $P = (g_0, g_1, \ldots, g_m)$ *is* viable *under an input vector* **x** *if at each* $g_i (i = 1, \ldots, m)$, *for each side input of* g_i *either of the following two holds.*

1. *The side input has the non-controlling value of* g_i, *or*
2. *There is a viable path from a primary input to the side input whose arrival time via the path is no earlier than* $arr(g_0) + \sum_{j=1}^{i-1} d(g_j)$.

[8] Brand and Iyengar [BI88] showed a simple example where a statically non-sensitizable path can propagate a signal event, and thus determine the delay of the circuit. McGeer and Brayton [MB91] also discussed the flaw of static sensitization using the same circuit.

Viability takes advantage of arrival times at the primary inputs and gate delays to achieve more accurate delay estimation than the Brand-Iyengar condition.

The *floating-mode condition* proposed by Chen and Du [CD93] is another sensitization condition that is known to have the same accuracy as viability for arrival time analysis of networks composed of simple gates.

Definition 2.8 *A path* $P = (g_0, g_1, \ldots, g_m)$ *is sensitizable in the* floating-mode condition *under an input vector* **x** *if for each* $g_i (i = 1, \ldots, m)$, *either*

1. g_i *has a controlled value and* g_{i-1} *gets the earliest controlling value of* g_i, *or*
2. g_i *has a non-controlled value and* g_{i-1} *gets the latest non-controlling value of* g_i.

Static co-sensitization is a sensitization condition independent of gate delays and arrival times. This was introduced by Devadas *et al.* in [DKM93] as a necessary condition for a path to be true under the floating mode condition. Although it is not as accurate as viability or floating-mode, it is a conservative condition.

Definition 2.9 *A path* $P = (g_0, g_1, \ldots, g_m)$ *is* statically co-sensitizable *under an input vector* **x** *if for each* $g_i (i = 1, \ldots, m)$ *that has a controlled value of* g_i, g_{i-1} *has a controlling value of* g_i.

All the correct sensitization conditions introduced in this section meet the monotone speed-up property [MB91].

2.4.6 Functional Arrival Time Analysis under the Extended Bounded Delay-0 Model

We review functional arrival time analysis under a delay model called the *extended bounded delay-0 model* (XBD0 model), originally proposed by Seger [Seg89] and adapted by McGeer *et al.* [MSBSV93]. The rest of the dissertation is built up on top of this delay model. This is the delay model underlying viability [MB91] and the floating mode [CD93]. The procedure for functional arrival time analysis under the XBD0 model developed by McGeer *et al.* [MSBSV93] is one of the most accurate and the most efficient algorithms applicable to large circuits.

Sensitization under the XBD0 Model

Under the XBD0 model, each gate in a network has a maximum positive delay and a minimum delay which is zero. Sensitization analysis assumes that each gate can take any delay between its maximum value and zero.

The core idea of [MSBSV93] is to recursively characterize the set of all input vectors that make the signal value of a primary output stable to a constant (either 0 or 1) by a given required time. Once these sets are identified both for values 0 and 1, one can compare these against the on-set and the off-set of the primary output respectively to see if the output is indeed stable for all input vectors by the required time. The overall scenario of computing output arrival time is to start by setting the required time to the longest topological delay minus $\delta > 0$ and gradually decrease it until some input vector cannot make the output stable by the required time. The second-to-last required time gives an approximation to the true arrival time at the output. This process of guessing the next required time can be sped up and refined by making use of a binary search.

Let us illustrate how we can compute these sets. Let n and $d(n)$ be a node (gate) in a Boolean network $\mathcal{N}$ and the maximum delay of the node n respectively[9]. Let $\chi^{\tau}_{n,v}$ be the characteristic function of the set of input vectors under which the output of the node n becomes stable to a value $v \in \{0,1\}$ by $t = \tau$. Let f_n be the local functionality of the node n in terms of immediate fanins $m_1, \ldots, m_k$ of n. For ease of explanation, let $f_n = m_1 m_2$, i.e., n is a two-input AND gate. It is clear from the functionality of the AND gate that to set n to a value 1 by $t = \tau$, both of the fanins of n, m_1 and m_2, are required to be stable at 1 by $t = \tau - d(n)$. This is equivalent to

$$\chi^{\tau}_{n,1} = \chi^{\tau-d(n)}_{m_1,1} \cdot \chi^{\tau-d(n)}_{m_2,1}.$$

Note that the two χ functions for the fanins are ANDed to take the intersection of the two sets. Similarly, to set n to a value 0 by $t = \tau$, at least one of the fanins must be stabilized to 0 by $t = \tau - d_n$.

$$\chi^{\tau}_{n,0} = \chi^{\tau-d(n)}_{m_1,0} + \chi^{\tau-d(n)}_{m_2,0}$$

Here the two χ functions are ORed to take the union of the two sets. It is easy to see that the above computations can be generalized to the case where the local functionality of n is given as an arbitrary function in terms of its fanins as follows.

$$\chi^{\tau}_{n,v} = \sum_{p \in P^v_n} [\prod_{m_i \in p} \chi^{\tau-d(n)}_{m_i,1} \cdot \prod_{\overline{m_i} \in p} \chi^{\tau-d(n)}_{m_i,0}]$$

where P^1_n and P^0_n are the sets of all primes of f_n and $\overline{f_n}$ respectively. One can easily verify that the recursive formulations for the AND gate shown above are captured in this general formulation by noticing $P^1_n = \{m_1 m_2\}, P^0_n = \{\overline{m_1}, \overline{m_2}\}$ for $f_n = m_1 m_2$. The terminal cases are given when the node

[9] It is possible to differentiate rise delays from fall delays. However, we do not distinguish them to simplify exposition.

n is a primary input x.

$$\begin{aligned} \chi^{\tau}_{x,1} &= x && \text{if } \tau \geq arr(x) \\ &= 0 && \text{otherwise} \\ \chi^{\tau}_{x,0} &= \bar{x} && \text{if } \tau \geq arr(x) \\ &= 0 && \text{otherwise} \end{aligned}$$

where $arr(x)$ denotes the arrival time of x. The above formulas simply say that a primary input is stable only after a given arrival time. The key observation of this formulation is that characteristic functions can be computed *recursively*.

Once characteristic functions for values 0 and 1 are computed at a primary output, two comparisons are made: one for the characteristic function for 1 against the on-set of the output, and the other for the characteristic function for 0 against the off-set of the output. Each comparison is done by creating a Boolean network which computes the difference between two functions and using a satisfiability (SAT) solver to check whether the output of the network is satisfiable[10]. The Boolean network is called a χ-network or a sensitization network. Experimental results in [MSBSV93] showed that this approach can analyze large networks in reasonable computation time.

McGeer *et al.* [MSBSV93, MSS+92] showed that viability and the floating mode condition are both "exact" for functional arrival time analysis in the sense that stable time estimates of the two sensitization conditions are the same as the one with χ network analysis for circuits composed of simple gates.

Optimal Construction of χ-Networks

We have mentioned that a χ-network is constructed recursively from a primary output. McGeer *et al.* discuss further optimization to reduce the size of χ-networks.

Given a required time at a primary output, assume that a backward topological required time propagation of $\mathcal{N}$ is done to primary inputs so that the list of all required times at each internal node is computed instead of the single earliest required time computed by regular topological analysis. If the χ-network is constructed naively, for each internal node in $\mathcal{N}$, a distinct node is to be created for each required time in the list. This, however, is not necessary since the internal node

[10]Larrabee discusses in [Lar92] how to construct a conjunctive-normal-form (CNF) SAT formula which is satisfiable if and only if the output of a given network is satisfiable. An effective heuristic for solving a CNF-SAT problem is also given. A CNF formula is a conjunction of disjunction of literals. Stephan *et al.* [SBSV96] later improved Larrabee's result. The implementation of [MSBSV93] was built on top of the SAT solver by Stephan *et al.*

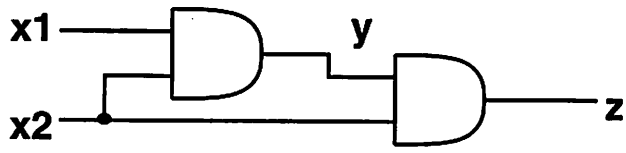

Figure 2.4: Example: Functional Arrival Time Analysis

may exhibit the same stability behavior at different required times, in which case having a single node in the χ-network for the required times is enough. To detect such a case a forward topological arrival time propagation from primary inputs to primary outputs is performed to compute the list of all potential arrival times at each node. Note that each potential arrival time corresponds to the topological delay of a path from a primary input to the internal node. The stability of the node can only change at those times. In other words between two adjacent potential arrival times, one cannot see any change in the stability of the node.

Consider an internal node $n \in \mathcal{N}$. Let $R = (r_1, \ldots, r_p)$ and $A = (a_1, \ldots, a_q)$ denote the sorted list of required times and that of arrival times respectively at node n. Consider χ function $\chi^{r_i}_{n,v}(v = 0, 1)$. Let $a_j \in A$ be the maximum arrival time such that $a_j \leq r_i$. Since there is no event happening between time a_j and r_i, $\chi^{r_i}_{n,v} = \chi^{a_j}_{n,v}$. Matchings from required times to arrival times are performed in this fashion to identify the subset of A that is required to compute the final χ function. This optimization avoids creating redundant nodes in the χ network thereby reducing the size of the χ network without losing any accuracy in analysis. Only those arrival times that have a match with a required time yield nodes in the χ network.

Another type of optimization suggested in [MSBSV93] is to generate the list of arrival times more carefully. For each potential arrival time, equivalence between the corresponding χ function and the on-set or the off-set (whichever appropriate) is checked by a satisfiability call and a new node is created in χ network only if the two functions are different. Otherwise, the original function or its complement is used as it is. Although this requires additional CPU time spent on satisfiability calls, it is experimentally confirmed that the size reduction of the final χ network is so significant that the the total run-time decreases in most cases.

Example

We illustrate functional arrival time analysis under the XBD0 model using a circuit in Figure 2.4. Assume that both of the AND gates have unit delays, and that both of the primary inputs

arrive at $t = 0$. The arrival time estimate of output z by topological arrival time analysis is $t = 2$ since there is a path of length 2 (x_1, y, z).

We will compute χ functions of z for values 1 and 0 at $t = 1$ to see if z gets stable by $t = 1$. The recursive definition of χ functions give:

$$\begin{aligned}
\chi^1_{z,1} &= \chi^0_{y,1} \cdot \chi^0_{x_2,1} \\
&= \chi^{-1}_{x_1,1} \cdot \chi^{-1}_{x_2,1} \cdot \chi^0_{x_2,1} \\
&= 0 \cdot 0 \cdot x_2 \\
&= 0 \\
\chi^1_{z,0} &= \chi^0_{y,0} + \chi^0_{x_2,0} \\
&= \chi^{-1}_{x_1,0} + \chi^{-1}_{x_2,0} + \chi^0_{x_2,0} \\
&= 0 + 0 + \overline{x_2} \\
&= \overline{x_2}.
\end{aligned}$$

The χ function of z for value 1 at $t = 1$ is 0, which means that no input vector can make z stable to 1 by $t = 1$. The χ function for value 0 is $\overline{x_2}$. Therefore, under $(x_1, x_2) = (0,0)$ and $(1,0)$ z gets stable to 0 by $t = 1$.

The two χ functions for values 1 and 0 are now compared against the on-set $(x_1 x_2)$ and the off-set $(\overline{x_1} + \overline{x_2})$ of z respectively by taking a set difference.

$$\begin{aligned}
z \cdot \overline{\chi^1_{z,1}} &= x_1 x_2 \\
\overline{z} \cdot \overline{\chi^1_{z,0}} &= (\overline{x_1} + \overline{x_2}) x_2 \\
&= \overline{x_1} x_2
\end{aligned}$$

Both formulas are satisfiable. The first formula indicates that z is not stable to 1 by $t = 1$ under $(x_1, x_2) = (1,1)$ while the second indicates that z is not stable to 0 by $t = 1$ under $(x_1, x_2) = (0,1)$. Since z does not get stable by $t = 1$ under all input vectors, the arrival time of z is later than $t = 1$. In fact, the arrival time estimate $t = 2$ by topological analysis is accurate for this circuit.

Chapter 3

Functional Required Time Analysis

As we saw in Chapter 2, the timing requirement of a combinational circuit is typically specified by arrival times at primary inputs and required times at primary outputs. Once an actual circuit is designed under a timing specification, we need to verify whether the circuit meets the performance goal or not. Arrival time analysis discussed in Section 2.4 is one way to validate the timing of the circuit, in which the arrival times at the primary inputs are propagated through the circuit *forward*, and the computed arrival times at the primary outputs are compared against the specified required times to see if the timing requirement is met. Depending on how much accuracy is required, we have two types of analyses: *topological* arrival time analysis and *functional* arrival time analysis. Functional arrival time analysis has been studied extensively in the literature as we discussed in Section 2.4.

Although timing analysis based on arrival time computation is a standard technique, that is not the only way to perform timing verification. Another approach is to propagate the required times at the primary outputs *backward* through the circuit to compute the required times at the primary inputs. By checking whether the arrival time at each input is no later than the corresponding computed required time the performance of the circuit can be validated. We call this backward propagation of required times *required time analysis*. A by-product of this approach is the availability of the criticality of each input at the end of the analysis, which was not the case for arrival time analysis.

Although required time analysis has been used in many synthesis operations such as slack computation, it has been mainly done by ignoring false paths. Therefore little is known about how to perform required time analysis by considering false paths[1]. The goal of this chapter is to leverage the

[1]The only work that we are aware of is [BI88], where the authors briefly mentioned that their functional timing analysis algorithm developed for arrival time computation can be used for required time computation. This is a natural conclusion since the Brand-Iyengar sensitization condition is independent of arrival times at primary inputs. This will be elaborated on in Section 4.6.

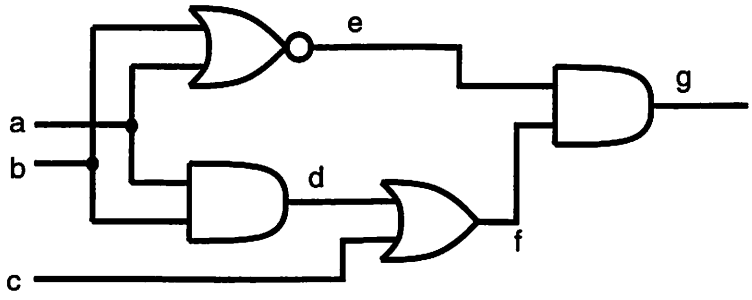

Figure 3.1: Example: Topological Required Time vs. Functional Required Time

understanding of *functional* required time analysis up to the level of functional arrival time analysis to make false-path-aware required time computation possible. We will see in subsequent chapters that functional required time analysis forms the basis for the solution of various timing-related problems.

This chapter is organized as follows. We start with an example where false-path aware required time analysis gives a more accurate estimate of required time than topological required time analysis in Section 3.1. Motivated by the example, the functional required time analysis problem is formulated in Section 3.2. An exact algorithm and two approximate algorithms are presented in Section 3.3. Section 3.4 discusses timing constraints of subcircuits and their computation as an application of functional required time analysis. Experimental results are given in Section 3.5 and the chapter is concluded in Section 3.6.

3.1 Example

Consider a circuit shown in Figure 3.1 taken from [BI88]. Assume the unit delay model, i.e. each gate has a delay of one. Suppose that the output g has a required time $t = 3$. What is the required time at each primary input?

Let us first try topological required time analysis. The longest topological path from a to the output is of length 3. Therefore assuming that this path is responsible for determining the output value, the required time at a is $t = 3 - 3 = 0$. The same argument holds for b. As to input c, the longest path to the output is of length 2. Thus, the required time is $t = 3 - 2 = 1$. As long as inputs a, b and c arrive by $t = 0, 0$ and 1 respectively, the output is guaranteed to be stable by $t = 3$.

Now consider the following arrival times at the inputs: $arr(a) = 1, arr(b) = 0, arr(c) = 1$. This arrival time condition does not meet the required times computed by topological analysis since

a arrives later than the computed required time of $t = 0$. However, if we perform functional arrival time analysis of the circuit under this condition, we can confirm that the output is indeed stable at $t = 3$, which indicates that $req(a) = 1, req(b) = 0, req(c) = 1$ are valid required times. Notice that this required time condition is looser than the one computed by topological analysis since a is only required at $t = 1$ while it was required at $t = 0$ before.

The reason why this more accurate required time condition was overlooked in topological analysis is that the analysis completely ignores false paths. In this circuit, under $arr(a) = 1, arr(b) = 0, arr(c) = 1$ the topological longest path from a to the output of length 3 is false and not responsible for the signal stability of the output. Therefore, the effective delay from a is only 2. However, topological analysis never detects this.

As is true for arrival time analysis, false path detection is crucial for accurate required time analysis. We will see in subsequent discussions how required time analysis can be achieved by taking false paths into account.

3.2 Problem Formulation

> **Functional Required Time Analysis:** Given a Boolean network $\mathcal{N}$, maximum delay $d(n)$ of each gate n, and required times at the primary outputs, compute required times at the primary inputs by considering false paths. The minimum delay of each gate is assumed to be zero as in the XBD0 model.

3.3 Functional Required Time Analysis

Required time analysis can be performed efficiently if it is done purely based on topological delays. Topological required time analysis first sorts all the nodes in a network in a reverse topological order. Each node n in the network is then visited in this order and the required time at the node $req(n)$ is computed as $req(n) = \min_{m \in FANOUT(n)}(req(m) - d(m))$. The procedure runs in time linear in the size of the network. Note that required times are uniquely determined in this algorithm, which is not necessarily the case once false paths are taken into account as we will see later.

The approach proposed in this section makes use of χ functions introduced in Section 2.4.6. For each primary output, χ functions for values 0 and 1 are computed for a given required time and they are compared against the on-set and the off-set of the output function respectively to extract con-

ditions on required times at primary inputs. The main difference between this problem and functional arrival time analysis discussed in Section 2.4.6 is that arrival times at primary inputs are unknown variables in our problem while they are given in the other. In spite of this difference the original recursive formulation for computing χ functions almost works. A modification is required only in terminal cases. Since we do not know when a primary input signal arrives, χ functions at primary inputs remain as unknown variables. Henceforth, we call χ functions at primary inputs *leaf χ variables*.

Let x be a primary input. Assume that after recursive construction of χ functions at primary outputs, leaf χ variables for x are needed at times $\tau_1 < \tau_2 < \ldots < \tau_{p_1(x)}$ for value 1 and at times $\tau'_1 < \tau'_2 < \ldots < \tau'_{p_0(x)}$ for value 0. Remember that each leaf χ variable represents the set of input vectors under which x is stable by a certain time. Once x becomes stable, it continues to be stable; thus for any $\tau_a < \tau_b$ the set of input vectors for $t = \tau_a$ must be contained in the set of input vectors for $t = \tau_b$. Therefore, the following ordering conditions among leaf χ variables must hold.

$$\emptyset \subseteq \chi_{x,1}^{\tau_1} \subseteq \chi_{x,1}^{\tau_2} \subseteq \ldots \subseteq \chi_{x,1}^{\tau_{p_1(x)}} \subseteq x \tag{3.1}$$

$$\emptyset \subseteq \chi_{x,0}^{\tau'_1} \subseteq \chi_{x,0}^{\tau'_2} \subseteq \ldots \subseteq \chi_{x,0}^{\tau'_{p_0(x)}} \subseteq \overline{x} \tag{3.2}$$

The formulas above indicate that leaf χ variables are

1. non-decreasing with respect to time and
2. bounded above by x and $\overline{x}$ for value 1 and 0 respectively.

The first constraint is imposed since, once an input vector becomes stable, it must continue to be stable. The second constraint is required so that leaf χ variables are compatible with the on-set and the off-set of the primary input x.

Let us go back to the original problem. For simplicity, assume that a Boolean network $\mathcal{N}$ has a single primary output z, whose required time T is given. Generalization to multiple primary outputs is trivial[2]. We are interested in computing required times at the primary inputs of the network.

Suppose that $\chi_{z,1}^T$ and $\chi_{z,0}^T$ are computed in terms of leaf χ variables at primary inputs, which we call χ_X. The goal is to assign Boolean functions $\chi_X(X)$ of primary inputs $X = (x_1, \ldots, x_n)$ to unknown χ_X variables so that when $\chi_X = \chi_X(X)$,

$$\begin{aligned} \chi_{z,1}^T(\chi_X) &= z(X) \\ \chi_{z,0}^T(\chi_X) &= \overline{z(X)} \end{aligned}$$

[2]It is just enough to take a conjunction of the Boolean constraints for all the primary outputs.

under the ordering constraints between χ_X variables in Equations 3.1 and 3.2, where $z(X)$ denotes the functionality of the primary output in terms of primary inputs X[3]. The sets of input vectors that make the output stable at value 1 and 0 by $t = T$ are constrained to be equal to the on-set and the off-set of the output function respectively.

3.3.1 Exact Analysis

One can formulate this problem as solving a Boolean equation where unknown variables are leaf χ variables χ_X. The Boolean constraints to be satisfied are:

$$\begin{aligned} \chi^T_{z,1}(\chi_X) &= z(X) \\ \chi^T_{z,0}(\chi_X) &= \overline{z(X)} \\ \text{for each } x \in X: &\quad \emptyset \subseteq \chi^{\tau_1}_{x,1} \subseteq \ldots \subseteq \chi^{\tau_{P_1(x)}}_{x,1} \subseteq x \\ \text{for each } x \in X: &\quad \emptyset \subseteq \chi^{\tau'_1}_{x,0} \subseteq \ldots \subseteq \chi^{\tau'_{P_0(x)}}_{x,0} \subseteq \bar{x} \end{aligned}$$

It is easy to transform the above set of Boolean equations to a single equivalent Boolean equation of form $F(X, \chi_X) = 1$ [Bro90] by taking the conjunction of all the constraints. In this equation, χ_X are variables to be solved while X are Boolean constants. One can think of $F(X, \chi_X)$ as the characteristic function of a Boolean relation [BS89a, BS89b] where X is the inputs and χ_X is the outputs. Any function in terms of X, compatible with F, satisfies the timing requirement at z[4].

Notice that the notion of required times at primary inputs is significantly generalized here. For each primary input, its required time is not simply a single constant any more. An input signal can arrive at different times depending on signal values of the other inputs. The analysis is "exact" in the same sense that viability and floating mode analyses are exact for the XBD0 model. More precisely, given an input vector **x** and any arrival time condition A compatible with F for **x**, functional arrival time analysis under **x** and A gives an arrival time at the output no greater than the given required time. Furthermore, all the latest arrival time conditions that meet the given required time at the output are guaranteed to be included in F. We will show how to extract the latest arrival time conditions later.

[3]It is possible to extend the theory to the case where z is specified as an incompletely specified function. If the on-set and the off-set are $z^1(X)$ and $z^0(X)$ respectively, the two equations need to be replaced with

$$\begin{aligned} z^1(X) &\Rightarrow \chi^T_{z,1}(\chi_X) \\ z^0(X) &\Rightarrow \chi^T_{z,0}(\chi_X). \end{aligned}$$

[4]One method for extracting such a function is Boolean unification [Bro90].

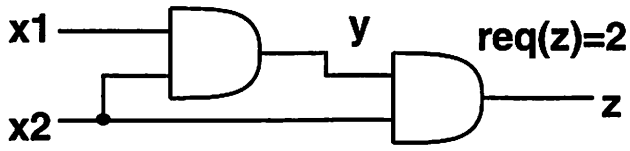

Figure 3.2: Example: Functional Required Time Analysis

Let us illustrate this in the circuit shown in Figure 3.2. For simplicity, assume that the maximum delay of the AND gate is 1 and the required time at the primary output z is 2. The required time computed by topological delay analysis is $t = 0$ for both inputs. χ functions for z can be computed as follows.

$$\begin{aligned}
\chi^2_{z,1} &= \chi^1_{y,1}\chi^1_{x_2,1} \\
&= \chi^0_{x_1,1}\chi^0_{x_2,1}\chi^1_{x_2,1} \\
\chi^2_{z,0} &= \chi^1_{y,0}+\chi^1_{x_2,0} \\
&= \chi^0_{x_1,0}+\chi^0_{x_2,0}+\chi^1_{x_2,0}
\end{aligned}$$

These equations along with the ordering constraints give the following Boolean relation.

x_1x_2	$\chi^0_{x_1,1}\chi^0_{x_2,1}\chi^1_{x_2,1}\chi^0_{x_1,0}\chi^0_{x_2,0}\chi^1_{x_2,0}$
00	{000100,000101,000001,000011,000111}
01	{000100,001100,011100}
10	{000001,000011,100001,100011}
11	{111000}

The interpretation of the relation as required times is as follows.

x_1x_2	$(req(x_1), req(x_2))$
00	$\{(0,\infty),(0,1),(\infty,1),(\infty,0),(0,0)\}$
01	$\{(0,\infty),(0,1),(0,0)\}$
10	$\{(\infty,1),(\infty,0),(0,1),(0,0)\}$
11	$\{(0,0)\}$

Let us examine the relation to see what kind of timing constraint needs to be imposed. For input vector 00, the first three leaf χ variables must be 0 in all the cases. This is natural since these χ variables are for value 1, and neither x_1 nor x_2 may become stable to 1 in this case. The first two and the last vectors correspond to the case where $\chi^0_{x_1,0}$ is 1, i.e. x_1 is stable to 0 by $t = 0$. In this case, z is guaranteed to be stable to 0 no matter how x_2 behaves. The only constraint to be satisfied is $\chi^0_{x_2,0} \subseteq \chi^1_{x_2,0}$. The third and fourth vectors are for the case $\chi^0_{x_1,0}$ is 0. This time, x_1 is not stable to

0 at $t = 0$, but as long as x_2 becomes 0 by $t = 1$, z will be stable by $t = 2$ (both vectors have 1 in the last position). Again, the ordering condition $\chi^0_{x_2,0} \subseteq \chi^1_{x_2,0}$ must hold.

One can think of this relation as a generalization of the existing notion of required time. Any signal behavior at primary inputs that is compatible with this relation meets the required time at the primary output. For example, if we pick 000100, 000100, 000001, and 111000 for input vectors 00, 01, 10, 11 respectively, then leaf χ variables will be:

$$\begin{aligned}
\chi^0_{x_1,1} &= x_1 x_2 \\
\chi^0_{x_2,1} &= x_1 x_2 \\
\chi^1_{x_2,1} &= x_1 x_2 \\
\chi^0_{x_1,0} &= \overline{x_1} \\
\chi^0_{x_2,0} &= 0 \\
\chi^1_{x_2,0} &= x_1 \overline{x_2}.
\end{aligned}$$

To focus on only the stability of signals, we define $\tilde{\chi}^\tau_n$ as follows.

$$\tilde{\chi}^\tau_n = \chi^\tau_{n,1} + \chi^\tau_{n,0}$$

This $\tilde{\chi}$ function of a node n at $t = \tau$ is the characteristic function of the set of all input vectors that make the signal n stable either to 0 or 1 by $t = \tau$. For the χ functions above,

$$\begin{aligned}
\tilde{\chi}^0_{x_1} &= \overline{x_1} + x_2 \\
\tilde{\chi}^0_{x_2} &= x_1 x_2 \\
\tilde{\chi}^1_{x_2} &= x_1.
\end{aligned}$$

The interpretation of this is that primary input x_1 must be stable by $t = 0$ just for the case $\overline{x_1} + x_2$, and if $x_1 \overline{x_2}$, it can delay forever without violating the given functional and temporal requirements. Notice that in topological analysis it always has to arrive no later than $t = 0$.

Let us look into how signal x_2 should behave. It must be stabilized by $t = 0$ for the case $x_1 x_2$. If $\tilde{\chi}^1_{x_2} \overline{\tilde{\chi}^0_{x_2}} = x_1 \overline{x_2}$, x_2 has to become stable by $t = 1$. For all the other cases, i.e. if $x_1 = 0$, however, x_2 can be indefinitely delayed.

One can easily see that the relation contains a compatible function corresponding to the required time computed by topological analysis[5], which gives the most pessimistic required time condition.

[5]Choose the last vector for each input vector in the relation.

We have seen that the relation represents all the permissible temporal behaviors, from an aggressive behavior where a signal never arrives under a certain condition to the most stringent condition exactly corresponding to topological analysis. The next question is how to extract the latest required time conditions from the relation since the later the required times are, the looser the condition is.

For each input vector the relation gives a set of permissible vectors for leaf χ variables. Since a 1 in a vector means that the corresponding leaf χ variable must be stable, having fewer 1's requires less stability. For example, under input vector 00 we have two vectors 000100 and 000101. Since the second vector strictly requires more stability than the first one, the second vector can be safely dropped. By repeatedly removing vectors that require more stability than another, we eventually have a subset of permissible vectors in which no vector is subsumed by another. We call these vectors *minimal-stability* vectors. The latest required time is now characterized by a subset relation of the original relation where each input vector can be mapped only to the minimal-stability vectors. Each minimal-stability vector captures a latest required time condition in the sense that delaying any primary input from the required time immediately causes the arrival time of the output to exceed the given required time. For the working example, the subset relation is shown below on the left while its interpretation as required times is shown on the right.

x_1x_2	$\chi^0_{x_1,1}\chi^0_{x_2,1}\chi^1_{x_2,1}\chi^0_{x_1,0}\chi^0_{x_2,0}\chi^1_{x_2,0}$
00	{000100,000001}
01	{000100}
10	{000001}
11	{111000}

x_1x_2	$(req(x_1), req(x_2))$
00	$\{(0,\infty),(\infty,1)\}$
01	$\{(0,\infty)\}$
10	$\{(\infty,1)\}$
11	$\{(0,0)\}$

An important point to notice is that there may be more than one loosest required time even after an input vector is specified unlike topological analysis where required time is always unique. In this particular example, either x_1 arriving by $t = 0$ or x_2 arriving by $t = 1$ is required for $x_1x_2 = 00$. Those two conditions are not comparable and each gives a different limiting condition.

3.3.2 Approximate Analysis

Although the exact analysis gives the most accurate information, it is computationally expensive. We will introduce two approximate analyses, which are less computationally intensive yet still more accurate than topological required time analysis. Both analyses are conservative approximations to the exact analysis.

Approximate Analysis via Simplified Modeling

In the exact analysis, a primary input signal can arrive at different times depending on signal values of the other inputs. In this first approximate analysis, we give up this flexibility and limit ourselves to the case where a primary input signal arrives at a fixed time, independent of signal values of the other inputs. By this modification if an input signal is required early only under a specific input vector, then the input is required early under many other input vectors since there is no way to distinguish these vectors. This is how accuracy is lost conservatively. Arrival times for values 0 and 1, however, are still distinguished[6]. This simplification allows a simpler Boolean modeling which is more efficient to solve.

In the exact approach, we need to impose the ordering constraints explicitly among leaf χ variables as Boolean constraints. Once we assume the vector-independent signal arrival behavior at a primary input, the constraints can be directly modeled by introducing additional Boolean variables. Specifically, Boolean variables $\alpha_1^x, \ldots, \alpha_{p_1(x)}^x, \beta_1^x, \ldots, \beta_{p_0(x)}^x$ are introduced to encode the ordering constraints in leaf χ variables as follows[7].

$$\begin{aligned}
\chi_{x,1}^{\tau_{p_1(x)}} &= x\alpha_1^x \\
\chi_{x,1}^{\tau_{p_1(x)-1}} &= x\alpha_1^x\alpha_2^x \\
&\cdots \\
\chi_{x,1}^{\tau_1} &= x\alpha_1^x\alpha_2^x\ldots\alpha_{p_1(x)}^x \\
\chi_{x,0}^{\tau'_{p_0(x)}} &= \bar{x}\beta_1^x \\
\chi_{x,0}^{\tau'_{p_0(x)-1}} &= \bar{x}\beta_1^x\beta_2^x \\
&\cdots \\
\chi_{x,0}^{\tau'_1} &= \bar{x}\beta_1^x\beta_2^x\ldots\beta_{p_0(x)}^x
\end{aligned} \tag{3.3}$$

Notice that all the ordering constraints are automatically satisfied by the use of the Boolean variables. The side effect of this encoding is that leaf χ variables can now either take x or 0 for value 1, and either $\bar{x}$ or 0 for value 0 under a 0-1 assignment to the α and β variables while they can take any function between 0 and x for value 1 and between 0 and $\bar{x}$ for value 0 in the exact analysis. This, however, directly corresponds to our new constraint that each primary input arrives at a fixed time

[6]One can design a more conservative approximation scheme by not distinguishing arrival times for values 0 and 1.

[7]One can employ a log-based encoding to decrease the number of Boolean variables introduced although this makes it difficult to extract the loosest required times later.

no matter how the other inputs behave. The remaining condition to be satisfied is that the two χ functions for the primary output are equal to the on-set and the off-set of the output respectively.

$$\begin{aligned} \chi^T_{z,1}(X,\alpha,\beta) &= z(X) \\ \chi^T_{z,0}(X,\alpha,\beta) &= \overline{z(X)} \end{aligned}$$

where α and β are the set of all α variables and the set of all β variables respectively. In the above all the leaf χ variables are substituted by the right-hand side expressions in Equation (3.3). Since these equations must be true regardless of X, X should be universally-quantified.

$$\begin{aligned} F(\alpha,\beta) &= \forall X.[(\chi^T_{z,1}(X,\alpha,\beta) = z(X))(\chi^T_{z,0}(X,\alpha,\beta) = \overline{z(X)})] \\ &= \forall X.[\chi^T_{z,1}(X,\alpha,\beta) = z(X)]\forall X.[\chi^T_{z,0}(X,\alpha,\beta) = \overline{z(X)}] \end{aligned}$$

Any satisfying assignment of $F(\alpha,\beta)$ meets the timing requirement.

Let us go one step further, as in the exact analysis, to see how we can compute the loosest required time at primary inputs from $F(\alpha,\beta)$. The following lemmas and theorems are useful.

Lemma 3.1 *$\chi^T_{z,1}(\chi_X)$ and $\chi^T_{z,0}(\chi_X)$ are monotone increasing functions in terms of χ_X.*

Proof By the definition of χ functions, each χ function can be represented by a Boolean network where the local functionality of each node is monotone increasing in terms of its fanins. Hence, the claim holds. □

Lemma 3.2 *$\chi^T_{z,1}(X,\alpha,\beta)$ and $\chi^T_{z,0}(X,\alpha,\beta)$ are monotone increasing functions in terms of α and β.*

Proof Let $\hat{\alpha}$ and $\hat{\beta}$ be 0-1 assignments to α and β respectively. Let $\hat{\chi_X}$ be the functions for leaf χ variables under $\hat{\alpha}$ and $\hat{\beta}$. By changing a 0 in α and β to a 1, it is easy to see that one cannot decrease the functions χ_X. Thus from Lemma 3.1 the claim is proved. □

Lemma 3.3

$$\begin{aligned} \chi^T_{z,1}(X,\alpha,\beta)\,|_{\alpha=(1,\ldots,1),\beta=(1,\ldots,1)} &= z(X) \\ \chi^T_{z,0}(X,\alpha,\beta)\,|_{\alpha=(1,\ldots,1),\beta=(1,\ldots,1)} &= \overline{z(X)} \end{aligned}$$

Proof Let $\tilde{\mathcal{N}}$ be the Boolean network for $\chi^T_{z,1}$. Let NL be the list of all the nodes in the network topologically sorted from primary inputs of $\tilde{\mathcal{N}}$ (leaf χ variables) to the primary output $\chi^T_{z,1}$. Note that each node is labeled of the form $\chi^\tau_{n,v}$. The proof is by induction on these sorted nodes.

Base case ($n \in X$): By setting $\alpha = (1,\ldots,1), \beta = (1,\ldots,1)$, $\chi^\tau_{n,1} = n$ and $\chi^\tau_{n,0} = \bar{n}$ for any τ.

Induction ($n \notin X$): From the inductive hypothesis, for any fanin of $\chi^\tau_{n,v}$, say $\chi^{\tau-d(n)}_{m,v'}$,

$$\begin{aligned} \chi^{\tau-d(n)}_{m,1}\,|_{\alpha=(1,\ldots,1),\beta=(1,\ldots,1)} &= m(X) \\ \chi^{\tau-d(n)}_{m,0}\,|_{\alpha=(1,\ldots,1),\beta=(1,\ldots,1)} &= \overline{m(X)}, \end{aligned}$$

where $m(X)$ is the functionality of node m in terms of X in the original network $\mathcal{N}$. If $v = 1$, then the local function at the node $\chi^\tau_{n,1}$ in $\tilde{\mathcal{N}}$ is the same as the local function at the node n in $\mathcal{N}$ since the former function is just the sum of all the primes of the latter function. Therefore $\chi^\tau_{n,1}\,|_{\alpha=(1,\ldots,1),\beta=(1,\ldots,1)} = n(X)$. Similarly the local function at the node $\chi^\tau_{n,0}$ in $\tilde{\mathcal{N}}$ is the same as the complement of the local function at the node n in $\mathcal{N}$. Thus, $\chi^\tau_{n,0}\,|_{\alpha=(1,\ldots,1),\beta=(1,\ldots,1)} = \overline{n(X)}$.

Hence, $\chi^T_{z,1}\,|_{\alpha=(1;\ldots,1),\beta=(1,\ldots,1)} = z(X)$. $\chi^T_{z,0}\,|_{\alpha=(1,\ldots,1),\beta=(1,\ldots,1)} = \overline{z(X)}$ can be proved similarly. □

Corollary 3.1

$$\begin{aligned} \forall\alpha,\beta : \chi^T_{z,1}(X,\alpha,\beta) &\subseteq z(X) \\ \forall\alpha,\beta : \chi^T_{z,0}(X,\alpha,\beta) &\subseteq \overline{z(X)} \end{aligned}$$

Proof From Lemma 3.2 and Lemma 3.3. □

Theorem 3.1 *$F(\alpha,\beta)$ is a monotone increasing function in terms of α and β.*

Proof

$$\begin{aligned} F(\alpha,\beta) &= \forall X.[(\chi^T_{z,1}(X,\alpha,\beta) = z(X))(\chi^T_{z,0}(X,\alpha,\beta) = \overline{z(X)})] \\ &= \forall X.(\chi^T_{z,1}(X,\alpha,\beta) = z(X))\forall X.(\chi^T_{z,0}(X,\alpha,\beta) = \overline{z(X)}) \end{aligned}$$

Consider a 0-1 assignment to α and β, say $\hat{\alpha}$ and $\hat{\beta}$ respectively. From Lemma 3.2 and Corollary 3.1, it is clear that changing 0's to 1's in $\hat{\alpha}$ and $\hat{\beta}$ does not decrease the function value of $F(\alpha,\beta)$ from 1

to 0. Therefore, $F(\alpha,\beta)$ is a monotone increasing function in terms of α and β. □

We have shown that $F(\alpha,\beta)$ captures all the required times that meet a given timing constraint at the output. Since having less 1's in an assignment to α and β requires less stability, we are interested in a satisfying assignment where no assignment of 1 to a variable can be changed to 0 without making the assignment non-satisfying. Since $F(\alpha,\beta)$ is a monotone increasing function, such an assignment has a one-to-one correspondence with a prime of $F(\alpha,\beta)$. Notice that any prime of the function has only positive literals. The variables with positive literals in a prime are those which must be set to 1. Thus, computing the latest required times from $F(\alpha,\beta)$ is equivalent to computing all the primes of $F(\alpha,\beta)$. Note that each prime gives a different limiting condition as in the exact analysis.

Let us go back to the previous example. By introducing α and β variables, leaf χ variables can be expressed as follows.

$$\begin{aligned}
\chi^0_{x_1,1} &= x_1\alpha_1^{x_1} \\
\chi^0_{x_1,0} &= \overline{x_1}\beta_1^{x_1} \\
\chi^1_{x_2,1} &= x_2\alpha_1^{x_2} \\
\chi^0_{x_2,1} &= x_2\alpha_1^{x_2}\alpha_2^{x_2} \\
\chi^1_{x_2,0} &= \overline{x_2}\beta_1^{x_2} \\
\chi^0_{x_2,0} &= \overline{x_2}\beta_1^{x_2}\beta_2^{x_2}
\end{aligned}$$

$$\begin{aligned}
\chi^2_{z,1} &= \chi^0_{x_1,1}\chi^0_{x_2,1}\chi^1_{x_2,1} = \alpha_1^{x_1}\alpha_1^{x_2}\alpha_2^{x_2}x_1x_2 \\
\chi^2_{z,0} &= \chi^0_{x_1,0}+\chi^0_{x_2,0}+\chi^1_{x_2,0} \\
&= \beta_1^{x_1}\overline{x_1}+\beta_1^{x_2}\overline{x_2}+\beta_1^{x_2}\beta_2^{x_2}\overline{x_2} \\
&= \beta_1^{x_1}\overline{x_1}+\beta_1^{x_2}\overline{x_2}
\end{aligned}$$

The F function for this example is:

$$\begin{aligned}
F(\alpha,\beta) &= \forall x_1,x_2.(\alpha_1^{x_1}\alpha_1^{x_2}\alpha_2^{x_2}x_1x_2 = x_1x_2)(\beta_1^{x_1}\overline{x_1}+\beta_1^{x_2}\overline{x_2} = \overline{x_1}+\overline{x_2}) \\
&= \alpha_1^{x_1}\alpha_1^{x_2}\alpha_2^{x_2}\beta_1^{x_1}\beta_1^{x_2}
\end{aligned}$$

	$(req(x_1), req(x_2))$		
x_1x_2	exact	approximate	topological
00	$\{(0,\infty),(\infty,1)\}$	$\{(0,1)\}$	$\{(0,0)\}$
01	$\{(0,\infty)\}$	$\{(0,0)\}$	$\{(0,0)\}$
10	$\{(\infty,1)\}$	$\{(0,1)\}$	$\{(0,0)\}$
11	$\{(0,0)\}$	$\{(0,0)\}$	$\{(0,0)\}$

Table 3.1: Example: Comparison of Three Required Time Analyses

There are two satisfying assignments for the function:

$$(\alpha_1^{x_1}\alpha_1^{x_2}\alpha_2^{x_2}\beta_1^{x_1}\beta_1^{x_2}\beta_2^{x_2}) = (111110, 111111).$$

The second assignment corresponds to topological analysis. The only prime of $F(\alpha,\beta)$ is $\alpha_1^{x_1}\alpha_1^{x_2}\alpha_2^{x_2}\beta_1^{x_1}\beta_1^{x_2}$, which corresponds to the first assignment. Let us look into the first assignment more carefully. The leaf χ variables under this assignment are:

$$\begin{aligned}
\chi^0_{x_1,1} &= x_1 \\
\chi^0_{x_1,0} &= \overline{x_1} \\
\chi^1_{x_2,1} &= x_2 \\
\chi^0_{x_2,1} &= x_2 \\
\chi^1_{x_2,0} &= \overline{x_2} \\
\chi^0_{x_2,0} &= 0.
\end{aligned}$$

This constraint means that x_1 has to arrive by $t = 0$ under all input vectors while x_2 has to arrive by $t = 0$ if $x_2 = 1$ but by $t = 1$ if $x_2 = 0$.

The results of required time analysis of the circuit in Figure 3.2 are summarized in Table 3.1. The first column shows a vector applied to the primary inputs. The second, the third and the fourth columns show the required times computed by the exact algorithm, the approximate algorithm and the topological algorithm respectively. Notice that the required time computed by the approximate algorithm is more restrictive than that based on the exact algorithm, but is looser than the required time based on topological analysis.

Approximate Analysis via Functional Arrival Time Analysis

The first approximate analysis was based on a relaxation of the exact formulation. The second approximate analysis takes a completely different approach, where functional arrival time

analysis is used as a subroutine to determine required times under the existence of false paths conservatively.

In the exact analysis leaf χ variables at different times need to be distinguished at each primary input x_i. Let R_i be the set that contains all those times for primary input x_i. For the sake of simplicity assume that R_i contains all the times needed for value 1 and those for value 0[8]. Let $R = R_1 \times \ldots \times R_n$. Let $r_\perp = (r_{\perp,1}, \ldots, r_{\perp,n}) \in R$ where $r_{\perp,i} = \min_{t \in R_i} t$. Similarly let $r_\top = (r_{\top,1}, \ldots, r_{\top,n}) \in R$ where $r_{\top,i} = \max_{t \in R_i} t$. Let $r, r' \in R$. A partial order $\prec$ is defined over R as follows: for $\forall r, r' \in R$, $r \prec r'$ if and only if $\forall i \in \{1, \ldots, n\}, r_i \leq r'_i$. This partial order forms a lattice over R, where the top and the bottom elements are $r_\top$ and $r_\perp$ respectively. Each $r \in R$ represents a candidate for the required time condition at the inputs.

$r_\perp$ corresponds to the required times at primary inputs obtained by topological analysis. Therefore, if the primary inputs arrive by $r_\perp$, the stability of the primary output by the given required time is guaranteed. Our goal is to find the largest $r \in R$ with respect to $\prec$ that guarantees the stability of the primary output. r may not be unique as in the first approximate analysis.

One way to find such r is to climb up the lattice gradually from $r_\perp$ by choosing larger r's in a greedy fashion. To test if the current r is a valid choice, one can simply perform functional arrival time analysis of the circuit under the arrival times corresponding to r at the primary inputs. If the arrival time at the primary output is no later than its required time, r is a safe condition. The largest r that meets this requirement gives a limiting condition. The search for r can be refined by the use of backtracking so that all the maximal r's satisfying the condition are enumerated. Furthermore, the greedy search for the next r can be biased so that a specific subset of primary inputs is delayed more aggressively. This results in more accuracy for those inputs in the subset. For example, the subset can be set to critical inputs.

The advantages of this second approximate analysis are twofold. First, one can directly use state-of-the-art functional arrival time analysis as a subroutine. Second, even if an entire analysis takes a huge amount of time, any validated r looser than topological analysis gives useful information immediately. Therefore, it is possible to set a time limit and return the largest r validated so far as an approximate solution.

[8]It is possible to extend the idea so that required times for values 1 and 0 are handled separately for each primary input.

3.4 Extraction of Local Timing Constraints by Functional Timing Analysis

In the previous section we discussed how required times at the primary outputs of a combinational circuit can be propagated through the circuit backward to obtain required times at the primary inputs. Specifically we focused on how to do this by considering false paths. Functional arrival time analysis is a counterpart of this for arrival time computation. Given these two techniques, we can extract an accurate local timing constraint of a subcircuit from a global timing constraint of an entire circuit. This issue will be addressed in this section.

Suppose a combinational network and arrival/required times at primary inputs/outputs are given. Assume that a subnetwork of this circuit is to be optimized. When this subnetwork is resynthesized, arrival times at subcircuit inputs and required times at subcircuit outputs must be specified along with the functional specification of the subcircuit so that replacing the subcircuit with an optimized circuit automatically preserves the original functional and timing specifications. This scheme enables us to resynthesize subcomponents locally without violating the functional and timing requirements of the whole system.

A naive solution for this problem is to compute arrival times and required times using topological delays. This approach is commonly used in most timing optimization algorithms in the literature. Although this conservative approach gives a quick and conservative approximation to the true timing constraint, the resulting timing requirement may be more stringent than necessary since false paths in the surrounding network are completely ignored. Therefore, the timing constraint computed in this manner may prevent resynthesis from exploring the entire design space thereby leading to an unsatisfactory circuit. The goal of this section is to solve this problem more rigorously by taking false paths into account so that a more accurate and thus more flexible timing constraint is computed for the subnetwork.

