Monographs in Computer Science

Editors

David Gries
Fred B. Schneider

Advisory Board
F.L. Bauer
S.D. Brookes
C.E. Leiserson
M. Sipser

Janusz A. Brzozowski
Carl-Johan H. Seger

Asynchronous Circuits

With 212 Figures

With a Foreword by Charles E. Molnar

Springer-Verlag
New York Berlin Heidelberg London Paris
Tokyo Hong Kong Barcelona Budapest

Janusz A. Brzozowski
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
CANADA

Carl-Johan H. Seger
Department of Computer Science
University of British Columbia
Vancouver, British Columbia V6T 1Z4
CANADA

Series Editors:
David Gries
Department of Computer Science
Cornell University
Upson Hall
Ithaca, NY 14853-7501
USA

Fred B. Schneider
Department of Computer Science
Cornell University
Upson Hall
Ithaca, NY 14853-7501
USA

Library of Congress Cataloging-in-Publication Data
Brzozowski, Janusz A.
Asynchronous circuits / Janusz A. Brzozowski, Carl-Johan H. Seger.
p. cm. — (Monographs in computer science)
Includes bibliographical references and index.
ISBN-13: 978-1-4612-8698-1
1. Asynchronous circuits. I. Seger, Carl-Johan H. II. Title.
III. Series.
TK7868.A79B79 1994
621.39'5—dc20 94-42873

Printed on acid-free paper.

© 1995 Springer-Verlag New York, Inc.
Softcover reprint of the hardcover 1st Edition 1995

All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use of general descriptive names, trade names, trademarks, etc., in this publication, even if the former are not especially identified, is not to be taken as a sign that such names, as understood by the Trade Marks and Merchandise Marks Act, may accordingly be used freely by anyone.

Production managed by Natalie Johnson; manufacturing supervised by Genieve Shaw.
Photocomposed using the authors' LaTeX files.

9 8 7 6 5 4 3 2 1
ISBN-13: 978-1-4612-8698-1 e-ISBN: 978-1-4612-4210-9
DOI: 10.1007/978-1-4612-4210-9

To Grażyna and Sheila

Foreword

The dramatic surge of academic and industrial interest in asynchronous design over the past decade has engaged workers with diverse talents using ideas from disciplines such as programming theory, process algebras, advanced silicon technology, system theory, and classical digital circuit theory. Spanning these ideas presents a serious difficulty to those new to the area, and to those more experienced workers who wish to broaden the ideas and skills that they can bring to bear on the many difficult problems that need to be solved if asynchronous circuits are to become a major new frontier of computer design.

Brzozowski and Seger have made a Herculean effort to collect and organize a uniquely broad yet thorough and rigorous collection of basic ideas, mathematical models, and techniques within a consistent analytical theme. Their approach, while rooted in classical switching theory, begins by exposing and explaining the limitations and inadequacies of classical methods in the face of modern design needs, and extends and refines the classical approaches using modern concepts and models drawn from other areas. A major strength of their treatment is that it provides a firm foundation for understanding the behavior of asynchronous circuits from the bottom up, rather than focusing on a narrower approach to the specification and synthesis of particular classes of circuits. In this sense this book is complementary to other recent treatments; thus its message seems likely to prove enduring as design methods and styles rapidly evolve.

Charles E. Molnar
Washington University
St. Louis, Missouri, USA

Preface

An asynchronous circuit is a digital circuit in which each component reacts to changes on its inputs as these changes arrive, and produces changes on its outputs when it concludes its computation. In essence, all digital circuits can be viewed as being asynchronous. A "synchronous" circuit is simply one designed according to special design rules and operated under special assumptions about its environment. In particular, in a synchronous circuit one or more signals are designated as "clocks." The structure of a synchronous circuit must be such that every closed path contains a "state-holding" element (latch or flip-flop) controlled by the clock; thus the circuit consists of combinational circuits alternating with state-holding elements. Restrictions are placed on the circuit inputs, including the clocks, and on the delays of the combinational parts. For example, the combinational logic must be stable for some time (the "setup" time) before a clock change, and for some time (the "hold" time) after the clock change. Not surprisingly, these severe restrictions make synchronous circuits easy to understand and to design. It should be noted, however, that a synchronous circuit operating in an environment that violates some of the design assumptions must be treated as asynchronous.

Although asynchronous circuits date back to the early 1950s, most of the digital circuits in use today are synchronous, except for some small asynchronous interface parts. Traditionally, asynchronous circuits have been viewed as difficult to understand and design. Consequently, the design of interface circuits has become almost an art, learned on the job through trial and error.

Recently, there has been a great surge of interest in asynchronous circuits. This interest stems partly from an increase in (asynchronous) communication activity in digital circuits, and partly from a desire to achieve higher performance with lower energy consumption and design cost. Also, the development of several new asynchronous design methodologies has made the design of much larger and more complex circuits possible.

Asynchronous design presents a serious challenge to the designer. On the one hand, it has (among other advantages) the following potential benefits:[1]

- **Increased speed:** Each computation is completed in the time needed for that particular computation, and not for the worst case.
- **Reduced power consumption:** In synchronous circuits, clock lines

[1] A comprehensive discussion of the pros and cons of asynchronous design is given in Chapter 15.

have to be toggled and circuit nodes have to be precharged and discharged, even in parts unused in the current computation. Transitions in asynchronous circuits need occur only in parts involved in the current computation.

- **Manageability of metastable phenomena:** Elements that guarantee mutual exclusion or synchronize external signals with a clock are subject to metastability—an unstable equilibrium in which a circuit can remain for an unbounded amount of time. Since all the elements in synchronous circuits must have bounded response times, occasional failures caused by metastability are unavoidable. Asynchronous circuits can wait until the mutual exclusion element leaves the metastable state.

On the other hand, the potential advantages above have remained by and large potential. Asynchronous circuits are more difficult to design in an ad hoc fashion than are synchronous circuits. Asynchronous communication protocols increase the computation time, and involve additional circuitry. The existing computer-aided design tools and implementation alternatives available for synchronous circuits either cannot be used at all in asynchronous design or require extensive modifications.

Recently, the above-mentioned state of affairs has begun to change with the development of new synthesis approaches that make design of asynchronous circuits less of an art and more of an algorithm. Also significant new insight has been gained into the theory of asynchronous circuits. In view of these developments, there is considerable reason for optimism that asynchronous design will be able to achieve many of its potential advantages.

This book provides a comprehensive theory of asynchronous circuits, including modeling, analysis, simulation, specification, verification, and an introduction to design. It is intended as a reference for computer scientists and engineers involved in research and development of asynchronous designs. It is also suitable as a text for a graduate course in asynchronous circuits, and has been used in courses at the Universities of British Columbia, Waterloo, and Western Ontario. Except for requiring some mathematical maturity and some basic knowledge of logic design, the book is self-contained.

The book is organized as follows: After an introductory first chapter intended to motivate the reader to study asynchronous phenomena, we give some mathematical background material in Chapter 2. Because delays play a crucial role in digital circuits, they are discussed next, in Chapter 3. Chapter 4 reviews the basic properties of gate circuits and describes our mathematical model of gate circuits, along with several "network" models used for deriving circuit behaviors. MOS transistor circuits are treated in a similar way in Chapter 5, where several switch-level models are described.

Chapter 6 contains a formalization of the classical binary analysis methods used to detect races and hazards in digital circuits. These methods are based on the assumption that component delays are inertial and bounded only from above. Chapter 7 describes ternary simulation, which efficiently provides some of the results of binary analysis. Chapter 8 presents analysis methods based on the realistic assumption that component delays are bounded from below as well as from above. The computational complexity of various analysis methods is discussed in Chapter 9. Chapter 10 provides the background material on finite automata and regular languages that is necessary for the specification of asynchronous behaviors. Mathematical definitions of specifications and implementations of asynchronous behaviors are discussed in Chapters 11 and 12. Chapter 13 discusses the limitations of the models that use delays bounded only from above. Chapter 14 describes symbolic methods, which provide efficient analysis and verification tools for asynchronous circuits. Finally, Chapter 15 contains a comprehensive survey of asynchronous design methods.

Much of the material in this book has appeared previously only in technical journals, conferences, or theses, but has not been treated in a coherent formalism in any book. Moreover, several results have never been published before, having been developed especially for this book.

Acknowledgments

The authors wish to express their gratitude to many people whose support made this book possible and whose comments led to significant improvements.

To begin with, we thank Martin Rem for suggesting that this material deserved to be presented in the form of a book, and Grzegorz Rosenberg for encouraging us to write a survey article on this topic for the Bulletin of the European Association for Theoretical Computer Science. Without these encouragements, we would not have undertaken this task.

We sincerely thank Scott Hauck for permitting us to use his survey paper [59] as a basis for Chapter 15. Although we have considerably modified the survey in order to incorporate it into our book, we found it very convenient to have a single source with an overview of many papers on asynchronous design. We also thank Scott for useful comments on our adaptation of his paper.

A number of people provided technical criticisms of early versions of the book. Among them, Charles Molnar has played a very major role. He has provided us with insightful, constructive comments that helped us to solve a number of technical, stylistic, and notational problems. We are most grateful for his very professional advice, and for his continued interest in our work. We thank Jo Ebergen for his contributions to, among other things, the technical development of network models and a better definition of

"outcome" of a transition, and for his numerous suggestions for improvements. We acknowledge many fruitful discussions regarding symbolic verification techniques and symbolic timing analysis with Mark Greenstreet. We are indebted to Tom Verhoeff for detailed technical comments concerning our model of asynchronous behaviors.

We thank Helmut Jürgensen for his comments on several chapters. We also thank Bill Coates, Luigi Logrippo, Huub Schols, and Michael Yoeli for their comments on our work on behaviors, and Steve Burns and Ting Fang for comments on Chapter 15.

We wish to express our gratitude to several graduate students from the Maveric Group at the University of Waterloo and from the Integrated Systems Design Laboratory at the University of British Columbia. Radu Negulescu made valuable technical contributions to a number of problems, especially those regarding Chapter 11, as well as providing useful comments on motivation, style, and presentation. Robert Black contributed extensive improvements to the book, and corrected many errors. Sylvain Gingras provided much of the material on the analysis of networks using the ideal-delay model. Igor Benko, Robert Berks, Robert Black, Radu Negulescu, and Richard C.-J. Shi frequently acted as a critical audience when the first author presented early versions of new definitions and results. Scott Hazelhurst and Andrew Martin served as constructive readers as well as critical sounding boards for many of the original ideas of Chapter 14.

The second author would like to acknowledge the stimulating, and often challenging, work environment provided by the faculty and students in the Integrated Systems Design Laboratory at the University of British Columbia.

The authors gratefully acknowledge the financial support of the research related to this book provided by a grant from the Information Technology Research Centre of Ontario, a fellowship from the British Columbia Advanced Systems Institute, and grants OGP0000871 and OGP0109688 from the Natural Sciences and Engineering Research Council of Canada.

J. A. Brzozowski
Waterloo, Ontario, Canada

C-J. H. Seger
Vancouver, British Columbia, Canada

Contents

Chapter 1

Introductory Examples

Digital circuits are normally designed as networks of interconnected components. There are two basic approaches to the coordination of the activities of such components: synchronous and asynchronous.

In a synchronous circuit the "state-holding" components operate under the control of a common periodic signal, called "clock." All the operations must take place at the correct time within each clock cycle. Thus, for example, certain data is required to be ready for some minimum amount of time (the "setup" time) before the clock becomes active, and should be held constant for some minimum amount of time (the "hold" time) while the clock is active, in order to be properly interpreted. The synchronous mode of operation greatly simplifies circuit design. Consequently, most digital circuits are synchronous. As designs grow larger and larger, however, it becomes increasingly more difficult to distribute the clock to all the parts of the system at the same time.

In contrast with the clocked operation of a synchronous circuit, asynchronous coordination of activity is performed through some kind of "handshaking" protocol among the communicating components. These communications take place only when required by the current computation. Here lies both the strength and the weakness of asynchronous design: On the one hand, there is a potential for increasing the average speed of the computations and lowering the power consumption. On the other hand, there is a considerable overhead associated with the handshaking protocols.

To ensure the correct setup and hold times in synchronous circuits, the designer must make certain assumptions about the delays of the various components and interconnecting wires. Synchronous design textbooks rarely stress the importance of these assumptions. In contrast to this, in asynchronous circuits, the absence of a clock makes the designer much more aware of the presence and importance of the various circuit delays. In this chapter we present four introductory examples, the main point of which is to demonstrate the important role of delays in various synchronous and asynchronous digital circuits, and the serious consequences that may arise if delays are not properly taken into account. Much of this book is devoted to the study of the effects of delays on circuit operation, and we will revisit in later chapters many of the phenomena that are discussed in these examples.

The first example illustrates the difficulty of computing the maximum delay of a combinational gate circuit. In the second example, we analyze a well-known clocked gate circuit, that of an RS flip-flop. We show that the circuit may function improperly if a gate delay is too large. Our third example considers a combinational dynamic CMOS circuit, which operates under the control of a clock. This circuit functions properly under the assumption that the components have some specific delay values, but malfunctions if the actual delay magnitudes deviate slightly from their nominal ones. The final example describes the design—with the aid of classical techniques—of a commonly used asynchronous divide-by-two counter. An analysis of the resulting circuit shows that it may not behave as desired, unless one imposes some restrictions on the relative sizes of the delays of the gates in the circuit. This points out a flaw in the classical, and still commonly taught, design techniques.

How To Read This Chapter

Each example begins with a short motivating paragraph and ends with a short concluding paragraph. The reader who is an experienced logic designer is quite likely to have seen such examples before. For such a reader it may suffice to look at the motivating and concluding paragraphs of most of the examples, for the amount of detail provided would likely be found boring. The reader who is not an expert in logic design, but who has had an introductory course in logic design (or equivalent experience), is not likely to have seen these examples. We provide enough detail to permit such a reader to follow the arguments presented. The CMOS example requires some knowledge of transistor circuits; the reader unfamiliar with such circuits should read Section 5.1 first. Finally, the reader who has no familiarity with logic design is advised to omit this chapter on first reading. Instead, that reader should first invest some time and effort in Chapter 2 for some mathematical background, Chapter 3 for an introduction to properties of delays, Chapter 4 for an introduction to gate circuits, and Chapter 5 for an introduction to transistor circuits. When some of that material is absorbed, the examples of Chapter 1 may be better appreciated.

1.1 Logic Gates

Before proceeding to the examples, we briefly define some notation and terminology used in connection with gates. The reader interested in more details concerning basic aspects of logic design may wish to consult an introductory text like [25, 77, 88, 94].

The two logic values are denoted by 0 and 1. A *Boolean function* of n variables is any mapping f of $\{0,1\}^n$ into $\{0,1\}$. A *gate* is a physical device intended to implement a Boolean function. Some commonly used

TABLE 1.1. Common Boolean functions.

X_1X_2	ID	NOT	OR	NOR	AND	NAND	XOR
0 0	0	1	0	1	0	1	0
0 1	0	1	1	0	0	1	1
1 0	1	0	1	0	0	1	1
1 1	1	0	1	0	1	0	0
	X_1	$\overline{X_1}$	$X_1 + X_2$	$\overline{X_1 + X_2}$	$X_1 * X_2$	$\overline{X_1 * X_2}$	$X_1 \oplus X_2$

one- and two-input Boolean functions are defined in Table 1.1. The name of the function is given above its column vector of values, and a Boolean expression corresponding to the function is given below the column vector.

Gate symbols corresponding to the functions of Table 1.1 are given in Figure 1.1. The identity function ID can be realized by a wire connecting the input to the output; hence, there is no special gate symbol for this function.

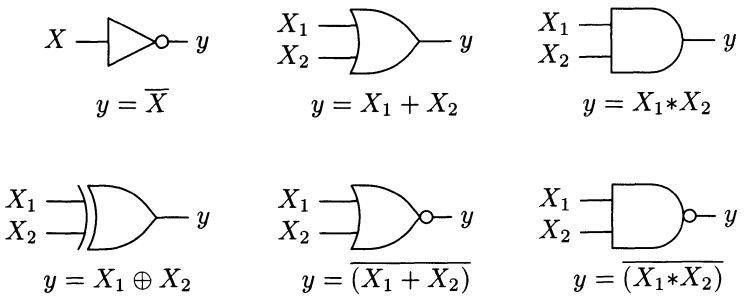


FIGURE 1.1. One- and two-input gates.

1.2 Performance Estimation

Motivation Our first example considers the common problem of finding an upper bound D on the delay of a combinational gate circuit. One approach to this problem is to assume an upper bound on the delay of each gate in the circuit, and to add up all these bounds in every input-to-output path. The largest such sum is then taken as the upper bound D. Our example shows that this approach may overestimate D in case the path with the largest sum is a "false path." A second approach is to simulate the circuit using some nominal values for the gate delays. Our example demonstrates that this method may underestimate D if some of the gate delays deviate slightly from their nominal values.

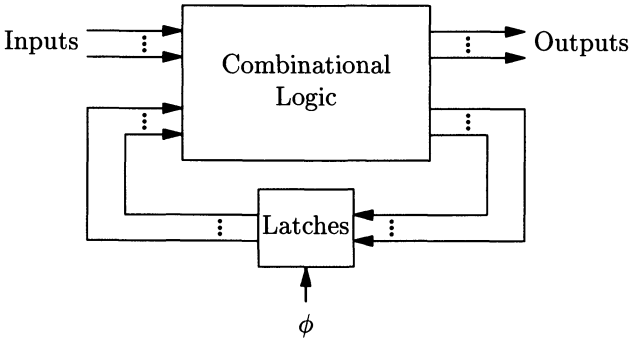


FIGURE 1.2. Generic synchronous circuit.

Synchronous circuits are often depicted as in Figure 1.2. Determining the delay through the combinational logic is a common requirement in designing such circuits. Since the maximum delay of the logic directly determines the maximum clock frequency—and thus the maximum speed of the complete circuit—it is important to compute this delay as accurately as possible. Furthermore, in some technologies [50] the smallest latch is of a "pass-through" type, i.e., when the clock is high the latch is transparent and its output immediately reflects its input. Here the minimum delay through the combinational logic is also of vital importance, since it determines the maximum width of the clock pulse for correct operation.

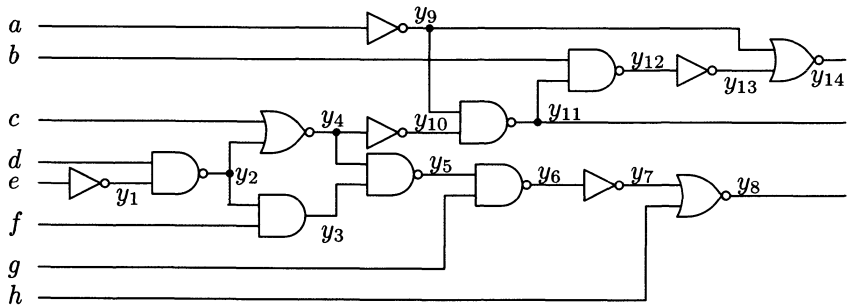


FIGURE 1.3. Combinational circuit containing false paths.

The running example in this subsection is the combinational circuit of Figure 1.3. The question we pose is the following: Assume that the circuit is in a stable state and some of the inputs change. What is the minimum amount of time we must wait until we can guarantee that the outputs (y_8, y_{11}, and y_{14}) have reached their final values? For simplicity, assume that all the gates have the same delay, say 5 ns. For convenience, we refer to "gate with output y_i" simply as y_i.

There are two basic approaches to determining the maximum delay through combinational logic [95]: path analysis and simulation. In standard path analysis the functionality of the combinational logic is completely ignored and only its topology is considered. Hence, the maximum delay is determined from the longest path from an input to an output. In the circuit of Figure 1.3, it is easy to verify that the longest path involves eight gates. Hence, this simple version of path analysis indicates that the maximum delay of the circuit in Figure 1.3 is 40 ns.

Ignoring the functionality of the circuit may result in what is usually called the "false path" problem. A path is said to be false if no input change can propagate through it. To illustrate the concept, consider the path y_1, y_2, y_4, y_{10}, y_{11}, y_{12}, y_{13}, y_{14}, which is the longest path in the circuit of Figure 1.3. Note that a change in input e propagates through this path to the output only if it goes through both y_{11} and y_{14}. For such a change to propagate through y_{11}, gate y_9 must be high. For the same change to propagate through y_{14}, however, y_9 must be low. Since y_9 will be stable by the time a change of e has propagated through y_1, y_2, y_4, and y_{10}, it follows that the change will be stopped either in y_{11} or in y_{14}. Hence, this topologically long path does not determine the maximum delay of the circuit. In summary, false paths may make path analysis overly pessimistic.

Simulation-based approaches encounter different problems. Here, the first difficulty is to know *what* to simulate. Unless sophisticated techniques, like symbolic simulation (see Chapter 14), are applied, exhaustive simulation is often not feasible. Thus only some small "representative" sample of all input changes can be simulated. If the input change that exhibits the longest delay is not simulated, we will underestimate the maximum delay of the combinational circuit. However, even if we do simulate the worst-case input change (through luck, insight, or intelligent choice of input patterns), we might still not compute an accurate estimate of the maximum delay. To illustrate this problem, consider the circuit of Figure 1.3 again. Since there are only eight inputs and the circuit is quite small, exhaustive simulation is feasible. In fact, exhaustive simulation of all present/next input pairs requires only 65,532 input patterns—a relatively small number in the context of circuit simulation. If we simulate all these patterns, we will discover that all the outputs are stable within five gate delays, i.e., within 25 ns.

To explain why 25 ns is the maximum delay according to a simulator, consider the different paths in the circuit. From the false path discussion above, we know that no input change can propagate through both y_{11} and y_{14}. Hence, a change in y_{14} must originate in a or in b. Since the longest path from a or b to y_{14} has five gates, y_{14} will be stable after at most 25 ns. Also, since the longest path from any input to output y_{11} has five gates, y_{11} will be stable after 25 ns. Finally, consider y_8. There are some paths from inputs d and e to y_8 with more than five gates. However, all these paths go through y_2 and y_5. In other words, for a change in d or e to propagate to y_8, there must be a change on y_2 followed later by a change on y_5. This

implies that the change must propagate through y_3 or y_4. There are three cases to consider: the change propagates only through y_3, only through y_4, or through both y_3 and y_4. First, if it propagates through y_3 but not y_4, then c must be high and y_4 must be low. Consequently, one of the inputs to y_5 will be low, stopping the change from propagating. Similarly, if the change propagates through y_4 but not y_3, then y_3 must be low and, again, one of the inputs to y_5 will be low. In the remaining case, the input change propagates through both y_3 and y_4. Since the circuit is started in a stable state, however, and the delays in y_3 and y_4 are identical, it follows that one of y_3 and y_4 will change from high to low and the other will change from low to high. As before, one of the inputs to y_5 will be low. Consequently the paths from d and e to y_8 with more than five gates cannot propagate any input changes.

We have seen that the answer obtained by path analysis is overly pessimistic. Is the simulation-based answer correct? If we assume that the delays of the circuits are exactly as given, i.e., exactly 5 ns each, then 25 ns is indeed the correct delay value. But what if the delay values are not exactly known? In particular, what if we know only that 5 ns is an upper bound on each gate delay? Will this change the answer? Intuitively, it would appear safe to conclude that *speeding up* some gates cannot *slow down* the complete circuit, but here our intuition leads us astray. Consider the case when the delays of y_3 and y_5 are reduced to 2.5 ns. In Figure 1.4 we show the sequence of states that results in this modified circuit when it is started in the stable state[1] $ab\ldots h{\cdot}y_1y_2\ldots y_{14} = 10010110{\cdot}10011010001101$, and the input e changes to 1. The states in Figure 1.4(a)–(i) are reached at times 0, 5, 10, 12.5, 15, 20, 25, 30, and 35, respectively. Clearly, the output y_8 is not stable until 35 ns—significantly later than predicted by the simulation-based maximum delay analysis carried out above! This can be explained as follows. Consider Figure 1.4(c); here, y_3 and y_4 are both unstable, and y_5 is stable. Since y_3 is faster than y_4, y_5 becomes unstable and will change. This results in changes in y_6, y_7, and y_8; this sequence was impossible if all the gates had the same delay. The complete new sequence of events is longer than the longest sequence encountered under the assumption of equal delays.

Conclusions Estimating the maximum delay of a combinational gate circuit is a difficult problem. False paths may overestimate this delay, and simulation methods based on nominal values may underestimate it. Variations in delays may cause significant differences in the maximum delay. The problem here is quite subtle, because a *decrease* in some gate delays may cause the overall circuit delay to *increase*!

---

[1] To simplify notation, we write tuples without parentheses and commas. The centered dot · is used as a separator to divide the input-state component from the internal-state component of the total state.

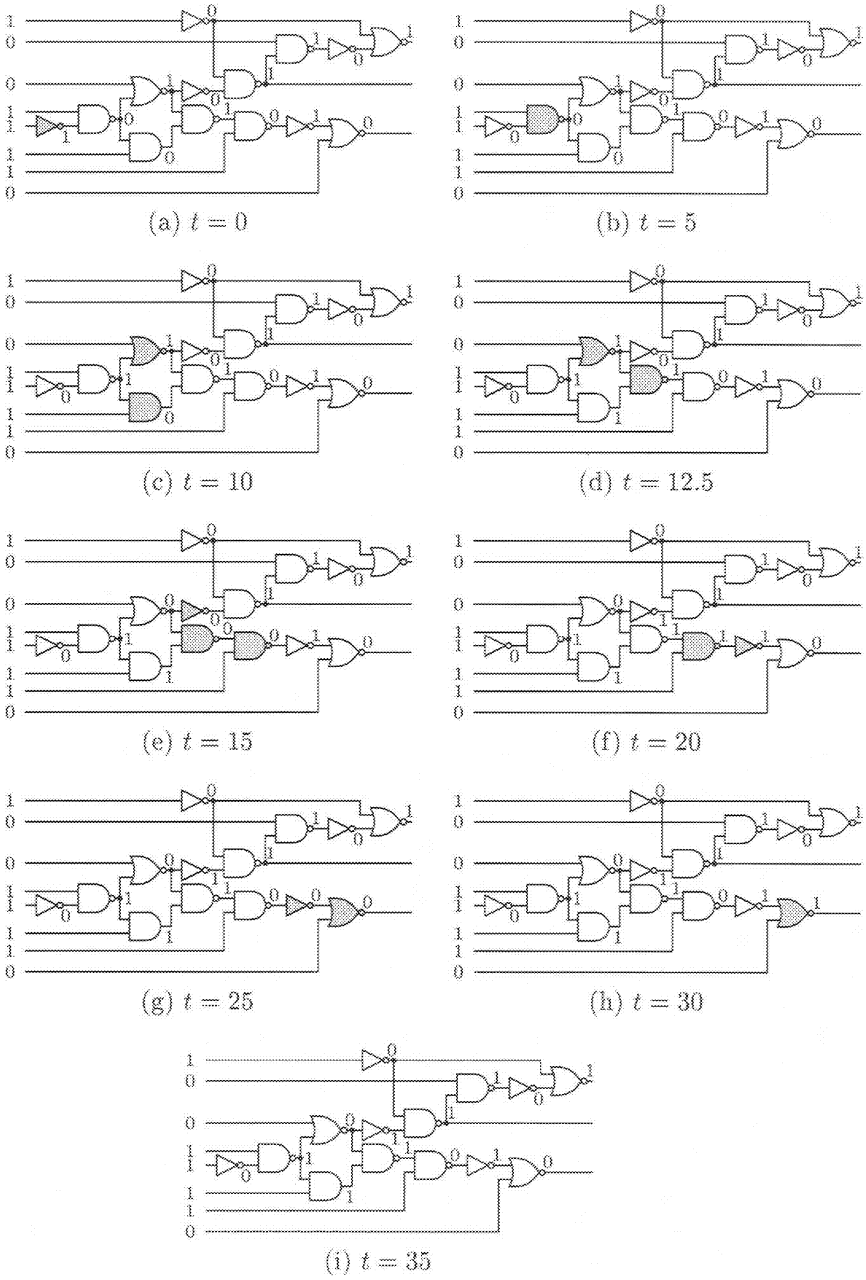


FIGURE 1.4. Speeding up some gates can slow down the circuit.

1.3 RS Flip-Flop

Motivation The purpose of this example is to illustrate how a clocked circuit may malfunction if, because of a slow inverter, a change in the clock reaches one of the components later than it reaches another component.

The circuit shown in Figure 1.5 is a master-slave reset/set (RS) flip-flop. We first describe a transition of this flip-flop assuming that the inverter delay is the same as the delay of the NAND gates. We demonstrate that the flip-flop behaves as expected under this assumption. Next, we assume that the inverter is considerably slower, and show that the behavior may no longer be correct.

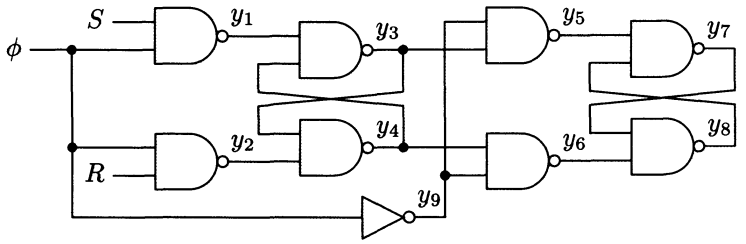


FIGURE 1.5. Master-slave RS flip-flop.

The circuit of Figure 1.5 has data inputs S ("set") and R ("reset") which have the constant values $S = 1$ and $R = 0$ for the purpose of this example. The third input, ϕ, is the clock which is a periodic binary signal. The internal state of the circuit is represented by the 9-tuple of values of the gate variables $y_1, \ldots, y_9$. Gates y_3 and y_4 constitute the master latch of the flip-flop, and y_7 and y_8 constitute the slave latch. The value stored in the master latch is y_3, and the complement of that value is stored in y_4 under stable conditions. Similarly, the value stored in the slave latch is y_7, with y_8 storing the complementary value. We begin with an initial stable total state in which $\phi = 0$. In such a state, the inputs y_1 and y_2 to the master latch both have the value 1. Assuming that the master latch contains the values $y_3y_4 = 01$ (or 10), we see that its state cannot change when $y_1y_2 = 11$; we say that the master latch is "logically isolated." The reader will verify that the values 01 (respectively 10) on y_3y_4 force y_5y_6 to become 10 (respectively 01) when ϕ is 0, and, consequently, y_9 is 1. This, in turn, forces y_7y_8 to become 01 (respectively 10). Thus, it is seen that the slave follows the master.

Consider the transition from stable total state $\phi{\cdot}y_1y_2\ y_3y_4\ y_5y_6\ y_7y_8\ y_9 = 0{\cdot}11\ 01\ 10\ 01\ 1$, where a small space has been inserted between consecutive pairs of y values in order to improve readability. Suppose now that the clock changes to 1. The situation is then as shown in Figure 1.6(a). Both gate y_1

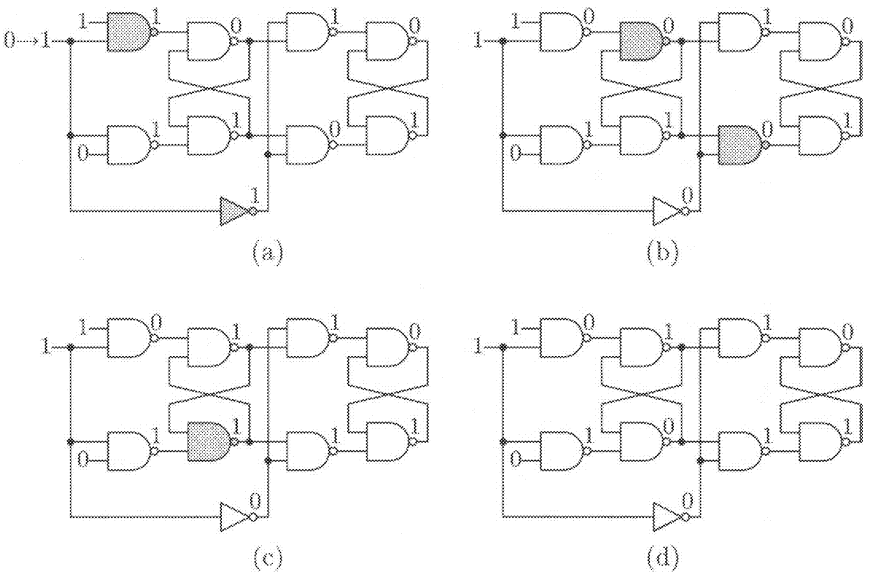


FIGURE 1.6. Sequence of states after clock rises.

and the inverter y_9 are unstable. If they change at the same time, the state of Figure 1.6(b) results, where gates y_3 and y_6 are unstable. When gate y_6 changes to 1, the slave becomes isolated; under these conditions, the state y_7y_8 of the slave cannot change. If y_3 and y_6 change at the same time, we reach the state shown in Figure 1.6(c). Here, only gate y_4 is unstable. When that gate changes, we reach the stable state 1·01 10 11 01 0 shown in Figure 1.6(d), where the master has now been set but the slave retains the old reset value. This is as it should be.

We now reanalyze the transition from state 0·11 01 10 01 1 caused by the change of the clock, but this time we assume that the inverter is slow. The initial state is repeated in Figure 1.7(a). Suppose y_1 changes first, and then y_3—as shown in Figure 1.7(b) and (c)—while the inverter still retains its old value. Gates y_4 and y_5, as well as the inverter, are now unstable; if all three change together, we reach the state of Figure 1.7(d). This time y_5, y_6 and y_7 are unstable; changing these gates leads to state Figure 1.7(e). In that state both gates of the slave latch are unstable. If y_8 changes first, we reach the stable state of Figure 1.7(f), where the slave, as well as the master, has changed.

The sequence of events described above happens if the inverter delay is at least as large as the sum of the delays in gates y_1, y_3, and y_5. While this may not be very likely, the designer should be aware of such possibilities. In Chapter 7 we develop efficient methods for detecting problems that may be caused by gate and wire delays.

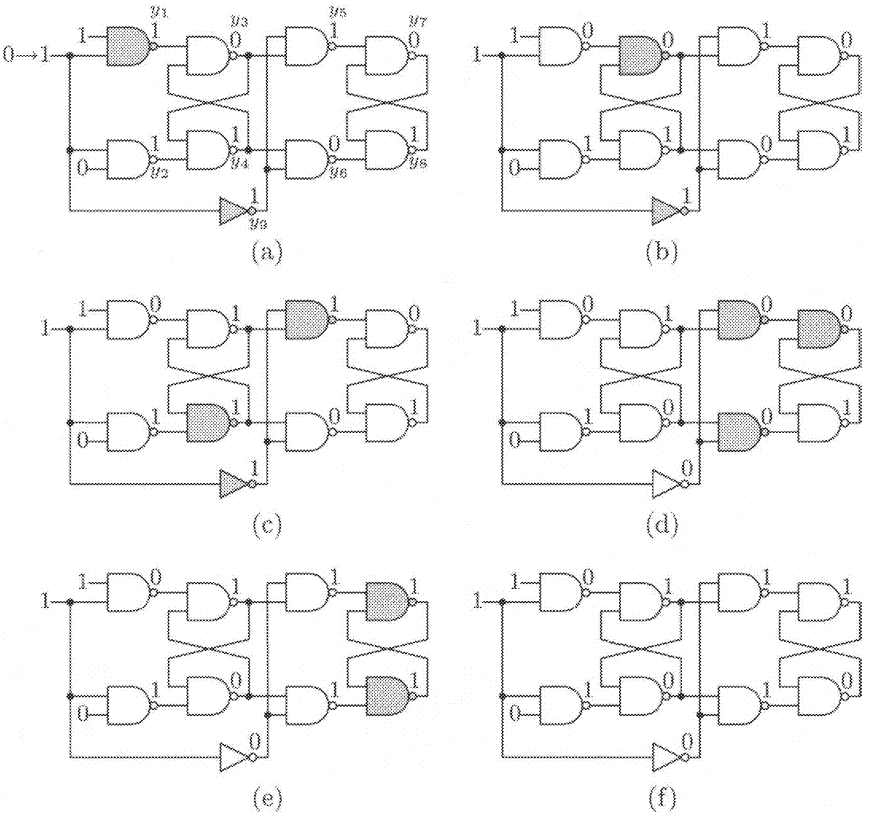


FIGURE 1.7. Possible flip-flop transition with slow inverter.

Conclusions The RS flip-flop uses the clock signal with its master latch and the inverted clock signal with its slave latch. It is the intention that changes in these two signals should take place approximately at the same time. This assumption is violated if the delay of the inverter is too large; then the flip-flop will not operate correctly.

1.4 Dynamic CMOS Logic

Motivation Our third example deals with timing problems in modern VLSI circuits. The example is a dynamic CMOS circuit using the normal precharge/evaluate clock cycle. It is intended to be a combinational circuit implementing a simple Boolean function. We show that it works well with nominal delay values, but fails if some delays deviate from their nominal values, even if the deviation is only 10%.

Figure 1.8 shows a simple dynamic CMOS circuit. The circuit operates under the control of a clock ϕ. The phase of the clock with $\phi = 0$ is called the "precharge" phase, and the phase with $\phi = 1$ is the "evaluate" phase. Assume that only nodes α, β, γ, δ, and *Out* have any significant capacitance associated with them. The circuit is intended to work as follows: When the clock signal ϕ is low, nodes α, β, and δ are all connected through a closed P-transistor to the high voltage V_{dd}, but are isolated from ground by an open N-transistor. Note that this is independent of the values on a, b, c, d, and e. As a result, α, β, and δ will be precharged to V_{dd}. This will cause the N-transistors connected to nodes γ and *Out* to be closed and the P-transistors connected to these nodes to be open. As a result, the voltage on both γ and *Out* will be low. This sequence of events should occur, irrespective of the previous state of the nodes, as long as ϕ is held low long enough to permit nodes α, β, and δ to be precharged. In other words, $\phi = 0$ is a "forcing" signal.

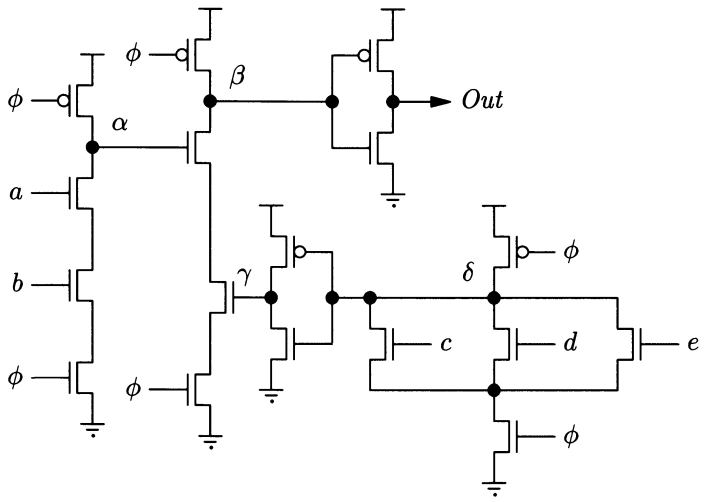


FIGURE 1.8. Dynamic CMOS circuit with timing problem.

In the evaluation phase, the clock signal ϕ becomes high. The behavior of the circuit, and consequently the final value on node *Out*, depends on the values on the input nodes a, b, c, d, and e. Consider first node δ. If at least one of c, d, and e is high, then δ will be connected through a path of closed N-transistors to ground; consequently, the charge stored on δ will be removed. Thus, only if all of c, d, and e are low will δ remain high. If the charge on δ is removed—and thus the voltage becomes low—the voltage on node γ will subsequently become high. Using the same type of argument as for node δ, the reader can verify that node α will remain high only if at least one of a and b is low. Node β will remain isolated—and thus at a high voltage—only if at least one of α and γ is low. Finally, *Out* will become

high only if β becomes low. In summary,

$$Out = \overline{\beta} = \alpha * \gamma = \overline{(a * b)} * \overline{\delta} = \overline{(a * b)} * (c + d + e).$$

Thus when the clock signal becomes high, the circuit should stabilize with the value of *Out* determined by the Boolean expression $\overline{(a * b)} * (c + d + e)$.

To determine whether the circuit behaves as above under various delay distributions, we need to know the rise and fall delays of the nodes of the circuit. In general, determining the delay of a node is a difficult problem since that delay depends on many factors such as the layout of the transistors around the node, the fan-out of the node, the wiring and layout of the transistors loading the node, and manufacturing parameters and defects. For a more complete discussion of the problem of determining circuit delays, the reader is referred to [30, 119]. We simply assume that the delays are given. In particular, suppose that the rise and fall delays of nodes α, β, γ, δ, and *Out* are as given (in nanoseconds) in Table 1.2.

TABLE 1.2. Nominal node delays in the dynamic CMOS circuit.

Node	Rise delay	Fall delay
α	5	14
β	5	4
γ	4	3
δ	5	9
Out	8	6

Consider first the precharge phase, i.e., the interval when $\phi = 0$. For the circuit to be properly precharged, we must keep ϕ low at least long enough for α, β, and δ to become high. For correct operation, however, that is not sufficient. We must also wait until γ becomes low before we raise ϕ. Otherwise, the charge stored at β could be prematurely discharged if ϕ becomes high while γ is still high. We must also wait for *Out* to become low. In summary, ϕ must be kept low for at least $\max\{5, (5+3), (5+6)\} = 11$ ns in order to ensure proper precharging and initialization.

Now turn to the evaluation phase. Since ϕ is kept high during that entire phase, it follows that a precharged node will either stay charged (stay high) or be discharged (change to low). Thus, no precharged node can change more than once during the evaluation phase.

To determine whether the circuit computes the Boolean function $\overline{(a * b)} * (c + d + e)$, we carry out an exhaustive case analysis. Because of symmetry, it suffices to consider the following three cases:

1. Inputs c, d, and e are all low,
2. At least one of c, d, and e is high, and at least one of a and b is low,
3. At least one of c, d, and e is high, and a and b are both high.

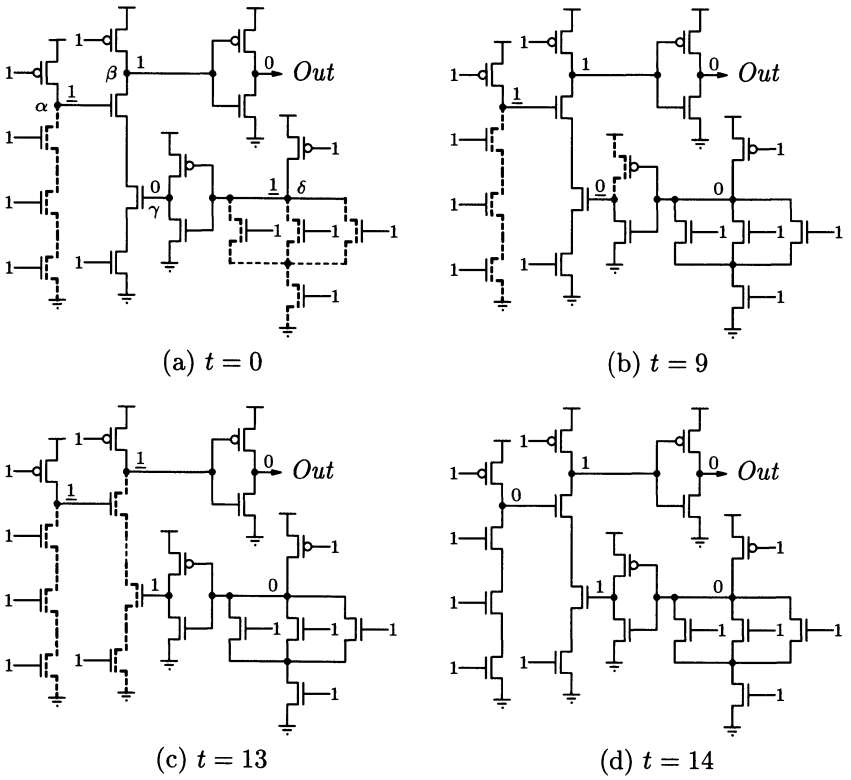


FIGURE 1.9. State sequence of dynamic CMOS circuit.

First, if c, d, and e are all low, it follows that δ remains high; consequently, γ remains low. Hence, β will not be connected to ground regardless of the value on α. Thus, β remains high, forcing *Out* to remain low. Therefore the circuit behaves correctly in this case.

Now consider Case 2. Since one of a and b is low, node α remains high throughout the evaluation phase. Because one of c, d, and e is assumed to be high, δ will be discharged. This will cause γ to become high. When this occurs β will become low, causing *Out* to become high. The circuit is now stable. Hence, if at least one of a and b is low and at least one of c, d, and e is high, *Out* will become high, as required.

Finally, if a, b, and at least one of c, d, and e is high, the circuit will go through the sequence of states shown in Figure 1.9, where c, d, and e are high. We have used the convention that an unstable node's value is underlined and at least one of the closed paths making that node unstable is dashed. Suppose the clock becomes high at time $t = 0$; then α and δ are unstable (Figure 1.9(a)). Since the fall delay of δ (9 ns) is significantly smaller than the fall delay of α (14 ns), the circuit reaches the state shown

in Figure 1.9(b) at $t = 9$, and γ becomes unstable. Since the rise delay of γ is only 4 ns, γ will change (at $t = 13$) before α, despite the fact that it became unstable 9 ns after α. At this point, β is starting to discharge through the closed path to ground, as illustrated in Figure 1.9(c). However, α changes to low at $t = 14$; consequently, β is unstable for only 1 ns. Since the fall delay of β is 4 ns, β remains high. Hence, node *Out* remains low as desired.

Since the circuit produces the expected result for all possible input combinations, we might be tempted to label it "correct." We show, however, that a malfunction may occur if the delays of the nodes depart slightly from the nominal delays of Table 1.2. Since dynamic properties—like temperature, age, and previous values assigned to a node—all affect the delay of a node, a circuit should be robust to changing delay values, in the sense that relatively small changes in the delays should not affect the basic functionality of the circuit.

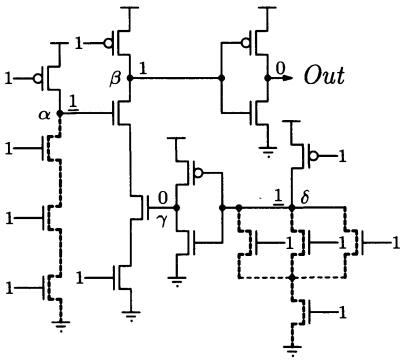
To illustrate the problem with the circuit of Figure 1.8, consider the case when the delays vary within $\pm 10\%$ of the nominal delay values. In such a situation, the delay values may be as shown in Table 1.3. Note that the delays of some nodes are larger than before, whereas the delays of other nodes are smaller. In Figure 1.10 we show the sequence of states that would

TABLE 1.3. Possible delay assignment with less than 10% deviation.

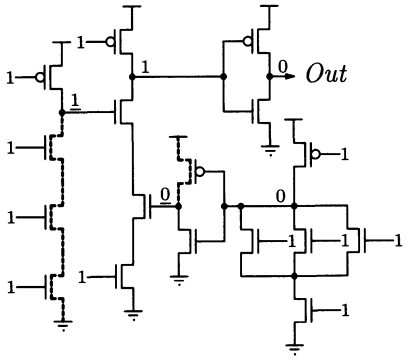
Node	Rise delay	Fall delay
α	5	15.3
β	5	3.6
γ	3.6	3
δ	5	8.1
Out	8	6

result when a, b, c, d, and e are all high. If we assume ϕ becomes high at time 0, the states shown in Figure 1.10(b), (c), (d), and (e) are reached at times 8.1, 11.7, 15.3, and 23.3, respectively. Node δ falls at $t = 8.1$ ns (Figure 1.10(b)). Next, γ rises at time $t = 8.1 + 3.6 = 11.7$ ns, and β becomes unstable (Figure 1.10(c)). All this time node α is unstable, but it does not change because of its large fall delay. This gives β enough time to discharge at $t = 11.7 + 3.6 = 15.3$ ns. At the same time, α finally discharges (Figure 1.10(d)), but it is now too late, and *Out* gets the incorrect value 1 at $t = 15.3 + 8 = 23.3$ ns (Figure 1.10(e)). This malfunction is caused by the fact that the transitions of δ and γ happen to be faster than expected, whereas the transition of α is slower than expected.

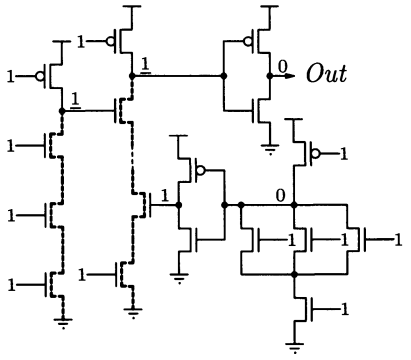
Conclusions This examples illustrates, once again, the danger of considering only nominal delay values in analyzing the behavior of a circuit. In this case, however, the context is switch-level analysis of dynamic CMOS circuits.



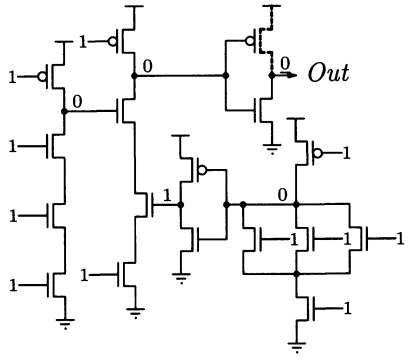
(a) $t = 0$



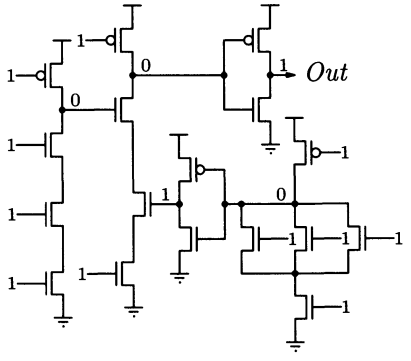
(b) $t = 8.1$



(c) $t = 11.7$



(d) $t = 15.3$



(d) $t = 23.3$

FIGURE 1.10. Timing error in dynamic CMOS circuit.

1.5 Divide-by-2 Counter

Motivation This example illustrates a flaw in the classical design techniques for asynchronous circuits. A circuit designed with these techniques may not function properly under all delay distributions. Thus an analysis is required after a design. We stress this point because many texts in logic design do not mention these difficulties.

Our final example illustrates the classical method of designing asynchronous sequential circuits [66, 67, 135]. We wish to implement a "divide-by-2" counter to operate as follows. The counter is to have one binary input X and one binary output z. In the initial state of the counter, we have $X = z = 0$. As the input X changes from 0 to 1 to 0, repeatedly, the circuit should produce one output change for every two input changes, as shown

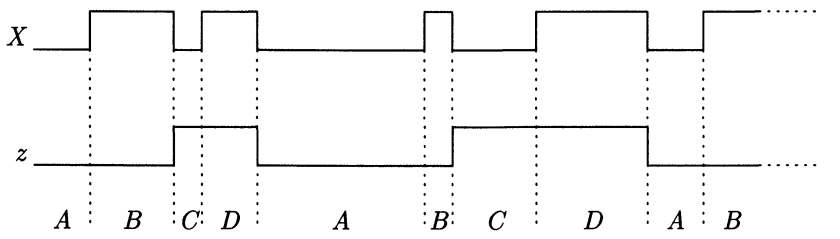


FIGURE 1.11. Waveforms for divide-by-2 counter.

in Figure 1.11. The input is permitted to change at any time, except for the restriction that each new input value lasts long enough for the circuit to stabilize. Also we assume that the rise and fall times in each waveform, as well as delays between input and output changes, are negligible; consequently, they are shown as zero in the figure. If the input waveform were periodic, then the output waveform would also be periodic, but would have twice the period of the input waveform—or half the frequency. For this reason, the circuit is called a divide-by-2 counter.

When we examine the input/output waveforms in Figure 1.11, we see that the circuit goes through four distinct input/output combinations, namely 0/0, 1/0, 0/1, and 1/1; this pattern (ABCD) then repeats. The behavior of the circuit can also be described by a "flow table," which specifies how the circuit's states change. Such a flow table is shown in Figure 1.12 and its interpretation is as follows. The column label specifies the present value of the input X, whereas the row label denotes the present "internal state" p of the circuit. The pair (X, p) is the "total state" of the circuit. The entry (q, z) corresponding to total state (X, p) specifies the next internal state q and the output z. In some states the value of the output is optional, i.e., it is a "don't care." For such states, we leave z unspecified and indicate this by $-$.

p \ X	0	1
A	(A), 0	B , 0
B	C , –	(B), 0
C	(C), 1	D , 1
D	A , –	(D), 1

q, z

FIGURE 1.12. Flow table for counter.

When the present input is 0 and the present internal state is A, the circuit is in its initial total state $(0, A)$. The entry corresponding to this total state indicates that the next internal state is also A and that the output is 0. Whenever the present internal state and the next internal state are the same, we say that the total state of the circuit is "stable," meaning that no internal state change will take place unless the input changes. Stable states are circled in the flow table. Suppose now that the circuit starts in the total state $(0, A)$ and the input changes (to 1). The new total state is $(1, A)$. The next-state entry in column 1, row A is B; this shows that the total state is unstable and the next internal state will become B, provided the input X is kept constant at its new value (1) long enough to permit the circuit to "stabilize." According to the waveforms for the counter, the output in total state $(1, B)$ should still be 0. We also want to avoid any output changes during the time when the circuit is in transition from stable state $(0, A)$ to stable state $(1, B)$; for this reason, the output entry for unstable total state $(1, A)$ is also 0.

Next, consider an input change from stable state $(1, B)$. The circuit will eventually reach stable state $(0, C)$ which has output 1. Thus, during the transition from stable state $(1, B)$ to stable state $(0, C)$, the output should change from 0 to 1. The output entry in unstable total state is left unspecified. This entry will be replaced eventually by either a 0 or a 1 in the circuit that realizes the flow table, but it is best not to introduce any unnecessary constraints too early.

The reader can follow the remaining input changes from stable state $(0, C)$ to stable state $(1, D)$, and from $(1, D)$ back to $(0, A)$; they are similar to the changes just described.

In the next step of the design procedure, we replace the abstract states A, B, C and D by tuples of values of binary "state variables," so that we can implement the circuit with binary logic gates. Two variables are sufficient, and we choose the assignment $A \mapsto 00, B \mapsto 01, C \mapsto 11, D \mapsto 10$. Notice that this assignment implies that, in every state transition, only one variable is required to change. This avoids situations called "races," in which

several variables are unstable at the same time. Since delays associated with the racing variables are not known in general, it is not possible to predict which variables will "win" the race by changing first. Because the subsequent circuit behavior may depend on the choice of the winning variables, many classical design techniques are based on race-free assignments.

When the assignment of state variables is made in the flow table of Figure 1.12, we obtain the table of Figure 1.13. This table is called the "excitation table" because it shows whether or not the state variables are "excited" to change (i.e., are unstable) in any given total state. The output entries are also shown in the excitation table, for convenience. Note that the comma between the state variables and the output separates the two types of variables. The present values of state variables are denoted by y_1 and y_2, whereas the excitations of these variables are Y_1 and Y_2.

y_1y_2 \ X	0	1
00	(00), 0	01 , 0
01	11 , –	(01), 0
11	(11), 1	10 , 1
10	00 , –	(10), 1

Y_1Y_2, z

FIGURE 1.13. Excitation table for counter.

The excitation table leads directly to a circuit, as we now show. The excitations Y_1 and Y_2 are Boolean functions of the input X and the state variables y_1 and y_2. Using standard methods [135], we find the following expressions for the excitations and the output:

$$\begin{aligned} Y_1 &: \ \overline{X}*y_2 + y_1*y_2 + X*y_1, \\ Y_2 &: \ \overline{X}*y_2 + \overline{y_1}*y_2 + X*\overline{y_1}, \\ z &: \ y_1. \end{aligned}$$

For the output, the "don't care" conditions were assigned binary values so as to simplify the resulting Boolean expression. The product y_1*y_2 in the expression for Y_1 is redundant from the logic point of view, but is included for the following reason: When the internal state is 11, the excitation Y_1 is 1 for both values of the input. When the input changes from 0 to 1, however, it is possible for Y_1 to become temporarily 0; this is called a "static hazard." This hazard can be avoided if the redundant product y_1*y_2 is included in the expression for Y_1. The expression for excitation Y_2 has been given a similar redundant product $\overline{y_1}*y_2$. More will be said about hazards in Chapter 7; for now it suffices to point out that the use of hazard-free circuits is required in the classical design methods.

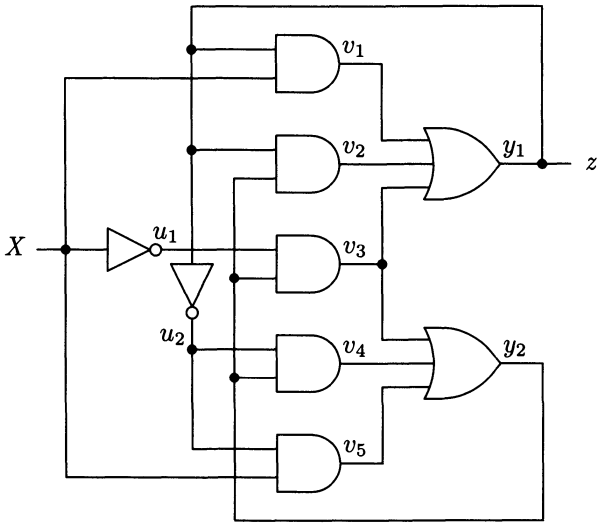


FIGURE 1.14. Gate circuit for counter.

The excitation expressions above are next implemented by two-level AND/OR circuits as shown in Figure 1.14. The variables y_1 and y_2 are identified with Y_1 and Y_2, respectively, and are associated with the same node in the circuit. The interpretation is that y_i is the present state of a node and Y_i is its excitation, for $i = 1, 2$.

The following analysis of the circuit just designed shows that, for some gate delay distributions, it does not behave as intended. Consider the initial stable total state $X = 0, y_1 = 0, y_2 = 0$, and suppose the input changes to 1. The situation is as shown in Figure 1.15(a). Two gates, u_1 and v_5 (shaded in the figure), are unstable in this state. We will assume that the inverter u_1 is very slow; then v_5 changes first, and the new state is as shown in Figure 1.15(b). Next, gate y_2 changes, and we reach the state in Figure 1.15(c). Changing v_3 leads to Figure 1.15(d), and y_1 can change next (along with u_1 and v_4), as shown in Figure 1.15(e). In summary, if the inverter u_1 is slow, the following sequence of signal changes can take place: v_5, y_2, v_3, y_1. This violates the specification given in the flow table, for the output z was not supposed to change during the transition from $(0, A)$ to $(1, B)$. This sequence of events will take place if the delay of inverter u_1 exceeds the sum of the delays of gates v_5, y_2, v_3, and y_1.

One can verify (with some additional work) that, under the stipulated delay conditions, the circuit may eventually reach stable total state $X = 1, y_1 = 1, y_2 = 0$, in which case the final state and output are also incorrect.

Conclusions As our example has shown, one cannot rely on the correctness of a circuit designed using classical methods without performing an elaborate analysis of the effects of the various circuit delays.

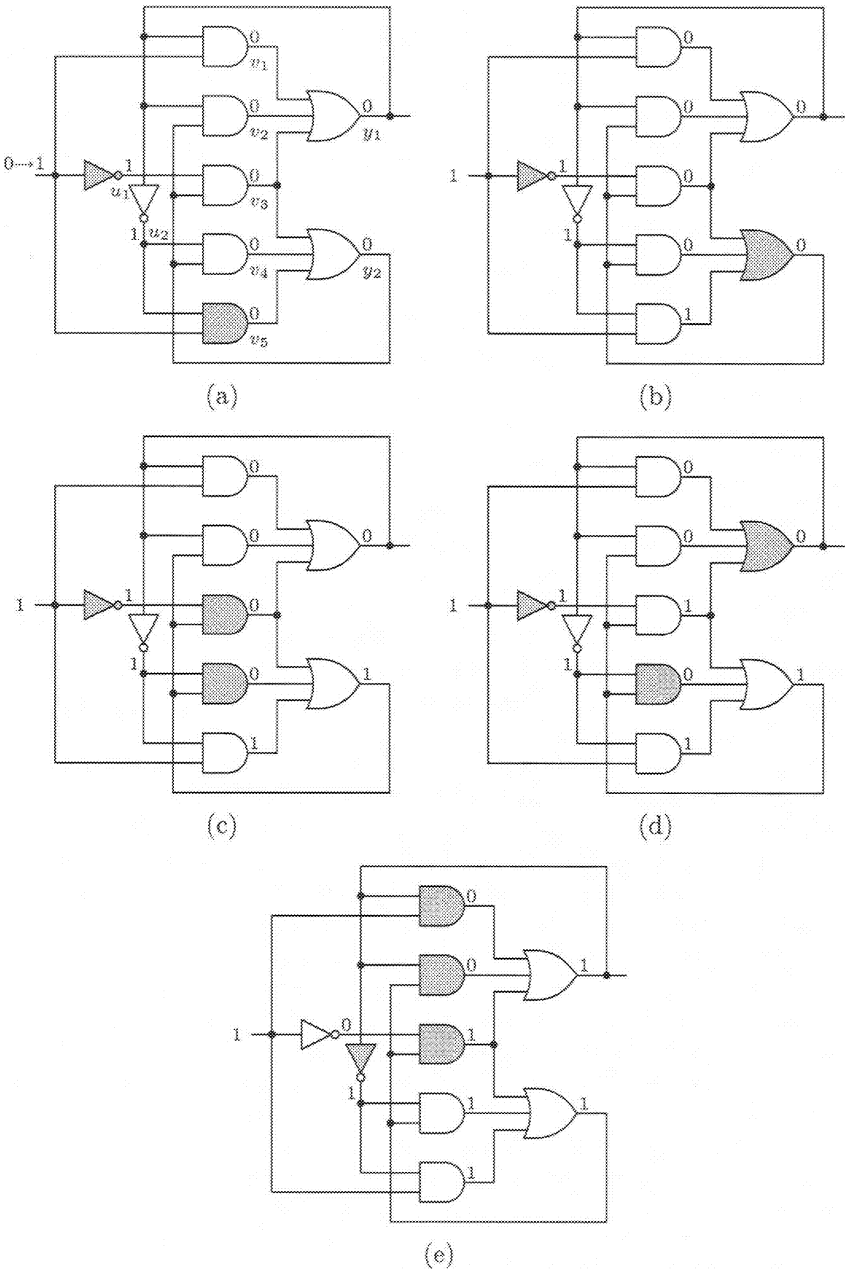


FIGURE 1.15. Possible state sequence for counter.

1.6 Summary

The four examples in this chapter illustrate the importance of delays in synchronous and asynchronous circuits. Various aspects of this topic are treated in greater detail as described below. To begin with, precise mathematical models of delays are introduced in Chapter 3. Similarly, in Chapters 4 and 5 we formalize our models of gate and transistor circuits, respectively.

The issues of performance estimation (as in our first example) and detection of timing problems (as in our CMOS example) are dealt with in Chapters 8 and 14. The techniques developed in these chapters allow us to analyze a circuit assuming that its component delays are bounded by minimum and maximum values, but can take on any value between these bounds. In particular, we show that the circuit of Figure 1.3 will always reach a stable output state within 35 time units, if the delay of each gate is bounded between 1 and 5 time units.

The detection of timing problems in synchronous and asynchronous circuits (as in the examples of the RS flip-flop and the divide-by-2 counter) is dealt with in Chapters 6, 7, and 8. The inherent complexity of these problems is examined in Chapter 9, where the intractability of computing the exact solutions is demonstrated.

The design issues raised by the last example (the divide-by-2 counter) are discussed in Chapters 11–13, and 15. In particular, Chapters 11 and 12 formalize the notion of specification and realization of asynchronous behaviors. In Chapter 13 we show that certain specifications are not realizable unless assumptions about the relative sizes of component delays are made. Finally, modern methods for asynchronous circuit design are surveyed in Chapter 15.

Chapter 2

Mathematical Background

Our goal in this chapter is to establish a mathematical foundation for the following chapters. We provide here brief introductions to set theory, Boolean algebra, ternary algebra, and directed graph theory. The material is presented rather concisely, and the reader may wish to refer to some introductory texts for additional explanations [5, 6, 25, 115]. We point out that, while most of the material in this chapter appears frequently in basic texts on discrete mathematics, this is not the case for the section on ternary algebra.

2.1 Sets and Relations

We use calligraphic letters $\mathcal{A}, \mathcal{B}, \ldots$ to denote sets and lower case letters $a, b, \ldots$ to denote individual elements of sets. Table 2.1 gives the terminology and notation pertaining to sets.

A mapping f of a set $\mathcal{A}$ to a set $\mathcal{B}$ is called a *function from* $\mathcal{A}$ *to* $\mathcal{B}$, written $f: \mathcal{A} \to \mathcal{B}$, if for every element $a \in \mathcal{A}$, there exists a unique element $b \in \mathcal{B}$ such that $f(a) = b$. $\mathcal{A}$ is called the *domain* of f and $\mathcal{B}$ its *codomain.* Let

$$\mathcal{B}' \stackrel{\text{def}}{=} \{b \in \mathcal{B} \mid b = f(a) \text{ for some } a \in \mathcal{A}\},$$

where $\stackrel{\text{def}}{=}$ means "is defined to be." Then $\mathcal{B}'$ is the *image* of $\mathcal{A}$ under f. In case $\mathcal{B} = \mathcal{B}'$, we say that f is *surjective* (or that f is a function from $\mathcal{A}$ *onto* $\mathcal{B}$, or that f is *onto*). Also, f is said to be *injective* (or *one-to-one*) if, for every $a_1, a_2 \in \mathcal{A}$, $a_1 \neq a_2$ implies $f(a_1) \neq f(a_2)$. Finally, f is said to be *bijective* if f is surjective and injective.

By an *n-tuple*, $n \geq 2$, we mean an ordered sequence of n elements (not necessarily distinct). The n-tuple with a_i as its ith element $(i = 1, \ldots, n)$ is written $(a_1, \ldots, a_n)$ or $\langle a_1, \ldots, a_n \rangle$. An *ordered pair* is a 2-tuple.

The *Cartesian product* $\mathcal{A} \times \mathcal{B}$ of two sets $\mathcal{A}$ and $\mathcal{B}$ is the set of all ordered pairs (a, b), where $a \in \mathcal{A}$ and $b \in \mathcal{B}$. The Cartesian product of n sets $\mathcal{A}_1, \ldots, \mathcal{A}_n$ is defined similarly.

A *unary operation* on a set $\mathcal{A}$ is a function from $\mathcal{A}$ to $\mathcal{A}$. A *binary operation* on $\mathcal{A}$ is a function from $\mathcal{A} \times \mathcal{A}$ to $\mathcal{A}$. If $\square$ denotes an arbitrary binary operation on $\mathcal{A}$, we usually write $a \square b$ for $\square((a, b))$.

A *binary relation* R *from* set $\mathcal{A}$ *to* set $\mathcal{B}$ is any subset of $\mathcal{A} \times B$. If $(a, b) \in R$, we write aRb and say that a is related by R to b. A binary relation *on* $\mathcal{A}$ is a binary relation from $\mathcal{A}$ to itself.

TABLE 2.1. Terminology for sets.

Symbol	Meaning
$a \in \mathcal{A}$	a *is an element of* $\mathcal{A}$
$a \notin \mathcal{A}$	a *is not an element of* $\mathcal{A}$
$\{a \in \mathcal{A} \mid a \text{ has property } P\}$	the set of all elements of $\mathcal{A}$ that have property P
$\mathcal{A} = \mathcal{B}$	sets $\mathcal{A}$ and B are equal
$\mathcal{A} \subseteq \mathcal{B}$ or $\mathcal{B} \supseteq \mathcal{A}$	$\mathcal{A}$ *is a subset of* $\mathcal{B}$
$\mathcal{A} \subset \mathcal{B}$ or $\mathcal{B} \supset \mathcal{A}$	$\mathcal{A}$ *is a proper subset of* $\mathcal{B}$
$\emptyset$	the *empty set*
$\mathcal{A} \cap \mathcal{B}$	the *intersection* of $\mathcal{A}$ and $\mathcal{B}$
$\mathcal{A} \cup \mathcal{B}$	the *union* of $\mathcal{A}$ and $\mathcal{B}$
$\mathcal{A} - \mathcal{B}$	the *difference* of $\mathcal{A}$ and $\mathcal{B}$ $\mathcal{A} - \mathcal{B} \stackrel{\text{def}}{=} \{a \mid a \in \mathcal{A} \text{ and } a \notin \mathcal{B}\}$
$\mathcal{A}\Delta\mathcal{B}$	the *symmetric difference* of $\mathcal{A}$ and $\mathcal{B}$ $\mathcal{A}\Delta\mathcal{B} \stackrel{\text{def}}{=} (\mathcal{A} - \mathcal{B}) \cup (\mathcal{B} - \mathcal{A})$
$\mathcal{A} \cap \mathcal{B} = \emptyset$	$\mathcal{A}$ and $\mathcal{B}$ are *disjoint*
$\mathcal{P}(\mathcal{A})$	the *power set* of $\mathcal{A}$, i.e., the set of all subsets of $\mathcal{A}$
$\lvert\mathcal{A}\rvert$	the *cardinality* of $\mathcal{A}$, i.e., the number of elements in $\mathcal{A}$

Given any binary relation R on $\mathcal{A}$, the *composition* of R with R, written RR or R^2, is defined as

$$\{(a, c) \in \mathcal{A} \times \mathcal{A} \mid aRb \text{ and } bRc \text{ for some } b \in \mathcal{A}\}.$$

Furthermore, let $R^1 \stackrel{\text{def}}{=} R$ and $R^{n+1} \stackrel{\text{def}}{=} RR^n$, for $n \geq 1$. The *transitive closure* R^+ of R is defined to be

$$\{(a, b) \mid aR^h b \text{ for some positive integer } h\}.$$

The *reflexive and transitive closure* R^* of R is defined as

$$\{(a, a) \mid a \in \mathcal{A}\} \cup R^+.$$

Let R be a binary relation on $\mathcal{A}$, i.e., $R \subseteq \mathcal{A} \times \mathcal{A}$.

1. R is *reflexive* if aRa for all $a \in \mathcal{A}$.
2. R is *symmetric* if aRb implies bRa for all $a, b \in \mathcal{A}$.
3. R is *antisymmetric* if aRb and bRa implies $a = b$ for all $a, b \in \mathcal{A}$.
4. R is *transitive* if aRb and bRc implies aRc for all $a, b, c \in \mathcal{A}$.

A binary relation on $\mathcal{A}$ which is reflexive, symmetric, and transitive is called an *equivalence relation* on $\mathcal{A}$. A binary relation on $\mathcal{A}$ which is reflexive, antisymmetric, and transitive is called a *partial order* on $\mathcal{A}$.

A *poset* (partially ordered set) is an ordered pair $\langle \mathcal{A}, \leq \rangle$, where $\mathcal{A}$ is a set and $\leq$ is a partial order on $\mathcal{A}$. If $\langle \mathcal{A}, \leq \rangle$ is a poset, $\mathcal{B}$ is a nonempty subset of $\mathcal{A}$, and $a \in \mathcal{A}$, then a is an *upper bound* of $\mathcal{B}$ if $s \leq a$ for all $s \in \mathcal{B}$. An upper bound a of $\mathcal{B}$ is called *least upper bound* of $\mathcal{B}$, written *lub* $\mathcal{B}$, if $a \leq b$ for every upper bound b of $\mathcal{B}$. Clearly, if *lub* $\mathcal{B}$ exists, it is unique.

To illustrate the definitions above, consider the poset $\langle \{0, \Phi, 1\}, \sqsubseteq \rangle$, where the "uncertainty" partial order $\sqsubseteq$ on the set $\{0, \Phi, 1\}$ is defined as follows:

$$0 \sqsubseteq 0,\ 1 \sqsubseteq 1,\ \Phi \sqsubseteq \Phi,\ 0 \sqsubseteq \Phi, \text{ and } 1 \sqsubseteq \Phi,$$

and no other pairs are related by $\sqsubseteq$. Here, the value Φ is used to denote "lack of information" or "uncertainty." On the other hand, both 0 and 1 are "certain" values, and neither of them contains more information than the other one. Thus, for $s, t \in \{0, \Phi, 1\}$, the statement $s \sqsubseteq t$ can be interpreted as t "has at least as much uncertainty" as s. When $s \sqsubseteq t$, we say that s is *covered* by t or that t *covers* s. It is easy to convince oneself that $lub\{0\} = 0$, $lub\{1\} = 1$, and that the *lub* of every other nonempty subset of $\{0, \Phi, 1\}$ is equal to Φ.

2.2 Boolean Algebra

An *algebraic system* is a set $\mathcal{A}$ together with one or more operations on $\mathcal{A}$ which satisfy specific axioms. A *Boolean algebra* is an algebraic system $\mathbf{B} = \langle \mathcal{A}, +, *, \bar{}, 0, 1 \rangle$, where $\mathcal{A}$ is a set, $+$ and $*$[1] are binary operations on $\mathcal{A}$, $\bar{}$ is a unary operation on $\mathcal{A}$, and 0 and 1 are two distinct elements of $\mathcal{A}$ such that all the axioms of Table 2.2 are satisfied.[2] The elements 0 and 1 are called the *universal bounds* of $\mathbf{B}$. Note that, except for B8, the axioms are listed in pairs; one axiom in a pair can be obtained from the other by interchanging 0 and 1, and addition and multiplication. This property of Boolean algebra is called *duality*. Axiom B8 is self-dual, since it does not involve 0, 1, addition, or multiplication.

The smallest Boolean algebra is $\mathbf{B}_0 = \langle \{0, 1\}, +, *, \bar{}, 0, 1 \rangle$, where the underlying set has only two elements, and the operations $+$, $*$, and $\bar{}$ are the OR, AND, and NOT (complement) operations as defined in Table 2.3.

A *Boolean function* of n variables is any function f from $\{0, 1\}^n$ to $\{0, 1\}$, for $n \geq 0$. To each n-tuple $a = (a_1, \ldots, a_n) \in \{0, 1\}^n$, the function f assigns

---

[1] We often represent the multiplication operator by juxtaposition, i.e., we write ab rather than $a*b$.

[2] There are several sets of axioms that can be used to define a Boolean algebra. The one shown in Table 2.2 is chosen for convenience.

TABLE 2.2. Axioms of Boolean algebra.

For all a, b, and c in $\mathcal{A}$, we have:

B1	$a+a = a$	B1′	$a*a = a$
B2	$a+b = b+a$	B2′	$a*b = b*a$
B3	$a+(b+c) = (a+b)+c$	B3′	$a*(b*c) = (a*b)*c$
B4	$a+(a*b) = a$	B4′	$a*(a+b) = a$
B5	$a+0 = a$	B5′	$a*1 = a$
B6	$a+1 = 1$	B6′	$a*0 = 0$
B7	$a+\overline{a} = 1$	B7′	$a*\overline{a} = 0$
B8	$\overline{(\overline{a})} = a$		
B9	$a+(b*c) = (a+b)*(a+c)$	B9′	$a*(b+c) = (a*b)+(a*c)$
B10	$\overline{(a+b)} = \overline{a}*\overline{b}$	B10′	$\overline{(a*b)} = \overline{a}+\overline{b}$

TABLE 2.3. The operations $+$, $*$, and $^-$ in $\mathbf{B}_0$.

a_1	a_2	a_1+a_2
0	0	0
0	1	1
1	0	1
1	1	1

a_1	a_2	a_1*a_2
0	0	0
0	1	0
1	0	0
1	1	1

a	$\overline{a}$
0	1
1	0

a unique value $f(a) \in \{0,1\}$. It is easy to verify that there are 2^{2^n} distinct Boolean functions of n variables.

Given Boolean functions $f\colon \{0,1\}^n \to \{0,1\}$ and $g\colon \{0,1\}^n \to \{0,1\}$, define, for each n-tuple $a = (a_1, \ldots, a_n) \in \{0,1\}^n$,

$$(f+g)(a) \stackrel{\text{def}}{=} f(a)+g(a),$$

$$(f*g)(a) \stackrel{\text{def}}{=} f(a)*g(a),$$

$$\overline{f}(a) \stackrel{\text{def}}{=} \overline{(f(a))},$$

where on the right-hand side of each equation the addition, multiplication, and complementation are the operations of $\mathbf{B}_0$. These operations are applicable since, for each a, the values of $f(a)$ and $g(a)$ are elements of $\{0,1\}$. Now introduce two special functions $|0|$ and $|1|$ as follows: For all $a \in \{0,1\}^n$, $|0|(a) \stackrel{\text{def}}{=} 0$, and $|1|(a) \stackrel{\text{def}}{=} 1$. It is now straightforward to prove the following theorem:

Theorem 2.1 *Let $\mathbf{B}_n$ be the set of all Boolean functions of n variables. Then $\langle \mathbf{B}_n, +, *, ^-, |0|, |1| \rangle$, is a Boolean algebra where the operations $+$, $*$, and $^-$ are the operations on Boolean functions defined above.*

Proof: We need to verify that the operations on functions as given above satisfy the axioms of Boolean algebra. This is quite straightforward, and we refer the interested reader to [25, 58]. □

We conclude this section by introducing Boolean expressions and relating them to Boolean functions. Let 0, 1, $x_1, \ldots, x_n$ be distinct symbols. A *Boolean expression* over $x_1, \ldots, x_n$ is defined inductively:

1. 0, 1, $x_1, \ldots, x_n$ are Boolean expressions.
2. If E and F are Boolean expressions, then so are $(E{+}F)$, $(E{*}F)$, and $\overline{E}$.
3. Any Boolean expression can be obtained by a finite number of applications of Rules 1 and 2.

To simplify notation we assume that $*$ has precedence over $+$; this allows us to omit some parentheses. Note that, so far, Boolean expressions represent an infinite set of well-formed strings of symbols. Their relation with Boolean functions has not yet been established. To do this, let $\mathcal{E}_n$ denote the set of all Boolean expressions over n variables. Define a mapping $|\ |: \mathcal{E}_n \to \mathbf{B}_n$ as follows:

1. The expressions 0, 1, $x_1, \ldots, x_n$ are mapped to the functions $|0|$, $|1|$, $|x_1|, \ldots, |x_n|$, respectively, where $|x_i|$ is defined by

   $$|x_i|(a_1, \ldots, a_n) \stackrel{\text{def}}{=} a_i, \text{ for all } (a_1, \ldots, a_n) \in \{0,1\}^n.$$

   Thus the function $|x_i|$ selects the ith component of its argument n-tuple.

2. $$|(E{+}F)| \stackrel{\text{def}}{=} (|E|{+}|F|),$$

   $$|(E{*}F)| \stackrel{\text{def}}{=} (|E|{*}|F|),$$

   $$|\overline{E}| \stackrel{\text{def}}{=} \overline{|E|}.$$

The mapping $|\ |$ assigns to each expression $E \in \mathcal{E}_n$ a unique Boolean function $|E| \in \mathbf{B}_n$. However, there is an infinite number of expressions denoting a given function. This can be seen as follows. First, every Boolean function has a canonical sum-of-products expression, in which each product is a product of complemented and uncomplemented variables [25, 58]. That expression is 0 if the function is identically zero; otherwise, it has a product corresponding to every binary n-tuple of variable values for which the function has the value 1. Once we have one expression, we can trivially obtain infinitely many others by using such axioms as $|E| = |(E{+}E)|$ arbitrarily many times.

2.3 Ternary Algebra

In analyzing the behavior of asynchronous circuits, it is often convenient to work in ternary, rather than Boolean, algebra [25, 26, 105]. We use 0 and 1 for the two Boolean values, and a third value Φ, which represents an "uncertain value" that is neither 0 nor 1. More will be said about this later. To improve readability, names of ternary variables will be set in boldface type.

A *ternary algebra* is an algebraic system $\mathbf{T} = \langle \mathcal{A}, +, *, \bar{}, 0, \Phi, 1 \rangle$, where $\mathcal{A}$ is a set, $+$ and $*$ are binary operations on $\mathcal{A}$, $\bar{}$ is a unary operation on $\mathcal{A}$, and 0, 1 and Φ are three distinct elements of $\mathcal{A}$ such that all the axioms of Table 2.4 are satisfied. Note the duality of the axioms.

TABLE 2.4. Axioms of ternary algebra.

For all $\mathbf{a}, \mathbf{b}$, and $\mathbf{c}$ in $\mathcal{A}$, we have:

T1	$\mathbf{a}+\mathbf{a} = \mathbf{a}$	T1′	$\mathbf{a}*\mathbf{a} = \mathbf{a}$
T2	$\mathbf{a}+\mathbf{b} = \mathbf{b}+\mathbf{a}$	T2′	$\mathbf{a}*\mathbf{b} = \mathbf{b}*\mathbf{a}$
T3	$\mathbf{a}+(\mathbf{b}+\mathbf{c}) = (\mathbf{a}+\mathbf{b})+\mathbf{c}$	T3′	$\mathbf{a}*(\mathbf{b}*\mathbf{c}) = (\mathbf{a}*\mathbf{b})*\mathbf{c}$
T4	$\mathbf{a}+(\mathbf{a}*\mathbf{b}) = \mathbf{a}$	T4′	$\mathbf{a}*(\mathbf{a}+\mathbf{b}) = \mathbf{a}$
T5	$\mathbf{a}+0 = \mathbf{a}$	T5′	$\mathbf{a}*1 = \mathbf{a}$
T6	$\mathbf{a}+1 = 1$	T6′	$\mathbf{a}*0 = 0$
T7	$\overline{(\bar{\mathbf{a}})} = \mathbf{a}$		
T8	$\mathbf{a}+(\mathbf{b}*\mathbf{c}) = (\mathbf{a}+\mathbf{b})*(\mathbf{a}+\mathbf{c})$	T8′	$\mathbf{a}*(\mathbf{b}+\mathbf{c}) = (\mathbf{a}*\mathbf{b})+(\mathbf{a}*\mathbf{c})$
T9	$\overline{(\mathbf{a}+\mathbf{b})} = \bar{\mathbf{a}}*\bar{\mathbf{b}}$	T9′	$\overline{(\mathbf{a}*\mathbf{b})} = \bar{\mathbf{a}}+\bar{\mathbf{b}}$
T10	$(\mathbf{a}+\bar{\mathbf{a}})+\Phi = \mathbf{a}+\bar{\mathbf{a}}$	T10′	$(\mathbf{a}*\bar{\mathbf{a}})*\Phi = \mathbf{a}*\bar{\mathbf{a}}$
T11	$\Phi = \bar{\Phi}$		

It should be emphasized that, although algebra $\mathbf{T}$ is very similar to a Boolean algebra, it is *not* a Boolean algebra. In particular, the two axioms concerning complements in a Boolean algebra, $\mathbf{a}+\bar{\mathbf{a}} = 1$ and $\mathbf{a}\bar{\mathbf{a}} = 0$, do not hold when $\mathbf{a} = \bar{\mathbf{a}} = \Phi$.

As in the case of Boolean algebra, there are many axiom sets defining ternary algebra. For example, it can be shown that axioms T10 and T10′ can be replaced by

T′10 $(\mathbf{a}+\bar{\mathbf{a}})+(\mathbf{b}*\bar{\mathbf{b}}) = \mathbf{a}+\bar{\mathbf{a}}$ T′10′ $(\mathbf{a}*\bar{\mathbf{a}})*(\mathbf{b}+\bar{\mathbf{b}}) = \mathbf{a}*\bar{\mathbf{a}}$

The smallest ternary algebra is $\mathbf{T}_0 = \langle \{0, \Phi, 1\}, +, *, \bar{}, 0, \Phi, 1 \rangle$, where the set has only three elements, and $+$, $*$, and $\bar{}$ are the ternary OR, AND, and NOT operations as defined in Table 2.5.[3]

[3]We use the same symbols for the ternary AND, OR, and NOT as we do for the binary functions. The context determines which functions are intended.

TABLE 2.5. The operations $+$, $*$, and $\bar{}$ in $\mathbf{T}_0$.

$\mathbf{a}_1+\mathbf{a}_2$	$\mathbf{a}_2$ 0	Φ	1
0	0	Φ	1
$\mathbf{a}_1$ Φ	Φ	Φ	1
1	1	1	1

$\mathbf{a}_1*\mathbf{a}_2$	$\mathbf{a}_2$ 0	Φ	1
0	0	0	0
$\mathbf{a}_1$ Φ	0	Φ	Φ
1	0	Φ	1

$\mathbf{a}$	$\bar{\mathbf{a}}$
0	1
Φ	Φ
1	0

A *ternary function* of n variables is any function $\mathbf{f}$ from $\{0,\Phi,1\}^n$ to $\{0,\Phi,1\}$, for $n \geq 0$. To each n-tuple $\mathbf{a} = (\mathbf{a}_1,\ldots,\mathbf{a}_n) \in \{0,\Phi,1\}^n$, the function $\mathbf{f}$ assigns a unique value $\mathbf{f}(\mathbf{a}) \in \{0,\Phi,1\}$. It is easy to verify that there are 3^{3^n} distinct ternary functions of n variables.

Given ternary functions $\mathbf{f},\mathbf{g}\colon \{0,\Phi,1\}^n \to \{0,\Phi,1\}$ define ternary functions $\mathbf{f}+\mathbf{g}$, $\mathbf{f}*\mathbf{g}$, and $\bar{\mathbf{f}}$, as follows. For each n-tuple $\mathbf{a} = (\mathbf{a}_1,\ldots,\mathbf{a}_n) \in \{0,\Phi,1\}^n$,

$$(\mathbf{f}+\mathbf{g})(\mathbf{a}) \stackrel{\text{def}}{=} \mathbf{f}(\mathbf{a})+\mathbf{g}(\mathbf{a}),$$

$$(\mathbf{f}*\mathbf{g})(\mathbf{a}) \stackrel{\text{def}}{=} \mathbf{f}(\mathbf{a})*\mathbf{g}(\mathbf{a}),$$

$$\bar{\mathbf{f}}(\mathbf{a}) \stackrel{\text{def}}{=} \overline{\mathbf{f}(\mathbf{a})},$$

where on the right-hand side of each equation the addition, multiplication, and complementation are the operations of $\mathbf{T}_0$. These operations are applicable since, for each $\mathbf{a}$, the values of $\mathbf{f}(\mathbf{a})$ and $\mathbf{g}(\mathbf{a})$ are elements of $\{0,\Phi,1\}$. Now introduce three special functions $|0|$, $|\Phi|$ and $|1|$ as follows: For all $\mathbf{a} \in \{0,\Phi,1\}^n$, $|0|(\mathbf{a}) \stackrel{\text{def}}{=} 0$, $|\Phi|(\mathbf{a}) \stackrel{\text{def}}{=} \Phi$, and $|1|(\mathbf{a}) \stackrel{\text{def}}{=} 1$. It is now straightforward to prove the following theorem.

Theorem 2.2 *Let $\mathbf{T}_n$ be the set of all ternary functions of n variables. Then $\mathbf{T}_n$ is a ternary algebra under the operations defined in Table 2.5 with $|0|$, $|\Phi|$, and $|1|$ acting as 0, Φ, and 1.*

Proof: We need to verify that the operations on functions as given above satisfy the axioms of Table 2.4. This is straightforward, and we leave the details to the interested reader. □

Paralleling the development in the previous section, we now introduce ternary expressions and relate them to ternary functions. Let 0, Φ, 1, $\mathbf{x}_1,\ldots,\mathbf{x}_n$ be distinct symbols. A *ternary expression* over $\mathbf{x}_1,\ldots,\mathbf{x}_n$ is defined inductively:

1. 0, Φ, 1, $\mathbf{x}_1,\ldots,\mathbf{x}_n$ are ternary expressions.
2. If E and F are ternary expressions, then so are $(E+F)$, $(E*F)$, and $\overline{E}$.
3. Any ternary expression can be obtained by a finite number of applications of Rules 1 and 2.

Now, let $\mathcal{F}_n$ denote the set of all ternary expressions over n variables. Define a mapping $|\ |: \mathcal{F}_n \to \mathbf{T}_n$ as follows:

1. The expressions 0, Φ, 1, $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are mapped to the functions $|0|$, $|\Phi|$, $|1|$, $|\mathbf{x}_1|, \ldots, |\mathbf{x}_n|$, respectively, where $|\mathbf{x}_i|$ is defined by

$$|\mathbf{x}_i|(\mathbf{a}_1, \ldots, \mathbf{a}_n) \stackrel{\text{def}}{=} \mathbf{a}_i \text{ for all } (\mathbf{a}_1, \ldots, \mathbf{a}_n) \in \{0, \Phi, 1\}^n.$$

2. $$|(E+F)| \stackrel{\text{def}}{=} (|E|+|F|),$$

$$|(E*F)| \stackrel{\text{def}}{=} (|E|*|F|),$$

$$|\overline{E}| \stackrel{\text{def}}{=} \overline{|E|}.$$

The mapping $|\ |$ assigns to each expression $E \in \mathcal{F}_n$ a unique ternary function $|E| \in \mathbf{T}_n$. It is interesting to note that not all ternary functions have corresponding ternary expressions, as we shall see later. If a function does have such an expression, however, then it has an infinite number of distinct ternary expressions.

We are now ready to consider the relation between Boolean and ternary functions. We need the following definitions. Define the uncertainty partial order $\sqsubseteq$ on $\{0, \Phi, 1\}$ as at the end of Section 2.1. We write $\mathbf{a} \sqsubset \mathbf{b}$ if $\mathbf{a} \sqsubseteq \mathbf{b}$ and $\mathbf{a} \neq \mathbf{b}$.

For $n \geq 1$, let $\{0, \Phi, 1\}^n$ denote the set of all possible n-tuples of ternary values. The partial order $\sqsubseteq$ is extended to $\{0, \Phi, 1\}^n$ in the natural way:

$$\mathbf{a} \sqsubseteq \mathbf{b} \text{ if and only if } \mathbf{a}_i \sqsubseteq \mathbf{b}_i \text{ for all } i,\ 1 \leq i \leq n,$$

where $\mathbf{a} = (\mathbf{a}_1, \ldots, \mathbf{a}_n)$ and $\mathbf{b} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$, are any two elements of $\{0, \Phi, 1\}^n$. Thus, for example,[4] $0\Phi10 \sqsubset 0\Phi1\Phi$, but $0\Phi1$ and $1\Phi1$ are not related by $\sqsubseteq$.

In the partially ordered set $\langle \{0, \Phi, 1\}, \sqsubseteq \rangle$, we define the concept of least upper bound as in Section 2.1. The definition is also extended to $\{0, \Phi, 1\}^n$ to be the component-by-component least upper bound. For example,

$$lub\{\Phi0101, 11101, 01001\} = \Phi\Phi\Phi01.$$

From the definition of the partial order $\sqsubseteq$, one easily verifies the following property of least upper bound:

Proposition 2.1 *For any two ternary variables* $\mathbf{a}$ *and* $\mathbf{b}$ *in* $\{0, \Phi, 1\}$,

$$lub\{\mathbf{a}, \mathbf{b}\} = \mathbf{a}*\mathbf{b} + (\mathbf{a} + \mathbf{b})*\Phi.$$

---

[4] As in Chapter 1, we usually omit parentheses and commas in n-tuples of binary or ternary symbols, e.g., we write 101Φ rather than $(1, 0, 1, \Phi)$.

For any Boolean function $f: \{0,1\}^n \rightarrow \{0,1\}$, we can define its *ternary extension* $\mathbf{f}: \{0,\Phi,1\}^n \rightarrow \{0,\Phi,1\}$ as

$$\mathbf{f}(\mathbf{a}) = lub\{f(t) \mid t \in \{0,1\}^n \text{ and } t \sqsubseteq \mathbf{a}\},$$

for all $\mathbf{a} \in \{0,\Phi,1\}^n$. Note that any Boolean function f agrees with its ternary extension $\mathbf{f}$ when the argument $\mathbf{a}$ is binary.

To illustrate the definition above, we let f be the two-argument OR function; then

$$\mathbf{f}(0,\Phi) = lub\{f(0,0), f(0,1)\} = lub\{0,1\} = \Phi, \text{ and}$$

$$\mathbf{f}(1,\Phi) = lub\{f(1,0), f(1,1)\} = lub\{1,1\} = 1.$$

The reader can verify that the functions defined in Table 2.5 are the ternary extensions of the Boolean functions OR, AND, and NOT.

The following important property, the *monotonicity* property, is easily verified to hold for the ternary extension $\mathbf{f}$ of any Boolean function f:

$$\mathbf{a} \sqsubseteq \mathbf{b} \text{ implies } \mathbf{f}(\mathbf{a}) \sqsubseteq \mathbf{f}(\mathbf{b}),$$

for all $\mathbf{a}, \mathbf{b} \in \{0,\Phi,1\}^n$. This property is interpreted as follows: If input vector $\mathbf{b}$ is at least as uncertain as input vector $\mathbf{a}$, then gate output $\mathbf{f}(\mathbf{b})$ is at least as uncertain as $\mathbf{f}(\mathbf{a})$.

In the case of Boolean algebra, every Boolean function can be represented by a Boolean expression. However, not every ternary function can be represented by a ternary expression. For example, it can be shown that the one-variable function that is Φ when the variable is 1 and is 0 otherwise cannot be represented by any ternary expression. In fact, the ternary functions that can be described by ternary expressions are precisely those that are *monotonic*, i.e., that satisfy the monotonicity property above, as the following theorem [105] shows.

Theorem 2.3 *A ternary function* $\mathbf{f}$ *is monotonic if and only if there exists a ternary expression* F *such that* $|F| = \mathbf{f}$.

Proof: We only sketch the proof here; for further details see [105]. Suppose that the ternary function $\mathbf{f}$ is denoted by the ternary expression F, i.e., that $\mathbf{f} = |F|$. It is easily seen that the functions denoted by the expressions 0, 1, Φ, $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are all monotonic. One then verifies that $\bar{\mathbf{f}}$ is monotonic if and only if $\mathbf{f}$ is. Next, from the definition of the operation $+$, it is easy to show that this operation is monotonic, i.e., that $\mathbf{a} \sqsubseteq \mathbf{a}'$ and $\mathbf{b} \sqsubseteq \mathbf{b}'$ imply $\mathbf{a}+\mathbf{b} \sqsubseteq \mathbf{a}'+\mathbf{b}'$. Similarly, $*$ is also monotonic. It then follows by induction on the number of operators ($\bar{}$, $+$, and $*$) in the ternary expression F that $\mathbf{f} = |F|$ is monotonic.

Conversely, suppose $\mathbf{f}: \{0,\Phi,1\}^n \rightarrow \{0,\Phi,1\}$ is monotonic. Consider the set $\mathbf{f}^{-1}(1)$, i.e., the set of all ternary n-tuples $\mathbf{a}$ such that $\mathbf{f}(\mathbf{a}) = 1$. Select all the maximal n-tuples (under the partial order $\sqsubseteq$) of $\mathbf{f}^{-1}(1)$. For each such n-tuple $\mathbf{a}$ construct a product P of variables and complemented variables

as follows. If $\mathbf{a}_i = 0$, include $\overline{\mathbf{x}_i}$ as a factor in P. If $\mathbf{a}_i = 1$, include $\mathbf{x}_i$ as a factor in P. Finally, if $\mathbf{a}_i = \Phi$, include 1 as a factor. It is clear that the product P accounts for all the 1's in $\mathbf{f}$ that are due to all the n-tuples below or equal to the maximal n-tuple being considered. Since $\mathbf{f}$ is monotonic, it follows that all such n-tuples are indeed in the set $\mathbf{f}^{-1}(1)$. Denote the sum of all such products derived from maximal n-tuples of $\mathbf{f}^{-1}(1)$ by F_1. In a similar way, derive a sum of products F_0 for the set $\mathbf{f}^{-1}(0)$. This accounts for all the 0's of the function. Noticing that $\mathbf{f}$ must be Φ if it is not 0 or 1, we obtain the ternary expression $F = F_1 + \Phi * \overline{(F_0 + F_1)}$, which denotes $\mathbf{f}$. □

The expression derived in the proof of the theorem can be simplified to $F = F_1 + \Phi * \overline{F_0}$ as the following proposition shows.

Proposition 2.2 *If* $\mathbf{a}$ *and* $\mathbf{b}$ *are arbitrary ternary values, then*

$$\mathbf{a}+\Phi*\overline{(\mathbf{b}+\mathbf{a})} = \mathbf{a}+\overline{\mathbf{b}}*\Phi.$$

Proof: This can be seen as follows:

$$\begin{aligned}
\mathbf{a} + \Phi*\overline{(\mathbf{b} + \mathbf{a})} &= \mathbf{a} + \Phi*\overline{\mathbf{b}}*\overline{\mathbf{a}} = (\mathbf{a}+\overline{\mathbf{a}})*(\mathbf{a}+\Phi*\overline{\mathbf{b}}) \\
&= (\mathbf{a}+\overline{\mathbf{a}}+\Phi)*(\mathbf{a}+\Phi*\overline{\mathbf{b}}) \\
&= \mathbf{a}+(\overline{\mathbf{a}}+\Phi)*\Phi*\overline{\mathbf{b}} = \mathbf{a}+\Phi*\overline{\mathbf{b}}*\overline{\mathbf{a}}+\Phi*\overline{\mathbf{b}} \\
&= \mathbf{a}+\Phi*\overline{\mathbf{b}} \\
&= \mathbf{a}+\overline{\mathbf{b}}*\Phi.
\end{aligned}$$

□

2.4 Directed Graphs

For several topics in this book it is convenient to represent certain concepts graphically. For this reason, we provide a very brief introduction to the theory of directed graphs.

A *directed graph* or *digraph* G is an ordered pair $\langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{V}$ is a set of elements called *vertices* and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of *edges*. An edge $e = (v, v')$ is said to be *from* v *to* v'; v is the *tail* of e, while v' is its *head*. Note that there can be at most one edge from one vertex to another. If (v, v') or (v', v) is an edge in a digraph, then the vertices v and v' are said to be *adjacent*. There may be an edge from a vertex to itself; such an edge is called a *loop* or *self-loop*. The *indegree* of a vertex v is the number of edges with v as the head. The *outdegree* of a vertex v is the number of edges with v as the tail.

To illustrate the definitions above, consider $G = \langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{V} = \{1, 2, 3, 4\}$ and $\mathcal{E} = \{(1,1), (1,2), (1,3), (2,3), (3,1)\}$. It is customary to represent digraphs as diagrams in which vertices are small circles and edges are lines with arrowheads. The digraph G above is represented in Figure 2.1. Node 3 has indegree 2 and outdegree 1. Node 1 has a loop.

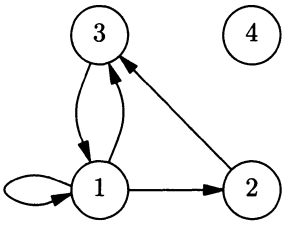


FIGURE 2.1. Digraph G.

A *walk* is a sequence of edges $(e_1, \ldots, e_p)$, such that $e_i = (v_{i-1}, v_i)$, i.e., the head of e_i is the same as the tail of e_{i+1}, for all $i = 1, \ldots, p-1$. The number p of edges in a walk is its *length*. A walk can also be uniquely specified by the sequence $v_0, \ldots, v_p$ of vertices encountered during the walk. If the edges of a walk are all distinct, it is called a *trail*. For example, the vertex sequence $(1, 2, 3, 1, 3, 1)$ in G, which describes a walk of length 5, is not a trail because the edge $(3, 1)$ appears twice. If all the vertices of a walk, except possibly the first and the last, are distinct, it is called a *path*. Thus $(2, 3, 1, 3)$ is a trail of G, but it is not a path. A walk, trail, or path is *closed* if it has positive length and its initial vertex is the same as its final vertex. A *cycle* is another name for a closed path. A digraph is *acyclic* if it has no cycles.

A digraph $G = \langle \mathcal{V}, \mathcal{E} \rangle$ is said to be *bipartite* if its vertices can be partitioned into two disjoint sets $\mathcal{V}_1$ and $\mathcal{V}_2$, such that no two vertices in the same set are adjacent.

A subset $\mathcal{K}$ of $\mathcal{V}$ is called a *feedback vertex set* if every cycle of G contains at least one vertex from $\mathcal{K}$. Thus, if all the vertices in $\mathcal{K}$ are removed from the graph along with all their incident edges, the remaining digraph is acyclic. For the digraph G of Figure 2.1 the sets $\{1, 3\}$ and $\{1\}$ are feedback vertex sets.

Chapter 3

Delay Models

The customary model of a logic gate is its Boolean function. It should be clear that this model does not take into account all of the properties of a physical gate. For example, physical gates have delays associated with their operation. Thus, if an input of a gate changes at some time, its output will respond to this change only at some later time, whereas the Boolean function model treats the response as instantaneous. In this chapter we consider the basic properties of delays, and introduce a number of mathematical models of delays. First, however, we discuss the possible behaviors of the environment of a circuit.

3.1 Environment Modes

In modeling a physical system we usually select a number of system variables to represent an abstraction of the system. We call these variables the *state variables* of the system. If the knowledge of the chosen variables is adequate to describe the aspects of the system behavior that are of interest to us, then we have an adequate model. Otherwise the set of variables must be augmented.

Every circuit operates in some *environment* that provides inputs to the circuit. The concept of a change in the input state is a very basic one, and one that we will use frequently in this book. This concept is formalized as follows: A variable $v(t)$ taking its values from a finite domain is said to *change at time* τ if $v(\tau) = \beta$, and there exists a $\delta > 0$ such that $v(t) = \alpha \neq \beta$, for $\tau - \delta \leq t < \tau$. In other words, v must have the new value at time τ and it must have had the old value for an interval just before, but not including, τ.

The environment may change the circuit inputs at any time, without paying any attention to the state of the circuit; such a mode of operation might be called the *completely unrestricted mode.* But a circuit might fail to operate correctly, if its inputs are changed too quickly or at the wrong time. Even if we ignore circuits entirely, a completely unrestricted environment might lead to some serious difficulties. For example, consider a signal that changes at the following times: $0, 1/2, 3/4, 7/8, \ldots$. Such a signal would have an infinite number of changes in the finite interval of one time unit.

Because the completely unrestricted mode cannot arise in practice and leads to considerable mathematical difficulties, we assume from now on that every environment satisfies the following:

- **Finiteness Condition**: Only a finite, but possibly unbounded, number of signal changes can occur in any finite interval.

An environment satisfying only the finiteness condition is called *unrestricted.*

We distinguish two restricted modes of operation: the fundamental mode and the input-output mode. The *fundamental mode* of operation assumes that the circuit starts in some stable total state, i.e., in a state in which its inputs, internal signals, and outputs all have fixed values and have no tendency to change. (More will be said about these concepts later; for the time being we appeal to the reader's intuition.) By definition, a stable total state persists permanently, unless the circuit inputs change. In such a stable state, the environment is permitted to change the circuit inputs. After that, however, the environment is not allowed to change the inputs again until the entire circuit stabilizes. Note that this assumes that the circuit does indeed stabilize; this assumption holds in most circuits of interest. The fundamental mode of operation has been used since the introduction of asynchronous circuits [66, 67, 93, 135]. In practice, this mode is realized as follows. One estimates the time required for a circuit to stabilize in the worst case, and then makes sure that the inputs remain constant for at least that amount of time. Note that the definition of the fundamental mode makes sense only if one assumes that the circuit delays are bounded from above.

More recent asynchronous design techniques use the *input-output mode* of operation [18, 19, 103, 147]. As before, the starting point is a stable total state of the circuit. Here, the environment is allowed to change the circuit inputs. The environment may change the inputs again only after the circuit has responded by producing an output change, or if no output response is expected. Note that this does *not* imply that the entire circuit must be stable, for some internal signals may still be changing.

We shall return to fundamental mode and input-output mode operations later and make these concepts more precise. For the present chapter we assume that the environment is using a mode that satisfies the finiteness condition but is otherwise unrestricted.

3.2 Gates with Delays

To motivate the body of this chapter we give a brief introduction to the problems one encounters when dealing with physical components such as gates. To keep the discussion simple, we consider an inverter. Our first model of an inverter represents it simply as the Boolean complement func-

tion. To obtain a more accurate model, we represent the physical inverter as an ideal inverter (i.e., the Boolean complement function) in series with a delay element. Figure 3.1 shows an ideal inverter with input X and output Y in series with a delay element with input Y and output y. The signals X and y are intended to represent the input and output signals of the physical inverter. The fictitious signal Y is called the "excitation" of the physical inverter; this represents the value toward which the inverter output is being driven.

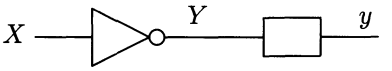


FIGURE 3.1. Model of physical inverter.

Figure 3.2 shows some waveforms of the signals associated with an inverter modeled as an ideal inverter in series with a delay element. The

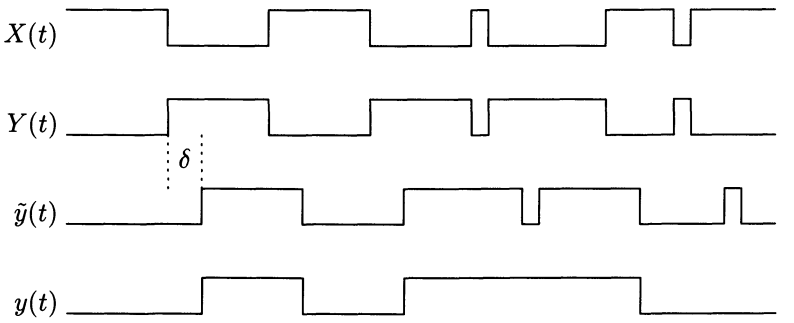


FIGURE 3.2. Waveforms for inverter with delay.

input signal $X(t)$ varies with time. It is assumed to be binary and capable of instantaneous changes from 0 to 1 and from 1 to 0. These changes may occur at any time and may result in wide or narrow "pulses," i.e., intervals during which the signal has a constant value. Thus we are assuming the environment is unrestricted except that it satisfies the finiteness condition. The signal Y is assumed to be the complemented version of the input X at all times; however, the physical inverter output y follows the changes occurring in the signal Y only after some delay δ. If the delay were constant and ideal, the output would appear as shown by the waveform $\tilde{y}(t)$. The actual output, shown as $y(t)$, is similar to $\tilde{y}(t)$, except that short pulses occurring in $\tilde{y}$ do not occur in y. This reflects the inertial nature of physical delays. In the following sections we consider a number of possible models for delay elements.

3.3 Ideal Delays

A *delay* is a "black box" that has one input and one output (see Figure 3.3) and an input/output behavior that is governed by a *delay model.* To simplify the discussion, assume that $X(t) = x(t) = \beta \in \{0, 1\}$ for $t < 0$. This assumption allows us to establish a well-defined starting point.

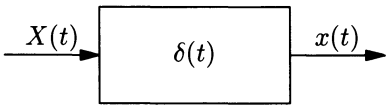


FIGURE 3.3. Delay component.

The concept of ideal delay was introduced informally in Figure 3.2. We would like to make precise the notion that a delay is ideal in that it is not inertial, i.e., does not "lose any pulses." There are several possible variations for such a definition, as we now show. In the *fixed ideal* delay (FID) model, the delay's behavior is specified by the following rule:

$$x(t) = X(t - d),$$

where the delay $d > 0$ is a fixed constant. Thus the output is an exact replica of the input but shifted to the future d units of time. An example of the response of an ideal delay ($d = 1$) to an input signal is shown in Figure 3.4.

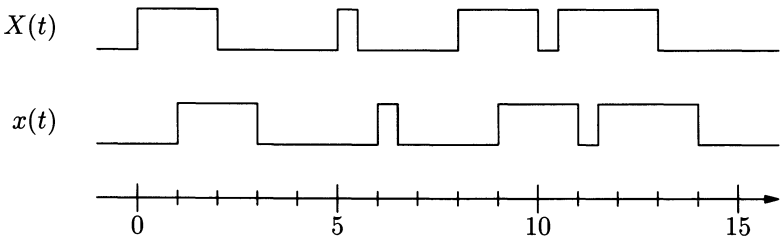


FIGURE 3.4. Ideal delay response.

In a physical circuit, we normally do not know the exact size of a delay; at best, we might have an estimate. Also, delays are normally not constant, but might vary with temperature or change with age. Moreover, the delay of a component might depend on its previous history. For example, a gate that has just changed from 0 to 1 might have a shorter delay for changing back to 0 than it would have, had it had the value 1 for a long time. To model such properties, we allow delays to vary in time, but only within some bounds.

In analyzing circuits, it turns out that it is not so much the absolute magnitude of a delay that is of importance, but rather the delay ratios. Suppose

delay d_1 is fixed, but another delay d_2 can be arbitrarily large. Then the ratio d_2/d_1 can also be arbitrarily large. To represent physical components realistically, we assume that every delay is bounded from above. Note that this assumption does not prevent the ratio d_2/d_1 from being arbitrarily large, if d_1 is permitted to be arbitrarily close to zero. Consequently, although every physical delay is bounded from below by zero, we do not consider zero to be a proper lower bound. This kind of reasoning leads to the terminology introduced below.

We distinguish between two types of assumptions about delay bounds. First, we might assume the delays are bounded both from below and from above (by nonzero constants). We say such delays are *bi-bounded.* Delays that are only bounded from above will be called *up-bounded.* Since we always assume that each delay is bounded from above, i.e., either up-bounded or bi-bounded, there is no need for the concept of "down-bounded" delays.

A *bi-bounded ideal* delay (BID) is defined as follows. Recall that we are assuming that the input and the output of the delay component have the same value initially and that the signals are binary. Under these special circumstances, we may represent the input waveform as a (finite or infinite) sequence $(t_1, t_2, \ldots)$ of increasing real numbers, where each real number t_i represents an instant at which the input signal changes. Thus the input waveform in Figure 3.4 could be represented by the sequence $(0, 2, 5, 5.5, 8, 10, 10.5, 13)$, together with the initial value 0. In a similar way, we represent the output waveform of a delay component by the sequence $(t'_1, t'_2, \ldots)$ of increasing real numbers. In the BID model, we have the following rules:

1. There is a one-to-one correspondence $t_i \mapsto t'_i$ between the sets of instants in the input and output waveforms.

2. There exist constants $d > 0$ and $D > d$, such that

$$t_i + d \leq t'_i < t_i + D,$$

for all $i = 1, 2, \ldots$.

The reader should observe the following difference between the nature of the definitions of the FID and BID models. In the FID model, given an input waveform, one may predict uniquely the output waveform. This is not possible in the BID model. Here, given an input waveform and a corresponding candidate for an output waveform, we can only decide whether or not the given output waveform is consistent with the BID model, i.e., whether it could occur as a response to the given input waveform. Thus, to each input waveform there corresponds an infinite family of possible output waveforms.

It is also useful to have the concept of an up-bounded ideal delay. Formally, the input and output waveforms of an *up-bounded ideal* delay (UID)

with upper bound D satisfy the following conditions: Let $(t_1, t_2, \ldots)$ and $(t'_1, t'_2, \ldots)$ be input and output sequences (respectively) of increasing real numbers. Then

1. there is a one-to-one correspondence $t_i \mapsto t'_i$ between the sets of instants in the input and output waveforms;
2. $t_i < t'_i < t_i + D$, for all $i = 1, 2, \ldots$.

Thus any UID delay is strictly greater than 0 and strictly less than D.

3.4 Inertial Delays

The ideal delay model is often not realistic, since it fails to capture the fact that many physical delays ignore very short pulses, i.e., tend to "smooth out" fast varying signals. For this reason we consider several types of inertial delays [100, 135]. Our approach follows that of [23, 24, 122].

The first delay model we consider is the *fixed inertial* (FIN) delay model. The delay $\delta(t)$ is constant in time, i.e., $\delta(t) = d > 0$. The behavior is defined by the following two rules:

1. If $x(t)$ changes from α to $\overline{\alpha}$ at time τ, then we must have had $X(t) = \overline{\alpha}$ for $\tau - d \leq t < \tau$.
2. If $X(t)$ changes from α to $\overline{\alpha}$ at time τ and $x(\tau) = \alpha$, then either (a) $x(t)$ changes to $\overline{\alpha}$ at time $\tau + d$ or (b) $X(t)$ changes back to α before $\tau + d$.

In Figure 3.5 we show how a fixed inertial delay ($d = 1$) would react to the same input signal as in Figure 3.4.

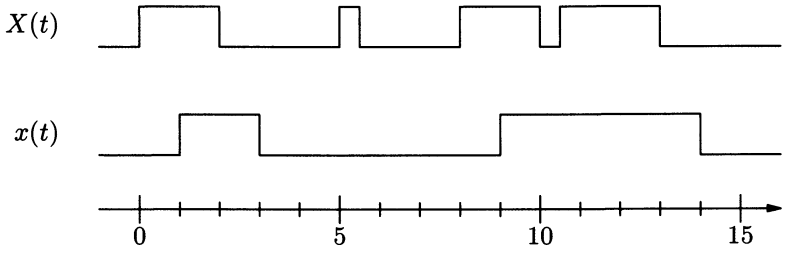


FIGURE 3.5. Fixed inertial delay response.

We next define bi-bounded inertial delays satisfying

$$0 < d \leq \delta(t) < D,$$

where d and D are positive real numbers. In the *bi-bounded inertial* (BIN) delay model the input/output behavior must obey the following two rules:

1. If $x(t)$ changes from α to $\overline{\alpha}$ at time τ, then we must have had $X(t) = \overline{\alpha}$ for $\tau - d \leq t < \tau$.

2. If $X(t) = \alpha$ for $\tau \leq t < \tau + D$, then there must exist a time $\tilde{\tau}$, $\tau \leq \tilde{\tau} < \tau + D$, such that $x(t) = \alpha$ for $\tilde{\tau} \leq t < \tau + D$.

The first rule deals with the minimum delay of the delay element. It states that the element must be unstable for at least d units of time in order to change. The second rule deals with the maximum delay possible; note that we have a strict inequality here, i.e., a delay element cannot be unstable for D units of time without changing. In Figure 3.6, we show two possible responses to the input waveform of Figure 3.4, when the delay is bounded by $1 \leq \delta(t) < 2$.