3.4.1 Problem Formulation

We restrict our attention to combinational circuits. Sequential circuits using edge-triggered flip-flops, however, can be easily handled within the same framework by assuming all the flip-flop inputs and outputs as primary outputs and inputs respectively, where the required times and arrival times of those are determined by the clock edge minus the setup time and the clock edge itself re-

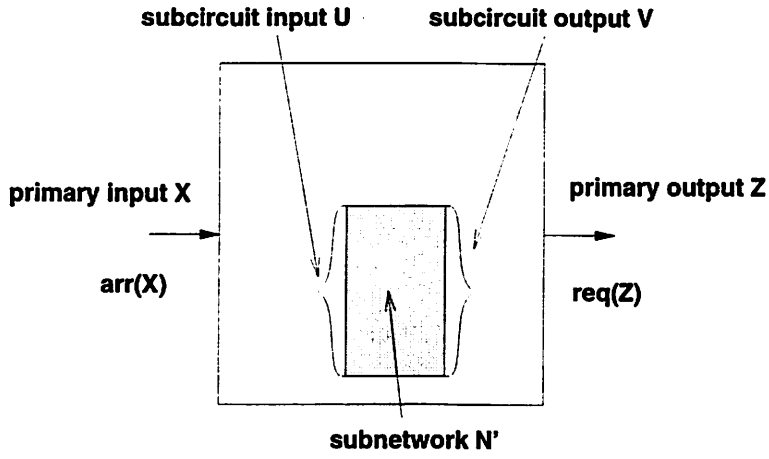

Figure 3.3: Resynthesis of a Subnetwork

spectively[9].

> **Local Timing Constraint Computation:** Given a Boolean network $\mathcal{N}$ and a subnetwork $\mathcal{N}'$ of $\mathcal{N}$, characterize the timing constraint of $\mathcal{N}'$ so that resynthesis of the subcircuit can be performed locally without violating the timing constraint of the entire network $\mathcal{N}$. Our assumption is that $\mathcal{N} \setminus \mathcal{N}'$ remains unchanged and only $\mathcal{N}'$ is to be resynthesized. $\mathcal{N}'$ must meet the condition that there is no path leading from a subcircuit output to a subcircuit input.

Let us introduce some notation for ease of explanation. Let $X = (x_1, \ldots, x_n)$ and $Z = (z_1, \ldots, z_m)$ be primary inputs and primary outputs of $\mathcal{N}$ respectively. Let $U = (u_1, \ldots, u_p)$ and $V = (v_1, \ldots, v_q)$ denote inputs and outputs of $\mathcal{N}'$ respectively. (See Figure 3.3.) We assume that arrival times at primary inputs X and required times at primary outputs Z are given. Our goal is to compute arrival times at subcircuit inputs U and required times at subcircuit outputs V by considering the effects of false paths in $\mathcal{N} \setminus \mathcal{N}'$ explicitly. One can think of this as mapping the timing requirement of the entire circuit onto the subcircuit.

[9]The use of a level-sensitive latch makes required time analysis more complex due to cycle stealing. Depending on the timing at a latch output, required time analysis needs to cross the latch backward and analyze the previous cycle. This remains as future work.

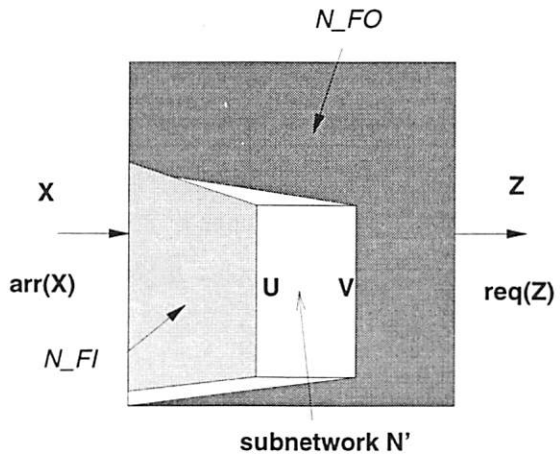

Figure 3.4: Local Timing Constraint of a Subnetwork

3.4.2 Computing Local Timing Constraints of Subcircuits

In this subsection, we show that the problem can be solved as a combination of functional arrival time analysis, which propagates arrival times forward from primary inputs to subcircuit inputs, and functional required time analysis introduced in Section 3.3, which propagates required times backward from primary outputs to subcircuit outputs.

Arrival Time Computation

The first step is to compute arrival times at the subcircuit inputs. The transitive fanin of the subcircuit inputs is extracted from $\mathcal{N}$, which we call $\mathcal{N}_{FI}$. (See Figure 3.4.) This network is then analyzed with a technique similar to functional arrival time analysis[10]. Notice that the primary outputs of $\mathcal{N}_{FI}$ are the subcircuit inputs, and the primary inputs of $\mathcal{N}_{FI}$ are a subset of primary inputs X of $\mathcal{N}$. The main difference between this problem and the standard functional arrival time analysis problem is that in the latter problem we only care about the latest arrival time for each output while in our problem interactions among arrival times of different outputs are of much interest to capture timing flexibility accurately[11].

Consider applying χ function analysis on $\mathcal{N}_{FI}$. For each subcircuit input $u_i \in U$, we list all

[10]To be precise, the delay of the fanin network is affected by changing its fanout, which is unknown in our setup since the fanout network is to be resynthesized. In this paper we do not take this load effect into consideration to simplify the explanation.

[11]Bahar *et al.*[BCH+94] proposed a functional arrival time analysis technique to compute input-vector dependent delay using ADDs[BFG+93]. This can be used as an alternative to the analysis below.

the potential arrival times at u_i, which is easily computed by propagating arrival time topologically from primary inputs to the subcircuit inputs while maintaining not a single latest arrival time but a set of arrival times at each node. Then, $\chi^{\tau}_{u_i,v}, v \in \{0,1\}$ is computed for each arrival time τ. Note that these χ functions are in terms of primary inputs X of $\mathcal{N}$. Then $\tilde{\chi}^{\tau}_{u_i} = \chi^{\tau}_{u_i,0} + \chi^{\tau}_{u_i,1}$ represents all the primary input vectors at X that make a signal at u_i stable by $t = \tau$. Assume that the list of potential arrival times at u_i is $\{\tau_1, \ldots, \tau_l\}$. Now the Boolean space $B^{|X|}$ can be partitioned into l disjoint sets $\{S_1, \ldots, S_l\}$ in terms of arrival times as follows.

$$\begin{aligned} S_1 &= \tilde{\chi}^{\tau_1}_{u_i} \\ S_k &= \tilde{\chi}^{\tau_k}_{u_i} \cdot \overline{\tilde{\chi}^{\tau_{k-1}}_{u_i}} \qquad (k = 2, \ldots, l) \end{aligned}$$

Note that $\tilde{\chi}^{\tau}_{u_i}$ increases as τ increases by its construction and S_k's are defined as differences between time-neighboring functions. The set $S_k (k = 1, \ldots, l)$ contains all the input vectors at X that make the signal u_i stable by $t = \tau_k$ but not by $t = \tau_{k-1}$, where $\tau_0 = -\infty$.

Once a partition of $B^{|X|}$ is computed for each subcircuit input, all the partitions are superimposed on $B^{|X|}$ to form a refined partition. This is equivalent to partitioning $B^{|X|}$ such that any input vector in a class has the same arrival time behavior at U.

The final step is to interpret this arrival time in terms of subcircuit inputs U so that the timing specification of the subcircuit is given locally in terms of its inputs. Remember that so far arrival time at U is computed in terms of X. The subcircuit, however, cannot distinguish input vectors applied at X unless they yield different vectors at U. Therefore, it is necessary to reinterpret this partition in terms of subcircuit inputs U to see what arrival time behaviors can be observed for each vector at U. This is easily computed from the partition of $B^{|X|}$ and the functionality of the transitive fanin network $\mathcal{N}_{FI}$. For each vector $\mathbf{u} \in B^{|U|}$ the set of all the vectors of $B^{|X|}$ yielding $\mathbf{u}$ at U is computed from the functionality of $\mathcal{N}_{FI}$. Using the partition of $B^{|X|}$ computed previously, one can list all the possible arrival times for the vector $\mathbf{u}$. All the latest arrival times are then extracted from them.

Let us illustrate this analysis with an example. Assume that the network in Figure 3.5 is $\mathcal{N}_{FI}$. For simplicity, we also assume that each gate has a unit delay and all the primary inputs arrive at $t = 0$. The χ function analysis gives the following.

$$\begin{aligned} \tilde{\chi}^{2}_{u_1} &= 1 \\ \tilde{\chi}^{1}_{u_1} &= \overline{x_1} \\ \tilde{\chi}^{2}_{u_2} &= 1 \\ \tilde{\chi}^{1}_{u_2} &= x_1 \end{aligned}$$

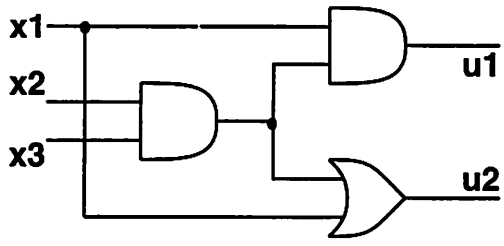

Figure 3.5: Example: $\mathcal{N}_{FI}$

The first two equations imply that if $x_1 = 0$, u_1 arrives at $t = 1$, but otherwise the signal arrives at $t = 2$. The last two equations then describe signal stability of u_2. If $x_1 = 1$, then u_2 arrives at $t = 1$, but otherwise the arrival time is $t = 2$. This can be summarized in the following table.

$x_1x_2x_3$	u_1u_2	$arr(u_1)\ arr(u_2)$
000	00	(1,2)
001	00	(1,2)
010	00	(1,2)
011	01	(1,2)
100	01	(2,1)
101	01	(2,1)
110	01	(2,1)
111	11	(2,1)

Now, notice that the subcircuit which $\mathcal{N}_{FI}$ feeds into cannot distinguish $x_1x_2x_3 = 011$ and 100 since both yield the same vector 01 at u_1u_2. Thus, when the subcircuit is resynthesized, we can only assume that the arrival time at the subcircuit input is either (1,2) or (2,1) when $u_1u_2 = 01$. Although it is possible to approximate this by having a single arrival time pair (2,2), it is not desirable since this is an over-constraint[12].

The following table is obtained by folding the table above in terms of the values of u_1u_2.

u_1u_2	$arr(u_1)\ arr(u_2)$
00	{(1,2)}
01	{(1,2),(2,1)}
10	$\{(\infty,\infty)\}$
11	{(2,1)}

[12]If an arrival time tuple is strictly earlier than another tuple, the former is dropped since the subcircuit $\mathcal{N}'$ must be synthesized under the worst-case scenario.

$\{(\infty,\infty)\}$ for $u_1u_2 = 10$ means that the subcircuit never observes the vector at the input. This corresponds to a satisfiability don't care [BRSVW87] among u_1 and u_2. It is interesting to observe that functional flexibility is captured in this framework naturally.

Required Time Computation

Computing required times at subcircuit outputs can be performed by analyzing a subnetwork of $\mathcal{N}$, $\mathcal{N}_{FO}$, with functional required time analysis. $\mathcal{N}_{FO}$ is the same network as $\mathcal{N}$ except all the subcircuit outputs V are relabeled as primary inputs. (See Figure 3.4.) Notice that required times at the subcircuit outputs are of interest. Since arrival times at X are known, there is no need to introduce leaf χ variables for those primary inputs of $\mathcal{N}_{FO}$ which are elements of X. Required time is computed for each vector $\mathbf{v} \in B^{|V|}$ at subcircuit outputs.

Towards More Accurate Timing Constraints

We consider a special case of the problem where no functional flexibility (e.g. don't cares) is explored in resynthesizing the subcircuit. In other words, the functional specification given for the subcircuit is the same functionality currently implemented. This allows us to compute a local timing constraint more accurately.

For arrival time computation at subcircuit inputs, instead of interpreting arrival time in terms of subcircuit inputs, we can simply keep arrival time in terms of primary inputs X. Required times at subcircuit outputs are computed for each vector $\mathbf{v} \in B^{|V|}$ in the previous subsection. Since the functionality of the subcircuit is preserved after resynthesis, the functionality of V in terms of X remains unchanged. Therefore, it is possible to interpret the required times in terms of primary inputs X. Now for each primary input vector $\mathbf{x} \in B^{|X|}$ we have a single arrival time at the subcircuit inputs and possibly multiple required times at the subcircuit outputs. One can then map this timing constraint to the subcircuit. Since arrival times and required times are coupled through X, analysis is more accurate compared to the one described before where arrival times and required times are computed independently.

3.4.3 Applications

Computation of local timing constraints has several practical applications.

The first is performance-oriented resynthesis. Suppose a combinational circuit was synthesized from a specification. Although one can optimize the entire circuit further to speed up late out-

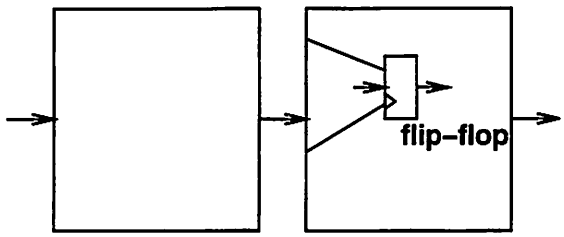

Figure 3.6: Timing Optimization of Hierarchical Circuits

puts, another promising approach is to extract a subcircuit containing part of critical paths and optimize it locally. This scheme is more likely to give a faster circuit because the circuit fed to synthesis is smaller. A similar approach is in fact taken in timing optimization techniques [SWBSV88, AMF97] published in the literature, but their delay computation is based on topological longest paths thereby failing to capture some of existing timing flexibility. Since our approach computes a local timing constraint of the subcircuit by considering false paths from the surrounding circuit, larger flexibility, i.e. less stringent timing requirement, is obtained, which makes resynthesis easier. An interesting subproblem of this application is to compute the true slack of a gate output, where the slack is calculated by taking false paths into account.

The second practical application is in hierarchical synthesis. Assume that a set of communicating sequential circuits does not meet a timing requirement, e.g. they do not satisfy a cycle time constraint. We now want to optimize component circuits one by one to speed up late signals. Optimizing the entire circuit as a single chunk is not desirable in this context because it destroys the hierarchy meaningful to designers and more importantly the whole circuit may be too large to handle for synthesis algorithms. Since the boundaries of components are not necessarily the inputs or the outputs of flip-flops, one may have to map arrival/required times for flip-flop inputs/outputs of the other components to the interface nodes of the component to be optimized. Figure 3.6 shows such a situation, where two sequential circuits are cascaded. Assume that a cycle time is given as a timing specification and we want to optimize only the left component with the right component unchanged. For simplicity, assume that there is a single flip-flop in the right component. The input of this flip-flop must become stable before the cycle time minus a set-up time. This constraint can be translated to that of the left component by propagating the required time at the flip-flop input backward through the combinational gates in the transitive fanin of the flip-flop to the boundary of the two components.

A similar scenario can arise in pure combinational synthesis. Consider a cascaded combinational circuit, where the driven circuit contains a fair amount of false paths. To resynthesize the

driving circuit effectively for improved performance it is critical to characterize the required times of the signals feeding the driven circuit as accurately as possible. Required times computed by topological analysis may completely mislead resynthesis due to the unawareness of false paths in the driven circuit.

3.5 Experimental Results

We have implemented on top of SIS[SSM+92] the exact and the two approximate algorithms for required time computation discussed in Section 3.3. The delay model we used in the experiments is the unit delay model. In all the experiments we set the required times of $t = 0$ at all the primary outputs and computed required times at primary inputs. All the Boolean operations in the exact and the first approximate methods are performed using BDDs [Bry86] while in the second approximate method SAT-based functional arrival time analysis [MSBSV93] is used.

The efficiency of the algorithms is dependent not only on the size of a network but also on the amount of reconvergence in the network. In the exact algorithm, we introduce one Boolean variable for each leaf χ variable. Thus, the existence of many reconvergences implies manipulation of χ functions of many input variables[13] in BDDs. The same observation is also true for the first approximate method, where a Boolean parameter variable is introduced for each leaf χ variable.

The second approximate algorithm is more robust than the first since the computation engine is a SAT solver. As mentioned before, an advantage of this approach is that any intermediate required time validated can be used as a safe approximation to the exact solution.

Table 3.2 shows a comparison between the exact and the approximate algorithms on MCNC benchmark circuits. CPU times are measured in seconds on DEC AlphaServer 8400 5/300. The exact algorithm and the first approximate algorithm were run with dynamic variable reordering [Rud93] being enabled. $*$ in the table denotes that the analysis gives a non-trivial required time strictly looser than topological analysis. The reason why the first approximate algorithm gives a looser constraint than the second in some examples is that the required times of values 0 and 1 for each primary input are distinguished in the first algorithm while the current implementation of the second algorithm only searches for value-independent required times for efficiency.

Table 3.3 shows CPU times of the second approximate algorithm on ISCAS combinational benchmark circuits. CPU times are measured in seconds on the same machine. The second column shows whether the algorithm was able to find non-trivial required times or not. The third and fourth

[13] In many ISCAS benchmark circuits the number of Boolean variables needed can easily go beyond hundreds.

circuit	#PI	#PO	CPU time (exact)	CPU time (approximate 1)	CPU time (approximate 2)
b9	41	21	-	1.1*	1.0
dalu	75	16	-	-	> 12 hours*
des	256	245	-	243.9	601.9
k2	45	45	1632.1*	22.3	86.0
rot	135	107	-	-	139.5*
t481	16	1	-	30631.2	12.4

Table 3.2: Required Time Computation – Exact vs. Approximate

circuit	Non-trivial required time?	CPU time first $r \neq r_{\perp}$ (in seconds)	CPU time r_{max} (in seconds)
C432	Yes	7.9	33.2
C499	No	-	40.1
C880	No	-	26.7
C1355	No	-	26.0
C1908	Yes	1.0	1356.4
C2670	Yes	2.8	2298.1
C3540	Yes	0.5	> 12 hours
C5315	Yes	77.7	359.6
C6288	Yes	1.0	> 12 hours
C7552	Yes	2.5	992.5

Table 3.3: Required Time Computation – ISCAS Examples

columns show CPU time spent until the first non-trivial required time was found, and CPU time for the entire analysis respectively. Although the algorithm could not finish on C3540 and C6288 within 12 hours of CPU time, it found non-trivial required times within a second. For C5315 36 primary inputs out of 178 had required times strictly better than the corresponding topological required times. For some of the primary inputs the additional accuracy over topological analysis was 3 time units, i.e. those primary inputs can safely arrive 3 time units later than their topological required times without violating the required times at the primary outputs. For C7552 70 out of 207 primary inputs had strictly better required times than their topological required times. The computed required time for one of the primary input was $t = -17$ while its topological required time was $t = -39$. This result shows that the impact of false paths in required time analysis can be significant.

3.6 Conclusions

We have studied how to perform required time analysis on combinational circuits more accurately than topological analysis, by taking false paths into account. The techniques proposed in this chapter, which is developed on top of the theory of functional arrival time analysis, can compute a more relaxed yet correct required time than the one computed by topological required time analysis. We have then shown that the combination of this functional required time analysis and existing functional arrival time analysis allows us to characterize an accurate timing constraint of a subcircuit given a timing constraint of an entire circuit. We will see other applications of functional required time analysis in the rest of this dissertation.

Even though this approach captures larger timing flexibility, existing timing optimization algorithms are not able to exploit the flexibility fully since timing specifications handled by timing optimization algorithms are of much simpler form than value-dependent constraints computed by our technique. A more sophisticated timing optimization algorithm compatible with the refined timing constraint proposed here is needed to fill this gap.

Another avenue for future research is to improve the efficiency of functional required time analysis by further approximations. In the current algorithms we distinguish between all potential required times at primary inputs. One possible approximation is to group them into clusters of neighboring required times conservatively. Controlling the number of clusters gives a trade-off between accuracy and CPU time for a more realistic delay model such as the mapped delay model.

In this chapter we have shown how to compute input-vector dependent required times at the primary inputs of a combinational circuit given traditional input-vector independent required

times at the primary outputs. A more general setup is to start with input-vector dependent required times at the outputs. This scenario naturally arises if functional required time analysis is performed in a hierarchical fashion. Although it is possible to extend the theory to capture this general case, its practical impact is yet to be determined.

Chapter 4

Delay Characterization of Combinational Modules

Accurate delay characterization of circuits at various levels of abstraction has always been one of the main focuses of timing analysis research. This chapter addresses the delay characterization problem of combinational modules at the gate level. A *combinational module* is a combinational circuit that can be used under any arrival time condition at primary inputs. An intellectual-property(IP) module is one example.

We discuss how to compute a false-path-aware delay abstraction of a combinational module. A delay abstraction is a compact representation of the delay information of the module, which carries pin-to-pin delay for each primary-input/primary-output pair[1]. The delay can be dependent on input vectors provided to the module. The existence of false paths is captured in the delay abstraction by keeping effective pin-to-pin delays instead of topological delays. The internal structural details of the module are abstracted away.

This chapter is organized as follows. After a brief introduction to the problem in Section 4.1, Section 4.2 shows that the false-path-aware delay abstraction of a combinational module can be computed exactly using functional required time analysis discussed in Chapter 3. A false-path-aware delay abstraction is given for an example in Section 4.3. Section 4.4 discusses delay analysis using delay abstractions. After introducing a novel framework for comparing the accuracy of delay abstractions in Section 4.5, we study a different approach to computing delay abstractions

[1]To the best of our knowledge Note *et al.* [NCGM92] were the first to use delay abstractions of this form. Kuehlman and Bergamaschi employed this idea later in [KB92]. Kobayashi and Malik [KM95, KM97] studied compact representations of delay abstractions. However, delay abstractions were independent of input vectors in the previous work.

using functional arrival time analysis in Section 4.6. Sections 4.7 and 4.8 discuss approximate computation of delay abstractions based on functional required time analysis and functional arrival time analysis respectively. Section 4.9 generalizes delay abstractions to the case where gate delays are variables. Section 4.10 summarizes related work in the literature. The chapter is concluded in Section 4.11

4.1 Introduction

The major difficulty in computing a false-path-aware delay abstraction of a combinational module is in the requirement that a delay abstraction be valid and accurate under any arrival time condition at the inputs. State-of-the-art path sensitization conditions (e.g. viability [MB91], floating-mode [CD93] and XBD0 [MSBSV93]) exploit arrival times at primary inputs to enable exact false path detection. Therefore, a direct application of those path sensitization conditions is not appropriate in this context since we cannot assume specific arrival times.

Several sensitization conditions are known such as static co-sensitization [DKM93] and the Brand-Iyengar condition [BI88] that do not refer to arrival times at the inputs. The use of such sensitization conditions guarantees the validity of a resulting delay abstraction under all arrival times. However, they are not as accurate as arrival-time dependent sensitization conditions. It appears that the two requirements, the validity and the accuracy, conflict with each other.

Our goal is to resolve this conflict positively by showing that an accurate delay abstraction can be computed under the XBD0 model without assuming a specific arrival time condition. Specifically, we show that the problem is reducible to functional required time analysis discussed in Chapter 3.

4.2 Delay Characterization by Functional Required Time Analysis

Let $\mathcal{M}$ be a single-output combinational module under analysis. Let $X = (x_1, \ldots, x_n)$ and z be the primary inputs and the primary output of $\mathcal{M}$ respectively.

Consider delay from the primary inputs X to the primary output z. The standard way to define the delay of a module is by computing the stable time of each output given arrival times at all the primary inputs. The difference between the output stable time and the arrival time of each input defines the delay from the input to the output. This approach, however, is not applicable to our setting since the arrival times at the primary inputs are unknown.

Our objective is to capture the timing characteristics of a given module valid and accurate under *any* surrounding environment. To achieve this the delay of a module is defined in a different way.

We first set a required time, say $t = 0$[2], to the output and analyze the given circuit to see when the primary inputs are required to be stable so that the output becomes stable by the required time[3]. The delay from an input to the output is then defined as the difference between the required time at the output ($t = 0$) and that of the input. This is exactly the same problem as functional required time analysis discussed in Chapter 3.

Functional required time analysis gives $\mathcal{T}_z \subseteq B^n \times R^n$, where a set of timing tuples is given for each input vector. Given an input vector $\mathbf{x} \in B^n$, each n-tuple $T = (t_1, \ldots, t_n)$ such that $(\mathbf{x}, T) \in \mathcal{T}_z$ represents valid required times at the inputs. The interpretation of a tuple T is that the output is guaranteed to be stable at $t = 0$ if the primary input vector $\mathbf{x} = (v_1, \ldots, v_n)$ arrives at or before $T = (t_1, \ldots, t_n)$ respectively. We assume that only those timing tuples not subsumed by another are contained under each input vector in $\mathcal{T}_z$ as in Section 3.3.1. Since the required time analysis is done in the XBD0 model, required times computed follow the monotone speedup property [MB91], i.e. if a signal arrives earlier than the required time specified, it never worsens the stability of the output.

Recall that $\mathcal{T}_z$ may contain more than one timing tuple for a given input vector, in which case each of the timing tuples captures a different permissible signal arrival behavior at the primary inputs. The *delay abstraction (timing abstraction)* D_z of $\mathcal{M}$ is then defined as

$$D_z = \{(\mathbf{x}, (-t_1, \ldots, -t_n)) \mid (\mathbf{x}, (t_1, \ldots, t_n)) \in \mathcal{T}_z\}.$$

The sign of required times is flipped in each timing tuple since each required time is subtracted from 0, the required time at the output.

To deal with a multiple-output module, the same analysis is applied to the transitive fanin cone of each output independently. Each output has its own delay abstraction this way.

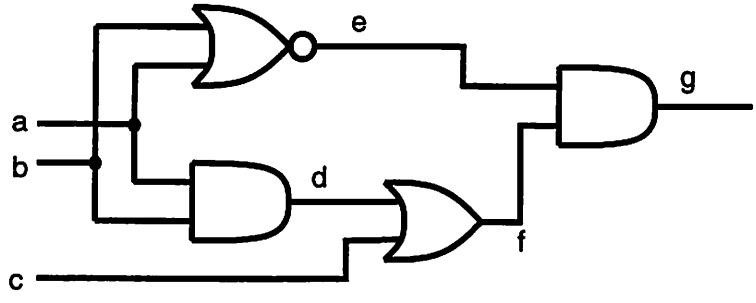

Figure 4.1: Example: A Combinational Module $\mathcal{M}$

4.3 Example

Consider a circuit shown in Figure 4.1, taken from [BI88]. Assume the unit delay model. The delay abstraction D_{exact} computed by functional required time analysis is:

abc	$d_{a\to g}d_{b\to g}d_{c\to g}$
000	$\{(3,-\infty,2),(-\infty,3,2)\}$
001	$\{(2,2,2)\}$
010	$\{(3,-\infty,2),(-\infty,2,-\infty)\}$
011	$\{(-\infty,2,-\infty)\}$
100	$\{(2,-\infty,-\infty),(-\infty,3,2)\}$
101	$\{(2,-\infty,-\infty)\}$
110	$\{(2,-\infty,-\infty),(-\infty,2,-\infty)\}$
111	$\{(2,-\infty,-\infty),(-\infty,2,-\infty)\}$

where $-\infty$ denotes that the availability of the corresponding input is irrelevant to the stability of the output.

The delay abstraction is a table indexed by input vectors. Given an input vector, the delay abstraction gives a set of delay tuples. Each delay tuple captures a distinct delay characteristic of the module under the input vector. For example, consider an input vector 000. There are two delay

[2]The absolute value of $t = 0$ is not important since we are only interested in time differences.

[3]A similar formulation was used by Min *et al.* [MZL96], in which path sensitization of combinational networks under *arbitrary* input waveforms was discussed without considering a specific arrival time condition at primary inputs. They used this formulation to compute an input waveform to sensitize a given path. The delay model used was the fixed delay model, where each gate has a constant delay. The discussion in this chapter is based on the XBD0 model. Therefore, we implicitly assume that once the value of an input waveform changes from an unknown value X to a Boolean constant, it keeps the constant. In other words, we consider only a specific class of input waveforms, which is natural in synchronous designs. Consideration of *arbitrary* input waveforms may be useful in aggressive design styles such as wave pipelining.

tuples $(3,-\infty,2)$ and $(-\infty,3,2)$ under the vector. The first tuple guarantees that the output becomes stable even without a stable value of b, in which case the effective delays from a and c to the output are 3 and 2 respectively. All the paths from b to the output can be thought of as false when this delay tuple is used. The second tuple captures a symmetric case where the roles of a and b are switched. In this case all the paths from a to the output are false. The key to the accuracy under arbitrary arrival times is in this capability of maintaining more than one delay tuple for the same input vector. This flexibility allows us to choose the delay tuple that gives the most accurate delay estimate at the output under a given arrival time condition.

Let us contrast this with the delay abstraction corresponding to topological delay analysis. Since the longest topological path lengths from a, b and c are 3, 3 and 2 respectively, the delay abstraction $D_{topological}$ is:

abc	$d_{a\to g}d_{b\to g}d_{c\to g}$
000	{(3,3,2)}
001	{(3,3,2)}
010	{(3,3,2)}
011	{(3,3,2)}
100	{(3,3,2)}
101	{(3,3,2)}
110	{(3,3,2)}
111	{(3,3,2)}.

$D_{topological}$ is independent of input vectors since topological delay analysis ignores the functionality of the circuit. No false paths are detected. The accuracy gain in D_{exact} is clear by comparing the two delay abstractions.

4.4 Delay Analysis using Delay Abstractions

Given an input vector $\mathbf{x}$ and arrival times $A = (arr(x_1),\ldots,arr(x_n)) = (a_1,\ldots,a_n)$, the signal stable time at the output z can be determined by using a delay abstraction D. Suppose that D has multiple delay tuples $\delta_1,\ldots,\delta_m$ for $\mathbf{x}$. For each delay tuple $\delta_i = (d_{i,1},\ldots,d_{i,n})(i = 1,\ldots,m)$ the signal stable time s_i at z under δ_i is computed as

$$s_i = \max_j (a_j + d_{i,j}).$$

Since all delay tuples are valid, the signal stable time s at z is determined by taking the earliest time among s_i's.

$$s = \min_i s_i = \min_i \max_j (a_j + d_{i,j})$$

For ease of exposition we introduce a function *delay_propagate* to compactly represent this operation. The function takes a delay abstraction $D \subseteq B^n \times R^n$, an input vector $\mathbf{x} \in B^n$ and arrival times $A = (a_1, \ldots, a_n) \in R^n$ as inputs, and returns a signal stable time at the output $s \in R$. The function is defined in exactly the same way as in the last paragraph.

Definition 4.1

$$delay\_propagate(D, \mathbf{x}, A) = \min_{(\mathbf{x},(d_{i,1},\ldots,d_{i,n}))\in D} \max_j (a_j + d_{i,j})$$

Consider the circuit in Figure 4.1 again. Let $\mathbf{x} = 000$ and $A = (arr(a), arr(b), arr(c)) = (1,0,0)$. There are two delay tuples $(3, -\infty, 2)$ and $(-\infty, 3, 2)$ under input vector 000. If the first tuple is used, the arrival time estimate s_1 at the output is:

$$s_1 = \max(1+3, 0-\infty, 0+2) = \max(4, -\infty, 2) = 4.$$

If the second tuple is used, the estimate s_2 is:

$$s_2 = \max(1-\infty, 0+3, 0+2) = \max(-\infty, 3, 2) = 3.$$

Finally, we choose the best estimate s by taking the minimum of s_1 and s_2.

$$s = \min(s_1, s_2) = \min(4, 3) = 3$$

Notice that once the exact delay abstraction of a module is computed, false path analysis can be performed under any arrival time condition only using the delay abstraction.

Definition 4.2 *Given delay tuples* $(d_1, \ldots, d_n)$ *and* $(d'_1, \ldots, d'_n) \in R^n$, $(d_1, \ldots, d_n) \trianglelefteq (d'_1, \ldots, d'_n)$ *if and only if* $\forall i, d_i \leq d'_i (i = 1, \ldots, n)$.

Lemma 4.1 $\trianglelefteq$ *is transitive.*

Proof Trivial from the definition of $\trianglelefteq$. □

We are ready to introduce a notion of *reduced* delay abstractions.

Definition 4.3 *A delay abstraction D is said to be* reduced *if for every input vector* x *and any pair of delay tuples* $(d_1,\ldots,d_n)$ *and* $(d'_1,\ldots,d'_n)$ *under* x

$$(d_1,\ldots,d_n) \not\trianglelefteq (d'_1,\ldots,d'_n)$$

This property is automatically satisfied if a delay abstraction is computed by functional required time analysis. Any delay abstraction can be made reduced by dropping delay tuples subsumed by others under the same input vector. Notice that the reduced delay abstraction has the same information as the original since the delay tuples dropped during the reduction are never useful in timing analysis. For example, suppose that a delay abstraction D contains delay tuples $(d_1,\ldots,d_n)$ and $(d'_1,\ldots,d'_n)$ under an input vector x, where

$$(d_1,\ldots,d_n) \trianglelefteq (d'_1,\ldots,d'_n).$$

Under an arrival time condition $A = (a_1,\ldots,a_n)$, the stable time estimate s of the output using the first tuple is:

$$s = \max_j (a_j + d_j).$$

The stable time estimate s' under the second tuple is:

$$s' = \max_j (a_j + d'_j).$$

Since $d_j \leq d'_j$ for any j, $s \leq s'$ for any A. Thus, the second tuple $(d'_1,\ldots,d'_n)$ can never give a strictly better estimate, and thus is safe to remove.

From now on we assume that delay abstractions are reduced.

4.5 Comparing Delay Abstractions

In the next section we will present a different approach to computing a delay abstraction of a combinational module using functional arrival time analysis, and argue its accuracy and correctness. To make a rigorous argument possible we introduce a criterion for comparing delay abstractions. Specifically a partial order over delay abstractions is defined, where $D_1 \preceq D_2$ intuitively means that stable time estimate at the output using delay abstraction D_1 is no later than that using delay abstraction D_2 under *any* vector and *any* arrival time condition at the primary inputs.

Let D_{exact} be the delay abstraction of a combinational module computed by exact functional required time analysis. If a delay abstraction D meets $D_{exact} \preceq D$, D is a correct delay abstraction since the use of D in delay estimation never underestimates exact stable time at the output computed by D_{exact} regardless of a choice of vectors and arrival times at the inputs.

On the other hand, if $D_{exact} \not\preceq D$, there exists a pair of an input vector and an arrival time condition at the inputs under which the use of D gives arrival time at the output strictly earlier than that computed by D_{exact}. Since this is a delay underestimation, any delay characterization method that can give such D is incorrect.

The partial order $\preceq$ over delay abstractions is defined as follows.

Definition 4.4 *Let D_1 and D_2 be the delay abstractions of a single-output combinational module $\mathcal{M}$. $D_1 \preceq D_2$ if for every input-vector/delay-tuple pair $(\mathbf{x}, (d_1, \ldots, d_n)) \in D_2$, there exists an input-vector/delay-tuple pair $(\mathbf{x}, (d'_1, \ldots, d'_n)) \in D_1$ such that $(d'_1, \ldots, d'_n) \trianglelefteq (d_1, \ldots, d_n)$.*

Theorem 4.1 $\forall \mathbf{x}, A = (a_1, \ldots, a_n), delay\_propagate(D_1, \mathbf{x}, A) \leq delay\_propagate(D_2, \mathbf{x}, A)$ *if and only if* $D_1 \preceq D_2$.

Proof

$\Leftarrow$ Let $\delta = (d_1, \ldots, d_n)$ be the delay tuple in D_2 under $\mathbf{x}$ that gives the earliest stable time s_2 at the output under A.

$$s_2 = delay\_propagate(D_2, \mathbf{x}, A) = \max_i (a_i + d_i)$$

From the assumption, there exists a delay tuple $\delta' = (d'_1, \ldots, d'_n) \trianglelefteq (d_1, \ldots, d_n)$ in D_1 under $\mathbf{x}$. Therefore, the signal stable time s_1 at the output is:

$$s_1 = delay\_propagate(D_1, \mathbf{x}, A) \leq \max_i (a_i + d'_i) \leq \max_i (a_i + d_i) = s_2.$$

$\Rightarrow$ Suppose for a contradiction that $(\mathbf{x}, (d_1, \ldots, d_n)) \in D_2$ does not have an input-vector/delay-tuple pair $(\mathbf{x}, (d'_1, \ldots, d'_n))$ in D_1 such that $(d'_1, \ldots, d'_n) \trianglelefteq (d_1, \ldots, d_n)$. Let $\mathbf{x}$ be this input vector and $A = (-d_1, \ldots, -d_n)$. The signal stable time s_2 at the output under D_2 is:

$$s_2 = delay\_propagate(D_2, \mathbf{x}, A) = 0$$

by using the delay tuple.

Now consider D_1. Let $(d'_1, \ldots, d'_n)$ be a delay tuple in D_1 under $\mathbf{x}$. There exists at least one i such that $d'_i > d_i$ for any choice of $(d'_1, \ldots, d'_n)$. Therefore, the signal stable time at the output s_1 is:

$$s_1 = delay\_propagate(D_1, \mathbf{x}, A) = \min_{(d'_1, \ldots, d'_n)} \max_i (-d_i + d'_i) > 0,$$

which is later than s_2. A contradiction.

□

We prove several properties of $\preceq$.

Theorem 4.2 *$\preceq$ is reflexive.*

Proof $\forall D, D \preceq D$. This is trivial from the definition. □

Theorem 4.3 *$\preceq$ is transitive.*

Proof Suppose that $D_1 \preceq D_2$ and $D_2 \preceq D_3$. We want to prove that $D_1 \preceq D_3$. Let $(\mathbf{x}, (d_1, \ldots, d_n)) \in D_3$. Since $D_2 \preceq D_3$, there exists $(\mathbf{x}, (d_1', \ldots, d_n')) \in D_2$ such that $(d_1', \ldots, d_n') \trianglelefteq (d_1, \ldots, d_n)$. Furthermore, since $D_1 \preceq D_2$, there exists $(\mathbf{x}, (d_1'', \ldots, d_n'')) \in D_1$ such that $(d_1'', \ldots, d_n'') \trianglelefteq (d_1', \ldots, d_n')$. By combining the two results with the transitivity of $\trianglelefteq$ (Lemma 4.1), for any $(\mathbf{x}, (d_1, \ldots, d_n)) \in D_3$, there exists $(\mathbf{x}, (d_1'', \ldots, d_n'')) \in D_1$ such that $(d_1'', \ldots, d_n'') \trianglelefteq (d_1, \ldots, d_n)$. □

Theorem 4.4 *$\preceq$ is antisymmetric.*

Proof Suppose that $D_1 \preceq D_2$ and $D_2 \preceq D_1$. We want to prove that $D_1 = D_2$. Since $D_1 \preceq D_2$, for any $(\mathbf{x}, (d_1, \ldots, d_n)) \in D_2$ there exists $(\mathbf{x}, (d_1', \ldots, d_n')) \in D_1$ such that $(d_1', \ldots, d_n') \trianglelefteq (d_1, \ldots, d_n)$. Furthermore, since $D_2 \preceq D_1$, for $(\mathbf{x}, (d_1', \ldots, d_n')) \in D_1$ above, there exists $(\mathbf{x}, (d_1'', \ldots, d_n'')) \in D_2$ such that $(d_1'', \ldots, d_n'') \trianglelefteq (d_1', \ldots, d_n')$. By the transitivity of $\trianglelefteq$ (Lemma 4.1) $(d_1'', \ldots, d_n'') \trianglelefteq (d_1, \ldots, d_n)$. Note that both of the delay tuples are under $\mathbf{x}$ in D_2. Suppose $(d_1'', \ldots, d_n'') \neq (d_1, \ldots, d_n)$ for a contradiction. This means that under $\mathbf{x}$ there are two distinct delay tuples where one is dominated by the other. This contradicts the assumption that D_2 is reduced. Therefore, $(d_1'', \ldots, d_n'') = (d_1, \ldots, d_n)$, and thus $(d_1', \ldots, d_n') = (d_1, \ldots, d_n)$, showing that any delay tuple under $\mathbf{x}$ in D_2 is also a delay tuple under $\mathbf{x}$ in D_1. The symmetric argument proves the other direction. □

4.6 Computing Delay Abstractions by Functional Arrival Time Analysis

Section 4.2 showed that functional required time analysis of a combinational module directly gives a false-path-aware delay abstraction of the module without assuming a specific arrival

time condition at primary inputs. Since no assumption is made about arrival time conditions at the inputs, the resulting delay abstraction is valid under any arrival time condition. Furthermore, the delay abstraction captures enough information so that a delay estimate is accurate under any arrival time condition.

In this section the delay characterization problem is approached from a different angle. We argued previously that the use of functional arrival time analysis in computing a delay abstraction is not appropriate. The reason was that state-of-the-art techniques for functional arrival time analysis take as input an arrival time condition at primary inputs, which can lead to a delay abstraction specialized for the given arrival time condition. The validity of the delay abstraction is questionable if it is used under different arrival times.

Suppose a delay abstraction is computed by functional arrival time analysis under a specific arrival time condition. If we can prove that the use of the resulting delay abstraction never leads to delay underestimation under any other arrival time condition, it is conservative and thus safe to use. On the other hand, if there exists some input vector and some arrival time condition under which the delay abstraction gives too optimistic delay estimation, the possibility of delay underestimation makes the delay abstraction unsafe. We will formalize this idea and show that the validity of delay abstractions depends on what path sensitization condition is used in functional arrival time analysis. Two different approaches will be presented. Throughout this chapter D^1 denotes a delay abstraction computed by the first approach while D^2 denotes a delay abstraction computed by the second approach.

4.6.1 Delay Abstractions by Path Classification

The first approach is based on the classification of paths into true and false paths under an arrival time condition. We first choose an arbitrary arrival time condition and perform functional arrival time analysis using a path sensitization condition thereby classifying all the paths of the circuit into true paths and false paths. All the false paths are then ignored and the effective delay from an input to an output is computed as the longest *true* path between the two terminals. A false-path-aware delay abstraction is constructed this way by using functional arrival time analysis.

The idea of ignoring false paths in delay analysis has been standard in commercial timing analysis tools [BS95]. In this scenario designers specify false paths, which are then ignored during topological analysis. If false paths are identified and ignored under the same arrival time condition, the approach is clearly correct. However, a typical usage of such a timing analysis tool is that false

paths are specified once and for all, which are assumed to be false in all subsequent analyses even under different arrival times. The correctness of this approach hinges on whether the false paths specified are conservative enough for any arrival time condition. To the best of our knowledge, the validity of this type of analysis has never been discussed in the literature.

Arrival-Time Independent Path Sensitization Conditions

Although most of the path sensitization conditions proposed recently refer to arrival times at primary inputs to determine the falsity of paths, there are some sensitization conditions independent of arrival times. Static co-sensitization [DKM93] and the Brand-Iyengar [BI88] conditions are in the category. For this class of sensitization conditions we do not need to select arrival times in the first place. Since the correctness of these conditions is guaranteed implicitly under all arrival time conditions, the resulting delay abstraction is a conservative delay abstraction to D_{exact}.

Theorem 4.5 *Let $D^1_{co-sens}$ be the delay abstraction constructed by static co-sensitization analysis of a single-output combinational module $\mathcal{M}$. Then $D_{exact} \preceq D^1_{co-sens}$.*

Proof See [DKM93]. □

Theorem 4.6 *Let D^1_{BI} be a delay abstraction constructed by Brand-Iyengar analysis of a single-output combinational module $\mathcal{M}$ under some fanin ordering. Then $D_{exact} \preceq D^1_{BI}$.*

Proof See [BI88]. □

Let us apply these two sensitization conditions to the circuit in Figure 4.1. In the following we focus on the case where an input vector is 000. Other input vectors can be analyzed in the same way.

First, static co-sensitization analysis is performed on the circuit. Paths (a, e, g) and (b, e, g) are false since g has a controlled value 0 of the AND gate while e does not have a controlling value 0. The other three paths are true. Since the longest topological path from each input is still true, the delay tuple for this vector is $(3, 3, 2)$, which is the same as the one computed by topological analysis.

Let us try Brand-Iyengar analysis. Brand-Iyengar analysis requires that a fanin ordering be given at each gate. Depending on the orderings the accuracy of delay abstractions varies.

Suppose $a < b$ at gate d and $e < f$ at gate g[4]. Paths (a,e,g) and (b,e,g) are false again since at the fanins of g the other fanin f, which is after e in the ordering, has a controlling value 0. The topological longest path from a, (a,d,f,g), is also false since at gate d the other fanin $b > a$ has a controlling value. All the other paths are true. Notice that both of the paths from a are false. This results in a delay tuple $(-\infty,3,2)$, which is one of the delay tuples in D_{exact} under input vector 000.

Let us keep the fanin ordering at d and flip the ordering at g to $f < e$. This change makes the two paths (a,e,g) and (b,e,g), which were false before, true since 0 at f does not block the paths any more since $f < e$. The falsity and the truth of the other paths remain the same. The delay tuple based on this analysis is $(2,3,2)$, which is less accurate than the delay tuple in the previous analysis $(-\infty,3,2)$.

The circuit is symmetric in terms of a and b. Therefore, by flipping the ordering at d, we obtain the following two tuples for each of the above: $(3,-\infty,2),(3,2,2)$. $(3,-\infty,2)$ is the same as the other delay tuple in D_{exact} under input vector 000.

Depending on fanin orderings, the accuracy of delay tuples varies. It can be as accurate as one of the delay tuples in D_{exact}, but can be as conservative as the topological delay tuple.

Viability Analysis

Unlike the two sensitization conditions discussed so far, viability [MB91] refers to arrival times at primary inputs. Thus, the same path can be true under some arrival time condition, but false under another even for the same input vector. This leads to a question whether a delay abstraction computed by viability analysis under some arrival time condition is still valid under a different arrival time condition. Daniel Brand has recently pointed out that viability analysis should give a valid delay abstraction no matter how arrival times are chosen by observing similarity between viability and Brand-Iyengar analyses [Bra98]. We will formalize this claim by providing a proof.

Let $\mathcal{M}$ be a single-output combinational module with primary inputs $x_1,\ldots,x_n$ and primary output z. Consider two arrival time conditions $A_1 = (a_{1,1},\ldots,a_{1,n})$ and $A_2 = (a_{2,1},\ldots,a_{2,n})$.

Definition 4.5 *Let $p = (g_0,\ldots,g_k)$ be a path from a primary input $g_0 = x_j$ to a gate g_k. Let $A = (a_1,\ldots,a_n)$ be an arrival time condition. Then,*

$$path\_delay(p,A) = a_j + \sum_{i=1}^{k} d(g_i).$$

[4]Since the orderings at e and f are irrelevant for this input vector, we do not specify them to simplify the argument.

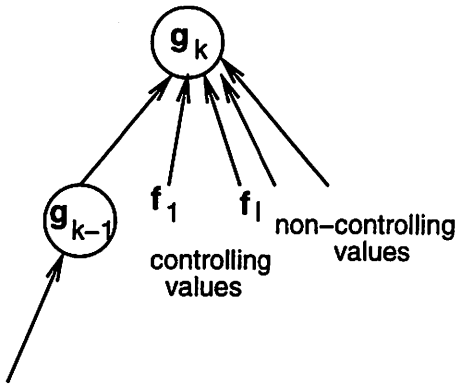

Figure 4.2: Proof of Theorem 4.7

Theorem 4.7 *Let* x, A_1 *and* A_2 *be an input vector and arrival time conditions chosen arbitrarily. Let* $p = (g_0, \ldots, g_k)$ *be a path from primary input* g_0 *to some node* g_k. *If* p *is viable for* x *under* A_1, *then there exists a path* p' *from a primary input to* g_k *that is viable for the same vector* x *under* A_2 *such that* $path\_delay(p, A_1) \leq path\_delay(p', A_1)$.