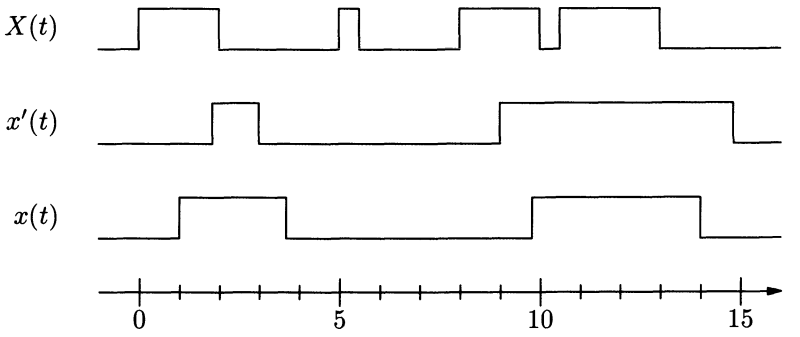


FIGURE 3.6. Possible bi-bounded inertial delay responses.

We can also define an *up-bounded inertial* (UIN) delay model, which must satisfy the following two properties:

1. If $x(t)$ changes from α to $\overline{\alpha}$ at time τ, then there exists $\delta > 0$ such that $X(t) = \overline{\alpha}$ for $\tau - \delta \leq t < \tau$.

2. If $X(t) = \alpha$ for $\tau \leq t < \tau + D$, then there exists a time $\tilde{\tau}$, $\tau \leq \tilde{\tau} < \tau + D$ such that $x(t) = \alpha$ for $\tilde{\tau} \leq t < \tau + D$.

Note that Property 2 implies that the δ in Property 1 must be less than D.

In the delay models above, we have assumed that all the signals are binary, and that changes from 0 to 1 or from 1 to 0 are instantaneous. This is clearly an idealized assumption. In a physical circuit, there is usually a voltage range, below the minimum voltage for 1 but above the maximum voltage for 0, in which the logical value of the node is indeterminate. To capture the fact that changing signals must be in this indeterminate voltage range for a nonnegligible amount of time, we force all changing values to go via Φ, i.e., $x(t)$ can only change from a binary value to Φ or from Φ to a binary value, but never directly from 0 to 1 or from 1 to 0. Using this approach, we define extended inertial delay models of the bi-bounded and

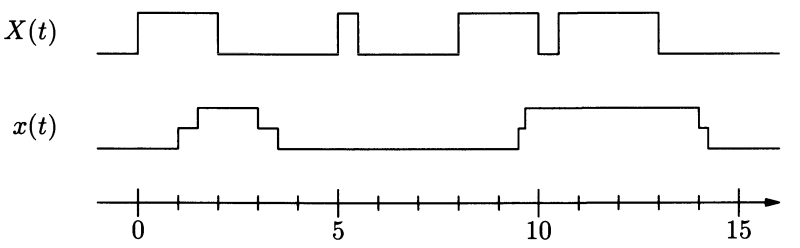


FIGURE 3.7. Possible XBIN delay response to binary input.

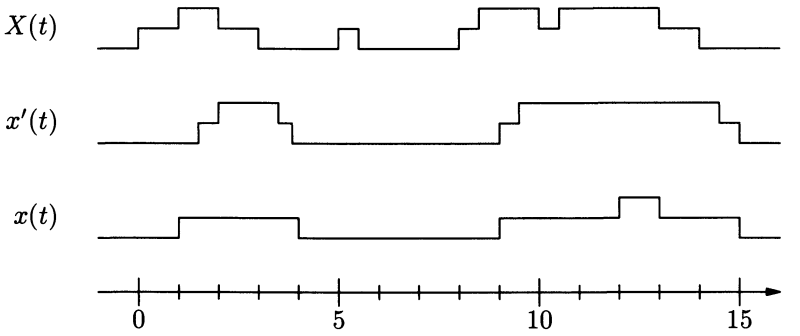


FIGURE 3.8. Possible XBIN delay responses to ternary input.

up-bounded types. Let $0 < d < D$; the *extended bi-bounded inertial* (XBIN) delay model satisfies the rules:

1. (a) If $x(t)$ changes from a binary value α to Φ at some time τ, then we must have had $X(t) \neq \alpha$ for $\tau - d \leq t < \tau$.
   (b) If $x(t)$ changes from Φ to a binary value α at some time τ, then we must have had $X(t) = \alpha$ for $\tau - d \leq t < \tau$.
2. (a) If $X(t) = \beta \in \{0, \Phi, 1\}$ for $\tau \leq t < \tau + D$, then there must exist a $\tilde{\tau}$, $\tilde{\tau} < \tau + D$, such that $x(t) = \beta$ for $\tilde{\tau} \leq t < \tau + D$.
   (b) If $X(t) \neq \alpha \in \{0, 1\}$ for $\tau \leq t < \tau + D$, then there must exist a $\tilde{\tau}$, $\tilde{\tau} < \tau + D$, such that $x(t) \neq \alpha$ for $\tilde{\tau} \leq t < \tau + D$.

We restrict Rule 2(b) to binary values in order to allow the output to be Φ when the input is oscillating between 0 and 1 (assuming each period of 0 (1) is strictly less than D, of course).

In Figure 3.7, we show a possible response of an XBIN delay with $d = 1$ and $D = 2$ to the same binary input signal as in Figure 3.4. However, the response of an XBIN to a signal containing Φs is more interesting. Two possible responses to such an input signal are shown in Figure 3.8. Note that the XBIN can both increase and decrease the duration of Φ periods.

The latter effect can occur, for example, when the delay element changes more slowly from 0 to Φ than from Φ to 1.

The *extended up-bounded inertial* (XUIN) delay model satisfies the rules:

1. (a) If $x(t)$ changes from a binary value α to Φ at some time τ, then there exists $\delta > 0$ such that $X(t) \neq \alpha$ for $\tau - \delta \leq t < \tau$.

   (b) If $x(t)$ changes from Φ to a binary value α at some time τ, then there exists $\delta > 0$ such that $X(t) = \alpha$ for $\tau - \delta \leq t < \tau$.

2. (a) If $X(t) = \beta \in \{0, \Phi, 1\}$ for $\tau \leq t < \tau + D$, then there must exist a $\tilde{\tau}$, $\tilde{\tau} < \tau + D$, such that $x(t) = \beta$ for $\tilde{\tau} \leq t < \tau + D$.

   (b) If $X(t) \neq \alpha \in \{0, 1\}$ for $\tau \leq t < \tau + D$, then there must exist a $\tilde{\tau}$, $\tilde{\tau} < \tau + D$, such that $x(t) \neq \alpha$ for $\tilde{\tau} < t < \tilde{\tau} + D$.

We will return to inertial delays in Chapter 6.

Chapter 4

Gate Circuits

Gate circuits have been in use for a number of years, are generally well known, and are relatively easy to model mathematically. In this chapter we define several gate circuit classes, some reflecting topological properties and others arising from behavioral characteristics. We show how gate circuits can be modeled in a very general, mathematically precise, framework. In the next chapter we show how modern MOS circuits can also be modeled in our framework; this enables us later to derive a theory applicable to gates as well as MOS circuits. For additional information concerning gate circuits the reader should refer to a basic text on logic design, for example [25, 77, 88, 94].

How To Read This Chapter

Section 4.5 can be omitted on first reading. The following comments clarify our terminology in this chapter. By the term *gate* we mean our model of a physical circuit that implements a Boolean function. Thus our gate is more of a mathematical object that a physical circuit. As we have mentioned in Chapter 3, a gate is represented by a Boolean function together with a delay element of some type. For some purposes, the delay is assumed to be zero; then the Boolean function alone suffices. Also, the input signals to our gates are idealized binary or ternary signals.

We talk about "structure" and "behavior" of a gate circuit rather informally. The structure of a gate circuit includes a list of gates contained in the circuit, the Boolean functions associated with these gates, and the wire connections among the gates. The notion of behavior is defined more precisely in Chapters 6, 11 and 12; for now it suffices to think of behavior as describing what will happen, i.e., how various signals in the circuit evolve with time.

4.1 Properties of Gates

A gate has one or more distinct *inputs* and one or more distinct *outputs*. There is a direction, from input to output, that is implicitly associated with each gate. Although some families of gates have multiple outputs,

most gates have only a single output. In this book we restrict ourselves to single-output gates. A list of commonly used one- and two-input Boolean functions and a list of gate symbols corresponding to these functions were given in Chapter 1, Table 1.1 and Figure 1.1.

If we apply binary signals at the inputs $X_1, \ldots, X_n$ of a gate, the resulting output value is determined by the gate type. In this book we assume that the binary values are realized by voltage levels. In fact, we use "positive" logic, where 0 is represented by a low voltage and 1 by a high voltage. In reality, since transitions are not instantaneous and voltage levels fluctuate slightly, any voltage below a certain threshold voltage V_L represents 0 and any voltage above a threshold voltage V_H represents 1. Obviously, $V_H > V_L$. If the voltage of at some node (point in the circuit) is above V_L, but below V_H, we consider the value of the node to be undefined. We return to this later in this chapter.

An input to a gate may be a constant (i.e., 0 or 1) or a binary variable X_i. The number of inputs to a gate is the *fan-in* of the gate. The fan-in of gates in most circuit families varies from one to eight.

The output of a gate can be connected to one or more inputs of other gates. Sometimes the output of a gate can be connected to the gate's own input(s), although in practical designs this is relatively uncommon. In this book we use several such circuits, mainly to reduce the size of examples.

If a gate forms a part of a larger circuit, the number of gate inputs connected to the output of a given gate is said to be the *fan-out* of that gate. Virtually all circuit families impose restrictions on the maximum fan-out. Even if no such restrictions are imposed, limiting the fan-out is desirable, since the delay of a gate usually increases with the fan-out.

Normally, the output of a gate cannot be connected to the output of another gate. Such a connection would result in an ill-defined value if the computed outputs of the two gates disagreed. In some gate implementations, however, the outputs of several gates *can* be connected together. Depending on the technology, the result is a *wired*-OR or a *wired*-AND connection. In a wired-OR technology, the value on the connection point is the logical OR of the output values the connected gates would have if the gates had not been wired together. A wired-AND connection works in a similar manner. Our approach to wired-OR (AND) connections is to introduce a virtual OR (AND) gate instead of the connection.

In certain technologies gates can have *tri-state* outputs. Here, the output of a gate can be electrically isolated. When this occurs, the output is said to be "floating." A floating output has no effect on any wire to which it may be connected. Thus, if the designer makes sure that only one of the several gates with connected outputs is in its "nonfloating" (also called "driven") state, then that single gate determines the value of the connection point. This type of connection is very common in bus-based designs, where more than one sender can supply the value to be sent on the bus. We return to these gate types later in this chapter.

4.2 Classes of Gate Circuits

Given a set of basic gates, we can implement more complex, arbitrary Boolean functions by connecting a number of gates together. For example, the circuit of Figure 4.1 corresponds to the expression[1]

$$y = \overline{(X_1X_2 + X_2X_3 + X_1X_3)}$$

and implements the "minority" function, which has the value 1 if a minority (i.e., one or zero) of its three inputs are 1.

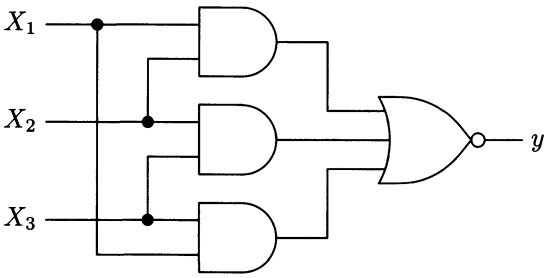


FIGURE 4.1. Feedback-free gate circuit.

We say that a gate circuit is *feedback-free*, if, starting at any point in the circuit and proceeding via connections through gates in the direction input-to-output, it is not possible to reach the same point twice. For example, the circuit of Figure 4.1 is feedback-free. A gate circuit that is not feedback-free is said to contain *feedback*.

In feedback-free circuits we define the concept of *level* of a gate inductively as follows. A gate with only external inputs is said to be at level 1. Inductively, if the inputs to a gate i are either external inputs or outputs of gates of level less than k, and at least one input is the output of a gate of level $k-1$, then gate i is of level k. A feedback-free circuit is a level-k circuit if k is the maximum of the levels of its gates.

It is well known that every Boolean function can be implemented by a feedback-free circuit. Conversely, if we are given a feedback-free gate circuit to analyze, we can use the Boolean function model to represent both the individual gates and also the entire circuit.

A circuit in which the outputs are uniquely determined (after some delay) by the input combination is called *combinational*. In contrast, if a circuit output is not uniquely determined by the present input combination, then it must depend also on previous inputs, i.e., on the input sequence; such a circuit is called *sequential*. We will consider combinational circuits as special cases of sequential circuits.

---

[1]Unless there is a danger of ambiguity, we omit the Boolean multiplication symbol $*$ from Boolean expressions.

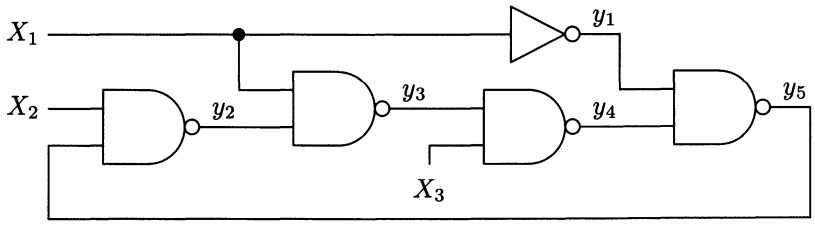


FIGURE 4.2. Combinational gate circuit with feedback.

Clearly, every feedback-free circuit is combinational. As our next example [74] shows, however, the converse does not hold. Consider the gate circuit of Figure 4.2. Suppose the inputs X_1, X_2, and X_3 are held constant for a sufficiently long time that the circuit reaches a stable state, if such a state exists. Assume first that $X_1 = 0$. After some time, determined by the delays of the gates with outputs y_1 and y_3, these outputs will be 1, since $X_1 = 0$ uniquely determines the output of the inverter as well as the NAND gate. Once y_3 becomes 1, y_4 will become (after some delay) $\overline{(1X_3)} = \overline{X_3}$. This will eventually cause the output of y_5 to become $\overline{(1\overline{X_3})} = X_3$. Finally, y_2 will become $\overline{(X_2X_3)}$. Note that the gate circuit is now stable, i.e., the output of every gate has the value computed by the gate from the current input values. In summary, we have just shown that any input state of the form $X = (0, X_2, X_3)$ (which represents the four 3-tuples 000, 001, 010, and 011) forces the gate circuit to a corresponding unique internal state of the form $y = (y_1, \ldots, y_5) = (1, \overline{(X_2X_3)}, 1, \overline{X_3}, X_3)$. Note that this analysis is completely independent of the initial state of the gate circuit.

A similar analysis takes care of the remaining input alternatives. Let $X = (1, X_2, X_3)$. After some delay, y_1 will become 0. This will force y_5 to become 1, which, in turn, will cause y_2 to become $\overline{(X_21)} = \overline{X_2}$. Consequently, gates y_3 and y_4 will eventually have the values X_2 and $\overline{(X_2X_3)}$, respectively. The circuit will then be stable. In summary, the input $X = (1, X_2, X_3)$ forces the circuit to a unique internal state $y = (0, \overline{X_2}, X_2, \overline{(X_2X_3)}, 1)$. Again, the analysis is completely independent of the initial state of the gate circuit.

We can now combine the two cases above as follows:

$$y_1 = 1 \text{ when } X_1 = 0$$

and

$$y_1 = 0 \text{ when } X_1 = 1.$$

Thus $y_1 = \overline{X_1}$. Furthermore,

$$y_2 = \overline{(X_2X_3)} \text{ when } X_1 = 0$$

and

$$y_2 = \overline{X_2} \text{ when } X_1 = 1.$$

Consequently,

$$y_2 = \overline{X_1}\,\overline{(X_2X_3)}+X_1\overline{X_2} = \cdots = \overline{X_2}+\overline{X_1}\,\overline{X_3}.$$

Similarly,

$$\begin{aligned} y_3 &= \overline{X_1}+X_2, \\ y_4 &= X_1\overline{X_2}+\overline{X_3}, \\ y_5 &= X_1+X_3. \end{aligned}$$

Since, after a certain delay, all the gate outputs are uniquely determined by the external inputs, the gate circuit is combinational, despite the fact that it contains feedback.

It should be clear that the Boolean function model does not take into account all of the properties of a physical gate. For example, physical gates have delays associated with their operation. However, in traditional analysis of combinational circuits, delay effects are usually considered secondary to the basic operation of the gate. In summary, for combinational circuit design, a gate is usually adequately described by the very simple Boolean function model. In fact, Boolean algebra has been a very effective tool for the analysis and design of combinational circuits over the years.

In contrast to the example of Figure 4.2, our next example shows a circuit that has feedback and is not combinational. For such a circuit, Boolean functions alone do not suffice, as we now show. Consider the circuit of Figure 4.3, which is a NOR latch that we use as a frequent example. If

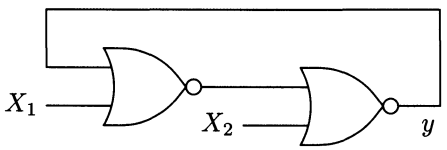


FIGURE 4.3. NOR latch circuit.

we try to analyze this circuit without introducing any delays, we run into difficulties. Using the Boolean approach we conclude that the output y of the second NOR gate depends on the input X_2 and the output of the first NOR gate. The latter signal depends on X_1 and on y. But then the output y depends on itself, for $y = \overline{(\overline{(X_1 + y)} + X_2)} = (X_1 + y)\overline{X_2}$. If $X_1 = X_2 = 0$, then the equation reduces to $y = y$. The latter equation has two solutions, $y = 0$ and $y = 1$. Since y is not uniquely determined by the inputs, this circuit is not combinational. To analyze its behavior properly, we must be able to explain how its state can change as a result of an input change, taking into account the present output value. Such reasoning is not possible if we represent each gate as a Boolean function. In fact, an analysis only becomes possible if delays are considered. For example, if a delay is added somewhere in the feedback loop of Figure 4.3, say as shown in Figure 4.4,

then we can talk about the old value of the output as stored at the output y of the delay and the new value of the output as it appears at the input Y of the delay.

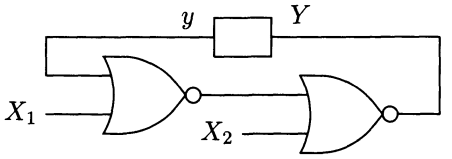


FIGURE 4.4. NOR latch with feedback delay.

Two questions need to be addressed before an analysis of a gate circuit is possible in the Boolean function/delay model. First, *what type* of delays should we use in our model? We normally use an inertial delay model, since it is appropriate in many cases and the classical analysis methods are based on that model. We also briefly consider analysis using ideal delays. Second, *where* in the circuit should the presence of delays be assumed, so that the results of the analysis are sufficiently realistic? Should one associate a delay with every gate? Would a model that associates a delay with every gate and every wire be more "accurate" in some sense? Is a model in which two delays are associated with each wire still more accurate than the one with a single delay per wire? We will be able to answer these questions in Chapters 6 and 7, but only after a considerable amount of theory has been established.

4.3 The Circuit Graph

We now formulate a mathematical model for gate circuits. The first part of this model describes the structural properties, concerning the gates, gate types, and connections. Formally, these structural properties are captured by the "circuit graph." Behavioral properties are treated later.

A *circuit graph* is a 5-tuple $G = \langle \mathcal{X}, \mathcal{I}, \mathcal{G}, \mathcal{W}, \mathcal{E} \rangle$, where

- $\mathcal{X}$ is a set of *input vertices*, labeled $X_1, X_2, \ldots, X_n$,
- $\mathcal{I}$ is a set of *input-delay vertices*, labeled $x_1, x_2, \ldots, x_n$,
- $\mathcal{G}$ is a set of *gate vertices*, labeled $y_1, y_2, \ldots, y_r$,
- $\mathcal{W}$ is a set of *wire vertices*, labeled $z_1, z_2, \ldots, z_p$, and
- $\mathcal{E} \subseteq (\mathcal{X} \times \mathcal{I}) \cup ((\mathcal{I} \cup \mathcal{G}) \times \mathcal{W}) \cup (\mathcal{W} \times \mathcal{G})$ is a set of *edges*.

All input vertices have indegree 0 and all wire vertices have indegree and outdegree 1. The directed graph defined by $((\mathcal{X} \cup \mathcal{I} \cup \mathcal{G} \cup \mathcal{W}), \mathcal{E})$ must be

bipartite with vertex classes $\mathcal{I} \cup \mathcal{G}$ and $\mathcal{X} \cup \mathcal{W}$. In other words, there is no edge between any two vertices in $\mathcal{I} \cup \mathcal{G}$, nor is there any edge between two vertices in $\mathcal{X} \cup \mathcal{W}$. Note that self-loops (edges of the form (v, v)) are also excluded.

One of the purposes of the circuit graph is to identify all those points in the circuit with which one might want to associate a state variable. Thus we might wish to associate state variables with input delays, gates, and wires. More is said about this in Section 4.4.

We now describe the construction of a circuit graph from a gate circuit. The reader may wish to refer to the circuit of Figure 4.5 and to Figure 4.6, which illustrates this construction. In the circuit graph we have $\mathcal{X} = \{X_1\}$, $\mathcal{I} = \{x_1\}$, $\mathcal{G} = \{y_1, y_2, y_3, y_4\}$, and $\mathcal{W} = \{z_1, z_2, z_3, z_4, z_5, z_6, z_7\}$.

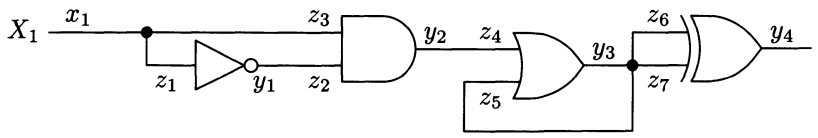


FIGURE 4.5. Gate circuit $C_{4.5}$.

In general, given a gate circuit, we obtain its circuit graph as follows. First, there is an input vertex X_i for every external input, and a gate vertex for every gate. For every input vertex X_i there is an input-delay vertex[2] x_i and an edge from X_i to x_i. For every input i of every gate g in the circuit there is a wire vertex z. There is an edge from the wire vertex z to the gate vertex corresponding to the gate g. If the input i is connected to an external input X_j, there is an edge from the input-delay vertex x_j to the wire vertex z. Otherwise, if the input i is connected to the output of gate g', there is an edge from the gate vertex corresponding to g' to the wire vertex z.

A circuit graph is a convenient and precise notation for describing how the gates of the circuit are connected. It is not sufficient, however, to also describe the behavior of the circuit; for that one needs to answer the following questions: What is the domain, i.e., the set of values that the vertices can take? What determines the behavior of an individual vertex? Which vertices have delays associated with them? What type of delays are assumed? What is the collective behavior of the whole graph? We next focus on the first two questions; we return to the remaining issues later.

In this book the domain $\mathcal{D}$ of a circuit graph is usually $\{0, 1\}$, but the ternary domain $\{0, 1, \Phi\}$ is used when we wish to represent a gate output state that is neither 0 nor 1.

---

[2]The input delay is introduced for somewhat technical reasons that will become clear in Chapter 7. We could use wire delays instead; however, the introduction of an input delay is more convenient.

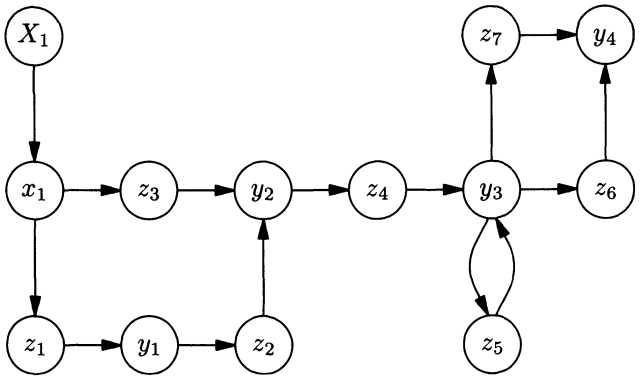


FIGURE 4.6. Circuit graph corresponding to gate circuit $C_{4.5}$.

The behavior of a vertex is governed by a function called the *vertex function* that we now define. The vertex function Y_i of a gate vertex y_i maps a wire-vertex state to $\mathcal{D}$, i.e., $Y_i: \mathcal{D}^{|\mathcal{W}|} \rightarrow \mathcal{D}$. This is the Boolean function of the gate corresponding to the gate vertex. For a wire vertex z_i, the vertex function Z_i, $Z_i: \mathcal{D}^{|\mathcal{I}|+|\mathcal{G}|} \rightarrow \mathcal{D}$, provides the value of the input-delay or gate vertex connected to the incoming edge of the wire vertex. For an input-delay vertex x_i, the vertex function is X_i. For an input vertex, the vertex function maps a state of the environment to the vertex domain $\mathcal{D}$. This function is called X_i. We may think of X_i as the input value provided by the environment; how the environment determines this value is of no interest to us.

To illustrate the concepts introduced so far, consider again the circuit shown in Figure 4.5 with circuit graph shown in Figure 4.6. Assume first that the value domain is $\{0,1\}$. Since we do not know anything about the environment, we simply write the input-delay vertex function as X_1. The vertex functions for the gate vertices correspond directly to the gate functions,

$$Y_1 = \overline{z_1}, \quad Y_2 = z_2 z_3, \quad Y_3 = z_4 + z_5, \quad Y_4 = z_6 \oplus z_7.$$

The wire vertices have the very simple vertex functions,

$$Z_1 = x_1, \; Z_2 = y_1, \; Z_3 = x_1, \; Z_4 = y_2, \; Z_5 = y_3, \; Z_6 = y_3, \; Z_7 = y_3.$$

Finally, the input-delay vertex function is X_1. Since the domain is $\{0,1\}$, the expressions above are all Boolean expressions. On the other hand, had the domain been $\{0,\Phi,1\}$, the expressions would have been ternary expressions representing ternary extensions of the gate functions.

In summary, we use the terms *binary (ternary) circuit graph* to mean a circuit graph together with a binary (ternary) domain and a set of Boolean (ternary) vertex functions, as specified above.

4.4 Network Models

The vertex functions defined in the previous section introduce a distinction between the present value of a vertex variable and the present value of the "excitation" of that vertex variable, i.e., the value computed by the vertex function. This permits us to associate a delay with every input, every gate, and every wire in the circuit.

To represent the state of the entire circuit, we need to select a set of *state variables.* Clearly, the circuit graph model easily permits us to select *all* of the vertex variables as state variables. We may think of such a model as the *input-, gate-, and wire-state* model. As we shall see, this model is too detailed for some applications, and simpler models may be preferred.

By changing the set of state variables, we effectively change the location of the assumed delays in the circuit. Although many different choices of state variables are possible, only some are meaningful. In general, a minimum requirement for a set of state variables is that at least one state variable should appear in every cycle of the circuit graph.

At this point, we are unable to decide which set of vertices should be chosen to represent the state, but an answer to this question is given in Chapters 6 and 7. For now, we only assume that the state variables are selected in such a way that they form a feedback vertex set in the circuit graph. This implies that, if all these vertices were "cut," no feedback would remain in the graph.

Having selected a set of vertices from the circuit graph to act as state variables, we associate with each such vertex two distinct items: the vertex variable and its "excitation function." The *excitation function* of a vertex in the state variable set is defined as follows. We start with the vertex function. We then repeatedly remove all dependencies on vertices that have not been chosen as state variables, by using functional composition of the vertex functions.

Once the state variables have been selected and the excitation functions derived, it is convenient to draw a graph showing the new functional dependencies. This graph, called the *network*, has two sets of vertices: *input excitation vertices* and *state vertices.* There is one input excitation vertex for every external input, and one state vertex for every state variable. There is an edge from vertex i to vertex j if the excitation function of vertex j depends[3] on the variable associated with vertex i.

We view a network as a model of the circuit graph. In general, the network model contains fewer variables than the circuit graph. After we analyze the network in terms of the chosen state variables we may wish to know

---

[3]By "depends" we mean here the usual notion of functional dependence: A function f of n variables $x_1, \ldots, x_n$ depends on x_i if there exist two input n-tuples $a = (a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$ and $a' = (a_1, \ldots, a_{i-1}, a_i', a_{i+1}, \ldots, a_n)$ such that $f(a) \neq f(a')$.

some or all of the other vertex variable values in the original circuit graph in this circuit model. Since these variables are not state variables, they have no delays associated with them. In other words, the value of such a variable in the network model is always equal to the value of its excitation. Because we assume that the state variables constitute a feedback-vertex set, the values of all the variables in the circuit graph are uniquely determined by the values of the external inputs and of the state variables. This dependence of a variable that is not a state variable on the inputs and the state variables is described with the aid of the *circuit equations*. The examples given below clarify these ideas.

To illustrate the concept of network, we show three different sets of state variables—leading to the "gate-state network," the "wire-state network," and an "input- and feedback-state network"—for the circuit graph in Figure 4.6. The domain associated with each network may be binary or ternary.

In the *gate-state network*, only gates are assumed to have delays. Consequently, there is one state variable associated with each gate. In our example, the state variables are y_1, y_2, y_3, and y_4. The excitation functions are obtained as follows:

$$Y_1 = \overline{z_1} = \overline{Z_1} = \overline{x_1} = \overline{X_1},$$

$$Y_2 = z_2 z_3 = Z_2 Z_3 = y_1 x_1 = y_1 X_1,$$

$$Y_3 = z_4 + z_5 = Z_4 + Z_5 = y_2 + y_3,$$

and

$$Y_4 = z_6 \oplus z_7 = Z_6 \oplus Z_7 = y_3 \oplus y_3 = 0,$$

where the last step is a simplification of the Boolean expression $y_3 \oplus y_3$.

From this we can derive the network graph shown in Figure 4.7. Note that vertex y_4 has indegree zero, because Y_4 is a constant. We also obtain the following circuit equations from the circuit graph:

$$x_1 = X_1,$$

$$y_1 = y_1, \quad y_2 = y_2, \quad y_3 = y_3, \quad y_4 = y_4,$$

$$z_1 = Z_1 = x_1 = X_1, \quad z_2 = Z_2 = y_1, \quad z_3 = Z_3 = x_1 = X_1,$$

$$z_4 = Z_4 = y_2, \quad z_5 = Z_5 = y_3, \quad z_6 = Z_6 = y_3, \quad z_7 = Z_7 = y_3.$$

In other words, the circuit equations are

$$x_1 = X_1,$$

$$y_1 = y_1, \quad y_2 = y_2, \quad y_3 = y_3, \quad y_4 = y_4,$$

$$z_1 = X_1, \quad z_2 = y_1, \quad z_3 = X_1, \quad z_4 = y_2, \quad z_5 = y_3, \quad z_6 = y_3, \quad z_7 = y_3.$$

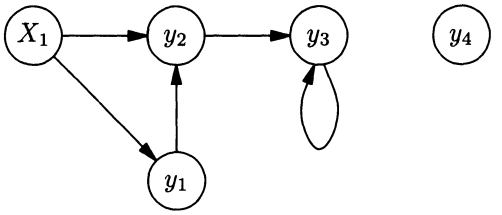


FIGURE 4.7. Binary gate-state network for circuit graph $C_{4.6}$.

On the other hand, if the domain is $\{0, \Phi, 1\}$, the gate-state network has the following excitation functions:

$$\mathbf{Y}_1 = \overline{\mathbf{z}_1} = \overline{\mathbf{Z}_1} = \overline{\mathbf{x}_1} = \overline{\mathbf{X}_1},$$

$$\mathbf{Y}_2 = \mathbf{z}_2\mathbf{z}_3 = \mathbf{Z}_2\mathbf{Z}_3 = \mathbf{y}_1\mathbf{x}_1 = \mathbf{y}_1\mathbf{X}_1,$$

$$\mathbf{Y}_3 = \mathbf{z}_4 + \mathbf{z}_5 = \mathbf{Z}_4 + \mathbf{Z}_5 = \mathbf{y}_2 + \mathbf{y}_3,$$

and

$$\mathbf{Y}_4 = \mathbf{z}_6 \oplus \mathbf{z}_7 = \mathbf{Z}_6 \oplus \mathbf{Z}_7 = \mathbf{y}_3 \oplus \mathbf{y}_3 = \mathbf{y}_3\overline{\mathbf{y}_3},$$

where the operators are ternary and the last step is a simplification of the ternary expression $\mathbf{y}_3 \oplus \mathbf{y}_3$. Note that $\mathbf{Y}_4$ is not identical to 0 in ternary algebra. Consequently, the network graph for this model is slightly different, as is shown in Figure 4.8. The circuit equations are the same as above, except that they are ternary.

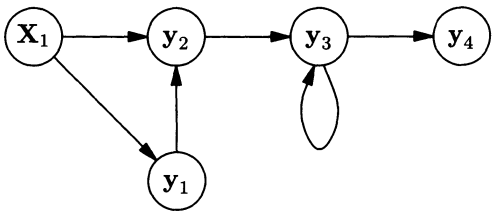


FIGURE 4.8. Ternary gate-state network for circuit graph $C_{4.6}$.

In the *wire-state network*, only wires have delays. Thus, there is one state variable associated with each wire vertex. In our example, the state variables are $z_1, z_2, \ldots, z_7$. We obtain the excitation functions as follows:

$$Z_1 = Z_3 = x_1 = X_1,$$

$$Z_2 = y_1 = Y_1 = \overline{z_1},$$

$$Z_4 = y_2 = Y_2 = z_2 z_3,$$

$$Z_5 = Z_6 = Z_7 = y_3 = Y_3 = z_4 + z_5.$$

The binary network graph for this model is shown in Figure 4.9. The circuit equations are

$$x_1 = X_1,$$

$$y_1 = \overline{z_1},\ \ y_2 = z_2 z_3,\ \ y_3 = z_4 + z_5,\ \ y_4 = z_6 \oplus z_7,$$

$$z_1 = z_1,\ \ z_2 = z_2,\ \ z_3 = z_3,\ \ z_4 = z_4,\ \ z_5 = z_5,\ \ z_6 = z_6,\ \ z_7 = z_7.$$

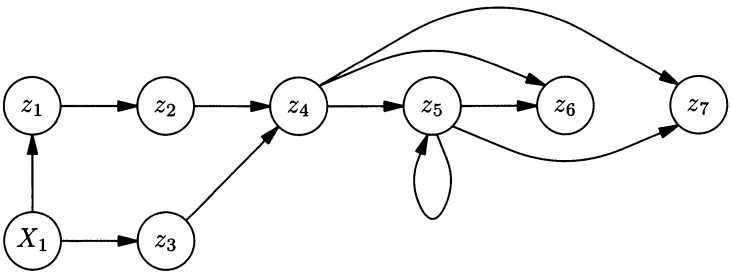


FIGURE 4.9. Binary wire-state network for circuit graph $C_{4.6}$.

Our final example of a choice of state variables gives a ternary *input- and feedback-state network*. Here the domain is $\{0, \Phi, 1\}$, and a delay is associated with each input vertex and with a set of feedback vertices. In our example we use the feedback vertex set $\{y_3\}$. We obtain the excitation functions $\mathbf{X}_1$ and

$$\begin{aligned}\mathbf{Y}_3 &= \mathbf{z}_4 + \mathbf{z}_5 = \mathbf{Z}_4 + \mathbf{Z}_5 = \mathbf{y}_2 + \mathbf{y}_3 = \mathbf{Y}_2 + \mathbf{y}_3 = \mathbf{z}_2\mathbf{z}_3 + \mathbf{y}_3 \\ &= \mathbf{Z}_2\mathbf{Z}_3 + \mathbf{y}_3 = \mathbf{y}_1\mathbf{x}_1 + \mathbf{y}_3 = \mathbf{Y}_1\mathbf{x}_1 + \mathbf{y}_3 = \overline{\mathbf{z}_1}\mathbf{x}_1 + \mathbf{y}_3 \\ &= \overline{\mathbf{Z}_1}\mathbf{x}_1 + \mathbf{y}_3 = \overline{\mathbf{x}_1}\mathbf{x}_1 + \mathbf{y}_3.\end{aligned}$$

Note that $\mathbf{Y}_3$ is not equal to $\mathbf{y}_3$, since the domain is $\{0, \Phi, 1\}$. The network graph corresponding to these functions is shown in Figure 4.10. The circuit equations are

$$\mathbf{x}_1 = \mathbf{x}_1,$$

$$\mathbf{y}_1 = \overline{\mathbf{x}_1},\ \ \mathbf{y}_2 = \mathbf{x}_1\overline{\mathbf{x}_1},\ \ \mathbf{y}_3 = \mathbf{y}_3,\ \ \mathbf{y}_4 = \mathbf{y}_3\overline{\mathbf{y}_3},$$

$$\mathbf{z}_1 = \mathbf{x}_1,\ \ \mathbf{z}_2 = \overline{\mathbf{x}_1},\ \ \mathbf{z}_3 = \mathbf{x}_1,\ \ \mathbf{z}_4 = \mathbf{x}_1\overline{\mathbf{x}_1},$$

$$\mathbf{z}_5 = \mathbf{y}_3,\ \ \mathbf{z}_6 = \mathbf{y}_3,\ \ \mathbf{z}_7 = \mathbf{y}_3.$$

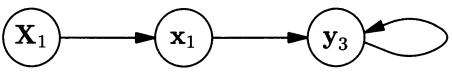


FIGURE 4.10. Ternary input- and feedback-state network for circuit graph $C_{4.6}$.

In summary, a *network* is a 5-tuple

$$N = \langle \mathcal{D}, \mathcal{X}, \mathcal{S}, \mathcal{E}, F \rangle,$$

where $\mathcal{D}$ is the domain, $\mathcal{X}$ is the set of input excitation vertices labeled $X_1, \ldots, X_n$, $\mathcal{S}$ is the set of state vertices with two sets of labels: state variable labels $(s_1, \ldots, s_m)$ and the corresponding excitation function labels $(S_1, \ldots, S_m)$, $\mathcal{E}$ is the set of edges, and F is the vector of circuit equations.

4.5 Models of More Complex Gates

As mentioned in Section 4.1, in some technologies the output of a gate can be made electrically isolated, or *floating*. Consequently, several such gates can have their outputs connected together. If this is the case, the voltage level on the connection point is a function of all the inputs to these gates. In general, there are two ways of modeling this kind of gate in our framework. We can leave the individual gates unaltered and introduce a new "virtual" gate that determines the logic level on the connection point, or we can merge all of the connected gates into one "super-gate," which computes the resulting value for every input combination. In this section we illustrate both approaches.

There are basically three types of gates whose output can be electrically isolated: gates with wired-AND outputs, gates with wired-OR outputs, and gates with tri-state outputs. The output of a wired-AND gate is either low or floating. By itself, such a gate cannot drive its output high. Some external component, usually a resistor, must be used to ensure that the output is properly pulled high when the gate is not pulling it low. If several wired-AND gates have their outputs connected together and one (or more) of them is pulling its output low, the voltage on the connection point will be low. Only if all of them have floating outputs, will the voltage on the connection point be high. In effect, the resulting voltage on the connection point is the AND of the computed values on the gates.

A wired-OR gate functions similarly, except it can only pull its output high. Consequently, if the outputs of several wired-OR gates are connected together, the resulting voltage on the connection point is the OR of the computed values on the gates.

It is straightforward to model a wired-AND or wired-OR connection in our general framework. We introduce a virtual gate to model the connection point. To illustrate the process, consider the gate circuit in Figure 4.11, where the two NAND gates are assumed to have wired-AND outputs.[4] In Figure 4.11 we have also indicated the names of the existing wires. In

[4]We do not show the external pull-up circuit, but the reader can imagine that there is a resistor between the output of, say, y_1 and the power supply.

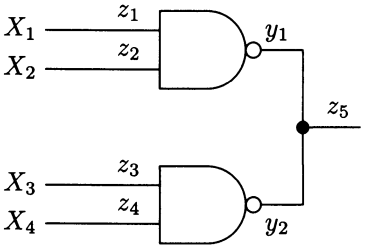


FIGURE 4.11. Circuit with two NAND gates with wired-AND outputs.

Figure 4.12 we illustrate how a "virtual" AND gate is added. Thus, in forming the circuit graph, we add a new "gate" vertex, y_3, with vertex function corresponding to the wired-AND function. To keep the circuit graph bipartite, we also add two new wire vertices, z_6 and z_7. We obtain the circuit graph shown in Figure 4.13 with input-delay vertex functions

$$X_1, \quad X_2, \quad X_3, \quad X_4,$$

gate vertex functions

$$Y_1 = \overline{z_1 z_2}, \quad Y_2 = \overline{z_3 z_4}, \quad Y_3 = z_6 z_7,$$

and wire vertex functions

$$Z_1 = x_1, \; Z_2 = x_2, \; Z_3 = x_3, \; Z_4 = x_4, \; Z_5 = y_3, \; Z_6 = y_1, \; Z_7 = y_2.$$

The dashed box in Figure 4.13 contains the added wire vertices and the new gate vertex. In effect, the connection point in the original circuit is translated into the vertices inside the box. Once the circuit graph has been obtained, a network can be derived in the usual fashion.

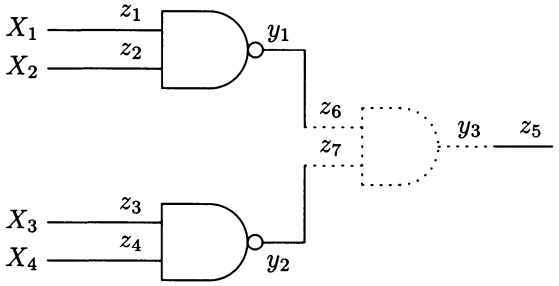


FIGURE 4.12. Virtual AND gate added to Figure 4.11.

Tri-state gates are slightly more complicated than wired-AND gates. When a tri-state gate is enabled, it is actively pulling its output to either high or low. The question arises: What happens if several tri-state gates, connected together, are enabled at the same time? If all of them agree on the

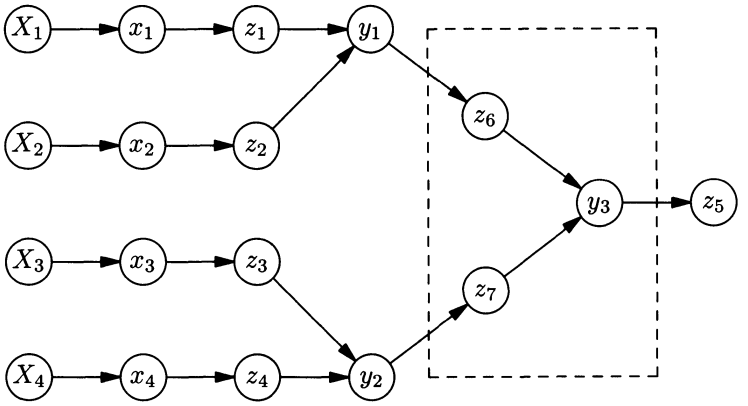


FIGURE 4.13. Circuit graph for circuit in Figure 4.11.

value, there is no problem. However, if some try to drive the output high at the same time as others try to drive it low, the resulting voltage level on the connection point is not well defined. Normally, the intention is not to have more than one gate driving the connection point. However, due to design errors—possibly caused by timing problems—we must be prepared to handle the situation with more than one driver. Our approach is to use the ternary domain and use the value Φ to indicate an undefined voltage level.

We model tri-state gates by merging connected tri-state gates into "super-gates." Each such super-gate has to incorporate all the functionality of the tri-state gates that are connected together. More specifically, assume that r tri-state gates have their outputs connected. Assume further that we can associate an enable function $\mathbf{T}_i$ with each gate. When $\mathbf{T}_i$ is low, the output of gate i is not driving the output (the gate is floating), and when $\mathbf{T}_i$ is high, the gate drives the gate output toward the value $\mathbf{G}_i$. In general, both $\mathbf{T}_i$ and $\mathbf{G}_i$ are functions of the inputs to the tri-state gate i.

For a circuit with some tri-state gates connected together, the corresponding circuit graph has only one gate vertex representing all of the tri-state gates that are connected together. The (ternary) gate function of this super-gate is

$$\left(\sum_{i=1}^{r}(\mathbf{T}_i\mathbf{G}_i)\right)\overline{\left(\sum_{i=1}^{r}(\mathbf{T}_i\overline{\mathbf{G}_i})\right)} + \left(\overline{\left(\sum_{i=1}^{r}(\mathbf{T}_i\overline{\mathbf{G}_i})\right)\overline{\left(\sum_{i=1}^{r}(\mathbf{T}_i\mathbf{G}_i)\right)}}\right)\Phi.$$

One verifies that this ternary function is 1 if at least one gate is driving the output high and no gate is driving it low. Similarly, the output will be low when at least one gate is driving the output low and no gate is driving it high. In all other cases the gate function will take on the value Φ.

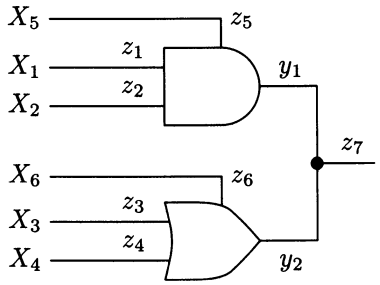


FIGURE 4.14. Circuit with two tri-state gates.

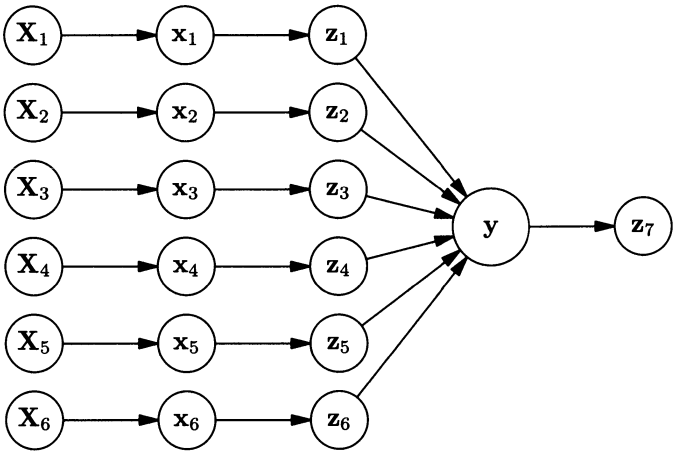


FIGURE 4.15. Circuit graph for circuit in Figure 4.14.

To illustrate how tri-state gates can be modeled, consider the gate circuit in Figure 4.14, where the $\mathbf{T}$ and $\mathbf{G}$ functions are

$$\mathbf{T}_1 = \mathbf{z}_5, \quad \mathbf{T}_2 = \mathbf{z}_6, \quad \mathbf{G}_1 = \mathbf{z}_1\mathbf{z}_2, \quad \mathbf{G}_2 = \mathbf{z}_3 + \mathbf{z}_4.$$

We obtain the circuit graph shown in Figure 4.15 with input-delay vertex functions $\mathbf{X}_1$, $\mathbf{X}_2$, $\mathbf{X}_3$, $\mathbf{X}_4$, $\mathbf{X}_5$, $\mathbf{X}_6$, gate vertex function

$$\begin{aligned}\mathbf{Y} \;=\; & \Big((\mathbf{z}_1\mathbf{z}_2\mathbf{z}_5 + \mathbf{z}_6(\mathbf{z}_3 + \mathbf{z}_4))\,\overline{\Big(\mathbf{z}_5\overline{(\mathbf{z}_1\mathbf{z}_2)} + \mathbf{z}_6\overline{(\mathbf{z}_3 + \mathbf{z}_4)}\Big)}\Big) \\ & + \overline{\Big(\Big(\mathbf{z}_5\overline{(\mathbf{z}_1\mathbf{z}_2)} + \mathbf{z}_6\overline{(\mathbf{z}_3 + \mathbf{z}_4)}\Big)\,\overline{(\mathbf{z}_1\mathbf{z}_2\mathbf{z}_5 + \mathbf{z}_6(\mathbf{z}_3 + \mathbf{z}_4))}\Big)}\,\Phi,\end{aligned}$$

and wire vertex functions

$$\mathbf{Z}_1 = \mathbf{x}_1, \; \mathbf{Z}_2 = \mathbf{x}_2, \; \mathbf{Z}_3 = \mathbf{x}_3, \; \mathbf{Z}_4 = \mathbf{x}_4, \; \mathbf{Z}_5 = \mathbf{x}_5, \; \mathbf{Z}_6 = \mathbf{x}_6, \; \mathbf{Z}_7 = \mathbf{y}.$$

Once the circuit graph has been obtained, a network can be derived in the usual fashion.

Chapter 5

CMOS Transistor Circuits

The theory that has been developed so far has been presented in terms of gates, albeit very general gates, i.e., components capable of realizing arbitrary Boolean functions. In practice, most VLSI circuits are implemented with MOS transistors as basic building blocks. Unfortunately, the theory developed for gates is not adequate for many MOS transistor circuits. In this chapter we show how these circuits can be modeled in our general framework.

How To Read This Chapter

In Sections 5.1 to 5.4 we introduce a basic switch-level model. Sections 5.5 and 5.6 refine this basic model to handle more complex circuit designs and can be omitted on first reading. Finally, in Section 5.7 we show how these switch-level models can be used to derive network models similar to the ones we described in Chapter 4.

5.1 CMOS Cells

In this section, we show how certain Boolean functions can be implemented by CMOS circuits called "cells"; more general circuits are considered later.

The fundamental components used in MOS (metal-oxide semiconductor) VLSI (very large scale integration) circuits are the "N-channel and P-channel field effect transistors"; we refer to them simply as N-transistors and P-transistors. These two types of transistors are used in the CMOS (complementary metal-oxide semiconductor) technology. Although the physical and electronic theory of such devices is quite involved [141], it is possible to use relatively simple mathematical models [10, 22, 27, 122] to capture the basic logical properties of circuits constructed with such components.

An *N-transistor* is a three-terminal device, which is represented by the diagram of Figure 5.1. The terminals t_1 and t_2 are called the *channel* terminals of the transistor and constitute the *switch* of the transistor. The state of the switch is controlled by the signal present at the so-called *gate* terminal X. When the voltage on X is low, no channel exists and the switch between t_1 and t_2 is open. When the voltage on X is high, a channel exists between t_1 and t_2, i.e., the switch is closed. If t_1 is connected to a low voltage, the signal at t_2 also becomes low. In case t_1 is connected to a high voltage, the signal at t_2 is also high, but is not as "strong" as the signal at t_1. The reason for this is that the voltage at t_2 will not be the same as that

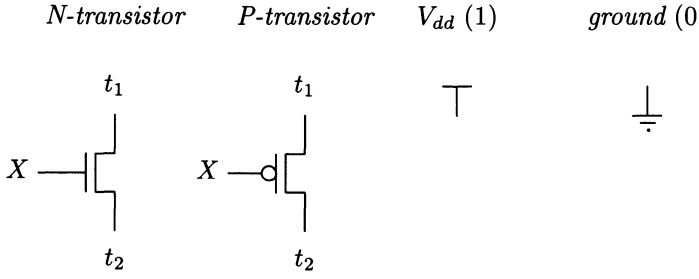


FIGURE 5.1. Transistor symbols.

at t_1, but rather V_T volts lower, where V_T, called the *threshold voltage*, is a parameter determined by the process technology and is typically between 0.5 V and 1.5 V. Hence, if a high voltage is connected through several closed N-transistors, the resulting output voltage may not have sufficient magnitude to be treated as high. In summary, when $X = 0$, an N-transistor is an open switch; when $X = 1$, it is a closed switch that transmits 0's well and 1's poorly.

Figure 5.1 also shows the symbol of a *P-transistor*. The small circle at the gate terminal denotes complementation. When $X = 1$, a P-transistor is an open switch; when $X = 0$, it is a closed switch that transmits 1's well and 0's poorly.

Finally, Figure 5.1 also shows our symbol for the supply voltage V_{dd} (logical 1), and ground (logical 0).

We define a *CMOS cell* [27] to be any circuit of N- and P-transistors with the properties below. We use the example of Figure 5.2 to illustrate the details of the definition.

The channel terminals of all the transistors are first (conceptually) connected to each other in any fashion. For example, in Figure 5.2(a) we have connected six transistors. The connection points so formed, and the remaining unconnected channel terminals, are called the *nodes* of the cell. In the example of Figure 5.2(a), we have formed five nodes labeled $A, \ldots, E$.

In the second (conceptual) step of constructing the cell, one of the nodes is connected to the constant input 0 (i.e., ground, as shown in Figure 5.1) and another to the constant input 1 (i.e., V_{dd}, as shown in Figure 5.1); these nodes are called the *supply* nodes. Suppose there are n external (variable) inputs. Some $k \leq n$ nodes (which are not supply nodes) are next chosen, and each of them is connected to a distinct external variable input $X_i, 1 \leq i \leq n$; these nodes are called *input* nodes. One of the remaining nodes is selected to be the *output* node. In the example, Figure 5.2(b), node D has been connected to V_{dd}, node E to ground, and node A to the external input X_2. Node B has been chosen as the output node; this is shown by the outgoing arrow. Nodes that are not supply, input, or output nodes are called *internal*. In the example, only node C is internal.

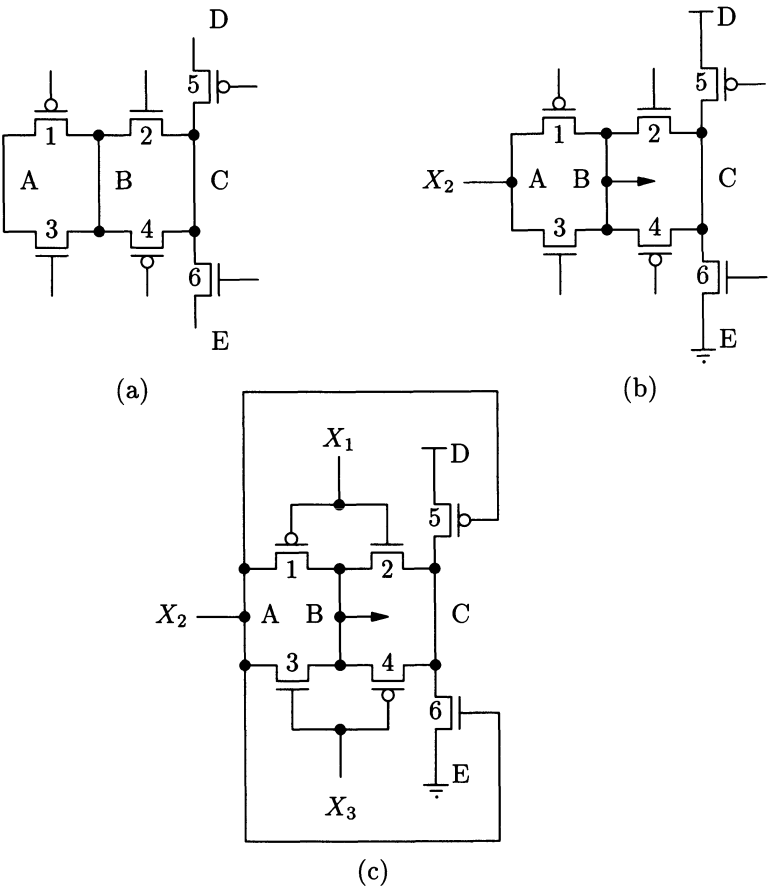


FIGURE 5.2. Illustrating the definition of a cell.

The gate terminal of each transistor is connected to exactly one input X_i. This means that if a node labeled X_i exists in the cell, then all the gate terminals that are to be controlled by X_i are connected to that node. However, if no input node labeled X_i exists, the gate terminals of all transistors that are to be controlled by X_i are connected to the input X_i, but *this input does not constitute a node of the cell.* The connections of inputs to gate terminals are shown in Figure 5.2(c) for our example.

It should be noted that the definition of a cell involves implicitly some assumptions about delays, namely, that the delays in the wires connecting several gate terminals to the same input are approximately the same and very small compared to other delays. Of course, when we later allow the inputs to a cell to be the outputs of other cells, the delays in the wires connecting cells can have delays associated with them. However, the "local" wiring is assumed to be essentially delay-free. This assumption follows the

one made in [10], where a cell is called a *channel-connected subnetwork*. We follow this approach here to simplify our discussion. Conceptually, it is easy to extend the ideas presented in this chapter to include local wire delays.

In drawing circuit diagrams, it is sometimes convenient *not* to show all the connections, in order to improve the clarity of the diagram. Thus, we may show two terminals labeled 0 or three terminals labeled X_i. It is then understood that the common label implies the existence of a connection.

We begin by considering *static* CMOS circuits; other types of CMOS circuits are mentioned later. Static CMOS circuits use N-transistors to transmit 0's and P-transistors to transmit 1's. An example of such a circuit is shown in Figure 5.3. An intuitive explanation for the working of the cell is as follows: The inputs are X_1 and X_2 and the output is the signal at node y. When either X_1 or X_2 or both are 1, y is not connected to 1 (V_{dd}), because at least one of the P-transistors is an open switch; however, y is connected to 0 (ground) through one or both of the N-transistors. Thus y becomes 0. In case X_1 and X_2 are both 0, y is connected to 1 through the two P-transistors in series, and it is not connected to 0. Hence y becomes 1. Altogether, the circuit performs the NOR function.

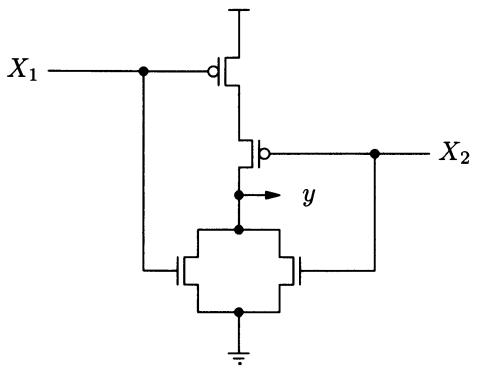


FIGURE 5.3. CMOS NOR gate.

A *binary input state* is a binary vector of length n. It follows that, for each binary input state, each transistor is either a closed or an open switch, i.e., the state of each transistor is well defined. A *path* in a transistor circuit is a sequence of connected transistor switches that does not go through a supply node. Note that paths can start and/or end in supply nodes, but the intermediate nodes in a path must be either internal or output nodes. An *N-path* is any path of N-transistor switches that are closed, i.e., any path of N-transistors with a 1 on their gate terminals. A *P-path* is any path of P-transistor switches that are closed, i.e., any path of P-transistors with a 0 on their gate terminals. Finally, an *M-path* (M for "mixed") is any path of N- or P-transistor switches that are closed.

For the cell output y, the following Boolean functions of the circuit inputs are defined in terms of paths. (The symbol g is used for "good" paths and m for "mixed" paths.)

$g_0 = 1$ if and only if there is an N-path from y to an input or supply node with value 0, i.e., if and only if there is good path to 0;

$g_1 = 1$ if and only if there is a P-path from y to an input or supply node with value 1, i.e., if and only if there is a good path to 1;

$m_0 = 1$ if and only if there is an M-path from y to an input or supply node with value 0, i.e., if and only if there is a path to 0;

$m_1 = 1$ if and only if there is an M-path from y to an input or supply node with value 1, i.e., if and only if there is a path to 1.

In a general CMOS cell, we distinguish the following three cases for the output y

$y = 0$ if and only if $g_0 = 1$ and $m_1 = 0$, i.e., if and only if there is a good path to 0 and no path to 1;

$y = 1$ if and only if $g_1 = 1$ and $m_0 = 0$, i.e, if and only if there is a good path to 1 but no path to 0;

$y = \Phi$, otherwise.

The significance of the three output values above is explained with the aid of several examples. The circuit of Figure 5.3 is a cell in which the output can only be 0 or 1 for every binary input state; such cells are called *Boolean.* A more complex Boolean cell is shown in Figure 5.4. Here $g_0 = m_0 = X_1X_2 + X_3(X_1 + X_2)$ and $g_1 = m_1 = \overline{X_1}(\overline{X_2} + \overline{X_3}) + \overline{X_2}\ \overline{X_3}$. One verifies that $g_0 = m_0 = \overline{g_1} = \overline{m_1}$, and that the cell is indeed Boolean.

The value $y = \Phi$ covers a number of cases. First, if y is connected neither to 0 nor to 1, it is said to be floating. Because of capacitance, such a floating node "remembers" its previous state by charge storage. With time, however, the stored charge may "leak out," leading to an uncertain signal value. Thus we assign Φ to such a node. Second, if y is connected to 1 only through "mixed" paths containing both N- and P-transistors and is not connected to 0, the output value is a "weak" 1. Hence the output is classified as Φ. A similar situation exists if the output is connected to 0 through mixed paths only and is not connected to 1. Finally, suppose that y is connected to both 0 and 1 at the same time; this condition is sometimes called a *fight.* In practice, transistors are not ideal switches but have some resistance. Therefore the output voltage will have a value intermediate between those corresponding to 0 and 1. This is also considered undesirable from the logical point of view, and we assign Φ to y. This approach represents a

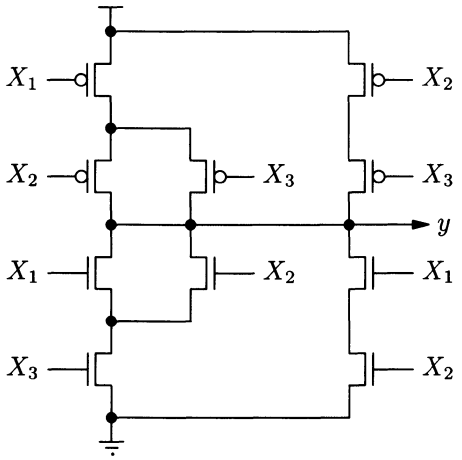


FIGURE 5.4. Cell for minority function.

rather strict set of design rules. Other, less stringent, sets of rules are also in use; we consider such variations in Sections 5.3–5.6.

A cell is said to be *redundant* if it has a node that can be removed without affecting the output value, or if it has a transistor that can be replaced by either an open or a short circuit without affecting the cell output. A cell that is not redundant is called *irredundant.*

A Boolean function f is said to be *positive* if there is a sum of products of uncomplemented variables that denotes f. For example, the majority function of three variables is positive because it can be denoted by $X_1X_2 + X_2X_3 + X_3X_1$, whereas the XOR function is not positive. A function is *negative* if its complement is positive. One can verify that a function is negative if and only if there is a sum of products of complemented variables that denotes it. Note that a function may be neither positive nor negative; for example, the XOR function is neither positive nor negative.

In Figure 5.5 the *P-part* is any network consisting entirely of P-transistors. The *N-part* is defined similarly. A cell is said to be *separated* if it has the form shown in Figure 5.5, where there are no input nodes (thus the inputs can only be connected to gate terminals of transistors); otherwise, it is *non-separated.* Note that $g_0 = m_0$ and $g_1 = m_1$ in a separated cell. A separated cell is Boolean if and only if $g_0 = \overline{g_1}$.

Several basic properties of Boolean cells are stated below without proof. The reader may construct the proofs as an exercise or see [27].

Proposition 5.1 *If an irredundant cell is Boolean, then it has no input nodes.*

Proposition 5.2 *If a cell is Boolean, then g_0 is positive and g_1 is negative.*

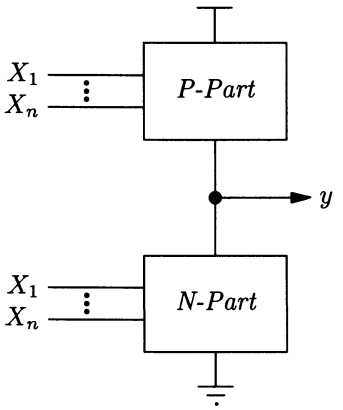


FIGURE 5.5. Form of separated cell.

Proposition 5.3 *Every negative Boolean function can be implemented by a separated cell that has as few transistors as any minimal nonseparated cell realizing the function.*

In summary, every Boolean cell implements a negative function and every negative function can be implemented by a separated cell, as shown in Figure 5.5, with the minimal number of transistors and with no input nodes.

5.2 Combinational CMOS Circuits

In this section we briefly discuss the realization of arbitrary Boolean functions by a CMOS circuit with several CMOS cells [27]. We have seen that any negative function can be realized by a single cell.[1]

It is well known that any Boolean function can be realized using inverters to produce complemented inputs, and two levels of NAND gates to implement a sum-of-products expression for the function. For example, consider the XOR function. We have $f = \overline{X_1}X_2 + X_1\overline{X_2}$. Let $y_1 = \overline{X_1}$ and $y_2 = \overline{X_2}$; these functions can be realized by two CMOS inverters with two transistors each. Now let $y_3 = \overline{(y_1 X_2)}$ and $y_4 = \overline{(X_1 y_2)}$; the y_3 and y_4 functions can be realized by two CMOS NAND gates with four transistors each. Finally the function f can be realized by another two-input CMOS NAND gate, since $f = \overline{y_3} + \overline{y_4} = \overline{(y_3 y_4)}$. Altogether, we have the circuit of Figure 5.6 with a total of 16 transistors.

[1]This is true in principle; in practice, if the cell is too large, it may have to be decomposed into smaller cells for performance reasons. However, such topics are outside the scope of this book.

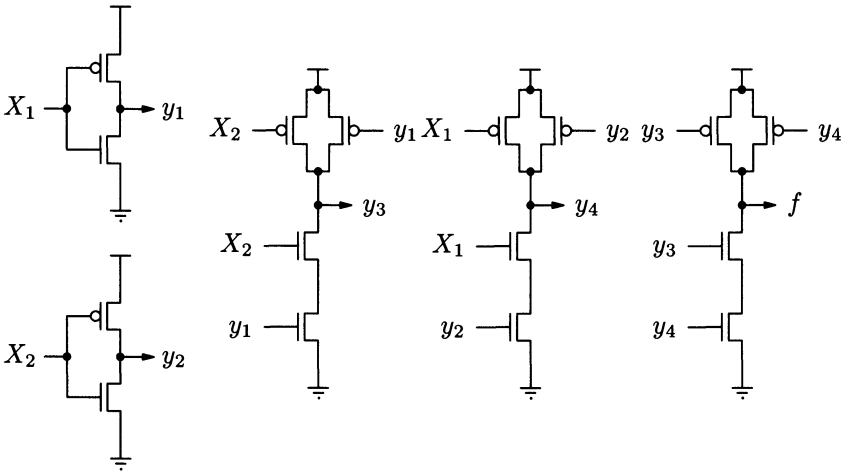


FIGURE 5.6. A CMOS circuit for XOR.

The number of transistors in a circuit is a very rough indication of the area that the circuit will occupy on a chip, because wires and empty space usually take much more room than the transistors themselves. Nevertheless, the number of transistors does give some indication of the complexity of a circuit. A realization of the XOR function with 10 transistors can be obtained from a different decomposition of the function into negative functions, namely $f = \overline{(X_1 X_2 + y)}$, where $y = \overline{(X_1 + X_2)}$.

A still more economical (in terms of transistors) implementation of the XOR function can be obtained using so-called transmission gates [27, 88, 94, 141]. A *transmission gate* controlled by X consists of an N-transistor

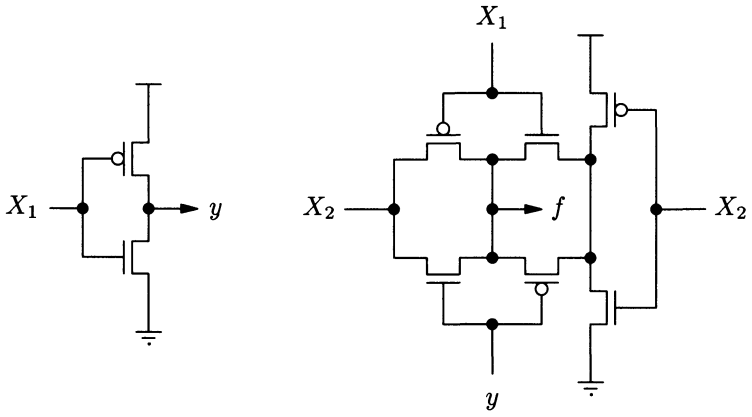


FIGURE 5.7. XOR with transmission gates.

controlled by X in parallel with a P-transistor controlled by $\overline{X}$. Figure 5.7 shows a circuit with two transmission gates. The left cell is an inverter producing $y = \overline{X_1}$. The right cell is not Boolean for, when $X_1 = X_2 = y = 1$, there is a path from f to both 0 and 1. However, when the two cells operate together, this condition can only arise as a transient condition, and the circuit does realize the XOR function properly.

Our first two examples of CMOS implementations of the XOR function have direct analogies to gate circuits, whereas the third one does not. For a more detailed discussion of combinational CMOS circuits see [27]. In the next section we generalize the concept of CMOS cell in such a way that it also includes sequential circuits.

5.3 General CMOS Circuits

We now consider general CMOS circuits and their corresponding models in which transistors are considered as controlled switches. The first formal "switch-level" model was introduced by Bryant [10] and such models were further studied in [12, 13]. The use of ternary methods for the detection of timing problems in CMOS circuits was considered in [8, 22, 27, 83, 122]. We follow here the approach of [22, 122].

A general CMOS circuit is defined like a CMOS cell, except that any transistor gate terminal may be connected either to an external input or to an internal node. An internal node that is connected to the gate of one or more transistors is called a *key internal* node. Internal and key internal nodes are labeled with *internal* and *key internal variables.*

To illustrate the definition above, consider the circuit of Figure 5.8. The circuit has supply nodes 0 and 1—connections should be supplied between different versions of the same node in the diagram—and it has no input nodes. It is convenient to number the ground node 0, the supply node (V_{dd}) 1, and the remaining nodes $2, 3, \ldots$. Thus the three internal nodes are labeled y_2, y_3, and y_4. Nodes y_2 and y_4 are key internal nodes, because each controls the gate terminals of two transistors; node y_3 is not a key internal node.

We now define several CMOS models, each reflecting different types of design rules and accuracy requirements. Consider a circuit with n input variables and m internal variables, of which k are key internal variables. A ternary *input-key state* is a length-$(n + k)$ vector of 0's, 1's, and Φ's, associating a value with each input node and each key internal node. A ternary *total state* is a length-$(n+m)$ vector of 0's, 1's, and Φ's, associating a value with each input node and each internal node. N-, P-, and M-paths are defined as in Section 5.1. An N^{Φ}-path is a path of N-transistors that is not an N-path and in which each gate terminal has either a 1 or a Φ associated with it. Thus each transistor in an N^{Φ}-path is either closed or

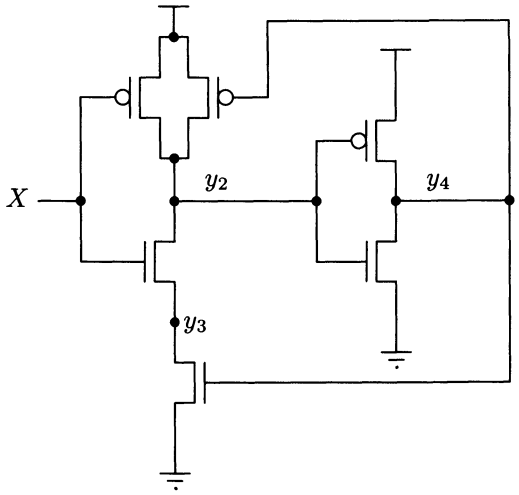


FIGURE 5.8. Circuit $C_{5.8}$.

its state is uncertain because of a Φ on its gate terminal, and there is at least one transistor with such an uncertain state. P^{Φ}-paths and M^{Φ}-paths are defined similarly.

We use the convention that nodes in circuits are labeled with symbols in italics (e.g., X, y_2, etc.). In the corresponding ternary model, however, the same vertices are shown in bold face (e.g., $\mathbf{X}$, $\mathbf{y}_2$, etc.) to stress their ternary domain. Given these conventions, for any pair of nodes i and j, $i \neq j$, we define the following *ternary* path functions, which depend on the current input and key internal node values:

$$\mathbf{p}_{ij} = \begin{cases} 1 \text{ if there is a P-path from } i \text{ to } j, \\ \Phi \text{ if there is no P-path but a } \mathrm{P}^{\Phi}\text{-path from } i \text{ to } j, \\ 0 \text{ if there is no P-path or } \mathrm{P}^{\Phi}\text{-path from } i \text{ to } j; \end{cases}$$

$$\mathbf{n}_{ij} = \begin{cases} 1 \text{ if there is an N-path from } i \text{ to } j, \\ \Phi \text{ if there is no N-path but an } \mathrm{N}^{\Phi}\text{-path from } i \text{ to } j, \\ 0 \text{ if there is no N-path or } \mathrm{N}^{\Phi}\text{-path from } i \text{ to } j; \end{cases}$$

$$\mathbf{m}_{ij} = \begin{cases} 1 \text{ if there is an M-path from } i \text{ to } j, \\ \Phi \text{ if there is no M-path but an } \mathrm{M}^{\Phi}\text{-path from } i \text{ to } j, \\ 0 \text{ if there is no M-path or } \mathrm{M}^{\Phi}\text{-path from } i \text{ to } j. \end{cases}$$

It is convenient to extend this definition by postulating that $\mathbf{p}_{ii} = \mathbf{n}_{ii} = \mathbf{m}_{ii} = 1$, for every node i. It should be noted that all path functions are symmetric, i.e., $\mathbf{p}_{ij} = \mathbf{p}_{ji}$ for every node i and j.

To illustrate the concepts of path functions, consider the the example of Figure 5.8. It is straightforward to verify the following path functions:

$$\mathbf{n}_{20} = \mathbf{X}\mathbf{y}_4, \quad \mathbf{p}_{20} = 0, \quad \mathbf{m}_{20} = \mathbf{X}\mathbf{y}_4,$$

$$\mathbf{n}_{21} = 0, \quad \mathbf{p}_{21} = \overline{\mathbf{X}} + \overline{\mathbf{y}}_4, \quad \mathbf{m}_{21} = \overline{\mathbf{X}} + \overline{\mathbf{y}}_4,$$

$$\mathbf{n}_{23} = \mathbf{X}, \quad \mathbf{p}_{23} = 0, \quad \mathbf{m}_{23} = \mathbf{X},$$

$$\mathbf{n}_{30} = \mathbf{y}_4, \quad \mathbf{p}_{30} = 0, \quad \mathbf{m}_{30} = \mathbf{y}_4,$$

$$\mathbf{n}_{31} = 0, \quad \mathbf{p}_{31} = 0, \quad \mathbf{m}_{31} = \mathbf{X}(\overline{\mathbf{X}} + \overline{\mathbf{y}_4}),$$

$$\mathbf{n}_{40} = \mathbf{y}_2, \quad \mathbf{p}_{40} = 0, \quad \mathbf{m}_{40} = \mathbf{y}_2,$$

and

$$\mathbf{n}_{41} = 0, \quad \mathbf{p}_{41} = \overline{\mathbf{y}_2}, \quad \mathbf{m}_{41} = \overline{\mathbf{y}_2}.$$

5.4 Node Excitation Functions

Given the basic path functions derived in the previous section, we can define a variety of different types of models applicable to CMOS circuits, depending on the details of the CMOS technology used and also on the design philosophy. We start by deriving some very simple models and then gradually expand these basic ideas.

The strictest set of CMOS design rules corresponds to the following definition of an (internal) *node excitation function*: Given an input-key state or a total state, a node excitation is declared to be 0 (1) if and only if that node is connected to 0 (1) through an N-path (P-path) and is not connected to 1 (0); in all other cases, the excitation is declared to be Φ, which represents an undefined signal. Using the ternary algebra of Section 2.3, we can denote the excitation function by the expression

$$\mathbf{Y}_i = \mathbf{p}_{i1}\overline{\mathbf{m}_{i0}} + \overline{(\mathbf{p}_{i1}\overline{\mathbf{m}_{i0}} + \mathbf{n}_{i0}\overline{\mathbf{m}_{i1}})}\Phi,$$

i.e., the node i is excited to 1 if there is a good connection (P-path) to node 1 (corresponding to V_{dd}) and no path to node 0 (corresponding to ground); it is excited to 0 if there is a good connection (N-path) to 0 and no connection to 1; and it is excited to Φ if it is not the case that it is excited to either 0 or 1. The reader should verify that the expression above does indeed correspond to the desired function. According to Proposition 2.2, it is possible to simplify this expression to

$$\mathbf{Y}_i = \mathbf{p}_{i1}\overline{\mathbf{m}_{i0}} + \overline{(\mathbf{n}_{i0}\overline{\mathbf{m}_{i1}})}\Phi.$$

For the example of Figure 5.8, we find the following excitation functions:

$$\begin{aligned}\mathbf{Y}_2 &= \mathbf{p}_{21}\overline{\mathbf{m}_{20}} + \overline{(\mathbf{n}_{20}\overline{\mathbf{m}_{21}})}\Phi \\ &= (\overline{\mathbf{X}} + \overline{\mathbf{y}}_4)\overline{(\mathbf{X}\mathbf{y}_4)} + \overline{(\mathbf{X}\mathbf{y}_4(\overline{\mathbf{X}} + \overline{\mathbf{y}}_4))}\Phi \\ &= \overline{\mathbf{X}} + \overline{\mathbf{y}}_4,\end{aligned}$$

$$\begin{aligned}\mathbf{Y}_3 &= \mathbf{p}_{31}\overline{\mathbf{m}_{30}} + \overline{(\mathbf{n}_{30}\overline{\mathbf{m}_{31}})}\Phi \\ &= 0\overline{\mathbf{y}_4} + \overline{(\mathbf{y}_4\overline{(\mathbf{X}(\overline{\mathbf{X}} + \overline{\mathbf{y}_4}))})}\Phi \\ &= (\mathbf{X}\overline{\mathbf{X}} + \overline{\mathbf{y}_4})\Phi = \mathbf{X}\overline{\mathbf{X}} + \overline{\mathbf{y}_4}\Phi,\end{aligned}$$

and

$$\begin{aligned}\mathbf{Y}_4 &= \mathbf{p}_{41}\overline{\mathbf{m}_{40}} + \overline{(\mathbf{n}_{40}\overline{\mathbf{m}_{41}})}\Phi \\ &= \overline{\mathbf{y}_2}\,\overline{\mathbf{y}_2} + \overline{(\mathbf{y}_2\overline{\mathbf{y}_2})}\Phi = \overline{\mathbf{y}_2}.\end{aligned}$$

Note that the node excitation defined above can be written in the form $\mathbf{u}_i + \overline{(\mathbf{u}_i + \mathbf{z}_i)}\Phi$, or simplified to $\mathbf{u}_i + \overline{\mathbf{z}_i}\Phi$, for some functions $\mathbf{u}_i$ (for unity) and $\mathbf{z}_i$ (for zero). Here, $\mathbf{u}_i$ denotes the conditions under which $\mathbf{Y}_i$ is 1, and $\mathbf{z}_i$ gives the conditions under which $\mathbf{Y}_i$ is 0. We can now generalize the model above and define four different basic models:

Model 1 $\quad \mathbf{Y}_i = \mathbf{p}_{i1}\overline{\mathbf{m}_{i0}} + \overline{(\mathbf{n}_{i0}\overline{\mathbf{m}_{i1}})}\Phi,$

Model 2 $\quad \mathbf{Y}_i = \mathbf{p}_{i1}\overline{\mathbf{n}_{i0}} + \overline{(\mathbf{n}_{i0}\overline{\mathbf{p}_{i1}})}\Phi,$

Model 3 $\quad \mathbf{Y}_i = \mathbf{m}_{i1}\overline{\mathbf{m}_{i0}} + \overline{(\mathbf{m}_{i0}\overline{\mathbf{m}_{i1}})}\Phi,$

Model 4 $\quad \mathbf{Y}_i = \mathbf{p}_{i1}\overline{\mathbf{n}_{i0}} + \mathbf{m}_{i1}\overline{\mathbf{m}_{i0}} + \overline{(\mathbf{n}_{i0}\overline{\mathbf{p}_{i1}} + \mathbf{m}_{i0}\overline{\mathbf{m}_{i1}})}\Phi.$

Model 1 is the one we introduced above. It was originally introduced in [27]. Model 2 assumes that a P-path to 1 (a "good" path) is stronger than any path to 0 containing at least one P-transistor (a "bad" path) (and vice versa for N-paths). We get $\mathbf{Y}_i = 1$ (0) as long as there is at least one good path to 1 (0), but no good path to 0 (1). Hence, a fight between a good path and a bad path is resolved in favor of the good path. This is a substantially more liberal rule, but may be necessary to explain certain very tricky designs [27]. Model 3 is more traditional and corresponds to a special case of the model in [10]. Here there is no distinction at all between P- and N-paths, and we have $\mathbf{Y}_i = 1$ (0) if and only if there is some mixed path to 1 (0) but no mixed path to 0 (1). Model 4 is a combination of Models 2 and 3. Here the rules are: good paths override bad paths; if there are no good paths, the bad paths determine the output.

To illustrate these four basic models, consider again the circuit of Figure 5.8. It is easy to convince oneself that $\mathbf{Y}_2 = \overline{\mathbf{X}} + \overline{\mathbf{y}_4}$ and that $\mathbf{Y}_4 = \overline{\mathbf{y}_2}$ for every model. However, for node $\mathbf{y}_3$, Models 1–4 yield the following node excitation functions:

$$\begin{aligned}
\text{Model 1:}\quad \mathbf{Y}_3 &= \mathbf{p}_{31}\overline{\mathbf{m}_{30}} + \overline{(\mathbf{n}_{30}\overline{\mathbf{m}_{31}})}\Phi = \\
&= (\mathbf{X}\overline{\mathbf{X}} + \overline{\mathbf{y}_4})\Phi\ , \\
\text{Model 2:}\quad \mathbf{Y}_3 &= \mathbf{p}_{31}\overline{\mathbf{n}_{30}} + \overline{(\mathbf{n}_{30}\overline{\mathbf{p}_{31}})}\Phi = \\
&= \overline{\mathbf{y}_4}\Phi, \\
\text{Model 3:}\quad \mathbf{Y}_3 &= \mathbf{m}_{31}\overline{\mathbf{m}_{30}} + \overline{(\mathbf{m}_{30}\overline{\mathbf{m}_{31}})}\Phi = \\
&= \mathbf{X}\overline{\mathbf{y}_4} + (\mathbf{X}\overline{\mathbf{X}} + \overline{\mathbf{y}_4})\Phi, \\
\text{Model 4:}\quad \mathbf{Y}_3 &= \mathbf{p}_{31}\overline{\mathbf{n}_{30}} + \mathbf{m}_{31}\overline{\mathbf{m}_{30}} + \overline{(\mathbf{n}_{30}\overline{\mathbf{p}_{31}} + \mathbf{m}_{30}\overline{\mathbf{m}_{31}})}\Phi = \\
&= \mathbf{X}\overline{\mathbf{y}_4} + \overline{\mathbf{y}_4}\Phi.
\end{aligned}$$

5.5 Path Strength Models

So far we have assumed that all the transistors have the same conductance (except that the conductance for transmitting a 1 may be different from the conductance for transmitting a 0). Sometimes it is useful to use transistors with significantly different conductances. We can represent this in a switch-level model by assuming that the transistors have different "strengths." We assume that there is a finite number of strengths denoted by the integers $1, \ldots, q$. A transistor of strength r has a conductance that is an order of magnitude higher than that of a transistor of strength p if and only if $r > p$. We also need to add the notion of strength to the path functions. A path is of strength s if all the transistors in the path have strength $\geq s$. Let $\mathbf{p}_{ij}^s$, $\mathbf{n}_{ij}^s$, and $\mathbf{m}_{ij}^s$ denote the path functions of strength s. A signal from a strong path overrides a signal from a weaker path. Model 1, extended to handle the different transistor strengths, gives the following functions:

$$\mathbf{u}_i = \overline{(\mathbf{m}_{i0}^q)}(\mathbf{p}_{i1}^q + \overline{(\mathbf{m}_{i0}^{q-1})}(\mathbf{p}_{i1}^{q-1} + \cdots + \overline{(\mathbf{m}_{i0}^1)}(\mathbf{p}_{i1}^1)\cdots))$$

and

$$\mathbf{z}_i = \overline{(\mathbf{m}_{i1}^q)}(\mathbf{n}_{i0}^q + \overline{(\mathbf{m}_{i1}^{q-1})}(\mathbf{n}_{i0}^{q-1} + \cdots + \overline{(\mathbf{m}_{i1}^1)}(\mathbf{n}_{i0}^1)\cdots)).$$

To illustrate these ideas, consider the CMOS circuit of Figure 5.9. The circuit is designed in a slight variation of the domino style [141]. Instead of relying on charge storage, a "weak" inverter provides feedback for the circuit so that it exhibits the appropriate behavior. The transistors have been assigned strengths as shown in Figure 5.9. Using this refined model, after simplifications, we get the following functions for the key internal nodes $\mathbf{y}_2$ and $\mathbf{y}_5$:

$$\mathbf{u}_2 = \overline{(\mathbf{m}_{20}^2)}(\mathbf{p}_{21}^2 + \overline{(\mathbf{m}_{20}^1)}(\mathbf{p}_{21}^1)) = \overline{\mathbf{X}_1} + (\overline{\mathbf{X}_2} + \overline{\mathbf{X}_3})\overline{\mathbf{y}_5},$$

$$\mathbf{z}_2 = \overline{(\mathbf{m}_{21}^2)}(\mathbf{n}_{20}^2 + \overline{(\mathbf{m}_{21}^1)}(\mathbf{n}_{20}^1)) = \mathbf{X}_1\mathbf{X}_2\mathbf{X}_3 + \mathbf{X}_1\mathbf{y}_5,$$

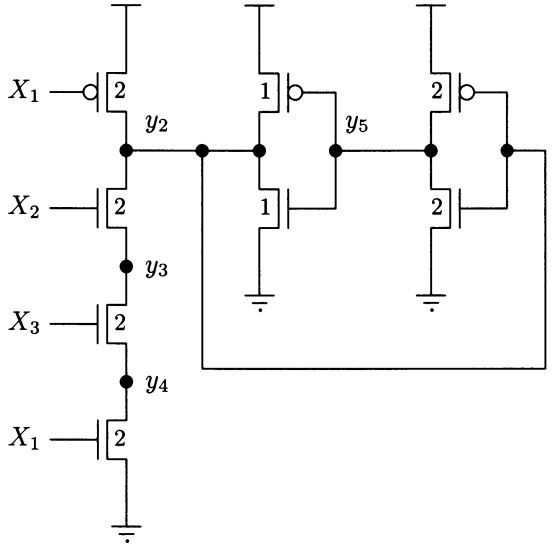


FIGURE 5.9. CMOS domino gate with "staticizer."

$$\mathbf{u}_5 = \overline{(\mathbf{m}_{50}^2)}(\mathbf{p}_{51}^2 + \overline{(\mathbf{m}_{50}^1)}(\mathbf{p}_{51}^1)) = \overline{\mathbf{y}_2},$$

$$\mathbf{z}_5 = \overline{(\mathbf{m}_{51}^2)}(\mathbf{n}_{50}^2 + \overline{(\mathbf{m}_{51}^1)}(\mathbf{n}_{50}^1)) = \mathbf{y}_2,$$

which yield the node excitation functions

$$\mathbf{Y}_2 = \mathbf{u}_2 + \overline{\mathbf{z}_2}\Phi = \overline{\mathbf{X}_1} + (\overline{\mathbf{X}_2} + \overline{\mathbf{X}_3})\overline{\mathbf{y}_5},$$
$$\mathbf{Y}_5 = \mathbf{u}_5 + \overline{\mathbf{z}_5}\Phi = \overline{\mathbf{y}_2}.$$

The model above is often sufficient for CMOS circuits with more than one transistor strength, but it is sometimes overly pessimistic. In the CMOS circuit in Figure 5.10 every path to node 3 is of strength 1. Consider the

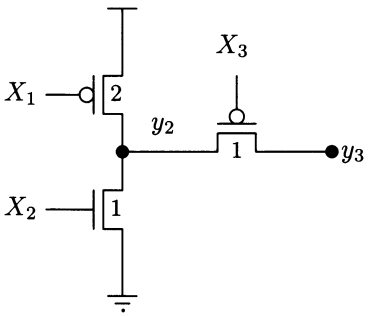


FIGURE 5.10. Illustration of path blocking.

situation when $\mathbf{X}_1 = 0$, $\mathbf{X}_2 = 1$, and $\mathbf{X}_3 = 0$. There is a P-path of strength 1 to the power supply, but also a mixed path of strength 1 to ground. Using the definition above, we conclude that $\mathbf{Y}_3 = \Phi$. At the same time, the excitation of node $\mathbf{y}_2$ is 1, because the strong path through the P-transistor overrides the weak path through the N-transistor. If we reexamine the path functions used in calculating the excitation of $\mathbf{y}_3$, we notice that they all go through node 2. From an electrical point of view, it is now reasonable to say that the path from ground to node 3 is *blocked* by the strong path from node 2 to supply node 1. This observation, that some weak paths can be blocked by stronger paths "along the way," can be formalized and incorporated in the excitation functions. We refer the interested reader to [12, 13] where not only is such a model defined, but very efficient methods for deriving the excitation functions are also given.

5.6 Capacitance Effects

The node excitation functions derived in the previous section fail to capture the fact that there is a certain amount of capacitance in CMOS circuits. In particular, the key internal nodes have a capacitance associated with them; hence there is some "memory" in each such node. The case when there is only one key internal node in every cell can be handled in a very straightforward way. Here we make two assumptions: 1) The capacitance of a key internal node is much larger than the capacitance of any other node in the cell. 2) A 1 (0) stored on a key internal node can only determine the excitation of that node if the node is completely isolated from the supply nodes. One can verify that the following modifications to Models 1–4 take this into account:

$$\text{Model } 1^{\text{M}} \quad \mathbf{Y}_i = (\mathbf{p}_{i1} + \mathbf{y}_i)\overline{\mathbf{m}_{i0}} + \overline{((\mathbf{n}_{i0} + \overline{\mathbf{y}_i})\overline{\mathbf{m}_{i1}})}\Phi,$$

$$\text{Model } 2^{\text{M}} \quad \mathbf{Y}_i = \mathbf{p}_{i1}\overline{\mathbf{n}_{i0}} + \mathbf{y}_i\overline{\mathbf{m}_{i0}} + \overline{(\mathbf{n}_{i0}\overline{\mathbf{p}_{i1}} + \overline{\mathbf{y}_i}\ \overline{\mathbf{m}_{i1}})}\Phi,$$

$$\text{Model } 3^{\text{M}} \quad \mathbf{Y}_i = \mathbf{m}_{i1}\overline{\mathbf{m}_{i0}} + \mathbf{y}_i\overline{\mathbf{m}_{i0}} + \overline{(\mathbf{m}_{i0}\overline{\mathbf{m}_{i1}} + \overline{\mathbf{y}_i}\ \overline{\mathbf{m}_{i1}})}\Phi,$$

$$\text{Model } 4^{\text{M}} \quad \mathbf{Y}_i = \mathbf{p}_{i1}\overline{\mathbf{n}_{i0}} + \mathbf{m}_{i1}\overline{\mathbf{m}_{i0}} + \mathbf{y}_i\overline{\mathbf{m}_{i0}},$$
$$+ \overline{(\mathbf{n}_{i0}\overline{\mathbf{p}_{i1}} + \mathbf{m}_{i0}\overline{\mathbf{m}_{i1}} + \overline{\mathbf{y}_i}\ \overline{\mathbf{m}_{i1}})}\Phi.$$

For example, using model 1^{M}, Y is 1 if there is a good path to 1 and no path to 0, or if the previous value was 1 and there is no path to 0. A dual situation holds for $Y = 0$.

To illustrate the idea above, consider the CMOS circuit of Figure 5.11. The circuit is a two-input AND gate implemented in (simple) domino CMOS technology [141]. A domino CMOS circuit works as follows. There are two phases, a precharge phase and an evaluation phase. In the *precharge* phase, the clock signal (called $\mathbf{X}_1$ in our example) is set to 0 causing the high-capacitance node $\mathbf{y}_2$ to be driven to a high voltage through the P-transistor.

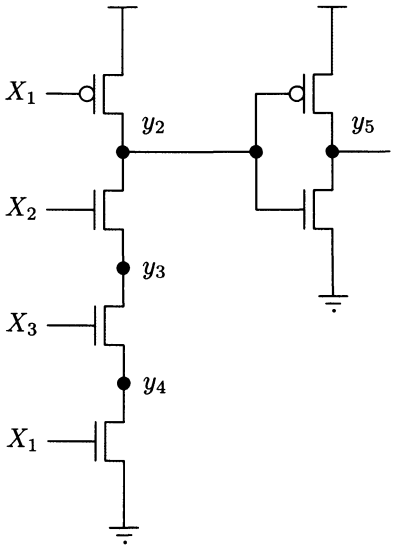


FIGURE 5.11. A two-input AND gate in CMOS domino style.

Note that this happens irrespective of the values of $\mathbf{X}_2$ and $\mathbf{X}_3$ because the bottommost N-transistor is not conducting. In the second phase, the *evaluation* phase, the clock signal is switched to 1. Node $\mathbf{y}_2$ will keep its high value unless it is connected to ground. That will happen only if $\mathbf{X}_2$ and $\mathbf{X}_3$ are both high. Our first model for this circuit assumes that node $\mathbf{y}_2$ has a capacitance that is an order of magnitude larger than those of nodes $\mathbf{y}_3$ and $\mathbf{y}_4$. Using model 1^{M} we get the following node excitation functions for the key internal nodes $\mathbf{y}_2$ and $\mathbf{y}_5$:

$$\begin{aligned}
\mathbf{Y}_2 &= (\mathbf{p}_{21} + \mathbf{y}_2)\overline{\mathbf{m}_{20}} + \overline{((\mathbf{n}_{20} + \overline{\mathbf{y}_2})\overline{\mathbf{m}_{21}})}\Phi \\
&= \ldots = \overline{\mathbf{X}_1} + \mathbf{y}_2(\overline{\mathbf{X}_2} + \overline{\mathbf{X}_3}), \\
\mathbf{Y}_5 &= (\mathbf{p}_{51} + \mathbf{y}_5)\overline{\mathbf{m}_{50}} + \overline{((\mathbf{n}_{50} + \overline{\mathbf{y}_5})\overline{\mathbf{m}_{51}})}\Phi \\
&= \ldots = \overline{\mathbf{y}_2}.
\end{aligned}$$

Note that, for a binary input-key state, the excitations of the key internal nodes are binary. We can generalize the concept of a Boolean cell from Section 5.1 to general CMOS circuits by defining a circuit to be Boolean if and only if, for every binary input-key state, the excitations can only be 0 or 1.

When there is more than one key internal node in each cell, a more complicated node excitation function must be used, because charge can "spill over" from one node to another causing the excitation to become unde-

fined. From here on, we assume that charge sharing comes into effect only when the nodes are isolated from the supply nodes. This is a reasonable assumption since paths connected to the supply nodes can provide essentially infinite amounts of charge. Below, we give the definitions of the functions $\mathbf{u}_i$ and $\mathbf{z}_i$ for the excitation function $\mathbf{Y}_i = \mathbf{u}_i + \overline{\mathbf{z}_i}\Phi$. Let $\mathcal{C}_i$ denote the set of key internal nodes in the cell containing node i. Model 1, extended to handle the case of multiple key internal nodes, gives the following functions

$$\begin{aligned}
\mathbf{u}_i &= \mathbf{p}_{i1}\overline{\mathbf{m}_{i0}} + \overline{\mathbf{m}_{i0}}\left(\prod_{j\in\mathcal{C}_i}(\mathbf{y}_j + \overline{\mathbf{m}_{ij}})\right)\left(\sum_{j\in\mathcal{C}_i}\mathbf{y}_j\mathbf{p}_{ij}\right) \\
&= \mathbf{p}_{i1}\overline{\mathbf{m}_{i0}} + \overline{\mathbf{m}_{i0}}\mathbf{y}_i\prod_{j\in\mathcal{C}_i}(\mathbf{y}_j + \overline{\mathbf{m}_{ij}}), \\
\mathbf{z}_i &= \mathbf{n}_{i0}\overline{\mathbf{m}_{i1}} + \overline{\mathbf{m}_{i1}}\left(\prod_{j\in\mathcal{C}_i}(\overline{\mathbf{y}_j} + \overline{\mathbf{m}_{ij}})\right)\left(\sum_{j\in\mathcal{C}_i}\overline{\mathbf{y}_j}\mathbf{n}_{ij}\right) \\
&= \mathbf{n}_{i0}\overline{\mathbf{m}_{i1}} + \overline{\mathbf{m}_{i1}\mathbf{y}_i}\prod_{j\in\mathcal{C}_i}(\overline{\mathbf{y}_j} + \overline{\mathbf{m}_{ij}}).
\end{aligned}$$

The basic idea is as follows. We only discuss the $\mathbf{u}_i$ function, but the arguments can be trivially extended to the $\mathbf{z}_i$ function. First, charge sharing comes into effect only when there are no paths to ground, i.e., $\mathbf{m}_{i0} = 0$. Second, if a node i is disconnected from the supply nodes, then all the key internal nodes connected via some path to i must have the value 1 in order to cause the excitation of i to be 1. This is captured in the second half of the formula. Either the value of a key internal node j must be 1 or node j must be disconnected from node i. Moreover, at least one of the nodes must be 1 and must be connected to i by a P-path in order to get $\mathbf{u}_i = 1$. (In the formulas above we assumed that $\mathbf{y}_i$ is a key internal node; hence $\mathbf{m}_{ii} = \mathbf{p}_{ii} = \mathbf{n}_{ii} = 1$, and the last equalities in the formulas follow.)

Node excitation functions for the other basic CMOS models are derived similarly. For example, for $\mathbf{u}_i$ in a model based on Model 2, one replaces $\mathbf{p}_{i1}\overline{\mathbf{m}_{i0}}$ by $\mathbf{p}_{i1}\overline{\mathbf{n}_{i0}}$.

In the discussion above only the key internal nodes were assumed to have memory. Furthermore, it was postulated that all the key internal nodes have the same "size." A common technique in CMOS circuits is the use of precharged lines, where certain nodes are designed with a substantially higher capacitance than that of all the other nodes. This can be modeled as if these nodes had "greater size" than normal nodes. To describe such excitation functions, we need the following notation. Let $\mathcal{C}_i^s$, $1 \le s \le q$, be the set of all nodes of size s in the cell that contains node i. Assume further that the sizes are totally ordered, so that a node in $\mathcal{C}_i^p$ is substantially bigger (has a substantially higher capacitance) than a node in $\mathcal{C}_i^r$ if and

only if $p > r$. Model 1, extended to handle multiple key internal nodes and different node sizes, gives the following functions

$$\begin{aligned}
\mathbf{u}_i \;=\; & \mathbf{p}_{i1}\overline{\mathbf{m}_{i0}} + \overline{\mathbf{m}_{i0}}\left(\prod_{j\in\mathcal{C}_i^q}(\mathbf{y}_j + \overline{\mathbf{m}_{ij}})\left(\sum_{j\in\mathcal{C}_i^q}\mathbf{y}_j\mathbf{p}_{ij}\right.\right. \\
& + \prod_{j\in\mathcal{C}_i^{q-1}}(\mathbf{y}_j + \overline{\mathbf{m}_{ij}})\left(\sum_{j\in\mathcal{C}_i^{q-1}}\mathbf{y}_j\mathbf{p}_{ij}\right. \\
& + \cdots \\
& \left.\left.\left. + \prod_{j\in\mathcal{C}_i^1}(\mathbf{y}_j + \overline{\mathbf{m}_{ij}})\left(\sum_{j\in\mathcal{C}_i^1}\mathbf{y}_j\mathbf{p}_{ij}\right)\cdots\right)\right)\right)
\end{aligned}$$

and

$$\begin{aligned}
\mathbf{z}_i \;=\; & \mathbf{n}_{i0}\overline{\mathbf{m}_{i1}} + \overline{\mathbf{m}_{i1}}\left(\prod_{j\in\mathcal{C}_i^q}(\overline{\mathbf{y}_j} + \overline{\mathbf{m}_{ij}})\left(\sum_{j\in\mathcal{C}_i^q}\overline{\mathbf{y}_j}\mathbf{n}_{ij}\right.\right. \\
& + \prod_{j\in\mathcal{C}_i^{q-1}}(\overline{\mathbf{y}_j} + \overline{\mathbf{m}_{ij}})\left(\sum_{j\in\mathcal{C}_i^{q-1}}\overline{\mathbf{y}_j}\mathbf{n}_{ij}\right. \\
& + \cdots \\
& \left.\left.\left. + \prod_{j\in\mathcal{C}_i^1}(\overline{\mathbf{y}_j} + \overline{\mathbf{m}_{ij}})\left(\sum_{j\in\mathcal{C}_i^1}\overline{\mathbf{y}_j}\mathbf{n}_{ij}\right)\cdots\right)\right)\right)
\end{aligned}$$

where we have used the convention that $\prod_{j\in\mathcal{C}_i^r}\ldots$ is equal to 1 if $\mathcal{C}_i^r = \emptyset$ and similarly that $\sum_{j\in\mathcal{C}_i^r}\ldots$ is equal to 0 if $\mathcal{C}_i^r = \emptyset$.

Once again, the basic idea is quite simple. First, the node must be isolated from the supply nodes before charge sharing effects should be considered. Furthermore, for a node of strength r with value 1 to be relevant to the node i, either every stronger node must be isolated from i or the stronger nodes that are connected to i must have the value 1 (thus making the value of the node of strength r irrelevant anyway).

It should be pointed out that the models above constitute only a few of the very many possibilities. For example, we required that a stored 1 on a node must drive the output node through a P-path in order to transmit a 1 properly. In some cases it may be appropriate to relax this condition and allow any kind of path. The derivation of such a model is left as an exercise for the interested reader.

Finally, it is possible now to combine the models of the previous section with the models in this section and obtain a "universal" model that captures path blocking, different transistor strengths, and node sizes. Again, we refer the interested reader to [12, 13].

5.7 Network Model of CMOS Circuits

The previous sections have shown how to derive ternary node excitation functions for general CMOS circuits. Given that there are many possible switch-level models to use in deriving the node excitation functions, there are many circuit graphs possible for any given CMOS circuit. Following our treatment of gate circuits in Chapter 4 we define the circuit graph to reflect the circuit topology. The network models reflect the behavioral properties of the circuit.

Given a CMOS circuit, the corresponding *circuit graph* is a 5-tuple $G = \langle \mathcal{X}, \mathcal{I}, \mathcal{T}, \mathcal{N}, \mathcal{E} \rangle$, where

- $\mathcal{X}$ is a set of *input vertices*, labeled $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n$,
- $\mathcal{I}$ is a set of *input-delay vertices*, labeled $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$,
- $\mathcal{T}$ is a set of *transistor vertices*, labeled $\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_p$,
- $\mathcal{N}$ is a set of *node vertices*, labeled $\mathbf{y}_2, \mathbf{y}_3, \ldots, \mathbf{y}_m$,
- $\mathcal{E} \subseteq (\mathcal{X} \times \mathcal{I}) \cup ((\mathcal{I} \cup \mathcal{N}) \times (\mathcal{T} \cup \mathcal{N})) \cup (\mathcal{T} \times \mathcal{N})$ is a set of *edges*.

All input vertices have indegree 0, all input-delay vertices have indegree and outdegree 1, and all the transistor vertices have indegree 1.

Given a CMOS circuit, we obtain its circuit graph as follows. First, there is an input vertex $\mathbf{X}_i$ for every external input X_i, a transistor vertex $\mathbf{t}_i$ for every transistor, and a node vertex $\mathbf{y}_i$ for every node in the circuit except for the supply nodes. For every input vertex $\mathbf{X}_i$ there is an input-delay vertex $\mathbf{x}_i$ and an edge from $\mathbf{X}_i$ to $\mathbf{x}_i$. There is an edge from every node vertex $\mathbf{y}$ to every node vertex $\mathbf{y}'$ in the same cell. Note that this includes an edge from $\mathbf{y}$ to $\mathbf{y}$. There is an edge from the transistor vertex $\mathbf{t}$ to every node vertex in the cell that contains $\mathbf{t}$. There is also an edge from input-delay or node vertex i to transistor $\mathbf{t}$ if the gate terminal of the transistor is connected to this input delay or node.

We illustrate the concept of a circuit graph using the CMOS circuit of Figure 5.12. The corresponding circuit graph, with $\mathcal{X} = \{\mathbf{X}_1\}$, $\mathcal{I} = \{\mathbf{x}_1\}$, $\mathcal{N} = \{\mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4\}$, and $\mathcal{T} = \{\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3, \mathbf{t}_4, \mathbf{t}_5, \mathbf{t}_6\}$, is shown in Figure 5.13.

As for gate circuits, a circuit graph is a convenient and precise notation for describing how the components of a CMOS circuit are connected—it is a structural representation of the circuit. To study its behavior, we need to add a domain. The domain for CMOS circuits is always the ternary domain, $\{0, 1, \Phi\}$, because CMOS circuits often have undefined node excitations for certain (usually transient) binary input values. Hence, even for binary circuits, we must be prepared to handle excitations that are Φ.

For an input-delay vertex $\mathbf{x}$, the vertex function maps a state of the (undefined) environment to the vertex domain $\mathcal{D}$. This function is called

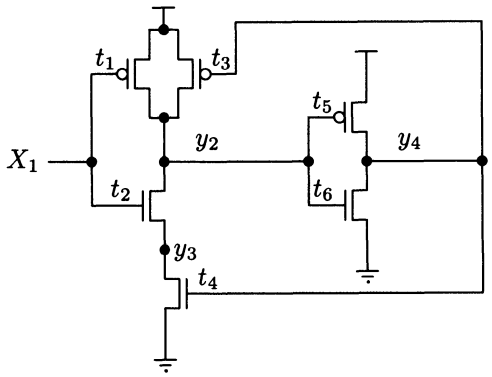


FIGURE 5.12. CMOS circuit C.

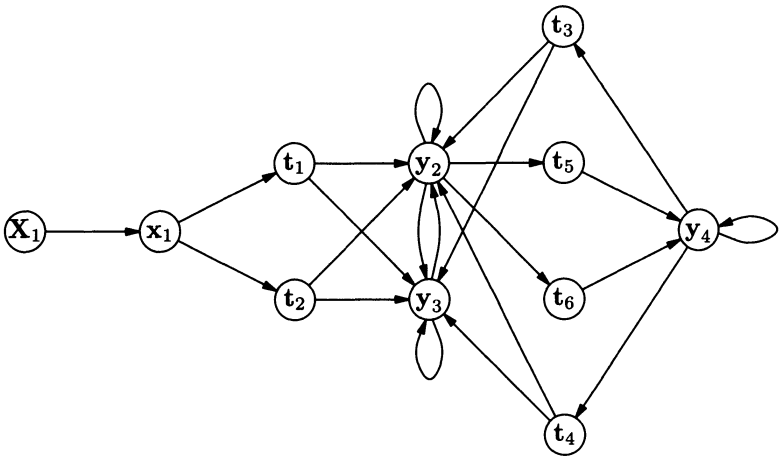


FIGURE 5.13. Circuit graph of CMOS circuit in Figure 5.12.

X. As in gate circuits, **X** is the input value provided by the environment, and the variable **x** holds the input value "seen" by the circuit.

For a transistor vertex, there are two cases. If the transistor is of the N type, the vertex function $\mathbf{T}: \mathcal{D}^{|\mathcal{I}|+|\mathcal{N}|} \rightarrow \mathcal{D}$ is the value of the input or component vertex connected to the gate terminal of the transistor. On the other hand, if the transistor is of P-type, the vertex function $\mathbf{T}: \mathcal{D}^{|\mathcal{I}|+|\mathcal{N}|} \rightarrow \mathcal{D}$ is the complement of the value of the input or component vertex connected to the gate terminal of the transistor.

For a node vertex, the vertex function **Y** maps a transistor-vertex state to $\mathcal{D}$, i.e., $\mathbf{Y}: \mathcal{D}^{|\mathcal{I}|+|\mathcal{N}|} \rightarrow \mathcal{D}$. The actual function here is determined from the topology of the cell containing the vertex as well as from the switch-level model used, as discussed in the previous section. This function is used

to represent the behavior of the collection of transistors in the cell and can be quite complex. However, by Theorem 2.3, it always satisfies the monotonicity property

$$\mathbf{s} \sqsubseteq \mathbf{t} \text{ implies } \mathbf{f}(\mathbf{s}) \sqsubseteq \mathbf{f}(\mathbf{t}),$$

for all ternary $\mathbf{s}$, $\mathbf{t}$.

To illustrate the concepts introduced so far, consider again the circuit shown in Figure 5.12 with circuit graph shown in Figure 5.13. The input-delay vertex function is $\mathbf{X}_1$. The transistor vertex functions are

$$\mathbf{T}_1 = \overline{\mathbf{x}_1},\ \mathbf{T}_2 = \mathbf{x}_1,\ \mathbf{T}_3 = \overline{\mathbf{y}_4},\ \mathbf{T}_4 = \mathbf{y}_4,\ \mathbf{T}_5 = \overline{\mathbf{y}_2},\ \mathbf{T}_6 = \mathbf{y}_2.$$

Assuming we use the strict CMOS Model 1 from Section 5.4, we obtain the node vertex functions

$$\mathbf{Y}_2 = (\mathbf{t}_1 + \mathbf{t}_3)(\overline{\mathbf{t}_2} + \overline{\mathbf{t}_4}) + \overline{(\mathbf{t}_2\mathbf{t}_4\overline{\mathbf{t}_1\mathbf{t}_3})}\Phi,$$

$$\mathbf{Y}_3 = \overline{(\mathbf{t}_4(\overline{\mathbf{t}_2} + \overline{\mathbf{t}_1\mathbf{t}_3}))}\Phi,$$

and

$$\mathbf{Y}_4 = \mathbf{t}_5\overline{\mathbf{t}_6} + \overline{(\overline{\mathbf{t}_5}\mathbf{t}_6)}\Phi.$$

As in gate circuits, the vertex functions defined above introduce a distinction between the present value of a vertex variable and the present value of the "excitation" of that vertex variable, i.e., the value computed by the vertex function. This permits us to associate a delay with every input, every transistor, and every node in the circuit.

To represent the state of the entire circuit, we need to select a set of *state variables.* As for gate circuits, by changing the set of state variables, we effectively change the locations of the assumed delays in the circuit. Clearly, the circuit graph model easily permits us to select *all* of the vertex variables as state variables. We may think of such a model as the *input-, transistor-, and node-state* model. As we shall see, this model is too detailed for many applications, and simpler models are often preferred. In particular, we often use an *input- and key-state* model.

As for gate circuits, having selected a set of vertices from the circuit graph to act as state variables, we associate with each such vertex two distinct items: the vertex variable and its "excitation function." The excitation function of a vertex in the state variable set is obtained as follows. We start with the vertex function. We then repeatedly remove all dependencies on vertices that have not been chosen as state variables, by using functional composition of the vertex functions.

Chapter 6

Up-Bounded-Delay Race Models

In this chapter we describe a formal model for the analysis of the behavior of asynchronous circuits modeled by networks in which the delays are inertial and have only upper bounds. The analysis is limited to a single transition: Suppose the network is in a given state and the input is kept constant. We would like to know what is the "outcome" of the transition, i.e., what are the possible states of the network a "long" time after the input change, i.e., after the "transients have died down." The analysis of the circuit behavior in response to a *sequence* of input changes can then be carried out as a series of such transition analyses. We postpone the discussion of the response to an input sequence until Chapters 11–13.

Our basic analysis model corresponds to the classical binary "race analysis," that has been in use for many years [66, 67, 135]. These methods were originally rather informal, but were formalized in 1979 in [26] as the "general multiple-winner" (GMW) model. In particular, the concept of outcome was formally defined there. We describe the GMW model in some detail in this chapter.

Next, we begin studying how the analysis is affected by the choice of state variables or, equivalently, of the delay locations in the network model. We show in Section 6.4 that any set of feedback variables is sufficient for the purpose of calculating the stable total states of a network. In Section 6.5, however, we demonstrate with the aid of several examples that the outcome of a transition depends on the choice of the state variables.

We also consider several variations of the basic model. First, we extend the GMW model to the ternary domain, thus obtaining the extended multiple-winner (XMW) model. Second, we briefly mention "single-winner" models in which only one state variable can change at a time. Third, we briefly discuss an ideal-delay model.

The models mentioned above are intuitively appealing, but computationally intractable for large circuits. In Chapter 7 we describe efficient algorithms that produce most of the results of interest obtainable from the binary race models.

How To Read This Chapter

There are several results in this chapter with rather technical proofs. These proofs have been grouped together as Section 6.9, and can be omitted on first reading.

6.1 The General Multiple-Winner Model

When gates are implemented by physical circuits, it is not possible to guarantee that the delays of two gates of the same type are exactly the same, even if it is the designer's intention to make them the same. Moreover, if one succeeded in producing two equal delays, they could become unequal as a result of such factors as aging, radiation, changes in temperature, etc. The behavior of a well-designed circuit should not change if the delay of one of its components deviates slightly from its nominal value. Thus, we have to accept the possibility that gate delays may be unequal. It is impractical to try to measure all the gate delays in a large circuit; consequently, we need an analysis model in which the exact delay values are not known. Similar remarks apply to wire delays. These considerations lead us to an analysis technique that examines *all* the possible relative delay values.

This section, which represents a formalization of the methods used earlier by many researchers (e.g., [66, 67, 135]), is based on [23, 25, 26]. For our present purposes, we do not refer to the circuit from which the network model has been derived, for we develop methods for the analysis of the network itself. Recall that a network has the form

$$N = \langle \mathcal{D}, \mathcal{X}, \mathcal{S}, \mathcal{E}, F \rangle,$$

where $\mathcal{D}$ is the domain, $\mathcal{X}$ is the set of input excitation vertices labeled $X_1, \ldots, X_n$, $\mathcal{S}$ is the set of state vertices with two sets of labels: state variable labels $(s_1, \ldots, s_m)$ and the corresponding excitation functions $(S_1, \ldots, S_m)$, $\mathcal{E}$ is the set of edges, and F is a vector of circuit equations that, given a state of the network, compute the implied values of all the components in the original circuit.