Proof We prove this by induction on the structure of $\mathcal{M}$. More specifically we choose a topological order of the nodes in $\mathcal{M}$, and prove the theorem inductively on this order. Viability of a path is always discussed for the input vector x in this proof unless otherwise noted.

Base case: $p = (g_0)$ consists of only a primary input node. Any such p is viable under any input vector and any arrival time condition. Therefore, $p' = p$, and we are done.

Induction: Assume that the last node of p is not a primary input.

If p is viable under A_2: $p' = p$, and we are done.

If p is not viable under A_2: We need to construct p' that meets the condition.

Consider the subpath q of p from g_0 to gate g_{k-1}, $q = (g_0, \ldots, g_{k-1})$. By induction, there exists a path q' from a primary input to g_{k-1} viable under A_2 such that $path\_delay(q, A_1) \leq path\_delay(q', A_1)$. Let $\tilde{p}$ be the path q' extended by g_k.

If $\tilde{p}$ is viable under A_2:

$$\begin{aligned} path\_delay(p, A_1) &= path\_delay(q, A_1) + d(g_k) \\ &\leq path\_delay(q', A_1) + d(g_k) \\ &= path\_delay(\tilde{p}, A_1). \end{aligned}$$

Therefore, $p' = \tilde{p}$ meets the condition.

If $\tilde{p}$ is not viable under A_2: Since $\tilde{p}$ is not viable under A_2, there exists a non-empty set S of side inputs at g_k with a controlling value of g_k. Let $S = \{f_1, \ldots, f_l\}$. (See Figure 4.2.) For each $f_i \in S$, consider a viable path r_i under A_1 from a primary input to f_i that maximizes $path\_delay(r_i, A_1)$. We have $path\_delay(q, A_1) \leq path\_delay(r_i, A_1)$ for $i = 1, \ldots, l$ since otherwise p would not be viable under A_1.

By induction, for each r_i, there exists a viable path r_i' under A_2 from a primary input to f_i such that $path\_delay(r_i, A_1) \leq path\_delay(r_i', A_1)$.

Now we would like to show that there exists $j \in \{1, \ldots, l\}$ such that r_j' can be extended with g_k by maintaining viability under A_2. Suppose none can be extended for a contradiction. For each f_i let $t_i = \max_s path\_delay(s, A_2)$, where the max operation is taken over all viable paths s from a primary input to f_i under A_2. For g_{k-1}, let $t_0 = \max_s path\_delay(s, A_2)$, where the max operation is taken over all viable paths s from a primary input to g_{k-1} under A_2. Since no r_i' can be extended, $t_0 < t_i (i = 1, \ldots, l)$ and g_{k-1} has a controlling value. However, under this condition $\tilde{p}$ is viable under A_2. A contradiction.

From the above, at least one r_i' can be extended with g_k while remaining viable under A_2. Let r_j' be such r_i'. Let p' be the extended path.

$$\begin{aligned} path\_delay(q, A_1) &\leq path\_delay(r_j, A_1) \\ &\leq path\_delay(r_j', A_1). \end{aligned}$$

By adding $d(g_k)$ to the inequality,

$$\begin{aligned} path\_delay(p, A_1) &= path\_delay(q, A_1) + d(g_k) \\ &\leq path\_delay(r_j', A_1) + d(g_k) \\ &= path\_delay(p', A_1). \end{aligned}$$

□

Corollary 4.1 *Let A_1 and A_2 be arbitrary arrival time conditions. For any path $p = (g_0, \ldots, g_k)$ from a primary input to a primary output viable under* **x** *and A_1, there exists a path p' from a primary input to g_k viable under* **x** *and A_2 such that $path\_delay(p, A_1) \leq path\_delay(p', A_1)$.*

The intuitive meaning of Corollary 4.1 is as follows. Let A_1 be the arrival time condition of the current interest. Suppose that we have analyzed this circuit under A_2 and classified all the paths into viable and non-viable under A_2. The corollary guarantees that we can safely pretend that this classification of paths remains the same under A_1 for the purpose of estimating output stable time. Even though the stable time may be overestimated, there is no risk of underestimation.

To be more concrete, let p be a viable path under the current arrival time condition. If p is the longest viable path under A_1, the actual output stable time is $path\_delay(p,A_1)$. Now, the corollary guarantees the existence of a viable path p' under A_2 where $path\_delay(p,A_1) \leq path\_delay(p',A_1)$. By assuming p' is viable under A_1, the output stable time under A_1 is estimated as $path\_delay(p',A_1)$, which is no earlier than the actual arrival time $path\_delay(p,A_1)$ by construction of p'. Thus, it is safe to reuse the classification of viable and non-viable paths under A_2 to A_1.

Notice that A_1 and A_2 above were chosen arbitrarily. Therefore, if viability analysis is performed under some arrival time condition, the resulting delay abstraction computed by ignoring false paths under the condition is a valid delay abstraction under any arrival time condition.

Theorem 4.8

$$\forall \mathbf{x}, A, A', delay\_propagate(D^1_{viability}(A), \mathbf{x}, A) \leq delay\_propagate(D^1_{viability}(A'), \mathbf{x}, A).$$

Proof In Corollary 4.1, let $A_1 = A$ and $A_2 = A'$. For any viable path p under $\mathbf{x}$ and A, there exists a viable path p' under $\mathbf{x}$ and A' that never gives an underestimate of the delay due to p. Thus the claim holds. □

Theorem 4.9 *For simple-gate circuits:*

$$\forall \mathbf{x}, A, delay\_propagate(D_{exact}, \mathbf{x}, A) = delay\_propagate(D^1_{viability}(A), \mathbf{x}, A).$$

Proof Under the same arrival time condition, XBD0 analysis gives the same delay estimate as viability [MSBSV93] for simple-gate circuits. Thus the equality holds. □

Theorem 4.10

$$\forall A, D_{exact} \preceq D^1_{viability}(A)$$

path	Arrival time condition $(arr(a),arr(b),arr(c))$				
	(0,0,0)	(1,0,0)	(0,1,0)	(2,0,0)	(0,2,0)
(a,e,g)	true	true	true	false	false
(b,e,g)	true	true	true	false	false
(a,d,f,g)	true	false	true	false	true
(b,d,f,g)	true	true	false	true	false
(c,f,g)	true	true	true	true	true
delay tuple	(3,3,2)	(2,3,2)	(3,2,2)	$(-\infty,3,2)$	$(3,-\infty,2)$

Table 4.1: Delay Characterization by Functional Arrival Time Analysis: Viability

Proof From Theorems 4.8 and 4.9,

$$\begin{aligned} \forall \mathbf{x},A,A', delay\_propagate(D_{exact},\mathbf{x},A) &= delay\_propagate(D^1_{viability}(A),\mathbf{x},A) \\ &\leq delay\_propagate(D^1_{viability}(A'),\mathbf{x},A). \end{aligned}$$

By the only-if part of Theorem 4.1 $D_{exact} \preceq D^1_{viability}(A')$. Renaming A' to A completes the proof. □

Now that the correctness of the approach is proved, let us see how it works in practice using the example in Figure 4.1. We will analyze the circuit under various arrival time conditions to see the effect of arrival times to the accuracy of the resulting delay abstractions.

Table 4.1 summarizes the result for input vector 000. The falsity and the truth of paths vary depending on arrival times, which affects the accuracy of the computed delay tuples. Under arrival times $A = (arr(a),arr(b),arr(c)) = (0,0,0)$ we obtain the delay tuple the same as topological analysis while under the last two arrival time conditions the resulting delay tuple is as accurate as one of the delay tuples in D_{exact}. We can confirm that in all the cases the computed delay tuples are conservative approximations to one of the exact delay tuples.

Floating Mode Analysis

We showed that viability analysis under an arbitrary arrival time condition gives a valid delay abstraction. The floating mode condition [CD93] is another well-accepted path sensitization condition known to be as accurate as viability for functional arrival time analysis of networks composed of simple gates. A natural question is whether the floating mode condition has this property or not. This is resolved negatively by showing that a delay abstraction computed by floating mode analysis may underestimate true delays.

path	Arrival time condition $(arr(a),arr(b),arr(c))$		
	(0,0,0)	(1,0,0)	(0,1,0)
(a,e,g)	false	false	false
(b,e,g)	false	false	false
(a,d,f,g)	true	false	true
(b,d,f,g)	true	true	false
(c,f,g)	false	false	false
delay tuple	$(3,3,-\infty)$	$(-\infty,3,-\infty)$	$(3,-\infty,-\infty)$

Table 4.2: Delay Characterization by Functional Arrival Time Analysis: Floating Mode

The circuit in Figure 4.1 is analyzed again using the floating mode condition for input vector 000. Table 4.2 shows floating mode analysis under three different arrival time conditions. In all the cases, the resulting delay abstractions are too optimistic. Recall that the delay tuples for this vector in D_{exact} are $(3,-\infty,2)$ and $(-\infty,3,2)$. Under the arrival time condition $(0,0,0)$ the computed delay tuple is $(3,3,-\infty)$. However,

$$(3,-\infty,2) \not\trianglelefteq (3,3,-\infty)$$
$$(-\infty,3,2) \not\trianglelefteq (3,3,-\infty).$$

Thus, $(3,3,-\infty)$ is a conservative approximation to neither of the exact delay tuples. The same is true for the other two delay tuples computed by floating mode analysis under different arrival time conditions.

The problem lies in the fact that if all the fanins of a gate have non-controlling values, floating mode analysis classifies the paths leading to any of the fanins but the latest as false. This is correct as far as delay estimation under the arrival time condition is concerned. However, once arrival time conditions change, a path with an early-arriving non-controlling value can potentially give the last non-controlling value thereby determining the delay up to the gate. Thus, simply assuming that the path remains false under a different arrival time condition may lead to delay underestimation. Viability does not suffer from this problem since those paths are classified as viable.

Note that the floating-mode condition does not produce delay underestimation for the case where a gate has a controlled value. Under this situation only the paths that give the first controlling value at a fanin of the gate is declared to be true. If we change arrival time conditions, a different fanin may give the earliest arriving controlling value. However, it is still conservative to assume that the fanin with the earliest controlling value under the original arrival time condition determines the stability of the gate output since it is equivalent to assuming that a controlling value, not necessarily

the first one, is responsible for the delay of the gate. This only results in delay overestimation. A similar argument applies to viability and the Brand-Iyengar conditions.

It has been known that floating mode analysis classifies fewer paths as false than viability analysis although both give the same delay estimation in functional arrival time analysis of networks composed of simple gates. This fact has been recognized as a favorable property to floating mode analysis since fewer paths are identified as critical paths. Timing optimization can then be performed more effectively on those fewer critical paths.

The discussion in this section demonstrates that the definition of false paths in floating mode analysis is so specialized to a given arrival time condition that the falsity and truth of paths are not conservative enough to be used under a different arrival time condition. On the other hand, viability is conservative in this sense by categorizing more paths as true. This creates a sharp contrast between the two sensitization conditions in favor of viability.

Relative Accuracy between Delay Abstractions

We have studied the correctness of delay characterization methods based on functional arrival time analysis, where a delay abstraction is computed by ignoring false paths under an arrival time condition. We have identified what sensitization conditions can be safely used in this approach. The relative accuracy of those correct sensitization conditions is discussed next.

For functional arrival time analysis the relative accuracy of various sensitization conditions is fully understood [MB91]. The accuracy can be argued by comparing output stable times estimated by different sensitization conditions under a given arrival time condition. Our interest here is different since we want to argue the relative accuracy of stable time estimates by considering all arrival time conditions.

Let us compare the Brand-Iyengar condition and viability. We showed that depending on the choice of fanin orderings delay tuples computed by Brand-Iyengar analysis can be as accurate as one of the tuples in D_{exact}, but can be the same as the delay tuple corresponding to topological analysis. Viability analysis has the same trend as in Table 4.1. The accuracy of delay tuples depends on the choice of arrival times at primary inputs. Therefore, unlike functional arrival time analysis, where viability has more accuracy than the Brand-Iyengar condition, there is no $\preceq$ relationship between $D^{\mathbf{l}}_{BI}$ and $D^{\mathbf{l}}_{viability}$ although both are guaranteed to give correct delay abstractions.

Static co-sensitization also gives a correct delay abstraction. In the analysis of the circuit in Figure 4.1 static co-sensitization gives a delay tuple that is the same as topological path analysis.

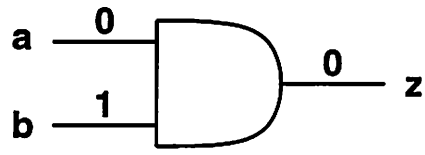

Figure 4.3: Static Co-sensitization vs. Viability/Floating Mode

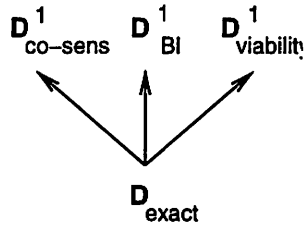

Figure 4.4: Relative Accuracy of Delay Abstractions Computed by Functional Arrival Time Analysis: Path Classification

As far as this example and the input vector 000 are concerned, it appears that $D^1_{BI} \preceq D^1_{co-sens}$ and $D^1_{viability} \preceq D^1_{co-sens}$ hold. However, Figure 4.3 gives a counterexample.

Consider the input vector $(a,b) = (0,1)$. Assume the unit delay model. Under static co-sensitization only path (a,z) is true. Path (b,z) is false since b does not have a controlling value of the AND gate. Therefore, the delay tuple based on static co-sensitization is $(1,-\infty)$. If we perform Brand-Iyengar analysis, both paths are true under the fanin ordering $a < b$, which results in the delay tuple $(1,1)$. Notice that $(1,-\infty) \trianglelefteq (1,1)$, indicating that static co-sensitization gives a more accurate delay tuple.

Viability can also give a delay tuple less accurate than that of static co-sensitization using the same example. Assume that a and b arrive at $t = 1$ and $t = 0$ respectively. The path from a is true since the side input b has a non-controlling value. The path from b is also true since the other fanin a, although being a controlling value, arrives later than b. Therefore we obtain the same delay tuple $(1,1)$ as in Brand-Iyengar analysis.

This example shows that static co-sensitization analysis can give a more accurate delay abstraction than the Brand-Iyengar condition or viability. Since we have already seen that the converse can happen in the circuit in Figure 4.1, it is concluded that there is no $\preceq$ relationship between static co-sensitization, Brand-Iyengar and viability.

Figure 4.4 summarizes the results so far. The diagram illustrates $\preceq$ relationship between different delay abstractions computed by the approach. An edge from D_1 to D_2 denotes $D_1 \preceq D_2$.

Since $D^1_{co-sens}$, D^1_{BI} and $D^1_{viability}$ are all correct delay abstractions, there are edges from D_{exact} to them. However the three delay abstractions computed by functional arrival time analysis are not comparable to each other since there is no $\preceq$ relationship.

4.6.2 Delay Abstractions based on Arrival Time Differences

In the previous subsection a delay abstraction of a combinational module was computed by ignoring false paths under a fixed arrival time condition. A set of path sensitization conditions that can be safely used under this context was identified.

This subsection takes a different approach to computing a delay abstraction. Although it is still based on functional arrival time analysis under a specific arrival time condition, we focus on signal stable times at the outputs of the module instead of classifying paths into false and true. The effective delay from an input to an output is defined as the time difference between the stable time of the output and the arrival time of the input. Recall that arrival times at the inputs are chosen arbitrarily again.

Let us illustrate the difference between this approach and the previous approach using the example in Figure 4.1. Suppose that viability analysis is performed under $(arr(a), arr(b), arr(c)) = (2,0,0)$. As in Table 4.1, the resulting delay tuple for input vector 000 was $(-\infty, 3, 2)$ in the first approach because both paths from a are false.

The new approach is based on stable times. The output stable time estimated by viability analysis is $t = 3$. Since a arrives at $t = 2$, the effective delay from a to the output is computed as $3 - 2 = 1$. For b, the delay is $3 - 0 = 3$. Finally, for c, the difference is $3 - 0 = 3$, which can be safely reduced to 2 since the longest topological path from c to the output is only 2. The resulting delay tuple is $(1, 3, 2)$, which is less accurate than $(-\infty, 3, 2)$ obtained in the first approach under the same arrival time condition.

If a difference of output stable time and input arrival time is not positive, the corresponding input is irrelevant to the stability of the output. Thus, the delay between the two terminals is set to $-\infty$.

The two approaches are related to each other. In fact we can prove that if the same path sensitization condition is used under the same arrival time condition, the delay abstraction constructed by the second approach is a conservative approximation to the one computed by the first approach.

Theorem 4.11 *Let $A = (a_1, \ldots, a_n)$ be arrival times at the inputs. Let $D^1(A)$ and $D^2(A)$ be the delay abstractions computed by the first approach and the second approach respectively, by functional*

arrival time analysis under A using the same path sensitization condition. Then, $D^1(A) \preceq D^2(A)$.

Proof Since $D^1(A)$ and $D^2(A)$ are computed by functional arrival time analysis, they have only one delay tuple for each input vector. Therefore, it is enough to show that for any input vector $\mathbf{x}$, $(d_{1,1},\ldots,d_{1,n}) \trianglelefteq (d_{2,1},\ldots,d_{2,n})$ where $(\mathbf{x},(d_{1,1},\ldots,d_{1,n})) \in D^1(A)$ and $(\mathbf{x},(d_{2,1},\ldots,d_{2,n})) \in D^2(A)$

Recall that $d_{1,i}$ is the longest true path length from input x_i to the output under $\mathbf{x}$ and A. Let s be the output stable time, which is computed as:

$$s = \max_i (a_i + d_{1,i}).$$

Suppose that the maximum is achieved when $i = j$. Then,

$$\begin{aligned} s &= a_j + d_{1,j} \\ \forall i, a_j + d_{1,j} &\geq a_i + d_{1,i}. \end{aligned}$$

$d_{2,i}$ is defined as $d_{2,i} = s - a_i$.

If $i = j$**:**

$$d_{2,j} = s - a_j = (a_j + d_{1,j}) - a_j = d_{1,j}.$$

If $i \neq j$**:**

$$d_{2,i} = s - a_i = (a_j + d_{1,j}) - a_i \geq d_{1,i}$$

Therefore,

$$\forall i, d_{1,i} \leq d_{2,i},$$

and thus $(d_{1,1},\ldots,d_{1,n}) \trianglelefteq (d_{2,1},\ldots,d_{2,n})$. Since this holds for all input vectors, $D^1(A) \preceq D^2(A)$.

Strictly speaking $d_{2,i}$ is refined from $s - a_i$. There are two special cases.

If $s - a_i$ is non-positive, $d_{2,i}$ is set to $-\infty$. This does not violate the inequality since no path is true from x_i, implying $d_{1,i} = -\infty$. Therefore, even after redefining $d_{2,i}$ to $-\infty$, $d_{1,i} \leq d_{2,i}$ still holds.

The second case is that if $s - a_i$ is greater than the longest topological delay l_i from x_i to the output, $d_{2,i}$ is set to l_i. Since $d_{1,i}$ is defined as the longest true path length from x_i to the output, it cannot be larger than l_i. Therefore, setting $d_{2,i}$ to l_i still gives $d_{1,i} \leq d_{2,i}$. □

This theorem immediately guarantees that static co-sensitization, Brand-Iyengar and viability can be safely used in the second approach without the risk of getting too optimistic delay abstractions. It also indicates that the delay abstraction computed in the second approach cannot be more accurate than the one in the first approach if the same sensitization condition and the same arrival time condition are used for underlying functional arrival time analysis.

Floating Mode Analysis

We showed that the use of floating mode analysis in the first approach can result in an incorrect delay abstraction. Interestingly, if this sensitization condition is used in conjunction with the second approach, a correct delay abstraction is computed. This is guaranteed by the following theorem.

Theorem 4.12 *Assume that a combinational module is composed of simple gates. Let* $A = (a_1, \ldots, a_n)$ *be arrival times at primary inputs* $x_1, \ldots, x_n$ *respectively. Let* $D^2_{viability}(A)$ *and* $D^2_{floating}(A)$ *be the delay abstractions computed by the second approach using viability and the floating mode condition respectively. Then,* $D^2_{viability}(A) = D^2_{floating}(A)$.

Proof Consider input vector **x**. Let $s_{viability}$ and $s_{floating}$ be the output stable times estimated by viability and floating mode analysis for **x** under A. It is known that

$$s_{viability} = s_{floating}$$

from the relative accuracy of the two sensitization conditions for functional arrival time analysis. Let $(d_{v,1}, \ldots, d_{v,n})$ and $(d_{f,1}, \ldots, d_{f,n})$ be the delay tuples of $D^2_{viability}(A)$ and $D^2_{floating}(A)$ under **x** respectively. Since $d_{v,i} = s_{viability} - a_i$ and $d_{f,i} = s_{floating} - a_i$, $\forall i, d_{v,i} = d_{f,i}$. Thus, $(d_{v,1}, \ldots, d_{v,n}) = (d_{f,1}, \ldots, d_{f,n})$ and $D^2_{viability}(A) = D^2_{floating}(A)$. □

An important observation in this theorem is that the relative accuracy of delay abstractions based on two sensitization conditions computed in the second approach is the same as that of the output stable times estimated by the two conditions. This can be generalized to the following theorem.

Theorem 4.13 *For simple-gate circuits:*

$$D_{exact} \preceq D^2_{viability}(A) = D^2_{floating}(A) \preceq \begin{matrix} D^2_{BI}(A) \\ D^2_{co-sens}(A) \end{matrix}$$

Proof Similar to the proof of Theorem 4.12. □

Finally, in this second approach, we can also use functional arrival time analysis based on the XBD0 model. Note that we could not use XBD0 analysis in the first approach since it does not classify true and false paths explicitly. However, it can be used in the second approach since only output stable times are required.

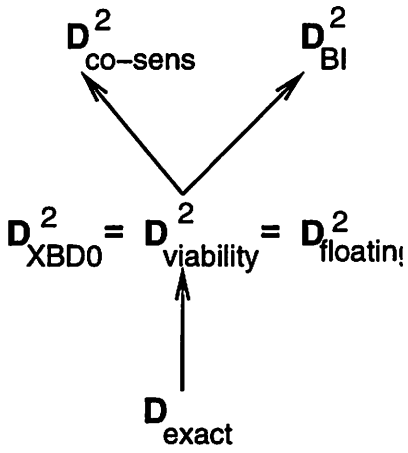

Figure 4.5: Relative Accuracy of Delay Abstractions Computed by Functional Arrival Time Analysis: Arrival Time Differences

Theorem 4.14

$$D_{exact} \preceq D^2_{XBD0}(A)$$

Proof Let s be the output stable time estimated by XBD0 analysis under A for $\mathbf{x}$. The delay tuple in $D^2_{XBD0}(A)$ for $\mathbf{x}$ is $(s-a_1,\ldots,s-a_n)$.

Now let us use D_{exact} to estimate the output stable time under A. Clearly,

$$s = delay\_propagate(D_{exact}, \mathbf{x}, A).$$

This means that for each $\mathbf{x}$, D_{exact} has a delay tuple $(d_1,\ldots,d_n)$ such that $\forall i, d_i \leq s-a_i$.

Therefore, $(d_1,\ldots,d_n) \trianglelefteq (s-a_1,\ldots,s-a_n)$ implying $D_{exact} \preceq D^2_{XBD0}(A)$. □

Finally, by adding $D^2_{XBD0}(A)$ we have the following result on the relative accuracy of delay abstractions computed in the second approach.

Theorem 4.15 *For simple-gate circuits:*

$$D_{exact} \preceq D^2_{XBD0}(A) = D^2_{viability}(A) = D^2_{floating}(A) \preceq \begin{matrix} D^2_{BI}(A) \\ D^2_{co-sens}(A) \end{matrix}$$

Proof The XBD0 analysis and viability analysis were shown to have the same accuracy for functional arrival time analysis [MSBSV93]. □

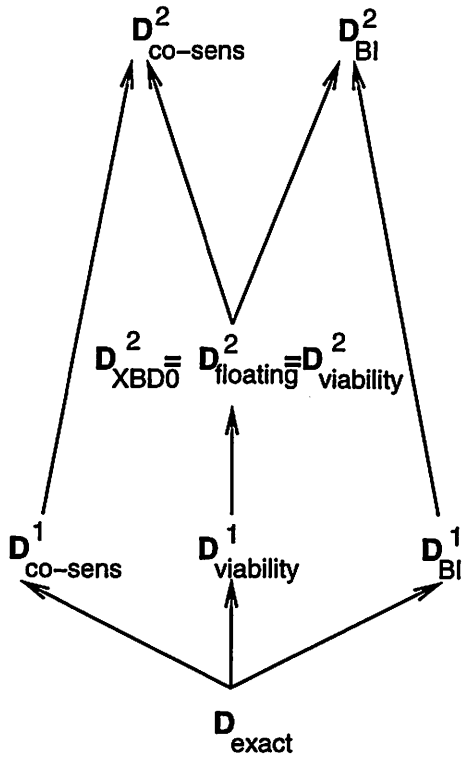

Figure 4.6: Relative Accuracy of Delay Abstractions

Figure 4.5 shows this relative accuracy pictorially. Note that this ordering holds if the same arrival time condition is used for all the sensitization conditions. If different arrival time conditions are used for different sensitization conditions, the ordering above is not necessarily true.

Figure 4.6 summarizes the relative accuracy of the two methods.

4.6.3 Refining Delay Abstractions by Multiple Functional Arrival Time Analyses

We presented two approaches to computing delay abstractions using functional arrival time analysis. A fundamental limitation in these approaches is that the resulting delay abstraction always has only one delay tuple for each input vector by its definition. The capability of having more than one delay tuple under the same input vector improves the accuracy of delay estimation by allowing adaptive choice of a delay tuple given an arrival time condition.

One way to refine a delay abstraction computed by functional arrival time analysis is to repeat the analysis under different arrival time conditions and enrich the delay abstraction by adding new delay tuples. For example let D_1 and D_2 be the delay abstractions of the same module. Let δ_1 and δ_2 be the delay tuples of D_1 and D_2 under $\mathbf{x}$ respectively. Let us improve the accuracy of D_1 by combining the information in D_2.

If $\delta_1 \trianglelefteq \delta_2$, δ_2 is subsumed by δ_1 in terms of accuracy. Therefore, no modification is made to D_1. If $\delta_2 \trianglelefteq \delta_1$, the situation is reversed. Therefore, we can replace δ_1 with δ_2 in D_1 under $\mathbf{x}$. Otherwise, $\delta_1 \not\trianglelefteq \delta_2$ and $\delta_2 \not\trianglelefteq \delta_1$. Since δ_1 and δ_2 are not comparable, each delay tuple can be used to give a better delay estimation than the other under some arrival time condition. Thus, we add δ_2 to D_1 under $\mathbf{x}$.

This operation can be defined as follows.

Definition 4.6 *Let D_1 and D_2 be delay abstractions of a combinational module. Let $D = D_1 \sqcup D_2$, where*

$$\begin{aligned} D \;=\; & \{(\mathbf{x}, \delta_1) \mid \nexists.(\mathbf{x}, \delta_2) \in D_2 \textit{ such that } \delta_2 \trianglelefteq \delta_1\} \\ & \cup \{(\mathbf{x}, \delta_2) \mid \nexists.(\mathbf{x}, \delta_1) \in D_1 \textit{ such that } \delta_1 \trianglelefteq \delta_2\} \end{aligned}$$

Theorem 4.16 *Let D_1 and D_2 be delay abstractions of a combinational module. Let $D = D_1 \sqcup D_2$. Then, $D \preceq D_1$ and $D \preceq D_2$.*

Proof We only prove $D \preceq D_1$. $D \preceq D_2$ can be proved by a symmetric argument.

Let $(\mathbf{x}, \delta_1) \in D_1$. If the same (input vector, delay tuple)-pair is in D, we are done. Otherwise, the pair was not included to D since there exists $(\mathbf{x}, \delta_2) \in D_2$ such that $\delta_2 \trianglelefteq \delta_1$. There exists no delay tuple δ_1' in D_1 under $\mathbf{x}$ such that $\delta_1' \trianglelefteq \delta_2$ since otherwise D_1 would include δ_1 and δ_1' under $\mathbf{x}$, where the first delay tuple is redundant. Thus, $(\mathbf{x}, \delta_2) \in D$. □

By trying more arrival time conditions and accumulating better delay tuples using $\sqcup$ operation, the delay abstraction approaches to the exact delay abstraction D_{exact}.

4.7 Delay Abstractions via Approximate Functional Required Time Analysis

In Section 4.2 we showed that the exact delay abstraction of a combinational module can be computed by performing functional required time analysis exactly. The use of approximate functional required time analysis in the same scenario gives an approximate delay abstraction in less computation time.

Consider the approximate functional required time analysis described in Section 3.3.2. The required time at the output is set to $t = 0$ as in the exact analysis. The approximate analysis computes input-vector independent required times $(t_1, \ldots, t_n)$ for $x_1, \ldots, x_n$ respectively. The required times computed are guaranteed to be a conservative approximation to the exact required times. More precisely, for each input vector $\mathbf{x}$, there exists an exact required time tuple $(t_1', \ldots, t_n')$ such that

$$(t_1, \ldots, t_n) \trianglelefteq (t_1', \ldots, t_n').$$

Intuitively, this means that the required times computed by approximate analysis are no later than those computed by exact analysis to be conservative.

An approximate delay abstraction D^1_{approx}, independent of input vectors, is then defined as $(-t_1, \ldots, -t_n)$, where the i-th element represents the delay from x_i to the output. From the inequality above, for any delay tuple $(-t_1, \ldots, -t_n) \in D^1_{approx}$ under $\mathbf{x}$, there exists $(-t_1', \ldots, -t_n') \in D_{exact}$ under $\mathbf{x}$ such that

$$(-t_1', \ldots, -t_n') \trianglelefteq (-t_1, \ldots, -t_n).$$

Therefore, $D_{exact} \preceq D^1_{approx}$.

4.8 Delay Abstractions via Approximate Functional Arrival Time Analysis

Approximate algorithms can also be employed in the delay characterization methods using functional arrival time analysis.

In the first approach based on path classification the falsity of a path was defined for each input vector separately. By approximating this input vector dependency conservatively, a path can be defined to be false if it is false under all input vectors. This is a more strict definition of false paths. The use of this new definition ignores a subset of the paths ignored in the exact approach. This gives no better delay tuple with respect to $\trianglelefteq$ than the exact approach. Hence, the approximation yields a delay abstraction more conservative than the original with respect to $\preceq$.

In the second approach, instead of computing output stable time for each input vector separately it can be conservatively defined as the earliest time when the output is stable under all input vectors. This results in overestimation of the output stable time under some input vectors. Since each input/output delay in a delay tuple is defined as the difference between the output stable time and input arrival time, this overestimation gives a larger delay for each input-output pair, resulting in a conservative delay abstraction.

4.9 Delay Characterization Independent of Gate Delay Assignments

So far we have studied various delay characterization techniques under the assumption that a delay assignment to all the gates is known. Because of this assumption, once a delay assignment changes, a delay abstraction needs to be recomputed. If a gate delay decreases, the monotone speedup property guarantees that the original delay abstraction is at least conservative. However, a delay increase of a gate invalidates the delay abstraction. In this section, we will show that the exact delay characterization technique discussed in Section 4.2 can be generalized to the case where gate delays are unknown. This generalized delay abstraction has a set of delay tuples under each input vector as before, but a delay number in a tuple is no longer a constant value determined by a given delay assignment, but a function of path lengths. Therefore, the same delay abstraction can be used as long as a circuit structure remains the same. One can obtain a delay abstraction specialized for a given delay assignment by simply evaluating all the delay functions.

Let us take the circuit in Figure 4.1 again as an example. We previously analyzed the circuit under the unit delay model. Now assume that gate delays are unknown. To distinguish the input-

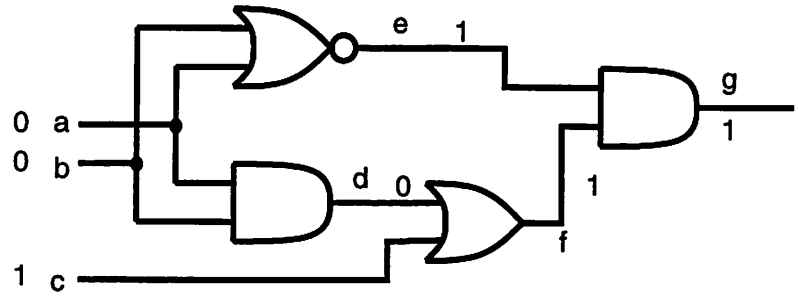

Figure 4.7: Signal Values of $\mathcal{M}$ under Input Vector 001

output paths of the circuit, we name the paths in the following way.

$$
\begin{aligned}
P_a &= a \rightarrow e \rightarrow g \\
P_b &= b \rightarrow e \rightarrow g \\
P'_a &= a \rightarrow d \rightarrow f \rightarrow g \\
P'_b &= b \rightarrow d \rightarrow f \rightarrow g \\
P_c &= c \rightarrow f \rightarrow g
\end{aligned}
$$

Consider an input vector $(a,b,c) = (0,0,1)$. Figure 4.7 shows final values of all the intermediate signals. Remember that the technique discussed in Section 4.2 was based on functional required time analysis. We will analyze the circuit backward from primary outputs to primary inputs as in functional required time analysis, but this time we will do so without using actual gate delays.

The primary output is settled to a constant 1 under this vector, which is the non-controlled value of the AND gate feeding the output. To guarantee this value at the output, all the inputs of the AND gate need to have the non-controlling value 1. Therefore both of the subpaths from the primary output up to the inputs e and f are responsible for the signal stability of the output.

Let us first examine the input e. e has a value 1 at the end. Since the gate feeding e is a NOR, it is the non-controlled value of the gate, and again both of the inputs need to present the non-controlling value 0. Thus, paths P_a and P_b are responsible for the signal stability of the primary output.

The other input of the AND gate f is settled to 1, which is fed by an OR gate. Since the value 1 is the controlled value of the gate, it is just enough to have the controlling value of the gate, a value 1, at one of its inputs. Since only c gives a value 1, this input value guarantees the output value 1 at the OR gate. Therefore, path P_c is responsible for delay while the other paths P'_a and P'_b

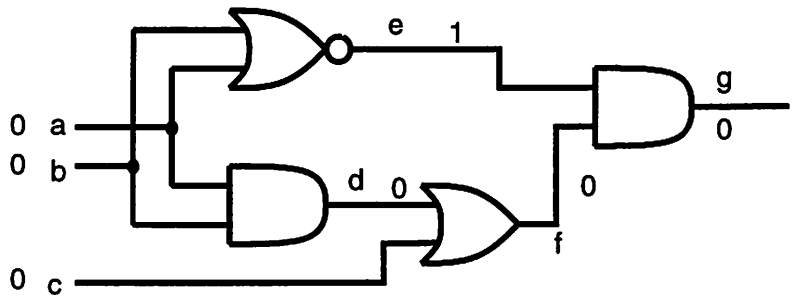

Figure 4.8: Signal Values of $\mathcal{M}$ under Input Vector 000

are irrelevant.

By combining the above, it is clear that only paths P_a, P_b and P_c are responsible for delay. The corresponding delay tuple is $(|P_a|, |P_b|, |P_c|)$[5], where $|P|$ denotes the path length of P. Note that under the unit delay model, $|P_a| = |P_b| = |P_c| = 2$, which gives $(2,2,2)$. The delay abstraction shown in Section 4.2 indeed has this delay tuple under input vector 001.

Although we only obtained one delay tuple in the above, in general a set of delay tuples is computed. For example, consider a different input vector $(a,b,c) = (0,0,0)$. Figure 4.8 shows signal values under this vector. This time the output value is the controlled value of the AND gate. Since only the bottom input f has the controlling value 0 of the gate, any path leading to the other input e is not responsible for delay. The value 0 at f is the non-controlled value of the OR gate, implying both of the inputs need further tracing. Since the bottom input is fed directly from input c, it is determined that P_c is responsible for delay. The other input d is settled to value 0, which is the controlled value of the AND gate. This time both of the inputs a and b have the controlling value of the AND gate. Thus, it is enough to have one of those to guarantee the value 0 at d. This translates to either P'_a or P'_b being responsible for delay. Since P_c is responsible independent of this choice, the delay tuples under this vector are $(|P'_a|, -\infty, |P_c|)$ and $(-\infty, |P'_b|, |P_c|)$.

A similar analysis for the other vectors gives a generalized delay abstraction of this circuit, which is shown in Table 4.3. The first, second and third columns show input vectors, sets of delay-responsible paths and sets of delay tuples respectively. Note that the delay abstraction in Section 4.2 is an instantiation of this under the unit delay model.

This delay-assignment-independent analysis has the following property. First, if a path is statically sensitizable under an input vector, then it is always categorized as a responsible path, and

[5]If a set of responsible paths has more than one path from the same primary input, the delay from the input is the maximum over all the paths from the input.

abc	responsible paths	$d_{a\to g}d_{b\to g}d_{c\to g}$
000	$\{\{P'_a,P'_c\},\{P'_b,P_c\}\}$	$\{(\lvert P'_a\rvert,-\infty,\lvert P_c\rvert),(-\infty,\lvert P'_b\rvert,\lvert P_c\rvert)\}$
001	$\{\{P_a,P_b,P_c\}\}$	$\{(\lvert P_a\rvert,\lvert P_b\rvert,\lvert P_c\rvert)\}$
010	$\{\{P'_a,P_c\},\{P_b\}\}$	$\{(\lvert P'_a\rvert,-\infty,\lvert P_c\rvert),(-\infty,\lvert P_b\rvert,-\infty)\}$
011	$\{\{P_b\}\}$	$\{(-\infty,\lvert P_b\rvert,-\infty)\}$
100	$\{\{P'_b,P_c\},\{P_a\}\}$	$\{(-\infty,\lvert P'_b\rvert,\lvert P_c\rvert),(\lvert P_a\rvert,-\infty,-\infty)\}$
101	$\{\{P_a\}\}$	$\{(\lvert P_a\rvert,-\infty,-\infty)\}$
110	$\{\{P_a\},\{P_b\}\}$	$\{(\lvert P_a\rvert.-\infty,-\infty),(-\infty,\lvert P_b\rvert,-\infty)\}$
111	$\{\{P_a\},\{P_b\}\}$	$\{(\lvert P_a\rvert,-\infty,-\infty),(-\infty,\lvert P_b\rvert,-\infty)\}$

Table 4.3: Generalized Delay Abstraction

its path length is referred to in all the delay tuples under the vector. Notice that under input vector 001, P_a, P_b and P_c are all statically sensitizable. Second, if a path is statically co-sensitizable, but not statically sensitizable under an input vector, then its path length is referred to in some of the delay tuples under the vector, but not all. Given such a path, there exists a gate on the path where 1) the output is a controlled value, and 2) the input on the path and at least one side input have a controlling value. Since only one of those controlling values is required to stabilize the output, it leads to a situation where multiple choices of responsible paths are available as in input vector 000. Thus, a path that is statically co-sensitizable, but not statically sensitizable under an input vector contributes to some of the delay tuples under the vector, but not all. Finally, if a path is not statically co-sensitizable under an input vector, it is never included in the set of responsible paths.

These results are consistent with the observation that static sensitization and co-sensitization are a sufficient condition and a necessary condition for a path to be responsible for delay respectively. This analysis also gives us additional information on precisely what subsets of static co-sensitizable paths have impact on delay.

4.10 Related Work

Das *et al.* [DJCM89] and Johannes *et al.* [JCM92, Joh93] discussed delay characterization of combinational modules in the context of hierarchical timing analysis. However, static sensitization was used to identify false paths. Since static sensitization can underestimate delays, this

approach is not conservative.

Yalcin and Hayes [YH95] introduced the notion of *conditional delay matrices* to represent input-vector-dependent delay characteristic of combinational modules. This is a restricted form of our delay abstractions in the sense that each input vector has a single delay tuple for each output in their conditional delay matrix. They proposed a technique to compute a delay matrix of a module using functional arrival time analysis under various path sensitization conditions, which is similar to the approach based on path classification discussed in Section 4.6.1. However they simply considered the case where all the primary inputs arrive simultaneously at $t = 0$. They did not argue the correctness of the approach under arrival-time dependent sensitization conditions such as viability and floating mode. In this chapter we have proved that viability is a safe condition in this context while floating mode can give an incorrect delay abstraction.

Belkhale and Suess [BS95] proposed a topological timing analysis technique under known false subgraphs, where topological analysis is performed by simply ignoring false subpaths specified by designers. Note that this is the same scenario as the first approach based on path classification discussed in Section 4.6.1. To validate this approach we need to argue how these false paths of a combinational module are computed. However, the paper did not discuss the issue. The result of Section 4.6.1 shows that viability, Brand-Iyengar or static co-sensitization can be used safely for this purpose.

Bhattacharya *et al.* [BDB96] presented a timing analysis technique based on path classification. They partitioned all the paths into a complete determining path set and a nondetermining path set, and showed that the delay of a circuit can be characterized as the topological longest delay of all the paths in the complete determining path set. The nondetermining path set may include a true path if there exists another path in the complete determining path set that gives the same or larger delay. However, they only gave an algorithm for constructing a complete determining path set from high-level descriptions.

4.11 Conclusions

We have studied various techniques for computing false-path-aware delay abstractions of combinational modules. The difficulty of this problem lies in the fact that state-of-the-art functional arrival time analysis is dependent on given arrival times at primary inputs. Although a direct application of the analysis leads to a correct delay abstraction for the arrival time condition under analysis, it is not clear whether the resulting delay abstraction is valid under other arrival times. Since we do not

know up front how a combinational module is used, it is impossible to choose a single representative arrival time condition.

There are several path sensitization conditions independent of arrival times. Although the use of such a condition resolves the difficulty, those sensitization conditions have a limited capability of false path detection. Thus, the accuracy of delay abstractions suffers.

A major contribution of this chapter is to show that the problem can be reduced to functional required time analysis studied in Chapter 3. Since the analysis is performed without any assumption about arrival times at the inputs, it is guaranteed to give a delay abstraction correct under any arrival time condition. We showed that in the most general form, a delay abstraction not only depends on input vectors but also needs to carry more than one delay tuple for each input vector, where a delay tuple is a list of effective delay values from each primary input. This capability of multiple delay tuples makes accurate delay estimation possible under any arrival time condition by choosing the best delay tuple adaptively.

We have then studied whether functional arrival time analysis can be used to compute conservative delay abstractions, where functional arrival time analysis is performed under an arrival time condition at the inputs chosen arbitrarily. Two approaches were proposed and the relative accuracy of various sensitization conditions in this framework was discussed. In the first approach false paths under the arrival time condition are simply ignored. A delay abstraction is constructed by computing the longest true path between each input/output pair under the arrival time condition. We proved that the use of viability in this scenario gives a delay abstraction valid not only for the arrival time condition under analysis but also for any other arrival time condition. On the other hand, the floating mode condition was shown to give an incorrect delay abstraction.

In the second approach the time difference between the stable time at an output and the arrival time at an input is used to define delay between the two terminals. The effect of false paths is implicitly taken into account by observing reduced output stable time. We showed that this approach computes a correct delay abstraction for any sensitization condition proven not to underestimate delays for functional arrival time analysis, hence in this case the floating mode condition is correct.

A practical aspect of this issue is worth studying thoroughly. In an actual design environment it may not be realistic to compute a fully input-vector dependent delay abstraction if a module has a large number of inputs. One way to alleviate this potential blowup in the size of delay abstractions is to take an approximate approach, discussed in Sections 4.7 and 4.8, which computes a delay abstraction independent of input vectors. This can be thought of as the other extreme. It would be interesting to explore delay abstractions partially dependent on input vectors. For example, if a sub-

set of inputs is identified as key variables for determining the timing characteristics of a module, it may be good enough to restrict vector dependency of a delay abstraction to the inputs in the set.

Yalcin *et al.* [Yal98] have recently proposed a technique with this flavor, where primary inputs are partitioned into control inputs and data inputs and vector-dependency is explored only for control inputs assuming that they have much more stronger influence on delay than data inputs. Computed delay abstractions are guaranteed to be conservative since the technique can be interpreted as a variation of Brand-Iyengar analysis, in which different fanin orderings are used for different input vectors. An idea on the size reduction of delay abstractions was also given.

The techniques presented in this chapter can be easily generalized to sequential circuits with edge-triggered flip-flops by analyzing the combinational portion of a sequential circuit. The input and the output of a flip-flop are assumed to be a primary output and a primary input respectively in the analysis. Sequential circuits containing level-sensitive latches cannot be handled in this approach because of cycle stealing [6]. This remains as future work.

[6] Venkatesh *et al.* [VPMS97] presented an algorithm for creating a delay abstraction of sequential circuits under topological delays. Although false paths are not taken into account, level-sensitive latches can be handled in the approach by assuming a maximum transparency.

Chapter 5

Hierarchical Functional Timing Analysis

Functional arrival time analysis has been discussed in the literature under the assumption that a circuit under analysis has a flat gate-level structure without any hierarchy. Therefore, even if a meaningful hierarchy exists in the circuit, it needs to be destroyed before analysis, which results in a potentially large circuit being passed to the analysis. Although existing flat timing analysis techniques are practical for circuits with up to thousands of gates, there is a limit in their capacity since the problem is NP-hard. Moreover, the analysis tends to become too slow for large circuits. Another serious drawback of flat analysis is that even a small change in a single module forces us to repeat an entire analysis from scratch. This prevents us from using functional arrival time analysis during logic synthesis. No incremental analysis method is known for functional arrival time analysis.

In this chapter we study *hierarchical functional arrival time analysis*, i.e., how to perform functional arrival time analysis of a hierarchical circuit by respecting the hierarchy. The analysis proceeds in a bottom up traversal of the hierarchy. The delay of a leaf-level module in a hierarchy is characterized as a delay abstraction using a technique discussed in Chapter 4. The delay abstraction is then used in a topological-like delay analysis at a higher level in the hierarchy. Since the delay abstraction captures enough timing information of the module, the delay analysis at the higher-level can be performed using only the delay abstraction without looking at the structural detail of the module. This hierarchical approach enables us to handle a hierarchical circuit without analyzing the entire circuit at one time at the gate-level. We discuss how false paths are detected in this analysis.

Hierarchical analysis naturally supports incremental analysis unlike flat analysis. Unless a module is modified, its delay abstraction computed for previous analysis remains valid and usable. This is a step forward to the use of functional timing analysis during logic optimization, which is not currently practiced.

This chapter is organized as follows. Hierarchical topological timing analysis is first reviewed in Section 5.1. Section 5.2 presents a hierarchical functional arrival time analysis algorithm where the timing characteristics of leaf modules are captured by false-path-aware delay abstractions independent of input vectors. The algorithm is illustrated using a simple example of cascaded carry-skip adders in Section 5.3. Section 5.4 proposes an improved algorithm for hierarchical timing analysis where the construction of false-path-aware delay abstractions is done in a demand-driven fashion. In Section 5.5 the general case where delay abstractions are input-vector dependent is discussed. Experimental results are shown in Section 5.6. Related work is discussed in Section 5.7. Section 5.8 concludes the chapter.