A *total state* $c = a{\cdot}b$ is an $(n + m)$-tuple of values from $\mathcal{D}$, the first n values (the n-tuple a) being the values of the input excitations, and the remaining m values (the m-tuple b) being the values of the variables $s_1, \ldots, s_m$, which we refer to as *(internal) state* variables. Given the total state $a{\cdot}b$ of a network, the *circuit state* consists of the values computed by the circuit equations for this total state.

In any total state $c = a{\cdot}b$, we define the set of unstable state variables as

$$\mathcal{U}(a{\cdot}b) = \{s_i \mid b_i \neq S_i(a{\cdot}b)\}.$$

Thus state c is *stable* if and only if $\mathcal{U}(c) = \emptyset$.

Until further notice, we use the binary domain in all the networks. We wish to know how the circuit behavior will evolve when it is started in a given initial state and the input is kept constant. For this purpose we define a binary relation R_a on the set $\{0,1\}^m$ of internal states of N for every input vector $a \in \{0,1\}^n$:

For any $b \in \{0,1\}^m$,

bR_ab, if $\mathcal{U}(a{\cdot}b) = \emptyset$, i.e., the total state $a{\cdot}b$ is stable,

$bR_ab^{\mathcal{K}}$, if $\mathcal{U}(a{\cdot}b) \neq \emptyset$, and $\mathcal{K}$ is any nonempty subset of $\mathcal{U}(a{\cdot}b)$,

where by $b^{\mathcal{K}}$ we mean b with all the variables in $\mathcal{K}$ complemented. No other pairs of states are related by R_a. The relation R_a is called the *general multiple-winner* (*GMW*) relation for the reasons explained below.

An internal state in which more that one state variable is unstable is called a *race*. In any race the GMW model permits any nonempty subset of unstable variables to change at the same time; thus there are "multiple winners" possible. Also, the relation is called "general" because no assumptions are made about the relative values of the delays (although we do assume that all the delays are up-bounded).

We frequently depict R_a by a directed graph, drawing an edge from b to b' if bR_ab'. Such an edge indicates that b' is a possible immediate successor of b. A loop from b to b indicates that the total state $a{\cdot}b$ is stable. The graph is then a description of the possible network behaviors under the assumption that the input excitation remains constant at the value a.

A number of different types of phenomena are possible, as we now show. We denote by $R_a(b)$ that portion of the graph of the relation R_a that contains only the states reachable from b.

Race-free transition to a unique stable state

To tie together a number of concepts, we now present a complete example of analysis. We begin with the NOR latch of Figure 6.1. Its circuit graph is shown in Figure 6.2. For this example, we associate delays only with the two gates; the gate-state network obtained from

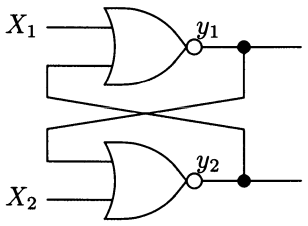


FIGURE 6.1. NOR latch.

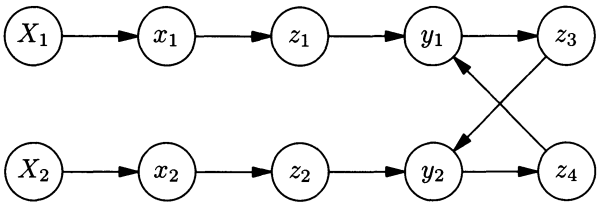


FIGURE 6.2. Circuit graph for NOR latch.

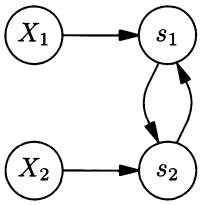


FIGURE 6.3. Gate-state network for NOR latch.

Figure 6.2 is shown in Figure 6.3, where $s_1 = y_1$ and $s_2 = y_2$. The excitation functions are

$$S_1 = \overline{(X_1 + s_2)} \text{ and } S_2 = \overline{(X_2 + s_1)}.$$

The graph of the GMW relation $R_{10}(10)$ is shown in Figure 6.4. Unstable variables are underlined in such figures. Here, in each unstable state there is only one unstable variable; thus there are no races. The unstable variable must eventually change in each case, since the delays are assumed to be finite. Furthermore, a unique stable state is reached. Hence this type of behavior presents no difficulties if we assume that our objective is to design a circuit that changes from one stable state to another as a result of an input change.

FIGURE 6.4. A race-free transition.

Noncritical race

Consider the circuit shown in Figure 6.5. In the gate-state network, we have the excitation functions $S_1 = \overline{X}$ and $S_2 = \overline{X}$. In total state 1·11 there is a race. However, the final outcome of this race is always the same, namely the stable state 1·00. The graph of $R_1(11)$ is shown in Figure 6.6. This race is *noncritical*, because its outcome is the same for all possible distributions of delays. Such a race corresponds to acceptable behavior, if we are only interested in the final stable state.

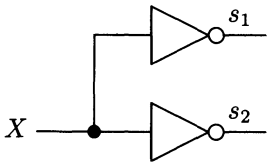


FIGURE 6.5. A circuit with a noncritical race.

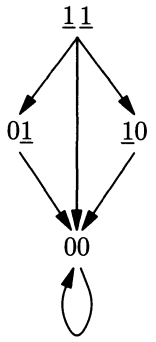


FIGURE 6.6. A noncritical race.

Critical race

Let us return to the latch of Figure 6.1 as represented by the gate-state model of Figure 6.3. In the total state 00·00, both gates are unstable (see Figure 6.7). This is a race, the final outcome of which depends on the relative values of the two delays; such a race is called *critical*. Let us consider this in more detail. Suppose the delays of the two gates are δ_1 and δ_2. If gate 1 is faster than gate 2, i.e., if $\delta_1 < \delta_2$, then gate 1 *wins the race* and the network reaches the internal state 10. The total state 00·10 is stable. The instability of gate 2 that was present in state 00·00 has been removed because the output of gate 1 changed. Gate 2 has been excited for time δ_1, a time that is shorter

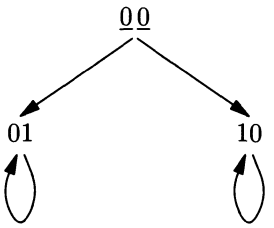


FIGURE 6.7. A critical race.

than its delay δ_2, and it has *lost the race.* The fact that gate 2 has not reacted to the short pulse of excitation reflects the inertial nature of its delay. Similar remarks apply if $\delta_1 > \delta_2$.

Race-free oscillation

Consider the network of Figure 6.8, which uses the gate-state model. The excitation function is $S = \overline{(X*s)}$. The graph of $R_1(0)$ is shown in Figure 6.9. Here, the network never reaches a stable state, but goes through a periodic succession of unstable states. Such a cycle of states is called an *oscillation.* Note, however, that there are no races; in each unstable state exactly one variable is unstable.

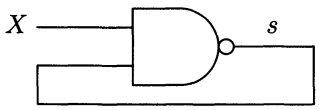


FIGURE 6.8. A circuit with an oscillation.

FIGURE 6.9. A race-free oscillation.

Match-dependent oscillation

Yet another phenomenon occurs in the NOR latch model in the gate-state network if one allows for the possibility that both delays are exactly equal. Starting in the total state 00·00, the network then moves to state 00·11. If we repeat the analysis from this state, we

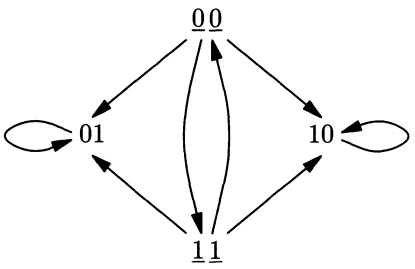


FIGURE 6.10. A match-dependent oscillation.

obtain the graph of Figure 6.10. The cycle consisting of the two states 00 and 11 is an oscillation. Oscillations may occur in physical circuits, but, of course, they will not have this idealized binary nature that is described by our very simple model. The following phenomenon also occurs in physical circuits: In the transition from state 00·00, the circuit may enter a *metastable state* in which the two outputs have an intermediate voltage value, between the voltages corresponding to the logical 0 and 1 signals [31]. Although the metastable state does not persist indefinitely, it is impossible to bound its duration. In a way, the oscillation shown above has similar properties. For the oscillation to continue, one would have to have perfectly matched delays at all times. This is highly unlikely. It is plausible that such a perfect match could exist for several cycles, and it would be difficult to predict how long the oscillation would last. We call this type of oscillation *match-dependent.* In a crude way it models the metastability phenomenon.

Transient oscillation

Figure 6.12 shows the graph of $R_1(011)$ for the gate-state network derived from the circuit of Figure 6.11. The excitation functions are $S_1 = X+s_1$, $S_2 = \overline{s_1}$, and $S_3 = \overline{(X*s_2*s_3)}$. There are two oscillations, (011,010) and (111,110). Each of these two cycles has the property that there is a variable that is unstable in both states of the cycle and has the same value in those states. In cycle (011,010), s_1 is the unstable variable in question. Since the delay of s_1 is assumed to

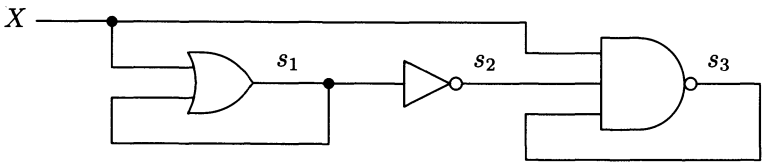


FIGURE 6.11. A circuit with transient oscillations.

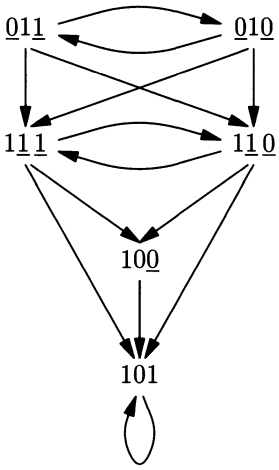


FIGURE 6.12. Illustrating transient oscillations.

be bounded from above, say by D, this oscillation cannot persist; s_1 will eventually change, causing the network to leave the cycle. We call such oscillations "transient." It is clear from Figure 6.12 that the network eventually reaches stable state 101; hence this behavior is acceptable, if one is only interested in the final stable state. In fact, from Figure 6.12 and the assumption that the delay of each component is bounded from above by D, it follows that the circuit is guaranteed to be in the state 101 after at most $3D$ time units.

From the example above, we may be tempted to say that transient cycles can always be disregarded if we are only interested in the outcome of a transition. But suppose the circuit first enters a nontransient cycle—where, by definition, it can stay for an arbitrarily long time— and then leaves this cycle and enters a transient cycle. Consider the circuit of Figure 6.13 started in stable state 1·00011 when the input changes to 0. In Figure 6.14 we show the graph $R_0(00011)$. Note that there are five cycles in the graph: two self-loops for stable

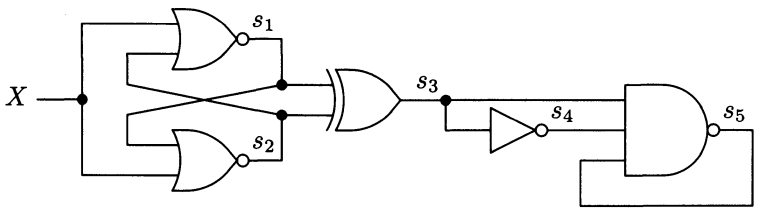


FIGURE 6.13. A circuit with transient states in the outcome.

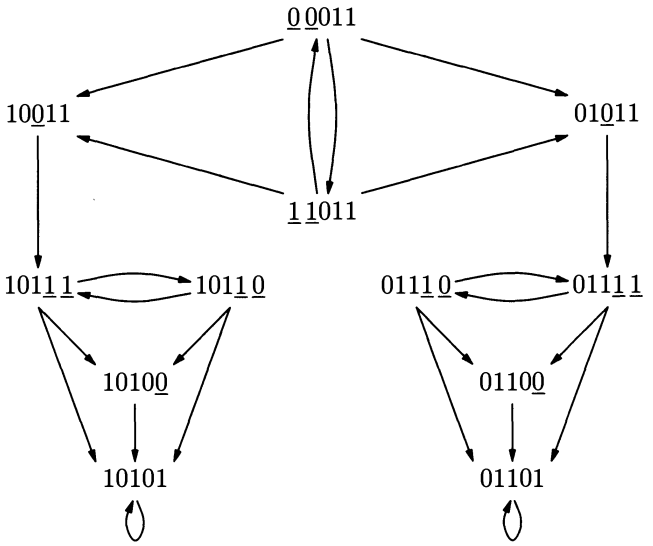


FIGURE 6.14. Transient cycles reachable from a nontransient cycle.

states 10101 and 01101, a nontransient cycle (00011, 11011), and two transient cycles (10111, 10110) and (01111, 01110). The circuit can stay in nontransient cycle (00011, 11011) for an arbitrarily long time, without violating any delay bounds. It can then move to state 10011 and later enter transient cycle (10111, 10110). This shows that it is possible for a circuit to *enter* a transient cycle after an arbitrarily long time. Such a transient cycle cannot be disregarded, and will be included in the definition of outcome of a transition.

Overlapping oscillations

The network of Figure 6.15 has the GMW behavior shown in Figure 6.16. To simplify the graph of Figure 6.16, we have used one edge

FIGURE 6.15. Two oscillators.

with two arrowheads between two states s and s' to represent an edge from s to s' and an edge from s' to s. Note that there are several transient oscillations, for example, (00, 01) and (10, 11). Note also that all four states take part in several nontransient oscillations, for example, (00, 01, 11, 10). This example illustrates the fact that two

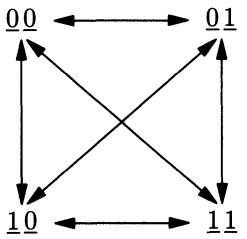


FIGURE 6.16. GMW behavior of two oscillators.

transient oscillations may be connected to each other, i.e., that it is possible for a network to be in one transient cycle for a while, then switch to a second transient cycle, then return to the first cycle, etc.

This concludes our series of examples of possible behaviors. We now formalize several concepts, so that we can treat the network analysis problem more precisely.

6.2 GMW Analysis and UIN Delays

In this section we state the result that the general multiple-winner race model captures exactly the behavior of a network under the assumption that each state variable s_j corresponds to an ideal, delay-free "gate" with excitation function $S_j(X(t)\cdot s(t))$ in series with an up-bounded inertial (UIN) delay. Throughout this chapter we assume that each delay δ_i in a network is a UIN delay bounded from above by some constant D_i, i.e., that $0 \leq \delta_i < D_i$. The maximum value of all the D_i in the network can be used as an upper bound for all the delays in the network.

Recall that a binary variable $v(t)$ is said to change at time τ if it was previously $\overline{\alpha}$ and is α at time τ, i.e., if $v(\tau) = \alpha$, and there exists a $\delta > 0$ such that $v(t) = \overline{\alpha}$, for $\tau - \delta \leq t < \tau$. Recall also that an up-bounded inertial delay with input S and output s must satisfy the following two properties:

1. *If s changes, then it must have been unstable.*
   Formally, if $s(t)$ changes from α to $\overline{\alpha}$ at time τ, then there exists $\delta > 0$ such that $S(t) = \overline{\alpha}$ for $\tau - \delta \leq t < \tau$.

2. *s cannot be unstable for D units of time without changing.*
   Formally, if $S(t) = \alpha$ for $\tau \leq t < \tau + D$, then there exists a time $\tilde{\tau}$, $\tau \leq \tilde{\tau} < \tau + D$ such that $s(t) = \alpha$ for $\tilde{\tau} \leq t < \tau + D$.

We want to show that the up-bounded inertial delay model and the GMW race model are mutually consistent. We describe this consistency

intuitively first. Suppose that, for each state variable s_j in the network, we have an input/output waveform $S_j(t)/s_j(t)$, where $s_j(t)$ is the state of the variable at time t and $S_j(t) = S_j(X(t)\cdot s(t))$ is the corresponding excitation at time t, computed from the total state $X(t)\cdot s(t)$ at time t. Suppose further that each such waveform obeys Properties 1 and 2 of UIN delays. Then we demonstrate that the state sequence obtained from the waveforms corresponds to the GMW analysis of the network. Before we can prove this claim, however, we need to define precisely the waveforms mentioned above; these waveforms are called a UIN_a-history.

Informally, a UIN_a-history of a network is a set of waveforms beginning at time t_0. The input excitation vector X is kept constant at the value a. The real numbers t_i represent the time instants at which the state vector s changes. The state is constant in any interval between t_i and t_{i+1}. The number of state changes can be either finite or infinite. If it is finite, then the last state reached must be stable. If it is infinite, then we allow only a finite number of state changes in any finite interval.

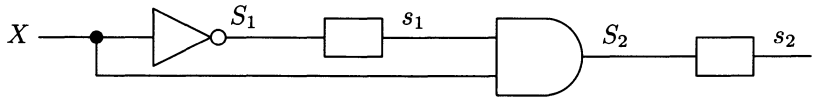


FIGURE 6.17. Network N.

The network of Figure 6.17 illustrates the concepts above. Suppose the two delays δ_1 and δ_2 are up-bounded inertial delays with upper bounds $D_1 = 4$ and $D_2 = 2$, respectively. In Figure 6.18 we show some waveforms

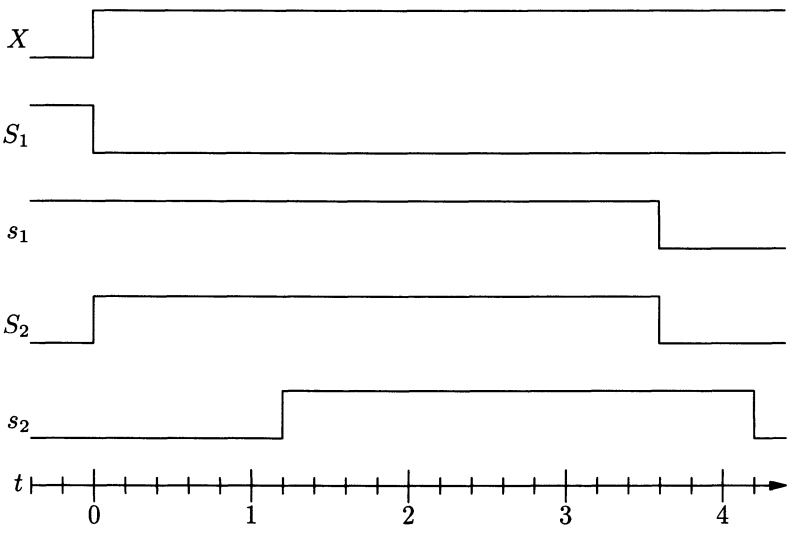


FIGURE 6.18. A UIN_1-history for N.

for the state variables s_1 and s_2 of N, along with those of the excitations S_1 and S_2. One easily verifies that these waveforms are consistent with the assumptions that δ_1 and δ_2 are inertial delays satisfying $0 \leq \delta_1 < 4$ and $0 \leq \delta_2 < 2$. The corresponding UIN_a-history μ includes (a) the sequence $\Theta = (t_0, t_1, t_2, t_3) = (0, 1.2, 3.6, 4.2)$, of consecutive instants of time at which some state change takes place; (b) the waveform of the input X, assumed to be constant for $t \geq t_0$; and (c) the waveforms of the state variables s_i. Each UIN_a-history μ determines a sequence of states. In our example, we have the sequence $\gamma(\mu) = (10, 11, 01, 00)$. It is clear that the waveforms can be uniquely reconstructed from the value a of X, the sequence Θ of time instants, and the state sequence $\gamma(\mu)$; note that the waveforms are of no interest before time t_0.

Formally, a *UIN_a-history* of a network N for some $a \in \{0,1\}^n$ is an ordered triple $\mu = \langle \Theta, X(t), s(t) \rangle$, where Θ is a strictly increasing sequence $\Theta = (t_0, t_1, \ldots)$ of real numbers, and $X(t)$ and $s(t)$ are functions, X mapping real numbers to $\{0,1\}^n$ and s mapping real numbers to $\{0,1\}^m$. These functions satisfy the following properties.

I. (a) $X(t) = a$ for all $t \geq t_0$.

(b) $s(t) = b^i$ for $t_i \leq t < t_{i+1}$, where $b^i \in \{0,1\}^m$, for all $i \geq 0$.

(c) $s(t_{i-1}) \neq s(t_i)$, for each $i \geq 1$, i.e., $s(t)$ changes at each t_i.

(d) If the sequence $(t_0, t_1, \ldots, t_r)$ is finite, then for all $t \geq t_r$ we have $s(t) = b^r$, for some $b^r \in \{0,1\}^m$ such that $a \cdot b^r$ is a stable total state of N.

(e) If the sequence $(t_0, t_1, \ldots)$ is infinite, then for every $t > 0$, there exists an i such that $t_i \geq t$. Note that this requirement implies that there is only a finite number of state changes in any finite time interval.

II. For each variable s_j, the input/output waveform $S_j(X(t) \cdot s(t))/s_j(t)$ is consistent with the assumption that variable s_j is represented by the delay-free excitation function S_j in series with an up-bounded inertial delay. In other words, the input/output waveform satisfies Properties 1 and 2 of up-bounded inertial delays.

The state sequence $\gamma(\mu)$ corresponding to a UIN_a-history μ is defined to be the sequence $(s(t_0), s(t_1), \ldots)$.

Next, we formalize the concept of a sequence of states derivable from a GMW analysis of a network. A sequence $\gamma = (s^0, s^1, \ldots)$ of states (i.e., binary m-tuples) is called an *R_a-sequence* for some $a \in \{0,1\}^n$ if and only if $s^i \neq s^{i+1}$, $s^i R_a s^{i+1}$ for $i \geq 0$, and either the sequence is infinite or the last state is stable. An infinite R_a-sequence is said to be *transient* if and only if there exists a state variable s_j and an integer $r \geq 0$ such that, for $i \geq r$, $s_j^i = \alpha \in \{0,1\}$ and $S_j(a \cdot s^i) = \overline{\alpha}$. Thus a transient sequence

contains a variable that, from some point in time, has the same value and is unstable. Such a situation can exist only if the delay of that variable is infinite. An R_a-sequence that is finite, or infinite but not transient, is said to be *nontransient.* Since we assume that all the circuit delays are bounded from above, we consider only nontransient R_a-sequences.

Continuing with our example, Figure 6.19 shows the graph of the relation $R_1(10)$ for the network of Figure 6.17. It is seen that the sequence $(10, 11, 01, 00)$ is a nontransient R_1-sequence.

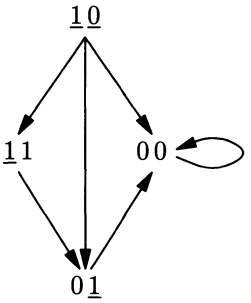


FIGURE 6.19. GMW relation $R_1(10)$ for N.

We now claim that for each UIN_a-history there is a corresponding nontransient R_a-sequence and vice versa.

Theorem 6.1 *There is a one-to-one correspondence between nontransient R_a-sequences and UIN_a-histories. In other words, the GMW analysis of a network is consistent with the up-bounded inertial delay model.*

Proof: See Section 6.9. The proof involves two lemmas. The first lemma shows that the state sequence $\gamma(\mu)$ corresponding to any UIN_a-history is always a nontransient R_a-sequence. The second lemma shows that for every R_a-sequence γ there is time sequence Θ such that the waveforms $\mu(\gamma)$ corresponding to γ and Θ constitute a UIN_a-history. □

6.3 The Outcome in GMW Analysis

As has been mentioned above, in many applications we are only interested in the "nontransient" states reached from a given state after an input change. We have called this as-yet-undefined set of states the outcome of a transition. We give a precise definition of outcome in this section.

It is not entirely clear at first glance how outcome should be defined. Since every graph of $R_a(b)$ is finite, every path from b must eventually

reach a cycle. (A stable state is a cycle of length one.) One may be tempted, therefore, to say that the network has reached its outcome when a cycle in the graph has been reached. This may not be the case, however, if the cycle is transient, as has been illustrated in the transient oscillation example and as is formally defined below.

A cycle in the relation diagram of $R_a(b)$ is *transient* if there exists a state variable s_i that has the same value in all the states of the cycle and is unstable in each state of the cycle. Let D be the maximum value of all the network delays. A network cannot stay in a given transient cycle for more than D units of time. Let the set of *cyclic states* reachable from b in the relation diagram of $R_a(b)$ be

$$cycl(R_a(b)) = \{s \in \{0,1\}^m \mid bR_a^* s \text{ and } sR_a^+ s\},$$

where R^+ is the transitive closure of R, and R^* is the reflexive and transitive closure of R. Next, define the set of nontransient cyclic states to be

$$cycl\_nontrans(R_a(b)) = \{s \mid s \text{ appears in a nontransient cycle}\}.$$

The *outcome* of the transition from b under input a is the set of all the states that appear in at least one nontransient cycle or are reachable from a state in a nontransient cycle. Mathematically, we have

$$out(R_a(b)) = \{s \mid bR_a^* c \text{ and } cR_a^* s, \text{ where } c \in cycl\_nontrans(R_a(b))\}.$$

The reason for this definition will become clearer soon. We define a state d to be *transient* if it is reachable from b (i.e., if $bR_a^* d$) but is not in the outcome $out(R_a(b))$, and to be *nontransient* if it is in $out(R_a(b))$.

Our analysis problem can now be formalized as follows: Given a total state $a{\cdot}b$, find $out(R_a(b))$. Note that this problem, as stated, is of exponential complexity because $out(R_a(b))$ can contain as many as 2^m states. However, we are able to show later that a "summarized version" of *out* can be efficiently computed. This will be done with the aid of ternary simulation, which is the topic of Chapter 7.

Before we proceed, we want to compare the outcome with the consequences of the assumption that all delays are up-bounded and inertial. Consider a network N in which all the delays are bounded by D. Let N be started in state b with the input held constant at a. A state c is said to be *D-transient* with limit τ for $a{\cdot}b$ if it is reachable from b and there exists a real number $\tau > 0$ such that in every UIN_a-history μ, the condition $t \geq \tau$ implies $s(t) \neq c$. This definition means that, as long as every delay in the network is less than D, after a certain time limit τ the network cannot be in a D-transient state. D-transient states are related to the transient states by the following result.

Proposition 6.1 *Under the conditions defined above, a state is transient if and only if it is D-transient.*

Proof: See Section 6.9. □

Recall that the fundamental mode requires a network to be in a stable state before an input change can be made. This implies that, after every input change, the environment must wait "long enough" for the network to stabilize. The next result makes precise the concept of fundamental mode operation of a network, by specifying how long is long enough.

Theorem 6.2 *Let N be any network with m up-bounded delays with upper bound D. Suppose N is in state b at time 0 and the input is held constant from time 0 until time $t \geq (2^m - 2)D$. Then the state of the network at time t is in $out(R_a(b))$.*

Proof: See Section 6.9. □

Note that the theorem applies in particular when state $a \cdot b$ is stable; consequently, this result gives a bound on the duration of constant-input intervals required for fundamental mode operation.

For every network with delays up-bounded by D, one can calculate a constant $\tau(D)$ that is less than or equal to $(2^m - 2)D$. If we start in state b and keep the input constant at a for time $\tau(D)$, we are guaranteed that the network will reach some state in $out(R_a(b))$. If $out(R_a(b))$ consists of a single stable state, we know that this stable state has been reached after time $\tau(D)$. The environment may then change the input, and a new transition begins.

6.4 Stable States and Feedback-State Networks

The first model for sequential circuit analysis was introduced by Huffman in 1954 [66, 67]. In his model, delays are associated only with a set of "feedback wires"—with the property that cutting all these wires results in the removal of all the feedback loops from the circuit. In terms of our network model, Huffman's model is a feedback-state network of Section 4.4.

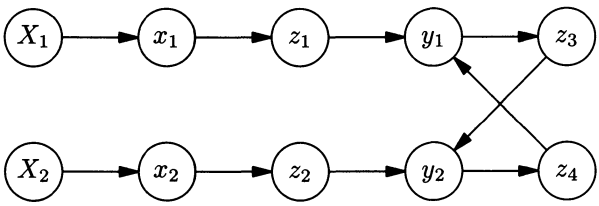


FIGURE 6.20. Input-, gate-, and wire-state network.

A feedback-state network properly describes the stable states of a circuit, as we illustrate with the example of Figure 6.1. The input-, gate-, and wire-state network is repeated in Figure 6.20. Suppose that the network is in

total state $X{\cdot}x\,y\,z = 00{\cdot}00\,01\,0001$; one verifies that this state is stable. Suppose further that we select $\{z_4\}$ as the feedback vertex set, i.e., as the state set. Then, the input and feedback variable values uniquely determine all the other variable values in the circuit. This is true because, in any stable state, each excitation function has the same value as the corresponding vertex variable. In our example we have the following circuit equations, (leaving out the equation for z_4):

$$x_1 = X_1,\ x_2 = X_2,$$

$$z_1 = Z_1 = x_1 = X_1,\ z_2 = Z_2 = x_2 = X_2,$$

$$z_3 = Z_3 = y_1 = Y_1 = \overline{(z_1 + z_4)} = \overline{(X_1 + z_4)},$$

$$y_2 = Y_2 = \overline{(z_2 + z_3)} = \overline{(X_2 + \overline{(X_1 + z_4)})}.$$

Thus, under the assumption that the network is stable, we have expressed all the vertex variables in terms of the input excitations and the feedback variables. This is always possible because, by definition, "cutting" the feedback vertices results in a feedback-free graph that corresponds to a combinational circuit. Hence, if we know the input values and the values at all the feedback vertices, we can reconstruct the rest of the vertex values.

Recall that by a circuit state we mean the vector of all the vertex variables in the circuit graph.

Proposition 6.2 *The set of stable circuit states computed using a feedback vertex set is independent of the choice of the feedback vertex set.*

Proof: Suppose a circuit state $X{\cdot}s = a{\cdot}b$ is stable. Let f and g be the vectors of values of two distinct feedback vertex sets $\mathcal{F}$ and $\mathcal{G}$ in this state. The values a together with the values f uniquely determine the values of all the variables in the circuit graph, and the result is circuit state $X{\cdot}s = a{\cdot}b$. Similarly, the values a together with the values g uniquely determine the rest of the variables, and this also results in state $X{\cdot}s = a{\cdot}b$. Hence, if a circuit state is found using the feedback vertex set $\mathcal{F}$, then it is also found using $\mathcal{G}$. □

We illustrate the computation of the stable states of the network of Figure 6.20 using the feedback vertex set $\{z_4\}$. The excitation function for the single feedback variable is $Z_4 = \overline{(X_2 + \overline{(X_1 + z_4)})}$. We find the stable states $(X{\cdot}z_4)$ of the feedback-state network to be 00·0, 00·1, 01·0, 10·1, and 11·0. The stable states of the input-, gate-, and wire-state network can now be computed using the circuit equations. These states are

$$00{\cdot}00\,01\,0001,\quad 00{\cdot}00\,10\,0010,\quad 01{\cdot}01\,10\,0110,\quad 10{\cdot}10\,01\,1001,$$

and

$$11{\cdot}11\,00\,1100.$$

6.5 GMW Analysis and Network Models

In this section we illustrate how the results of a GMW analysis differ as we vary the network model. Although feedback variables suffice for stable state calculations, this is not the case when we want to compute *transitions* among states. Our next example shows that transitions among states *do* depend on the choice of the feedback-vertex set.

Feedback-state network with feedback-vertex set $\{y_1\}$

Consider the feedback-state network of the NOR latch of Figure 6.1 with feedback-vertex set $\{y_1\}$. We have the excitation function

$$Y_1 = \overline{(X_1 + \overline{(X_2 + y_1)})} = \overline{X_1}(X_2 + y_1)$$

and the circuit equation

$$y_2 = \overline{(X_2 + y_1)}.$$

The state $X{\cdot}y_1 = 11{\cdot}0$ is stable. If the input excitation changes to 00, the feedback-state network moves to state $00{\cdot}0$. The excitation Y_1 in this state evaluates to 0, showing that gate 1 remains stable. The value of gate 2 changes to 1 as a result of this input change. Altogether, the model with feedback-vertex set $\{y_1\}$ predicts that the gate state of the NOR latch will be 01.

Feedback-state network with feedback-vertex set $\{y_2\}$

Now repeat the same analysis using feedback-vertex set $\{y_2\}$. It is clear from the symmetry of the circuit that the predicted final gate state in this case is 10!

The fact that the feedback-state model does not always predict the correct transitions has been known for a long time [66, 67]. Attempts were made to correct these deficiencies by developing a theory of "hazards." We shall return to hazard phenomena in Chapter 7.

The examples above strongly suggest that a "more accurate" model is needed. Consequently, we leave the feedback-state model now and consider other alternatives. In Section 7.4 we will show that the feedback-state model *can* be used to give correct results, but with a different race model.

Gate-state network

In 1955 Muller introduced a model in which a delay was associated with each gate [100, 106, 107]. Let us analyze the NOR latch using this model. The excitation functions are $Y_1 = \overline{(X_1 + y_2)}$ and $Y_2 = \overline{(X_2 + y_1)}$. The transition that leads to difficulties in the feedback-state model is from state $11{\cdot}00$ when the input changes to 00. If we

use $\{y_1\}$ as the feedback vertex set, we get 00·01 as the final state, whereas $\{y_2\}$ predicts 00·10. In the gate-state model, in the total state 00·00, both gates are unstable. This is the race that we have analyzed in Figure 6.10. The network may go to stable state 00·01, or to stable state 00·10, or it may enter the match-dependent oscillation between 00·00 and 00·11. In any case, the outcome predicted by this gate-state network is much different than those predicted by the feedback-state networks.

Gate- and wire-state network

The next example shows that gate delays are not sufficient. In the gate-state network corresponding to the circuit of Figure 6.21, the states 10·00 and 01·00 are both stable. Hence, the gate-state network predicts no changes in gate outputs when the input changes from 10 to 01. However, suppose that we add the wire vertex z to our network model. Now the excitation functions are

$$Y_1 = z{*}X_2,\ Y_2 = y_1 + y_2,\ Z = X_1.$$

The following sequence of states is possible in the GMW analysis, if the wire delay is appreciably large. Let the state of the new network be $X_1X_2{\cdot}y_1y_2\,z$. First, $10{\cdot}00\,1 \rightarrow 01{\cdot}00\,1$ as a result of the input change; then the network may respond with

$$01{\cdot}00\,1 \rightarrow 01{\cdot}10\,1 \rightarrow 01{\cdot}11\,1 \rightarrow 01{\cdot}11\,0 \rightarrow 01{\cdot}01\,0.$$

Thus, the presence of the wire delay makes two outcomes possible: 01·00 0 if the wire delay is negligible and 01·01 0 if the sequence above is followed. Therefore the gate-state network also appears to be inadequate.

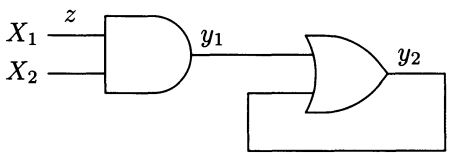


FIGURE 6.21. Gate circuit $C_{6.21}$.

At this point we can conclude that the gate- and wire-state network is more accurate than either some feedback-state networks or the gate-state network. We are not in a position to answer the question whether gate and wire delays suffice. Indeed, this turns out to be the case, but the proof of this result will come in Chapter 7.

6.6 The Extended GMW Model

A generalization of the GMW model to a three-valued multiple-winner model is now presented. Like the GMW model, this model is intuitively easy to describe, but very inefficient. We show later that a summary of the model's results can be found efficiently.

In the *extended multiple-winner* (*XMW*) model we are about to define, state variables are allowed to go through the intermediate value Φ when changing from one binary value to the other. Let

$$\mathbf{N} = \langle \{0, \Phi, 1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, F \rangle$$

be a network and let $\mathbf{a}\cdot\mathbf{b}$ be any ternary state of $\mathbf{N}$. The set $\mathcal{U}(\mathbf{a}\cdot\mathbf{b})$ of unstable delay vertices is defined as before. Similarly, we define a binary relation $\mathbf{R_a}$ on $\{0, \Phi, 1\}^m$ as the smallest relation satisfying

- $\mathbf{b}\mathbf{R_a}\mathbf{b}$, if $\mathbf{a}\cdot\mathbf{b}$ is stable,
- $\mathbf{b}\mathbf{R_a}\mathbf{b}'$, if $\mathbf{b} \neq \mathbf{b}'$ and $\mathbf{b}'_i \in \{\mathbf{b}_i, \mathbf{S}_i(\mathbf{a}\cdot\mathbf{b}), lub\{\mathbf{b}_i, \mathbf{S}_i(\mathbf{a}\cdot\mathbf{b})\}\}$, $1 \leq i \leq m$.

Thus any unstable state variable may keep the old value, become equal to its excitation, or become Φ.

The definition of the XMW relation is illustrated in Figure 6.22 for the gate-state network of the NOR latch, when started in state 10·10. The subscript on an unstable variable gives its excitation.

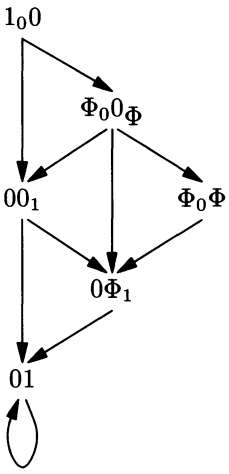


FIGURE 6.22. Example of XMW analysis.

Next, we need to define the outcome of a transition. The concept of transient cycle is somewhat more complicated here. A cycle is called *transient* in the XMW model when there is a vertex i that is unstable in all the

states in the cycle, has the same value in all these states, and that value is either binary or the excitation of that vertex is the same in all the states of the cycle. Thus we can have the following two cases. The variable may have the value Φ and a constant binary excitation, or it may have a binary value b and either $\bar{b}$ or Φ as excitation. The assumption is that a binary signal being "pulled" alternately to its complement and to Φ cannot persist indefinitely. However, a Φ-value pulled alternately to 0 and 1 can remain Φ indefinitely.

With this modification, one can define the sets *trans, cycle,* and *out* as in the case of the GMW relation.

6.7 Single-Winner Race Models

One of the basic assumptions in the GMW model is that there can be multiple winners in a race. This assumption corresponds closely to the notion of "true concurrency" [42]. In the literature dealing with concurrent systems, it is common to replace true concurrency with an interleaved model of concurrency. In this model, when there is more than one unstable component, only one can change at a time. However, the order in which the components change cannot be predicted. We now define a race model, called the *general single winner* (*GSW*) model, that corresponds to such an interleaved model of concurrency. We then compare and contrast the results obtained by using the GSW model and the GMW model.

Given a network N we define the binary GSW relation U_a on the set $\{0,1\}^m$ of internal states of N for every input vector $a \in \{0,1\}^n$.

For any $b \in \{0,1\}^m$,
bU_ab, if $\mathcal{U}(a{\cdot}b) = \emptyset$, i.e., the total state $a{\cdot}b$ is stable,
$bU_ab^{\{k\}}$, for every $k \in \mathcal{U}(a{\cdot}b)$,

where by $b^{\{k\}}$ we mean b with the variable k complemented. No other pairs of states are related by U_a.

We define the outcome of a GSW analysis in the same way as for the GMW analysis, i.e.,

$$out(U_a(b)) = \{s \mid bU_a^*c \text{ and } cU_a^*s, \text{ where } c \in cycl\_nontrans(U_a(b))\},$$

where *cycl_nontrans* is defined exactly as in the GMW relation.

The following proposition characterizes the results obtained by a GMW and a GSW analysis.

Proposition 6.3 *Given a network N we have $out(U_a(b)) \subseteq out(R_a(b))$. Furthermore, there exist networks for which the inclusion is proper.*

Proof: The first claim follows trivially from the definition of $out(U_a(b))$ and $out(R_a(b))$. For the second claim, consider the gate-state network for

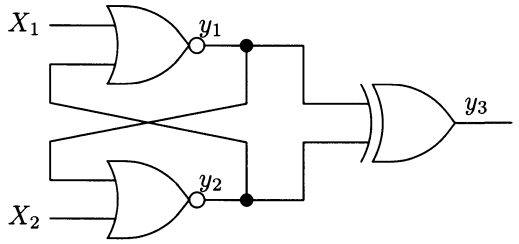


FIGURE 6.23. Circuit for which $out(U_a(b)) \subset out(R_a(b))$.

the circuit in Figure 6.23 started in the stable state 11·000 when the inputs change to 00. In Figure 6.24 we show the $R_{00}(000)$ and the $U_{00}(000)$ relations. Note that $out(U_{00}(000)) = \{011, 101\}$, whereas $out(R_{00}(000)) = \{000, 110, 011, 101\}$. In particular, in the GMW relation it is possible for the XOR gate to remain 0 indefinitely, whereas in the GSW model it must change to 1 within time $2D$, where D is the largest gate delay. □

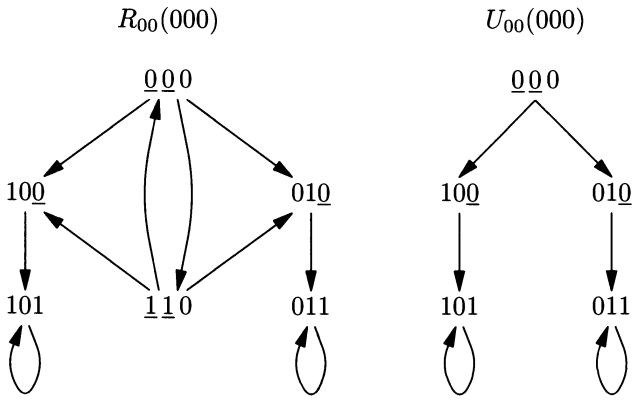


FIGURE 6.24. $R_{00}(000)$ and $U_{00}(000)$ relations.

We show in Chapter 7 that with a sufficient number of delays in the network model, a GSW analysis yields the same outcome as a GMW analysis.

6.8 Up-Bounded Ideal Delays

Recall that an ideal delay has the property that every input transition eventually appears at the output. Thus each delay must "remember" how many "unsatisfied" transitions have occurred. For this reason, the analysis of networks with ideal delays is somewhat more complex. This material is based partly on [56].

Let $N = \langle \{0,1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, F \rangle$ be a network. The ID-state of N is an ordered pair (b,u), where $b \in \{0,1\}^m$ is the state of N and u is a vector of m nonnegative integers. Each such integer u_i is called the *count* and gives the number of unsatisfied transitions for variable s_i, as is explained below. The count u_i is 0 if and only if the variable s_i is stable. We now define a binary relation on the set of ID-states of a network with ideal delays.

The *ideal multiple-winner relation* (*IMW* relation) I_a is defined as follows. For any $q = (b,u)$, we have qI_aq if and only if u is the all-zero vector. Otherwise, qI_aq' if $q' = (b',u')$ satisfies the conditions below. Select any number of state variables in b that have positive counts and change them to obtain b'. Each count u_i' is then calculated as follows:

$$u_i' = u_i - 1 \text{ if } b_i' \neq b_i \text{ and } S_i(a{\cdot}b) = S_i(a{\cdot}b'),$$

$$u_i' = u_i, \text{ if } b_i' = b_i \text{ and } S_i(a{\cdot}b) = S_i(a{\cdot}b'),$$

$$u_i' = u_i, \text{ if } b_i' \neq b_i \text{ and } S_i(a{\cdot}b) \neq S_i(a{\cdot}b'),$$

$$u_i' = u_i + 1 \text{ if } b_i' = b_i \text{ and } S_i(a{\cdot}b) \neq S_i(a{\cdot}b').$$

The interpretation of these rules is quite straightforward. When a variable changes, one of the changes that has previously appeared at the input of its delay appears now at the output of the delay. Thus one of the previously unsatisfied transitions has become satisfied; consequently, the count is reduced by 1. However, if the excitation of the variable changes during the transition, then a new transition has occurred at the input of the delay and the count must be increased by 1. Note that it is possible for a variable and its excitation to change at the same time. In that case, the count remains unchanged.

We illustrate the relation I_a with the gate-state network of Figure 6.3. Suppose the network starts in stable total state 11·00. The corresponding ID-state is $(00,00)$. Assume that the input changes to 00; thus we are interested in the relation I_{00}. After the input changes, both gates become

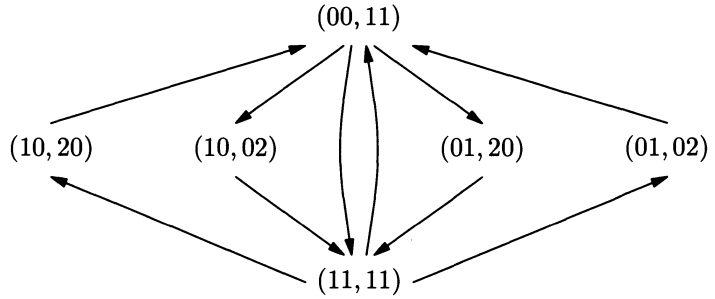


FIGURE 6.25. Analysis of latch with ideal delays.

unstable and the ID-state becomes $(00, 11)$. This is our initial ID-state for this transition. The graph of the I_a relation is shown in Figure 6.25. Notice that there are no stable states in this graph. Note also that states 10 and 01, which are stable in GMW analysis, are unstable here; in fact each state corresponds to two ID-states with different counts.

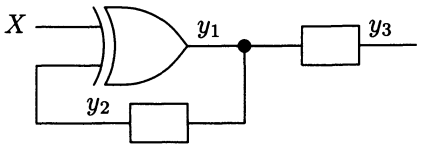


FIGURE 6.26. Circuit with infinite transition graph.

Our second example illustrates that the count variables may be unbounded and the I_a graph may be infinite. Consider the circuit of Figure 6.26. The network with state variables y_1, y_2, y_3 satisfies the excitation equations $Y_1 = X \oplus y_2$, $Y_2 = y_1$, and $Y_3 = y_1$. State 0·000 is stable. Changing the input to 1 we obtain the ID-state $(000, 100)$. One verifies that the following sequence is part of the I_a relation graph:

$$(000, 100) \rightarrow (100, 011) \rightarrow (110, 101) \rightarrow (010, 012) \rightarrow (000, 102).$$

Thus we have returned the first two variables to their initial state, and the third variable has not changed but has acquired two unsatisfied transitions. We can repeat the cycle on the first two variables without changing the value of the third. This would lead to the state $(000, 004)$. Clearly, this can be done arbitrarily many times, resulting in an infinite relation graph. Note that, although the delay of y_3 is up-bounded, the delays of y_1 and y_2 can be arbitrarily smaller than that of y_3.

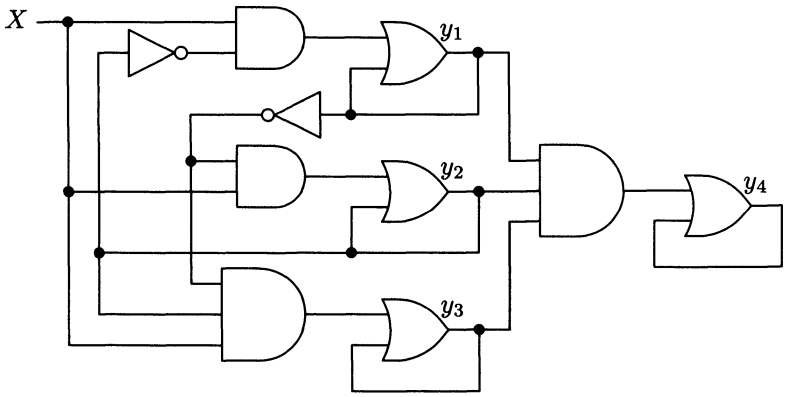


FIGURE 6.27. Circuit in which IMW differs from GMW.

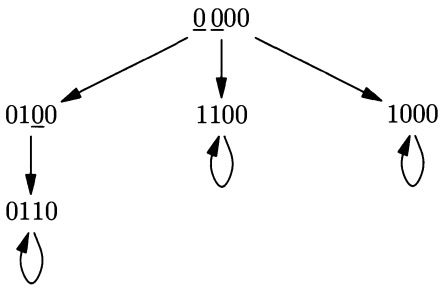


FIGURE 6.28. Graph of R_1 for the circuit of Figure 6.27.

Our third example shows that it is possible to reach certain states in I_a that are not possible in R_a. Consider the circuit of Figure 6.27. Suppose we use the feedback-state network with state variables y_1, y_2, y_3, y_4. The excitation equations are

$$Y_1 = y_1 + X*\overline{y_2}, \qquad Y_2 = y_2 + X*\overline{y_1},$$
$$Y_3 = y_3 + X*\overline{y_1}*y_2, \quad Y_4 = y_4 + y_1*y_2*y_3.$$

Starting in the stable state 0·0000 and changing the input to 1, we find the relation graph of $R_1(0000)$ as shown in Figure 6.28. Notice that the variable y_4 has the value 0 under the inertial delay model.

In contrast to the above, Figure 6.29 shows a path in the I_1 graph for the same network. Here it is possible to reach the state 0111, which is not in the outcome of the GMW analysis.

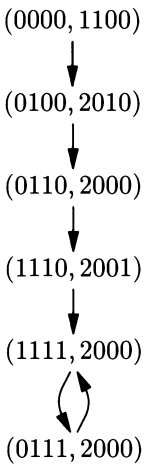


FIGURE 6.29. Part of graph of I_1 for the circuit of Figure 6.27.

6.9 Proofs

The proofs that were omitted in the main body of the chapter are now provided along with some auxiliary results.

6.9.1 Proofs for Section 6.2

The following observation is required in later proofs. Recall that an R_a-sequence is nontransient if there is no variable that is unstable and has the same value from some point on in the sequence.

Proposition 6.4 *An R_a-sequence $\gamma = (s^0, s^1, \ldots)$ is nontransient if and only if, for every state variable s_j and for every integer $r \geq 0$ such that $s_j^r = \overline{\alpha} \in \{0,1\}$ and $S_j(a{\cdot}s^r) = \alpha$, there exists an integer $p > 0$ such that $s_j^{r+p} = \alpha$ or $S_j(a{\cdot}s^{r+p}) = \overline{\alpha}$.*

This proposition implies that, in a nontransient sequence, the "runs" during which a variable has a constant value and is unstable are all finite.

We now show that for each UIN_a-history there is a corresponding nontransient R_a-sequence. Given a UIN_a-history $\mu = \langle \Theta, X(t), s(t) \rangle$, recall that $\gamma(\mu) = (s^0, s^1, \ldots)$, where $s^i = s(t_i)$ for $i \geq 0$.

Lemma 6.1 *Let μ be a UIN_a-history. Then $\gamma(\mu) = (s^0, s^1, \ldots)$ is a nontransient R_a-sequence.*

Proof: First we note that $s(t_{i-1}) \neq s(t_i)$ for all $i > 0$, by definition of μ. Thus $s^{i-1} \neq s^i$. Let $\mathcal{K}$ be the set of all the state variables that change in going from s^{i-1} to s^i. To show that $(s^0, s^1, \ldots)$ is an R_a-sequence, we need to verify that each of the variables in $\mathcal{K}$ is unstable in the total state $a{\cdot}s^{i-1}$. Suppose, on the contrary, that $s_j \in \mathcal{K}$ but $S_j(a{\cdot}s^{i-1}) = s_j^{i-1} = \alpha$. Then $S_j(a{\cdot}s(t)) = s_j(t) = \alpha$, for $t_{i-1} \leq t < t_i$. This contradicts Property 1, which requires the existence of a δ such that $t_i - \delta \leq t < t_i$ with $S_j(t) = \overline{\alpha}$. One verifies also that, if the R_a-sequence constructed above is finite, its last state must be stable, because the last state in the UIN_a-history is stable. Therefore $(s^0, s^1, \ldots)$ is an R_a-sequence. Suppose the sequence $(s^0, s^1, \ldots)$ is transient, i.e., there exists a state variable s_j and an integer $r \geq 0$ such that $s_j^i = \overline{\alpha}$ and $S_j(a{\cdot}s^i) = \alpha$ for $i \geq r$. By the definition of the sequence $(s^0, s^1, \ldots)$ this implies that $s_j(t) = \overline{\alpha}$ and $S_j(a{\cdot}s(t)) = \alpha$ for $t \geq t_i$, contradicting Property 2. Thus the R_a-sequence is nontransient. □

Next, we want to show that, for every nontransient R_a-sequence, there exists a corresponding UIN_a-history. Before we can prove this result, we must introduce a number of concepts. Consider a nontransient R_a-sequence $\gamma = (s^0, s^1, \ldots)$. A *run* from state s^i of γ is a sequence $(s^i, s^{i+1}, \ldots, \ldots s^{p-1})$ of consecutive states of γ with the following property: There exists a j,

$1 \leq j \leq m$ such that

$$s_j^i = s_j^{i+1} = \ldots = s_j^{p-1} = \overline{\alpha},$$
$$S_j(a{\cdot}s^i) = S_j(a{\cdot}s^{i+1}) = \ldots = S_j(a{\cdot}s^{p-1}) = \alpha,$$

but $s_j^p = \alpha$ or $S_j(a{\cdot}s^p) = \overline{\alpha}$. The length of a run is the number of states in it. Since we are assuming that γ is nontransient, the length of the longest run from any state is finite. The longest run from s^i is called a *maximal run* of γ. Let $P(s^i)$ denote the length of the longest run from state s^i. Note that a stable state has no runs.

We are interested in the lengths of the maximal runs for the following reason. To construct a UIN_a-history corresponding to an R_a-sequence, we must assign a time t_i to each state s^i in the R_a-sequence. The difference between the times assigned to states $s^{P(s^i)}$ and s^i in the sequence must not exceed the upper bound D_j for any unstable variable s_j defining the run; otherwise, we would be violating Property 2 of UIN delays. At the same time, when assigning times to the states in the R_a-sequence to form the corresponding UIN_a-history, we must be careful not to assign an infinite number of states to a finite interval. The following construction presents one method of avoiding this problem.

We view γ as consisting of a sequence of certain maximal runs that will be called *segments*. The first segment is $s^0, s^1, \ldots, s^{P(s^0)-1}$, where $P(s^0)$ is the length of the longest run from s^0. Let D be the minimum value of all the network delays. State s^0 will be assigned time 0 and state $s^{P(s^0)}$ will be assigned time $D/2$. The next segment is $s^{P(s^0)}, s^{P(s^0)+1}, \ldots, s^{P(s^{P(s^0)}-1)}$. In other words, we first move to the first state after the end of the first segment and reach the state $s^{P(s^0)}$. We then find the maximal run from that state, and advance further by that run. The state $s^{P(s^{P(s^0)})}$ will be assigned time $2D/2$. We then continue in the same fashion.

In general, each segment will be assigned exactly $D/2$ time units. The intermediate states (if any) reached during a segment will be assigned times in such a way that they are equally spaced between the initial states of consecutive segments. By this construction, and the fact that each run is of finite length, we are ensuring that only a finite number of changes occurs in any finite interval. Furthermore, since the time for each segment is $D/2$ and any run can involve at most two segments, the length of the longest possible run of γ is strictly less than the time assigned to two segments, i.e., D. Thus, the length of any run will be strictly less than D, as required by the maximum delay assumption.

To illustrate this construction, consider the nontransient R_a-sequence $\gamma = (s^0, s^1, \ldots, s^9) =$

$$(\underline{0}\,1\,\underline{1}, \underline{0}\,1\,\underline{0}, \underline{0}\,1\,\underline{1}, 1\,\underline{1}\,\underline{1}, 1\,\underline{1}\,\underline{0}, 1\,\underline{1}\,\underline{1}, 1\,\underline{1}\,\underline{0}, 1\,0\,\underline{0}, 1\,0\,1).$$

(This sequence is derived from Figure 6.12.) One verifies that s^0 has a maximal run of length 3, s^1 of length 2, s^2 of length 2, s^3 of length 4, etc.

We divide γ into segments as follows:

$$(0\underline{1}\,1,0\underline{1}\,0,0\underline{1}\,1)\ (1\underline{1}\,1,1\underline{1}\,0,1\underline{1}\,1,1\underline{1}\,0)\ (1\,0\underline{0})\ (1\,0\,1),$$

where the final state 101, corresponds to a special final segment that has no runs. Suppose the least upper bound of the three delays is $D = 6$ ns. Then the time sequence Θ (divided into segments to improve readability) assigned by this construction is

$$(0, 1, 2)\ (3, 3.75, 4.5, 5.25)\ (6)\ (9).$$

Formally, suppose that $\gamma = (s^0, s^1, \ldots)$ is a nontransient R_a-sequence. The UIN_a-history $\mu(\gamma) = \langle \Theta, X(t), s(t) \rangle$ is constructed in two steps. First we compute the index of the state that starts segment j. We denote this index by σ_j and define it recursively as

$$\sigma_j = \begin{cases} 0 & \text{if } j = 0, \\ \sigma_{j-1} + P(s^{\sigma_{j-1}}) & \text{otherwise.} \end{cases}$$

Θ will be chosen in such a way that interval j will begin at time $jD/2$. This ensures that the time associated with state s^{σ_j} is $jD/2$. Thus define Θ by

$$t_i = \begin{cases} \frac{jD}{2} + \frac{(i-\sigma_j)D}{2(\sigma_{j+1}-\sigma_j)} & \text{if } \sigma_j \le i < \sigma_{j+1}, \\ \\ \frac{jD}{2} & \text{if } \sigma_j = i = \sigma_{j+1}. \end{cases}$$

Let $X(t) = a$ for $t \ge t_0$. For all $i \ge 0$, let $s(t) = s^i$ for $t_i \le t < t_{i+1}$, if s^i is unstable; let $s(t) = s^i$ for $t_i \le t$, if s^i is stable.

Proposition 6.5 *Suppose the length of the longest run from s^i is p. Then $t_{i+p} - t_i < D$.*

Proof: We prove the claim by contradiction. Assume $t_{i+p} - t_i \ge D$. Clearly, s^i cannot be the first state of any segment, because the first state after s_i's longest run is $D/2 < D$ time units away from s^i. There must be a segment with a nonzero run that starts after s^i; if the following segment were the final stable state, then $t_{i+p} - t_i$ could not exceed $D/2$. This implies that j satisfies the conditions

$$t_i < \frac{jD}{2} \quad \text{and} \quad \frac{(j+1)D}{2} \le t_{i+p}.$$

Now we have a run from time t_i to time t_{i+p} and this run properly includes the segment between σ_j and σ_{j+1}. This is a contradiction that the run between σ_j and σ_{j+1} is of maximal length. □

Lemma 6.2 *Let γ be a nontransient R_a-sequence. Then $\mu(\gamma)$ is a UIN_a-history.*

Proof: We need to establish that $\mu(\gamma)$ satisfies Conditions I and II of the definition of UIN_a-history in Section 6.2. Conditions I(a), (b) and (e) follow immediately from the construction of $\mu(\gamma)$. Conditions I(c) and (d) follow from the fact that γ is an R_a-sequence. Now, let s^j be any state variable in N. To prove the lemma we need to verify Condition II, i.e., that the waveform $S_j(X(t), s(t))/s_j(t)$ in $\mu(\gamma)$ satisfies Properties 1 and 2 for the input/output behavior of a UIN delay.

First we verify Property 1. Suppose $s_j(t)$ changes from $\overline{\alpha}$ to α at time τ. From the definition of $s(t)$, this means that there exists $k > 0$ such that $t_k = \tau$ and $s_j^{k-1} = \overline{\alpha}$ and $s_j^k = s_j(\tau) = \alpha$. Since $(s^0, s^1, \ldots)$ is an R_a-sequence, $s^{k-1} R_a s^k$. Hence the variable s_j must be unstable in total state $a{\cdot}s^{k-1}$, i.e., $S_j(a{\cdot}s^{k-1}) = \alpha$. By definition of $s(t)$, we have $s^{k-1} = s(t)$, for $t_{k-1} \leq t < t_k$. Hence $S_j(a{\cdot}s(t)) = \alpha$ for $\tau - (t_k - t_{k-1}) \leq t < \tau$, and Property 1 holds for each variable s_j.

For Property 2, assume that $S_j(X(t), s(t)) = \alpha$ for $\tau \leq t < \tau + D$, but that $s_j(t) = \overline{\alpha}$ in the same interval. Suppose $t_k \leq \tau < t_{k+1}$, and $t_l \leq \tau + D < t_{l+1}$; then the same conditions also hold for the interval $t_k \leq t < t_{l+1}$. But this implies the existence of a run that lasts for time $t_{l+1} - t_k$ that is greater than or equal to D. It follows from our construction and from Proposition 6.5 that such a run cannot exist. Hence, we conclude that $S_j(X(t), s(t))/s_j(t)$ satisfies Property 2 for all variables s_j. □

The main theorem of Section 6.2 that relates the GMW model to up-bounded inertial delays now follows from the two lemmas above.

6.9.2 Proofs for Section 6.3

Lemma 6.3 *If state d is in $out(R_a(b))$, then it is not D-transient.*

Proof: Suppose $d \in out(R_a(b))$; then there exists a nontransient state c such that bR_a^*c and cR_a^*d. Then we can find sequences of the form

$$s^0, \ldots, s^i, (s^{i+1}, \ldots, s^j)^m, s^{i+1}, \ldots, s^k,$$

where $s^0 = b$, $s^{i+1} = c$, $s^k = d$, $(s^{i+1}, \ldots, s^j)$ is a nontransient cycle, and m is an arbitrarily large integer. Corresponding to each such sequence, we can construct a UIN_a-history $\mu = \langle \Theta, X(t), s(t) \rangle$. In this history, let $\delta > 0$ be the time required to go once around the cycle involving state c. Assume that d is D-transient with τ as the limit. Then we can choose m large enough to satisfy $m\delta > \tau$. But this means that state d can be reached in μ by a sequence of length greater than τ; this is a contradiction. □

Lemma 6.4 *If $d \notin out(R_a(b))$, then d is D-transient.*

Proof: If d is not in the outcome, then it can be reached by a sequence that either contains no cycles or contains only transient cycles.

Consider first the case of a sequence $\gamma = s^0, \ldots, s^k$, where $s^i R_a s^{i+1}$ for $i = 0, \ldots, k-1$, $s^0 = b$, $s^k = d$, and all the states are distinct. Since at least one of the 2^m states of N must be in the outcome, and $s^k = d$ is not in the outcome, it follows that k must be strictly less than $2^m - 1$. Consider now the prefix μ, corresponding to γ, of any UIN_a-history satisfying the bound D. Each interval $t_{i+1} - t_i$ must be less than D, by Property 2. Hence the time to reach state d is less that $(2^m - 2)D$.

It may be possible to increase the length of a sequence from b to d by inserting transient cycles. So suppose that we have a sequence

$$s^0, \ldots, s^i, (s^{i+1}, \ldots, s^j), s^{i+1}, s^l, \ldots, s^k,$$

where $s^0 = b$, $s^{i+1} = c$, $s^k = d$ and the transient cycle $(s^{i+1}, \ldots, s^j)$ has been inserted. The transient cycle is entered when the transition from s^i to the first s^{i+1} is made. The cycle is left when the transition from the second s^{i+1} to s^l is made. The sequence $(s^{i+1}, \ldots, s^j), s^{i+1}$ is a run with an unstable variable having a constant value. This run must last less than D units of time. But this means that the time to go from the first s^{i+1} to s^l must be less that D. Therefore the total time for going from b to d is still less that $(2^m - 2)D$, even if a transient cycle is inserted. Since any sequence from b to d can be viewed as a sequence of distinct states into which transient cycles have been inserted, it follows that the time to reach d is always less that $(2^m - 2)D$. Thus, if d is not in the outcome, then it is D-transient with limit $(2^m - 2)D$. □

From these two lemmas, Proposition 6.1 and Theorem 6.2 now follow.

Chapter 7

Ternary Simulation

As we have seen in Chapter 6, the race analysis methods—that examine all possible successors of a given state—are computationally intractable. We now describe a method that is quite efficient and produces results that can be viewed as a summary of the results produced by the multiple-winner methods. For many purposes, such summarized results are sufficient.

The idea of ternary simulation is relatively straightforward, as we illustrate in Section 7.1 by several examples. Next, the two parts of the ternary algorithm are defined and their results are characterized in Sections 7.2 and 7.3. The result of Algorithm A can be thought of as providing a summary for the set of all the states reachable from the initial state in the GMW analysis of an input-, gate-, and wire-state network. Similarly, Algorithm B is shown to provide a summary of the outcome of the same GMW analysis. In Section 7.4 we show that every input- and feedback-state model gives correct results for transitions among states, provided it is analyzed with the aid of ternary simulation. We next show in Section 7.5 that ternary simulation not only gives the correct outcome, but is also capable of detecting static hazards in the network outputs. In Sections 7.6 and 7.7 we relate ternary simulation to the GSW and the XMW analysis methods, respectively. The latter result shows that ternary simulation is also applicable to CMOS networks. We close the chapter with Section 7.8, which contains the proofs of the more difficult results.

How To Read This Chapter

Some of the proofs of the results in this chapter are quite complex. We have included such proofs in Section 7.8. That section can be omitted on first reading. While the proofs remaining in the main body of the chapter are relatively simple, the reader may wish to get an overview of the chapter by omitting them on first reading.

7.1 Introductory Examples

In 1965 Eichelberger developed a method [49] for the detection of hazards and races. The method uses three values for logic signals: $0, 1$, and Φ.

The precise mathematical meaning of Φ was given in Section 2.3, where ternary algebra was discussed; intuitively, we interpret Φ as a changing, uncertain signal. Eichelberger's method consists of two parts, which we call Algorithms A and B, and assumes that the circuit is started in a stable state and some inputs are changed. In this chapter we present a more general method that does not require the initial circuit state to be stable. This material is closely based on [126].

Assume we are given a network **N** started in some total state $a{\cdot}b$. In Algorithm A, *every state variable is repeatedly set to the least upper bound (lub) of its current value and its excitation.* Thus, in the first step, all the unstable state variables are set to Φ. Intuitively, this indicates that these variables are uncertain. The uncertainty so introduced is then spread throughout the network. In Algorithm B, *every state variable is repeatedly set to its excitation.* Consequently, some of the uncertainties are removed. If they are all removed, then the outcome is a stable binary state. If some uncertain state variables remain, the outcome for these state variables may be either 0 or 1, depending on the relative sizes of the delays. Together, Algorithms A and B constitute *ternary simulation.* We illustrate these ideas with three informal examples.

Example 1

Consider the circuit of Figure 7.1. Suppose it was in stable initial total state $X{\cdot}xy_1y_2y_3 = 0{\cdot}0011$, and the input just changed to 1. Note that we assume there is a delay associated with the input. We now run the algorithm starting the network in total state $1{\cdot}0011$. First, the input delay has an excitation that differs from its current value; thus it changes to Φ, i.e., the state becomes $1{\cdot}\Phi011$. Now the output of the OR gate becomes Φ, since one of its inputs is 0 and the other is uncertain. Similarly, the NAND gate's output becomes uncertain, and the circuit reaches ternary state $1{\cdot}\Phi\Phi1\Phi$. In this state, the inverter input is uncertain, and state $1{\cdot}\Phi\Phi\Phi\Phi$ is reached. This state is stable, and Algorithm A terminates here. These steps are shown in Figure 7.2.

For Algorithm B, the circuit starts in the state reached in Algorithm A, i.e., in $1{\cdot}\Phi\Phi\Phi\Phi$. In the first step, the input delay changes and the

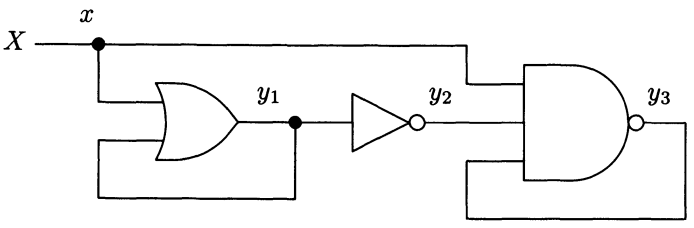


FIGURE 7.1. Circuit for Example 1.

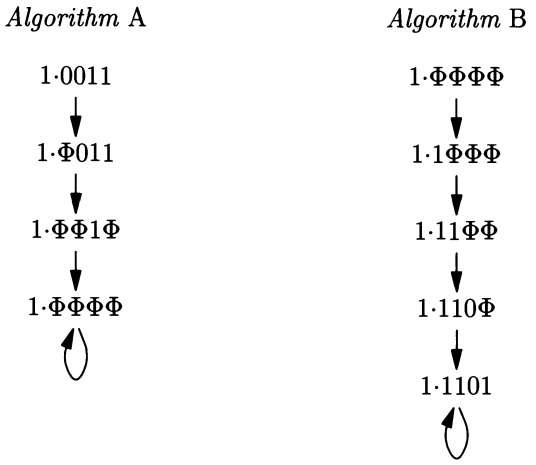


FIGURE 7.2. Ternary simulation for Example 1.

circuit moves to 1·1ΦΦΦ. Now the OR gate responds to the input delay change, and state 1·11ΦΦ is reached. As a result of this, the inverter, and then the NAND gate, also change as shown in Figure 7.2. Thus the result of the ternary simulation is the binary state 1·1101.

The transition we have analyzed above corresponds to the GMW analysis shown in Figure 7.3, where we used the input- and gate-state model. Here the two methods of analysis give the same result, because the outcome of the GMW analysis is also the state 1·1101.

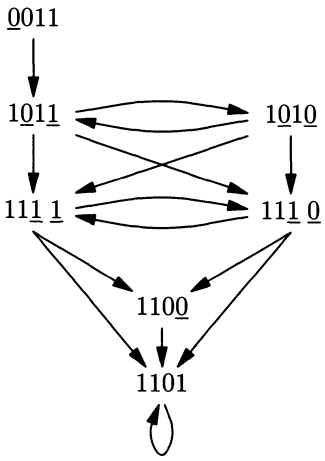


FIGURE 7.3. GMW analysis for Example 1.

Example 2

For our second example, consider the circuit of Figure 7.4, first without the wire delay, i.e., with excitation functions $Y_1 = x \oplus x$, and $Y_2 = y_1 + y_2$. If the circuit is in stable total state $X{\cdot}xy_1y_2 = 0{\cdot}000$ and the input changes to 1, we can carry out the analysis starting in state $1{\cdot}000$. If we do so, we find

Algorithm A:

$$1{\cdot}000 \longrightarrow 1{\cdot}\Phi 00 \longrightarrow 1{\cdot}\Phi\Phi 0 \longrightarrow 1{\cdot}\Phi\Phi\Phi.$$

Algorithm B:

$$1{\cdot}\Phi\Phi\Phi \longrightarrow 1{\cdot}1\Phi\Phi \longrightarrow 1{\cdot}10\Phi.$$

Here, the ternary simulation result disagrees with the GMW analysis in the input- and gate-state model; the latter predicts the stable state $1{\cdot}100$ as the final outcome, as contrasted with $1{\cdot}10\Phi$ predicted above by Algorithm B. If the wire delay z is added, however, and the network's total state is $X{\cdot}xy_1y_2z$, we have

Algorithm A:

$$1{\cdot}0000 \longrightarrow 1{\cdot}\Phi 000 \longrightarrow 1{\cdot}\Phi\Phi 0\Phi \longrightarrow 1{\cdot}\Phi\Phi\Phi\Phi.$$

Algorithm B:

$$1{\cdot}\Phi\Phi\Phi\Phi \longrightarrow 1{\cdot}1\Phi\Phi\Phi \longrightarrow 1{\cdot}1\Phi\Phi 1 \longrightarrow 1{\cdot}10\Phi 1.$$

It is easy to see that a GMW analysis of this extended network predicts two stable states, $1{\cdot}1001$ and $1{\cdot}1011$, for this transition. If we accept the ternary state $1{\cdot}10\Phi 1$ as representing both of these binary states, then the results of ternary simulation agree with the GMW analysis.

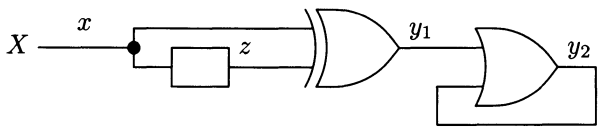


FIGURE 7.4. Circuit for Example 2.

Example 3

In our third example, the input delay to the circuit is stable, but an internal gate is unstable. It may appear unusual to start in a state in which the input has not changed, but there is some internal instability. In practice, however, such situations occur quite frequently. In particular, when power is applied to a circuit, it is usually impossible to know its starting state.

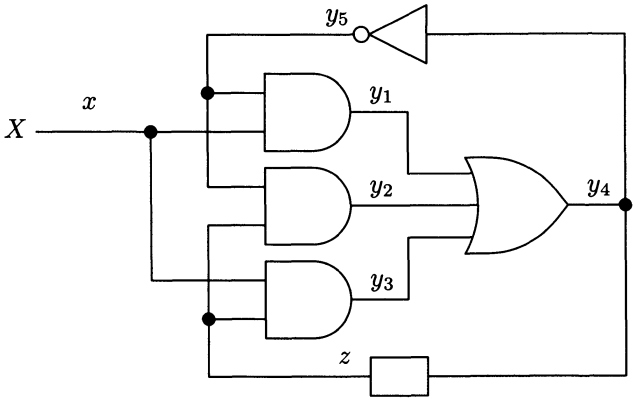


FIGURE 7.5. Circuit for Example 3.

Consider the circuit of Figure 7.5 started in unstable total state $X{\cdot}x\,y_1y_2y_3y_4y_5\,z = 0{\cdot}0\,0\,1\,0\,1\,0\,1\,0\,1$. Note that only gate y_2 is unstable in this state. The analysis yields

Algorithm A:

$$0{\cdot}0\,0\,1\,0\,1\,0\,1\,0 \longrightarrow 0{\cdot}0\,0\,\Phi\,0\,1\,0\,1 \longrightarrow 0{\cdot}0\,0\,\Phi\,0\,\Phi\,0\,1 \longrightarrow 0{\cdot}0\,0\,\Phi\,0\,\Phi\,\Phi\,\Phi.$$

Algorithm B:

$$0{\cdot}0\,0\,\Phi\,0\,\Phi\,\Phi\,\Phi.$$

Suppose we now perform a GMW analysis; the result shows that the circuit is not guaranteed to reach a unique stable state. For example, the nontransient cycle

$$0{\cdot}0\,0\underline{1}\,0\,1\,0\,1 \longrightarrow 0{\cdot}0\,0\,0\,0\underline{1}\,0\,1 \longrightarrow 0{\cdot}0\,0\,0\,0\,0\underline{0}\,\underline{1} \longrightarrow$$

$$0{\cdot}0\,0\underline{0}\,0\,0\,1\,\underline{1} \longrightarrow 0{\cdot}0\,0\,1\,0\underline{0}\,1\,\underline{1} \longrightarrow 0{\cdot}0\,0\,1\,0\,1\,\underline{1}\,1 \longrightarrow 0{\cdot}0\,0\underline{1}\,0\,1\,0\,1$$

is in the outcome. From this observation, and the fact that the wire z is unstable in three of the states in the nontransient oscillation (and can therefore become 0), one has

$$0{\cdot}0\,0\,\Phi\,0\,\Phi\,\Phi\,\Phi \sqsubseteq \mathit{lub}\ \mathit{out}(R_a(0{\cdot}0\,0\,1\,0\,1\,0\,1)).$$

For convenience, an input-, gate-, and wire-state network model of a circuit C is called the *complete network* of C. The main theorem of this chapter states that ternary simulation of a complete network N agrees with the results of the GMW analysis, i.e., ternary simulation computes the *lub* of the GMW outcome. We postpone the formal statement of this theorem until more background has been established.

7.2 Algorithm A

For ternary simulation we use the ternary domain in the network model. To distinguish two versions of the same network, one with a binary domain and the other with a ternary domain, we denote them by N and $\mathbf{N}$, respectively. Let $N = \langle \{0,1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, F \rangle$ be a binary network, and $\mathbf{N} = \langle \{0,\Phi,1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, \mathbf{F} \rangle$ its ternary counterpart, called the *ternary extension* of N. There are n inputs and m state variables in N and $\mathbf{N}$. We use the convention that state variable vectors in the ternary domain are denoted by $\mathbf{s}$ and the input and vertex excitation function vectors by $\mathbf{X}$ and $\mathbf{S}$. Let $a{\cdot}b$ be a (binary) total state of $\mathbf{N}$. We remind the reader that the ternary excitation function $\mathbf{S}_i$ associated with each state variable s_i satisfies the monotonicity property

$$\mathbf{a} \sqsubseteq \mathbf{b} \text{ implies } \mathbf{S}_i(\mathbf{a}) \sqsubseteq \mathbf{S}_i(\mathbf{b}),$$

for all $\mathbf{a}, \mathbf{b} \in \{0,\Phi,1\}^n$. This is true because $\mathbf{S}_i$ is always the ternary extension of a Boolean function, and the monotonicity property holds for ternary extensions (see Section 2.3).

The first algorithm of ternary simulation is formally defined as follows:

Algorithm A
$h := 0$;
$\mathbf{s}^0 := b$;
repeat
  $h := h + 1$;
  $\mathbf{s}^h := lub\{\mathbf{s}^{h-1}, \mathbf{S}(a{\cdot}\mathbf{s}^{h-1})\}$;
until $\mathbf{s}^h = \mathbf{s}^{h-1}$;

In the following, we use A (roman) to denote the name of the algorithm and A (italic) to denote the length of the sequence of states that the algorithm generates. Propositions 7.1 and 7.2 below are based on [26].

Proposition 7.1 *Algorithm* A *generates a finite sequence* $\mathbf{s}^0, \ldots, \mathbf{s}^A$ *of states, where* $A \leq m$. *Furthermore, this sequence is monotonically increasing, i.e.,*

$$\mathbf{s}^h \sqsubset \mathbf{s}^{h+1}, \text{ for } 0 \leq h < A.$$

Proof: First, by the fact that $\mathbf{t} \sqsubseteq lub\{\mathbf{t}, \mathbf{t}'\}$, for all $\mathbf{t}$ and $\mathbf{t}'$, it follows that

$$\mathbf{s}^h \sqsubseteq lub\{\mathbf{s}^h, \mathbf{S}(a{\cdot}\mathbf{s}^h)\} = \mathbf{s}^{h+1}, \text{ for } 0 \leq h < A.$$

Second, in each step of the algorithm, at least one state variable must become Φ; otherwise the algorithm terminates. Since there are m state variables, it now follows that A cannot exceed m. □

Let N be a network in state b with inputs held constant at a. Define the set of all states reachable from b in the GMW analysis as: $reach(R_a(b)) = \{c \mid bR_a^*c\}$. In the following, if $h > A$, by $\mathbf{s}^h$ we mean $\mathbf{s}^A$.

Proposition 7.2 *The least upper bound of the set of all the states reachable in the GMW analysis of a network N is covered by the result of Algorithm* A *for* **N**, *i.e.,*

$$lub\, reach(R_a(b)) \sqsubseteq \mathbf{s}^A.$$

Moreover,

$$b(R_a)^h c \text{ implies } c \sqsubseteq \mathbf{s}^h.$$

Proof: The proof of the second claim is by induction on h. For $h = 0$, we have $b(R_a)^0 c$ implies $c = b$. But also $\mathbf{s}^0 = b$. Hence $b(R_a)^0 c$ implies $c \sqsubseteq \mathbf{s}^0$. Assume now that $b(R_a)^h c$ implies $c \sqsubseteq \mathbf{s}^h$, and suppose that $cR_a d$. By definition of R_a, each component d_i of d has either the value of the corresponding component c_i in c or it is equal to the excitation $S_i(a{\cdot}c)$. Thus $d \sqsubseteq lub\{c, S(a{\cdot}c)\}$. The latter expression is equal to $lub\{c, \mathbf{S}(a{\cdot}c)\}$, since the ternary extension **S** agrees with S on binary arguments. Using the induction hypothesis, the monotonicity of **S**, and the monotonicity of *lub*, we find $d \sqsubseteq lub\{\mathbf{s}^h, \mathbf{S}(a{\cdot}\mathbf{s}^h)\} = \mathbf{s}^{h+1}$. Thus the second claim holds. By Proposition 7.1, $\mathbf{s}^A$ covers $\mathbf{s}^h$ for every h; hence the main claim is established. □

The formulation of Algorithm A above is very general, in the sense that it makes no assumptions about the starting state $a{\cdot}b$. However, if the network starts in a stable total state, then only input-delay vertices are unstable after an input change. In this case it is convenient to use a slightly simpler formulation of Algorithm A; we call this version Algorithm Ã. Assume **N** is started in the stable total state $\hat{a}{\cdot}b$ and the input changes to a. Algorithm Ã is defined as follows:

Algorithm Ã
$h := 0$;
$\mathbf{a} := lub\{\hat{a}, a\}$;
$\mathbf{s}^0 := b$;
repeat
  $h := h + 1$;
  $\mathbf{s}^h := \mathbf{S}(\mathbf{a}{\cdot}\mathbf{s}^{h-1})$;
until $\mathbf{s}^h = \mathbf{s}^{h-1}$;

The reader can verify that the monotonicity result of Proposition 7.1 holds also for Algorithm Ã. Thus we have

Proposition 7.3 *Algorithm* Ã *generates a finite sequence* $\tilde{\mathbf{s}}^0, \ldots, \tilde{\mathbf{s}}^{\tilde{A}}$ *of states, where* $\tilde{A} \leq m$. *Furthermore, this sequence is monotonically increasing, i.e.,*

$$\tilde{\mathbf{s}}^h \sqsubset \tilde{\mathbf{s}}^{h+1}, \text{ for } 0 \leq h < \tilde{A}.$$

The reader can also verify that Algorithm $\tilde{\text{A}}$ applied to the network of Figure 7.1 yields exactly the same results as those of Figure 7.2.

The next lemma shows that the simpler version of the first algorithm is sufficient when the network N has input delays and is in a stable total state initially. For convenience, we assume that the first n state variables are the n input-delay variables, i.e., that $s_j = x_j$ for $j = 1, \ldots, n$.

Lemma 7.1 *Let* $\mathbf{N}$ *be a ternary network with input delays started in the stable total (binary) state* $\hat{a}\cdot b$ *and let the input change to* a. *Let* $\tilde{\mathbf{s}}^{\tilde{A}}$ *denote the result of Algorithm* $\tilde{\text{A}}$. *Similarly, let* $\mathbf{s}^A$ *denote the result of Algorithm* A *when* $\mathbf{N}$ *is started in the total state* $a\cdot b$. *Then* $\tilde{\mathbf{s}}^{\tilde{A}} = \mathbf{s}^A$.

Proof: Let $\mathbf{s}^h$, $0 \leq h \leq A$ and $\tilde{\mathbf{s}}^h$, $0 \leq h \leq \tilde{A}$ be the results of Algorithms A and $\tilde{\text{A}}$, after h steps. We prove that $\mathbf{s}_j^h = \tilde{\mathbf{s}}_j^h$ for $h \geq 0$. The lemma then follows immediately by Propositions 7.1 and 7.3. Consider first an input-delay vertex j; its excitation is $\mathbf{X}_j$. Note that $b_j = \hat{a}_j$, since $\hat{a}\cdot b$ is a stable total state. There are two subcases to consider. If $\hat{a}_j = a_j$, then $\mathbf{a}_j = a_j$ and it follows trivially that $\mathbf{s}_j^h = b_j$ and $\tilde{\mathbf{s}}_j^h = b_j$ for $h \geq 0$. On the other hand, if $\hat{a}_j \neq a_j$, then

$$\mathbf{s}_j^h = \tilde{\mathbf{s}}_j^h = \begin{cases} b_j & \text{if } h = 0, \\ \Phi & \text{otherwise.} \end{cases}$$

Altogether, if j is an input-delay vertex, then $\mathbf{s}_j^h = \tilde{\mathbf{s}}_j^h$ for $h \geq 0$.

We now show by induction on h, that $\mathbf{s}^h = \tilde{\mathbf{s}}^h$ for $h \geq 0$. Since $\mathbf{s}^0 = \tilde{\mathbf{s}}^0 = b$, the basis follows trivially. Assume inductively that $\mathbf{s}^h = \tilde{\mathbf{s}}^h$ holds for some $h \geq 0$. If j is an input vertex, we already know from the above that $\mathbf{s}_j^{h+1} = \tilde{\mathbf{s}}_j^{h+1}$. If j is an internal vertex, then $\mathbf{S}_j(\mathbf{a}\cdot\mathbf{c}) = \mathbf{S}_j(\mathbf{a}'\cdot\mathbf{c})$ for every $\mathbf{a}, \mathbf{a}' \in \{0, \Phi, 1\}^n$ and $\mathbf{c} \in \{0, \Phi, 1\}^m$. In particular, $\mathbf{S}_j(a\cdot\mathbf{s}^h) = \mathbf{S}_j(\mathbf{a}\cdot\mathbf{s}^h)$. This, together with the induction hypothesis, implies that $\mathbf{S}_j(a\cdot\mathbf{s}^h) = \mathbf{S}_j(\mathbf{a}\cdot\tilde{\mathbf{s}}^h) = \tilde{\mathbf{s}}_j^{h+1}$. We now claim that $\mathbf{S}_j(a\cdot\mathbf{s}^h) = \mathit{lub}\{\mathbf{s}_j^h, \mathbf{S}_j(a\cdot\mathbf{s}^h)\}$. From the properties of *lub*, it follows that $\mathbf{S}_j(a\cdot\mathbf{s}^h) \sqsubseteq \mathit{lub}\{\mathbf{s}_j^h, \mathbf{S}_j(a\cdot\mathbf{s}^h)\}$. If $\mathbf{S}_j(a\cdot\mathbf{s}^h) = \Phi$ then, trivially, $\mathit{lub}\{\mathbf{s}_j^h, \mathbf{S}_j(a\cdot\mathbf{s}^h)\} \sqsubseteq \mathbf{S}_j(a\cdot\mathbf{s}^h)$.

This leaves the case where $\mathbf{S}_j(a\cdot\mathbf{s}^h) \in \{0, 1\}$. Recall that $\mathbf{S}_j$ is a monotonic function of its arguments. Recall also that the network is started in a stable total state $a\cdot b$ in which all the arguments of $\mathbf{S}_j$ are binary, and in which $\tilde{\mathbf{s}}_j = \mathbf{S}_j(a\cdot b) = b_j$. Note that each $\tilde{\mathbf{s}}_j$ can either remain at its initial binary value throughout the algorithm or it can change to Φ. The latter can only occur if $\mathbf{S}_j$ has changed to Φ in the previous step. In summary, if $\mathbf{S}_j(a\cdot\mathbf{s}^h) \in \{0, 1\}$, then $\mathbf{s}_j^h = \mathbf{S}_j(a\cdot\mathbf{s}^h)$, and we have proved our claim that $\mathbf{S}_j(a\cdot\mathbf{s}^h) = \mathit{lub}\{\mathbf{s}_j^h, \mathbf{S}_j(a\cdot\mathbf{s}^h)\}$. Using this claim and the fact that $\mathbf{S}_j(a\cdot\mathbf{s}^h) = \mathbf{S}_j(\mathbf{a}\cdot\tilde{\mathbf{s}}^h)$, we now have

$$\mathbf{s}_j^{h+1} = \mathit{lub}\{\mathbf{s}_j^h, \mathbf{S}_j(a\cdot\mathbf{s}^h)\} = \mathbf{S}_j(a\cdot\mathbf{s}^h) = \mathbf{S}_j(\mathbf{a}\cdot\mathbf{s}^h) = \tilde{\mathbf{s}}_j^{h+1}.$$

Thus the induction step goes through, and the claim follows. □

The main result of this section is the following [9, 21].

Theorem 7.1 *Let* $N = \langle \{0,1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, F \rangle$ *be a complete binary network, and let* $\mathbf{N} = \langle \{0, \Phi, 1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, \mathbf{F} \rangle$ *be its ternary counterpart. If* N *is started in total state* $a{\cdot}b$*, then the result* $\mathbf{s}^A$ *of Algorithm* A *for* $\mathbf{N}$ *is equal to the lub of the set of all the states reachable from the initial state in the GMW analysis of* N*, i.e.,*

$$\mathbf{s}^A = lub\, reach(R_a(b)).$$

Proof: By Proposition 7.2, $lub\, reach(R_a(b))$ is covered by the result of Algorithm A for $\mathbf{N}$. It remains to be shown that the *lub* of the reachable states of N covers $\mathbf{s}^A$. This follows from Corollary 7.2 on page 136. In the corollary, it is shown that, for every vertex j, there is a state $s^j \in \{0,1\}^m$ such that $bR_a^* s^j$ and $\mathbf{s}_j^A \sqsubseteq lub\{b_j, s_j^j\}$. This sufices to prove the result. □

7.3 Algorithm B

In Algorithm B, we see how much of the uncertainty introduced by Algorithm A is eventually removed; hence we start the network in the state generated by Algorithm A.

Algorithm B
$h := 0$;
$\mathbf{t}^0 := \mathbf{s}^A$;
repeat
  $h := h + 1$;
  $\mathbf{t}^h := \mathbf{S}(a{\cdot}\mathbf{t}^{h-1})$;
until $\mathbf{t}^h = \mathbf{t}^{h-1}$;

Proposition 7.4 *Algorithm* B *generates a finite sequence* $\mathbf{t}^0, \ldots, \mathbf{t}^B$ *of states, where* $B \leq m$*. Furthermore, this sequence is monotonically decreasing, i.e.,*

$$\mathbf{t}^h \sqsupset \mathbf{t}^{h+1}, \text{ for } 0 \leq h < B.$$

Proof: We first prove by induction on h that $\mathbf{t}^h \sqsupseteq \mathbf{t}^{h+1}$. For the basis, observe that $\mathbf{s}^A = lub\{\mathbf{s}^A, \mathbf{S}(a{\cdot}\mathbf{s}^A)\}$. It follows from the properties of *lub* that $\mathbf{t}^0 = \mathbf{s}^A \sqsupseteq \mathbf{S}(a{\cdot}\mathbf{s}^A) = \mathbf{t}^1$. Now assume inductively that $\mathbf{t}^h \sqsupseteq \mathbf{t}^{h+1}$. By the monotonicity of $\mathbf{S}$ it follows that $\mathbf{t}^{h+1} = \mathbf{S}(a{\cdot}\mathbf{t}^h) \sqsupseteq \mathbf{S}(a{\cdot}\mathbf{t}^{h+1}) = \mathbf{t}^{h+2}$ and the induction step goes through.

In view of this result, at least one state variable must change from Φ to a binary value in each step of the algorithm; otherwise the algorithm terminates. Since there are m state variables, B cannot exceed m, and the proposition follows. □

Proposition 7.5 *The least upper bound of the set of all the states in the outcome of the GMW analysis of a network N is covered by the result of Algorithm* B *for* $\mathbf{N}$, *i.e.,*

$$lub\, out(R_a(b)) \sqsubseteq \mathbf{t}^B.$$

Moreover, for every $h \geq 0$,

$$lub\, out(R_a(b)) \sqsubseteq \mathbf{t}^h.$$

Proof: We prove the latter claim by induction on h. If $h = 0$, then $\mathbf{t}^h = \mathbf{s}^A$. Since $out(R_a(b)) \subseteq reach(R_a(b))$, we have $lub\, out(R_a(b)) \sqsubseteq \mathbf{s}^A$ by Theorem 7.1, and the basis holds. Now suppose that $h > 0$ and that $\mathbf{t}^h$ satisfies the claim, but $\mathbf{t}^{h+1}$ does not. Then there must exist $c \in out(R_a(b))$ and a vertex i such that $c_i \not\sqsubseteq (\mathbf{t}^{h+1})_i$. Since $c_i \in \{0,1\}$, this can only happen if $(\mathbf{t}^{h+1})_i = \overline{c_i}$. We now assert that the excitation $S_i(a{\cdot}d)$ is equal to $(\mathbf{t}^{h+1})_i$ for every state d in $out(R_a(b))$. First note that

$$S(a{\cdot}d) = \mathbf{S}(a{\cdot}d) \sqsubseteq \mathbf{S}(a{\cdot}\mathbf{t}^h) = \mathbf{t}^{h+1},$$

where the inequality follows from the inductive assumption (which implies $d \sqsubseteq \mathbf{t}^h$) and the monotonicity of $\mathbf{S}$. Now, since $(\mathbf{t}^{h+1})_i$ is binary and $\sqsupseteq S_i(a{\cdot}d)$, it must be equal to $S_i(a{\cdot}d)$, as claimed. Now consider any nontransient cycle in $out(R_a(b))$. Since the excitation of the i-th variable is constant throughout the cycle, the value of the variable must be constant throughout the cycle. Since the cycle is nontransient, that value must be equal to the excitation. Thus $d_i = (\mathbf{t}^{h+1})_i = \overline{c_i}$ for every state d in the cycle. Since the nontransient cycle was arbitrary, we have shown that $d_i = (\mathbf{t}^{h+1})_i = \overline{c_i}$ for every state d in every nontransient cycle in $out(R_a(b))$. This, together with the fact that $S_i(a{\cdot}e) = \overline{c_i}$ for every state e in $out(R_a(b))$, implies that every state $d \in out(R_a(b))$ reachable from a nontransient cycle will also have $d_i = (\mathbf{t}^{h+1})_i = \overline{c_i}$. However, these results together imply that $d_i = \overline{c_i}$ for every $d \in out(R_a(b))$, contradicting the assumption that $c \in out(R_a(b))$. Hence, the induction step goes through. The main claim of the proposition now follows in view of Proposition 7.4. □

The characterization of the results of Algorithm B is given by:

Theorem 7.2 *Let* $N = \langle \{0,1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, F \rangle$ *be a complete binary network, and let* $\mathbf{N} = \langle \{0, \Phi, 1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, \mathbf{F} \rangle$ *be its ternary counterpart. If* N *is started in total state* $a{\cdot}b$, *then the result* $\mathbf{t}^B$ *of Algorithm* B *is equal to the lub of the outcome of the GMW analysis, i.e.,*

$$\mathbf{t}^B = lub\, out(R_a(b)).$$

Proof: By Proposition 7.5, $lub\, out(R_a(b))$ is covered by the result of Algorithm B. It remains to be shown that the *lub* of all the states in the outcome of N covers $\mathbf{t}^B$. This follows from Lemma 7.9 given on page 141. In the lemma it is shown that there exists a nontransient cycle Z reachable from the initial state and such that the *lub* of all the states in Z covers $\mathbf{t}^B$. This suffices to prove the theorem. □

7.4 Feedback-Delay Models

We have seen in Section 6.5 that the feedback-delay model used in conjunction with the GMW analysis is not quite correct; while it predicts the correct stable states, it does not predict the correct transitions among them. We show in this section that the feedback-delay model is completely correct when used in conjunction with ternary simulation.

Recall that a set $\mathcal{F}$ of vertices of a directed graph $\mathcal{G}$ is called a feedback-vertex set if the removal from $\mathcal{G}$ of all the vertices in $\mathcal{F}$, along with all their incident edges, results in an acyclic graph. We show that, to obtain the same results as does the GMW analysis of a complete network, it is sufficient to associate state variables with inputs and any feedback-vertex set, if ternary simulation is used. The results are independent of the choice of the vertex set. First, however, we describe the process of removing one state variable from the ternary network.

Without loss of generality, we assume that the variable to be removed is variable m—if some other variable is to be removed, we can always renumber the vertices. Let $\mathbf{N} = \langle \{0, \Phi, 1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, \mathbf{F} \rangle$ be any ternary network in which $\mathbf{s}_m$ is a state variable with excitation function $\mathbf{S}_m$ that does not depend on any input excitation function nor on the value $\mathbf{s}_m$ itself. Since $\mathbf{S}_m$ is independent of $\mathbf{s}_m$, we have $\mathbf{S}_m(\mathbf{X}{\cdot}\mathbf{s}_1 \cdots \mathbf{s}_{m-1}{\cdot}\mathbf{s}_m) = \mathbf{S}_m(\mathbf{X}{\cdot}\mathbf{s}_1 \cdots \mathbf{s}_{m-1}{\cdot}\Phi)$ for every $\mathbf{X}$ and $\mathbf{s}_i$, $1 \leq i \leq m$.

Given $\mathbf{N}$ we now define $\dot{\mathbf{N}} = \langle \{0, \Phi, 1\}, \mathcal{X}, \dot{\mathcal{S}}, \dot{\mathcal{E}}, \mathbf{F} \rangle$ as follows: $\dot{\mathcal{S}} = \mathcal{S} - \{m\}$ with labels $\dot{\mathbf{s}} = \dot{\mathbf{s}}_1 \ldots \dot{\mathbf{s}}_{m-1}$ and excitation functions $\dot{\mathbf{S}} = \dot{\mathbf{S}}_1 \cdots \dot{\mathbf{S}}_{m-1}$, where

$$\dot{\mathbf{S}}_i = \mathbf{S}_i(\mathbf{X}{\cdot}\dot{\mathbf{s}}{\cdot}\mathbf{S}_m(\mathbf{X}{\cdot}\dot{\mathbf{s}}{\cdot}\Phi)) \quad \text{for } 1 \leq i < m,$$

and with circuit equations

$$\dot{\mathbf{F}}_i(\mathbf{X}{\cdot}\dot{\mathbf{s}}) = \begin{cases} \mathbf{S}_m(\mathbf{X}{\cdot}\dot{\mathbf{s}}{\cdot}\Phi) & \text{if } i = m, \\ \mathbf{F}_i(\mathbf{X}{\cdot}\dot{\mathbf{s}}{\cdot}\mathbf{S}_m(\mathbf{X}{\cdot}\dot{\mathbf{s}}{\cdot}\Phi)) & \text{otherwise.} \end{cases}$$

As usual, the edge set defines the dependencies. It should be noted that $\dot{\mathcal{E}}$ can contain new edges introduced by this process. Note also that we must use ternary, and not Boolean, algebra in manipulating and simplifying the expressions for the excitation functions and circuit equations.

To illustrate the removal of a state variable, consider the network $\mathbf{N}$ in Figure 7.6, with excitation functions

$$\mathbf{S}_1 = \mathbf{X}_1, \quad \mathbf{S}_2 = \overline{(\mathbf{s}_1 + \mathbf{s}_4)} \quad \mathbf{S}_3 = \mathbf{X}_2, \quad \mathbf{S}_4 = \overline{(\mathbf{s}_2 + \mathbf{s}_3)},$$

and circuit equations

$$\mathbf{F}_1 = \mathbf{s}_1, \quad \mathbf{F}_2 = \mathbf{s}_2, \quad \mathbf{F}_3 = \mathbf{s}_3, \quad \mathbf{F}_4 = \mathbf{s}_4, \quad \mathbf{F}_5 = \mathbf{s}_2, \quad \mathbf{F}_6 = \mathbf{s}_4.$$

We remove the state variable $\mathbf{s}_4$. The reduced network $\dot{\mathbf{N}}$ is shown in Figure 7.6.

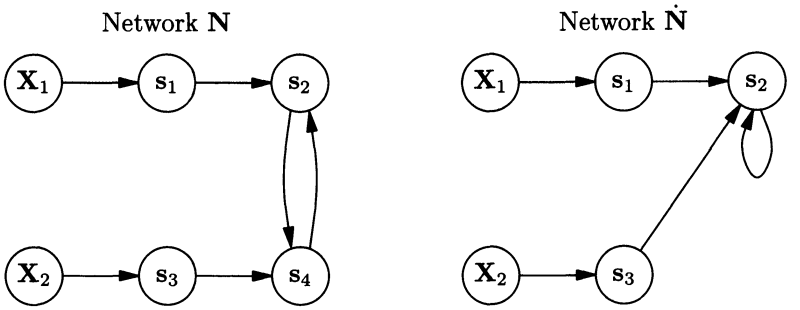


FIGURE 7.6. Illustrating state removal.

Here the excitation functions are

$$\dot{\mathbf{S}}_1 = \mathbf{X}_1, \quad \dot{\mathbf{S}}_2 = \overline{(\dot{\mathbf{s}}_1 + \overline{(\dot{\mathbf{s}}_2 + \dot{\mathbf{s}}_3)})}, \quad \dot{\mathbf{S}}_3 = \mathbf{X}_2,$$

and the circuit equations are

$$\begin{array}{lll} \mathbf{F}_1 = \dot{\mathbf{s}}_1, & \mathbf{F}_2 = \dot{\mathbf{s}}_2, & \mathbf{F}_3 = \dot{\mathbf{s}}_3, \\ \mathbf{F}_4 = \overline{(\dot{\mathbf{s}}_2 + \dot{\mathbf{s}}_3)}, & \mathbf{F}_5 = \dot{\mathbf{s}}_2, & \mathbf{F}_6 = \overline{(\dot{\mathbf{s}}_2 + \dot{\mathbf{s}}_3)}. \end{array}$$

Proposition 7.6 *Assume $a{\cdot}b$ is a (binary) total state of network $\mathbf{N}$ such that vertex m is stable, i.e., $b_m = \mathbf{S}_m(a{\cdot}b)$. Let $\mathbf{s}^A$ ($\mathbf{t}^B$) be the result of Algorithm* A (B) *for $\mathbf{N}$, when $\mathbf{N}$ is started in state $a{\cdot}b$. Similarly, let $\dot{\mathbf{s}}^A$ ($\dot{\mathbf{t}}^B$) be the result of Algorithm* A (B) *for $\dot{\mathbf{N}}$, when $\dot{\mathbf{N}}$ is started in state $a{\cdot}\dot{b}$, where $\dot{b}_j = b_j$ for $1 \le j \le m-1$. Then, for $1 \le j \le m-1$,*

$$\mathbf{s}_j^A = \dot{\mathbf{s}}_j^A \quad \textit{and} \quad \mathbf{t}_j^B = \dot{\mathbf{t}}_j^B.$$

Proof: Consider Algorithm A for network $\mathbf{N}$. If the excitation of m never changes, i.e., if $\mathbf{S}_m(a{\cdot}\mathbf{s}^i) = \mathbf{s}_m^i = b_m$, for $0 \le i \le A$, the proposition holds trivially. Hence, assume $\mathbf{S}_m(a{\cdot}\mathbf{s}^i)$ changes for the first time at step $r > 0$. From the monotonicity of Algorithm A, the fact that $\mathbf{S}_m$ does not depend on any input excitation, and the assumption that m was stable in the total state $a{\cdot}b$, we can conclude that

$$\mathbf{S}_m(a{\cdot}\mathbf{s}^i) = \begin{cases} b_m & \text{if } i < r, \\ \Phi & \text{if } i \ge r. \end{cases}$$

From this and the definition of Algorithm A it follows that

$$\mathbf{s}_m^i = \begin{cases} b_m & \text{if } i < r+1, \\ \Phi & \text{if } i \ge r+1. \end{cases}$$

Clearly, $\dot{\mathbf{s}}_j^i = \mathbf{s}_j^i$ for $1 \le j \le m-1$ and $0 \le i \le r$. Since $\mathbf{S}_m(a{\cdot}\mathbf{s})$ does not depend on $\mathbf{s}_m$, it follows that for $i \le r$ we have

$$\mathbf{S}_m(a{\cdot}\dot{\mathbf{s}}^i{\cdot}\Phi) = \mathbf{S}_m(a{\cdot}\dot{\mathbf{s}}^i{\cdot}b_m) = \mathbf{S}_m(a{\cdot}\mathbf{s}^i) = \begin{cases} b_m & \text{if } i < r, \\ \Phi & \text{if } i = r. \end{cases}$$

However, by Proposition 7.1, $\dot{\mathbf{s}}^r \sqsubseteq \dot{\mathbf{s}}^i$ for $i \geq r$; since $\mathbf{S}_m$ is monotone, it follows that

$$\mathbf{S}_m(a{\cdot}\dot{\mathbf{s}}^i{\cdot}\Phi) = \begin{cases} b_m & \text{if } i < r, \\ \Phi & \text{if } i \geq r. \end{cases}$$

Consequently,

$$\mathbf{s}_m^i \sqsubseteq \mathbf{S}_m(a{\cdot}\dot{\mathbf{s}}^i{\cdot}\Phi) \sqsubseteq \mathbf{s}_m^{i+1}. \tag{7.1}$$

We now proceed by induction on i to show that

$$\mathbf{s}_j^i \sqsubseteq \dot{\mathbf{s}}_j^i \sqsubseteq \mathbf{s}_j^{i+1} \quad \text{for } 1 \leq j \leq m-1. \tag{7.2}$$

For the basis, observe that $\mathbf{s}_j^0 = b_j$ and $\dot{\mathbf{s}}_j^0 = b_j$ for $1 \leq j \leq m-1$. By Proposition 7.1, $\mathbf{s}^0 \sqsubseteq \mathbf{s}^1$ and the basis holds. Assume inductively that $\mathbf{s}_j^i \sqsubseteq \dot{\mathbf{s}}_j^i \sqsubseteq \mathbf{s}_j^{i+1}$ for $1 \leq j \leq m-1$. This hypothesis, the monotonicity of *lub* and of $\mathbf{S}$, and (7.1), yield for $1 \leq j \leq m-1$

$$\begin{aligned} \mathbf{s}_j^{i+1} &= lub\{\mathbf{s}_j^i, \mathbf{S}_j(a{\cdot}\mathbf{s}^i)\} \\ &\sqsubseteq lub\{\dot{\mathbf{s}}_j^i, \mathbf{S}_j(a{\cdot}\dot{\mathbf{s}}^i{\cdot}\mathbf{s}_m^i)\} \\ &\sqsubseteq lub\{\dot{\mathbf{s}}_j^i, \mathbf{S}_j(a{\cdot}\dot{\mathbf{s}}^i{\cdot}\mathbf{S}_m(a{\cdot}\dot{\mathbf{s}}^i{\cdot}\Phi))\} = lub\{\dot{\mathbf{s}}_j^i, \dot{\mathbf{S}}_j(a{\cdot}\dot{\mathbf{s}}^i)\} = \dot{\mathbf{s}}_j^{i+1} \\ &\sqsubseteq lub\{\dot{\mathbf{s}}_j^i, \mathbf{S}_j(a{\cdot}\dot{\mathbf{s}}^i{\cdot}\mathbf{s}_m^{i+1})\} \\ &\sqsubseteq lub\{\mathbf{s}_j^{i+1}, \mathbf{S}_j(a{\cdot}\mathbf{s}^{i+1})\} = \mathbf{s}_j^{i+2}, \end{aligned}$$

and the induction goes through.

Given (7.2) and the fact that Algorithm A converges (Proposition 7.1), it follows immediately that $\mathbf{s}_i^A = \dot{\mathbf{s}}_i^A$, for $1 \leq j \leq m-1$. The proof of the second claim follows in a dual fashion. □

Given any feedback-vertex set, we can construct a corresponding reduced network, as is shown in the next result.

Proposition 7.7 *For any feedback-vertex set $\mathcal{F}$ of a complete network $\mathbf{N}$ there exists a reduced network $\dot{\mathbf{N}}$ with vertex set equal to $\mathcal{X} \cup \mathcal{I} \cup \mathcal{F}$, where $\mathcal{I}$ is the set of input-delay vertices.*

Proof: By definition of feedback-vertex set, an acyclic graph results when the feedback vertices are removed from $\mathbf{N}$. The vertices in $\mathbf{N}$ can be classified as follows. Feedback vertices, input vertices, and input-delay vertices are all of rank 0. Inductively, the vertices with incoming edges only from vertices in ranks 0 to $i-1$, which are not of rank 0 through $i-1$, are rank i vertices. Use the vertex removal process to remove all vertices of rank 1. Note that no feedback is introduced by this construction in the acyclic graph consisting of the vertices that are not in $\mathcal{F}$. Thus vertices that were of rank 2 are now of rank 1 in the new graph. This process can be repeated until all the vertices not in $\mathcal{X} \cup \mathcal{I} \cup \mathcal{F}$ have been removed. □

Intuitively, one might think that the converse of Proposition 7.7 holds, i.e., that the vertices remaining after a network has been reduced constitute a feedback vertex set of the original network. This is not the case, however, as the following example shows. Consider the network in Figure 7.7, where the excitation functions are

$$\mathbf{S}_1 = \mathbf{X}, \qquad \mathbf{S}_2 = \mathbf{s}_1, \qquad \mathbf{S}_3 = \mathbf{s}_1,$$
$$\mathbf{S}_4 = \mathbf{s}_2\mathbf{s}_3 + \mathbf{s}_2\mathbf{s}_4 + \mathbf{s}_3\mathbf{s}_4, \quad \mathbf{S}_5 = \mathbf{s}_4 + \mathbf{s}_5.$$

If we first remove vertex $\mathbf{s}_2$, we get the reduced network $\dot{\mathbf{N}}$ shown in Figure 7.7 with excitation functions

$$\mathbf{S}_1 = \mathbf{X}, \qquad \mathbf{S}_3 = \mathbf{s}_1,$$
$$\mathbf{S}_4 = \mathbf{s}_1\mathbf{s}_3 + \mathbf{s}_1\mathbf{s}_4 + \mathbf{s}_3\mathbf{s}_4, \quad \mathbf{S}_5 = \mathbf{s}_4 + \mathbf{s}_5.$$

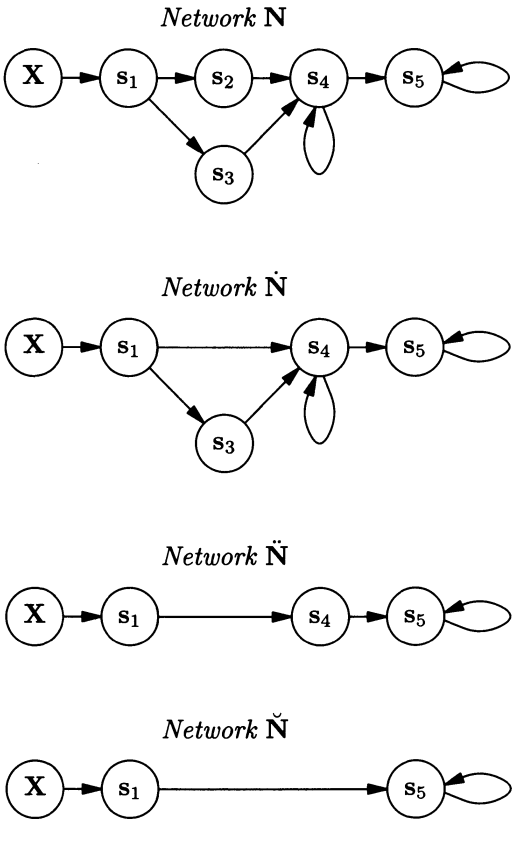


FIGURE 7.7. Reduction below feedback vertex set.

We now remove vertex s_3, obtaining the network $\ddot{\mathbf{N}}$ shown in Figure 7.7 with excitation functions

$$\mathbf{S}_1 = \mathbf{X}, \quad \mathbf{S}_4 = s_1 s_1 + s_1 s_4 + s_1 s_4 = s_1, \quad \mathbf{S}_5 = s_4 + s_5.$$

Note that the dependency on vertex s_4 was removed from $\mathbf{S}_4$ in this tranformation. Finally, we remove vertex s_4, and obtain network $\breve{\mathbf{N}}$ shown in Figure 7.7 with excitation functions

$$\mathbf{S}_1 = \mathbf{X}, \quad \mathbf{S}_5 = s_1 + s_5.$$

Reductions below the minimal feedback vertex set, such as the one described here, are possible because some of the excitation functions become degenerate. On the other hand, the feedback vertex set reflects only the structure of the graph.

We are now ready to state the main theorem of this section.

Theorem 7.3 *The results of Algorithm* A *for a network* $\mathbf{N}$ *and for any reduced version* $\dot{\mathbf{N}}$ *of* $\mathbf{N}$ *are equal with respect to the feedback variables. The same is true for Algorithm* B.

Proof: Proposition 7.6 states that the results are equal for the remaining vertices if only one vertex is removed. The result now follows by induction on the number of vertices not in $\mathcal{X} \cup \mathcal{I} \cup \mathcal{F}$ and by Proposition 7.7. □

Corollary 7.1 *The results of Algorithm* A *for a network* $\mathbf{N}$ *and for any feedback-vertex model of* $\mathbf{N}$ *are equal with respect to the feedback variables. The same is true for Algorithm* B.

7.5 Hazards

Until now, our main concern in the analysis of a network has been its outcome (in the case of GMW analysis) or the least upper bound of the outcome (in the case of ternary simulation). While it is true that the final stable state reached by a network is of great importance, there are other concerns that must be addressed. Typical examples of such secondary constraints are the so-called static hazards, where a network output has the same value after a transition as it did before the transition, but may receive the complementary value for a short time during the transition. In this chapter we consider two types of hazard phenomena that have been studied in the classical theory of switching circuits.

7.5.1 Static Hazards

We illustrate the phenomenon of static hazards with a simple example. Consider the network of Figure 7.8. The output y_4 is defined by the Boolean

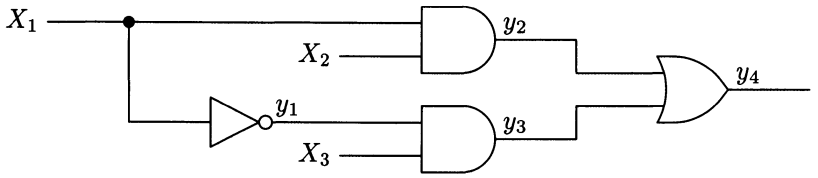


FIGURE 7.8. Circuit with a static hazard.

expression $y_4 = X_1X_2 + \overline{X_1}X_3$. One verifies that y_4 is 1 when $X_1 = 0, X_2 = 1, X_3 = 1$, and also when $X_1 = 1, X_2 = 1, X_3 = 1$. Suppose that the network is started in the stable total state $X \cdot y = 111 \cdot 0101$, and that X_1 changes to 0. From the equation for y_4, one would expect the value of y_4 to remain 1. If the delay of the inverter is appreciable, however, gate y_2 may become 0 before the state of the inverter changes and y_3 becomes 1. Thus y_4 may become temporarily 0; this situation is called a static hazard in the output y_4. Static hazards are undesirable whenever a circuit is used as a component of a larger system. In this case, a second component of the system may incorrectly respond to the short pulse that may appear on the output of the first component, thus causing an error in the outcome of the second component.