5.1 Hierarchical Topological Timing Analysis

In topological timing analysis, hierarchical approaches have been used extensively in practice to manage the complexity of industrial circuits [NST+82, TON83]. The reason why hierarchical analysis is prevalent is that the delay of a module under topological analysis is completely independent of a surrounding environment since all paths are assumed to propagate signals. This assumption makes hierarchical analysis trivial. However, false paths are completely ignored in this approach thereby making accurate analysis difficult.

5.2 Hierarchical Functional Arrival Time Analysis

We first consider the case where a given combinational circuit is composed of subcircuits, each of which has no hierarchy inside. In other words, the hierarchy depth of the circuit is 1. We call such a subcircuit without hierarchy a *leaf module*. No glue logic is in the top level for the sake of simplicity. We also assume that a topological order of subcircuits exists, i.e. there is no path from an output of a subcircuit to an input of the same subcircuit[1]. We will discuss an additional technique required to analyze a circuit with a multi-level hierarchy in Section 5.2.3.

Hierarchical arrival time analysis is performed in two steps. The first step constructs the delay abstraction of each leaf module by a direct application of the techniques described in Chapter 3. Arrival times at subcircuit boundaries are then determined in a topological order from primary inputs to primary outputs using the delay abstractions.

[1] Even if there does not exist a topological order among subcircuits, subcircuit outputs can always be ordered topologically unless combinational cycles exist. Therefore the general case can be handled in a similar way.

5.2.1 Delay Characterization of Leaf Modules

Given a hierarchical circuit, we first analyze each leaf module to characterize its timing property. If there is more than one instance of the same leaf module used in the circuit, it is analyzed once[2]. Notice that when we characterize the delay of a leaf module, no information is available on arrival times at the module inputs. Therefore, a leaf module is a combinational module and various technique discussed in Chapter 4 are directly applicable here.

Let $\mathcal{M}$ be a leaf module. Let $X = (x_1, \ldots, x_n)$ and $Z = (z_1, \ldots, z_m)$ be the primary inputs and the primary outputs of $\mathcal{M}$ respectively. Assume that an approximate delay abstraction independent of input vectors in Section 3.3.2 is used for the sake of simplicity. The delay abstraction $D_{z_j}^{approx} \subseteq R^n$ has a set of delay tuples $(d_1, \ldots, d_n)$, where d_i represents the effective delay from x_i to z_j. Remember that $D_{z_j}^{approx}$ may contain more than one delay tuple, in which case each of the delay tuples captures different timing characteristics of the module. This flexibility will be exploited fully later. To compute the delay abstraction of $\mathcal{M}$, one can apply the same analysis above for each output independently. Each output of the leaf module has the corresponding delay abstraction at the end of this first step.

5.2.2 Hierarchical Delay Computation

Assume that arrival times are given at the primary inputs of the top-level circuit under analysis. The goal of the second step is to compute the arrival time for each primary output of the top-level circuit.

Let $C_1 < C_2 < \ldots < C_p$ be a topological order of the subcircuits. Delay analysis is performed by visiting subcircuits and determining signal stable times at subcircuit outputs in this order. This guarantees that when subcircuit C_i is visited, arrival times at the subcircuit inputs are known. We then combine these arrival times and the delay abstraction of the corresponding leaf module to compute arrival times at the subcircuit outputs.

The core of this computation is in how arrival times at subcircuit inputs are propagated through a subcircuit. This has been already discussed in Section 4.4 for the case where a delay abstraction is input-vector dependent. Assume that C_i is under analysis. Let $X = (x_1, \ldots, x_n)$ and $Z = (z_1, \ldots, z_m)$ be the inputs and the outputs of C_i respectively. Let $A = (a_1, \ldots, a_n)$ be the arrival

[2]If a load-dependent delay model is used, delay characterization must be done for each load at an output of the module. Even under a load-independent delay model, delay characterization can be done for each instance so that satisfiability don't care (SDC) and observability don't care (ODC) [BHSV90] at the inputs of the instance are taken care of. This yields a more accurate customized delay abstraction.

times at X. Let D be the delay abstraction for the output z_k. The arrival time at z_k is computed as:

$$\min_{\delta_i=(d_{i,1},\ldots,d_{i,n})\in D} \max_j (a_j + d_{i,j}).$$

The max operation corresponds to standard topological analysis under a delay tuple δ_i in D. The min operation then examines all the delay tuples in D and chooses the *earliest* stable time at the output. The important differences between this analysis and topological analysis are that our false-path-aware delay abstractions are more accurate than topological delay abstractions and can maintain multiple delay tuples for an output to preserve accuracy. This min-max computation is linear in $n|D|$, where n and $|D|$ denote the number of inputs of the subcircuit and the number of delay tuples in D respectively. $|D|$ is typically a small constant if D is a delay abstraction independent of input vectors.

Theorem 5.1 *The above analysis gives a conservative approximation to the true delay of the entire circuit under the XBD0 model.*

Proof It is enough to show by induction on the topological order of subcircuits that the arrival time of any subcircuit input/output estimated by this analysis is no earlier than its true arrival time. This is trivially satisfied at the primary inputs of the entire circuit. Assume that the inputs of subcircuit C_i meet the above condition. If the delay abstraction D_i of C_i meets $D_i^{exact} \preceq D_i$, where D_i^{exact} is the exact delay abstraction of C_i, arrival time estimates of subcircuit outputs are never underestimated even if exact arrival times are used at subcircuit inputs. Since actual arrival times used at the subcircuit inputs are the same as or later than the corresponding exact arrival times, the arrival times at subcircuit outputs are never underestimated. □

5.2.3 Delay Characterization of Circuits Composed of Subcircuits

We have considered the case where a given circuit has a single level of hierarchy. Even if a circuit has a hierarchy depth more than one, the overall strategy of hierarchical functional arrival time analysis is still the same;

1. the delay abstractions of subhierarchies are constructed in a bottom-up way from the lowest-level leaf modules to the subcircuits directly under the top level, and
2. arrival times at the primary inputs are propagated using the delay abstractions of the top-level subcircuits.

An additional task involved in this general case is to characterize the delay abstraction of a subhierarchy composed of subcircuits whose delay abstractions have been already computed.

To simplify the argument we focus on the case where a subhierarchy under analysis has a single output[3]. Assume that the subhierarchy is described as connections of subcircuits $C_1, \ldots, C_p$ without any glue logic. Let $C_1 < C_2 < \ldots < C_p$ be a topological order of the subcircuits. As in the delay characterization step in Section 5.2.1, we need to determine the delay abstraction of the subhierarchy valid under any surrounding environment. The analysis starts by setting a required time, $t = 0$, to the output of the subhierarchy. This required time is then propagated backwards using the delay abstractions of the subcircuits. If each output of those subcircuits has a delay abstraction consisting of a single delay tuple, it is just enough to perform standard topological required time analysis on $C_1, \ldots, C_p$ using the delay abstractions. The absolute value of the required time at each subhierarchy input gives the effective delay from the input to the output. The delay abstraction of the subhierarchy has a single delay tuple.

In general the delay abstraction of each circuit can have more than one delay tuple. The simplest approach is to try topological required time analysis for every combination of delay tuples and extract the most relaxed required time from the result. Since the number of delay tuples in delay abstractions independent of input vectors is typically a small number[4], this explicit enumeration is still a reasonable strategy in many cases. It is, however, possible to explore all possibilities by symbolically propagating required times through the circuit.

Assume that the circuit in Figure 5.1 is a subhierarchy under analysis. This circuit consists of two subcircuits C_1 and C_2. Suppose that each subcircuit has a single output and that the delay abstractions D_1 and D_2 of C_1 and C_2 respectively are as follows.

$$\begin{aligned} D_1 &= \{(2,2),(3,1)\} \\ D_2 &= \{(1,4),(2,3)\} \end{aligned}$$

where in each delay tuple the first value and the second value are the delays from the left and the right inputs of the corresponding subcircuit respectively. There are 4 possible cases to consider. Figure 5.2 shows the results of topological required time analysis for each case. A pair of integers attached to a subcircuit is a delay tuple used while an integer on an edge denotes the required time of the connection. The valid required times at the subhierarchy inputs are $(req(i_1), req(i_2)) =$

[3] As in Section 5.2.1, if a subhierarchy has multiple outputs, it is enough to repeat the same analysis for each output separately.

[4] We experimentally confirmed this on benchmark circuits.

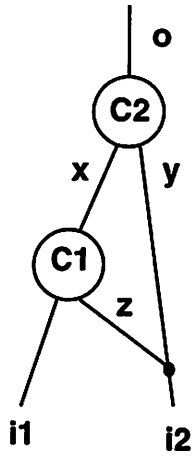

Figure 5.1: Subhierarchy in a Hierarchical Circuit

$(-3,-4),(-4,-4),(-5,-3)$. Therefore the delay abstraction of this subhierarchy is characterized as a set of delay tuples $\{(3,4),(5,3)\}$. Note that $(4,4)$ has been dropped since $(3,4)$ subsumes $(4,4)$.

We will now show how this analysis can be done symbolically without any explicit case analysis. The basic idea is to propagate required times along with Boolean conditions so that the choices of delay tuples are encoded. For example, there are two choices of delay tuples for subcircuit C_2. To encode these two possibilities a Boolean variable α is introduced. Let $\alpha = 1$ and $\alpha = 0$ denote the cases where the first and the second tuples are chosen respectively. The required time at the subhierarchy output is propagated backwards through C_2 along with a Boolean condition on α. For example, the required time at x, $req(x)$, is $\{(-1,\alpha),(-2,\bar{\alpha})\}$, in which the first element of each tuple is a required time while the second element is a Boolean condition associated with the required time. The required time at y is computed similarly as follows.

$$req(y) = \{(-4,\alpha),(-3,\bar{\alpha})\}$$

Another Boolean parameter β is introduced for the choice of delay tuples for C_1. Let $\beta = 1$ denote the condition that the first tuple is used and $\beta = 0$ the condition that the second is used. The conditional required times at i_1 and z are:

$$\begin{aligned} req(i_1) &= \{(-3,\alpha\beta),(-4,\alpha\oplus\beta),(-5,\overline{\alpha\beta})\} \\ req(z) &= \{(-2,\alpha\bar{\beta}),(-3,\alpha\overline{\oplus}\beta),(-4,\overline{\alpha\beta})\}. \end{aligned}$$

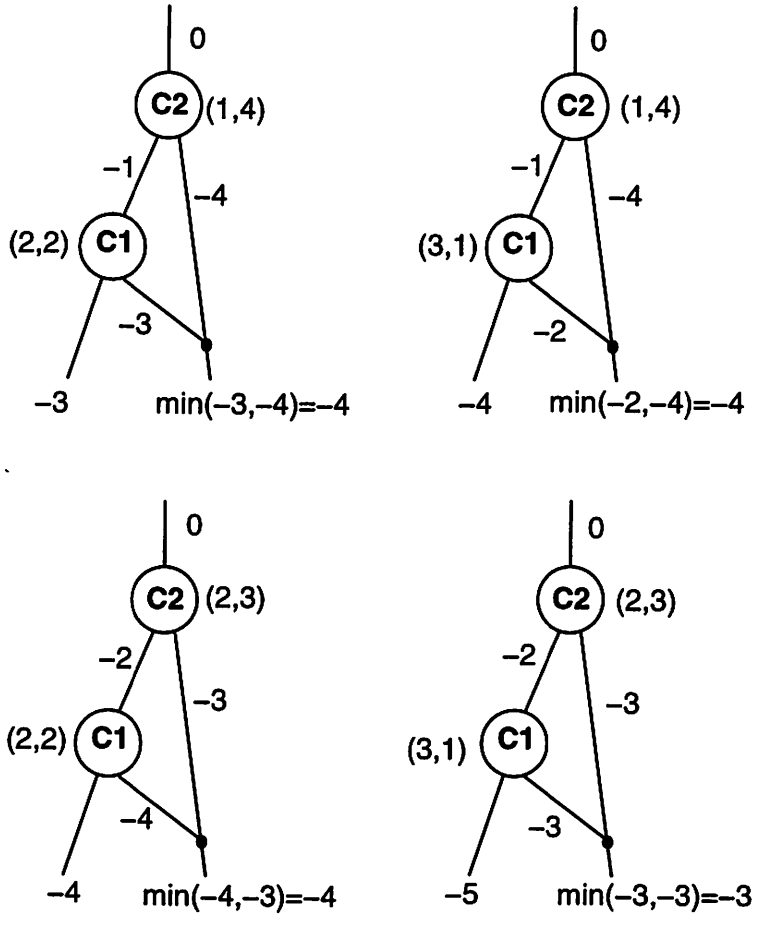

Figure 5.2: Required Time Analysis of a Subhierarchy

Finally to determine the required time at i_2 the minimum of $req(y)$ and $req(z)$ is taken symbolically, which gives

$$req(i_2) = \{(-3, \overline{\alpha}\overline{\beta}), (-4, \alpha+\beta)\}.$$

All of these symbolic computation can be performed using ADDs [BFG+93] or MTBDDs [CMZ+93].

By combining $req(i_1)$ and $req(i_2)$ symbolically, the following required times at the subhierarchy inputs are determined.

$$(req(i_1), req(i_2)) = \{((-3,-4), \alpha\beta), ((-4,-4), \alpha \oplus \beta), ((-5,-3), \overline{\alpha}\overline{\beta})\}$$

in which the first element of each tuple is a pair of required times at i_1 and i_2 and the second element is the Boolean constraint associated with it. Enumeration of all distinct required time pairs followed by removal of dominated required time pairs is the final step necessary to complete the delay characterization of the subhierarchy, which gives $D = \{(3,4), (5,3)\}$. Note that the sign of all the required times is flipped at the end since they are subtracted from $t = 0$.

This step can be thought of as hierarchical functional required time analysis.

5.2.4 Incremental Timing Analysis

Incremental timing analysis can be easily incorporated into our formulation. Once the delay abstraction of a leaf module is obtained, it remains valid no matter what changes are made in other modules. Therefore a modification of a single module only leads to

1. delay characterization of the modified module and
2. top-level analysis.

Even without any change in a given circuit it is likely that one is interested in performing arrival time analysis of the same hierarchical circuit under different arrival time conditions. If flat analysis is employed, each arrival time condition requires a separate analysis while in hierarchical analysis the delay characterization stage can be shared by all the analyses. Thus once the delay abstractions of all the modules in a given circuit are computed, it is enough to perform top-level analysis for each case.

5.3 Example

To illustrate the technique proposed in Section 5.2 we take a simple example.

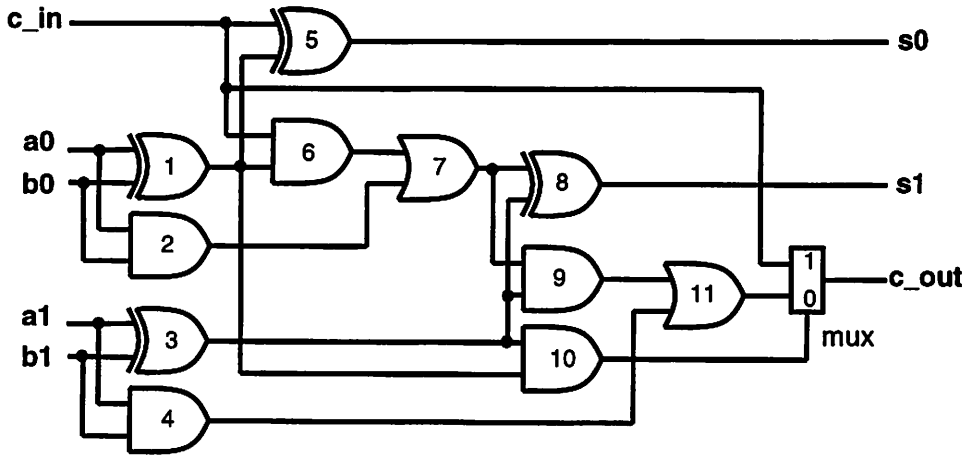

Figure 5.3: 2-bit Carry-Skip Adder

Consider a 2-bit carry-skip adder taken from [KMS91] shown in Figure 5.3. Cascading this adder n times yields a carry-skip adder of $2n$ bits. We show how the performance of this cascade adder can be estimated accurately by hierarchical analysis. Figure 5.4 shows a 4-bit adder composed of two 2-bit carry-skip adders, where c_{out} of the first adder is fed into c_{in} of the second adder.

Assume that a gate delay of 1 for the AND gate and the OR gate, and gate delays of 2 for the XOR gate and the MUX gate.

The input-vector independent delay abstractions D_{s_0}, D_{s_1} and $D_{c_{out}}$ are:

$$\begin{aligned} D_{s_0} &= \{(2,4,4,-\infty,-\infty)\} && \text{(topological delay)} \\ D_{s_1} &= \{(4,6,6,4,4)\} && \text{(topological delay)} \\ D_{c_{out}} &= \{(2,8,8,6,6)\} \end{aligned}$$

where primary inputs are ordered as $c_{in} < a_0 < b_0 < a_1 < b_1$. In this particular circuit each delay abstraction has a single delay tuple. The delay abstractions for s_0 and s_1 are exactly the same as those under topological analysis. $D_{c_{out}}$ is more accurate than its topological delay abstraction since the effective delay from c_{in} to c_{out} is 2[5] in $D_{c_{out}}$ while the longest topological path is of length 6[6].

Let us proceed to the second step of the analysis. Assume that all the primary inputs of the cascade circuit in Figure 5.4 arrive at $t = 0$. We focus on computing the arrival time of c_4 since it is the most interesting output in terms of analysis. The arrival times for all the other primary outputs are the same as their topological arrival times. The arrival time at the intermediate signal *tmp* is first

[5] (c_{in}, mux, c_{out})

[6] $(c_{in}, g_6, g_7, g_9, g_{11}, mux, c_{out})$

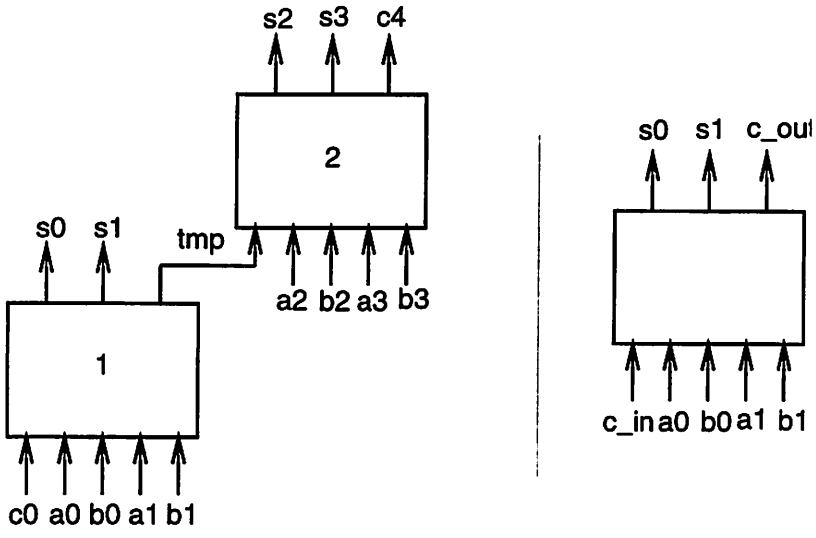

Figure 5.4: 4-bit Carry-Skip Adder Composed of Two 2-bit Adders

computed. Since all the inputs of the first adder arrive simultaneously at $t = 0$, the arrival time at *tmp* is determined as $t = 8$, where a_0 and b_0 are critical.

The arrival time of the carry output c_4 can now be computed by analyzing the second adder. The arrival time at *tmp* is $t = 8$ while the other inputs arrive at $t = 0$. Combining the arrival times and the delay abstraction $D_{c_{out}}$, we see that a path from *tmp* is critical. This gives the arrival time at c_4 $t = 8+2 = 10$, which matches the result of flat analysis.

It is more intuitive to see the analysis pictorially. Figure 5.5 illustrates $D_{c_{out}}$ as a polygon. The polygon describes the fact that to have the signal c_{out} stabilized, inputs c_{in}, a_0, b_0, a_1 and b_1 must arrive 2, 8, 8, 6 and 6 time units before respectively[7]. To determine the arrival time of *tmp*, the polygon is pushed down from the top as much as possible, as in Figure 5.6, so as not to intersect a given arrival time constraint. In this example, all the primary inputs arrive simultaneously at $t = 0$. Therefore the bottom edges for a_0 and b_0 first touch the constraint, which gives the arrival time of $t = 8$ at *tmp*. Shadowed regions represent arrival time constraints. Next, another polygon of the same shape for the second adder is stacked on the first polygon. This time the bottom edge of c_0 first hits the arrival time constraint of *tmp*. (Note that c_{in} in the module is connected to *tmp* and all the other inputs of the modules are connected to primary inputs, whose arrival times are $t = 0$.) Therefore, the arrival time of c_4 is 10.

[7]If there is more than one delay tuple in a delay abstraction, we have multiple polygons. Whenever arrival times are propagated through a subcircuit, all the polygons are tried and the best one that gives the earliest arrival time is chosen.

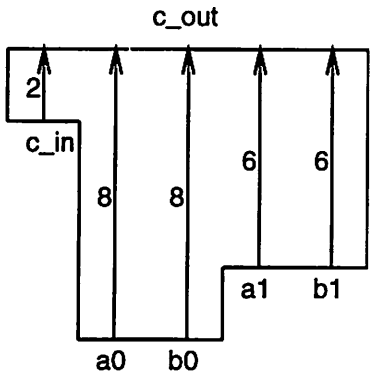

Figure 5.5: Delay Abstraction of the 2-bit Adder

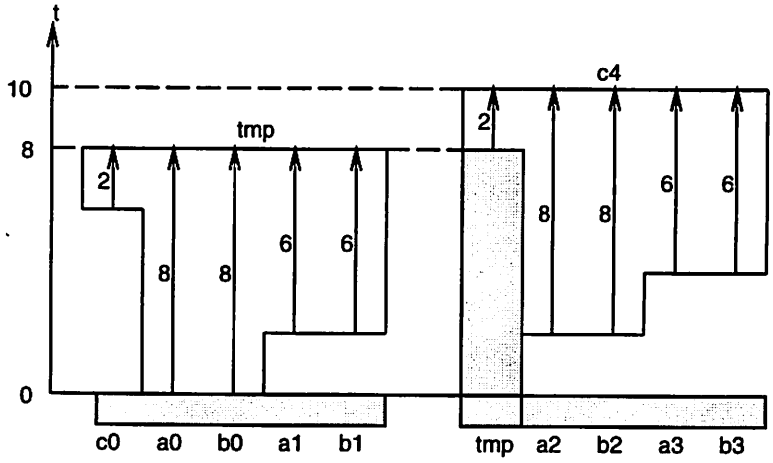

Figure 5.6: Hierarchical Analysis of the 4-bit Carry-Skip Adder Composed of Two 2-bit Adders

It is easy to see from the analysis above that the delay of the last carry output of the circuit composed of n adders is $t = 8+2(n-1) = 2n+6$ if all the primary inputs arrive at $t = 0$. We have verified that this delay estimation matches the results of the flat analysis at least up to $n = 8$. This ability of parameterized analysis is missing in flat analysis.

Keutzer *et al.* [KMS91] analyzed the circuit in Figure 5.3 under $arr(c_{in}) = 5, arr(a_0) = arr(b_0) = arr(a_1) = arr(b_1) = 0$. It is easy to see that a_0 and b_0 are critical in this case from Figure 5.7. The delay of c_{out} is $t = 0+8 = 8$, which is again the same as the result of flat analysis under the arrival times. From Figure 5.7 we can even claim that delaying c_{in} by one time unit does not change the signal arrival time at c_{out}, i.e. the slack of c_{in} is 1. This can be thought of as an "exact"

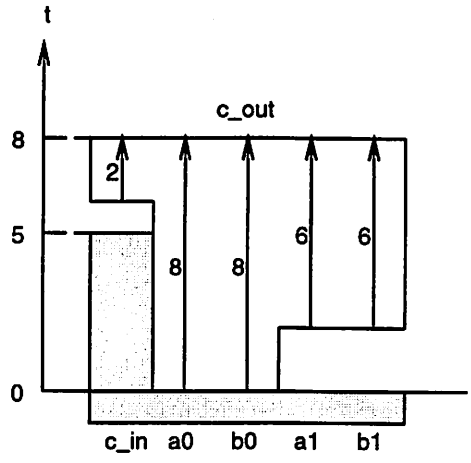

Figure 5.7: Delay Analysis of the 2-bit Adder under $arr(c_{in}) = 5, arr(\text{others}) = 0$

slack available at c_{in}[8]. Notice that if the slack of this input is computed topologically, it is -3 indicating that the signal needs to be sped up 3 time units to meet the required time $t = 8$ at c_{out}, which is completely opposite to the above. This is because false paths are completely ignored in the latter case.

5.4 Improved Algorithm for Hierarchical Functional Arrival Time Analysis

We have seen a two-step hierarchical timing analysis algorithm where delay characterization of leaf modules is followed by arrival time computation. Although this separation makes the understanding of the entire flow easy, timing analysis can be performed more efficiently by interleaving the two steps in a smart way.

The main computational disadvantage of the two-step approach is that the effective delay of each input-output pair of every leaf module is computed first even if the input-output paths are never critical in any instance of the module. Therefore part of the additional accuracy achieved in resulting delay abstractions may not contribute to the accuracy of the delay estimate of the entire circuit. In this sense some of the CPU time spent in the computation of delay abstractions is wasted. Since delay characterization of leaf modules is not always a cheap operation, this loss in CPU time is not negligible in general.

[8]Chen *et al.* [CCHD93] studied false-path-aware slack computation under a fixed arrival time condition.

One way to alleviate this is to start from crude delay abstractions based on topological delay analysis and to refine them gradually as we estimate the delay of a given hierarchical circuit. This way delay abstractions are refined only if additional accuracy is required to get a better delay estimate of the circuit under analysis. Hence delay abstractions are never more accurate than necessary.

Assume that a circuit of hierarchical depth 1 is given, i.e. the top-level circuit consists of several leaf modules. Consider a directed acyclic graph $G = (V, E)$, called *timing graph*, constructed in the following way. Each input or output of the leaf modules forms a vertex in G. If an output of a module is connected to an input of another module, these two nodes share the same vertex. A directed edge is added from a vertex to another vertex if there is a topological path between the corresponding nodes within a single leaf module. Each edge corresponds to an input-output pair of a single leaf module. The edge is initially labeled the longest topological path delay between the nodes. The timing graph is simply an abstract representation of the hierarchical circuit in terms of delays.

We start by assigning to each vertex corresponding to a primary input of the top-level circuit its arrival time. The arrival times are then propagated forward topologically based on the current edge weights. Once the latest stabilizing primary output is determined, its arrival time is asserted at each primary output vertex as a required time. These required times are then propagated backward topologically through the timing graph again using the edge weights. Once an arrival time and a required time are computed at each vertex, the slack at the vertex is defined as the difference between the required time and the arrival time. Any vertex whose slack is zero is on a critical path.

Once the slack computation is done on the timing graph, critical edges of the timing graph are identified. A critical edge is an edge in the timing graph which connects two vertices whose slacks are zero. To get a better delay estimate of the entire circuit, one needs to have a better delay estimate of such critical edges. Therefore each critical edge is examined one by one to see if the corresponding input-output delay can be improved further by considering false paths. More specifically, the transitive fanin cone of the corresponding leaf module from the output is examined to see if the effective delay from the input to the output is smaller than the topological delay between the two nodes. This can be checked easily by performing functional arrival time analysis of the cone as follows.

Let $x_1, \ldots, x_n$ be the inputs of the cone and z be the output of the cone. Let l_i be the longest topological path length from x_i to z. Assume that x_k is the critical input and that the second longest topological path length from x_k to z is $l'_k < l_k$. Now consider the case where input $x_i (i \neq k)$ arrives at $t = -l_i$ and critical input x_k arrives at $t = -l'_k$. If z still gets stabilized by $t = 0$ under these arrival times, the effective input-output delay between x_k and z is no greater than l'_k. We can then update the

weight of the edge from l_k to l'_k and repeat timing analysis of the timing graph to see which edges are critical under this refined delay abstraction[9]. Otherwise the current delay estimation of l_k is accurate and cannot be improved since x_k is the only primary input that arrives later than its topological required time, and thus is responsible for the instability of z at $t = 0$. Therefore the edge is marked to indicate that no further improvement is possible. Notice that the stabilization check at z can be realized by performing functional arrival time analysis of the cone under the arrival time conditions above.

As we refine weights of critical edges this way, a more accurate delay estimate of the entire circuit is obtained[10]. This iterative process stops once all critical edges are marked.

5.5 Hierarchical Delay Computation using Input-Vector Dependent Delay Abstractions

We have discussed hierarchical functional arrival time analysis, where the delay of a leaf module is conservatively characterized with a delay abstraction independent of input vectors. We showed that by removing input-vector dependency from delay abstractions at the lowest level of the hierarchy, delay analysis above the level is reduced into a variation of topological analysis and thus becomes computationally efficient. Since an effective algorithm exists for computing an input-vector independent delay abstraction of a combinational module, this approach seems to be a practical way to perform hierarchical analysis.

However, the restriction that a delay abstraction needs to be input-vector independent can make analysis too pessimistic thereby resulting in delay overestimation. The only false paths detected in this approach are those paths whose subpaths, completely contained in a single leaf module, are false regardless of input vectors to the module. We call this type of false paths *locally false paths* since the falsity of the paths can be determined locally by analyzing the leaf module. They are false no matter how the the module interacts with other modules.

For example, in the carry-skip adder example in Figure 5.3, the longest topological path from the carry input to the carry output was locally false and this information was used to compute a delay abstraction strictly more accurate than the delay abstraction by topological analysis. It turned out that all long false paths in the circuit are locally false. Therefore we were able to achieve the

[9] If there is more than one instance of the same module in the given circuit, edge weights are updated in all instances.

[10] Once an edge is assigned a new weight, it never gets increased again in this algorithm. Therefore, a delay abstraction computed for a module is dependent on the order of edge weight updates.

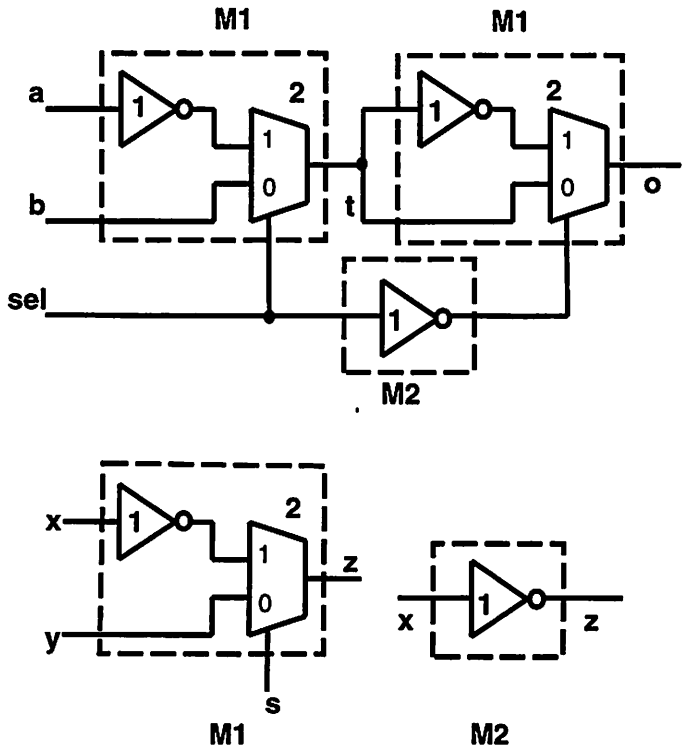

Figure 5.8: A Limitation of Input-Vector Independent Delay Abstractions

same accuracy of flat analysis in the hierarchical analysis even with approximate delay abstractions.

However, not all false paths are locally false. There is a path that is false not simply due to the falsity of a subpath contained in a single leaf module but due to the interaction of several modules. We call such false paths *globally false paths*. To make detection of these false paths possible, delay abstractions need to be more accurate by carrying input-vector dependent delay information. Delay analysis is then performed by taking this dependency into account.

5.5.1 Example

We will illustrate how a path becomes globally false using a hierarchical circuit in Figure 5.8. The circuit has two leaf modules. The leaf module M_1 is composed of a multiplexer and an inverter and has two instances. The leaf module M_2 only consists of an inverter. Assume that the inverters have delay of 1 while the multiplexers have delays of 2. Suppose that all the primary inputs arrive at $t = 0$. Our goal is to estimate the arrival time at the output o.

Flat analysis of this circuit estimates the arrival time as $t = 5$. This means that the longest

topological path of length 6 from a to the output via the two upper subpaths is false.

Let us see if this false path can be detected using the hierarchical analysis in Section 5.2. We first need to compute input-vector independent delay abstractions of M_1 and M_2. M_2 is composed of an inverter. Thus, the exact delay abstraction $D^{M_2}_{exact}$ of M_2 is:

x	$d_{x\to z}$
0	$\{(1)\}$
1	$\{(1)\}$.

Since this is already input-vector independent, we can directly use it as the delay abstraction of M_2, i.e. $D^{M_2}_{approx} = \{(1)\}$.

The exact delay abstraction $D^{M_1}_{exact}$ of M_1 is:

(s,x,y)	$d_{s\to z}d_{x\to z}d_{y\to z}$
000	$\{(2,-\infty,2)\}$
001	$\{(2,-\infty,2)\}$
010	$\{(2,-\infty,2),(-\infty,3,2)\}$
011	$\{(2,-\infty,2)\}$
100	$\{(2,3,-\infty)\}$
101	$\{(2,3,-\infty)\}$
110	$\{(2,3,-\infty),(-\infty,3,2)\}$
111	$\{(2,3,-\infty)\}$.

Let us first compute an input-vector independent delay abstraction $D^{M_1}_{approx}$ that is a conservative approximation to the exact one, i.e. $D^{M_1}_{exact} \preceq D^{M_1}_{approx}$. Let $D^{M_1}_{approx} = \{(d_s,d_x,d_y)\}$. In order to be conservative, for any input vector $\mathbf{x}$ $D^{M_1}_{exact}$ must have a delay tuple (d'_s,d'_x,d'_y) such that $(d'_s,d'_x,d'_y) \trianglelefteq (d_s,d_x,d_y)$. Consider input vector 000. The requirement above gives the inequality: $(2,-\infty,2) \trianglelefteq (d_s,d_x,d_y)$, implying $d_s \geq 2$ and $d_y \geq 2$. Similarly, input vector 100 gives $(2,3,-\infty) \trianglelefteq (d_s,d_x,d_y)$, by which d_s and d_x are constrained to be $d_s \geq 2$ and $d_x \geq 3$. The analysis of these two vectors already gives $D^{M_1}_{approx} = \{(2,3,2)\}$, which is the same as the topological delay abstraction. This delay tuple is a conservative approximation for all the other vectors.

The fact that the input-vector independent delay abstractions of both leaf modules are the same as the corresponding topological delay abstractions implies that the analysis is only as accurate as topological analysis. Therefore, the arrival time at the output is overestimated as $t = 6$, thereby missing the false path from a of length 6. Although the exact delay abstraction of M_1 shows that

some paths are false conditionally, this information is lost when the delay abstraction of M_1 is approximated.

This false path can be detected once we use the exact delay abstractions of the modules. Let us see how it is detected in detail. The path is decomposed into two subpaths: a path from a to the output of the left multiplexer t and a path from t to the output o. These subpaths correspond to the same input-output path from x to z in M_1. This input-output path of length 3 in M_1 is true only if $s = 1$[11]. Therefore, the subpath from a to t is true only if $sel = 1$ while the subpath from t to the output is true only if $sel = 0$. Since these two conditions are conflicting, it is impossible to sensitize both to make the entire path from a to o true. Since the path is false due to the interaction of multiple leaf modules, it is a globally false path.

5.5.2 Generalized XBD0 Analysis for Input-Vector Dependent Delay Abstractions

Hierarchical analysis using input-vector independent delay abstractions has a limitation that globally false paths are overlooked thereby potentially overestimating delays. This subsection shows that delay analysis using input-vector dependent delay abstractions can be formulated as a generalization of XBD0 analysis for flat circuits.

Notice that the delay analysis at the top level of a hierarchy has exactly this situation. Each subhierarchy at that level can be thought of as a gate with complex functionality. The delay abstraction of the subhierarchy has been constructed in a bottom-up fashion and is potentially dependent on input vectors applied to the subhierarchy inputs.

In regular XBD0 analysis a circuit under analysis is a flat network composed of gates with known delay characteristics. In the simplest case the delay of each gate is given as a single delay value independent of input pins and signal directions (rise/fall delays) as in Section 2.4.6[12]. In XBD0 analysis this delay value is conditionally used to perform signal stability computation. However, the actual delay abstraction of a gate underlying in XBD0 analysis is not explicitly defined in [MSBSV93]. It is embedded in the recursive formulation of χ functions.

Consider a two-input AND gate n for example. We are interested in the delay abstraction of the gate used implicitly in XBD0 analysis. Let m_1 and m_2 be the fanins of the gate. Suppose that

[11] In fact the path can be true under input vector 010. However, it is true only if input s is late. In this example s always arrives earlier than x and y in both instances. Thus, we can safely ignore this tuple for ease of exposition.

[12] Although we did not discuss this aspect in Section 2.4.6, it is easy to generalize the analysis so that gate delays vary depending on input pins and signal directions.

the delay of this gate is d for simplicity. The recursive definition of χ functions for the AND gate is:

$$\begin{aligned}\chi^{\tau}_{n,1} &= \chi^{\tau-d}_{m_1,1} \cdot \chi^{\tau-d}_{m_2,1} \\ \chi^{\tau}_{n,0} &= \chi^{\tau-d}_{m_1,0} + \chi^{\tau-d}_{m_2,0}.\end{aligned}$$

The first equation for value 1 at n implies that the stability of n at value 1 at $t = \tau$ is guaranteed if both fanins are stable at value 1 by $t = \tau - d$. Therefore, the equivalent delay tuple is (d,d). The second equation is for value 0. The interpretation of the equation is that the stability of n at value 0 at $t = \tau$ is guaranteed if either of the fanins is stable at value 0 by $t = \tau - d$. This is captured by two delay tuples $(d,-\infty)$ and $(-\infty,d)$. The first delay tuple is applicable if $m_1 = 0$ while the second is applicable if $m_2 = 0$. If $m_1 = m_2 = 0$, both of them are valid.

This observation leads to the following delay abstraction for the two-input AND gate implicitly assumed in XBD0 analysis when a delay value d is given.

(m_1,m_2)	$d_{m_1 \to n} d_{m_2 \to n}$
00	$\{(d,-\infty),(-\infty,d)\}$
01	$\{(d,-\infty)\}$
10	$\{(-\infty,d)\}$
11	$\{(d,d)\}$.

In the general case where the functionality of a gate is a complex function, the recursive definition of χ function is:

$$\chi^{\tau}_{n,v} = \sum_{p \in P^v_n} [\prod_{m_i \in p} \chi^{\tau-d}_{m_i,1} \cdot \prod_{\overline{m_i} \in p} \chi^{\tau-d}_{m_i,0}]$$

where P^1_n and P^0_n are the sets of all primes of f_n and $\overline{f_n}$ respectively. Each prime $p \in P^v_n$ gives a delay tuple $(d_1,\ldots,d_k)$, where $d_i = d$ if $m_i \in p$ or $\overline{m_i} \in p$, and $d_i = -\infty$ otherwise. The delay tuple is assigned to all input vectors where p evaluates to 1.

In summary, a gate is assumed to be an atomic functional unit whose output becomes stable d time units after a subset of fanins that are enough to determine an output value is all stabilized. Each prime meets the condition of the subset defined above since if all the literals of a prime have final values, the output of the gate is uniquely determined without waiting for the other inputs to get stabilized.

We have shown that the delay propagation mechanism of a gate underlying in XBD0 analysis can be captured as an input-vector dependent delay abstraction. Given this, functional arrival time analysis using χ functions in Section 2.4.6 can be thought of as a delay analysis algorithm for

a network where each gate is characterized as a delay abstraction of a special form. We are interested in whether it is possible to generalize the algorithm so that it can handle a network of complex gates each of which is characterized by a delay abstraction of an arbitrary form. Specifically, if the recursive definition of χ functions can be extended so that an arbitrary input-vector-dependent delay abstraction is allowed, one can simply mimic regular XBD0 analysis for all the other tasks.

This generalization is in fact possible. Let n be a gate in a network. Let $m_1, \ldots, m_k$ be the fanins of n. Assume that the delay abstraction of this node is available. The recursive construction of χ function is modified as follows. Each delay tuple in the delay abstraction gives a condition that the output is guaranteed to be stable. For example, suppose there is a delay tuple $(d_1, \ldots, d_k)$ under input vector $\mathbf{x} = (v_1, \ldots, v_k)$. Suppose that $\mathbf{x}$ is an on-set vector of n. This tuple indicates that if fanin m_i is available at $t = \tau - d_i$ at value v_i for all $i = 1, \ldots, k$, the output stability at 1 is guaranteed at $t = \tau$. This is the same as having $\prod_{i=1}^{k} \chi_{m_i,v_i}^{\tau-d_i}$ in the recursive definition of χ functions. Since the output stability to value 1 is guaranteed by any of those delay tuples under on-set vectors, we can take the disjunction of all the products created from the delay tuples. Thus,

$$\chi_{n,1}^{\tau} = \sum_{(\mathbf{x},(d_1,\ldots,d_k))\in D, f_n(\mathbf{x})=1} \prod_{i=1}^{k} \chi_{m_i,v_i}^{\tau-d_i}$$

where $\mathbf{x} = (v_1, \ldots, v_k)$. The χ function for value 0 can be defined in a similar way.

$$\chi_{n,0}^{\tau} = \sum_{(\mathbf{x},(d_1,\ldots,d_k))\in D, f_n(\mathbf{x})=0} \prod_{i=1}^{k} \chi_{m_i,v_i}^{\tau-d_i}.$$

Note that $\chi_{m_i,v_i}^{\infty} = m_i$ if $v_1 = 1$ and $\overline{m_i}$ if $v_i = 0$. This applies if a delay tuple has a delay value $-\infty$ under some input. Further simplification is possible in this case. Let $(d_1, \ldots, d_k)$ be a delay tuple under $\mathbf{x} = (v_1, \ldots, v_k)$. Assume that $\mathbf{x}$ is in the on-set of the gate. Suppose $d_j = -\infty$. This means that the functionality of the gate is independent of d_j under $\mathbf{x}$. Thus, we should have the same delay tuple under input vector $\mathbf{x}'$ that is obtained from $\mathbf{x}$ by flipping the phase of v_j. The delay tuple under $\mathbf{x}$ gives a product term $\prod_{i=1}^{k} \chi_{m_i,v_i}^{\tau-d_i} = \chi_{m_j,v_j}^{\infty} \prod_{i=1,i\neq j}^{k} \chi_{m_i,v_i}^{\tau-d_i} = m_j \prod_{i=1,i\neq j}^{k} \chi_{m_i,v_i}^{\tau-d_i}$. The delay tuple under $\mathbf{x}'$ contributes a product term $\overline{m_j} \prod_{i=1,i\neq j}^{k} \chi_{m_i,v_i}^{\tau-d_i}$. If we take the disjunction of these two product terms, the result is $\prod_{i=1,i\neq j}^{k} \chi_{m_i,v_i}^{\tau-d_i}$. Therefore, if there is a delay tuple where some component is $-\infty$, we can safely drop the corresponding input from our consideration in its product term. This can be generalized to the case where more than one component of a delay tuple is $-\infty$.

Now that we know how to construct χ functions for arbitrary delay abstractions, the remaining analysis can be performed in the same way as in regular XBD0 analysis, where the determination of a delay estimate is reduced to the satisfiability problem. A potential difficulty of this

approach is that each gate in the network generates too many nodes in the χ network since the delay abstraction of a gate can be very complex with many delay tuples. The practicality of this approach is yet to be determined by experiments.

5.5.3 Other Approaches to Delay Computation using Input-Vector Dependent Delay Abstractions

The previous subsection showed how regular XBD0 analysis can be generalized so that input-vector dependent delay abstractions are handled in the construction of χ functions. A similar generalization is possible for other known techniques for delay analysis.

Devadas *et al.* [DKMW93] formulate functional arrival time analysis of flat circuits as a special type of ATPG, termed timed ATPG, where a pair of a value and a time, called a timed event, is propagated and justified. In order to see if an output is stable by $t = \tau$, timed events $(0, \tau)$ and $(1, \tau)$ are asserted at the output and checked to see if either event is justifiable under some input vector available at specified arrival times. Since the timed ATPG is based on PODEM [Goe81], a timed event is only propagated forward. Therefore, the use of input-vector dependent delay abstractions in this framework is simple. We just need to compute an output timed event from timed events at the fanins of a gate by respecting its input-vector dependent delay abstraction. The only complication is the possibility of having more than one delay tuple under an input vector. We need to take the earliest timed event over all the delay tuples applicable to the situation under analysis.

Yalcin and Hayes [YH95] proposed an algorithm where conditional events are propagated symbolically. The only restriction is that they can only handle a delay abstraction where each input vector has one delay tuple. The generalization to the case without this restriction is trivial. We have only to propagate the earliest event.

5.5.4 Computing Input-Vector Dependent Delay Abstractions of Subhierarchies

If the depth of a hierarchy is just one, by replacing the delay propagation algorithm in Section 5.2 with the generalized XBD0 analysis presented in Section 5.5.2 hierarchical analysis can be performed using delay abstraction dependent on input vectors.

If the depth of a hierarchy is more than one, we need to consider one more problem, i.e. how to create a delay abstraction of an intermediate subhierarchy from the delay abstractions of the subcircuits at that level, where all the delay abstractions can be input-vector dependent. A similar problem was discussed in Section 5.2.3 for the case where delay abstractions are input-vector inde-

pendent. Even under this simplified situation, we showed that the possibility of having more than one delay tuple makes the problem difficult since it is necessary to keep track of which delay tuple is used for each subcircuit during required time analysis. In the general case where delay abstractions are input-vector dependent, we need to do the same analysis for each input vector separately. Suppose we have a circuit composed of subcircuits, each of which has an input-vector dependent delay abstraction. By choosing an input vector of the circuit, the local input vector of each subcircuit is fixed. By referring to the corresponding delay abstraction under the vector, we can obtain a set of delay tuples to be used for the subcircuit under this vector. Now a set of delay tuples for the entire circuit under the global input vector can be computed in the same way as in Section 5.2.3. This approach is obviously expensive since each input vector needs to be analyzed independently. Whether there is an efficient algorithm to handle this problem is yet to be seen.

As a special case, if we restrict ourselves to the case where any delay abstraction has one delay tuple for each input vector, there is no need to keep track of the choice of delay tuples, which makes the problem easier. All input vectors can be processed symbolically by using ADDs [BFG+93] or MTBDDs [CMZ+93].

5.6 Experimental Results

We have implemented the improved timing analysis algorithm described in Section 5.4 on top of SIS [SSM+92]. CPU time reported in this section was measured on DEC AlphaServer 8400 5/300 and is reported in seconds. In all experiments the unit delay model was used.

Table 5.1 shows the results of hierarchical arrival time analysis of various types of carry-skip adders. Each circuit csa$n.m$ is an n-bit adder structured as a cascade connection of n/m m-bit carry-skip adders. The same circuits were also analyzed by flat timing analysis [MSBSV93] after expanding out all the existing hierarchy to get equivalent flat circuits. In all the circuits primary inputs were assumed to arrive at $t = 0$. Accuracy of estimated delay was fully preserved in all cases. CPU time saving of hierarchical analysis is significant.