Since Huffman [68] first defined the static hazard in 1957, many methods have been proposed for finding static hazards in both combinational and sequential circuits. A majority of these, including those proposed by [68] and [92], operate in the well-known domain of Boolean algebra and have exponential time complexity in the worst case, while others [52, 99] use complex multivalued algebra. These methods are suitable only for small combinational circuits, but ternary simulation [23, 49, 148] provides us with a powerful algorithm that has polynomial time complexity for detecting static hazards on any state variable in a sequential circuit.

The fact that ternary algebra can be used to detect static hazards was first shown by [148] in 1964, but only for single input changes and combinational circuits. The work in [49] expanded this method into an algorithm that is able to detect all static hazards in sequential circuits during multiple input changes. We now describe the algorithm of [49] adapted to our notation.

Let $N = \langle \{0,1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, F \rangle$ be a binary network, and let $\mathbf{N} = \langle \{0, \Phi, 1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, \mathbf{F} \rangle$ be its ternary extension. Let $a \in \{0,1\}^n$, $b \in \{0,1\}^m$, and let R_a be the GMW relation for N.

A network N is said to have a *static hazard* on a state variable s_i from state $a \cdot b$ if and only if

- $b_i = c_i$, for every $c \in out(R_a(b))$, but
- there exists a state d such that bR_a^+d and $d_i = \overline{b_i}$.

Theorem 7.4 *A complete network N has a static hazard on variable s_i from state $a{\cdot}b$ if and only if its ternary extension* **N** *has the following property: The result $\mathbf{s}_i^A$ of Algorithm* A *applied to $a{\cdot}b$ is Φ, while the result $\mathbf{t}_i^B$ of Algorithm* B *is equal to the initial value b_i.*

Proof: The initial binary value b_i is the same as the result of Algorithm B if and only if the value in any state of the outcome is also that binary value. The result of Algorithm A gives the least upper bound of all the states reachable from b. The result of Algorithm A is Φ, if and only if there exists a state d reachable from b such that $d_i \neq b_i$. Hence the theorem follows. □

7.5.2 Dynamic Hazards

Informally, a network N is said to have a *dynamic hazard* if and only if there is a variable that has one value in the initial state, the complementary value in all the states of the outcome, and changes more than once in going from the initial state to some state in the outcome. Formally, N has a *dynamic b_i to $\overline{b_i}$ hazard on a state variable s_i from state $a{\cdot}b$* if and only if

- $c_i = \overline{b_i}$, for every $c \in out(R_a(b))$, but
- there exist states d and e such that bR_a^+d, dR_a^+e, $d_i = \overline{b_i}$, and $e_i = b_i$.

The network of Figure 7.9 has a dynamic hazard on state variable s_4 from state 1·1000. One verifies that, in the GMW model using the gate-state network, the variables may change in the following order: $s_3, s_4, s_1, s_3, s_4, s_2, s_4$. The outcome of the transition is stable state 1·0101.

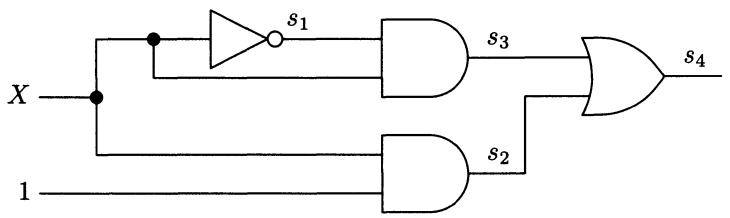


FIGURE 7.9. Circuit with a dynamic hazard.

To the best of our knowledge, there are no efficient methods for finding dynamic hazards.

7.6 Ternary Simulation and the GSW Model

We now show that, if a sufficient number of delays is included in the network model, a GSW analysis yields the same outcome as a GMW analysis. This

shows that concurrency can be simulated by the interleaving of actions in gate networks. Also, this result provides another illustration of the trade-off between the power of the race model and the amount of detail in the network model. This section is based on [48].

Consider a complete network $N = \langle \{0,1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, F \rangle$, and a related network $\widehat{N} = \langle \{0,1\}, \mathcal{X}, \widehat{\mathcal{S}}, \widehat{\mathcal{E}}, \widehat{F} \rangle$, where we have added a (wire) state variable s_i' for every state variable s_i in N as follows. If the state of N is denoted by $X \cdot s$, then the state of $\widehat{N}$ is denoted by $X \cdot ss'$, where $s = s_1, \ldots, s_m$ and $s' = s_1', \ldots, s_m'$. Furthermore, if the excitation vector S of N is $S(X \cdot s)$, then the excitation vector $\widehat{S}$ of $\widehat{N}$ is $S(X \cdot s')S'(s)$, where S' is the identity function taking an m-tuple as argument. In other words, $S_i' = s_i$, for all $i = 1, \ldots, m$. A similar change is made in all the circuit equations. We denote by $R_a(bb)$ the GMW relation of $\widehat{N}$, and by $\widehat{R}_a(bb)$ the GSW relation of $\widehat{N}$.

Proposition 7.8 *Suppose N is started in state $a \cdot b$. For each path beginning with b in the $R_a(b)$ relation of N, there is a similar path beginning with bb in the $\widehat{R}_a(bb)$ relation of $\widehat{N}$, where each total state $X \cdot s$ of N corresponds to the total state $X \cdot ss$ of $\widehat{N}$.*

Proof: The proof is by induction on the path length l. Clearly the result holds trivially for $l = 0$, by construction of $\widehat{N}$. Now suppose state c has been reached in N and state cc in $\widehat{N}$. We show that, for each possible one-step transition from c to d, there is a multistep transition from cc to dd. In the transition of N, some nonempty set $\mathcal{V}$ of unstable variables has been changed. Corresponding to this change, we obtain a sequence of transitions in $\widehat{N}$ from $ss' = cc$, by first changing the unstable variables of $\mathcal{V}$ in the vector s one by one. It is easily seen that these variables are indeed unstable, and that changing one of them does not affect the instability of the others, since that is determined by the variables in s'. Thus we reach the state dc. All the variables in s' corresponding to set $\mathcal{V}$ are now unstable. We proceed to change them one at a time, thus reaching state dd in $\widehat{N}$. □

From this result it follows that the GSW model with an extra wire delay for each original state variable of N can simulate each behavior in the GMW analysis of N.

Next, we define the outcome of the GSW analysis of any network in the same way as we have defined it in the GMW analysis, except that the relation graph $\widehat{R}_a(bb)$ is used. For convenience let $out(\widehat{R}_a(bb)) \downarrow s$ denote the set $\{q \in \{0,1\}^m \mid qq' \in out(\widehat{R}_a(bb))$ for some $q' \in \{0,1\}^m\}$. We are now in a position to prove the main result of this section.

Theorem 7.5 *The outcome of the GSW analysis of $\widehat{N}$ restricted to the first m state components is equal to the outcome of the GMW analysis of N, i.e.,*

$$out(\widehat{R}_a(bb)) \downarrow s = out(R_a(b)).$$

Proof: Using the construction in the proof of Proposition 7.8, we see that, for each state c in a cycle of the $R_a(b)$ relation graph of N, state cc also occurs as a state of a cycle in the $\widehat{R}_a(bb)$ relation graph of $\widehat{N}$. Moreover, from the definition of nontransient cycle, it follows that the cycle in $\widehat{R}_a(bb)$ constructed as above to correspond to a nontransient cycle of N is also nontransient. Consequently, the outcome of the GMW analysis of N is a subset of the outcome of the GSW analysis of $\widehat{N}$ restricted to the first m state components. In symbols,

$$out(R_a(b)) \subseteq out(\widehat{R}_a(bb)) \downarrow s.$$

Clearly, the outcome of the GMW analysis of $\widehat{N}$ contains the outcome of the GSW analysis of $\widehat{N}$, i.e.,

$$out(\widehat{R}_a(bb)) \subseteq out(R_a(bb)).$$

This relationship is preserved under the restriction to s, i.e.,

$$out(\widehat{R}_a(bb)) \downarrow s \subseteq out(R_a(bb)) \downarrow s.$$

Note that the variables of N constitute a feedback vertex set, because we have assumed that N is a complete network. Consequently, Theorem 7.3 applies, implying that the outcome of the GMW analysis of $\widehat{N}$ restricted to the first m variables is equal to the outcome of the GMW analysis of N. In symbols, $out(R_a(bb)) \downarrow s = out(R_a(b))$. Our claim now follows. □

7.7 Ternary Simulation and the XMW Model

The extended multiple-winner (XMW) race model defined in Section 6.6 is particularly appropriate for analyzing the behavior of switch-level circuits, since the network model for such circuits is usually defined using a ternary domain. Like the GMW race model, however, the XMW model is computationally intractable. In this section we show that ternary simulation can be used to determine the outcome of the XMW analysis. First we prove that the result of Algorithm A computes the *lub* of all the reachable states in the XMW relation. Second, we show that the result of Algorithm B is equal to the *lub* of the outcome of the XMW analysis. A major difference between these results and those of Sections 7.2 and 7.3 is that the correspondence is not limited to complete networks. Thus, for example, determining the *lub* of the outcome of the XMW analysis for an input- and feedback-state network model can be carried out by performing ternary simulation.

Theorem 7.6 *Let* $\mathbf{N} = \langle \{0, \Phi, 1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, \mathbf{F} \rangle$ *be a ternary network. If* $\mathbf{N}$ *is started in binary total state* $a{\cdot}b$*, then the result* $\mathbf{s}^A$ *of Algorithm* A *for* $\mathbf{N}$ *is equal to the lub of the set of all the states reachable from the initial state in the XMW analysis of* $\mathbf{N}$*, i.e.,* $\mathbf{s}^A = lub\{\mathbf{s} \in \{0, \Phi, 1\}^m \mid b\mathbf{R}_a^*\mathbf{s}\}$.

Proof: First we claim that $\mathbf{s}^A \sqsubseteq lub\{\mathbf{s} \in \{0,\Phi,1\}^m \mid b\mathbf{R}_a^*\mathbf{s}\}$. To verify this, it is sufficient to establish that consecutive states in Algorithm A are $\mathbf{R}_a$ related, i.e., that $b\mathbf{R}_a\mathbf{s}^h$, for $h \geq 0$. This would imply, in particular, that $\mathbf{s}^A \in \{\mathbf{s} \in \{0,\Phi,1\}^m \mid b\mathbf{R}_a^*\mathbf{s}\}$. Our claim follows by induction on h. By the definition of the $\mathbf{R}_a$ relation in Section 6.6, one of the possible successor states is the *lub* of the current state and the excitation—precisely the value used by Algorithm A.

To show the converse, we claim that

$$lub\{c \in \{0,\Phi,1\}^m \mid b\mathbf{R}_a^h c\} \sqsubseteq \mathbf{s}^A.$$

This follows by induction on h and the observation that, if $\mathbf{c},\mathbf{d} \in \{0,\Phi,1\}^m$ and $\mathbf{c}\mathbf{R}_a\mathbf{d}$, then $\mathbf{d} \sqsubseteq lub\{\mathbf{c}, \mathbf{S}(a{\cdot}\mathbf{c})\}$. □

Theorem 7.7 *Let* $\mathbf{N} = \langle\{0,\Phi,1\},\mathcal{X},\mathcal{S},\mathcal{E},\mathbf{F}\rangle$ *be a ternary network. If* $\mathbf{N}$ *is started in binary total state* $a{\cdot}b$*, then the result* $\mathbf{t}^B$ *of Algorithm* B *for* $\mathbf{N}$ *is equal to the lub of the outcome of the XMW analysis, i.e.,*

$$\mathbf{t}^B = lub\,\mathbf{out}(\mathbf{R}_a(b)).$$

Proof: We begin by establishing that $\mathbf{t}^B \sqsubseteq lub\,\mathbf{out}(\mathbf{R}_a(b))$. Since $\mathbf{t}^0 = \mathbf{s}^A$ and, $b\mathbf{R}_a^*\mathbf{s}^A$ (as shown above), we have $b\mathbf{R}_a^*\mathbf{t}^0$. By the definition of the XMW relation, it follows that $\mathbf{t}^h\mathbf{R}_a\mathbf{t}^{h+1}$, for $0 \leq h < B$. Altogether, we have $b\mathbf{R}_a\mathbf{t}^B$. Since $\mathbf{t}^B$ is a stable state, it must be in the outcome of the XMW relation. Hence, the claim follows.

To prove the converse, we show that $lub\,\mathbf{out}(\mathbf{R}_a(b)) \sqsubseteq \mathbf{t}^h$, for $h \geq 0$, by induction on h. The basis follows trivially from Theorem 7.6 and the observation that the outcome is a subset of the set of reachable states. For the induction step, assume that $\mathbf{s}$ is an arbitrary state in the outcome of the XMW analysis and that $\mathbf{s} \sqsubseteq \mathbf{t}^h$. We now claim that $\mathbf{s} \sqsubseteq \mathbf{t}^{h+1}$. Note first that $\mathbf{t}^{h+1} = \mathbf{S}(a{\cdot}\mathbf{t}^h)$. By the monotonicity of $\mathbf{S}$, it follows that $\mathbf{S}(a{\cdot}\mathbf{s}) \sqsubseteq \mathbf{S}(a{\cdot}\mathbf{t}^h) = \mathbf{t}^{h+1}$. Consider any vertex j. If $\mathbf{t}_j^{h+1} = \Phi$, then $\mathbf{s}_j \sqsubseteq \mathbf{t}_j^{h+1}$ holds trivially. Therefore, suppose that $\mathbf{t}_j^{h+1} = \alpha \in \{0,1\}$. We have shown above that $\mathbf{S}(a{\cdot}\mathbf{s}) \sqsubseteq \mathbf{t}^{h+1}$. Thus, the excitation of vertex j is equal to α in state $\mathbf{s}$. Because $\mathbf{s}$ is an arbitrary state in the outcome, the excitation of vertex j is equal to α in every state in the outcome. Since α is binary, it follows that $\mathbf{s}_j = \alpha$; thus $\mathbf{s}_j \sqsubseteq \mathbf{t}_j^{h+1}$. Since j was arbitrary, we can conclude that $\mathbf{s} \sqsubseteq \mathbf{t}^{h+1}$, and the induction step goes through. □

7.8 Proofs of Main Results

Let N and $\mathbf{N}$ be as in Theorem 7.1. Note that they are complete networks. Let $\mathcal{G}$ denote the set of gate vertices and let $\mathcal{W}$ denote the set of wire

vertices. It is often necessary to select the fan-in and fan-out vertices for a given vertex. Thus, for any gate vertex i, let its fan-in vertices be

$$\alpha^i = \{j \mid (j,i) \in \mathcal{E}\}.$$

Note that $\alpha^i \cap \alpha^j = \emptyset$ if $i \neq j$. With a slight abuse of notation, given a vector v of length m and $\alpha^i \neq \emptyset$, we write $\alpha^i(v)$ to denote the components of the vector corresponding to the fan-in vertices; thus, if $\alpha^i = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}$, then $\alpha^i(v) = v_{\alpha_1}, v_{\alpha_2}, \ldots, v_{\alpha_r}$. Similarly, for any gate vertex i, let its fan-out vertices be

$$\beta^i = \{j \mid (i,j) \in \mathcal{E}\}.$$

Again, note that $\beta^i \cap \beta^j = \emptyset$ if $i \neq j$. Given a vector v of length m and $\beta^i \neq \emptyset$, we write $\beta^i(v)$ to denote the components of the vector corresponding to the fan-out vertices of vertex i. Finally, if $\mathbf{s}^h$, $0 \leq h \leq A$, denotes the results of Algorithm A after h steps, and vertex $\mathbf{s}_j^A = \Phi$, let γ_j denote the step in which this vertex changes to Φ, i.e., if $\mathbf{s}_j^{k-1} = b_j$ and $\mathbf{s}_j^k = \Phi$ then $\gamma_j = k$.

The following technical result will be needed in the proof of Lemma 7.2.

Proposition 7.9 *Let N and $\mathbf{N}$ be as in Theorem 7.1. Let j be a gate vertex with indegree d_j and fan-in set α^j. Given $\mathbf{s} \in \{0,\Phi,1\}^m$, such that $\mathbf{s}_j = b_j \in \{0,1\}$ and $lub\{\mathbf{s}_j, \mathbf{S}_j(a{\cdot}\mathbf{s})\} = \Phi$, there exists $c^j \in \{0,1\}^{d_j}$ such that $c^j \sqsubseteq \alpha^j(\mathbf{s})$ and $S_j(a{\cdot}s) = \overline{b_j}$ for every $s \in \{0,1\}^m$ such that $\alpha^j(s) = c^j$.*

Proof: We prove the claim by contradiction. Assume that, for all $c^j \in \{0,1\}^{d_j}$ such that $c^j \sqsubseteq \alpha^j(\mathbf{s})$, there is some state $s \in \{0,1\}^m$ such that $c^j = \alpha^j(s)$ and $S_j(a{\cdot}s) = b_j$. Since S_j depends only on the vertices in α^j, we can conclude that $S_j(a{\cdot}s) = b_j$ implies $S_j(a{\cdot}s') = b_j$ for every $s' \in \{0,1\}^m$ such that $\alpha^j(s) = \alpha^j(s')$. Altogether, we have that $S_j(a{\cdot}s) = b_j$ for every $s \in \{0,1\}^m$ such that $\alpha^j(s) \sqsubseteq \alpha^j(\mathbf{s})$. However, by the definition of ternary extension, this implies that $\mathbf{S}_j(\mathbf{s}) = b_j$, which means that $lub\{\mathbf{s}_j, \mathbf{S}_j(a{\cdot}\mathbf{s})\} = lub\{b_j, b_j\} = b_j$, contradicting the assumption that $lub\{\mathbf{s}_j, \mathbf{S}_j(a{\cdot}\mathbf{s})\} = \Phi$. □

In showing that ternary simulation is covered by GMW analysis, the following lemma is the key result for Algorithm A.

Lemma 7.2 *Let N and $\mathbf{N}$ be as in Theorem 7.1, and let $\mathbf{s}^h$, $0 \leq h \leq A$ be the result of Algorithm A after h steps. Then, for each h, there exists $s^h \in \{0,1\}^m$ such that*

1. *$bR_a^* s^h$;*
2. *if j is an input-delay or gate vertex then*

$$s_j^h = \begin{cases} b_j & \text{if } \mathbf{s}_j^h = b_j, \\ \overline{b_j} & \text{if } \mathbf{s}_j^h = \Phi; \end{cases}$$

3. *if j is a wire vertex in the fan-out set of vertex i and in the fan-in set of vertex k ($i = k$ is possible), and $\mathbf{s}_k^h = b_k$ then*

$$s_j^h = \begin{cases} b_j & \text{if } \mathbf{s}_i^h = b_i \text{ and } \mathbf{s}_j^h = b_j \ , \\ \overline{S_j(a{\cdot}s^h)} & \text{if } \mathbf{s}_i^h = \Phi \text{ or } \mathbf{s}_j^h = \Phi; \end{cases}$$

4. *if j is a wire vertex in the fan-out set of vertex i and in the fan-in set of vertex k ($i = k$ is possible), and $\mathbf{s}_i^h = \mathbf{s}_k^h = \Phi$ then*

$$\gamma_i \geq \gamma_k \ \ \textit{implies} \ \ s_j^h = \overline{S_j(a{\cdot}s^h)}.$$

Proof: We prove the lemma by induction on h. The reader may find it useful to follow the construction in the proof of the lemma in parallel with the construction in Figure 7.11 for the circuit of Figure 7.10 started in the (unstable) total state 1·00101110000011.

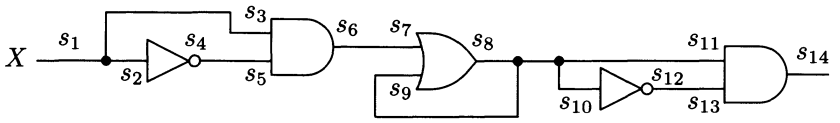


FIGURE 7.10. Circuit used to illustrate Lemma 7.2.

Algorithm A	GMW
$\mathbf{s}^0$ 0_1 0 1_0 0_1 1_0 0_1 1_0 0_1 1_0 0 0 1 1 0	s^0 0 0 1 0 1 0 1 0 1 0 0 1 1 0
	$\tilde{s}^1$ 0 0 1 0 1 0 0 0 1 0 0 1 1 0
$\mathbf{s}^1$ $\Phi_1 0_\Phi \Phi$ $\Phi_1 \Phi$ Φ Φ Φ Φ $0_\Phi 0_\Phi 1$ 1 0	s^1 1 0 0 1 0 1 0 1 0 0 0 1 1 0
	$\tilde{s}^2$ 1 0 0 1 0 1 0 1 0 0 0 1 1 0
$\mathbf{s}^2$ $\Phi_1 \Phi$ Φ Φ Φ Φ Φ Φ Φ Φ Φ 1_Φ 1 0_Φ	s^2 1 0 0 1 0 1 0 1 0 0 0 1 1 0
	$\tilde{s}^3$ 1 0 0 1 0 1 0 1 0 1 1 1 1 0
$\mathbf{s}^3$ $\Phi_1 \Phi$ Φ Φ Φ Φ Φ Φ Φ Φ Φ $1_\Phi \Phi$	s^3 1 0 0 1 0 1 0 1 0 1 1 0 1 1
	$\tilde{s}^4$ 1 0 0 1 0 1 0 1 0 1 1 0 1 1
$\mathbf{s}^4$ $\Phi_1 \Phi$ Φ Φ Φ Φ Φ Φ Φ Φ Φ Φ Φ Φ	s^4 1 0 0 1 0 1 0 1 0 1 1 0 1 1

FIGURE 7.11. Illustrating Lemma 7.2.

The basis, $h = 0$, follows trivially since $\mathbf{s}^0 = b \in \{0,1\}^m$. Assume inductively that state s^h has been reached and that s^h satisfies Properties 1–4. We show how to reach a state s^{h+1} that satisfies all four properties. We do this in two steps. We first show that there is a state $\tilde{s}^{h+1}$ reachable from s^h in which all input-delay and gate vertices that change to Φ in step $h+1$ in Algorithm A are unstable. We then conclude the proof by showing how s^{h+1} can be reached from $\tilde{s}^{h+1}$.

It is convenient to introduce the following shorthand. Let $\mathcal{C}^{h+1}$ be the set of gate and input-delay vertices that change to Φ in step $h+1$, i.e.,

$$\mathcal{C}^{h+1} = \{j \in \mathcal{I} \cup \mathcal{G} \mid \mathbf{s}_j^h = b_j \ \text{ and } \ \mathbf{s}_j^{h+1} = \Phi\}.$$

Now, let $\tilde{s}_j^{h+1} = s_j^h$ for every input-delay and gate vertex. If $j \in \mathcal{C}^{h+1}$, let $\alpha^j(\tilde{s}^{h+1}) = c^j$, where $c^j \in \{0,1\}^{d_j}$ is such that $c^j \sqsubseteq \alpha^j(\mathbf{s}^h)$ and $S_j(a{\cdot}s) = \overline{b_j}$ for all $s \in \{0,1\}^m$ such that $\alpha^j(s) = c^j$. By Proposition 7.9 such c^j is guaranteed to exist. If $j \notin \mathcal{C}^{h+1}$, let $\alpha^j(\tilde{s}^{h+1}) = \alpha^j(s^h)$. Note that this completely determines $\tilde{s}^{h+1}$. We first claim that every vertex in $\mathcal{C}^{h+1}$ is unstable in $\tilde{s}^{h+1}$. To verify this, we consider two cases. First, if j is an input-delay vertex, then h must be 0 and the input-delay excitation function X_j must be $\overline{b_j}$. Thus input-delay vertex j is unstable at $h = 0$. Since, by construction, no input-delay vertex changes in going from s^h to $\tilde{s}^{h+1}$, input-delay vertex j must still be unstable in $\tilde{s}^{h+1}$. On the other hand, if j is a gate vertex, then $\alpha^j(\tilde{s}^{h+1}) = c^j$. But c^j was chosen so that $S_j(a{\cdot}\tilde{s}^{h+1}) = \overline{b_j}$. By Property 2 of the induction hypothesis, $s_j^h = b_j$; thus vertex j is unstable in $\tilde{s}^{h+1}$.

We now claim that $s^h R_a^* \tilde{s}^{h+1}$. Clearly, the claim holds if $\tilde{s}^{h+1} = s^h$. Hence, assume $\tilde{s}^{h+1} \neq s^h$. It is sufficient to show that each vertex that changes in going from s^h to $\tilde{s}^{h+1}$ is unstable in total state $a{\cdot}s^h$. Let j be such a vertex, i.e., assume $\tilde{s}_j^{h+1} \neq s_j^h$. By construction, it follows that j must be a wire vertex in the fan-in set of some vertex $k \in \mathcal{C}^{h+1}$ and in the fan-out set of some vertex i. However, $\alpha^k(\tilde{s}^{h+1}) = c^k$ and, by definition, $c^k \sqsubseteq \alpha^k(\mathbf{s}^h)$. In particular, $c_j^k = \tilde{s}_j^{h+1} \sqsubseteq \mathbf{s}_j^h$. If $\mathbf{s}_i^h = b_i$ and $\mathbf{s}_j^h = b_j$ then, by Property 3 of the induction hypothesis, $s_j^h = b_j$. However, $\mathbf{s}_j^h = b_j$ implies that $\tilde{s}_j^{h+1} = b_j$ and thus in this case $\tilde{s}_j^{h+1} = s_j^h$. On the other hand, if $\mathbf{s}_i^h = \Phi$ or $\mathbf{s}_j^h = \Phi$ then, again by Property 3 of the induction hypothesis, we have $s_j^h = \overline{S_j(a{\cdot}s^h)}$, and thus vertex j is unstable in total state $a{\cdot}s^h$. Altogether, $\tilde{s}_j^{h+1}$ is either equal to s_j^h or $S_j(a{\cdot}s^h)$ for $1 \leq i \leq m$; thus $s^h R_a^* \tilde{s}^{h+1}$.

We are now ready to construct s^{h+1}. If $j \in \mathcal{C}^{h+1}$ let $s_j^{h+1} = S_j(a{\cdot}\tilde{s}^{h+1})$ and $\beta^j(s^{h+1}) = \beta^j(S(a{\cdot}\tilde{s}^{h+1}))$. If $j \notin \mathcal{C}^{h+1}$, let $s_j^{h+1} = \tilde{s}_j^{h+1}$ and $\beta^j(s^{h+1}) = \beta^j(\tilde{s}^{h+1})$. Note that this uniquely determines s^{h+1}. We now must verify that s^{h+1} satisfies Properties 1–4. First, it follows immediately from the construction that $\tilde{s}^{h+1} R_a^* s^{h+1}$. From the fact that $s^h R_a^* \tilde{s}^{h+1}$ and from the induction hypothesis, it follows that $b R_a^* s^{h+1}$ and Property 1 holds. Second, by construction, $\tilde{s}_j^{h+1} = s_j^h$ for every input-delay and gate vertex and the only gate and input-delay vertices that are changed in going from $\tilde{s}^{h+1}$ to s^{h+1} are those that change to Φ at step $h+1$ in Algorithm A; hence it follows from the induction hypothesis that Property 2 holds for every gate and input-delay vertex in s^{h+1}.

Now, consider any wire vertex j that is in the fan-out set of vertex i and in the fan-in set of vertex k and for which $\mathbf{s}_k^{h+1} = b_k$. If $\mathbf{s}_i^{h+1} = \mathbf{s}_i^h$ and $\mathbf{s}_j^{h+1} = \mathbf{s}_j^h$ then, by the construction, $s_i^{h+1} = s_i^h$ and $s_j^{h+1} = s_j^h$. Thus, by the induction hypothesis, Property 3 holds for j. On the other hand, if $i \in \mathcal{C}^{h+1}$, the construction of s^{h+1} ensures that each wire vertex in the fan-out set of gate i is unstable, since we simultaneously set its output to

its current excitation and change its input. Hence, Property 3 holds in this case too. Finally, if $\mathbf{s}_i^{h+1} = b_i$ but $\mathbf{s}_j^{h+1} = \Phi$, it follows immediately that h must be 0 and that the circuit was started in a state in which wire vertex j was unstable, i.e., $b_j = \overline{S_j(a{\cdot}b)}$. Since neither i nor k is in $\mathcal{C}^1$, it follows that $s_i^1 = s_i^0 = b_i$ and that $s_j^1 = s_j^0 = b_j$. Since the excitation of wire vertex j is completely determined by the value on a gate or input-delay vertex i, it follows that wire vertex j will remain unstable in total state $a{\cdot}s^{h+1}$ and Property 3 holds.

Finally, consider any wire vertex j such that $j \in \beta^i$, $j \in \alpha^k$, $\mathbf{s}_i^{h+1} = \mathbf{s}_k^{h+1} = \Phi$ and $\gamma_i \geq \gamma_k$. There are two cases to consider. If $i \in \mathcal{C}^{h+1}$ then, by the construction of s^{h+1}, we have $s_j^{h+1} = \overline{S_j(a{\cdot}s^{h+1})}$. On the other hand, if $i \notin \mathcal{C}^{h+1}$, then $k \notin \mathcal{C}^{h+1}$, since otherwise $\gamma_k > \gamma_i$. However, if neither i nor k is in $\mathcal{C}^{h+1}$ then none of i, j, and k changes in going from s^h to s^{h+1}. Since j is a wire vertex, its excitation depends only on the value on vertex i. Consequently, the excitation of vertex j does not change in going from s^h to s^{h+1}. By Property 4 of the induction hypothesis, it therefore follows that $s_j^{h+1} = \overline{S_j(a{\cdot}s^{h+1})}$. □

From this result, we immediately obtain the following:

Corollary 7.2 *Let N and $\mathbf{N}$ be as in Theorem 7.1. Then, for $1 \leq j \leq m$, there exists a state $s^j \in \{0,1\}^m$ such that $bR_a^* s^j$ and*

$$lub\{b_j, s_j^j\} \sqsupseteq \mathbf{s}_j^A.$$

Proof: If $\mathbf{s}_j^A = b_j$, the result follows trivially. So assume $\mathbf{s}_j^A = \Phi$. If j is an input-delay or gate vertex, then the result follows immediately from Lemma 7.2, Property 2. So assume j is a wire vertex between vertices i and k, i.e., $(i,j) \in \mathcal{E}$ and $(j,k) \in \mathcal{E}$. If vertex j is unstable in total state $a{\cdot}b$, then we can reach a state in which $s_j = \overline{b_j}$. Hence, assume wire vertex j is stable in state $a{\cdot}b$. The excitation of wire vertex j is completely determined by the value on vertex i; thus $S_j(a{\cdot}s) = b_j$ for every $s \in \{0,1\}^m$ such that $s_i = b_i$. Assume vertex j changes to Φ at step r in Algorithm A. This implies that vertex i must have changed to Φ in step $r-1$, and thus $\mathbf{s}_i^A = \Phi$. By Lemma 7.2, Property 2, this implies that we can reach a state $s \in \{0,1\}^m$ such that $s_i = \overline{b_i}$. This means that $S_j(a{\cdot}s) = \overline{b_j}$; thus we can reach a state $\tilde{s}$ in which $\tilde{s}_j = \overline{b_j}$. □

Given the result $\mathbf{t}^B$ of Algorithm B if $\mathbf{t}_j^B = \Phi$, we say that vertex j is an *indefinite vertex*; otherwise it is a *definite vertex*. Note that every input-delay vertex is definite since we assume that the inputs to the circuit are always binary. Let $\mathcal{D}$ denote the set of definite vertices and $\mathcal{J}$ the set of indefinite vertices.

Assuming there is at least one indefinite vertex j (i.e., Algorithm B does not yield a binary result), there must be some other vertex $i \in \alpha^j$ that is also indefinite. Otherwise, all inputs to vertex j would be binary and its excitation could not be Φ. Since the network $\mathbf{N}$ is finite, we must

have at least one cycle of indefinite vertices; such a cycle will be called *indefinite*. Note that, since the network is complete—and thus every loop in the network is of length at least two—there must be at least one gate vertex and one wire vertex in every indefinite cycle.

Eventually we want to show that if the result of Algorithm B contains at least one Φ, there exists a nontransient cycle of length ≥ 2 (i.e., an oscillation) in the graph of the relation R_a for N such that all indefinite vertices "take part" in the oscillation, i.e., each vertex variable takes on both values 0 and 1 in the cycle. Furthermore, that cycle is reachable from the initial state of N.

The following definitions help to simplify the proofs. A total state $a{\cdot}c$ of N is *compatible* with $a{\cdot}\mathbf{t}^B$ if $c \sqsubseteq \mathbf{t}^B$. Also, a total state $a{\cdot}c$ of N is *definite-stable* if all the definite vertices are stable in that state. Finally, a total state $a{\cdot}c$ of N is *loop-unstable* if there is at least one unstable wire vertex in each indefinite cycle of N.

Lemma 7.3 *Let N and $\mathbf{N}$ be as in Theorem 7.1 and let $s^A \in \{0,1\}^m$ be a state derived as in the proof of Lemma 7.2. If $\mathbf{t}^h$ is the result of Algorithm* B *after h steps, $0 \leq h \leq B$, then there is a state $t^h \in \{0,1\}^m$ such that*

$$I.\ bR_a^* t^h, \quad II.\ t^h \sqsubseteq \mathbf{t}^h, \quad III.\ \text{if } \mathbf{t}_j^h = \Phi \text{ then } t_j^h = s_j^A$$

Proof: We proceed by induction on h. For the basis, let $t^0 = s^A$. Properties I–III follow trivially from the fact that $\mathbf{t}^0 = \mathbf{s}^A$, from Proposition 7.2, and from the assumption that s^A satisfies Properties 1–3 of Lemma 7.2.

Assume inductively that t^h has been constructed and let

$$t_j^{h+1} = \begin{cases} S_j(a{\cdot}t^h) & \text{if } \mathbf{t}_j^{h+1} \in \{0,1\}, \\ t_j^h & \text{otherwise.} \end{cases}$$

Clearly, $t^h R_a^* t^{h+1}$. Together with the induction hypothesis Property I, it follows that $bR_a^* t^{h+1}$. For Property II, consider any vertex j. If $\mathbf{t}_j^{h+1} = \Phi$, then it follows trivially that $t_j^{h+1} \sqsubseteq \mathbf{t}_j^{h+1}$. Hence assume that $\mathbf{t}_j^{h+1} \in \{0,1\}$. By definition of Algorithm B, $\mathbf{t}_j^{h+1} = \mathbf{S}_j(a{\cdot}\mathbf{t}^h)$. By the induction hypothesis Property II and the monotonicity of $\mathbf{S}$, it follows that $\mathbf{S}_j(a{\cdot}t^h) \sqsubseteq \mathbf{S}_j(a{\cdot}\mathbf{t}^h) = \mathbf{t}_j^{h+1}$. But $\mathbf{S}_j(a{\cdot}t^h) = S_j(a{\cdot}t^h)$, since the ternary extension $\mathbf{S}$ agrees with S on binary arguments. By construction, $\mathbf{t}_j^{h+1} \in \{0,1\}$ implies that $t_j^{h+1} = S_j(a{\cdot}t^h)$. Thus, it follows that $t_j^{h+1} \sqsubseteq \mathbf{t}_j^{h+1}$. Since j was arbitrary, Property II follows. Finally, by the monotonicity of Algorithm B (Proposition 7.4) if $\mathbf{t}_j^{h+1} = \Phi$, then $\mathbf{t}_j^h = \Phi$. However, by construction, if $\mathbf{t}_j^{h+1} = \Phi$, then $t_j^{h+1} = t_j^h$. This, together with the induction hypothesis Property III, implies that if $\mathbf{t}_j^{h+1} = \Phi$, then $t_j^{h+1} = t_j^h = s_j^A$ and Property III follows. Since Properties I–III hold for t^{h+1}, the induction goes through and the lemma follows. □

Lemma 7.4 *Let $t \in \{0,1\}^m$ be any state such that $t \sqsubseteq \mathbf{t}^B$. Then t is definite-stable.*

Proof: By Proposition 7.4, $\mathbf{t}^B = \mathbf{S}(a{\cdot}\mathbf{t}^B)$. Now consider any definite vertex j. By the definition of definite it follows that $\mathbf{t}_j^B \in \{0,1\}$ and thus $\mathbf{t}_j^B = \mathbf{S}_j(a{\cdot}\mathbf{t}^B) \in \{0,1\}$. However, by assumption, $t \sqsubseteq \mathbf{t}^B$, and by the monotonicity of $\mathbf{S}$, it follows that $\mathbf{S}_j(a{\cdot}t) \sqsubseteq \mathbf{S}_j(a{\cdot}\mathbf{t}^B) = \mathbf{t}_j^B \in \{0,1\}$ and therefore $\mathbf{S}_j(a{\cdot}t) = \mathbf{t}_j^B$. But $\mathbf{S}_j(a{\cdot}t) = S_j(a{\cdot}t)$, since the ternary extension $\mathbf{S}$ agrees with S on binary arguments. Altogether, if $\mathbf{t}_j^B \in \{0,1\}$ then $S_j(a{\cdot}t) = \mathbf{t}_j^B = t_j^B$, where the last equality follows from the fact that $t_j^B \sqsubseteq \mathbf{t}_j^B$. □

Corollary 7.3 *Let t^B be a state derived as in the proof of Lemma 7.3. Then t^B is definite-stable.*

Proof: The proof follows immediately from Lemma 7.4 and by Property II of Lemma 7.3. □

Lemma 7.5 *Let t^B be a state derived as in the proof of Lemma 7.3. Then t^B is loop-unstable.*

Proof: It is sufficient to prove the claim for each indefinite simple cycle, where a cycle is simple if it has no repeated vertices except for the first and the last vertex in the cycle. Let C be an arbitrary indefinite simple cycle in $\mathbf{N}$. Note that C only contains gate and wire vertices, since no input-delay vertex can be indefinite. A gate vertex i in C is said to be *terminating* if no other gate vertex in C becomes Φ in Algorithm A after vertex i. Clearly, there must be at least one terminating vertex in C. Assume vertex i is terminating in C and that it became Φ at step r of Algorithm A. Since i is in C, one of the wire vertices in β^i must be the successor vertex to i in C; assume this is vertex j. We now claim that j is unstable in t^B. Note first that since i and j are indefinite vertices, i.e., $\mathbf{t}_i^B = \mathbf{t}_j^B = \Phi$, by Property III of Lemma 7.3, we can conclude that $t_i^B = s_i^A$ and $t_j^B = s_j^A$. Furthermore, since j is a wire vertex, its excitation is completely determined by the value on gate vertex i. Thus, if j is unstable in s^A, then it is also unstable in t^B. Finally, since i is terminating, it follows that $\gamma_i \geq \gamma_k$ for every other gate vertex in C. In particular, if $j \in \alpha^k$ ($k = i$ is possible), then $\gamma_i \geq \gamma_k$. By Property 4 of Lemma 7.2 it follows that $s_j^A = \overline{S_j(a{\cdot}s^A)}$. □

The proof now proceeds as follows. Starting with a state $s \in \{0,1\}^m$ we first exhibit a sequence of states

$$s = s^0,\ s^1, \ldots, s^r$$

where r is the number of indefinite gate vertices, and, for $0 \leq k \leq r$, exactly k indefinite gate vertices in s^k have values complementary to those in s, and the other indefinite gate vertices are the same as in s. Note that we do not say anything about the indefinite wire vertices. For convenience, we will say that k indefinite vertices have been "marked" in this way. By repeating this

process of marking (i.e., complementing) all of the indefinite gate vertices, we show the existence of an oscillation involving all the indefinite gate vertices. We then show that every indefinite wire vertex also oscillates in the constructed cycle.

Lemma 7.6 *Let $\mathbf{t}^B$ be the result of Algorithm B and let $a{\cdot}s$ be any total state compatible with $a{\cdot}\mathbf{t}^B$, definite-stable, and loop-unstable. Assume that zero or more, but not all, indefinite gate vertices are marked. Assume also that every wire vertex between a marked and an unmarked indefinite gate vertex is unstable. Then there exists at least one unmarked indefinite gate vertex k, such that all indefinite wire vertices in α^k are unstable.*

Proof: Consider the directed graph $G = (\mathcal{V}', \mathcal{E}')$, where

$$\begin{aligned} \mathcal{V}' &= \{i \in \mathcal{S} \mid \mathbf{t}_i^B = \Phi\}, \text{ and} \\ \mathcal{E}' &= \{(i,j) \in \mathcal{E} \mid i \in \mathcal{G} \text{ or } s_i = S_i(a{\cdot}s)\} \cap (\mathcal{V}' \times \mathcal{V}'). \end{aligned}$$

G can be obtained from the network graph by retaining only the indefinite vertices and those edges between indefinite vertices that either leave a gate vertex or leave a wire vertex that is stable in $a{\cdot}s$. G has two important properties:

i. there is no path from a marked vertex to an unmarked vertex, and

ii. there is no cycle in G.

Both properties follow from the construction of G and the assumptions in the lemma.

Now consider a reverse path in G. Start at some unmarked gate vertex $k \in \mathcal{V}'$ and traverse G backward. From (ii) and the fact that G is finite, it follows that a reverse path in G started at vertex k must stop at some vertex, say j. Note that j must be a gate vertex, and, by (i), must be unmarked. Furthermore, since each indefinite gate vertex has at least one indefinite wire vertex in its fan-in set, it follows that all indefinite wire vertices in α^j must be unstable; otherwise the reverse path would not have stopped at j. □

Lemma 7.7 *Let $\mathbf{t}^B$ be the result of Algorithm B and let $a{\cdot}s$ be any total state compatible with $a{\cdot}\mathbf{t}^B$, definite-stable, and loop-unstable. If, for some gate vertex j, all indefinite vertices in α^j are unstable, then there exists a state $\dot{s}$ reachable from s, compatible with $\mathbf{t}^B$, definite-stable and loop-unstable,such that*

i. $\dot{s}_j = \overline{s_j}$, and

ii. all indefinite wire vertices in β^j are unstable in $\dot{s}$.

Proof: We construct $\dot{s}$ in a two-step process. First we show that there is a state $\tilde{s}$ reachable from s such that $\tilde{s}_j = \overline{S_j(a{\cdot}\tilde{s})}$. We then show how to reach $\dot{s}$ from $\tilde{s}$.

For every input and gate vertex k, let $\tilde{s}_k = s_k$. If $k \neq j$, let $\alpha^k(\tilde{s}) = \alpha^k(s)$. Let $\alpha^j(\tilde{s}) = c^j$, where $c^j \in \{0,1\}^{d_j}$ is such that $c^j \sqsubseteq \alpha^j(\mathbf{t}^B)$ and $S_j(a{\cdot}c) = \overline{b_j}$ for all $c \in \{0,1\}^m$ such that $\alpha^j(c) = c^j$. We claim that such c^j is guaranteed to exist. Suppose it did not, i.e., assume that, for all $c^j \in \{0,1\}^{d_j}$ such that $c^j \sqsubseteq \alpha^j(\mathbf{t}^B)$, there is some state $w \in \{0,1\}^m$ such that $c^j = \alpha^j(w)$ and $S_j(a{\cdot}w) = w_j$. Since S_j depends only on the vertices in α^j, we can conclude that $S_j(a{\cdot}w) = w_j$ implies $S_j(a{\cdot}w') = w_j$ for every $w' \in \{0,1\}^m$ such that $\alpha^j(w) = \alpha^j(w')$. Altogether, we have that $S_j(a{\cdot}w) = w_j$ for every $w \in \{0,1\}^m$ such that $\alpha^j(w) \sqsubseteq \alpha^j(\mathbf{t}^B)$. By the definition of ternary extension, this implies that $\mathbf{S}_j(\mathbf{t}^B) = w_j$. But, by Lemma 7.4, $\mathbf{S}(\mathbf{t}^B) = \mathbf{t}^B$; thus $\mathbf{t}^B_j = w_j \in \{0,1\}$. This contradicts the assumption that j is an indefinite gate vertex. Hence, our claim that such c^j exists is true.

It remains to be shown that $sR_a^*\tilde{s}$. This follows from the fact that

$$c^j \sqsubseteq \alpha^j(\mathbf{t}^B),$$

the fact that all indefinite vertices in α^j are unstable, and the fact that s is compatible with $\mathbf{t}^B$.

We now are ready to construct $\dot{s}$. For every input and gate vertex i let

$$\dot{s}_i = \begin{cases} S_j(a{\cdot}\tilde{s}) & \text{if } i = j, \\ \tilde{s}_i & \text{otherwise,} \end{cases}$$

and

$$\beta^i(\dot{s}) = \begin{cases} \beta^j(S(a{\cdot}\tilde{s})) & \text{if } i = j, \\ \beta^i(\tilde{s}) & \text{otherwise.} \end{cases}$$

Clearly, $\tilde{s}R_a^*\dot{s}$ and thus $sR_a^*\dot{s}$. By construction, $\tilde{s}_j = S_j(a{\cdot}\tilde{s}) = \overline{s_j}$, and Property (i) holds. On the other hand, the construction of $\tilde{s}$ ensures that each wire vertex in the fan-out set of gate i is unstable, since we simultaneously set its output to its current excitation and change its input. Thus Property (ii) follows. If gate vertex j is indefinite, then each wire vertex in β^j is also indefinite. Consequently, it is straightforward to verify that $\tilde{s}$ is definite-stable, loop-unstable, and compatible with $\mathbf{t}^B$. □

Lemma 7.8 *Let $\mathbf{t}^B$ be the result of Algorithm B and let $a{\cdot}s$ be any total state compatible with $a{\cdot}\mathbf{t}^B$, definite-stable, and loop-unstable. Assume there are r indefinite gate vertices. Then, for each k, $0 \leq k \leq r$, there is a state $s^k \in \{0,1\}^m$ with k vertices marked such that $sR_a^*s^k$ and $a{\cdot}s^k$ is compatible with $a{\cdot}\mathbf{t}^B$, definite-stable, loop-unstable, and every wire vertex between a marked and an unmarked indefinite gate vertex is unstable.*

Proof: We proceed by induction on the number of indefinite gate vertices that have been marked, i.e., complemented. For the basis, $k = 0$, let $s^0 = s$ and the claim follows trivially. Now assume inductively that the claim holds for $k \geq 0$. By Lemma 7.6, it follows that there exists an unmarked indefinite gate vertex j such that all indefinite gate vertices in α^j are unstable in $a{\cdot}s^k$. But Lemma 7.7 guarantees the existence of a state s^{k+1}, such that $s^k R_a^* s^{k+1}$ and $a{\cdot}s^{k+1}$ is compatible with $a{\cdot}\mathbf{t}^B$, definite-stable, and loop-unstable. Furthermore, gate vertex j is complemented, and all the indefinite wire vertices in β^j are unstable in $a{\cdot}s^{k+1}$. Now mark vertex j, and note that all indefinite wire vertices between marked and unmarked indefinite gate vertices are still unstable. Hence, the induction step goes through and the lemma follows. □

Corollary 7.4 *Let $\mathbf{t}^B$ be the result of Algorithm B and let $a{\cdot}s$ be any total state compatible with $a{\cdot}\mathbf{t}^B$, definite-stable, and loop-unstable. Then there is a state $\tilde{s}$ reachable from s and such that $a{\cdot}\tilde{s}$ is compatible with $a{\cdot}\mathbf{t}^B$, definite-stable, loop-unstable, and all indefinite gate vertices have complementary values in s and $\tilde{s}$.*

Proof: This follows immediately from Lemma 7.8 for k equal to the number of indefinite gate vertices. □

We are now ready to state and prove the main result of this section:

Lemma 7.9 *Let N and $\mathbf{N}$ be as in Theorem 7.2. Then there exists a nontransient cycle Z that is reachable from the initial state b such that*

$$lub\{s \mid s \in Z\} \sqsupseteq \mathbf{t}^B.$$

Proof: By Lemmas 7.3 and 7.5, and Corollary 7.3, it follows that $a{\cdot}t^B$ is compatible with $a{\cdot}\mathbf{t}^B$, definite-stable, and loop-unstable. Hence, Corollary 7.4 can be applied. Since Corollary 7.4 can be applied any number of times and there is only a finite number of possible states, there must exist a cycle in the R_a graph. By the construction of Corollary 7.4 it follows that each indefinite gate vertex oscillates. By the construction in Lemmas 7.7 and 7.8, it is also easy to see that every indefinite wire vertex in the fan-out sets of the indefinite gate vertices also oscillates. However, a wire vertex j in the fan-out set β^i of some gate vertex i is indefinite if and only if gate vertex j is indefinite. Hence, every indefinite vertex is oscillating. Since all definite vertices are stable, it follows that the cycle is nontransient. □

Chapter 8

Bi-Bounded Delay Models

The race models discussed so far correspond to delays that are only bounded from above; consequently, the ratio between two delays can be arbitrarily large. The use of such models often leads to very conservative results. For example, suppose a GMW analysis indicates a possible timing problem, say a critical race. It may be the case that, unless the delay in one particular gate is greater than the sum of the delays in a large number of other gates, this critical race will always be resolved in the same way. To obtain more realistic results, we must place some constraints on the relative values of gate and wire delays.

One possible assumption would be that each component (gate and wire) has a fixed nominal delay value, but that value may differ from component to component; this is, in fact, what many commercial simulators do. Unfortunately, in such an approach small perturbations in the delay values can change the predicted behavior of the network. In other words, models using nominal delays have no ability to detect any race or hazard phenomena; hence, they are not very suitable for analyzing the behavior of asynchronous networks.

To overcome the above-mentioned deficiencies, we need models in which the delays are not completely known and may, in fact, vary in time, but cannot take on arbitrarily small values. In this chapter, we develop models in which the delay magnitudes lie between lower and upper bounds, and we present algorithms for analyzing networks with the aid of such models. All the delays in this chapter are inertial.

How To Read This Chapter

Section 8.1 contains several examples and some constructive proofs that are not difficult to follow. Sections 8.2—8.4 deal with continuous models and can be omitted on first reading. In Section 8.5 we return to discrete models, but in the ternary domain. Since there are no proofs in this section, it should be relatively accessible.

8.1 Discrete Binary Models

In this section we study several discrete models. To simplify the discussion, we associate delays with gates only; it is trivial to generalize the results to networks containing also wire delays.

The simplest approach to bi-bounded delay analysis is to use a "slow" mode—in which all the gates have their maximal delays—and a "fast" mode, with minimal delays. It is clear that such a model suffers from the same problem as the nominal-delay model, i.e., that a small perturbation of one of the delays can completely change the behavior of the network. There is an even more subtle problem, however. Intuitively, it sounds plausible that the transition time predicted by a simulator operated in "slow" mode should be an upper bound on the time the network would actually take for this transition. Unfortunately, this is not the case. We have already seen this in the performance estimation example of Chapter 1. Below we provide another simple example of this problem.

A problem with the slow mode

Consider the network $C_{8.1}$ of Figure 8.1 started in stable total state $X{\cdot}y = 0{\cdot}1001$ when the input changes to 1 and the delay in each gate is between 1 and 3 time units. It is easy to see that only gates 1 and 2 will change, if we assume that all the gates have a delay of exactly 3 time units. Thus, according to such an analysis, the maximum time it takes for this network to stabilize (by reaching state $y = 0101$) is 3 time units. However, if gates $1, 2, 3$, and 4 have delays of $3, 1, 2$ and 2 time units, respectively, the network will actually take 7 time units to reach the same state, as is easily verified.

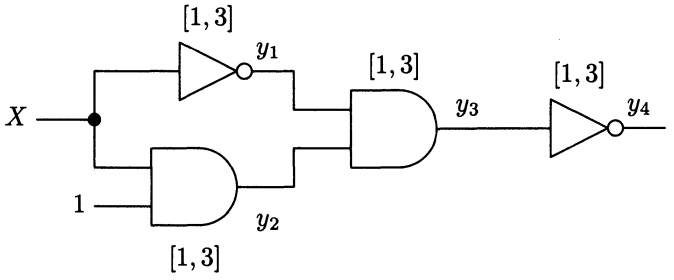


FIGURE 8.1. Network $C_{8.1}$.

The question now arises whether it suffices to analyze a network for all possible combinations of lower and upper bounds on the delays. Again, it turns out that the answer is negative.

A problem with all slow-fast combinations

Consider gate network $C_{8.2}$ of Figure 8.2. Assume that it is started in the stable state $X \cdot y = 0 \cdot 0111100$ and that the input changes to $X = 1$. Furthermore, suppose the delays are bounded as shown in Figure 8.2. If we only consider "extreme-case" delays, i.e., 1 or 5, it can be verified that the only possible nontransient states reachable are 1011000, 1010000 and 1011001, i.e., at least one of y_4 and y_7 does not change. If we also allow the delays to be anywhere between the bounds, however, the network can also reach 1010001—a state in which both y_4 and y_7 have changed. This can happen if, for example, $\delta_1 = 3$, $\delta_5 = 5$, and the remaining delays are equal to 1. In fact, one can verify that this state is reachable only if $2 \leq \delta_1 \leq 4$.

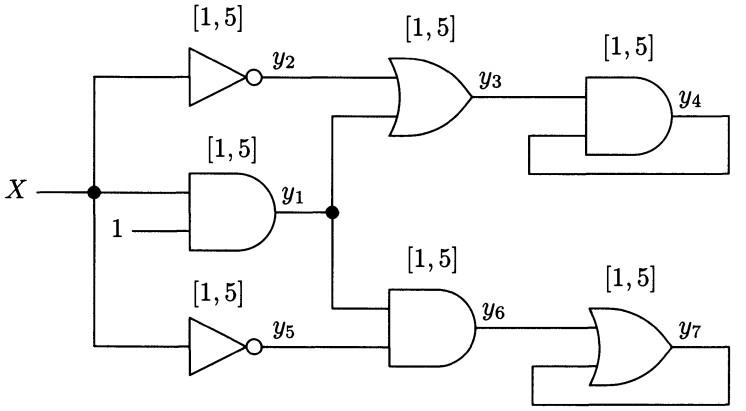


FIGURE 8.2. Network $C_{8.2}$.

Can a correct analysis be carried out by considering only all the integer delay values between the minimum and maximum delays of each component? Unfortunately, even such an elaborate analysis is not always correct.

A problem with fixed discrete resolution

Consider network $C_{8.3}$ of Figure 8.3 started in the stable state $X = 0$, $y = 11100000$ with X changing to 1. Assume the delays are bounded as shown in the figure. If only integer values are used for the delays, the single nontransient state reachable by the network is $y = 00000000$, i.e., the "OR-latch" (gate 8) does not change. This can be seen as follows. If y_8 is to change, it must be unstable for one time unit. Hence, y_7 must change to 1 at some time. For this to happen, y_4 and y_6 must be 0 and y_5 must be 1. To change gate y_5, gates y_1 and y_3 must have different values for at least one time unit. Since $1 \leq \delta_1(t) \leq 2$ and $1 \leq \delta_3(t) \leq 2$, however, this implies that one of

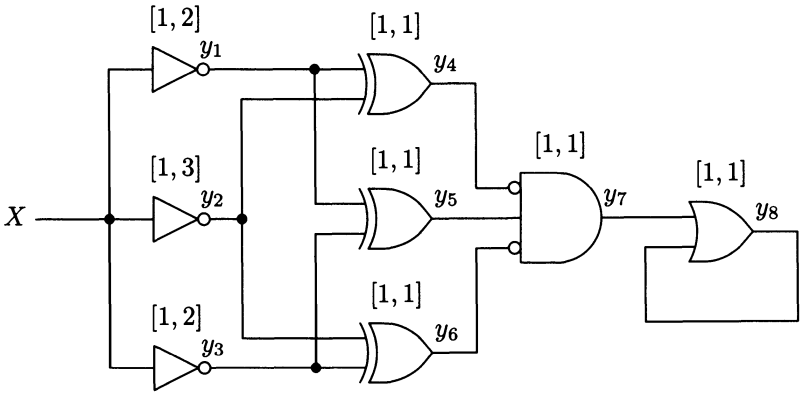


FIGURE 8.3. Network $C_{8.3}$.

y_1 and y_3 must change at time 1 and the other at time 2. Since the network is completely symmetric, we may assume that y_1 changes at time 1 and y_3 at time 2. To propagate the change in y_5 to the OR latch, we must choose the delay of y_2 in such a way that neither y_4 nor y_6 changes at the same time as y_5. This is impossible using only discrete delay values, since $\delta_2 = 1$ will cause y_6 to change at the same time as y_5, and $\delta_2 = 2$ or 3 will cause y_4 to change at the same time as y_5. Hence, the OR-latch will remain 0. On the other hand, if we remove the discreteness restriction, the network can reach a state where the OR-latch has the value 1. For example, let $\delta_1 = 1$, $\delta_2 = 1.5$, and $\delta_3 = 2$. In this case, gates y_4 and y_6 will be unstable for only 0.5 time units—less than their minimum delays—and thus will not change.

In the example above, we have used only integer values for delays. Would a finer (discrete) resolution help? We show below that, no matter how fine the resolution, a discrete model can never be as accurate as the continuous one.