This example of carry-skip adders is suited very well for hierarchical analysis since the circuit structure is regular, i.e. the same leaf module is used repeatedly. Therefore once the accurate delay abstraction of a component carry-skip adder is obtained, it can be used many times to improve the accuracy of the delay estimate of the entire circuit.

It is interesting to see how the algorithm performs on hierarchical circuits with irregular structures. Although ideally this experiment should be done on realistic hierarchical circuits, we

circuit	topological delay	hierarchical analysis		flat analysis	
		estimated delay	CPU time	estimated delay	CPU time
csa32.4	82	26	0.3	26	7.1
csa64.4	162	42	0.3	42	33.4
csa128.4	322	74	0.3	74	173.1
csa32.8	74	26	1.1	26	6.0
csa64.8	146	34	1.1	34	30.5
csa128.8	290	50	1.1	50	151.5
csa32.16	70	38	5.5	38	3.9
csa64.16	138	42	5.5	42	23.4
csa128.16	274	50	5.5	50	116.2

Table 5.1: Timing Analysis of Carry-Skip Adders – Hierarchical vs. Flat

circuit	topological delay	hierarchical analysis		flat analysis	
		estimated delay	CPU time	estimated delay	CPU time
C1908	40	38	62.7	37	5.7
C2670	32	31	5.5	30	19.8
C3540	47	46	24.2	46	8.1
C5315	49	47	7.5	47	2.2
C6288	124	124	51.0	123	273.9
C7552	43	42	4.0	42	1.2

Table 5.2: Timing Analysis of ISCAS Circuits – Hierarchical vs. Flat

had no hierarchical benchmark circuit available. Therefore we created artificial hierarchical circuits from ISCAS combinational circuits in the following way. Each benchmark circuit was partitioned into two circuits in a cascade structure so that one circuit drives the other. We then assumed that each partitioned circuit is a leaf module. This way we constructed a simple hierarchical circuit from a benchmark circuit.

Each hierarchical circuit was then analyzed by the algorithm in Section 5.4. Original benchmark circuits were also analyzed by flat timing analysis to compare accuracy of delay estimates and CPU time. Table 5.2 summarizes the results of this experiment. We were able to confirm that accuracy is preserved well in hierarchical analysis although small overestimation occurred on some circuits. In our current approach only locally false paths are detected. Therefore globally false paths which are false due to the interaction of various leaf modules are overlooked. The fact that accuracy is maintained reasonably well in this experiment indicates that many false paths in real circuits are in fact locally false.

Since these circuits are not large enough, flat analysis can finish timing analysis very quickly. Therefore we could not see any speedup in terms of CPU time for the case where a delay estimate of hierarchical analysis matches that of flat analysis. In fact, since functional arrival time analysis is performed repeatedly on smaller circuits in hierarchical analysis, it takes more time to complete the analysis than flat analysis in many cases. However this result should not be taken negatively. The underlying algorithm for both hierarchical and flat analyses is functional arrival time analysis of a flat combinational network. In the hierarchical approach the analysis is performed only on a single leaf module while it is performed on an entire circuit in flat analysis. Given that functional arrival time analysis can only be applied to circuits of a limited size, it is clear that hierarchical analysis is more scalable[13].

5.7 Related Work

Das *et al.* [DJCM89] and Johannes *et al.* [JCM92, Joh93] presented hierarchical timing analysis algorithms based on static sensitization. Since static sensitization can underestimate true delays, their approach is not conservative for timing verification.

[13] An alternative is to perform flat analysis of subcircuits in a topological order. The accuracy loss of this method is the same as that of hierarchical analysis since only locally false paths can be detected. However, each instance of the same module must be analyzed separately given different arrival times at its inputs. Furthermore incremental analysis capability is very limited since a modification of a subcircuit invalidates all the analyses for its transitive fanout cone whereas in hierarchical timing analysis the delay abstractions of the modules in the transitive fanout cone are still valid.

More recently Yalcin and Hayes [YH95] studied hierarchical timing analysis using conditional delay matrices. They characterize the delay of a combinational module by functional arrival time analysis under a fixed arrival time condition. This step is called *tagged-mode analysis* and is essentially the same as the delay characterization based on path classification studied in Section 4.6.1. Although they use the resulting delay abstraction for arbitrary arrival time conditions, no argument was given on the correctness of the approach. As discussed in Section 4.6.1, the approach is not conservative if the floating-mode condition is used as a path sensitization condition.

In this chapter we have performed hierarchical timing analysis by respecting the hierarchy of a given circuit. However, hierarchy is typically introduced for designers to simplify design tasks and thus the given hierarchy may not be suitable for timing analysis to achieve accurate delay estimation. Johannes *et al.* [JCM93, Joh93] presented how to find a hierarchy appropriate for efficient and accurate timing analysis. By noticing that a path becomes false due to a structural reconvergence, a heuristic was developed for generating a hierarchy effective for timing analysis. The idea is applicable to our techniques.

5.8 Conclusions

We have proposed a hierarchical functional arrival time analysis technique for combinational circuits. The analysis proceeds in a bottom-up fashion by following the original hierarchical structure of a circuit. At the lowest level of a hierarchy the delay abstraction of each leaf module is computed. Various techniques studied in Chapter 4 are directly applicable. False paths inside a leaf module are correctly identified in this step. We then compute the delay abstraction of each intermediate subhierarchy using the delay abstractions of subcircuits directly below that level. At the top level of the hierarchy delay computation is performed using the delay abstractions of the subcircuits at the level. If the delay abstractions are input-vector independent, the top-level delay computation is achieved by a variation of topological analysis. Although it is efficient, only a subset of false paths can be detected this way potentially resulting in delay overestimation. For the case where the delay abstractions are input-vector dependent, we have shown that the delay computation problem can be formulated as a generalization of the XBD0 analysis in Section 2.4.6.

Unlike conventional flat analysis the hierarchical approach never analyzes an entire circuit at the gate level at one time, and thus can potentially handle larger circuits than flat analysis. It is well-suited for the analysis of industrial circuits whose analysis is difficult or too slow for flat analysis. The hierarchical approach naturally supports incremental analysis capability, which is completely

missing in flat analysis.

We have experimentally shown that the hierarchical approach maintains enough accuracy on benchmark circuits even if approximate delay abstractions independent of input vectors are used. Further experiments are necessary to examine the effectiveness of this approach for larger industrial circuits.

Although the approach has been described for combinational circuits, generalization to sequential circuits with edge-triggered flip-flops is trivial. An extension of the approach to level-sensitive latches remains as future work.

Another potential research direction is input vector generation. Although the analysis presented in this chapter gives the delay estimate of a circuit, an input vector that realizes the delay is not explicitly generated by the algorithm. Such an input vector can be used in timing simulation at a lower level of abstraction for detailed analysis, and thus is of practical use.

The delay model assumed in this chapter is a simplistic model where the effect of load and slew is completely ignored. Adaptation of the hierarchical analysis for a more realistic delay model is an important research direction yet to be explored.

Chapter 6

Timing-Safe Replaceability for Combinational Modules

Chapter 4 showed that the delay abstraction of a combinational module can be computed exactly under the XBD0 model by considering false paths of the module. The delay abstraction is dependent on input vectors so that it can characterize paths conditionally false under some vectors. In addition, to guarantee the accuracy of the abstraction for arbitrary arrival time conditions at the inputs, more than one delay tuple, representing effective delays between the inputs and the output, can be maintained for an input vector. The best delay tuple that gives the most accurate delay estimate is chosen adaptively given an arrival time condition at the inputs.

The problem addressed in this chapter is how to compare the timing characteristics of two combinational modules that are functionally equivalent, but have different timing behaviors. Specifically, we are interested in a condition that one module is no slower than the other *under any arrival time condition at the inputs*. If this condition holds between two combinational modules, the latter module can be safely replaced with the former under any surrounding environment without the risk of deteriorating the original performance. To formalize this idea a new notion called *timing-safe replaceability* will be introduced for combinational modules. We will show that whether one is a timing-safe replacement of the other can be determined given the exact delay abstractions of the two modules.

This chapter is organized as follows. Section 6.1 introduces the notion of timing-safe replaceability and discusses how to determine whether a combinational module is a timing-safe replacement of another. Examples are given in Section 6.2 to illustrate the notion. Section 6.3 dis-

cusses concurrent timing optimization of combinational circuits as an application of timing-safe replaceability. Section 6.4 concludes the chapter.

6.1 Timing-Safe Replaceability

Suppose that we have a combinational module $\mathcal{M}_{org}$ and another combinational module $\mathcal{M}_{new}$ that is claimed to be a sped-up version of $\mathcal{M}_{org}$. Assume that they are single-output modules and functionally equivalent. We are interested in verifying whether $\mathcal{M}_{new}$ is indeed no slower than $\mathcal{M}_{org}$. More specifically, we need to verify whether using $\mathcal{M}_{new}$ instead of $\mathcal{M}_{org}$ worsens the delay through this module under some vector and arrival times at the inputs of the module. If there exist such a vector and arrival times, $\mathcal{M}_{new}$ is not a timing-safe replacement of $\mathcal{M}_{org}$ since under that particular situation the use of $\mathcal{M}_{new}$ instead of $\mathcal{M}_{org}$ in fact deteriorates the performance of $\mathcal{M}_{org}$. Throughout this chapter we assume that the only timing property that needs to be preserved is when the output of the module is stabilized. It is acceptable for the output to be available earlier than in the original module, but it should never become stable later.

Definition 6.1 *$\mathcal{M}_{new}$ is said to be a* timing-safe replacement *of $\mathcal{M}_{org}$, $\mathcal{M}_{new} \preceq_M \mathcal{M}_{org}$, if there exists no (input vector, arrival times)-pair at the inputs such that the output becomes stable later in $\mathcal{M}_{new}$ than in $\mathcal{M}_{org}$.*

The partial order $\preceq$ over delay abstractions was originally introduced in Definition 4.4 to compare the accuracy of approximate delay abstractions for the same combinational module. We generalize this partial order so that delay abstractions of functionally equivalent circuits can be compared against each other.

Definition 6.2 *Let D_1 and D_2 be the exact delay abstractions of single-output combinational modules $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively, where $\mathcal{M}_1$ and $\mathcal{M}_2$ are functionally equivalent and have the same input/output interface. $D_1 \preceq D_2$ if for every (input vector,delay tuple)-pair $(\mathbf{x}, (d_1, \ldots, d_n)) \in D_2$, there exists an (input vector,delay tuple)-pair $(\mathbf{x}, (d'_1, \ldots, d'_n)) \in D_1$ such that $(d'_1, \ldots, d'_n) \trianglelefteq (d_1, \ldots, d_n)$.*

The following theorem states that, given two combinational modules, whether one is a timing-safe replacement of the other can be determined by checking if a certain property holds between their exact delay abstractions. Note that all the properties proved for $\preceq$ in Chapter 4 still hold for this new definition of $\preceq$.

Theorem 6.1 *Let D_{org} and D_{new} be the exact delay abstractions of $\mathcal{M}_{org}$ and $\mathcal{M}_{new}$ respectively. $\mathcal{M}_{new} \preceq_M \mathcal{M}_{org}$ if and only if $D_{new} \preceq D_{org}$.*

Proof Trivial from Theorem 4.1. □

The following properties of $\preceq_M$ can be directly obtained from the corresponding properties of $\preceq$ proved in Chapter 4.

Theorem 6.2 *$\preceq_M$ is reflexive.*

Proof Trivial from Theorem 4.2. □

Theorem 6.3 *$\preceq_M$ is transitive.*

Proof Trivial from Theorem 4.3. □

Theorem 6.4 *$\preceq_M$ is antisymmetric.*

Proof Trivial from Theorem 4.4. □

The notion of safe replaceability was first proposed by Singhal and Pixley [SP94] for sequential circuits, where a sequential circuit is said to be a safe replacement of another if the replacement of the latter with the former is never detectable under any surrounding environment in terms of functionality. This notion allows us to take an arbitrary piece of sequential logic and replace it with another regardless of the surrounding logic. Timing-safe replaceability proposed here is a natural extension of the safe replaceability notion to the timing domain since one can safely replace a combinational module with another without increasing the delay through the module under any surrounding environment (i.e. arrival time condition) if the latter is a timing-safe replacement of the former.

Aziz *et al.* [ABBS95] proposed a different notion of timing-safe replaceability, where a circuit is called a timing-safe replacement of an original circuit if and only if the delay range of the output is completely contained in the delay range of the original. Note that each gate is given a minimum delay and a maximum delay. Our definition is more relaxed since the only delay property of our interest is maximum delay. Therefore, speeding up a circuit preserves timing-safe replaceability

in our definition while it may not in the definition of Aziz *et al.* Moreover, it is much more expensive to determine if one circuit is a timing-safe replacement of another in the notion of Aziz *et al.* than in the notion defined here.

6.2 Examples

Consider a circuit $\mathcal{M}$ shown in Figure 6.1 taken from [BI88]. Assume the unit delay model. The exact delay abstraction D of $\mathcal{M}$ is:

abc	$d_{a\to g}d_{b\to g}d_{c\to g}$
000	$\{(3,-\infty,2),(-\infty,3,2)\}$
001	$\{(2,2,2)\}$
010	$\{(3,-\infty,2),(-\infty,2,-\infty)\}$
011	$\{(-\infty,2,-\infty)\}$
100	$\{(2,-\infty,-\infty),(-\infty,3,2)\}$
101	$\{(2,-\infty,-\infty)\}$
110	$\{(2,-\infty,-\infty),(-\infty,2,-\infty)\}$
111	$\{(2,-\infty,-\infty),(-\infty,2,-\infty)\}$.

The input edge of path $P_a = (a,d,f,g)$ is both stuck-at-0 and stuck-at-1 redundant. Figure 6.2 shows the circuit $\mathcal{M}_0$ obtained from $\mathcal{M}$ by removing the stuck-at-0 redundancy at the input edge of path P_a. The delay abstraction D_0 of $\mathcal{M}_0$ is:

abc	$d_{a\to g}d_{b\to g}d_{c\to g}$
000	$\{(-\infty,-\infty,1)\}$
001	$\{(2,2,1)\}$
010	$\{(-\infty,-\infty,1),(-\infty,2,-\infty)\}$
011	$\{(-\infty,2,-\infty)\}$
100	$\{(2,-\infty,-\infty),(-\infty,-\infty,1)\}$
101	$\{(2,-\infty,-\infty)\}$
110	$\{(2,-\infty,-\infty),(-\infty,2,-\infty),(-\infty,-\infty,1)\}$
111	$\{(2,-\infty,-\infty),(-\infty,2,-\infty)\}$.

It is easy to see that $D_0 \preceq D$. For example, under input vector $(0,0,0)$, D has two timing tuples, $(3,-\infty,2)$ and $(-\infty,3,2)$. D_0 has a single tuple $(-\infty,-\infty,1)$ for this vector, which gives $(-\infty,-\infty,1) \trianglelefteq$

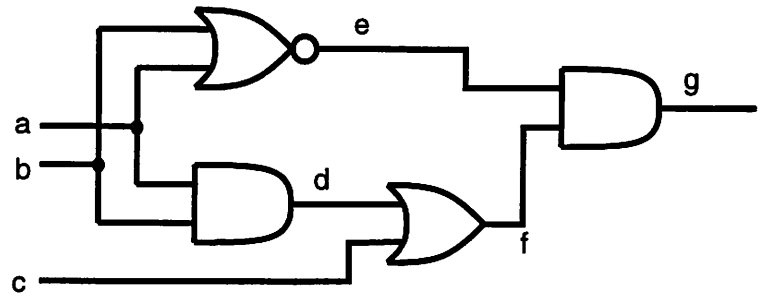

Figure 6.1: Example: A Combinational Module $\mathcal{M}$

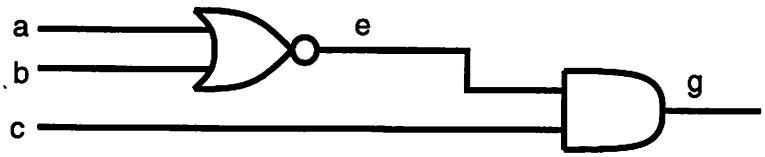

Figure 6.2: Example: A Timing-Safe Replacement Module $\mathcal{M}_0$

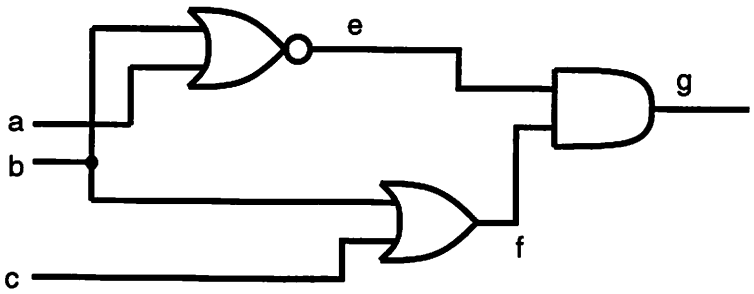

Figure 6.3: Example: A Timing-Non-Safe Replacement Module $\mathcal{M}_1$

$(3,-\infty,2)$ and $(-\infty,-\infty,1) \trianglelefteq (-\infty,3,2)$. All the other input vectors have this property. Therefore, from Theorem 6.1, $D_0 \preceq D$ implies $\mathcal{M}_0 \preceq_M \mathcal{M}$. $\mathcal{M}_0$ can always be used instead of $\mathcal{M}$ under any environment without having a negative impact on delay.

If we remove the stuck-at-1 redundancy of the input edge instead, another circuit $\mathcal{M}_1$ shown in Figure 6.3 is obtained. The delay abstraction D_1 of $\mathcal{M}_1$ is:

abc	$d_{a\to g}d_{b\to g}d_{c\to g}$
000	$\{(-\infty,2,2)\}$
001	$\{(2,2,2)\}$
010	$\{(-\infty,2,-\infty)\}$
011	$\{(-\infty,2,-\infty)\}$
100	$\{(2,-\infty,-\infty),(-\infty,2,2)\}$
101	$\{(2,-\infty,-\infty)\}$
110	$\{(2,-\infty,-\infty),(-\infty,2,-\infty)\}$
111	$\{(2,-\infty,-\infty),(-\infty,2,-\infty)\}$.

$D_1 \not\preceq D$ since, for example, under input vector $(0,0,0)$ D has a delay tuple $(3,-\infty,2)$, but the only timing tuple is $(-\infty,2,2)$ in D_1, where $(-\infty,2,2) \not\trianglelefteq (3,-\infty,2)$. Therefore, if $(a,b,c)=(0,0,0)$ and $(arr(a),arr(b),arr(c))=(-3,0,-2)$, the output becomes stabilized at $t=0$ in $\mathcal{M}$ while under the same condition the output only becomes stable at $t=2$ in $\mathcal{M}_1$. $D_1 \not\preceq D$ implies $\mathcal{M}_1 \not\preceq_M \mathcal{M}$.

6.3 Application: Concurrent Timing Optimization of Combinational Circuits

Consider timing optimization of combinational circuits. Suppose that a combinational network does not meet a given timing constraint and needs to be sped up. Assume that instead of optimizing the network as a single circuit, it is partitioned into subnetworks, each of which is then optimized and replaced with an optimized subnetwork. Furthermore, consider a scenario that the timing optimization of the subnetworks is performed concurrently not sequentially. This is a realistic scenario in the context of hierarchical synthesis. For example, if a design is partitioned into blocks, it is often the case that synthesis is performed respecting the partition, i.e. each component is synthesized separately. In many cases different blocks belong to different designers, and thus they are likely to be optimized concurrently.

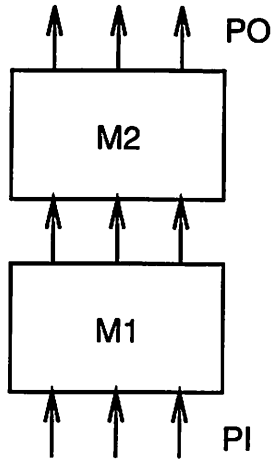

Figure 6.4: Concurrent Timing Optimization of Combinational Circuits

Consider an example shown in Figure 6.4, where two combinational circuits are connected in cascade. Suppose that we need to speed up the circuit by optimizing $\mathcal{M}_1$ and $\mathcal{M}_2$ separately.

First, consider timing optimization where delays are estimated by topological delays. Under this delay model, as long as each component is optimized so that the topological delay between each input-output pair within the component never increases, the optimized component is never slower than the original no matter how it is used. Therefore, the use of optimized components always gives a better performance thereby making concurrent timing optimization easy.

Now assume that the delay of the circuit is estimated more accurately by considering false paths. This makes the timing interaction of the two components tighter. Suppose $\mathcal{M}_1$ is optimized so that the outputs of the module arrive no later than the original under the original arrival time condition at the module inputs. Let $\mathcal{M}'_1$ be the optimized circuit. Suppose $\mathcal{M}_2$ is optimized in the same way. Let $\mathcal{M}'_2$ be the optimized circuit.

The inputs of $\mathcal{M}_1$ are primary inputs of the entire circuit. Therefore, the arrival time condition at the inputs is preserved even after the timing optimization. Thus, if we replace $\mathcal{M}_1$ with $\mathcal{M}'_1$, the signals on the component boundary arrive earlier than before. If we keep $\mathcal{M}_2$ as it is, the replacement of $\mathcal{M}_1$ with $\mathcal{M}'_1$ is guaranteed not to slow down the circuit since the new arrival time condition at the inputs of $\mathcal{M}_2$ is a monotone speed-up of the original, and thus never increases the delay estimation of $\mathcal{M}_2$ at the outputs of the module.

Now suppose $\mathcal{M}_2$ is concurrently optimized with $\mathcal{M}_1$ assuming that the outputs of the first-level module arrive in the same way as in the original configuration. It is possible that $\mathcal{M}'_2$ with $\mathcal{M}'_1$

gives performance worse than $\mathcal{M}_2$ with $\mathcal{M}'_1$. The reason is that $\mathcal{M}'_2$ is guaranteed to be no slower than $\mathcal{M}_2$ only under the original arrival time condition. Under a different arrival time condition, the original circuit $\mathcal{M}_2$ can be faster than the optimized circuit $\mathcal{M}'_2$. This is not desirable since the effort of timing optimization of $\mathcal{M}_2$ may be nullified depending on the optimization performed for $\mathcal{M}_1$. We are interested in a more consistent timing optimization scenario in which the optimization of each component and the use of the resulting component always gives better performance.

The problem above is that $\mathcal{M}_2$ was optimized just for a particular arrival time condition as if it were fixed. In the context of concurrent timing optimization, the arrival time condition at the inputs of $\mathcal{M}_2$ can change. More specifically, the arrival time condition can be sped up as the result of timing optimization of $\mathcal{M}_1$ although the actual arrival time condition is unknown since the two modules are optimized concurrently. Therefore, in order to guarantee that an optimized circuit always gives performance no worse than the original, the timing optimization of $\mathcal{M}_2$ needs to be done so that the resulting circuit $\mathcal{M}'_2$ is no slower than $\mathcal{M}_2$ under any arrival time condition which is a sped-up version of the original.

The notion of timing-safe replaceability is suitable in this context. Suppose that a new circuit $\mathcal{M}'_2$ is a timing-safe replacement of $\mathcal{M}_2$. It is guaranteed that $\mathcal{M}'_2$ is no slower than $\mathcal{M}_2$ under any arrival time condition. Although the notion is stronger than necessary[1] for this application, timing optimization under the notion of timing-safe replaceability gives the property of consistent optimization[2].

A timing optimization flow based on the notion of timing-safe replaceability is as follows. A combinational circuit is partitioned into subcircuits, each of which is optimized so that a new subcircuit is a timing-safe replacement of the original. All the subcircuits are optimized simultaneously under this condition. We can then guarantee that a final circuit constructed by any combination of the optimized subcircuits gives performance no worse than the original.

6.4 Conclusions

A new notion called timing-safe replaceability has been proposed for combinational modules. If a combinational module is a timing-safe replacement of another, the delay through the former

[1] We only need the performance guarantee under sped-up arrival time conditions.

[2] Timing optimization under the notion of timing-safe replaceability has a similar flavor to functional optimization under a compatible set of permissible functions (CSPF) [MKLC89]. The timing constraint posed on a component under timing-safe replaceability is tighter than that under the case where the component is optimized while all the other components are fixed (corresponding to a maximum set of permissible functions (MSPF) [MKLC89] in functional optimization). However, it allows us to perform concurrent optimization as in CSPF-based logic simplification.

module is guaranteed not to be larger than that through the latter under any arrival time condition. Thus, no matter what the surrounding environment of the original module is, the use of a timing-safe replacement gives performance no worse than that of the original. The notion of timing-safe replaceability gives a criterion for comparing the timing characteristics of combinational modules under unknown arrival times at primary inputs. We have shown that timing-safe replaceability can be determined if the exact delay abstractions of two modules are given.

Although a decision procedure for timing-safe replaceability was given based on the analysis of the exact delay abstractions of two modules, the computation of the exact delay abstractions itself is difficult for large circuits, which creates a bottleneck. An efficient way to verify whether a circuit is a timing-safe replacement of another needs to be investigated.

Another interesting direction is to study timing optimization techniques whose optimization results are provably timing-safe replacements of the original. One timing optimization scheme that guarantees this property is speeding up gate delays without changing the structure of a network [CDL93]. Due to the monotone speed-up property of the XBD0 model, the resulting circuit is guaranteed to be a timing-safe replacement of the original. To the best of our knowledge all the other existing timing optimization algorithms restructure a given circuit under a particular arrival time condition so that the resulting circuit is faster than the original under the condition. Thus, none of them meets this requirement. Timing optimization techniques satisfying the requirement alleviates the need for potentially expensive verification of timing-safe replaceability. In Chapter 8, a technique in this category will be proposed.

Chapter 7

Strongly False Paths in Combinational Modules

Chapter 4 showed that the delay characteristics of a combinational module can be compactly represented in the form of a delay abstraction, in which the effective input/output delay information of the module under each input vector is separately stored so that false paths under the vector are taken into consideration. One of the important observations was that the effective input/output delay may not be unique even after an input vector is specified, and can vary depending on arrival time conditions at the inputs. This directly implies that the falsity of a path in a combinational module is not a property independent of arrival times at the inputs, but is relative to arrival times at primary inputs. Therefore, the same input-output path of a module can be true under some arrival time condition, while false under another. This chapter introduces a more stringent notion of false paths, termed *strongly false paths*, where a path is said to be strongly false if it is false under *any* arrival time condition. Such false paths can be defined uniquely for a combinational module independent of arrival times at the primary inputs. Applications of this new class of false paths include false path removal of combinational modules, which will be discussed in Chapter 8.

This chapter is organized as follows. Section 7.1 starts with a motivating example showing that the falsity of a path is different for different arrival time conditions. We define a new class of false paths called strongly false paths for combinational modules in Section 7.2 and show that a strongly false path can be identified by examining the exact delay abstraction of a combinational module. Section 7.3 then proposes an algorithm for determining the strong falsity of a path without constructing the exact delay abstraction of a module. Relation between strong falsity and static

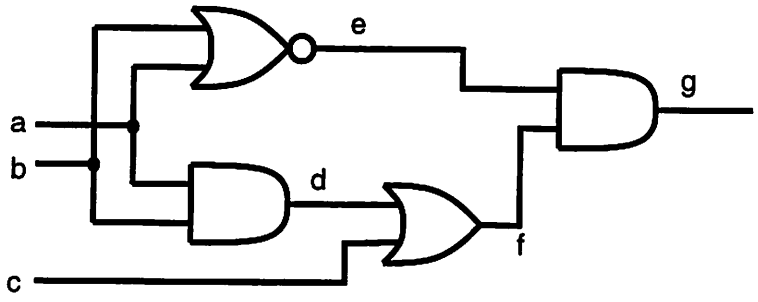

Figure 7.1: Example: A Combinational Module $\mathcal{M}$

co-sensitization is argued in Section 7.4. The chapter is concluded in Section 7.5.

7.1 Example

Consider a circuit $\mathcal{M}$ in Figure 7.1. Assume the unit delay model. If the primary inputs a, b and c arrive at $t = 1, 0$ and 1 respectively, functional arrival time analysis guarantees that the output g is stabilized at $t = 3$. Note that the topological delay of this circuit under the arrival time condition is $t = 4 (> 3)$ due to the path $P_a = (a, d, f, g)$. Since g is available at $t = 3$, P_a is false.

Consider another path $P_b = (b, d, f, g)$. Given an input vector $(a, b, c) = (0, 0, 0)$, P_b is true under this arrival time condition.

Now, let us analyze the same circuit under a different condition where a, b and c arrive at $t = 0, 1$ and 1 respectively. The output g is again available at $t = 3$. Therefore P_b is false since otherwise the output would become stable at $t = 4$. Note that P_b was true under the previous condition. P_a, which was false before, is true this time, for example, under input vector $(a, b, c) = (0, 0, 0)$.

These two cases clearly demonstrate that the falsity of a path is relative to a given arrival time condition, and that the same path can be false in one condition and true in another.

One can examine this circuit more systematically by computing the exact delay abstraction

of the circuit as described in Section 4.2. The exact delay abstraction D of this circuit is:

abc	$d_{a\to g}d_{b\to g}d_{c\to g}$
000	$\{(3,-\infty,2),(-\infty,3,2)\}$
001	$\{(2,2,2)\}$
010	$\{(3,-\infty,2),(-\infty,2,-\infty)\}$
011	$\{(-\infty,2,-\infty)\}$
100	$\{(2,-\infty,-\infty),(-\infty,3,2)\}$
101	$\{(2,-\infty,-\infty)\}$
110	$\{(2,-\infty,-\infty),(-\infty,2,-\infty)\}$
111	$\{(2,-\infty,-\infty),(-\infty,2,-\infty)\}$.

In this circuit P_a (P_b) is the only path of length 3 from a (b) to g. Therefore, the fact that the delay abstraction has a delay tuple whose first (second) element is 3 means that, given the corresponding input vector, it is possible to make P_a (P_b) true.

For example under $(a,b,c)=(0,0,0)$, when $(arr(a),arr(b),arr(c))=(1,0,1)$, the second delay tuple $(-\infty,3,2)$ gives an earlier signal stable time of $t=3$ at the output than the first delay tuple $(3,-\infty,2)$ giving the stable time of $t=4$. As described in Section 4.4, the delay tuple that gives the earliest stable time can be used in determining the timing behavior of the output. Since the delay from b to the output in this second delay tuple is 3, the corresponding path P_b is true. P_a is false since the delay tuple indicates that input a is irrelevant. If $(arr(a),arr(b),arr(c))=(0,1,1)$, however, the first delay tuple gives an earlier stable time than the second, showing that P_a is true and P_b is false.

7.2 Strongly False Paths

Definition 7.1 *Let $\mathcal{M}$ be a single-output combinational module whose primary inputs are $x_1,\ldots,x_n$. Let z be the primary output of the module. The path set $\Pi(x_i,L)$ is the set of all input-output paths that start from x_i and end at z and whose topological delays are greater than or equal to L.*

Definition 7.2 $\Pi(x_i,L)$ *is said to be* strongly false *under input vector* $\mathbf{x}$ *if no path in $\Pi(x_i,L)$ is responsible for the stability of the output under* $\mathbf{x}$ *in any arrival time condition at the inputs. $\Pi(x_i,L)$ is said to be strongly false if $\Pi(x_i,L)$ is strongly false under all input vectors.*

The following theorem shows that if the exact delay abstraction of a combinational module is available, strong falsity of paths can be determined easily.

Theorem 7.1 *The path set $\Pi(x_i, L)$ of a single-output combinational module $\mathcal{M}$ is strongly false under input vector* **x** *if and only if the exact delay abstraction D of $\mathcal{M}$ contains no delay tuple where the delay corresponding to x_i is greater than or equal to L under* **x**.

Proof

$\Leftarrow$ The effective delay from x_i to the output is less than L for any choice of the delay tuples under **x**. Thus, no path in $\Pi(x_i, L)$ is responsible for the stability of the output for any arrival time condition at the inputs.

$\Rightarrow$ We prove this by contradiction. Suppose that D has under **x** a delay tuple $(d_1, \ldots, d_n)$ where the delay from x_i, d_i, meets $d_i \geq L$. Consider two arrival time conditions

$$\begin{aligned} A &= (arr(x_1), \ldots, arr(x_n)) = (-d_1, \ldots, -d_n) \\ A' &= (-d_1, \ldots, -d_{i-1}, -d_i + \varepsilon, -d_{i+1}, \ldots, -d_n), \end{aligned}$$

where $\varepsilon > 0$. Under A, the output is stable at $t = 0$. However, if we change the arrival time condition to A', where ε is a small positive number, the output stable time is delayed to $t = \varepsilon$. Note that we can take ε small enough so that no other delay tuple under **x** gives better delay estimation than the current delay tuple. The output becomes stable later because some path of length $d_i \geq L$ in $\Pi(x_i, L)$ determines the output stability. A contradiction.

□

Corollary 7.1 $\Pi(x_i, L)$ *is said to be* strongly false *if and only if the exact delay abstraction D of $\mathcal{M}$ contains no delay tuple in which the delay corresponding to x_i is greater than or equal to L under any input vector.*

The idea behind this new definition of false paths is that a path set is said to be strongly false if all the paths in the set are false under *any* arrival time condition. This definition of falsity is more stringent than the standard definition of falsity where the falsity is argued under a specific arrival time condition.

Under this new definition neither $\Pi(a, 3) = \{P_a\}$ nor $\Pi(b, 3) = \{P_b\}$ in Figure 7.1 is strongly false since it is possible to sensitize these paths under some arrival time condition as we saw in the previous section.

We can also define another class of strongly false paths where the strong falsity of a path set is claimed only under a specific value at an input.

Definition 7.3 $\Pi(x_i, L)$ *is said to be* strongly false for value 0 (1) at x_i *if and only if* $\Pi(x_i, L)$ *is strongly false under all the input vectors that have value 0(1) for* x_i.

Note that $\Pi(x_i, L)$ is strongly false if $\Pi(x_i, L)$ is strongly false both for value 0 and value 1 at x_i.

In Figure 7.1 $\Pi(a, 3)$ is strongly false for 1 at a since under $a = 1$ there is no delay tuple where the delay from a is 3. Likewise $\Pi(b, 3)$ is strongly false for 1 at b.

This notion of strong falsity of a path set for a constant value at an input[1] will be exploited in Chapter 8. Specifically, we will show that any path set that is strongly false under some constant value can be removed safely without increasing the delay of the module under any arrival time condition.

7.3 Algorithm for Detecting Strongly False Paths

In the previous section we showed that strongly false paths of a combinational module can be easily identified if the exact delay abstraction of the module is available. However, the computation of the exact delay abstraction is expensive for large circuits. This section presents an algorithm for identifying strongly false paths of a combinational module without constructing the exact delay abstraction of the module. The algorithm reduces the problem to a satisfiability problem, which can be solved by a SAT solver. The use of χ functions introduced in Section 2.4.6 forms the basis of this reduction. Since there are practical SAT solvers available that scale to large problems, this approach is applicable to the analysis of large networks.

Assume that a combinational module $\mathcal{M}$ has primary inputs $x_1, \ldots, x_n$ and a single primary output z. We first compute χ functions for values 0 and 1 at z by assuming the required time of $t = 0$ at z as in functional required time analysis in Section 3.3. Let $\chi^0_{z,0}$ and $\chi^0_{z,1}$ denote the χ functions. These functions are in terms of leaf χ variables at the primary inputs. Let l_i be the longest topological path length from x_i to the output. We are interested in determining whether all the longest topological paths from x_i of length l_i, $\Pi(x_i, l_i)$ in the path set notation, are strongly false or not.

Intuitively if either of the two χ functions at the output z is sensitive to the signal stability of x_i at $t = -l_i$, the output stability depends on some path in $\Pi(x_i, l_i)$. Therefore, $\Pi(x_i, l_i)$ is not strongly

[1] Existing path sensitization conditions also have the notion of falsity under a specific value, but the focus is an output value. For example, static co-sensitization can be further classified into static co-sensitization to a value 0 and a value 1 [DKM93]. A path is said to be statically co-sensitizable to a value v if it is statically co-sensitizable and the output of the path is value v. Instead of focusing on output values our classification is based on the value of an input edge.

false. Note that the signal stability of x_i at $t = -l_i$ is represented by two leaf χ variables $\chi^{-l_i}_{x_i,0}$ and $\chi^{-l_i}_{x_i,1}$ for values 0 and 1 respectively. Recall that the two χ functions at the output are represented in terms of leaf χ variables including the two leaf χ variables above.

On the other hand, if both of the χ functions are independent of the two leaf χ variables, the stability of x_i at $t = -l_i$ never affects the output stability at $t = 0$. Thus, $\Pi(x_i, l_i)$ is strongly false.

Notice that neither an arrival time condition at the primary inputs nor an input vector applied to the module is known. Our goal is to check whether there are an arrival time condition and a vector at the inputs under which the stability of x_i at $t = -l_i$ directly determines the stability of z at $t = 0$. If such a pair exists, $\Pi(x_i, l_i)$ is not strongly false since under the condition some path in $\Pi(x_i, l_i)$ is responsible for stabilizing the output at $t = 0$. Otherwise, $\Pi(x_i, l_i)$ is strongly false.

A key operation here is to determine whether a function is dependent on a variable. This is required to determine if the χ functions at the output, $\chi^0_{z,0}$ and $\chi^0_{z,1}$, depend on input variables $\chi^{-l_i}_{x_i,0}$ and $\chi^{-l_i}_{x_i,1}$. Whether a function f depends on an input x can be tested by checking whether $f|_{x=0} \oplus f|_{x=1}$ is satisfiable, where $\oplus$ is the XOR operation. This is directly applicable here.

χ functions have a special property that they are monotone increasing in terms of leaf χ variables by construction (Lemma 3.1). Thus, the following holds.

If $x_i = 0$:

$$\chi^0_{z,0}|_{\chi^{-l_i}_{x_i,0}=0} \subseteq \chi^0_{z,0}|_{\chi^{-l_i}_{x_i,0}=1}$$
$$\chi^0_{z,1}|_{\chi^{-l_i}_{x_i,0}=0} \subseteq \chi^0_{z,1}|_{\chi^{-l_i}_{x_i,0}=1}$$

If $x_i = 1$:

$$\chi^0_{z,0}|_{\chi^{-l_i}_{x_i,1}=0} \subseteq \chi^0_{z,0}|_{\chi^{-l_i}_{x_i,1}=1}$$
$$\chi^0_{z,1}|_{\chi^{-l_i}_{x_i,1}=0} \subseteq \chi^0_{z,1}|_{\chi^{-l_i}_{x_i,1}=1}$$

We can take advantage of this property to simplify the dependency check. Namely it is enough to check whether either of the following is satisfiable.

If $x_i = 0$:

$$\chi^0_{z,0}|_{\chi^{-l_i}_{x_i,0}=1}\overline{\chi^0_{z,0}|_{\chi^{-l_i}_{x_i,0}=0}}, \text{ or}$$
$$\chi^0_{z,1}|_{\chi^{-l_i}_{x_i,0}=1}\overline{\chi^0_{z,1}|_{\chi^{-l_i}_{x_i,0}=0}}$$

If $x_i = 1$:

$$\chi^0_{z,0}|_{\chi^{-l_i}_{x_i,1}=1}\overline{\chi^0_{z,0}|_{\chi^{-l_i}_{x_i,1}=0}}, \text{ or}$$
$$\chi^0_{z,1}|_{\chi^{-l_i}_{x_i,1}=1}\overline{\chi^0_{z,1}|_{\chi^{-l_i}_{x_i,1}=0}}$$

Finally leaf χ variables are not fully independent. The leaf χ variables for the same primary input need to satisfy the ordering constraint discussed in Section 3.3. Therefore, the χ functions for the output is tested for its dependency on the leaf χ variables for x_i at $t = -l_i$ only under the ordering constraint.

This idea is generalized into the following theorem.

Theorem 7.2 *Let V be the set of all leaf χ variables of x_i for value $v (v = 0, 1)$ for any time $t \leq -L$, where $L \leq l_i$. $\Pi(x_i, L)$ is strongly false for value v at x_i if and only if neither $(x_i = v)\chi^0_{z,0}|_{\forall u \in V: u=1}$ $\overline{\chi^0_{z,0}|_{\forall u \in V: u=0}}$ nor $(x_i = v)\chi^0_{z,1}|_{\forall u \in V: u=1}$ $\overline{\chi^0_{z,1}|_{\forall u \in V: u=0}}$ is satisfiable under the ordering constraints for the leaf χ variables.*

Proof

$\Leftarrow$ The term $\chi^0_{z,0}|_{\forall u \in V: u=1}$ and $\chi^0_{z,0}|_{\forall u \in V: u=0}$ represent the sets of input vectors that make the output stable to 0 by $t = 0$ if x_i arrives at $t = -l_i$ and at $t = -L + \varepsilon$ respectively, where ε is a small positive number. Therefore, the fact that the product term $(x_i = v)\chi^0_{z,0}|_{\forall u \in V: u=1}$ $\overline{\chi^0_{z,0}|_{\forall u \in V: u=0}}$ is not satisfiable under the ordering constraint means that if $x_i = v$, there are no input vector and arrival time condition under which the output stability to value 0 is differentiated between the case where x_i arrives at $t = -l_i$, which is early enough to propagate any signal event up to the output from x_i, and the case where x_i only arrives at $t = -L + \varepsilon > -l_i$. Thus, no path in $\Pi(x_i, L)$ is responsible for stabilizing the output to value 0 by $t = 0$ if $x_i = v$. A similar argument holds for value 1. Hence, $\Pi(x_i, L)$ is strongly false for value v at x_i.

$\Rightarrow$ We prove this by contradiction. Suppose that $(x_i = v)\chi^0_{z,0}|_{\forall u \in V: u=1}$ $\overline{\chi^0_{z,0}|_{\forall u \in V: u=0}}$ is satisfiable under the ordering constraint. Let $\mathbf{x}$ and $a_j (j \neq i)$ be the input vector and the arrival time at x_j corresponding to a satisfying assignment respectively. Note that $\mathbf{x}$ meets $x_i = v$. Under $\mathbf{x}$ and $arr(x_j) = a_j (j \neq i)$, if $arr(x_i) = -l_i$, the output is stable to 0 by time 0 while it is not if $arr(x_i) = -L + \varepsilon$. This implies that some path in $\Pi(x_i, L)$ is responsible for the stability of the output. Thus, $\Pi(x_i, L)$ is not strongly false for value v at x_i. A contradiction. The case where the output is 1 can be handled in the same way.

□

The theorem guarantees that the strong falsity of a path set under an input value can be determined by performing two satisfiability checks. As in XBD0 analysis, we construct a single-output Boolean network whose output function is equal to a formula under a satisfiability check. Section 2.4.6 showed that χ functions can be represented in Boolean networks by following their recursive definitions.

Let us see how we can construct a Boolean network whose output has the functionality $(x_i = v)\chi^0_{z,0}\,|_{\forall u \in V: u=1}\,\overline{\chi^0_{z,0}\,|_{\forall u \in V: u=0}}$. We first make two copies of the Boolean network for $\chi^0_{z,0}$: $\mathcal{N}_1$ for $\chi^0_{z,0}\,|_{\forall u \in V: u=1}$ and $\mathcal{N}_0$ for $\chi^0_{z,0}\,|_{\forall u \in V: u=0}$.

In order to respect the ordering constraint among leaf χ variables, we employ the same technique used in the approximate required time analysis via simplified modeling in Section 3.3.2. Namely, each leaf χ variable is represented as the corresponding input variable with the appropriate phase multiplied by newly introduced Boolean variables. These Boolean variables referred to as α variables and β variables in Section 3.3.2 constrain leaf χ variables so that the ordering constraint is automatically satisfied.

Each leaf χ variable not for x_i is fed by a new node whose functionality is given in Equation (3.3). Notice that the node is shared between the two networks $\mathcal{N}_1$ and $\mathcal{N}_0$. A leaf χ variable for x_i, on the other hand, needs to be distinguished between the two networks since different constraints need to be imposed. Both in $\mathcal{N}_1$ and $\mathcal{N}_0$, each leaf χ variable for x_i is fed by a new node whose functionality is defined in Equation (3.3) as before. However, two different sets of α and β variables are introduced for leaf variables of x_i in $\mathcal{N}_1$ and $\mathcal{N}_0$. In $\mathcal{N}_1$ all α and β variables for x_i are set to 1 to represent that x_i arrives at $t = -l_i$. In $\mathcal{N}_0$ the α and β variables are set properly so that all leaf χ variables of x_i for $t \leq -L$ are 0 while the other leaf χ variables of x_i are set to either x_i or $\overline{x_i}$ depending on the phase.

Finally, the two networks are connected by inserting an inverter after the output of $\mathcal{N}_0$, and connecting the output of $\mathcal{N}_1$ and the output of the inverter with an AND gate. The primary input x_i is then set to v. The output of this AND gate has the functionality exactly the same as $(x_i = v)\chi^0_{z,0}\,|_{\forall u \in V: u=1}\,\overline{\chi^0_{z,0}\,|_{\forall u \in V: u=0}}$. One can easily create a SAT formula from this network. Note that the ordering constraint is implicitly imposed in the resulting network with the use of α and β variables. The same construction works for the other formula.

Since there is considerable similarity between $\mathcal{N}_1$ and $\mathcal{N}_0$, the resulting network can be simplified by sharing subcircuits. The two networks are only different in how the leaf χ variables

for x_i are constrained. Therefore, the portion of $\mathcal{N}_1$ that is not in the transitive fanout of the leaf χ variables for x_i is shared with $\mathcal{N}_0$. This reduces the size of the resulting network, thereby giving a smaller SAT formula.

To test strong falsity of paths, L is set to l_i first. As long as neither of the two formulas is satisfiable for $x_i = 0$ or $x_i = 1$, L is reduced and the satisfiability checks are repeated. Eventually either of the formulas becomes satisfiable for $x_i = v(v = 0, 1)$. It is then concluded that $\Pi(x_i, L)$ is strongly false for value v at x_i for the L next to the last one.

Let us see how the algorithm works on the circuit in Figure 7.1. The χ functions at the output g for values 0 and 1 are:

$$\begin{aligned}
\chi^0_{g,1} &= \chi^{-1}_{e,1}\chi^{-1}_{f,1} \\
&= (\chi^{-2}_{a,0}\chi^{-2}_{b,0})(\chi^{-2}_{d,1}+\chi^{-2}_{c,1}) \\
&= (\chi^{-2}_{a,0}\chi^{-2}_{b,0})(\chi^{-3}_{a,1}\chi^{-3}_{b,1}+\chi^{-2}_{c,1}) \\
\chi^0_{g,0} &= \chi^{-1}_{e,0}+\chi^{-1}_{f,0} \\
&= (\chi^{-2}_{a,1}+\chi^{-2}_{b,1})+(\chi^{-2}_{d,0}\chi^{-2}_{c,0}) \\
&= (\chi^{-2}_{a,1}+\chi^{-2}_{b,1})+(\chi^{-3}_{a,0}+\chi^{-3}_{b,0})\chi^{-2}_{c,0}.
\end{aligned}$$

Consider $\Pi(a, 3)$. The leaf χ variables corresponding to this path set are $\chi^{-3}_{a,0}$ and $\chi^{-3}_{a,1}$.

We first check whether $\Pi(a, 3)$ is strongly false for value 1 at a. If the χ functions at the output are sensitive to the leaf χ variable $\chi^{-3}_{a,1}$ under $a = 1$, the path set is not strongly false. The other leaf χ variable $\chi^{-3}_{a,0}$ is irrelevant since $\chi^{-3}_{a,0} = 0$ under $a = 1$. Consider $\chi^0_{g,1}$. If $a = 1$, $\chi^{-2}_{a,0} = 0$. Therefore, $\chi^0_{g,1} = 0$ independent of $\chi^{-3}_{a,1}$ although $\chi^{-3}_{a,1}$ is referred to in $\chi^0_{g,1}$. The other χ function at the output $\chi^0_{g,0}$ does not depend on $\chi^{-3}_{a,1}$ in the first place. Since both of the output χ functions are independent of the leaf χ variable $\chi^{-3}_{a,1}$, $\Pi(a, 3)$ is strongly false for value 1 at a, which coincides with the result in Section 7.2.

To test whether $\Pi(a, 3)$ is strongly false for value 0 at a, the same analysis needs to be done under $a = 0$ to see if the output χ functions are dependent on $\chi^{-3}_{a,0}$. $\chi^0_{g,1}$ is independent of $\chi^{-3}_{a,0}$. $\chi^0_{g,0}$ refers to $\chi^{-3}_{a,0}$ and requires further investigation.

Under $a = 0$, $\chi^0_{g,0}$ can be simplified as follows.

$$\chi^0_{g,0} = \chi^{-2}_{b,1}+(\chi^{-3}_{a,0}+\chi^{-3}_{b,0})\chi^{-2}_{c,0}.$$

Boolean parameters are now introduced as in Section 3.3.2.

$$\chi^{-2}_{b,1} = b\alpha^b$$

$$\begin{aligned}\chi^{-3}_{a,0} &= \bar{a}\beta^a \\ \chi^{-3}_{b,0} &= \bar{b}\beta^b \\ \chi^{-2}_{c,0} &= \bar{c}\beta^c\end{aligned}$$

To see if $\Pi(a,3)$ is strongly false for value 0 at a, it is enough to check whether $\chi^0_{g,0}$ is sensitive to the value of β^a. Recall that the condition $\beta^a = 1$ corresponds to the case where a arrives at $t = -3$ while the condition $\beta^a = 0$ corresponds to the case where a arrives later than $t = -3$. Whether $\chi^0_{g,0}$ depends on β^a can be checked by taking a Boolean difference of $\chi^0_{g,0}$ with respect to β^a. The Boolean difference is simplified to:

$$\chi^0_{g,0}\,|_{\beta^a=1}\,\overline{\chi^0_{g,0}\,|_{\beta^a=0}} = \bar{a}\bar{c}\beta^c(b\overline{\alpha^b} + \bar{b}\overline{\beta^b}).$$

Since the difference is satisfiable, the output χ function $\chi^0_{g,0}$is indeed sensitive to the leaf χ variable $\chi^{-3}_{a,0}$, and thus $\Pi(a,3)$ is not strongly false for value 0 at a. The Boolean difference is satisfiable under two cases. The first case is:

$$a = 0, b = 0, c = 0, \beta^b = 0, \beta^c = 1,$$

which corresponds to $(a,b,c) = (0,0,0)$, $arr(b) = \infty$ and $arr(c) = -2$. In other words, if b does not arrive and c arrives at $t = -2$ under the input vector $(0,0,0)$, the output gets stable if a arrives at $t = -3$, but does not if a arrives later. This shows that $\Pi(a,3)$ is responsible for the stable time of the output, and thus is not strongly false for value 0 at a. Note that this corresponds to delay tuple $(3,-\infty,2)$ under input vector 000 in the exact delay abstraction. The second case is:

$$a = 0, b = 1, c = 0, \alpha^b = 0, \beta^c = 1.$$

Although $b = 1$ this time, $\alpha^b = 0$ means that the arrival time of b is again set to ∞. Therefore, the arrival time condition is the same as the previous case. Notice that the exact delay abstraction has a corresponding delay tuple $(3,-\infty,2)$ under input vector 010.

Preliminary experimental results of this algorithm will be given in Section 8.4.

7.4 Relationship with Static Co-sensitization

Devadas *et al.* [DKM93] introduced static co-sensitization as a necessary condition for a path to be responsible for delay under the floating mode condition. This path sensitization condition is a purely Boolean condition, and independent of gate delays and arrival times at the inputs. If a path

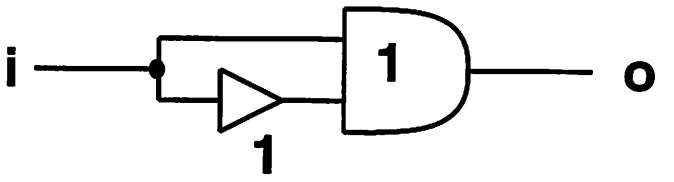

Figure 7.2: A Strongly False Path Can Be Statically Co-sensitizable

is not statically co-sensitizable, the path cannot be responsible for delay regardless of gate delays and arrival times. Thus, any path not statically co-sensitizable is strongly false. However, the converse is not true; a strongly false path may be statically co-sensitizable.

Consider a circuit in Figure 7.2. Both the AND gate and the buffer have delay of 1. Suppose $i = 0$. Both of the fanins of the AND gate have controlling values 0, which makes the output of the gate 0. The upper path and the lower path are both statically co-sensitizable. On the other hand, the upper path is not strongly false, but the lower path $\Pi(i, 2)$ is strongly false as we can see from the exact delay abstraction of this circuit D,

i	$d_{i\to o}$
0	(1)
1	(2).

Note that the lower path is strongly false, but statically co-sensitizable if $i = 0$. The reason why the notion of strong falsity can detect paths not responsible for delay more accurately than static co-sensitization is that gate delays are taken into account during the analysis while static co-sensitization ignores this factor. How much difference the two notions make on realistic circuits is yet to be studied.

Since any path not statically co-sensitizable is strongly false, non static co-sensitizability can be used to estimate strongly false paths conservatively.

Cheng and Chen [CC96] argued false paths in the context of delay-fault testing. Under the existence of path delay faults gate delays and wire delays cannot be bounded from above as in timing analysis. However, if a path is false under all possible delay assignments to gates and wires, a path delay fault has no effect on the performance of the circuit, and thus there is no need to consider the path. A path delay fault is said to be *f-redundant* if the path is false under all delay assignments to gates and wires. They showed that a path that is not *functionally sensitizable* under any input vector is f-redundant, where functional sensitization is defined as follows.

Definition 7.4 *A path $P = (g_0, \ldots, g_m)$ is* functional sensitizable *under an input vector* **x** *if for each $g_i (i = 0, \ldots, m-1)$ that has a non-controlling value of g_{i+1} all the side inputs of g_{i+1} have non-controlling value of g_{i+1}.*

This sensitization condition is in fact equivalent to static co-sensitization[2]. Thus, the claim by Cheng and Chen that a functionally unsensitizable path is f-redundant and thus false under any gate/wire delay assignment is simply a rephrase of a path not statically co-sensitizable being false under any delay assignment, which was known in [DKM93] already. Notice that arbitrary arrival time conditions can be realized by changing delays of wires after primary inputs.

Theorem 7.3 *A path $P = (g_0, \ldots, g_m)$ is statically co-sensitizable under an input vector* **x** *if and only if it is functional sensitizable under* **x**.

Proof

$\Rightarrow$ If P is statically co-sensitizable under **x**, for each $g_i (i = 1, \ldots, m)$ that has a controlled value of g_i, g_{i-1} has a controlling value of g_i. Therefore,

$$\forall i \in \{1, \ldots, m\}, g_i \text{ has a controlled value} \Rightarrow g_{i-1} \text{ has a controlling value of } g_i$$

This is equivalent to:

$$\forall i \in \{1, \ldots, m\}, g_{i-1} \text{ has a non-controlling value of } g_i \Rightarrow g_i \text{ has a non-controlled value.}$$

If g_i has a non-controlled value, all the fanins of g_i have non-controlling values of g_i. Therefore,

$\forall i \in \{1, \ldots, m\}$,

g_{i-1} has a non-controlling value of g_i

$\Rightarrow$ all the side inputs of g_i has a non-controlling value of g_i.

Hence P is functional sensitizable under **x**.

$\Leftarrow$ If P is functional sensitizable,

$\forall i \in \{0, \ldots, m-1\}$,

g_i has a non-controlling value of g_{i+1}

$\Rightarrow$ all the side inputs of g_{i+1} has a non-controlling value of g_{i+1}

$\Rightarrow$ g_{i+1} has a non-controlled value.

[2]Static co-sensitization was first published in [DKM91] before [CC96]. However, Cheng and Chen did not refer to static co-sensitization.

This is equivalent to:

$\forall i \in \{0,\ldots,m-1\}, g_{i+1}$ has a controlled value $\Rightarrow g_i$ has a controlling value of g_{i+1},

showing that P is statically co-sensitizable.

□

7.5 Conclusions

We have introduced a new class of false paths for combinational modules. These false paths, called strongly false paths, are the paths that are never responsible for delay under any arrival time condition. Under the assumption that arrival times at the inputs are unknown as in intellectual property blocks these are the only paths guaranteed to be false under any surrounding environment. After showing that strong falsity of a path can be determined by examining the exact delay abstraction of a module, we showed that the problem is reducible to a satisfiability problem, which makes it possible to check the strong falsity of a path set without computing the exact delay abstraction.

In Chapter 8 we will discuss the removal of strongly false paths from a combinational module without slowing down the module under any arrival time condition, i.e. the resulting module is a timing-safe replacement of the original.

Chapter 8

False Path Removal for Combinational Modules

Chapter 7 introduced a new class of false paths called strongly false paths for combinational modules. A path in a combinational module is said to be strongly false if it is false under any arrival time condition at primary inputs. Since the actual environment under which a combinational module is to be used is unknown, strongly false paths are the only paths that can be safely assumed to be false for a combinational module. Since they are never responsible for the stability of an output under any arrival time condition, it may be desirable if they can be structurally removed from the module by a circuit transformation. If such a transformation is possible, the resulting false-path-free module can be analyzed more accurately than the original module by topological timing analysis, which is much more efficient than functional timing analysis. Although this transformation is attractive, we do not want to slow down the original circuit by the transformation especially in the context of high-performance designs. Thus, the structural transformation also needs to guarantee that the resulting module $\mathcal{M}'$ is no slower than the original $\mathcal{M}$ under any arrival time condition. In this chapter we present an algorithm that removes strongly false paths from a combinational module $\mathcal{M}$ without increasing the delay of the module under any arrival time condition. $\mathcal{M}'$ is proved to be a timing-safe replacement of $\mathcal{M}$, i.e. $\mathcal{M}' \preceq_M \mathcal{M}$.

This chapter is organized as follows. Section 8.1 overviews the KMS algorithm. Although it was originally proposed as a redundancy removal algorithm that does not increase delay, it can be thought of as a procedure for removing long false paths. We argue why the direct use of the KMS algorithm is not appropriate in removing false paths from a combinational module. Section 8.2 illus-

trates the problems using examples. Section 8.3 presents an algorithm for removing strongly false paths from a combinational module and proves that the algorithm is guaranteed to give a timing-safe replacement of the original. Experimental results are given in Section 8.4. The chapter is concluded in Section 8.5.

8.1 The KMS Algorithm

Keutzer, Malik and Saldanha [KMS91] showed that redundancy is not necessary to reduce delay. The motivating example for the work was a carry-skip adder. This circuit has a single stuck-at redundancy, but the direct removal of the redundancy makes a long false path true thereby slowing down the circuit. The redundancy in the circuit is a by-product of making its longest topological path false to improve the performance. However, such a redundant circuit is problematic since the existence of the fault causes the circuit to slow down, but the fault is not detectable by conventional testing techniques.

A natural question is whether redundancy is necessary to reduce delay in general. They resolved this issue negatively by giving a constructive algorithm, commonly known as the KMS algorithm, which transforms a given redundant circuit to a functionally equivalent irredundant circuit with no penalty in its performance under a given arrival time condition at the primary inputs.

The KMS procedure [KMS91] takes 1) a gate-level redundant combinational circuit and 2) arrival time for each primary input, and returns a functionally equivalent irredundant circuit no slower than the original under the given arrival times. Suppose that the longest topological path is true in a given circuit. Then any stuck-at redundancies in the circuit can be removed in an arbitrary order since this never worsens the delay of the circuit; it is impossible to find a longer path in the circuit. This approach, however, does not work if the longest topological path is false since redundancy removal can make the false path true thereby slowing down the circuit. The core of the KMS algorithm consists in how to handle this case.

Given a circuit whose longest topological path is false, the KMS algorithm first checks if a multiple fanout node exists on the path. If there is no such node, the input edge of this path from the primary input is guaranteed to be both stuck-at-0 and stuck-at-1 redundant. Furthermore, removing either of these redundancies does not slow down the circuit since constant propagation from the edge never deteriorates the delay of the path.

Suppose there is a multiple fanout node on the path. Even if the path itself is still insensitive to the signal value of the input edge, it is possible that it is stuck-at-0 or stuck-at-1 irredundant

through other paths branching off from the path under analysis. Thus the simple constant propagation employed in the previous case is not appropriate. The KMS algorithm first finds the last multiple fanout node of the path and duplicates the circuit up to the node for all the other paths so that the path under analysis is isolated as a fanout-free path. Since this transformation reduces the multiple-fanout case to the previous case, the same approach above then applies.

This process is repeated as long as the longest topological path is false. Once the longest topological path becomes true, all the remaining redundancies are removed directly.

Since the original algorithm proposed in [KMS91] processed a single path at one time, it had a severe limitation in the size of circuits. Saldanha *et al.* [SBSV94] later resolved this complexity issue by proposing an algorithm which does not require explicit path enumeration[1].

The KMS algorithm removes long false paths from a given combinational circuit by a structural transformation as part of the algorithm. It guarantees that the longest topological path of the final circuit is true under the given arrival time condition. This property is desirable since the topological delay of the final circuit matches its exact delay; hence topological timing analysis gives better accuracy on the transformed circuit than the original circuit. Given a combinational circuit whose longest topological paths are false under a given arrival time condition, one can simply apply the KMS procedure to obtain a false-path-free circuit which is no slower than the original. The final circuit can then be used as a replacement of the original without the risk of slowing down the circuit. Since false paths have been removed, the circuit can be analyzed accurately even with topological analysis.

This approach, however, has fundamental limitations to be used for false path removal of combinational modules.

First, the KMS procedure takes an arrival time condition at the primary inputs of a circuit and works under this particular condition. Therefore, it is not directly applicable to a combinational module since the arrival times at the inputs are unknown. If a representative arrival time condition is chosen and the procedure is applied under the condition, the delay of the resulting circuit is not guaranteed once it is used under a different arrival time condition.

Second, redundancy removal performed as the final step of the KMS algorithm can increase delay even under the arrival time condition chosen for the analysis if the delay of the circuit is examined for each input vector separately. Keutzer *et al.* [KMS91] argued that straight-forward

[1] Chen *et al.* [CDC92] showed that the original KMS algorithm can be improved by removing false *subpaths* instead of false input-output paths. This results in less gate duplication and less area overhead. However, this algorithm still works on a subpath at one time, and thus inherits the complexity problem in the original KMS algorithm.

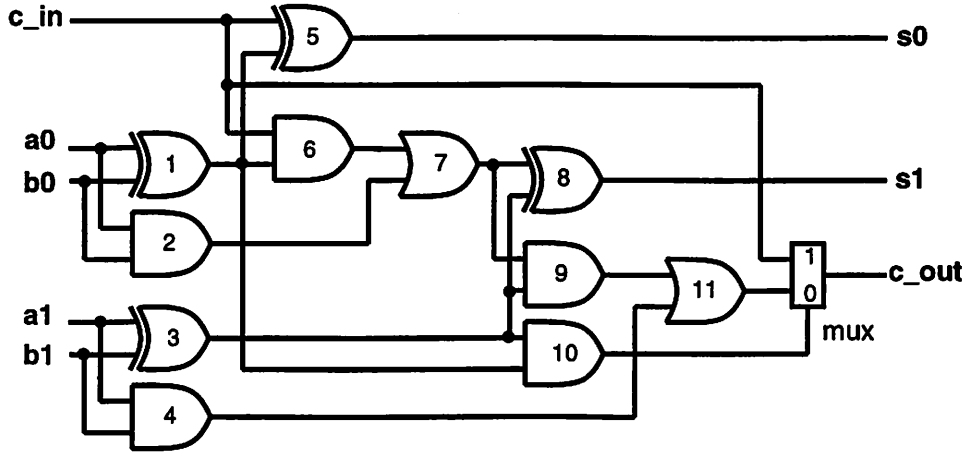

Figure 8.1: 2-bit Carry-Skip Adder

redundancy removal cannot slow down the circuit since the topological longest path is true after false path removal. In this argument the delay of a circuit is defined as the earliest time when *all* the primary outputs are stabilized for *all* input vectors under a given arrival time condition at the inputs. However, the delay of an output under an input vector can increase as the result of redundancy removal although it never increases so much as to increase the "delay" of the circuit. Under the delay definition of [KMS91] this local delay increase for an input vector does not cause the increase of the "delay". However, since there exists a surrounding environment of the module which can detect this delay increase, it should be thought of as a delay increase in the context of combinational modules.

8.2 Motivating Examples

This section shows why a simple-minded application of the KMS algorithm is not appropriate to remove false paths from combinational modules.

The first example is a carry-skip adder. This is the circuit that motivated the entire research on the KMS algorithm. Figure 8.1 shows a 2-bit carry-skip adder described in [KMS91]. We focus on the subcircuit computing the carry output. Assume a gate delay of 1 for the AND gate and the OR gate, and gate delays of 2 for the XOR gate and the MUX gate. The selector input of the multiplexor is stuck-at-0 redundant since under the existence of the fault, the circuit simply degenerates into a ripple-carry adder, which is functionally equivalent to the original circuit. The performance of the circuit, however, is deteriorated by the fault since the ripple-carry adder is slower than the carry-skip adder.

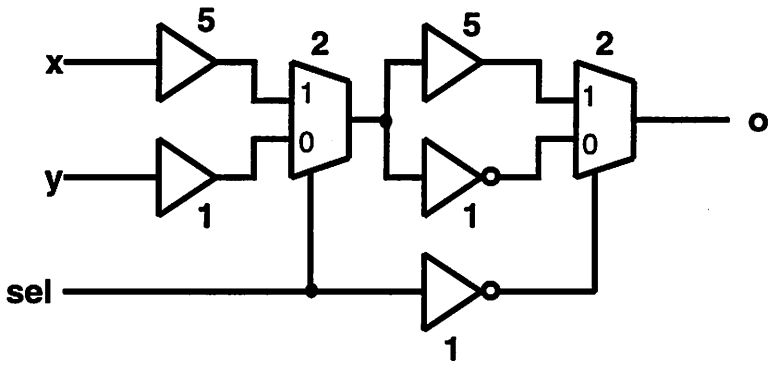

Figure 8.2: Circuit with Multiplexors

In [KMS91] this circuit is analyzed under the condition where the carry input arrives at $t = 5$ and all other inputs arrive at $t = 0$. In this particular situation the longest topological delay is 11 by the path of length 6 $(c_{in}, g_6, g_7, g_9, g_{11}, mux, c_{out})$. Since this longest path is false under the given arrival times, the KMS algorithm is invoked to remove the path.

If the resulting circuit is used under the same arrival time condition, it is guaranteed to be no slower than the original. However, under a different arrival time condition it is possible that the performance of the resulting circuit is worse than that of the original.

Saldanha [Sal91](page 69) applied the KMS algorithm to a cascaded carry-skip adder in such a way that each block is made irredundant by the KMS algorithm assuming that the carry input arrives later than the other inputs. The choice of the arrival times was done in an ad-hoc way. Apparently he replaced each block with a new irredundant block assuming that the new block is no slower than the original under any arrival time condition. This assumption is not correct.

Let us analyze the same circuit under different arrival times to see the problem. Assume that all the inputs arrive at $t = 0$. The topological longest paths are now the paths of length 8 from a_0 and b_0 to c_{out}[2]. These paths are true under the arrival time condition. Therefore if one simply follows the KMS procedure, any redundancy can be removed arbitrarily without slowing down the circuit, which results in a ripple carry adder. Notice that although the effective delay from the carry input to the carry output has increased in this transformation, the delay of 8 from a_0 and b_0 still determines the circuit performance. Thus the transformation is valid under the given arrival times. Now assume that the resulting circuit is used where the carry input arrives at $t = 5$ and the other inputs arrive at $t = 0$. Obviously we now observe a larger delay of 11 instead of 8. This example clearly shows

[2]The long path from c_{in} considered in the previous case has length 6 and is no more the longest.

that the delay non-increasing property of the KMS algorithm is only guaranteed for a given arrival time condition at primary inputs. In order to remove false paths from a combinational module we are interested in a more robust algorithm which never slows down the circuit under *any* arrival time condition.

Consider another example shown in Figure 8.2. The number attached to each gate is the delay of the gate. Assume that x and *sel* arrive at $t = 0$ while y arrives at $t = 10$. The circuit is fully irredundant in terms of single stuck-at faults. The longest topological path is the one of length 10 from y to the output via the upper path after the first multiplexor, giving the delay of $20 = 10 + 10$. Since this path is true, the KMS algorithm does nothing to this circuit. Given different arrival times, however, the path from x to the output through the two upper paths can be false. For example, when all the primary inputs arrive at $t = 0$, the path is the longest and false. From the viewpoint of false path removal, this implies that an arbitrary choice of arrival times is not enough to remove false paths fully from combinational modules.

Finally consider the carry-skip adder in Figure 8.1 again. Assume that the carry input arrives at $t = 5$ and all the other inputs arrive at $t = 0$. The removal of the long false path from the carry input to the carry output yields the circuit $\mathcal{M}_0$ in Figure 8.3. For the sake of simplicity only the fanin cone of the carry output is shown. Each of the fanin edges of g_2 is stuck-at-1 redundant. If one follows the KMS procedure, any of these redundancies can be removed without slowing down the circuit. If the a_0 edge is replaced with a constant 1, the circuit $\mathcal{M}_{0-RR}$ in Figure 8.4 is obtained. Now that the b_0 edge is not redundant any more, this is the final result of the KMS procedure.

We are now ready to show that this redundancy removal in fact increases the delay of the circuit even under the arrival time condition analyzed, once the delay is determined for each input vector separately. Consider the input vector $(a_0, a_1, b_0, b_1, c_{in}) = (0,0,1,0,0)$. In the circuit before the redundancy removal, the path $(a_0, g_2, g_9, g_{11}, mux, c_{out})$ is the longest true path. Therefore the delay under this vector is 5. On the other hand, in the circuit after the redundancy removal, the path $(\{a_1, b_1\}, g_3, g_9, g_{11}, mux, c_{out})$, which was false before the redundancy removal, becomes true and gives delay $6 (> 5)$. Notice that there exists an input vector that sensitizes the longest topological path from c_{in} to c_{out} of length 7 in both of the circuits. Therefore, the redundancy removal is safe under the traditional definition of delay. However, if we need to preserve the performance of the circuit under any surrounding environment, redundancy removal can worsen the delay.

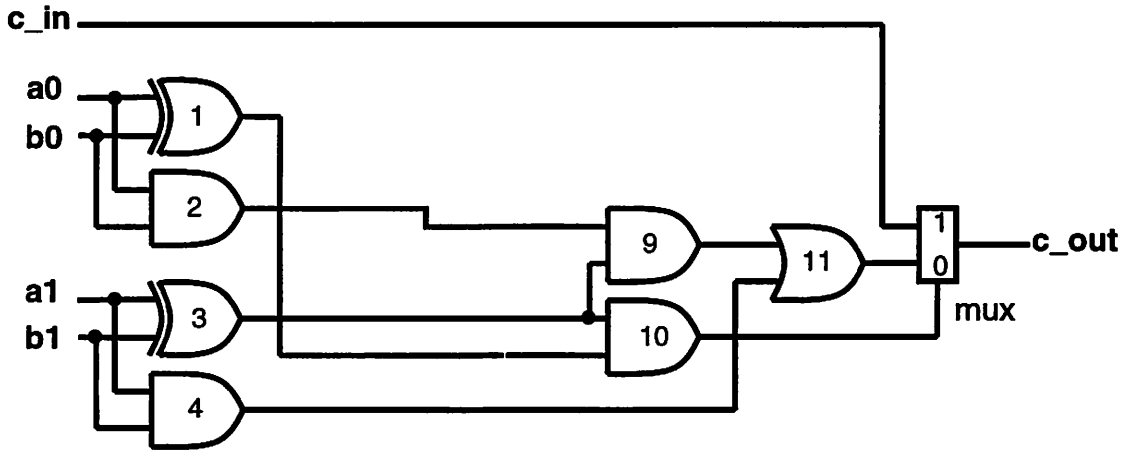

Figure 8.3: 2-bit Carry-Skip Adder $\mathcal{M}_0$ before Redundancy Removal

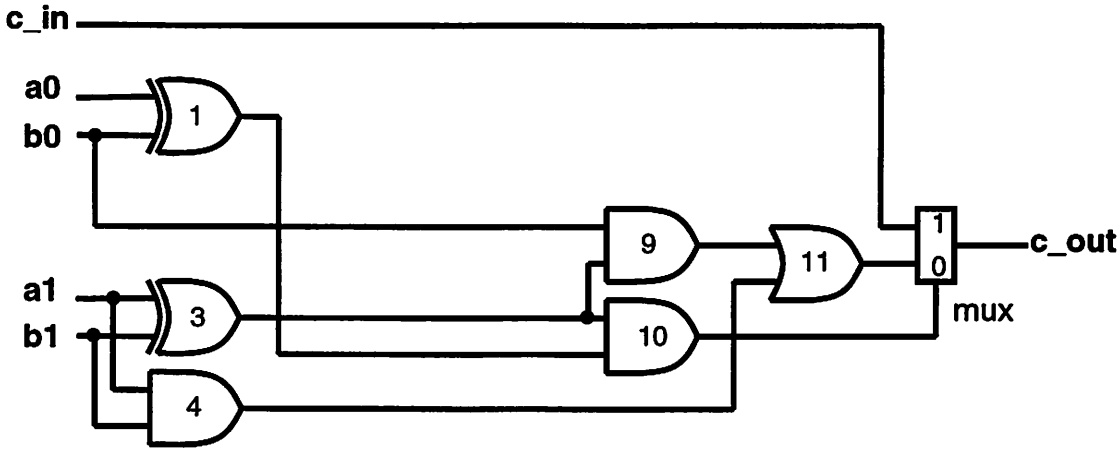

Figure 8.4: 2-bit Carry-Skip Adder $\mathcal{M}_{0\_RR}$ after Redundancy Removal

8.3 False Path Removal of Combinational Modules

Motivated by the examples in the previous section, we describe how to remove false paths safely from combinational modules. Our goal is to design an algorithm which takes a gate-level circuit $\mathcal{M}$ and returns a false-path-free circuit $\mathcal{M}'$ such that $\mathcal{M}' \preceq_M \mathcal{M}$. Section 8.2 showed that a simple-minded application of the KMS algorithm is not enough for our purpose.

The first problem is that the KMS algorithm only removes long false paths under given arrival times. Because of this strategy, false paths not critical under the situation remain in the circuit. To make matters worse, those paths can become true long paths after redundancy removal thereby slowing down the circuit under a different arrival time condition. This was illustrated in the carry-skip adder and the multiplexer-based circuit in Section 8.2. Moreover, since false paths removed by the KMS algorithm are not necessarily strongly false, even false path removal alone can slow down the circuit. We will show an example of this later.

To alleviate this problem all long strongly false paths are removed from each input by a circuit transformation. As a result, the topological longest path from any input is sensitizable under some input vector and some arrival time condition.

The second problem is that the final redundancy removal in the KMS algorithm can slow down a circuit if the delay of the circuit is computed for each primary input vector. This is unacceptable for combinational modules since there exists a surrounding environment whose performance is deteriorated by this delay increase. Therefore the redundancy removal is dropped intentionally.

8.3.1 Algorithm for False Path Removal

We first illustrate the key idea of the algorithm using an example. Consider again the circuit $\mathcal{M}$ in Figure 7.1. We have already shown that $\Pi(a,3) = \{P_a\}$ is strongly false for 1 at a in Section 7.2. This means that if $a = 1$, this path is never responsible for the signal stability at the output under any arrival time condition. Therefore, the input edge of the path can be safely replaced with a constant 0 without slowing down the circuit. Notice that the path is fanout-free and this modification cannot adversely affect the other paths. Propagating this constant through the circuit is also a safe operation. We already saw in Section 6.2 that the resulting circuit $\mathcal{M}_0$ in Figure 6.2 is a timing-safe replacement of $\mathcal{M}$.

If the original KMS algorithm is applied to $\mathcal{M}$ in Figure 7.1 under $(arr(a), arr(b), arr(c)) = (1,0,1)$, P_a is identified as false. One can then choose either constant 0 or 1 to replace the input edge of P_a with. If a constant 1 is chosen, the circuit $\mathcal{M}_1$ in Figure 6.3 is obtained, which is not a timing-

safe replacement of $\mathcal{M}$ as discussed in Section 6.2. This example shows that strongly false paths are the appropriate paths to be removed for combinational modules. Keutzer *et al.* [KMS91] only suggest that one pick the constant that gives better simplification of the circuit. However, they use this choice only as an optimization.

We are ready to prove the correctness of the transformation formally. We need a couple of definitions.

Definition 8.1 *The set of all paths beginning at an primary input edge e and ending at a primary output is called the* path set of *e.*

Definition 8.2 *An* L-path disjoint circuit[3] with respect to primary input x_i *is a circuit where the path set of any primary input edge from* x_i *consists of either paths of length* $\geq L$ *or paths of length* $< L$.

Given a combinational module, one can always construct a module that is L-path disjoint with respect to x_i by fully preserving the original functional and timing properties. This detail can be found in [SBSV94][4].

Let $\mathcal{M}$ be a single-output combinational module whose primary inputs are $x_1, \ldots, x_n$. Let $\mathcal{M}'$ be an L-path disjoint circuit with respect to x_i that is obtained from $\mathcal{M}$.

Lemma 8.1 *If path set* $\Pi(x_i, L)$ *is strongly false for value* $v (v = 0, 1)$ *at* x_i *in* $\mathcal{M}$, *the input edge of the path set is stuck-at-*$\bar{v}$ *redundant in* $\mathcal{M}'$.

Proof Suppose it is not stuck-at-$\bar{v}$ redundant for contradiction. Then there exists an input vector $\mathbf{x}$ where $x_i = v$ such that the output value of $\mathcal{M}'$ is different between the fault-free circuit and the faulty circuit. Now consider the arrival time condition $arr(x_i) = -L + \varepsilon$ and $arr(x_j) = -l_j$ if $j \neq i$, where l_j is the longest topological path length from x_j to the output. Suppose we apply the same vector $\mathbf{x}$ to $\mathcal{M}$ under this arrival time condition. The fact that $\mathbf{x}$ is a test pattern directly implies that the output of $\mathcal{M}$ is different between $t = 0$ and $t = \infty$. Thus, the output of $\mathcal{M}$ is not stable by $t = 0$. Since all the primary inputs except x_i arrive early enough, the output is not stabilized because of the delay in x_i. This implies that some path from x_i of length $\geq L$ is responsible for determining the output under $\mathbf{x}$. This contradicts that $\Pi(x_i, L)$ is strongly false for value v at x_i. □

[3]This definition is a variation of L-path disjoint circuits introduced in [SBSV94].

[4]The procedure in [SBSV94] is applicable by assuming that x_i arrives at $t = 0$ and all the other inputs arrive at $t = -\infty$. All the paths from the other inputs are ignored effectively this way.

Theorem 8.1 *Suppose path set $\Pi(x_i, L)$ is strongly false for value v at x_i in $\mathcal{M}$. Let $\mathcal{M}'$ be an L-path disjoint circuit with respect to x_i obtained from $\mathcal{M}$. Let $\mathcal{M}''$ be the circuit obtained from $\mathcal{M}'$ by substituting $\bar{v}$ for the input edge of $\Pi(x_i, L)$ in $\mathcal{M}'$. Finally let $\tilde{\mathcal{M}}$ be the circuit obtained from $\mathcal{M}''$ by performing a constant propagation of $\bar{v}$. Then $\tilde{\mathcal{M}} \preceq_M \mathcal{M}$.*

Proof From Lemma 8.1 this constant substitution does not change the functionality of the circuit. Therefore we have only to check if the performance of the circuit is not deteriorated.

Notice that the χ functions at the output for $\mathcal{M}$ and those for $\mathcal{M}'$ are exactly the same since the circuit transformation for L-path disjoint property does not change the timing characteristic of the circuit. Since $\Pi(x_i, L)$ is strongly false for value v at x_i in $\mathcal{M}$, χ functions at the outputs $\chi^0_{z,1}$ and $\chi^0_{z,0}$ are independent of any leaf χ variable $\chi^{-L'}_{x_i,v}$ where $L' \geq L$. Therefore, setting $\chi^{-L'}_{x_i,v}$ to 0 does not worsen the signal stability of the output under any arrival time condition. This exactly corresponds to the circuit transformation we applied to get $\mathcal{M}''$ from $\mathcal{M}'$. Notice that if $x_i = \bar{v}$, this transformation has no effect since the input edge gets the same value as before.

Finally, the effect of the constant propagation is the same as assuming that the value of the input edge is available at $t = -\infty$. Since the analysis is based on the XBD0 model, the monotone speedup property guarantees that this never worsens the stability of the χ functions at the output. Hence $\tilde{\mathcal{M}} \preceq_M \mathcal{M}$. □

Based on Theorem 8.1 one can design a procedure that takes a single-output combinational module and removes strongly false paths to create a timing-safe replacement module where the longest topological path from any input is strongly false neither for 0 nor for 1 at the input. The algorithm examines primary inputs one by one by checking whether the longest topological paths from a primary input are strongly false for either 0 or 1. If either is true, the circuit is modified based on Theorem 8.1. This process is repeated until no change is observed. To handle a combinational module with multiple outputs the same procedure is applied to the transitive fanin cone of each primary output separately and the resulting circuits are merged into a single circuit by sharing isomorphic subcircuits. This step is guaranteed not to change the timing characteristics of the circuits[5].

8.3.2 Examples

Figure 8.5 shows the transitive fanin of the carry input of the 2-input carry-skip adder in Figure 8.1. Assume again a gate delay of 1 for the AND gate and the OR gate and gate delays of 2

[5]As in [KMS91] we assume that gate delay is independent of loads.

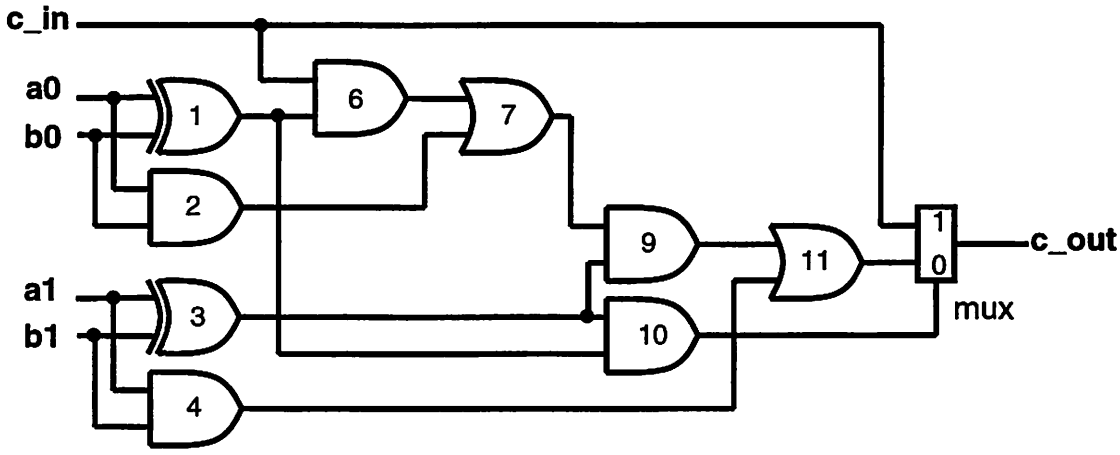

Figure 8.5: 2-bit Carry-Skip Adder $\mathcal{M}$

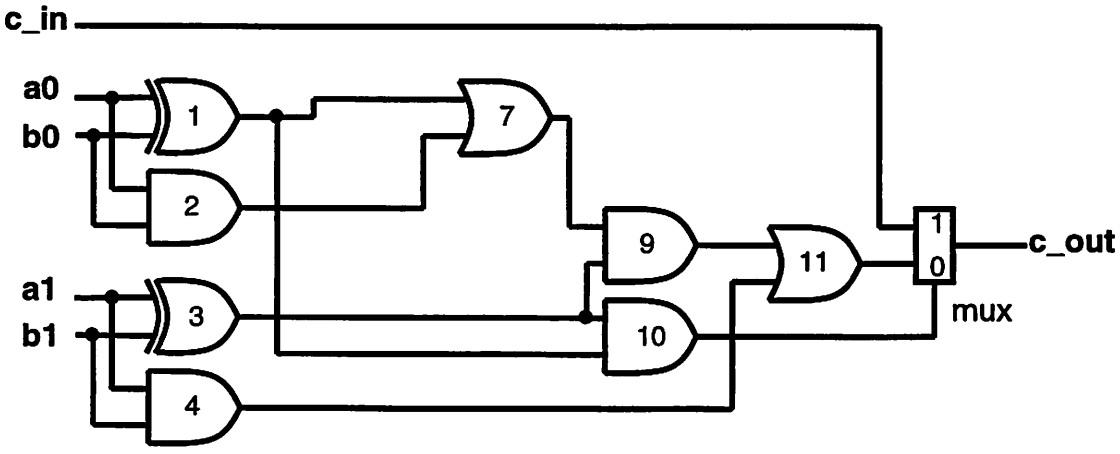

Figure 8.6: 2-bit Carry-Skip Adder $\mathcal{M}_1$ after Propagating Constant 1 from c_{in}

for the XOR gate and the MUX gate. The analysis of strongly false paths on this circuit indicates that path set $\Pi(c_{in}, 6)$ is strongly false for value 1 at c_{in}. Therefore, one can safely assert a constant 0 at the input edge of the long path of length 6 from the carry input to the carry output. Note that the circuit is already L-path disjoint with respect to c_{in} for $L = 6$. Figure 8.3 shows the resulting circuit $\mathcal{M}_0$. The exact delay abstractions D and D_0 of the two circuits $\mathcal{M}$ and $\mathcal{M}_0$ are shown in Tables 8.1 and 8.2 respectively. Since $D_0 \preceq D$, $\mathcal{M}_0 \preceq_M \mathcal{M}$, i.e. $\mathcal{M}_0$ is indeed a timing-safe replacement of $\mathcal{M}$.

In Section 8.2 we showed that redundancy removal on $\mathcal{M}_0$ does not preserve the timing-safe replacement property. The exact delay abstraction D_{0-RR} of $\mathcal{M}_{0-RR}$ is shown in Table 8.3. Since $D_{0-RR} \not\preceq D_0$, $\mathcal{M}_{0-RR}$ is not a timing-safe replacement of $\mathcal{M}_0$.

To see this consider the input vector $(a_0, a_1, b_0, b_1, c_{in}) = (0,0,1,0,0)$. D_0 has a delay tuple $(5,5,-\infty,5,-\infty)$ under the vector. D_{0-RR} only has a delay tuple $(-\infty,6,-\infty,6,-\infty)$. Since $(-\infty,6,-\infty,6,-\infty) \not\trianglelefteq (5,5,-\infty,5,-\infty)$, $D_{0-RR} \not\preceq D_0$. Thus, $\mathcal{M}_{0-RR} \not\preceq_M \mathcal{M}_0$.

$\mathcal{M}_{0-RR}$ is not a timing-safe replacement of the original circuit $\mathcal{M}$ either. For example, under input vector $(a_0, a_1, b_0, b_1, c_{in}) = (0,1,1,0,0)$ D has a delay tuple $(6,-\infty,-\infty,4,6)$. The only delay tuple in D_{0-RR} under the vector is $(5,5,5,5,2)$. Since $(5,5,5,5,2) \not\trianglelefteq (6,-\infty,-\infty,4,6)$, $\mathcal{M}_{0-RR} \not\preceq_M \mathcal{M}$.

In the original KMS algorithm, this circuit was analyzed under the assumption that c_{in} arrives at $t = 5$ and all the other inputs arrive at $t = 0$. Under this condition the longest path from the carry input to the carry output is false and it was argued that the input edge of the path can be replaced with either 0 or 1. However, since $\Pi(c_{in}, 6)$ is not strongly false for value 0, the substitution of 1 at the input edge of c_{in} does not give a timing-safe replacement. The resulting circuit $\mathcal{M}_1$ is shown in Figure 8.6. The exact delay abstraction of the circuit is shown in Table 8.4. Under the input vector $(a_0, a_1, b_0, b_1, c_{in}) = (1,0,0,0,0)$ D has a delay tuple $(-\infty,-\infty,6,4,6)$. However, D_1 has only $(-\infty,6,-\infty,6,-\infty) \not\trianglelefteq (-\infty,-\infty,6,4,6)$. Thus, $D_1 \not\preceq D$ and $\mathcal{M}_1 \not\preceq_M \mathcal{M}$. This shows that the removal of conventional false paths under a given arrival time condition is too aggressive[6].

Consider the circuit with multiplexors in Figure 8.2. The topological longest paths from x, y and *sel* to the output are 14, 10 and 9 respectively. The false path analysis of this circuit shows that the path set $\Pi(x, 14)$ is strongly false for value 1 at x. A circuit duplication and a constant propagation

[6]Note that the original KMS algorithm [KMS91] removes statically non-sensitizable paths unlike the improved algorithm [SBSV94] where non-viable paths are removed. A statically sensitizable path is viable, but the converse is not true, Therefore, the original KMS algorithm is even more aggressive than just removing false paths, and thus has the same problem of violating the timing-safe replacement property.

Although the constant propagation of value 1 from c_{in} is not safe in terms of timing-safe replaceability, it does not increase the delay of the circuit under the same arrival time condition.

$a_0a_1b_0b_1c_{in}$	$d_{a_0\to c_{out}}\, d_{a_1\to c_{out}}\, d_{b_0\to c_{out}}\, d_{b_1\to c_{out}}\, d_{c_0\to c_{out}}$
00000	$\{(-\infty,4,6,-\infty,6)(8,-\infty,8,4,-\infty)(6,-\infty,-\infty,4,6)(8,4,8,-\infty,-\infty)$ $(-\infty,-\infty,6,4,6)(6,4,-\infty,-\infty,6)(-\infty,6,-\infty,6,-\infty)\}$
00001	$\{(8,4,8,-\infty,-\infty)(8,-\infty,8,4,-\infty)(-\infty,6,-\infty,6,-\infty)\}$
00010	$\{(-\infty,4,6,-\infty,6)(6,4,-\infty,-\infty,6)(8,4,8,-\infty,-\infty)\}$
00011	$\{(8,4,8,-\infty,-\infty)\}$
00100	$\{(6,-\infty,-\infty,4,6)(-\infty,6,-\infty,6,-\infty)(6,4,-\infty,-\infty,6)\}$
00101	$\{(-\infty,6,-\infty,6,-\infty)\}$
00110	$\{(5,5,5,5,2)(6,4,-\infty,-\infty,6)\}$
00111	$\{(5,5,5,5,2)\}$
01000	$\{(6,-\infty,-\infty,4,6)(8,-\infty,8,4,-\infty)(-\infty,-\infty,6,4,6)\}$
01001	$\{(8,-\infty,8,4,-\infty)\}$
01010	$\{(-\infty,5,-\infty,5,-\infty)(5,4,5,4,-\infty)\}$
01011	$\{(5,4,5,4,-\infty)(-\infty,5,-\infty,5,-\infty)\}$
01100	$\{(6,-\infty,-\infty,4,6)(5,5,5,5,2)\}$
01101	$\{(5,5,5,5,2)\}$
01110	$\{(-\infty,5,-\infty,5,-\infty)\}$
01111	$\{(-\infty,5,-\infty,5,-\infty)\}$
10000	$\{(-\infty,-\infty,6,4,6)(-\infty,6,-\infty,6,-\infty)(-\infty,4,6,-\infty,6)\}$
10001	$\{(-\infty,6,-\infty,6,-\infty)\}$
10010	$\{(5,5,5,5,2)(-\infty,4,6,-\infty,6)\}$
10011	$\{(5,5,5,5,2)\}$
10100	$\{(-\infty,6,-\infty,6,-\infty)\}$
10101	$\{(-\infty,6,-\infty,6,-\infty)\}$
10110	$\{(6,6,6,6,-\infty)\}$
10111	$\{(6,6,6,6,-\infty)\}$
11000	$\{(-\infty,-\infty,6,4,6)(5,5,5,5,2)\}$
11001	$\{(5,5,5,5,2)\}$
11010	$\{(-\infty,5,-\infty,5,-\infty)\}$
11011	$\{(-\infty,5,-\infty,5,-\infty)\}$
11100	$\{(6,6,6,6,-\infty)\}$
11101	$\{(6,6,6,6,-\infty)\}$
11110	$\{(5,4,5,4,-\infty)(-\infty,5,-\infty,5,-\infty)\}$
11111	$\{(-\infty,5,-\infty,5,-\infty)(5,4,5,4,-\infty)\}$

Table 8.1: Exact Delay Abstraction D of $\mathcal{M}$

$a_0a_1b_0b_1c_{in}$	$d_{a_0\to c_{out}}\, d_{a_1\to c_{out}}\, d_{b_0\to c_{out}}\, d_{b_1\to c_{out}}\, d_{c_0\to c_{out}}$
00000	$\{(5,-\infty,5,4,-\infty)(5,4,5,-\infty,-\infty)(-\infty,4,5,-\infty,2)$ $(5,4,-\infty,-\infty,2)(-\infty,6,-\infty,6,-\infty)(-\infty,5,5,5,-\infty)$ $(5,5,-\infty,5,-\infty)(-\infty,-\infty,5,4,2)(5,-\infty,-\infty,4,2)\}$
00001	$\{(5,5,-\infty,5,-\infty)(-\infty,5,5,5,-\infty)(5,4,5,-\infty,-\infty)$ $(-\infty,6,-\infty,6,-\infty)(5,-\infty,5,4,-\infty)\}$
00010	$\{(5,4,5,-\infty,-\infty)(5,4,-\infty,-\infty,2)(-\infty,4,5,-\infty,2)\}$
00011	$\{(5,4,5,-\infty,-\infty)\}$
00100	$\{(-\infty,6,-\infty,6,-\infty)(5,-\infty,-\infty,4,2)(5,5,-\infty,5,-\infty)(5,4,-\infty,-\infty,2)\}$
00101	$\{(5,5,-\infty,5,-\infty)(-\infty,6,-\infty,6,-\infty)\}$
00110	$\{(5,4,-\infty,-\infty,2)\}$
00111	$\{(5,5,5,5,2)\}$
01000	$\{(5,-\infty,5,4,-\infty)(5,-\infty,-\infty,4,2)(-\infty,-\infty,5,4,2)\}$
01001	$\{(5,-\infty,5,4,-\infty)\}$
01010	$\{(5,4,5,4,-\infty)(-\infty,5,-\infty,5,-\infty)\}$
01011	$\{(5,4,5,4,-\infty)(-\infty,5,-\infty,5,-\infty)\}$
01100	$\{(5,-\infty,-\infty,4,2)\}$
01101	$\{(5,5,5,5,2)\}$
01110	$\{(-\infty,5,-\infty,5,-\infty)\}$
01111	$\{(-\infty,5,-\infty,5,-\infty)\}$
10000	$\{(-\infty,-\infty,5,4,2)(-\infty,5,5,5,-\infty)(-\infty,4,5,-\infty,2)(-\infty,6,-\infty,6,-\infty)\}$
10001	$\{(-\infty,5,5,5,-\infty)(-\infty,6,-\infty,6,-\infty)\}$
10010	$\{(-\infty,4,5,-\infty,2)\}$
10011	$\{(5,5,5,5,2)\}$
10100	$\{(-\infty,6,-\infty,6,-\infty)\}$
10101	$\{(-\infty,6,-\infty,6,-\infty)\}$
10110	$\{(5,6,5,6,-\infty)\}$
10111	$\{(5,6,5,6,-\infty)\}$
11000	$\{(-\infty,-\infty,5,4,2)\}$
11001	$\{(5,5,5,5,2)\}$
11010	$\{(-\infty,5,-\infty,5,-\infty)\}$
11011	$\{(-\infty,5,-\infty,5,-\infty)\}$
11100	$\{(5,6,5,6,-\infty)\}$
11101	$\{(5,6,5,6,-\infty)\}$
11110	$\{(5,4,5,4,-\infty)(-\infty,5,-\infty,5,-\infty)\}$
11111	$\{(-\infty,5,-\infty,5,-\infty)(5,4,5,4,-\infty)\}$

Table 8.2: Exact Delay Abstraction D_0 of $\mathcal{M}_0$

$a_0a_1b_0b_1c_{in}$	$d_{a_0\to c_{out}}\, d_{a_1\to c_{out}}\, d_{b_0\to c_{out}}\, d_{b_1\to c_{out}}\, d_{c_0\to c_{out}}$
00000	$\{(-\infty,6,-\infty,6,-\infty)(-\infty,-\infty,4,4,2)(-\infty,4,4,-\infty,2)$
	$(5,-\infty,5,4,-\infty)(-\infty,5,4,5,-\infty)(5,4,5,-\infty,-\infty)\}$
00001	$\{(5,4,5,-\infty,-\infty)(-\infty,5,4,5,-\infty)(-\infty,6,-\infty,6,-\infty)(5,-\infty,5,4,-\infty)\}$
00010	$\{(5,4,5,-\infty,-\infty)(-\infty,4,4,-\infty,2)\}$
00011	$\{(5,4,5,-\infty,-\infty)\}$
00100	$\{(-\infty,6,-\infty,6,-\infty)\}$
00101	$\{(-\infty,6,-\infty,6,-\infty)\}$
00110	$\{(5,5,5,5,2)\}$
00111	$\{(5,5,5,5,2)\}$
01000	$\{(5,-\infty,5,4,-\infty)(-\infty,-\infty,4,4,2)\}$
01001	$\{(5,-\infty,5,4,-\infty)\}$
01010	$\{(5,4,5,4,-\infty)(-\infty,5,-\infty,5,-\infty)\}$
01011	$\{(5,4,5,4,-\infty)(-\infty,5,-\infty,5,-\infty)\}$
01100	$\{(5,5,5,5,2)\}$
01101	$\{(5,5,5,5,2)\}$
01110	$\{(-\infty,5,-\infty,5,-\infty)\}$
01111	$\{(-\infty,5,-\infty,5,-\infty)\}$
10000	$\{(-\infty,6,-\infty,6,-\infty)(-\infty,4,4,-\infty,2)(-\infty,5,4,5,-\infty)(-\infty,-\infty,4,4,2)\}$
10001	$\{(-\infty,5,4,5,-\infty)(-\infty,6,-\infty,6,-\infty)\}$
10010	$\{(-\infty,4,4,-\infty,2)\}$
10011	$\{(5,5,5,5,2)\}$
10100	$\{(-\infty,6,-\infty,6,-\infty)\}$
10101	$\{(-\infty,6,-\infty,6,-\infty)\}$
10110	$\{(5,6,5,6,-\infty)\}$
10111	$\{(5,6,5,6,-\infty)\}$
11000	$\{(-\infty,-\infty,4,4,2)\}$
11001	$\{(5,5,5,5,2)\}$
11010	$\{(-\infty,5,-\infty,5,-\infty)\}$
11011	$\{(-\infty,5,-\infty,5,-\infty)\}$
11100	$\{(5,6,5,6,-\infty)\}$
11101	$\{(5,6,5,6,-\infty)\}$
11110	$\{(-\infty,5,-\infty,5,-\infty)(5,4,5,4,-\infty)\}$
11111	$\{(5,4,5,4,-\infty)(-\infty,5,-\infty,5,-\infty)\}$

Table 8.3: Exact Delay Abstraction D_{0-RR} of $\mathcal{M}_{0-RR}$

$a_0a_1b_0b_1c_{in}$	$d_{a_0\to c_{out}}\, d_{a_1\to c_{out}}\, d_{b_0\to c_{out}}\, d_{b_1\to c_{out}}\, d_{c_0\to c_{out}}$
00000	$\{(7,-\infty,7,4,-\infty)(-\infty,6,-\infty,6,-\infty)(7,4,7,-\infty,-\infty)\}$
00001	$\{(7,4,7,-\infty,-\infty)(-\infty,6,-\infty,6,-\infty)(7,-\infty,7,4,-\infty)\}$
00010	$\{(7,4,7,-\infty,-\infty)\}$
00011	$\{(7,4,7,-\infty,-\infty)\}$
00100	$\{(-\infty,6,-\infty,6,-\infty)\}$
00101	$\{(-\infty,6,-\infty,6,-\infty)\}$
00110	$\{(5,5,5,5,2)\}$
00111	$\{(5,5,5,5,2)\}$
01000	$\{(7,-\infty,7,4,-\infty)\}$
01001	$\{(7,-\infty,7,4,-\infty)\}$
01010	$\{(5,4,5,4,-\infty)(-\infty,5,-\infty,5,-\infty)\}$
01011	$\{(5,4,5,4,-\infty)(-\infty,5,-\infty,5,-\infty)\}$
01100	$\{(5,5,5,5,2)\}$
01101	$\{(5,5,5,5,2)\}$
01110	$\{(-\infty,5,-\infty,5,-\infty)\}$
01111	$\{(-\infty,5,-\infty,5,-\infty)\}$
10000	$\{(-\infty,6,-\infty,6,-\infty)\}$
10001	$\{(-\infty,6,-\infty,6,-\infty)\}$
10010	$\{(5,5,5,5,2)\}$
10011	$\{(5,5,5,5,2)\}$
10100	$\{(-\infty,6,-\infty,6,-\infty)\}$
10101	$\{(-\infty,6,-\infty,6,-\infty)\}$
10110	$\{(6,6,6,6,-\infty)\}$
10111	$\{(6,6,6,6,-\infty)\}$
11000	$\{(5,5,5,5,2)\}$
11001	$\{(5,5,5,5,2)\}$
11010	$\{(-\infty,5,-\infty,5,-\infty)\}$
11011	$\{(-\infty,5,-\infty,5,-\infty)\}$
11100	$\{(6,6,6,6,-\infty)\}$
11101	$\{(6,6,6,6,-\infty)\}$
11110	$\{(-\infty,5,-\infty,5,-\infty)(5,4,5,4,-\infty)\}$
11111	$\{(-\infty,5,-\infty,5,-\infty)(5,4,5,4,-\infty)\}$

Table 8.4: Exact Delay Abstraction D_1 of $\mathcal{M}_1$

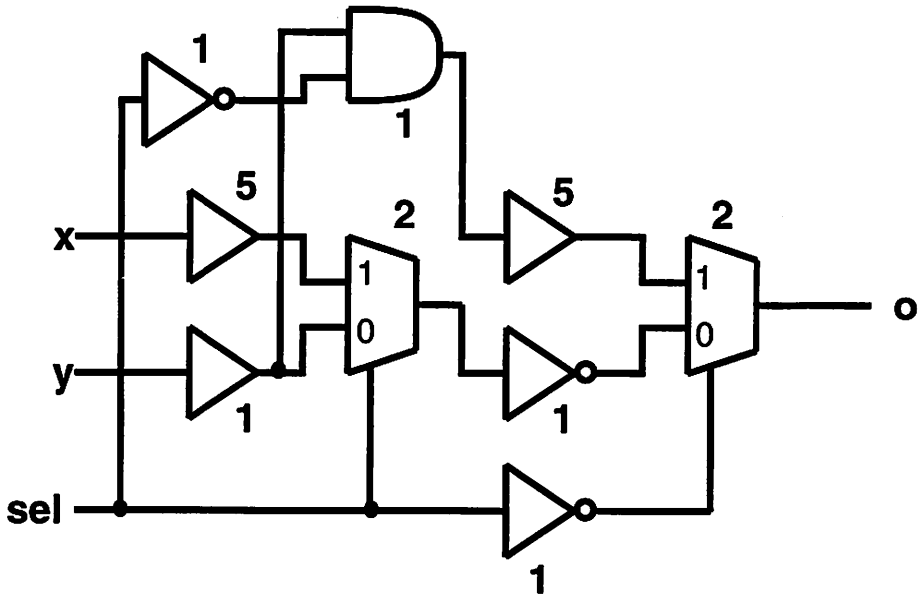

Figure 8.7: False-path-free Circuit with Multiplexors

gives a circuit shown in Figure 8.7[7].

8.4 Experimental Results

We implemented on top of SIS a procedure to check if a path set is strongly false for a value (0 or 1) at the corresponding primary input, and to remove such a path set structurally. The procedure determines the strong falsity of a path set without constructing the delay abstraction of a given module explicitly, and thus is applicable to large networks. Refer to Section 7.3 for the details of the algorithm. The removal of a false path set is a simple structural transformation, and thus takes negligible time compared with strong falsity checking. We only summarize the result of a representative circuit, the largest primary output cone of C7552. The cone has 194 primary inputs and 1096 gates. Recall that the strong falsity of a path set is defined for a single-output network. For each primary input the strong falsity of the longest paths from the input was tested for both values 0 and 1, and the paths were removed if they are strongly false. We found out that for 8 out of 194 primary inputs the longest paths from each primary input are strongly false for either value 1 or 0 at the input. A strong

[7]Although the circuit can be further simplified by logic sharing, we keep the left multiplexor as it is so that the correspondence between the old and the new circuit is clear.

falsity check for a primary input took 37.5 seconds in the average on DEC AlphaServer 8400 5/625[8]. These strongly false paths were then removed by the proposed procedure. The resulting circuit has 1216 nodes, only 11% area increase from the original.

8.5 Conclusions

We have discussed how false paths in a combinational module can be removed without slowing down the circuit under *any* arrival time condition. We have shown by examples that a simple application of the KMS algorithm to a module under some arrival time condition can slow down the circuit once the transformed circuit is used under different arrival times. This suggests that the KMS algorithm cannot be used directly to remove false paths safely from combinational modules if the performance of the circuit needs to be guaranteed under all possible environments. Motivated by this observation, we have proposed a new approach to removing a specific set of false paths called strongly false paths with no penalty in circuit speed. The resulting circuit is no slower than the original under *any* arrival times at the primary inputs.

The algorithm has practical importance in hierarchical synthesis, where the proposed technique can be used to resynthesize a combinational module so that the new module is free from false paths. No knowledge on arrival times at the primary inputs is required in this circuit transformation, which makes it possible to resynthesize the module before the surrounding design is fixed. This local resynthesis enables one to perform more accurate timing analysis on the hierarchy via topological timing analysis. If the same module is used more than once, the master module can be made false-path-free once and for all without affecting the performance of any instance.

We have shown that the final redundancy removal in the KMS algorithm can increase the delay of a circuit if the delay is defined for each primary input vector separately. Therefore the proposed algorithm does not make the circuit irredundant after false path removal. Thus, one of the advantages of the KMS algorithm, i.e. removing redundancies that cannot be detected by conventional testing, but can adversely affect the timing of the circuit, is absent. Whether redundancies can be removed without slowing down the circuit under this strict definition of delay is still open.

[8]We expect that this CPU time can be further reduced by incorporating existing techniques for generating simplified SAT formulas developed for functional timing analysis [MSBSV93].

Chapter 9

Approximate Functional Arrival Time Analysis

Motivated by the need for functional arrival time analysis techniques that can handle industrial circuits in reasonable CPU time, we presented in Chapter 5 how functional arrival time analysis can be performed hierarchically without flattening an existing hierarchy. The hierarchical approach only performs detailed false path analysis on a leaf module at one time, and thus is better suited to analyze large circuits than flat analysis. An alternative to the hierarchical approach is *approximate* flat functional arrival time analysis, where flat timing analysis is conservatively performed in a less CPU-time intensive way. In this chapter we will study approximate algorithms for flat functional arrival time analysis. The goal is to compute a conservative yet accurate enough approximation to true delays in less computation time to make the analysis of large circuits tractable.

This chapter is organized as follows. Section 9.1 reviews previous research on approximate functional arrival time analysis. The limitations of flat analysis are summarized in Section 9.2. Section 9.3 presents algorithms for approximate functional arrival time computation. Experimental results are reported in Section 9.4. The chapter is concluded in Section 9.5.

9.1 Previous Work

Several researchers have proposed approximate functional arrival time analysis algorithms in the literature.

Huang *et al.* [HPS91, HPS94] proposed, as part of optimization techniques used in exact analysis, a simple approximation heuristic, in which a complex timed Boolean expression at an

internal node is simplified to a new independent variable arriving at the latest time referred to in the original expression. This simplification is applied only when the number of terms in the timed Boolean expression exceeds a certain limit, to control the computational complexity. Accuracy loss comes from the fact that the original functional relationship is completely lost by the substitution.

They also investigated a more powerful approximation technique in [HPS93, HPS96], in which each timed Boolean formula is under- and over-approximated by sum of literals and products of literals respectively so that each sensitizability check, which is a satisfiability problem in the exact analysis, can be performed conservatively in polynomial time. Since this approximation is fairly aggressive to guarantee the polynomial time complexity, delay estimates do not seem accurate enough to be useful. Their results, shown in [HPS93], are not clear about the accuracy of approximate delays. They merely showed ratios of internal nodes whose delays match the exact delays at the nodes. No result was shown on the accuracy of circuit delays.

More recently Yalcin *et al.* [YHS96] proposed an approximation technique, which utilizes user's knowledge about primary inputs. They categorize each primary input either as data or control and label all internal nodes either data or control using a certain rule. The sensitization condition at each node is then simplified conservatively so that it becomes independent of the data variables. The intuition behind this is that the delay of a circuit is most likely determined by control signals while data signals have only minor effects in the final delay. They showed experimentally that a dramatic speed-up is possible without losing much accuracy for unit-delay timing analysis based on static sensitization. However this path sensitization condition is known to underestimate true delays, i.e. it is not a safe condition, which defeats the whole purpose of timing analysis. More recently they confirmed that a similar speed-up and accuracy can be achieved for a correct sensitization condition (floating mode) without losing accuracy under the unit-delay model [Yal97b, Yal97a]. Although an application of the same technique to more sophisticated delay models is theoretically possible, it is not clear whether their algorithm can handle large circuits under such delay models. Moreover, their CPU times for exact analysis are much worse than state-of-the-art implementations available, which cancels some of the speed-up since their speed-up is reported relative to this slower algorithm[1].

In this chapter we apply their idea of using data/control separation to a functional arrival time analysis technique based on the XBD0 model [MSBSV93] (see Section 2.4.6) to design approximate algorithms. In addition a novel technique to trade off the complexity of the analysis and the

[1] One of the reasons why their exact algorithm is slower is that they try to represent in a BDD all the input vectors that activate the longest sensitizable delay while most of the state-of-the-art techniques determine the delay without representing these input vectors explicitly.

accuracy of delay estimates is proposed. The combination of these two ideas leads to a new approximation scheme, which for some extreme cases shows a speed-up of 70x, while maintaining accuracy within the noise range.

9.2 Limitation of Exact Functional Arrival Time Analysis

Although the exact algorithm proposed by McGeer *et al.* [MSBSV93] can handle many circuits of thousands of gates, it still has a size limitation. If a large network is given and timing analysis is performed under a detailed delay model such as the technology mapped delay model, it is likely that the algorithm runs practically forever[2]. Even if timing analysis is tractable, the computation time can be too large to be practical.

As seen in Section 2.4.6 the exact timing analysis in [MSBSV93] consists of repeated SAT solver calls. More precisely, for each candidate arrival time tested at a primary output, a χ-network is constructed such that the network computes the difference between the on-set (off-set) of the primary output and the set of input vectors which make the primary output stable to value 1 (0) by the given time. If the output never becomes 1 for any input assignment, i.e. it is not satisfiable, the output becomes stable completely by the time tested. To test whether this condition holds, a SAT formula which is satisfiable only if the output is satisfiable is created directly from the χ network, and a SAT solver is called on it. The size of the SAT formula is roughly proportional to the size of the χ network. The main difficulty in the analysis of large networks is that due to a potentially large size of the χ networks, the size of SAT formulas generated can be too large for a SAT solver to solve even after the optimization discussed in Section 2.4.6 has been applied[3]. Based on this observation we next discuss how to trade off the size of χ networks and the accuracy of delay estimates.

9.3 Approximate Functional Arrival Time Analysis

9.3.1 Reducing the Size of χ Networks

The main reason why χ networks become large in the exact analysis is that χ functions at many distinct arrival times must be computed for internal nodes. This size increase occurs when there are many distinct path delays from primary inputs to internal nodes due to the reconvergence

[2]The algorithm is CPU intensive rather than memory intensive since the core part of the algorithm is SAT.

[3]Theoretically it is not necessarily true that a smaller SAT formula is easier to solve. However we have observed that the size of SAT formulas is well correlated with the time the solver takes.

of the circuit. Therefore our goal is to control the number of distinct arrival times considered at each internal node. More specifically only a small number of χ functions are created at each internal node. This strategy avoids the creation of huge χ networks thereby controlling the size of SAT formulas generated.

Although this idea certainly helps reduce the size of χ networks, it must be done carefully so that the correctness of the analysis is guaranteed. We must never underestimate true delays since otherwise the timing analysis could miss timing violations when used in the context of timing verification. Overestimation is acceptable as long as reasonable accuracy is maintained. We guarantee this property by selectively underapproximating stability of signals. This underapproximation in turn overapproximates instability of signals thereby guaranteeing that delay estimates are never underapproximated.

The key idea on approximation is to modify the mapping from required times to arrival times discussed in Section 2.4.6 so that only a small set of arrival times forms the image of the mapping. Given the sorted set of required times $R = (r_1, \ldots, r_p)$ and the sorted set of arrival times $A = (a_1, \ldots, a_q)$ at an internal node n, the mapping $f : R \rightarrow A$ used in the exact analysis is defined as

$$f(r) = \begin{cases} \max a_i \in A \text{ such that } a_i \leq r & \text{if } r \geq a_1 \\ -\infty & \text{otherwise} \end{cases}$$

Since the stability of the signal at the node increases monotonically as time elapses by the definition of χ functions, it is safe to change the mapping so that it maps a required time to a time earlier than the time defined above. This corresponds to underapproximation of the signal stability. Thus, by modifying the mapping under this constraint so that only a small set of arrival times is required, one can control the number of nodes to be introduced in the χ network without violating the correctness of the analysis. Depending on how the original mapping in the exact analysis is changed several conservative approximation schemes can be devised. Two such approximation schemes are described next.

Topological Approximation

The most aggressive approximation, which we call *topological approximation*, is to map required times either to the topological arrival time (a_q[4]) or to $-\infty$. More formally, the mapping f^T

[4]To be precise, a_q can be earlier than the topological arrival time if an intermediate satisfiability call has already verified that by time a_q the signal is stabilized completely.

is defined as follows.

$$f^T(r) = \begin{cases} a_q & \text{if } r \geq a_q \\ -\infty & \text{otherwise} \end{cases}$$

It is easy to see that f^T is a conservative approximation of f. Since $\chi_{n,1}^{a_q} = n$ and $\chi_{n,0}^{a_q} = \overline{n}$, there is no need to create a new node for the χ function in the χ network[5]. Instead the node function or its complement of the original network can be used for the χ function. For the other arrival time $-\infty$, $\chi_{n,v}^{-\infty} = 0$ for $\forall v \in \{0,1\}$. Therefore it is sufficient to have a constant zero node in the χ network and use it for all the cases where the zero function is needed. Since neither of the arrival times needs any additional node in the χ network, this approximation never increases the size of the χ network. If this reduction is applied to all nodes, the analysis simply becomes pure topological analysis. Therefore, this approximation makes sense only if it is selectively invoked on a subset of nodes. A selection strategy is described later.

Semi-Topological Approximation

The second approximation scheme, called *semi-topological approximation*, is slightly milder than the first in terms of the power of simplifying χ networks. In this, required times are mapped to two arrival times again, but the times chosen are different. The times to be selected are 1) the arrival time, say a_e, matched with r_1 in the exact mapping f and 2) the topological arrival time a_q, which is the same as in the first approximation. The first approximation and this one are different only if $a_e \neq -\infty$, in which case the second one gives a more accurate approximation. To be precise, the definition of the new mapping function f^S is as follows.

$$f^S(r) = \begin{cases} a_e & \text{if } r < a_q \\ a_q & \text{otherwise} \end{cases}$$

If $a_e \neq -\infty$, the χ function for time a_e is now computed explicitly, and the corresponding node is added to the χ network. Similar extensions which give tighter approximations are possible by allowing more arrival times to remain after the mapping. A set of various approximations gives a tradeoff between the compactness of χ networks and the accuracy of analysis.

9.3.2 Control/Data Dichotomy in Approximation Strategies

Yalcin *et al.* [YHS96] proposed to use designer's knowledge on control-data separation of primary inputs for effective approximate timing analysis. They applied this idea to speed up their

[5]Notice that the χ network always includes the original circuit.

functional arrival time analysis technique using conditional delays [YH95] by simplifying signal propagation conditions of data variables. We adapt their idea, of using this knowledge, to the XBD0 analysis to develop a selection strategy of various approximation schemes.

Labeling Data/Control Types

Given data/control types of all primary inputs, each internal node is labeled data or control based on the following procedure. All the nodes in the network are visited from primary inputs to primary outputs in a topological order. At each node the types of its fanins are examined. If all of them are data, the node is labeled data; otherwise it is labeled control. Hence nodes labeled data are *pure* data variables with no dependency on control variables, while those labeled control are all the other variables with some dependency on control variables. This labeling policy is different from the one used in [YHS96], where a node is labeled data if at least one of its fanins is labeled data. In their labeling, nodes labeled data are variables with some dependency on data whereas nodes labeled control are pure control variables. The difference between the two labelings is whether pure data variables or pure control variables are distinguished. Our labeling will lead to tighter approximations.

Applying Different Approximations based on Data/Control Types

Once all the nodes are labeled, different approximation schemes are applied to different nodes based on their types. The strategy is as follows.

If a node is a control variable, the semi-topological approximation f^S is applied while if a node is a data variable, the topological approximation f^T is applied. The intuition is to use a tighter approximation for control variables to preserve accuracy while performing maximum simplification for data variables assuming they have less impact on delays than control variables.

Extracting Control Circuitry for Further Approximation

If the approximation so far is not powerful enough to make analysis tractable, further approximation is possible by extracting only the control-intensive portion of the circuit and performing timing analysis on the subcircuit. The extraction of the control portion is done by stripping off all pure data nodes from the original network under analysis as in Figure 9.1. Note that any circuit can be decomposed into a cascade circuit where all the nodes in the driving circuit are labeled data and those in the driven circuit labeled control by the definition of data variables. Therefore, the primary

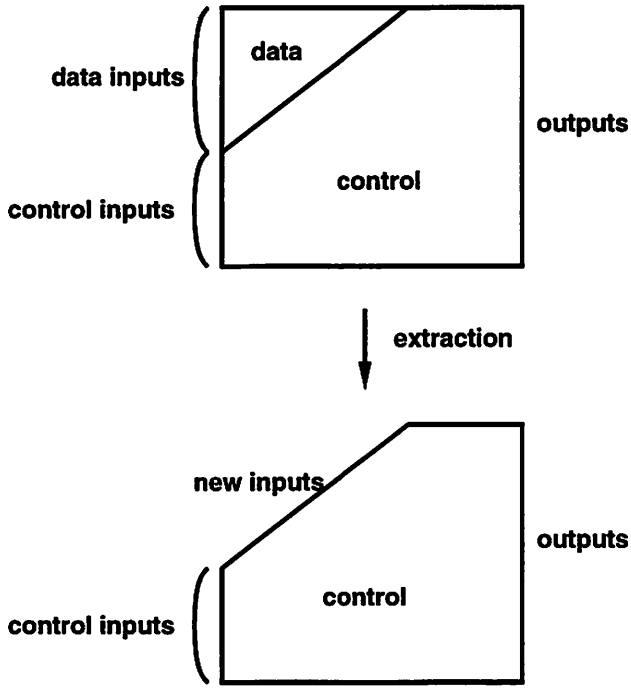

Figure 9.1: Extraction of a Control Subnetwork

inputs of the subcircuit include the boundary variables which separate the subcircuit from the pure data portion. We assume conservatively that delays of the pure data portion of the circuit are the same as topological delays, which gives arrival times at the new primary inputs of the extracted circuit. Analysis is then performed on this subcircuit as if it were the circuit given. Notice that this has a similar flavor to the approximation proposed in [HPS91].

The difference between this approximation and the previous method is that the subcircuit has a new set of primary inputs, which are assumed independent. However, it is possible that in the original circuit only a certain subset of signal combinations appears at the boundary variables. Since this approximation assumes that all signal combinations can show up, the analysis becomes pessimistic[6]. For example, if a signal combination which does not appear on the cut makes a long path sensitizable, it can make a delay estimate unnecessarily pessimistic. Although this method is more conservative than the one without subcircuit extraction, it reduces the size of a circuit to be analyzed much more significantly.

[6]If the set of all possible signal combinations at the boundary variables can be represented compactly, one can safely avoid this pessimism by multiplying the additional constraint to the SAT formula generated.

9.4 Experimental Results

We implemented the new approximation algorithms on top of the implementation of McGeer *et al.* [MSBSV93] under SIS environment [SSM+92]. To evaluate the effectiveness of the approximation, we focused on arrival time analysis of mapped ISCAS combinational circuits, which is generally much more time-consuming than analysis based on simpler delay models. In Table 9.1[7] the results on three circuits whose exact analysis takes more than 20 seconds on a DEC Alpha Server 7000/610 are shown[8]. Each circuit is technology-mapped first with the option specified in the second column using the `lib2.genlib` library. The delay of the circuit is then analyzed using three techniques. The first one (exact) is the exact method presented in [MSBSV93]. The remaining two are approximate methods; the second, called approx(1), is the first technique in Section 9.3.2 and the third, called approx(2), is the second one in Section 9.3.2 which involves subcircuit extraction. Control/Data specification for the primary inputs of these circuits are the same as those in [YHS96][9]. For each of the three analyses, delay estimates and CPU time are shown in the last two columns. One can observe that accuracy is preserved in the three examples in both of the approximation methods while CPU time is reduced significantly.

Table 9.2 summarizes a similar experiment for C6288, an integer multiplier, which is known to be difficult for exact timing analysis due to a huge amount of reconvergence. Since all the primary inputs are data variables, the approximate techniques proposed are degenerated into topological analysis. To avoid this inaccuracy all the primary inputs were set to control. Note that this sets all intermediate nodes to control. We then applied the first approximate method under this labeling. Although the approximation is not so powerful as the original algorithms, this at least enables us to reduce the size of χ networks without giving up accuracy completely. Since there is no data variable in the network, only approx(1) was tried. Significant time saving was achieved with only a slight overestimation in terms of analysis quality. The exact analysis is not only more CPU-time intensive but also much more memory-intensive than the approximate analysis. In fact we were not able to complete any of the three exact analyses within 150MB of memory. They ran out of memory in a couple of minutes. These exact analyses were possible after the memory limit was expanded to 1GB. The last example needs an additional explanation. In this example the delay estimate by the approx-

[7]Timing analysis was done in the linear search mode [MSBSV93] where the decrement time step is 0.1 and the error tolerance is 0.01.

[8]If exact analysis is already efficient, approximation cannot make significant improvement in CPU time; in fact the overall performance can be degraded due to additional tasks involved in approximation.

[9]More precisely, C1908(1) and C3540(1) in [YHS96] were used.

circuit	tech.map	#gates	top. delay	type of approx.	delay estimates	CPU time
C1908	-m 1	536	39.25	exact	34.77	29.1
				approx(1)	34.77	8.9
				approx(2)	34.77	5.4
C1908	-m 0	602	40.76	exact	35.76	41.2
				approx(1)	35.76	12.0
				approx(2)	35.76	5.2
C3540	-n 1 -AFG	1113	35.88	exact	35.66	727.0
				approx(1)	35.66	559.5
				approx(2)	35.66	502.9

Table 9.1: Functional Arrival Time Analysis: Exact vs. Approximate (CPU time in seconds on DEC AlphaServer 7000/610)

circuit	tech.map	#gates	top. delay	type of approx.	delay estimates	CPU times
C6288	-m 1	2429	127.23	exact	123.87	7850.2
				approx(1)	123.94	169.2
	-m 0	2371	123.51	exact	119.16	18956.2
				approx(1)	119.21	257.1
	-n 1 -AFG	2911	114.62	exact	112.92	15610.5
				approx(1)	112.86	1690.9

Table 9.2: Functional Arrival Time Analysis of C6288: Exact vs. Approximate (CPU time in seconds on DEC AlphaServer 7000/610)

imate algorithm is smaller than that by the exact algorithm although in Section 9.3 we claimed that the approximation algorithm never underestimates exact delay. The reason for this is that the SAT solver is not perfect. Given a very hard SAT problem, the solver may not be able to determine the result under a given resource, in which case the solver simply returns *unknown*. This is conservatively interpreted as being satisfiable in the timing analysis. In this particular example the SAT solver returned *unknown* during the exact timing analysis, which resulted in a delay overestimation, while in the approximate analysis the SAT solver never aborted because of the simplification of χ networks and gave a better overestimation. This example shows that the approximate analysis gives not only computational efficiency but also better accuracy in some cases.

To compare the exact and the approximate methods further, we examined the total CPU time of the exact analysis to see how it can be broken down. For the first example of C6288 the exact analysis took 714.7 seconds to conclude that any path of length 123.93 is false, which is about four times longer for the approximate analysis to conclude that the delay of the circuit is 123.94. The situation is much worse in the second example, where the exact analysis took 18390.8 seconds to conclude that any path of length 119.21 is false while the approximate method took only about 1.4% of this time to finish off the entire analysis.

9.5 Conclusions

We have studied approximate functional arrival time analysis as an extension to the existing exact analysis algorithm under the XBD0 model [MSBSV93]. Although the exact algorithm is applicable to large circuits, the performance is not satisfactory for circuits with a large number of reconvergences. The size of χ networks gets large for such circuits, and thus SAT formulas solved during the analysis become complex.

The core idea of the approximate algorithms was to control the size of χ networks used in the analysis to prevent the size of SAT formulas to be solved from getting large. This simplification of χ networks is performed systematically so that delay estimates are never underapproximated. To preserve accuracy as much as possible the use of knowledge on data/control separation of primary inputs originally proposed by Yalcin *et al.* [YHS96] was adapted to choose an appropriate approximation at each node. We have shown experimentally that the approximate algorithms compute arrival time estimates for large circuits with many reconvergences in significantly less CPU time while maintaining accuracy well within the accuracy of the delay model.

The approximation scheme presented in this chapter can be used as a replacement of exact

functional arrival time analysis whenever the exact algorithm is too slow or cannot complete. Since functional required time analysis can be performed by using functional arrival time analysis as in Section 3.3.2, this technique is also applicable to required time analysis.

Chapter 10

Delay-Optimal Technology Mapping

We have addressed various timing analysis problems in the first part of the dissertation. The main focus was how to estimate the timing characteristics of a given design accurately and efficiently under the existence of false paths.

In the second part of the dissertation we will discuss a synthesis aspect of timing-driven designs, i.e. how to synthesize a high-speed circuit automatically. Specifically we will focus on a synthesis step called *technology mapping*. Previous results on technology mapping show that if our objective function is to minimize the delay of a mapped circuit[1], the problem is solvable in polynomial time for Look-up Table (LUT) FPGAs while the complexity of the problem is unknown for library-based designs. In this chapter this gap will be closed by showing that a polynomial-time algorithm called FlowMap for delay-optimal technology mapping of FPGA designs [CD94a] can be adapted to library-based designs so that it guarantees delay optimality under a load-independent delay model. The algorithm runs in time linear in the size of a network.

This chapter is organized as follows. Section 10.1 introduces basic concepts of technology mapping and summarizes previous results in the literature. Section 10.2 gives an overview of the FlowMap algorithm [CD94a] for delay-optimal mapping for FPGAs. Section 10.3 then discusses how the FlowMap algorithm can be adapted to library-based technology mapping. Extensions to sequential circuits are presented in Section 10.4. Section 10.5 gives experimental results of the proposed technique. The chapter is concluded in Section 10.6.

[1]Delay in this chapter is the longest topological delay. Optimality is argued under this assumption. Delay-optimal technology mapping aware of false paths is a difficult problem, and no practical approach is known.

10.1 Preliminaries

10.1.1 Library-Based Technology Mapping

Logic synthesis [BHSV90] typically consists of two phases. The first step, called *technology-independent optimization*, is the step in which a given Boolean network is restructured without knowing an actual gate library or technology to be used. Generic optimization such as factoring, resubstitution and don't care minimization is performed to seek a good multi-level structure. This step is followed by *technology mapping*, in which the optimized network in the previous step is implemented by only using gates in a given library. The importance of technology mapping is increasing significantly since it is very difficult in deep sub-micron designs to estimate the effect of a generic optimization without knowing an actual technology to be used.

Technology mapping was initially performed by rule-based transformations in the early 80's [DBG+84, GBdGH86]. The approach is ad-hoc and has no guarantee about mapping quality. Furthermore different sets of transformation rules need to be maintained for different libraries.

In 1987 Keutzer [Keu87] proposed an algorithmic approach to the technology mapping problem, in which he observed similarity between this problem and the code optimization problem for programming languages and adapted an existing tree-covering technique for the latter problem to technology mapping. This approach soon dominated the rule-based approach and became the de facto standard.

In Keutzer's formulation a technology-independent circuit and each gate in a given library are decomposed into circuits only composed of two-input NAND gates (NAND2) and inverters (INV). The decomposed circuit is called a *subject graph* while each decomposed gate is called a *pattern graph*. Typically all possible NAND2/INV decompositions are generated for each gate in the library modulo isomorphism so that the gate is utilized maximally in a final implementation. Each pattern graph is associated with the area, delay and other characteristics of the corresponding gate. The technology mapping problem can then be formulated as covering the subject graph by using pattern graphs to optimize a given criterion. A subject graph is a DAG in general since it is derived from a given network. A pattern graph is a tree for most gates in a typical library while it can be a DAG for some gates, e.g. an XOR gate and a multiplexor gate. For the sake of simplicity assume that all the pattern graphs are trees for now.

Keutzer showed that if a subject graph is a DAG, graph covering for *minimum area mapping* is NP-hard [KR89]. Having demonstrated the inherent complexity of the original problem, he

considered the case where a subject graph and pattern graphs are trees. It turned out that this special case can be solved optimally in linear time using dynamic programming. Based on these results he proposed the three-step approach based on tree covering as an approximation to the DAG covering problem.

1. Decompose a subject DAG into a disjoint set of trees
2. Solve the technology mapping problem optimally for each tree
3. Glue the results together.

This separation of the problem again has become a standard approach due to the theoretical justification about the complexity of minimum-area DAG covering.

Inspired by Keutzer's result, technology mapping has been studied extensively to optimize different criteria. Rudell [Rud89] worked on *minimum-delay* technology mapping and showed that if loading effects are completely ignored, the minimum-delay mapping problem for subject *trees* can be solved optimally by dynamic programming in linear time. He also considered the minimum-delay mapping problem for trees under loading effects and showed that by maintaining the best mapping for each possible load at each node the same dynamic programming approach can guarantee optimal results. Touati [TMBW90, Tou90] further refined this idea later by combining the optimal tree mapping with sophisticated buffer tree construction. An interesting fact is that they directly started looking at *tree covering* without studying the complexity of *DAG covering* for minimum delay. To the best of our knowledge no one has investigated the exact complexity of the minimum-delay technology mapping problem where a subject graph is a DAG. Probably it was simply assumed that the problem is NP-hard without giving much thought.

Now consider the case where some pattern graphs are DAGs. Rudell showed that as long as those are leaf DAGs, the tree covering approach can be used without any modification [Rud89]. A leaf DAG is a DAG in which the only nodes with multiple fanouts are primary inputs. An XOR gate and a multiplexor gate have leaf DAG pattern graphs and thus can be handled without any problem.

So far we have focused on the case where a subject graph is a tree. To conclude this subsection we review previous work on DAG covering without tree decomposition. Detjens *et al.* studied this problem for area minimization in [DGR+87]. However since a heuristic approach was taken for covering, the results were not encouraging. In fact the DAG covering approach gave results of lesser quality than the tree-based approach. Although they also described an idea on node duplication similar to [CD94a] to be detailed later, it was apparently only tried for area optimization and no results

are reported on this approach in [DGR+87].

Touati [Tou90](page 115) conducted experiments on performance-oriented technology mapping by allowing overlaps between trees. The idea is similar to our mapping algorithm presented later. Although significant delay decrease was observed, he did not pursue this direction any further. No argument was given on the optimality or the complexity of the algorithm.

10.1.2 Technology Mapping for LUT-based FPGAs

In parallel to the works on library-based technology mapping the emergence of Field Programmable Gate Arrays (FPGAs) posed a new technology mapping problem in the early 90's. Due to their unique architecture the technology mapping problem for FPGAs has been tackled in completely different ways from library-based technology mapping.

LUT(Look-Up Table)-based FPGAs can implement any function of k inputs by a single LUT, where k is a fixed constant specific to a given FPGA family. By assuming the existence of a library containing all k-input functions, one can solve the technology mapping problem for FPGAs as an instance of the library-based mapping problem. This approach, however, has a serious drawback since this virtual library contains 2^{2^k} gates[2]. Even though the minimum-area tree covering for library-based designs can be solved in time *linear* in the size of pattern graphs, the number of gates in this virtual library makes the algorithm highly inefficient and impractical. Based on this observation many ideas have been proposed for the FPGA mapping problem again under different cost criteria. A survey of FPGA technology mapping algorithms is available in [CD96]. As for minimum area mapping Levin and Pinter [LP93] and Farrahi and Sarrafzadeh [FS94] proved that the problem is NP-hard for $k = 4$ and $k \geq 5$ respectively. As in the library-based mapping, once a network is restricted to a tree, the problem can be solved optimally in polynomial time [FS94]. Minimum-delay mapping, on the other hand, was shown for LUT-based FPGAs to be solvable in polynomial time by Cong and Ding in [CD92, CD94a]. The given circuit is directly mapped without decomposing its DAG structure to trees in this algorithm unlike conventional library-based mapping.

10.1.3 Summary

Technology mapping problems for library-based designs and FPGA-based designs have been investigated almost independently so far although the two problems are closely related to each

[2]Strictly speaking the library need not contain all the 2^{2^k} functions since some are equivalent to each other under input permutation and thus having one representative is good enough. However even after this simplification the library is still huge.

other.

The area-optimal mapping problem for DAG networks is NP-hard both for library-based designs and FPGA designs. As to the delay-optimal mapping problem, it is solvable in polynomial time for FPGAs while no polynomial-time algorithm is known for library-based designs. To the best of our knowledge the exact complexity of the delay-optimal technology mapping problem for library-based designs has never been discussed in the literature. How to close this gap between the two problems will be the topic of this chapter.

10.2 Delay-Optimal Technology Mapping for FPGAs

We will have a close look at the FlowMap algorithm proposed by Cong and Ding [CD94a] since this gives the basis of our proposed algorithm.

Assume that a network is decomposed into a k-bounded network [CD94a], which is a Boolean network where the number of fanins of each node is less than or equal to k. If a given network is not k-bounded, simple decomposition can yield an equivalent k-bounded network. In the following we assume that an LUT has a unit delay and that wiring delay is negligible.

The key idea of the FlowMap algorithm is in the labeling procedure that labels each node of the network its optimal depth achievable. The algorithm visits each node in the network in a topological order. All primary inputs are labeled 0 assuming that they are available at $t = 0$. At each intermediate node the goal is to investigate all cuts of size less than or equal to k in the fanin cone of the node and to find the best delay realizable at the node. Each such cut represents a mapping of the node. More specifically the node can be implemented by a single LUT whose inputs are the nodes forming the cut. The constraint on the size of cuts comes from the fact that an LUT can implement any function of up to k-inputs. Since nodes are visited in a topological order, by the time the current node is examined, the optimal depths of all the nodes in its transitive fanin are available. Therefore the optimal depth of the current node x can be computed as follows by dynamic programming.

$$optimal\_depth(x) = \min_{\text{cut } X:|X|\leq k} \max_{x_i \in X} (optimal\_depth(x_i) + 1)$$

Notice that this cost criterion meets *the principle of optimality* of dynamic programming. The cut X that realizes the optimal depth is stored at the node along with the depth. Although explicit enumeration of all valid cuts is possible by a brute-force approach, the complexity is pseudo polynomial $O(n^k)$ [CD94a], where n is the number of nodes in a given network. Cong and Ding showed that

this optimal depth computation at each node can be formulated as network flow computation whose runtime is strongly polynomial with respect to k [CD94a].

Once all the nodes have been labeled by their optimal depths, the network is traversed backward from primary outputs to primary inputs. At each primary output an LUT is created whose fanins are the same as the best cut stored at the node. The LUT creation is repeated for each of those fanins. This process is continued until either a primary input or a node whose output is already available in the mapping is reached. An important fact is that some intermediate nodes are automatically duplicated in an optimal way to guarantee optimal depths while in tree mapping no duplication is allowed.

The complexity of the entire algorithm is $O(kmn)$, where m is the number of edges in the network.

10.3 Delay-Optimal Technology Mapping for Library-Based Designs

Although the FlowMap algorithm was originally developed for FPGAs, the basic principle of the labeling procedure is not necessarily specific to those architectures[3]. In this section we will show how the FlowMap algorithm can be easily adapted to the standard library-based technology mapping under a load-independent delay model, where each gate has an intrinsic delay and loading has no effect in delays. This extension leads to a linear-time algorithm for delay-optimal technology mapping of DAG networks. We assume that a given network is decomposed into a subject DAG as usual. Therefore, the optimality of delay is claimed with respect to this subject DAG.

10.3.1 Computation of Optimal Delay at Internal Nodes

The only difference between FPGAs and library-based designs is how an internal node is mapped. In FPGAs all the local mappings that cover an internal node and part of its fanin cone are examined by enumerating all k-cuts of the fanin cone, which gives the best possible delay realized at the node. This step needs to be modified for library-based designs so that all successful matches for a given set of pattern graphs are systematically examined. However this can be easily done by mimicking the standard pattern matching step used in conventional technology mapping. More precisely the standard matching procedure against pattern graphs can be applied to the fanin cone of the

[3]It is interesting to note that Cong and Ding have a comment as follows.
"Our result makes a sharp contrast with the fact that the conventional technology mapping problem in library-based designs is NP-hard for general Boolean networks."(page 2 [CD94a])

node to exhaustively check all the successful matches. This way the best delay achievable at each intermediate node can be computed in a similar way to FlowMap. The only difference is that actual pin-to-pin delays of gates specified in a given library need to be used in our case instead of unit delay in FlowMap. As with FPGA mapping, the principle of optimality is still valid here.

Notice that as long as delay is optimized, any DAG pattern graph can be used directly without losing the optimality, i.e. it is not necessary to restrict the library to pattern graphs of trees and leaf DAGs. General DAG patterns are problematic only in the context of area optimization.

10.3.2 Pattern Matching

We now examine how pattern matching is performed between a subject graph and a pattern graph.

Pattern matching between a subject graph and a pattern graph in the context of technology mapping was studied extensively by Keutzer [Keu87] and Rudell [Rud89]. A match between a subject graph $G_s = (V_s, E_s)$ and a pattern graph $G_p = (V_p, E_p)$ is defined as follows [Rud89].

Definition 10.1 *A (standard) match of a pattern graph $G_p = (V_p, E_p)$ on a subject graph $G_s = (V_s, E_s)$ is a* one-to-one *mapping of the pattern graph nodes into the subject graph nodes $I : V_p \rightarrow V_s$ such that:*

1. $\forall e = (e_1, e_2) \in E_p, (I(e_1), I(e_2)) \in E_s$,
2. $\forall v \in V_p, |i(v)| \neq 0 \Rightarrow |i(v)| = |i(I(v))|$, *where* $i(v) = \{w \mid (w, v) \in E\}$ *for* $G = (V, E)$.

The first condition requires that the edge relationship in the pattern graph is completely preserved in the subject graph. The second condition constrains the in-degree of a non-primary-input node in the pattern graph to be the same as that of the matching node in the subject graph. Notice that it is allowed for a subject-graph node covered by an intermediate pattern-graph node to have fanout to nodes which are not covered by the pattern graph. However, in the conventional tree-covering based approach such a match is invalid, i.e. all the fanouts of a subject-graph node matched with an intermediate pattern-graph node need to be covered by the same pattern. A match satisfying this additional constraint is called an *exact* match [Rud89] and defined as follows.

Definition 10.2 *An* exact *match of a pattern graph $G_p = (V_p, E_p)$ on a subject graph $G_s = (V_s, E_s)$ is a one-to-one mapping of the pattern graph nodes into the subject graph nodes $I : V_p \rightarrow V_s$ such that:*

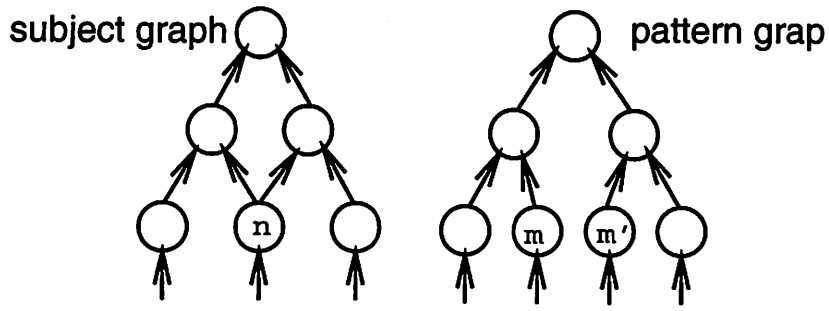

Figure 10.1: Standard Match vs. Extended Match

1. $\forall e = (e_1, e_2) \in E_p, (I(e_1), I(e_2)) \in E_s$,
2. $\forall v \in V_p, |i(v)| \neq 0 \Rightarrow |i(v)| = |i(I(v))|$,
3. $\forall v \in V_p, |i(v)| \neq 0$ *and* $|o(v)| \neq 0 \Rightarrow |o(v)| = |o(I(v))|$, *where* $o(v) = \{w \mid (v,w) \in E\}$ *for* $G = (V,E)$.

Rudell proposed an algorithm called `graph_match` [Rud89] for the general case where both a subject graph and a pattern graph are DAGs. We can simply use this matching algorithm to enumerate all successful standard matches instead of exact matches.

Although a constraint that a mapping is *one-to-one* is posed in the above two definitions by Rudell, this can be safely dropped as follows, which leads to the definition of a larger class of matches.

Definition 10.3 *An* extended *match of a pattern graph* $G_p = (V_p, E_p)$ *on a subject graph* $G_s = (V_s, E_s)$ *is a mapping of the pattern graph nodes into the subject graph nodes* $I : V_p \rightarrow V_s$ *such that:*

1. $\forall e = (e_1, e_2) \in E_p, (I(e_1), I(e_2)) \in E_s$,
2. $\forall v \in V_p, |i(v)| \neq 0 \Rightarrow |i(v)| = |i(I(v))|$.

The only difference between extended matches and standard matches is that in extended matches the requirement of a mapping from the pattern-graph nodes into the subject-graph nodes being one-to-one is dropped. Therefore extended matches subsume standard matches. This relaxation of the requirement allows duplication of subject-graph nodes while searching for a match by unfolding a DAG structure. Figure 10.1 shows an example where a pattern graph is matched successfully as an extended match but not as a standard match. Assume that a two-input node is a NAND2 gate and a single-input node is an inverter. Consider pattern matching at the top node of the subject graph

shown on the left against the pattern graph on the right. An extended match exists by mapping both m and m' to n while a standard match does not since such a mapping violates the one-to-one mapping property. A simple modification to the `graph_match` algorithm makes the algorithm search all extended matches instead of all standard matches without changing its asymptotic complexity.

10.3.3 Constructing an Optimum Mapping

Once a (best delay, best gate)-pair is computed at each node, a delay-optimal network can be constructed in exactly the same way as in FlowMap. We maintain a queue which contains nodes to be created in the final mapping. This queue is initialized to the set of all primary outputs. A node is taken from the queue and the best gate at the node is created in the mapping. Each fanin node of the gate is then inserted to the queue if the fanin is not a primary input and does not yet have a corresponding gate in the mapping. Once the queue becomes empty, the mapping is complete.

Lemma 10.1 *The labeling procedure described in Section 10.3.1 computes the optimum delay achievable at each node n of a subject graph under a load-independent delay model.*

Proof We can prove this by an induction on the structure of a given subject graph.

Base case (n is a primary input): The delay value stored at n is the arrival time of n. Therefore it is indeed the optimum delay of n.

Induction (n is not a primary input): Suppose by induction that all the nodes in the transitive fanin of n have their optimum delay values. Since the delay model is load-independent, the optimum delay value of any node in the transitive fanin of n remains valid regardless of the fanout structure of the node. Therefore by performing exhaustive pattern matching and choosing the minimum delay over all the successful matches at n, the minimum delay at n is computed.

□

Theorem 10.1 *The technology mapping algorithm described above gives a mapped circuit whose delay is optimum under a load-independent delay model.*

Proof The construction of a mapped circuit is performed recursively from primary outputs. During this recursive construction each node is implemented by the gate that achieves the best delay at the node. Since the delay value computed at each node in the labeling phase is guaranteed to be

optimum from Lemma 10.1, the resulting network has an optimum delay. □

10.3.4 Complexity of DAG Mapping for Delay Minimization

An application of `graph_match` to enumerate all successful matches at a single node is $O(p)$ [Rud89], where p is the number of nodes in the entire *unique* pattern graphs [4]. Since this procedure is called once for each node in a subject graph, the complexity of the labeling step is $O(sp)$, where s is the number of nodes in the subject graph. The final step of constructing a delay-optimal mapping only costs $O(s)$. Therefore the complexity of DAG mapping is $O(sp)$. Since p is a constant defined by a given library, the procedure is linear in the size of a subject graph.

10.3.5 Comparison between DAG Mapping and Tree Mapping

In the past, performance-oriented technology mapping has been done by a combination of tree covering and buffer tree construction [Tou90]. The fundamental limitation of this conventional tree-covering approach is that the search space is highly limited by the structure of a given subject graph since multiple-fanout points in the subject graph are completely preserved in the final results. On the other hand, since DAG mapping does not respect initial multiple-fanout points at all, it can explore a strictly larger search space. In other words multiple-fanout points are created as the result of delay optimization as we will see later in Figures 10.2, 10.3 and 10.4. Buffering techniques proposed in the literature can be directly used in conjunction with DAG covering to speed up such multiple-fanout points.

Another major difference is how subject-graph nodes are duplicated during technology mapping. DAG mapping can duplicate subject-graph nodes while creating final mappings whereas in tree mapping no duplication is allowed since each subject-graph node is covered only once by a single pattern. In some sense, subject-graph node duplication is limited to the buffer tree construction phase in the tree-mapping-based approach.

Figure 10.2 illustrates how duplication of subject graph nodes helps reduce the delay of a mapping. Consider a subject graph shown on the left. Suppose that a pattern graph on the right is available in a given library. If tree mapping is invoked on this subject graph, the pattern graph is of no use since there is no exact match between the subject graph and the pattern graph. If, on the other

[4] Note that p is not equal to the number of nodes in the entire pattern graphs since during matching a single pattern graph is tried for all possible permutations of its inputs. p is thus the number of nodes in the expanded pattern graphs. See [Rud89] for details.

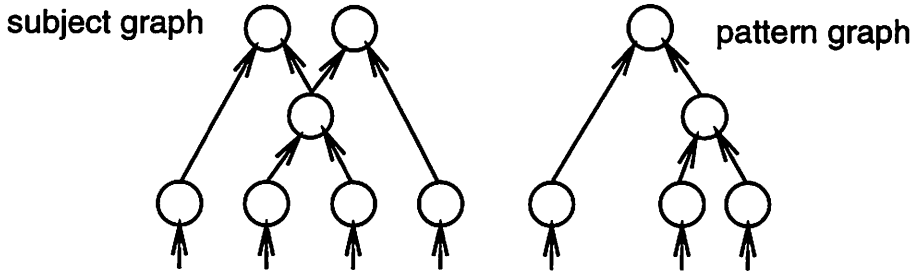

Figure 10.2: DAG Mapping vs. Tree Mapping

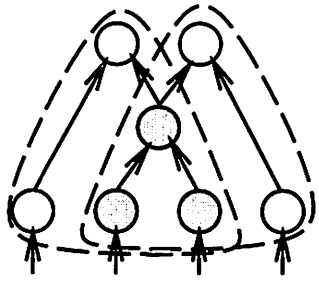

Figure 10.3: Matchings in DAG Mapping

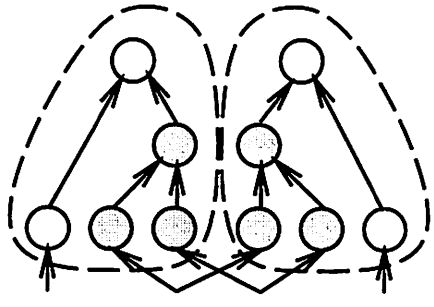

Figure 10.4: Duplication of Subject-Graph Nodes in DAG Mapping

hand, DAG mapping is employed, the two output nodes in the subject graph are matched with the pattern graph as in Figure 10.3. The mapped circuit corresponding to these matchings is shown in Figure 10.4. The shaded nodes in the subject graph in Figure 10.3 are duplicated in this mapping, which makes effective use of the pattern graph possible.

This example also illustrates how multiple-fanout points are created in DAG mapping. Since the middle node of the subject graph with multiple fanouts is an internal node of each of the matchings in Figure 10.3, the mapped circuit does not inherit the multiple fanout point. On the other hand, the two primary inputs of the subject graph in the middle have multiple fanouts in the mapped circuit while each of the inputs has a single fanout in the subject graph.

10.3.6 Example

Figure 10.5 shows a subject graph and pattern graphs. We will take this example to show the effectiveness of the DAG mapping compared with the conventional tree covering. Assume that each gate in the library has a unit delay for any input-output pair, and that the primary inputs of the circuit arrives at $t = 0$. The goal is to synthesize the fastest mapped circuit.

The subject graph has a multiple fanout point in the middle. Therefore in the conventional approach the subject graph is partitioned into three trees as in Figure 10.6. Each tree is mapped optimally by dynamic programming in a topological order. Figure 10.7 shows the optimal delay achievable at each node under the tree covering approach. The best implementation found in the tree covering is a mapped circuit where each subject-graph node is implemented by the corresponding gate in the library. The top and the bottom outputs are available at $t = 3$ and $t = 4$ respectively.

The use of the DAG mapping does not require the initial tree decomposition. This makes it possible to find matches crossing the multiple fanout point shown in Figure 10.8. These matches are never found in the tree covering. Figure 10.9 shows the optimal delay at each node under the DAG mapping. Both of the outputs are available at $t = 2$. The resulting mapped circuit is in Figure 10.10. Note that the NAND gate and the inverter marked with $\surd$ in Figure 10.9 are duplicated in the final circuit in Figure 10.10.

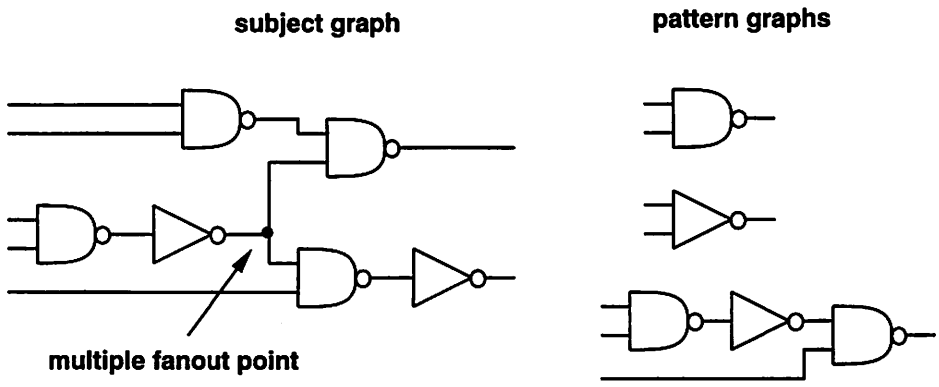

Figure 10.5: Example: DAG Covering vs. Tree Covering

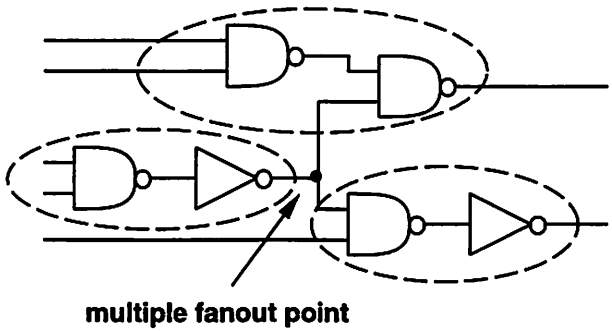

Figure 10.6: Example: Tree Decomposition

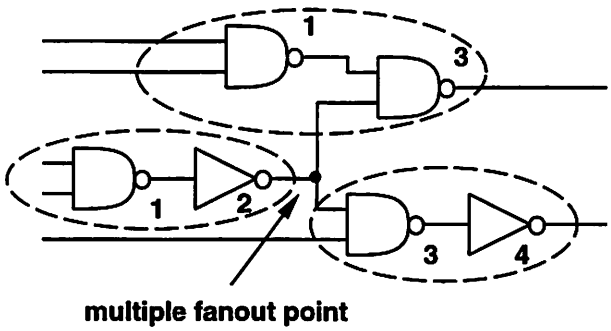

Figure 10.7: Example: Tree Covering

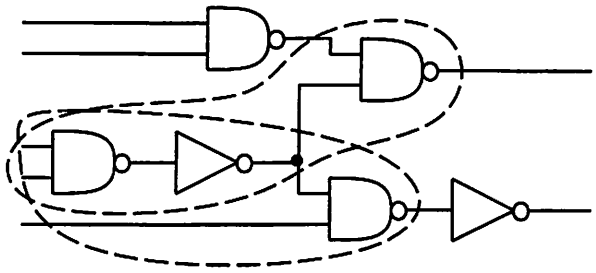

Figure 10.8: Example: Matches Crossing a Multiple Fanout Point

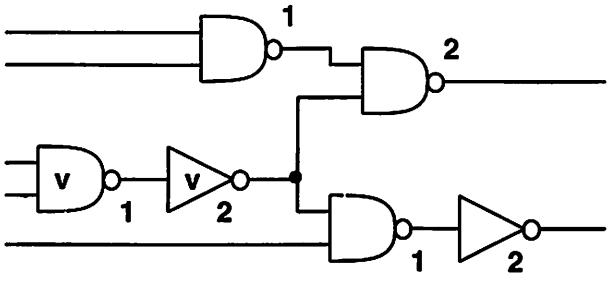

Figure 10.9: Example: DAG Covering

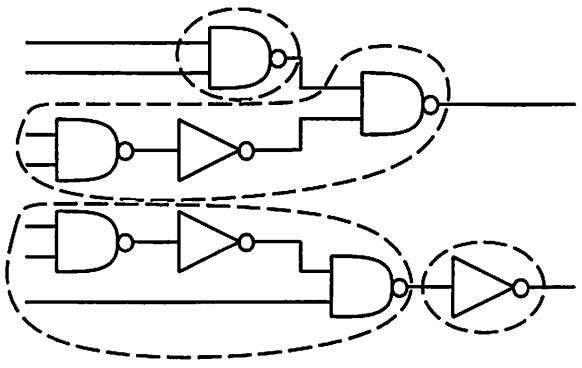

Figure 10.10: Example: Delay-Optimal Mapped Circuit

10.4 Extensions

The approach of Section 10.3 can be generalized to sequential circuits so that optimal cycle time is guaranteed in conjunction with retiming [LS83, LS91][5]. We only consider sequential circuits with edge-triggered latches all of which are controlled by a single clock.

This problem was studied for LUT-based FPGAs by Pan and Liu [PL96, PL98]. Given a k-bounded network consider the following three-step transformation.

1. Retime an initial circuit
2. Perform technology mapping of the combinational portion of the circuit
3. Retime the resulting mapped circuit.

Pan and Liu proposed a polynomial-time algorithm for computing the minimum cycle-time mapping among all the mapped circuits obtained by the above transformation, which was later improved by Cong and Wu [CW96, CW98a]. The key ingredient is a polynomial-time decision procedure which determines whether there exists a mapping whose cycle time is less than or equal to a given value. This procedure is used repeatedly to guide a binary search to determine the minimum cycle time achievable by retiming and optimal technology mapping. The core of this decision procedure is again a labeling scheme quite similar to the one used in FlowMap. All k-cuts at each intermediate node are explored by considering retiming possibility. This is again done implicitly by converting the original problem to a network flow problem. This step of examining all k cuts can be replaced by pattern matching as was done for combinational mapping. All the other theories hold without any modification.

More recently, Cong and Wu proposed a new optimal FPGA mapping algorithm for sequential circuits where only forward retiming is explored [CW98b]. This restriction makes the computation of the initial states of retimed circuits easy. The algorithm can also be easily adapted to library-based designs in a similar way.

So far optimality is guaranteed in terms of a subject graph constructed arbitrarily from a given circuit by decomposition. Since a single subject graph is chosen among a huge number of different decompositions without knowing an actual library to be used, it is likely that many potentially good mappings are simply not explored due to this initial choice. Lehman *et al.* [LWGH95,

[5]Grodstein *et al.* [GLH+94] considered retiming in the context of technology mapping for library-based designs and proposed an area-optimal algorithm for trees.

circuit	Delay		Area		CPU time	
	tree	DAG	tree	DAG	tree	DAG
C432	12.13	10.29	442	484	0.5	0.5
C499	10.16	8.03	904	960	0.9	1.1
C880	9.43	7.87	710	755	0.8	0.9
C1355	13.06	9.66	1146	1488	1.1	1.2
C1908	13.87	10.71	1223	1572	1.5	1.7
C2670	11.54	9.43	1552	2008	2.3	2.6
C3540	17.20	14.00	2075	2926	3.1	3.7
C5315	16.55	13.04	3687	4275	5.4	6.0
C6288	56.99	41.95	4107	9291	4.9	5.9
C7552	14.23	11.06	4983	6452	6.8	8.4

Table 10.1: Tree mapping vs. DAG mapping for `lib2.genlib`

LWGH97] have recently resolved this issue by encoding various decompositions into a single extended subject graph called *mapping graph*[6] and performing technology mapping on it. Since this technique is orthogonal to our technique, the two can be combined to produce even better results. In fact, we have recently become aware [Wat97] that the actual implementation of Lehman *et al.* performed DAG covering similar to ours although they discussed their algorithm for subject trees in [LWGH97]. It is interesting to know how much delay improvement in [LWGH97] is due to DAG covering.

10.5 Experimental Results

To show the effectiveness of this approach the technology mapper of SIS [SSM+92] was extended so that delay-optimal mapping is obtained for combinational circuits by DAG covering [7]. As discussed in the previous sections, the delay model used in this experiment is the intrinsic delay model where a fixed, load-independent delay is given between each input and the output of a gate. This is in fact the delay model used in [LWGH97]. Although loading effects are certainly an important factor in delays, there are several justifications. In design scenarios where continuous sizing of any gate is permissible, one way to capture this flexibility in technology mapping is to approximate

[6] They showed that the flexibility of retiming can be encoded in a mapping graph at the expense of the size increase of the graph. The relationship between this approach and the adaptation of Pan-Liu's algorithm is yet to be clarified.

[7] In this experiment we used `graph_match` for finding only standard matches instead of extended matches. Therefore the optimality of the results is claimed with respect to standard matches. So far we have not been able to see any major difference in mapping quality between the use of standard matches and extended matches.

this flexibility by having many discretely sized gates. Unfortunately this approach is known to be very expensive. The approach taken in [LWGH97] is to pick a single delay for each gate and perform technology mapping by ignoring loads. Each gate in the final mapping is then continuously sized by considering actual loads so that the delay matches the one associated with the gate. Even without the capability of continuous sizing, the buffer tree construction methods of [BCD89, SSV90, Tou90] can be used later at multiple fanout points to reduce load dependence. Therefore the use of this delay model is at least justified as an approximation to the minimum-delay mapping problem under realistic delay models.

Table 10.1 shows the comparison of the quality of final circuits between the DAG mapping approach proposed in this paper and the standard tree mapping approach. In this experiment each benchmark circuit was first decomposed into a subject graph. We then applied the DAG mapping algorithm and the tree mapping algorithm on this same subject graph using MCNC gate library `lib2.genlib`[8]. No technology independent optimization was applied to benchmark circuits before technology mapping. No fanout optimization was used. The effectiveness of the DAG mapping algorithm is clear. We were able to obtain significantly faster circuits. CPU time was obtained on DEC AlphaServer 8400 5/300 and is reported in seconds. The increase of CPU time from tree mapping to DAG mapping is reasonable.

The same experiment was repeated using different libraries to see how the DAG mapping algorithm performs on rich libraries. MCNC libraries `44-1.genlib` and `44-3.genlib` were used in this comparison. The former library only contains 7 gates while the latter library has 625 gates, many of which are complex gates with many inputs[9]. `44-3.genlib` is a strict superset of `44-1.genlib`. Table 10.2 and Table 10.3 summarize the results of `44-1` and `44-3` respectively. We can see that the difference in mapping quality between the DAG and tree mapping approaches is further pronounced with the use of richer libraries.

It is interesting to observe that DAG mapping can generate faster and smaller results in some cases, for examples in C1355 and C6288 in Figure 10.2. The reason is that complex gates are used more frequently in DAG mapping, which leads to area effective covering in some cases in spite of potential node duplication.

[8]Each gate has a non-zero load-dependent delay specified in `lib2.genlib`. In the experiment this was simply assumed to be zero.

[9]The largest gate has 16 inputs.

circuit	Delay		Area		CPU time	
	tree	DAG	tree	DAG	tree	DAG
C432	24	19	784	1006	0.4	0.4
C499	25	16	1772	2220	0.8	0.8
C880	20	15	1250	1337	0.7	0.7
C1355	27	22	2100	1546	1.0	1.0
C1908	37	24	2251	3058	1.3	1.3
C2670	27	18	2998	4568	2.0	2.0
C3540	42	30	4007	6640	2.7	2.8
C5315	46	33	6817	8352	4.6	4.8
C6288	125	120	7782	7121	4.3	4.4
C7552	39	28	9552	11149	6.0	6.3

Table 10.2: Tree mapping vs. DAG mapping for `44-1.genlib`

circuit	Delay		Area		CPU time	
	tree	DAG	tree	DAG	tree	DAG
C432	21	11	624	1094	21.5	38.5
C499	18	9	1324	1910	35.3	68.9
C880	18	8	946	1466	35.2	55.9
C1355	26	10	1796	2440	41.5	69.3
C1908	28	11	1755	2587	57.2	123.5
C2670	22	10	2314	3943	92.2	159.7
C3540	28	13	2983	6148	128.2	255.6
C5315	31	15	5115	6685	220.4	341.5
C6288	125	42	7694	14775	155.1	229.5
C7552	27	13	7062	13267	248.7	491.0

Table 10.3: Tree mapping vs. DAG mapping for `44-3.genlib`

10.6 Conclusions

We have shown that the delay-optimal technology mapping problem can be solved optimally under load-independent delay models without decomposing a subject DAG into trees. The algorithm is an adaptation of a polynomial time algorithm for delay-optimal mapping of look-up table type FPGAs, and runs in time linear in the size of a subject graph. This is the first result showing that the delay-optimal technology mapping problem for library-based designs is solvable in polynomial time for general DAG networks. We have experimentally shown that the proposed approach gives significant improvement in delay compared to conventional tree mapping. Extensions of this technique to sequential circuits have also been discussed. The relationship between technology mapping problems for library-based designs and FPGA designs has been clarified.

In this chapter we focused on delay minimization without any area consideration. Therefore at each intermediate node the fastest mapping is simply created no matter how critical the node is. By constructing slower but smaller mapping for non-critical subnetworks a better control over area increase can be achieved. Cong and Ding proposed heuristics to address this issue in [CD94a] for FPGA mapping. They also have results on area-delay tradeoff based on the FlowMap algorithm [CD94b]. Chaudhary and Pedram [CP92, CP95] studied area-delay tradeoff for library-based technology mapping. Touati *et al.* [TSB91, Tou90] gave an area-recovery heuristic applicable to technology-independent delay optimization. The incorporation of these ideas remains as future work.

The use of load-independent delay models has been assumed in this chapter. Although the capability of continuous gate sizing justifies such a delay model in theory, sizing is only allowed up to a specified limit in reality. Buffer tree construction needs to be used in conjunction with DAG mapping in this case. Interaction between buffer tree construction and DAG mapping is yet to be studied.

Chapter 11

Conclusions

We have studied various problems arising in timing analysis and optimization of high-performance digital circuits. The contribution of the dissertation is summarized below.

The first part of the dissertation focused on timing analysis of gate-level circuits. The goal of timing analysis at the gate level is to estimate timing characteristics of a given gate-level circuit under the assumption that the delay of each gate in the circuit has been pre-characterized by detailed simulation at a lower level. Two procedures essential in gate-level timing analysis are:

1. arrival time analysis, in which given arrival times at the primary inputs of a circuit are propagated forward to compute the arrival time at a primary output, and

2. required time analysis, in which given required times at the primary outputs of a circuit are propagated backward to compute the required time at a primary input.

A major difficulty in accurate gate-level timing analysis is in detection of false paths. A false path is a topological path not responsible for the signal stability of an output. Since false paths have no impact on the delay of a circuit, they need to be excluded when timing analysis is performed. Detection of false paths is crucial in accurate delay analysis. Timing analysis with the ability of false path detection is called functional timing analysis.

Although functional arrival time analysis has been studied extensively in the last decade, little was known about functional required time analysis. This problem was studied in Chapter 3. We showed that the underlying theory of an existing functional arrival time analysis technique can be extended to functional required time analysis. The consideration of false paths in required time analysis led to a generalized notion of required times, where an input of a circuit is required at different times under different input vectors. We further showed that input-vector dependent required time

is not necessarily unique even if an input vector is specified unlike topological required time analysis. An exact algorithm for functional required time analysis was presented for the XBD0 model, followed by a set of approximate algorithms applicable to the analysis of large circuits.

Chapters 4 to 8 dealt with various timing analysis problems for combinational modules. A combinational module is a combinational circuit which can be used under any surrounding environment.

Chapter 4 discussed delay characterization of a combinational module. The goal of the delay characterization is to compute a compact delay abstraction of the module that captures the timing characteristics of the module accurately. The fact that the arrival times at the primary inputs are unknown makes false-path-aware delay characterization difficult since state-of-the-art functional arrival time analysis techniques are dependent on arrival time conditions at the inputs. Our major contribution in this chapter was to show that the problem can be solved by a direct application of functional required time analysis. The resulting exact delay abstraction is valid and accurate under any arrival time condition. We then discussed several other methods of computing a delay abstraction of a combinational module using functional arrival time analysis. Accuracy and correctness of these methods were clarified.

Chapter 5 discussed hierarchical functional arrival time analysis as an application of the delay characterization techniques for combinational modules in Chapter 4. A common assumption in existing functional arrival time analysis techniques is that a circuit under analysis has a flat structure without any hierarchy. Therefore, if a hierarchical circuit is given, we need to flatten it before analysis, potentially resulting in a huge circuit. Chapter 5 addressed hierarchical functional arrival time analysis where timing analysis is performed in a bottom-up fashion by respecting a given hierarchy. The false-path-aware delay characterization techniques presented in Chapter 4 are directly applicable to compute a delay abstraction of a module used in the lowest level of a hierarchical circuit. The assumption made in Chapter 4 that the surrounding environment of a combinational module is unknown played a key role. Timing analysis was then performed at each level of the hierarchy in a bottom-up way to compute a delay abstraction of the level from the delay abstractions of sub-hierarchies. This hierarchical approach naturally supports incremental analysis capability, which is missing in traditional flat analysis.

Chapter 6 introduced a notion of timing-safe replaceability applicable to combinational modules. A module is said to be a timing-safe replacement of another if the former is no slower than the latter under any arrival time condition and any input vector. If a new module is a timing-safe replacement of a module, we can safely replace the original module with the new without deterio-

rating the performance regardless of how the module was used originally. We showed that whether a module is a timing-safe replacement of another can be determined by examining the exact delay abstractions of the two modules.

False paths in a combinational module are known to be relative to arrival time conditions at the inputs. Therefore, a false path under some arrival time condition can be true under another. Chapter 7 introduced a more stringent notion of false paths, called strongly false paths, which can be uniquely defined independent of arrival time conditions. A path is said to be strongly false if the path is not responsible for the stability of the outputs of a module under any arrival time condition. These false paths can be safely assumed to be false for combinational modules. After showing that strongly false paths can be determined from the exact delay abstraction of a module, we presented an algorithm to detect such paths without an explicit computation of the delay abstraction.

Based on the results of the previous two chapters Chapter 8 addressed false path removal from combinational modules. An algorithm to remove strongly false paths from a combinational module was presented, which is guaranteed to give a timing-safe replacement of the original. Since the module after the transformation is false-path free, it can be analyzed more accurately with topological analysis than the original. The resulting module has timing characteristics no worse than the original under any surrounding environment. Thus this can be thought of as a timing optimization technique for combinational modules.

The first part of the dissertation was concluded in Chapter 9, where an approximation scheme for flat functional arrival time analysis was investigated. Although flat functional arrival time analysis is well understood as the result of intensive research in the past decade, flat analysis of large circuits with a large number of reconvergences is still CPU time intensive. We adapted an existing functional arrival time analysis technique based on satisfiability so that the size of a satisfiability problem created during timing analysis is controlled by introducing conservative approximation. We experimentally confirmed that the approximation technique is effective in reducing CPU time only with minor overestimation of delays.

The second part of the dissertation addressed timing optimization of digital circuits. Specifically, delay optimization in technology mapping was discussed. Technology mapping takes a technology-independent Boolean network and generates a functionally equivalent network in which every gate is a member of a given gate library. The most prevalent technology mapping technique currently used for delay optimization is based on tree matching, where a technology-independent network is partitioned into trees and each of them is mapped separately to minimize delay. Although the delay-minimal technology mapping problem for trees has been known to be solvable optimally,

this approach is suboptimal because of the initial partitioning step. No optimization crossing tree boundaries is possible. In Chapter 10, inspired by a previous work on technology mapping for FPGAs, we presented a delay-optimal technology mapping algorithm applicable to DAG networks and showed that a delay-optimal mapping can be computed in time linear in the size of a given network under a load-independent delay model. This result implies that tree partitioning of an initial network is not necessary for delay optimization unlike area optimization, where without the partitioning the problem becomes NP-hard.

Future Work

The recent advent of deep-submicron designs has provided various new challenges at almost every level of design methodology. Since various analog effects play a larger role in timing behaviors, timing analysis under more refined delay models such as a slew-sensitive delay model and a table-lookup delay model needs to be investigated to achieve desirable accuracy under industrial environments. It would be interesting to see how the timing analysis and optimization results obtained in the dissertation can be extended to a more realistic delay model.

Among various analog effects crosstalk is becoming one of the most critical issues at logic synthesis level and below. If two wires are running in parallel only a small distance apart, the signal behavior of one wire can be affected by that of the other. This is called a crosstalk. For example, if the values on both wires are rising from 0 to 1 almost simultaneously, the effective capacitance between the two wires is smaller than the one without any interaction. Therefore, the delays of the wires decrease. On the other hand, if both wires are switching in different directions, the delays increase. Existing timing analysis techniques are not capable of taking into account this crosstalk effect accurately since the delays of gates and wires are assumed to be independent. Although conservative analysis is possible by assigning the worst possible delay under crosstalk to each gate or wire, this is likely to give too pessimistic results since crosstalk can only happen conditionally. Recent progress on crosstalk-aware functional arrival time analysis and delay modeling can be found in [Kir97] and [TKB97].

We have studied input-vector dependent delay abstractions and their implications to timing analysis in this dissertation. Although different delays can be associated with different input vectors under this framework, the effect of relative arrival times at the fanins of a gate is still ignored. In dynamic circuits, frequently used in high-end designs, even if a gate receives the same input vector at its fanins, the delay characteristics of the gate may vary depending on the relative arrival times of the

fanins. This is because the configuration of pull-down conducting paths is sensitive to the relative arrival times of the fanins. Furthermore, the amount of charges to drain depends on the input vector previously applied, and has a direct impact on the delay of the gate. To capture these behaviors accurately more refined analysis is required. Gray *et al.* [GLRKCI94] and Sun *et al.* [SDC94, SDC98] have preliminary results for this problem.

Delay fault testing is a methodology for testing whether a manufactured circuit has a timing violation or not. The goal of delay fault testing is to generate a pair of input vectors, the application of which in that order can verify whether there is a delay fault on a given path by observing the response at primary outputs. Unlike timing analysis we cannot assume any upper bound for gate delays. Although the problem is different from timing analysis, the two problems are related with each other. For example false paths play a major role in delay testing since if a path is false under all delay assignments to the gates, there is no need to test the path in the first place. Although interrelated, the two research problems have been studied almost independently, and thus the connections between the two are far from clear. Recent work by Sivaraman and Strojwas [SS97] is one example that applied a result in delay fault testing to timing analysis.

Bibliography

[ABBS95] A. Aziz, R. K. Brayton, F. Balarin, and V. Singhal. Timing-safe replaceability for combinational designs. In *Proceedings of TAU 95: ACM/SIGDA International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 121–128, November 1995.

[AMF97] R. Aggarwal, R. Murgai, and M. Fujita. Speeding up technology-independent timing optimization by network partitioning. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 83–90, November 1997.

[BCD89] C. L. Berman, J. L. Carter, and K. F. Day. The fanout problem: From theory to practice. In C. L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the 1989 Decennial Caltech Conference*, pages 69–99, March 1989.

[BCH+94] R. I. Bahar, H. Cho, G. D. Hachtel, E. Macii, and F. Somenzi. Timing analysis of combinational circuits using ADD's. In *Proceedings of the European Design and Test Conference*, pages 625–629, March 1994.

[BDB96] S. Bhattacharya, S. Dey, and F. Brglez. Fast true delay estimation during high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1088–1105, September 1996.

[BFG+93] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 188–191, November 1993.

[BHSV90] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):1062–1081, February 1990.

[BI88] D. Brand and V. S. Iyengar. Timing analysis using functional analysis. *IEEE Transactions on Computers*, 37(10):1309–1314, October 1988.

[BMCM87] J. Benkoski, E. Vanden Meersch, L. Claesen, and H. De Man. Efficient algorithms for solving the false path problem in timing verification. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pages 44–47, November 1987.

[BMCM90] J. Benkoski, E. V. Meersch, L. J. M. Claesen, and H. De Man. Timing verification using statically sensitizable paths. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(10):1073–1084, October 1990.

[Bra98] D. Brand. Personal communication, March 1998.

[Bro90] F. M. Brown. *Boolean Reasoning*. Kluwer Academic Publishers, 1990.

[BRSVW87] R. K. Brayton, R. Rudell, A. L. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level interactive logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(6):1062–1081, November 1987.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BS89a] R. K. Brayton and F. Somenzi. Boolean relations and the incomplete specification of logic networks. In G. Musgrave and U. Lauther, editors, *VLSI 89: Proceedings of the IFIP TC10/10.5 International Conference on Very Large Scale Integration*, pages 231–240. North Holland, August 1989.

[BS89b] R. K. Brayton and F. Somenzi. An exact minimizer for Boolean relations. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pages 316–319, November 1989.

[BS95] K. P. Belkhale and A. J. Suess. Timing analysis with known false sub graphs. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 736–740, November 1995.

[CC96] K.-T. Cheng and H.-C. Chen. Classification and identification of nonrobust untestable path delay faults. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(8):845–853, August 1996.

[CCHD93] H.-C. Chen, S. W. Cheng, Y.-C. Hsu, and D. H.-C. Du. A path sensitization approach to area reduction. In *Proceedings of IEEE International Conference on Computer Design*, pages 73–76, October 1993.

[CD92] J. Cong and Y. Ding. An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 48–53, November 1992.

[CD93] H.-C. Chen and D. H.-C. Du. Path sensitization in critical path problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(2):196–207, February 1993.

[CD94a] J. Cong and Y. Ding. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(1):1–12, January 1994.

[CD94b] J. Cong and Y. Ding. On area/depth trade-off in LUT-based FPGA technology mapping. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(2):137–148, June 1994.

[CD96] J. Cong and Y. Ding. Combinational logic synthesis for LUT based field programmable gate arrays. *ACM Transactions on Design Automation of Electronic Systems*, 1(2):145–204, April 1996.

[CDC92] H.-C. Chen, D. H. C. Du, and S. W. Cheng. Circuit enhancement by eliminating long false paths. In *Proceedings of 29th Design Automation Conference*, pages 249–252, June 1992.

[CDL93] H.-C. Chen, D. H.-C. Du, and L.-R. Liu. Critical path selection for performance optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(2):185–195, February 1993.

[CMZ+93] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transformation for large Boolean functions with application to technology mapping. In *Proceedings of 30th Design Automation Conference*, pages 54–60, June 1993.

[CP92] K. Chaudhary and M. Pedram. A near optimal algorithm for technology mapping minimizing area under delay constraints. In *Proceedings of 29th Design Automation Conference*, pages 492–498, June 1992.

[CP95] K. Chaudhary and M. Pedram. Computing the area versus delay trade-off curves in technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(12):1480–1489, December 1995.

[CW96] J. Cong and C. Wu. An improved algorithm for performance optimal technology mapping with retiming in LUT-based FPGA design. In *Proceedings of IEEE International Conference on Computer Design*, pages 572–578, October 1996.

[CW98a] J. Cong and C. Wu. An efficient algorithm for performance-optimal FPGA technology mapping with retiming. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(9):738–748, September 1998.

[CW98b] J. Cong and C. Wu. Optimal FPGA mapping and retiming with efficient initial state computation. In *Proceedings of 35th Design Automation Conference*, pages 330–335, June 1998.

[DBG+84] J. Darringer, D. Brand, J. Gerbi, W. Joyner, and L. Trevillyan. LSS: A system for production logic synthesis. *IBM Journal of Research and Developments*, 28(5):537–545, September 1984.

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

[DJCM89] P. Das, P. Johannes, L. Claesen, and H. De Man. Hierarchical timing view generation including accurate modeling for false paths. In *Proceedings of IEEE 1989 Custom Integrated Circuits Conference*, pages 13.3.1–13.3.4, May 1989.

[DKM91] S. Devadas, K. Keutzer, and S. Malik. Delay computation in combinational logic circuits: Theory and algorithms. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pages 176–179, November 1991.

[DKM93] S. Devadas, K. Keutzer, and S. Malik. Computation of floating mode delay in combinational circuits: Theory and algorithms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(12):1913–1923, December 1993.

[DKMW93] S. Devadas, K. Keutzer, S. Malik, and A. Wang. Computation of floating mode delay in combinational circuits: Practice and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(12):1924–1936, December 1993.

[DKMW94] S. Devadas, K. Keutzer, S. Malik, and A. Wang. Certified timing verification and the transition delay of a logic circuit. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(3):333–342, September 1994.

[FS94] A. Farrahi and M. Sarrafzadeh. Complexity of the lookup-table minimization problem for FPGA technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(11):1319–1332, November 1994.

[GBdGH86] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel. Socrates: A system for automatically synthesizing and optimizing combinational logic. In *Proceedings of 23rd Design Automation Conference*, pages 79–85, June 1986.

[GLH+94] J. Grodstein, E. Lehman, H. Harkness, H. Touati, and B. Grundmann. Optimal latch mapping and retiming within a tree. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 242–245, November 1994.

[GLRKCI94] C. T. Gray, W. Liu, and H.-Y. Hsieh R. K. Cavin III. Circuit delay calculation considering data dependent delays. *Integration, the VLSI Journal*, 17(1):1–23, August 1994.

[Goe81] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Transactions on Computers*, C-30(3):215–222, March 1981.

[Hit82] R. B. Hitchcock. Timing verification and the timing analysis program. In *Proceedings of 19th Design Automation Conference*, pages 594–604, June 1982.

[HPS91] S.-T. Huang, T.-M. Parng, and J.-M. Shyu. A new approach to solving false path problem in timing analysis. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pages 216–219, November 1991.

[HPS93] S.-T. Huang, T.-M. Parng, and J.-M. Shyu. A polynomial-time heuristic approach to approximate a solution to the false path problem. In *Proceedings of 30th Design Automation Conference*, pages 118–122, June 1993.

[HPS94] S.-T. Huang, T.-M. Parng, and J.-M. Shyu. Timed Boolean calculus and its applications in timing analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(3):318–337, March 1994.

[HPS96] S.-T. Huang, T.-M. Parng, and J.-M. Shyu. A polynomial-time heuristic approach to solving the false path problem. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 43(5):386–396, May 1996.

[HSC82] R. B. Hitchcock, G. L. Smith, and D. D. Cheng. Timing analysis for computer hardware. *IBM Journal of Research and Development*, 26(1):100–105, January 1982.

[JCM92] P. Johannes, L. Claesen, and H. De Man. Performance through hierarchy in static timing verification. In *Proceedings of IFIP 12th World Congress*, pages 703–709, September 1992.

[JCM93] P. Johannes, L. Claesen, and H. De Man. On the use of reconvergence analysis for efficient hierarchical static sensitizable path analysis. In *Proceedings of TAU 93: ACM/IEEE International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1993.

[Joh93] P. Johannes. *Delay Characterization and Hierarchical Timing Verification for Synchronous Circuits*. PhD thesis, Katholieke Universiteit Leuven, Belgium, January 1993.

[Jou83] N. P. Jouppi. TV: An nMOS timing analyzer. In R. Bryant, editor, *Proceedings of the Third Caltech Conference on Very Large Scale Integration*, pages 71–85. Computer Science Press, 1983.

[KB92] A. Kuehlman and R. A. Bergamaschi. Timing analysis in high-level synthesis. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 349–354, November 1992.

[Keu87] K. Keutzer. DAGON: Technology binding and local optimization by DAG matching. In *Proceedings of 24th Design Automation Conference*, pages 617–623, June 1987.

[Kir97] D. A. Kirkpatrick. *The Implications of Deep Sub-micron Technology on the Design of High Performance Digital VLSI Systems*. PhD thesis, University of California, Berkeley, December 1997.

[KM95] N. Kobayashi and S. Malik. Delay abstraction in combinational logic circuits. In *Proceedings of the ASP-DAC 95 / CHDL 95 / VLSI 95*, pages 453–458, August 1995.

[KM97] N. Kobayashi and S. Malik. Delay abstraction in combinational logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(10):1205–1212, October 1997.

[KMS91] K. Keutzer, S. Malik, and A. Saldanha. Is redundancy necessary to reduce delay? *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(4):427–435, April 1991.

[KR89] K. Keutzer and D. Richards. Computational complexity of logic synthesis and optimization. In *Proceedings of International Workshop on Logic Synthesis*, May 1989.

[Lar92] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computers-Aided Design of Integrated Circuits and Systems*, 11(1):4–15, January 1992.

[LB94] W. K. C. Lam and R. K. Brayton. *Timed Boolean Functions: A Unified Formalism for Exact Timing Analysis*. Kluwer Academic Publishers, 1994.

[LBSV93] W. K. C. Lam, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Circuit delay models and their exact computation using timed Boolean functions. In *Proceedings of 30th Design Automation Conference*, pages 128–134, June 1993.

[LP93] I. Levin and R. Y. Pinter. Realizing expression graphs using table-lookup FPGAs. In *Proceedings of the European Design Automation Conference*, pages 306–311, September 1993.

[LS83] C. E. Leiserson and J. B. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, 1983.

[LS91] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[LWGH95] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness. Logic decomposition during technology mapping. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 264–271, November 1995.

[LWGH97] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness. Logic decomposition during technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):813–834, August 1997.

[MB91] P. C. McGeer and R. K. Brayton. *Integrating Functional and Temporal Domains in Logic Design.* Kluwer Academic Publishers, 1991.

[MKLC89] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The Transduction method – design of logic network based on permissible functions. *IEEE Transactions on Computers*, 38(10):1404–1424, October 1989.

[MSBSV93] P. C. McGeer, A. Saldanha, R. K. Brayton, and A. Sangiovanni-Vincentelli. Delay models and exact timing analysis. In T. Sasao, editor, *Logic Synthesis and Optimization*, pages 167–189. Kluwer Academic Publishers, 1993.

[MSS+92] P. C. McGeer, A. Saldanha, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. Delay models and sensitization criteria in the false path problem. Technical Report UCB/ERL M92/63, University of California, Berkeley, 1992.

[MZL96] Y. Min, Z. Zhao, and Z. Li. An analytical delay model based on Boolean processes. In *Proceedings of 9th International Conference on VLSI Design*, pages 162–165, January 1996.

[NCGM92] S. Note, F. Catthoor, G. Goossens, and H. J. De Man. Combined hardware selection and pipelining in high-performance data-path design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(4):413–423, April 1992.

[NST+82] M. Nomura, S. Sato, N. Takano, T. Aoyama, and A. Yamada. Timing verification system based on delay time hierarchical nature. In *Proceedings of 19th Design Automation Conference*, pages 622–628, June 1982.

[Ous83] J. K. Ousterhout. Crystal: A timing analyzer for nMOS VLSI circuits. In R. Bryant, editor, *Proceedings of the Third Caltech Conference on Very Large Scale Integration*, pages 57–69. Computer Science Press, 1983.

[PL96] P. Pan and C. L. Liu. Optimal clock period FPGA technology mapping for sequential circuits. In *Proceedings of 33rd Design Automation Conference*, pages 720–725, June 1996.

[PL98] P. Pan and C. L. Liu. Optimal clock period FPGA technology mapping for sequential circuits. *ACM Transactions on Design Automation of Electronic Systems*, 3(3), July 1998.

[Rud89] R. Rudell. *Logic Synthesis for VLSI Design.* PhD thesis, University of California, Berkeley, April 1989. UCB/ERL M89/49.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 42–47, November 1993.

[Sal91] A. Saldanha. *Performance and Testability Interactions in Logic Synthesis.* PhD thesis, University of California, Berkeley, October 1991. UCB/ERL M91/100.

[SBSV93] N. V. Shenoy, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Minimum padding to satisfy short path constraints. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 156–161, November 1993.

[SBSV94] A. Saldanha, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Circuit structure relations to redundancy and delay. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(7):875–883, July 1994.

[SBSV96] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computers-Aided Design of Integrated Circuits and Systems*, 15(9):1167–1176, September 1996.

[SDC94] S.-Z. Sun, D. H. C. Du, and H.-C. Chen. Efficient timing analysis for CMOS circuits considering data dependent delays. In *Proceedings of IEEE International Conference on Computer Design*, pages 156–159, October 1994.

[SDC98] S.-Z. Sun, D. H. C. Du, and H.-C. Chen. Efficient timing analysis for CMOS circuits considering data dependent delays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(6):546–552, June 1998.

[Seg89] C.-J. Seger. A bounded delay race model. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pages 130–133, November 1989.

[She93] N. V. Shenoy. *Timing Issues in Sequential Circuits*. PhD thesis, University of California, Berkeley, December 1993. UCB/ERL M93/97.

[SP94] V. Singhal and C. Pixley. The verification problem for safe replaceability. In *Proceedings of 6th International Conference on Computer-Aided Verification, CAV'94*, pages 311–323, June 1994.

[SS97] M. Sivaraman and A. J. Strojwas. Timing analysis based on primitive path delay fault identification. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 182–189, November 1997.

[SSM+92] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proceedings of IEEE International Conference on Computer Design*, pages 328–333, October 1992.

[SSV90] K. J. Singh and A. L. Sangiovanni-Vincentelli. A heuristic algorithm for the fanout problem. In *Proceedings of 27th Design Automation Conference*, pages 357–360, June 1990.

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

[TKB97] S. Taşıran, Y. Kukimoto, and R. K. Brayton. Computing delay with coupling using timed automata. In *Proceedings of TAU97: ACM/IEEE International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, pages 232–242, December 1997.

[TMBW90] H. Touati, C. W. Moon, R. K. Brayton, and A. Wang. Performance-oriented technology mapping. In W. J. Dally, editor, *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, pages 79–97. MIT Press, April 1990.

[TON83] E. Tamura, K. Ogawa, and T. Nakano. Path delay analysis for hierarchical building block layout system. In *Proceedings of 20th Design Automation Conference*, pages 403–410, June 1983.

[Tou90] H. J. Touati. *Performance-Oriented Technology Mapping*. PhD thesis, University of California, Berkeley, November 1990. UCB/ERL M90/109.

[TSB91] H. J. Touati, H. Savoj, and R. K. Brayton. Delay optimization of combinational logic circuits by clustering and partial collapsing. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pages 188–191, November 1991.

[VPMS97] S. V. Venkatesh, R. Palermo, M. Mortazavi, and K. A. Sakallah. Timing abstraction of intellectual property blocks. In *Proceedings of Custom Integrated Circuit Conference*, pages 99–102, May 1997.

[Wat97] Y. Watanabe. Private communication, October 1997.

[Yal97a] H. Yalcin. *Hierarchical Timing Analysis of Digital Circuits*. PhD thesis, University of Michigan, 1997.

[Yal97b] H. Yalcin. Private communication, March 1997.

[Yal98] H. Yalcin. Private communication, June 1998.

[YH95] H. Yalcin and J. P. Hayes. Hierarchical timing analysis using conditional delays. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 371–377, November 1995.

[YHS96] H. Yalcin, J. P. Hayes, and K. A. Sakallah. An approximate timing analysis method for datapath circuits. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 114–118, November 1996.