A problem with variable discrete resolution

Consider a binary network $N = \langle \{0,1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, F \rangle$. Let the delay bounds for variable s_i be the nonnegative integers d_i and D_i. Let $out_k(R_a(b))$ denote the outcome of a race analysis when the network is started in the state b, the input changes to a, and the delay in any variable s_i is allowed to take on only values from the set $\{d_i, d_i + \frac{1}{k}, d_i + \frac{2}{k}, \ldots, D_i - \frac{1}{k}\}$. Similarly, let $out(R_a(b))$ denote the outcome when the delays can take on *any* value in the interval $[d_i, D_i)$. (As usual, $[d, D]$ denotes a closed interval, $(d, D]$ and $[d, D)$ denote half-open intervals, and (d, D) denotes an open interval.) The next theorem shows that this kind of resolution can also fail.

Theorem 8.1 *For any constant integer $k \geq 1$, $out_k(R_a(b)) \subseteq out(R_a(b))$. Furthermore, for some networks the inclusion is proper.*

Proof: The first part of the theorem is trivial. The example above shows that the inclusion can be proper when $k = 1$. Hence, we focus on proving the claim for $k > 1$. We prove this by constructing a network with an input transition such that a certain final state can be reached only if at least one component of the network has a delay strictly greater than 1 and strictly less than $1\frac{1}{k}$. In fact, we will construct a network such that $out_k(R_a(b)) \subset out_{2k}(R_a(b))$.

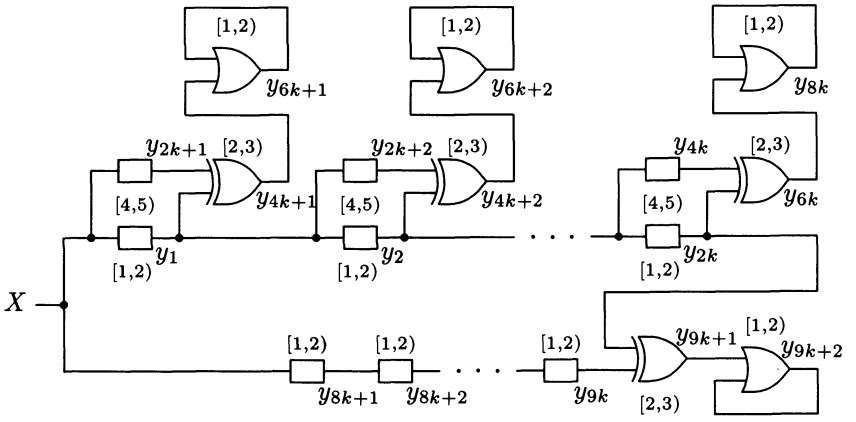


FIGURE 8.4. Network construction for Theorem 8.1.

The construction is shown in Figure 8.4. It consists of two chains of delays, of length $2k$ and k, respectively. The delays of the delay elements in the chains are all assumed to be in the interval $[1, 2)$. The input and output of each delay element in the longer chain are also used by a small network consisting of a larger delay, a XOR gate, and an "OR latch." Assume now that the network is started in state $X{\cdot}y = X{\cdot}y_1 \ldots y_{9k+2} = 0{\cdot}0 \ldots 0$, and that the input changes to 1. Can all the OR gates be 0 after this change has occurred and the network has reached a stable state?

Note that, if any OR gate changes to 1 during the transition, it cannot change back to 0 later. Note also that, if a XOR gate feeding an OR gate changes to 1 at any time, the OR gate must also change to 1, since the minimum delay of the XOR gate is greater than the maximum delay of the OR gate. Now consider subnetwork i on the longer delay chain. To avoid changing, the XOR gate y_{4k+i} must be unstable for strictly less than 3 time units. Since the delay of y_i is strictly smaller than that of y_{2k+i}—and the delay of y_{2k+i} is at least 4 time units—it follows that the delay of y_i must be strictly greater than 1 for y_{4k+i} not to change.

First, we analyze this network using a discrete model with resolution $\frac{1}{k}$. Since the delay of y_i must be strictly greater than 1, it follows that it must

belong to the set $\{1\frac{1}{k}, 1\frac{2}{k}, \ldots, 2-\frac{1}{k}\}$. If all the OR gates in the longer delay chain remain 0, the total delay through the longer delay chain must be greater than or equal to $2k(1\frac{1}{k}) = 2k+2$. Now consider the shorter delay chain. The maximum total delay through this chain must be less than or equal to $k(2-\frac{1}{k}) = 2k-1$. Thus, if all the OR gates in the longer delay chain are to remain 0, the XOR gate y_{9k+1} must be unstable for at least $2k+2-(2k-1) = 3$ time units and must therefore change. In summary, no state in $out_k(R_a(b))$ has $y_{6k+1} = y_{6k+2} = \ldots = y_{8k} = y_{9k+2} = 0$.

TABLE 8.1. Possible delay assignment for $\frac{1}{2k}$ resolution.

y_i	for $1 \le i \le 2k$	$1+\frac{1}{k}$
y_{2k+i}	for $1 \le i \le 2k$	4
y_{4k+i}	for $1 \le i \le 2k$	$3-\frac{1}{2k}$
y_{6k+i}	for $1 \le i \le 2k$	1
y_{8k+i}	for $1 \le i \le k$	$2-\frac{1}{2k}$
y_{9k+1}		$3-\frac{1}{2k}$
y_{9k+2}		1

On the other hand, if we assume the resolution is $\frac{1}{2k}$, we would have to consider—among many other delay assignments—the case where the delays in the different components are as listed in Table 8.1. Since here the delay of y_i is $1\frac{1}{k}$, that of y_{2k+i} is 4, and that of XOR gate y_{4k+i} is $3-\frac{1}{2k}$, it follows that the XOR gate will be unstable for a time shorter than its delay and will not change. Similarly, the difference between the total delays through the longer and the shorter delay chains will be $2k(1\frac{1}{k}) - k(2-\frac{1}{2k}) = 2+\frac{1}{2} < (3-\frac{1}{2k})$ for $k > 1$. Hence, XOR gate y_{9k+1} will not change. Consequently, there is a state in $out_{2k}(R_a(b))$ in which $y_{6k+1} = y_{6k+2} = \ldots = y_{8k} = y_{9k+2} = 0$. Since there was no such state in $out_k(R_a(b))$, the proper inclusion has been established and the claim follows. □

The result above shows that no fixed resolution is sufficient for every circuit. To the best of our knowledge it is still not known whether a discrete model with resolution $f(n+m)$, for some function f, is guaranteed to be correct for every network with n inputs and m state variables. We conjecture that this is indeed the case. It is easy to generalize the proof above, however, to show that f must be at least singly exponential in m. Hence, the existence of such a resolution function is more of theoretical than practical interest.

8.2 Continuous Binary Model

As was shown in the previous section, it appears necessary to treat time as a continuous quantity, if an accurate bi-bounded delay analysis is desired. The model and algorithm described in this section are based on the work

in [45] and [84], which uses an analysis with binary delays. In the next section we describe another continuous algorithm, but one that assumes the delays are of the extended ternary type.

In the GMW model, the current state of input-excitation and internal-state variables is sufficient to determine the set of possible successor states of the network. Clearly this is not the case in a bi-bounded delay analysis. For example, if two gates with nonoverlapping delay bounds, say $[1,2)$ and $[3,5)$, become unstable at the same time, the gate with the smaller delay must change first. Consequently, we need to record a certain amount of previous excitation history for each variable. In fact, we must remember either how long an unstable variable has been unstable or how much longer the variable can be unstable without changing. In this section we use the latter approach and introduce a *time-left* variable. When a variable becomes unstable because of an input or state change, the time-left variable can be set to any time consistent with the delay bounds. We use the convention that stable variables have an infinite amount of time left. More formally, assume we have a binary network $N = \langle \{0,1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, F \rangle$, started in stable total state $\hat{a}\cdot b$, and wish to study the transition caused by changing the input to a at time 0. Assume also that the delay $\delta_j(t)$ of variable j, $1 \leq j \leq m$, satisfies $d_j \leq \delta_j(t) < D_j$, for some nonnegative integers d_j and D_j. A *race state* is a pair $\langle c, r \rangle$, where c is the current state of the network, i.e., $c \in \{0,1\}^m$, and r is a vector of size m such that $r_i \in (0, D_i)$ for unstable variables and $r_i = \infty$ for stable variables. Intuitively, the variables in r serve as *alarm clocks* for the unstable variables. When an alarm clock goes off (expires), the corresponding variable must change. If a variable becomes stable because of some other change, the corresponding alarm clock is turned off, i.e., set to ∞. This takes care of the inertial nature of the delays.

Let $\mathcal{Q}$ denote the (infinite) set of race states that are reachable according to the bi-bounded delay race model, and let R_a denote a binary relation on $\mathcal{Q}$ defining possible successor states. Furthermore, let $\mathcal{J} \subseteq \mathcal{Q}$ be the set of initial race states of network N when it is started in stable total state $\hat{a}\cdot b$ and the input changes to a at time 0. The choice of starting state is very flexible. In fact, the only requirements are that 1) the state of the network is b, 2) stable variables have their alarm clocks set to ∞, and 3) unstable variables have their alarm clocks set to values that are consistent with the minimum and maximum delay values of their variables. Now, given a race state $\langle c, r \rangle$, we determine a possible successor state as follows: If $a\cdot c$ is a stable total state, the network remains in this state indefinitely; thus a stable state has only itself as successor. Otherwise, a possible successor state is obtained by waiting until the first alarm clock goes off, i.e., waiting for $r_{\min} = \min\{r_i \mid 1 \leq i \leq m\}$ time units. When this occurs, the corresponding variables are complemented. (Note that, if several alarm clocks have the same value, all of the corresponding variables change at the same time.) We then update the alarm clocks. Three cases

are possible: First, if a variable is stable in the new network state, its alarm clock is turned off, i.e., r_i is set to ∞. Second, if a variable has been unstable for some time, but the alarm clock did not expire in this transition (i.e., $r_i > r_{\min}$), then the time left on the alarm clock is simply decreased by $r_{\min}$. Finally, if neither of these two situations applies, the variable starts a new *race unit*; its alarm clock is set to any value consistent with its delay bounds. Note that this case includes both the situation when a stable variable becoms unstable as well as the situation when an unstable variable changes but remains unstable after the change.

As in Chapter 6, let $\mathcal{U}(a{\cdot}c)$ denote the set of variables that are unstable in the total state $a{\cdot}c$, i.e., $\mathcal{U}(a{\cdot}c) = \{s_j \mid S_j(a{\cdot}c) \neq c_j\}$.

Formally, $\mathcal{J}$, $\mathcal{Q}$, and R_a are defined inductively as follows.

Basis: $\mathcal{J} = \mathcal{Q}$ is defined to be the set of all acceptable starting race states, i.e., race states $\langle b, r\rangle$ such that, for each $i, 1 \leq i \leq m$, either $s_i \in \mathcal{U}(a{\cdot}b)$ and $d_i \leq r_i < D_i$, or s_i is stable and $r_i = \infty$.

Induction step: Given $\langle c, r\rangle \in \mathcal{Q}$:

1. If $\mathcal{U}(a{\cdot}c) = \emptyset$, then $\langle c, r\rangle R_a \langle c, r\rangle$.
2. If $\mathcal{U}(a{\cdot}c) \neq \emptyset$, then select any $\mathcal{P} \subseteq \mathcal{U}(a{\cdot}c)$, $\mathcal{P} \neq \emptyset$, and any real number $r_{\min}$ such that $r_i > r_{\min}$ for every $s_i \in \mathcal{U}(a{\cdot}c) - \mathcal{P}$ and $r_i = r_{\min}$ for every $s_i \in \mathcal{P}$. Let $\langle c', r'\rangle \in \mathcal{Q}$, and $\langle c, r\rangle R_a \langle c', r'\rangle$, where

$$c_i' = \begin{cases} S_i(a{\cdot}c) & \text{if } s_i \in \mathcal{P}, \\ c_i & \text{otherwise,} \end{cases}$$

and

$$r_i' = \begin{cases} \infty & \text{if } s_i \notin \mathcal{U}(a{\cdot}c'), \\ r_i - r_{\min} & \text{if } s_i \in \mathcal{U}(a{\cdot}c') \text{ and } r_i > r_{\min}, \\ \tilde{r}_i & \text{otherwise,} \end{cases}$$

for any choice of $\tilde{r}_i$ such that $d_i \leq \tilde{r}_i < D_i$.

Example 1

To illustrate the definition above, consider network $C_{8.5}$ of Figure 8.5. Assume that it is started in stable total state 0·11 and the input

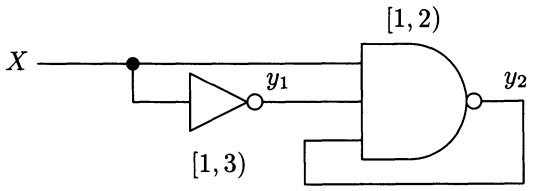


FIGURE 8.5. Network $C_{8.5}$.

changes to 1 at time 0. Two possible race sequences are shown in Figure 8.6. The numbers to the right of the arrows are the different values of $r_{\min}$.

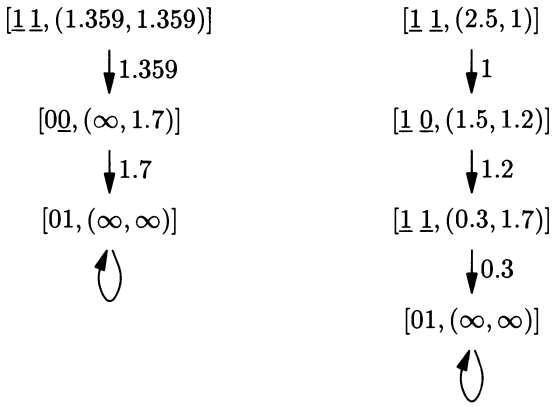


FIGURE 8.6. Two possible race sequences.

Let g denote a sequence $\langle s^0, r^0\rangle$, $\langle s^1, r^1\rangle, \ldots$ of race states such that $\langle s^0, r^0\rangle \in \mathcal{J}$ and $\langle s^k, r^k\rangle R_a \langle s^{k+1}, r^{k+1}\rangle$, for $k \geq 0$. To define the state of the network at a particular time according to such a sequence, we need to know when the sequence changes from one state to the next. We can accomplish this by associating a time sequence with each race sequence; this is simply a sequence t^0, $t^1, \ldots$ of real numbers defined inductively as follows. Let $t^0 = 0$. Assuming that t^h has been defined, then $t^{h+1} = t^h$, if $\mathcal{U}(a{\cdot}s^h) = \emptyset$, and $t^{h+1} = t^h + \min_{1 \leq i \leq m} r_i^h$, otherwise.

Let $\mathcal{G}$ be the set of all race sequences for network N when it is started in stable total state $\hat{a}{\cdot}b$ and the input changes to a at time 0. For $g \in \mathcal{G}$ with corresponding time sequence t^0, $t^1, \ldots$, the state of the network at time t according to this race sequence is written as $s^g(t)$ and is equal to

$$s^g(t) = \begin{cases} b & \text{for } t < 0, \\ s^h & \text{for } t^h \leq t < t^{h+1} \text{ if } \mathcal{U}(a{\cdot}s^h) \neq \emptyset, \\ s^h & \text{for } t^h \leq t \text{ if } \mathcal{U}(a{\cdot}s^h) = \emptyset. \end{cases}$$

The set of possible states of the network at time t is called $Reach(t)$ and is defined as

$$Reach(t) = \{s \mid s = s^g(t) \text{ for some } g \in \mathcal{G}\}.$$

In the example above, it is easy to verify that $Reach(t) = \{11\}$ for $0 \leq t < 1$, that $Reach(t) = \{00, 01, 10, 11\}$ for $1 \leq t < 2$. It is quite laborious, however, to compute $Reach(t)$ for $t \geq 2$. We will return shortly to the question of computing $Reach(t)$ efficiently. Before doing so, we show that

the bi-bounded race model captures exactly the behavior of a network consisting of ideal delay-free excitation functions connected in series with bi-bounded inertial delays, as defined in Section 3.4. We do this by proving two rather technical lemmas. The first lemma shows that the behavior of a network according to a bi-bounded delay race sequence is consistent with the bi-bounded inertial delay model. More precisely:

Lemma 8.1 *Let* $g = \langle s^0, r^0 \rangle, \langle s^1, r^1 \rangle, \ldots$ *be an arbitrary bi-bounded delay race sequence for network* N*, when it is started in the stable total state* $\hat{a}{\cdot}b$ *and the input changes to* a *at time* 0*. Furthermore, let*

$$X(t) = \begin{cases} \hat{a} & \text{for } t < 0, \\ a & \text{for } t \geq 0. \end{cases}$$

Then the input/output behavior of every variable s_j *is consistent with the bi-bounded inertial delay model, i.e.,* $S_j(X(t), s^g(t))/s_j^g(t)$ *is an acceptable input/output waveform according to the bi-bounded inertial delay model.*

Proof: The proof is straightforward but somewhat tedious and is left as an exercise for the interested reader. □

The converse result is stated in Lemma 2. Let $X(t)$ and $s(t)$ denote the input and state of network N at time t.

Lemma 8.2 *Assume that* $S_j(X(t), s(t))/s_j(t)$*, for* $1 \leq j \leq m$*, is an acceptable input/output waveform according to the bi-bounded inertial delay model, when* N *is started in stable total state* $\hat{a}{\cdot}b$ *and the input changes to* a *at time* 0*. Then we can construct a valid bi-bounded delay race sequence corresponding to this input/output waveform.*

Proof: Again, the proof is straightforward but tedious and is left as an exercise for the interested reader. □

Together, the two lemmas above establish the desired connection between the bi-bounded inertial delay model and the behavior of a network according to the bi-bounded delay race model. Unfortunately, the bi-bounded delay race model is not very useful because of its continuous nature.

8.3 Algorithms for Continuous Binary Analysis

In the analysis of the previous section, there may be infinitely many starting states and, given a race state, there may be infinitely many successor states. Since these states may differ only in their r-components, the question arises whether it is possible to deal with equivalence classes of race states instead. The main result of [45] and [84] is that this is indeed possible.

Before we can describe the algorithm, we need some properties of intervals. Let τ denote a time interval. Let $\lfloor \tau \rfloor$ ($\lceil \tau \rceil$) denote the lower (upper) bound on the interval τ. The intersection of two intervals and the addition

of two intervals are defined in the obvious way. The subtraction of an interval τ is defined as addition of $-\tau$, where $-\tau$ is defined by $\lfloor -\tau \rfloor = -\lceil \tau \rceil$, $\lceil -\tau \rceil = -\lfloor \tau \rfloor$, and each end of $-\tau$ is closed or open according to whether the opposite end of τ is closed or open. The main property of intervals that should be emphasized is that, although a nonempty interval represents an infinite number of possible values, if the bounds on the intervals are integers, all the operations above can be performed very efficiently. Furthermore, the operations yield intervals with integer bounds.

It is tempting to believe that intervals could be used to bound the possible values of the alarm clocks of the unstable variables, and thus that we could remove the continuous parts of the bi-bounded delay race model. Unfortunately, this is not as easy as it may first appear. The problem can be illustrated by a simple example. Consider a race state in which three variables, say s_1, s_2, and s_3, are unstable and the corresponding alarm clocks, r_1, r_2, and r_3, are bounded by $[1,2)$, $[7,8)$, and $[7,8)$, respectively. Assume that variable s_1 changes first; this can happen any time in the interval $[1,2)$. The question now arises: How much time remains on the alarm clocks r_2 and r_3 after variable s_1 has changed? If we simply subtract the interval $[1,2)$ we would get the result $[5,7)$ and $[5,7)$, respectively. It is true that the time remaining on alarm clocks r_2 and r_3 can be as small as 5 and can be almost 7. It is not possible, however, for the time remaining on alarm clock r_2 to be 5, while the time remaining on alarm clock r_3 is greater than 6. What we have lost is the relation between the times left on alarm clocks r_2 and r_3. Thus, we must also keep track of bounds on the differences between the time remaining on pairs of alarm clocks. One of the results of [45] and [84] is that it is sufficient to keep bounds on the individual alarm clocks and the differences between all pairs of alarm clocks in order to carry out an efficient and accurate bi-bounded delay analysis.

Let us now return to defining a more efficient bi-bounded delay analysis. The basic idea is to associate with each state a convex linear *region* describing the possible values of the alarm clocks, rather than one specific instance of these clocks. Consequently, define a *bd-state* to be the pair $\langle c, \Sigma \rangle$, where c is the current state of the network and Σ is an $m \times (m+1)$ lower triangular matrix of intervals. For $1 \leq j < i \leq m$ the interval σ_{ij} bounds the difference between the times r_i and r_j remaining on alarm clocks i and j when both s_i and s_j are unstable variables. For $1 \leq i \leq m$ the interval σ_{i0} bounds the time remaining on alarm clock i. The *feasible region* for such matrix is

$$\mathcal{S}(\Sigma) = \{\langle r_1, \ldots, r_m \rangle \mid r_i - r_j \in \sigma_{ij} \text{ for } 0 \leq j < i \leq m \text{ and } r_0 = 0\}.$$

If $\mathcal{S}(\Sigma) \neq \emptyset$ the matrix is said to be a *feasible matrix*; otherwise it is an *infeasible matrix.*

There are two difficulties with using such a lower triangular matrix of intervals to represent a feasible region: First, there are, in general, many different matrices representing the same region. Hence, comparing regions

can be difficult. Second, it is often difficult to determine whether a matrix is feasible or not. One of the keystones in the analysis procedure we will describe shortly is Algorithm 1 of Table 8.2.

TABLE 8.2. Algorithm 1.

for $k = 0$ **to** m **such that** $k = 0$ **or** $s_k \in \mathcal{U}(a \cdot c)$
  for $i = 1$ **to** m **such that** $i \neq k$ **and** $s_i \in \mathcal{U}(a \cdot c)$
    for $j = 0$ **to** $i - 1$ **such that** $j \neq k$ **and** $j = 0$ **or** $s_j \in \mathcal{U}(a \cdot c)$

$$\sigma_{ij} = \begin{cases} \sigma_{ij} \cap (\sigma_{ik} + \sigma_{kj}) & \text{if } j < k < i, \\ \sigma_{ij} \cap (\sigma_{ik} - \sigma_{jk}) & \text{if } k < j, \\ \sigma_{ij} \cap (\sigma_{kj} - \sigma_{ki}) & \text{if } i < k. \end{cases}$$

All other σ_{ij} **remain the same.**

Algorithm 1 is an adaptation of the Floyd-Warshall all-pairs shortest-path algorithm for directed graphs. The algorithm can be used to derive a canonical matrix for every feasible region, as the following lemma illustrates:

Lemma 8.3 *Given a state c of a network and a lower triangular matrix Σ of intervals, the result of Algorithm 1, denoted $\theta(c, \Sigma)$, is a lower triangular matrix of intervals satisfying the following two properties: First, if Σ is feasible, then $\theta(c, \Sigma)$ is the unique lower triangular matrix for the feasible region for Σ. Second, if Σ is infeasible, then at least one of the intervals of $\theta(c, \Sigma)$ is empty.*

Proof: We refer the reader to [84] for the proof of this result. □

It should be pointed out that we are normally interested only in σ_{ij} entries for which both s_i and s_j are unstable variables or $j = 0$. Other entries are usually $(-\infty, \infty)$.

Let $\mathcal{W}$ denote the set of bd-states that are reachable according to the bi-bounded delay analysis method, and let $\mathcal{R}_a$ denote a binary relation on the set $\mathcal{W}$ denoting the possible successor states. The network starts in the bd-state $\langle b, \Sigma \rangle$, where $\Sigma = \{\sigma_{ij}\}$, is a lower triangular matrix such that $\sigma_{i0} = [d_i, D_i)$, and $\sigma_{i,j}$ is $(d_i - D_j, D_i - d_j)$ when $s_i, s_j \in \mathcal{U}(a \cdot b)$ and $(-\infty, \infty)$ otherwise. Intuitively, this single state captures every race state in the set $\mathcal{J}$ defined earlier. Now given a bd-state $\langle c, \Sigma \rangle \in \mathcal{W}$ we progress as in the GMW model by choosing some subset $\mathcal{P}$ of the unstable variables as candidates to be complemented. Not every subset of unstable variables, however, is a suitable candidate for change. We must also consider the amount of time left on the corresponding alarm clocks. There are three conditions that must be satisfied: First, every active alarm clock must be positive, since we are trying to compute a successor state, i.e., a state reached later in time. Second, every variable in $\mathcal{P}$ must have the same value on its alarm clock, i.e., σ_{ij} must be equal to $[0, 0]$ for every s_i and s_j in $\mathcal{P}$. Finally, the alarm clock of every unstable variable that does not

change must be strictly larger than any of the changing variables' alarm clocks. We determine whether such a $\mathcal{P}$ is possible by intersecting Σ with intervals derived according to these three conditions, and then applying Algorithm 1 to deduce whether the matrix obtained is feasible or not. If it is, we compute the new state reached as follows. The network state is obtained by complementing the variables in $\mathcal{P}$. Computing the new matrix is more complex. Intuitively, the computation is to move the reference time point, or time r_0 in the matrix, to the time when the variables in $\mathcal{P}$ change.

More formally, the procedure is defined inductively as follows:

Basis: $\langle b, \theta(b, \Sigma)\rangle \in \mathcal{W}$, where $\Sigma = \{\sigma_{ij}\}$ is a lower triangular matrix of intervals defined for $0 \leq j < i \leq m$ as

$$\sigma_{ij} = \begin{cases} [d_i, D_i) & \text{if } j = 0, \\ (-\infty, \infty) & \text{otherwise.} \end{cases}$$

Induction step: Given $q = \langle c, \Sigma\rangle \in \mathcal{W}$,

1. if $\mathcal{U}(a{\cdot}c) = \emptyset$, then $q \mathcal{R}_a q$;
2. if $\mathcal{U}(a{\cdot}c) \neq \emptyset$, select any $\mathcal{P} \subseteq \mathcal{U}(a{\cdot}c)$, $\mathcal{P} \neq \emptyset$ such that $\ddot{\Sigma} = \theta(c, \dot{\Sigma})$ is feasible, where $\dot{\Sigma} = \{\dot{\sigma}_{ij}\}$ is defined below. For $0 \leq j < i \leq m$

$$\dot{\sigma}_{ij} = \begin{cases} \sigma_{ij} \cap (0, \infty) & \text{if } j = 0, \\ \sigma_{ij} \cap [0, 0] & \text{if } s_i \in \mathcal{P} \text{ and } s_j \in \mathcal{P}, \\ \sigma_{ij} \cap (0, \infty) & \text{if } s_i \in \mathcal{U}(a{\cdot}c) - \mathcal{P} \text{ and } s_j \in \mathcal{P}, \\ \sigma_{ij} \cap (-\infty, 0) & \text{if } s_i \in \mathcal{P} \text{ and } s_j \in \mathcal{U}(a{\cdot}c) - \mathcal{P}, \\ \sigma_{ij} & \text{otherwise.} \end{cases}$$

Let $\tilde{q} = \langle \tilde{c}, \theta(\tilde{c}, \tilde{\Sigma})\rangle \in \mathcal{W}$ and $q \mathcal{R}_a \tilde{q}$, where

$$\tilde{c}_i = \begin{cases} S_i(a{\cdot}c) & \text{if } s_i \in \mathcal{P}, \\ c_i & \text{otherwise,} \end{cases}$$

and $\tilde{\Sigma} = \{\tilde{\sigma}_{ij}\}$ is defined as follows. For any $s_k \in \mathcal{P}$ and for $0 < i \leq m$ let

$$\tilde{\sigma}_{i0} = \begin{cases} \ddot{\sigma}_{ik} & \text{if } i \in O \text{ and } k < i, \\ -\ddot{\sigma}_{ki} & \text{if } i \in O \text{ and } k \geq i, \\ [d_i, D_i) & \text{otherwise.} \end{cases}$$

For $0 < j < i \leq m$ let

$$\tilde{\sigma}_{ij} = \begin{cases} \ddot{\sigma}_{ij} & \text{if } i \in O \text{ and } j \in O, \\ (-\infty, \infty) & \text{otherwise,} \end{cases}$$

where $O = \{i \mid s_i \in \mathcal{U}(a{\cdot}c) \cap \mathcal{U}(a{\cdot}\tilde{c}) \text{ and } s_i \notin \mathcal{P}\}$.

To illustrate the method, consider circuit $C_{8.5}$ of Figure 8.5 started in stable total state $X{\cdot}y_1y_2 = 0{\cdot}11$, with X changing to 1 at time 0. Assume the delay of the inverter is bounded by $[1, 3)$ and the delay of the NAND

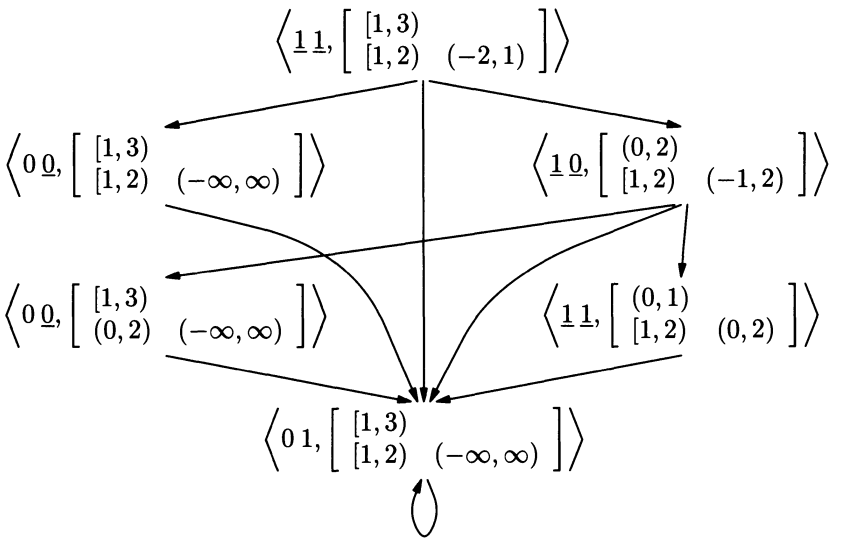


FIGURE 8.7. Example of bi-bounded delay analysis.

gate is bounded by $[1,2)$. In Figure 8.7 we show $\mathcal{W}$ and $\mathcal{R}_a$ according to such a bi-bounded delay analysis.

To conclude this section we need to show that the method described above is both correct and viable. It is easy to convince oneself that the method produces a finite number of bd-states in $\mathcal{W}$. The main reason for this is that the operations we perform on the matrices are all such that the intervals always have integer bounds either between $D = \max_{1 \le i \le m} D_i$ and $-D$, or equal to $\pm\infty$. The correctness of the method is shown in the following theorem which relates the $\mathcal{W}$ and $\mathcal{R}_a$ with the results predicted by the bi-bounded delay race model.

Theorem 8.2 *The outcome computed by the method above is identical to the outcome predicted by the bi-bounded delay race model. Formally,*

$$\{s \mid \langle b, \hat{\Sigma}\rangle \mathcal{R}_a^* \langle s, \Sigma\rangle \text{ and } \langle s, \Sigma\rangle \mathcal{R}_a^+ \langle s, \Sigma\rangle\} = \{s \mid \exists \tau > 0 \text{ such that } s \in \mathit{Reach}(t) \text{ for all } t \ge \tau\},$$

where $\langle b, \hat{\Sigma}\rangle$ is the bd-state defined by the basis case in the algorithm.

Proof: We refer the reader to [45] or [84] for the proof. □

8.4 Continuous Ternary Model

The main problem with the analysis method described in the previous section is its large computational requirement; the algorithm is significantly

more time-consuming than even a GMW analysis. Thus, the method is only useful for very small networks. In this section, which is based on the work in [122], we present a bi-bounded delay race model related to the extended bi-bounded inertial delay model, and an efficient analysis method for this model.

The underlying race model is called the *extended bi-bounded delay* (*XBD*) model and is defined for ternary networks. Assume that a ternary network $\mathbf{N} = \langle \{0, \Phi, 1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, \mathbf{F} \rangle$ is started in stable total state $\hat{a}\cdot\mathbf{b}$, and that the input changes to a at time 0. Note that we always assume the inputs are binary; thus, both $\hat{a}$ and a belong to $\{0,1\}^n$. We also assume that the delay $\delta_j(t)$ of variable $\mathbf{s}_j$, $1 \leq j \leq m$, satisfies $d_j \leq \delta_j(t) < D_j$, for some nonnegative integers d_j and D_j.

A certain amount of previous excitation history is needed in the XBD race model. Define a *race state* to be a 4-tuple $[\mathbf{c}, u, v, t]$ as follows. The first component, $\mathbf{c} \in \{0, \Phi, 1\}^m$, is the current state of the network. The second component, u, is a vector of m real numbers and is used to remember how long an unstable variable with a binary value has been unstable. The third component, v, serves a similar purpose, but is used to remember how long an unstable variable has had a binary excitation. The last component, t, is a real number denoting the time at which this state was reached. Note that the input is assumed to change (from $\hat{a}$ to a) at time 0.

Let $\mathcal{Q}$ denote the set of race states reachable according to the XBD race model, and let R_a denote a binary relation (to be defined) on the set $\mathcal{Q}$ denoting the possible successor states. An *XBD race sequence* is an infinite sequence $[\mathbf{s}^0, u^0, v^0, t^0], [\mathbf{s}^1, u^1, v^1, t^1], \ldots$ of race states such that

$$[\mathbf{s}^0, u^0, v^0, t^0] = [\mathbf{b}, (0, \ldots, 0), (0, \ldots, 0), 0]$$

and

$$[\mathbf{s}^h, u^h, v^h, t^h] R_a [\mathbf{s}^{h+1}, u^{h+1}, v^{h+1}, t^{h+1}], \text{ for } h \geq 0.$$

One can view an XBD race sequence as a sequence of "snapshots" of the network. The network starts in state $[\mathbf{b}, (0, \ldots, 0), (0, \ldots, 0), 0]$ of $\mathcal{Q}$ at time 0. Given some state $[\mathbf{c}, u, v, t]$ in $\mathcal{Q}$, we determine a possible successor state as follows. If $a\cdot\mathbf{c}$ is a stable total state, the network remains in this state indefinitely; thus a stable state has only itself as successor. Otherwise, we take a new snapshot of the network at time $t + \delta$ for some $\delta > 0$. Since this snapshot is to capture the next change of the network state, δ must be chosen carefully. Because a variable with a binary value must be unstable for at least d_j units of time before it can change to Φ, and an unstable variable must have the same binary excitation for at least d_j units of time before it can change to this excitation, δ must be chosen so that there exists at least one unstable variable $\mathbf{s}_j$ for which $u_j + \delta$ or $v_j + \delta$ is greater than or equal to d_j. On the other hand, since a variable cannot be unstable and have a binary value, or be unstable and have a binary excitation, for D_j

units of time without changing, δ must be chosen so that each of $u_j + \delta$ and $v_j + \delta$ is strictly smaller than D_j for all unstable variables $\mathbf{s}_j$.

Once δ has been selected, two sets, C^{Φ} and C^{B}, are computed. The set C^{Φ} contains all the variables that, for this δ, are candidates for changing to Φ, i.e., it contains all the variables that are binary, unstable, and for which $u_j + \delta$ is greater than or equal to d_j. The set C^{B} contains all the variables that, for this δ, are candidates for changing from Φ to a binary value, i.e., all the variables that are Φ, are unstable, and for which $v_j + \delta$ is greater than or equal to d_j. Finally, some nonempty subset of $C^{\Phi} \cup C^{B}$ is chosen, the appropriate variable values are changed, and u and v are updated accordingly.

It is important to note that, if a state in $\mathcal{Q}$ is unstable, it has infinitely many possible successor race states, though the number of $\mathbf{s}$ values is, of course, finite.

As before, let $\mathcal{U}(a{\cdot}\mathbf{c})$ denote the set of vertices that are unstable in the total state $a{\cdot}\mathbf{c}$, i.e.,

$$\mathcal{U}(a{\cdot}\mathbf{c}) = \{\mathbf{s}_j \mid \mathbf{S}_j(a{\cdot}\mathbf{c}) \neq \mathbf{c}_j\}.$$

Define the set of vertices that have a binary value in state $\mathbf{c}$ to be

$$\mathcal{B}(\mathbf{c}) = \{\mathbf{s}_j \mid \mathbf{c}_j \in \{0,1\}\}.$$

Also, let the set of vertices that have a binary excitation in the total state $a{\cdot}\mathbf{c}$ be

$$\mathcal{BE}(a{\cdot}\mathbf{c}) = \{\mathbf{s}_j \mid \mathbf{S}_j(a{\cdot}\mathbf{c}) \in \{0,1\}\}.$$

Now $\mathcal{Q}$ and R_a are formally defined inductively as follows:

Basis: Let $[\mathbf{b}, (0,\ldots,0), (0,\ldots,0), 0] \in \mathcal{Q}$.

Induction step: Given $q = [\mathbf{c}, u, v, t] \in \mathcal{Q}$,

1. if $\mathcal{U}(a{\cdot}\mathbf{c}) = \emptyset$, then qR_aq;
2. if $\mathcal{U}(a{\cdot}\mathbf{c}) \neq \emptyset$, then for any $\delta > 0$ such that $\delta_{\min} \leq \delta < \delta_{\max}$, where

$$\delta_{\min} = \min\{\min\{d_j - u_j, d_j - v_j\} \mid \mathbf{s}_j \in \mathcal{U}(a{\cdot}\mathbf{c})\},$$

and

$$\delta_{\max} = \min\{\min\{D_j - u_j, D_j - v_j\} \mid \mathbf{s}_j \in \mathcal{U}(a{\cdot}\mathbf{c})\},$$

let

$$C^{\Phi} = \mathcal{U}(a{\cdot}\mathbf{c}) \cap \mathcal{B}(\mathbf{c}) \cap \{\mathbf{s}_j \mid u_j + \delta \geq d_j\}$$

and

$$C^{B} = \mathcal{U}(a{\cdot}\mathbf{c}) \cap \{\mathbf{s}_j \mid \mathbf{c}_j = \Phi\} \cap \{\mathbf{s}_j \mid v_j + \delta \geq d_j\}.$$

Finally, for any $\mathcal{P} \subseteq C^{\Phi} \cup C^{B}$, $\mathcal{P} \neq \emptyset$, let $q' = [\mathbf{c}', u', v', t + \delta] \in \mathcal{Q}$ and $qR_a q'$, where

$$\mathbf{c}'_j = \begin{cases} \Phi & \text{if } \mathbf{s}_j \in \mathcal{P} \cap C^{\Phi}, \\ \mathbf{S}_j(a{\cdot}\mathbf{c}) & \text{if } \mathbf{s}_j \in \mathcal{P} \cap C^{B}, \\ \mathbf{c}_j & \text{otherwise,} \end{cases}$$

$$u'_j = \begin{cases} u_j + \delta & \text{if } \mathbf{s}_j \in \mathcal{U}(a{\cdot}\mathbf{c}) \cap \mathcal{B}(\mathbf{c}) \cap \mathcal{U}(a{\cdot}\mathbf{c}') \cap \mathcal{B}(\mathbf{c}'), \\ 0 & \text{otherwise,} \end{cases}$$

and

$$v'_j = \begin{cases} v_j + \delta & \text{if } \mathbf{s}_j \in \mathcal{U}(a{\cdot}\mathbf{c}) \cap \mathcal{BE}(a{\cdot}\mathbf{c}) \cap \mathcal{U}(a{\cdot}\mathbf{c}') \cap \mathcal{BE}(a{\cdot}\mathbf{c}'), \\ 0 & \text{otherwise.} \end{cases}$$

Example 2

To illustrate the definition above, consider network $C_{8.8}$ of Figure 8.8 started in stable total state 1·100 when the input changes to 0. Two possible XBD race sequences are shown in Figure 8.9. We use the notation 0_{Φ}, (0_1, Φ_0, etc.) to denote a variable that currently has the value 0 (0, Φ, etc.) and excitation Φ (1, 0, etc.). The numbers to the left of an arrow denote the limits for δ, and the number to the right of an arrow corresponds to the δ chosen.

To illustrate some steps in the computation, consider the left XBD race sequence of Figure 8.9. We start in state $[1_0 00, (0,0,0), (0,0,0), 0]$, where $\mathcal{U}(0{\cdot}100) = \{\mathbf{s}_1\}$. Now, for all the unstable variables $\mathbf{s}_j$ (in this case variable $\mathbf{s}_1$ only), we compute the values of $d_j - u_j$, $d_j - v_j$, $D_j - u_j$, and $D_j - v_j$ in order to find the lower and the upper bounds on δ. We get $\delta_{\min} = 1$ and $\delta_{\max} = 3$. In our example, we choose $\delta = 2$. Once δ has been chosen, we must determine the set C^{Φ} of variables that, for this choice of δ, are candidates for changing to Φ, and the set C^{B} of candidates for changing from Φ to a binary value. More specifically, the variables in C^{Φ} are all unstable, all have

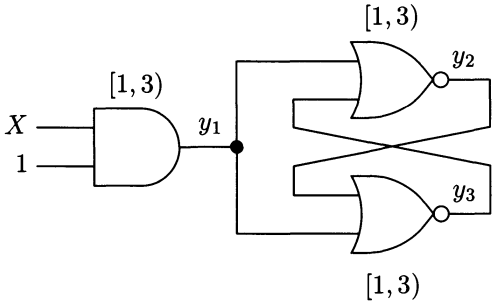


FIGURE 8.8. Network $C_{8.8}$.

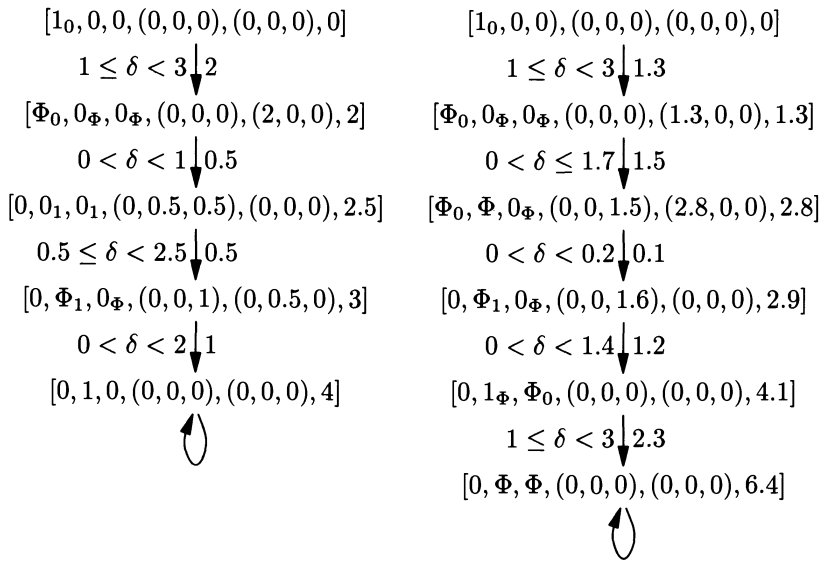


FIGURE 8.9. Two possible XBD sequences.

a binary value, and for all of them $u_j + \delta \geq d_j$. The set C^B is defined similarly. In the present situation $C^\Phi = \{\mathbf{s}_1\}$ and $C^B = \emptyset$. Once C^Φ and C^B have been computed, we have to choose some nonempty subset of $C^\Phi \cup C^B$ as the set $\mathcal{P}$. Here there is no choice; $\mathcal{P}$ must be equal to $\{\mathbf{s}_1\}$ Given $\mathcal{P}$ and δ, we now proceed as follows. The state of variable $\mathbf{s}_j$ is changed to Φ if $\mathbf{s}_j$ is in $C^\Phi \cap \mathcal{P}$, and to its excitation value if $\mathbf{s}_j$ is in $C^B \cap \mathcal{P}$. No other variables are changed. In our example, only $\mathbf{s}_1$ is changed, and it is changed to Φ. When this is done, the vectors u and v are updated as follows: If a variable $\mathbf{s}_j$ was unstable and had a binary excitation in the previous state and the same situation holds in this new state, then u_j is incremented by δ. Otherwise, u_j is set to 0. Similarly, a variable that was unstable and had a binary excitation in the previous as well as the new state, gets v_j incremented by δ ; all other variables get v_j set to 0. In our example, variable $\mathbf{s}_1$ is not binary in the new state and thus $u'_1 = 0$. On the other hand, variable $\mathbf{s}_1$ satisfies all the conditions for getting v_1 incremented, and thus $v'_1 = \delta = 2$. Variables $\mathbf{s}_2$ and $\mathbf{s}_3$ were not unstable in the previous race state; thus $u'_2 = u'_3 = v'_2 = v'_3 = 0$. Hence, a possible successor state is $[\Phi_0 0_\Phi 0_\Phi, (0, 0, 0), (2, 0, 0), 2]$.

Next we determine a successor state to $[\Phi_0 0_\Phi 0_\Phi, (0, 0, 0), (2, 0, 0), 2]$. First, $U(0{\cdot}\Phi 00) = \{\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3\}$. We then find that $\delta_{\min} = -1$ and $\delta_{\max} = 1$. Since δ must be nonnegative, it must be in the open interval

$(0,1)$. In our example, δ is chosen to be 0.5. It is easy to verify that, for this choice of δ, $C^{\Phi} = \emptyset$ and $C^{B} = \{\mathbf{s}_1\}$. Although variables $\mathbf{s}_2$ and $\mathbf{s}_3$ are unstable, they have only been unstable from time 2. Since the maximum delay in variable $\mathbf{s}_1$ must be less than 3, it follows that only variable $\mathbf{s}_1$ will be a candidate for changing. As above, we get $\mathcal{P} = \{\mathbf{s}_1\}$. The reader can verify that the network reaches state $[00_1 0_1, (0, 0.5, 0.5), (0, 0, 0), 2.5]$.

Let $\mathcal{G}$ be the set of all XBD race sequences for network $\mathbf{N}$, when $\mathbf{N}$ is started in stable total state $\hat{a}\cdot\mathbf{b}$ and the input changes to a at time 0. For $g \in \mathcal{G}$, the state of the network according to this race sequence is written as $\mathbf{s}^g(t)$, and is equal to

$$\mathbf{s}^g(t) = \begin{cases} \mathbf{b} & \text{for } t < 0, \\ \mathbf{s}^i & \text{for } t^i \le t < t^{i+1} \text{ if } \mathcal{U}(a\cdot\mathbf{s}^i) \neq \emptyset, \\ \mathbf{s}^i & \text{for } t^i \le t \text{ if } \mathcal{U}(a\cdot\mathbf{s}^i) = \emptyset. \end{cases}$$

The set of possible states of the network at time t is called $Reach(t)$ and is defined as

$$Reach(t) = \{\mathbf{s} \mid \mathbf{s} = \mathbf{s}^g(t) \text{ for some } g \in \mathcal{G}\}.$$

In the example above, it is easy to verify that $Reach(t) = \{100\}$ for $0 \le t < 1$, that $Reach(1) = \{100, \Phi 00\}$, and that $Reach(t) = \{100, \Phi 00, 000\}$ for $1 < t < 2$. It is very laborious, however, to compute $Reach(t)$ for $t \ge 2$. We will return shortly to the question of computing $Reach(t)$ efficiently. Before doing this, we establish that the XBD race model captures exactly the behavior of a network consisting of ideal, delay-free excitation functions connected in series with extended bi-bounded inertial delays. We do this by proving two lemmas. The first lemma shows that the behavior of a network according to an XBD sequence is consistent with the extended bi-bounded inertial-delay model. More precisely:

Lemma 8.4 *Let* $g = [\mathbf{s}^0, u^0, v^0, t^0], [\mathbf{s}^1, u^1, v^1, t^1], \ldots$ *be an arbitrary XBD race sequence for network* $\mathbf{N}$*, when* $\mathbf{N}$ *is started in stable total state* $\hat{a}\cdot\mathbf{b}$ *and the input changes to* a *at time* 0*. Furthermore, let*

$$X(t) = \begin{cases} \hat{a} & \text{for } t < 0, \\ a & \text{for } t \ge 0. \end{cases}$$

Then the input/output behavior of every variable $\mathbf{s}_j$ *is consistent with the extended bi-bounded inertial-delay model, i.e.,* $\mathbf{S}_j(X(t), \mathbf{s}^g(t))/\mathbf{s}_j^g(t)$ *is an acceptable input/output waveform according to the XID model.*

Proof: The proof is straightforward but quite lengthy. Hence, it is not included here and we refer the interested reader to [122]. □

The converse result is stated in Lemma 2. Let $X(t)$ and $\mathbf{s}(t)$ denote the input and state of network $\mathbf{N}$ at time t.

Lemma 8.5 *Assume that* $\mathbf{S}_j(X(t), \mathbf{s}(t))/\mathbf{s}_j(t)$, *for* $1 \leq i \leq m$, *is an acceptable input/output waveform according to the XID model, when* $\mathbf{N}$ *is started in stable total state* $\hat{a}\cdot\mathbf{b}$ *and the input changes to* a *at time* 0. *Then we can construct a valid XBD sequence corresponding to this input/output waveform.*

Proof: Again, the proof is straightforward but lengthy and the interested reader is referred to [122]. □

In summary, the two lemmas above establish a direct link between the extended bi-bounded inertial-delay model and the behavior of a network according to the extended bi-bounded delay race model.

8.5 Discrete Ternary Model

The XBD race model is more of theoretical interest than of practical use, since each state can have infinitely many successor states. We now define an efficient algorithm, called the *ternary bi-bounded delay algorithm* (TBD algorithm), for simulating a network. We will show that the results obtained by applying this method summarize the outcome predicted by the extended bi-bounded delay race model.

The basic idea behind the TBD algorithm is quite simple and is given in the following two rules:

1. Change an unstable variable to Φ as soon as allowed by its minimum delay.

2. Change a variable from Φ to a binary value as late as possible.

An informal description of a similar algorithm was given by [32] in 1971.

In the TBD algorithm, as in the XBD race model, it is necessary to remember a certain amount of previous excitation history. For this reason, define a *tbd-state* to be the triple $\langle \mathbf{z}, U, V \rangle$. The first component, $\mathbf{z}$, is the current "summarized" state of the network. The second component, U, is a vector of m integer values. For a stable variable $\mathbf{s}_j$, $U_j = 0$, whereas for an unstable variable with a binary value, U_j denotes the current "race unit" of the variable. For example, in the starting state, every unstable binary variable will have $U_j = 1$, denoting that the variable is currently in its first unstable time slot. Similarly, V is used like U, but here the criterion is that the variable is unstable and has a binary excitation. The summarized state of the network is computed at times $1, 2, 3, \ldots$, in a two-step process. Given the state $\langle \mathbf{z}^{h-1}, U^{h-1}, V^{h-1} \rangle$, we first compute an intermediate state $\langle \tilde{\mathbf{z}}^h, \tilde{U}^h, \tilde{V}^h \rangle$. Intuitively, this state is the summarized state of the network at time $h - \epsilon$ for an arbitrarily small ϵ. To compute $\tilde{\mathbf{z}}^h$, we change only the variables that have to change to their binary excitation. These are

the variables that have $V_j^{h-1} = D_j$. Owing to these changes, some other variables may become stable, and they are removed from U and V.

Once the intermediate tbd-state is calculated, we compute the new "next" state, i.e., the summarized state of the network at time h. First, to obtain $\mathbf{z}^h$, we change all the variables that may change to Φ. These are the variables that have $\tilde{U}_j^h = d_j$. All other variables are unchanged. Second, the vectors U and V are updated. This time the update consists of incrementing U_j and V_j by 1, if variable $\mathbf{s}_j$ satisfies the conditions below, and setting them to 0 otherwise. For example, a variable $\mathbf{s}_j$ that is unstable and has a binary value in $\mathbf{z}^h$ will get $U_j^h = \tilde{U}_j^h + 1$, whereas a variable that became stable in $\mathbf{z}^h$ will get $U_j^h = 0$.

Formally, the TBD algorithm is defined inductively as follows:

Basis: $\mathbf{z}^0 = \mathbf{b}$.

$$U_j^0 = \begin{cases} 1 & \text{if } \mathbf{s}_j \in \mathcal{U}(a{\cdot}\mathbf{z}^0) \cap \mathcal{B}(\mathbf{z}^0), \\ 0 & \text{otherwise;} \end{cases}$$

$$V_j^0 = \begin{cases} 1 & \text{if } \mathbf{s}_j \in \mathcal{U}(a{\cdot}\mathbf{z}^0) \cap \mathcal{BE}(a{\cdot}\mathbf{z}^0), \\ 0 & \text{otherwise.} \end{cases}$$

Induction step: Given $\langle \mathbf{z}^{h-1}, U^{h-1}, V^{h-1} \rangle$, we first compute the intermediate tbd-state $\langle \tilde{\mathbf{z}}^h, \tilde{U}^h, \tilde{V}^h \rangle$ as follows:

$$\tilde{\mathbf{z}}_j^h = \begin{cases} \mathbf{S}_j(a{\cdot}\mathbf{z}^{h-1}) & \text{if } V_j^{h-1} = D_j, \\ \mathbf{z}_j^{h-1} & \text{otherwise;} \end{cases}$$

$$\tilde{U}_j^h = \begin{cases} U_j^{h-1} & \text{if } \mathbf{s}_j \in \mathcal{U}(a{\cdot}\tilde{\mathbf{z}}^h) \cap \mathcal{B}(\tilde{\mathbf{z}}^h), \\ 0 & \text{otherwise;} \end{cases}$$

$$\tilde{V}_j^h = \begin{cases} V_j^{h-1} & \text{if } \mathbf{s}_j \in \mathcal{U}(a{\cdot}\tilde{\mathbf{z}}^h) \cap \mathcal{BE}(a{\cdot}\tilde{\mathbf{z}}^h), \\ 0 & \text{otherwise.} \end{cases}$$

Once this intermediate state is known, the state $\langle \mathbf{z}^h, U^h, V^h \rangle$ is computed as follows:

$$\mathbf{z}_j^h = \begin{cases} \Phi & \text{if } \tilde{U}_j^h = d_j, \\ \tilde{\mathbf{z}}_j^h & \text{otherwise;} \end{cases}$$

$$U_j^h = \begin{cases} \tilde{U}_j^h + 1 & \text{if } \mathbf{s}_j \in \mathcal{U}(a{\cdot}\mathbf{z}^h) \cap \mathcal{B}(\mathbf{z}^h), \\ 0 & \text{otherwise;} \end{cases}$$

$$V_j^h = \begin{cases} \tilde{V}_j^h + 1 & \text{if } \mathbf{s}_j \in \mathcal{U}(a{\cdot}\mathbf{z}^h) \cap \mathcal{BE}(a{\cdot}\mathbf{z}^h), \\ 0 & \text{otherwise.} \end{cases}$$

Example 3

To illustrate the algorithm, consider network $C_{8.8}$ of Figure 8.8 started in stable total state 1·100, when the input changes to $X = 0$. In Figure 8.10 we show the results of the TBD algorithm. It is easy to verify that $\mathbf{z}^3$ is stable; hence the algorithm terminates at this point.

$$
\begin{array}{ll}
z_0: & \langle 1_0 00, (1,0,0), (1,0,0)\rangle \\
 & \downarrow \\
\tilde{z}_1: & \langle 1_0 00, (1,0,0), (1,0,0)\rangle \\
 & \downarrow \\
z_1: & \langle \Phi_0 0_\Phi 0_\Phi, (0,1,1), (2,0,0)\rangle \\
 & \downarrow \\
\tilde{z}_2: & \langle \Phi_0 0_\Phi 0_\Phi, (0,1,1), (2,0,0)\rangle \\
 & \downarrow \\
z_2: & \langle \Phi_0 \Phi\Phi, (0,0,0), (3,0,0)\rangle \\
 & \downarrow \\
\tilde{z}_3: & \langle 0\Phi\Phi, (0,0,0), (0,0,0)\rangle \\
 & \downarrow \\
z_3: & \langle 0\Phi\Phi, (0,0,0), (0,0,0)\rangle \\
 & \circlearrowleft
\end{array}
$$

FIGURE 8.10. TBD analysis of $C_{8.8}$.

Example 4

Our next example is more complicated. Consider the network $C_{8.11}$ of Figure 8.11 started in stable total state $X = 0$, $s = 0000001$,

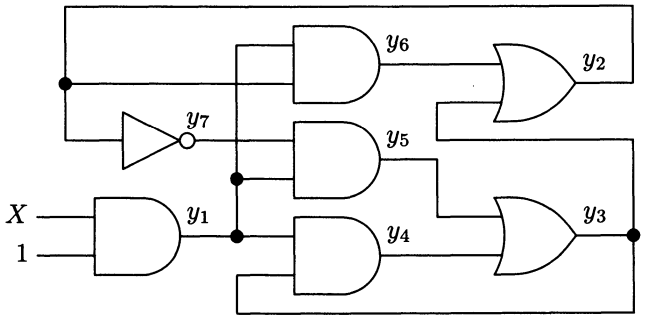


FIGURE 8.11. Network $C_{8.11}$.

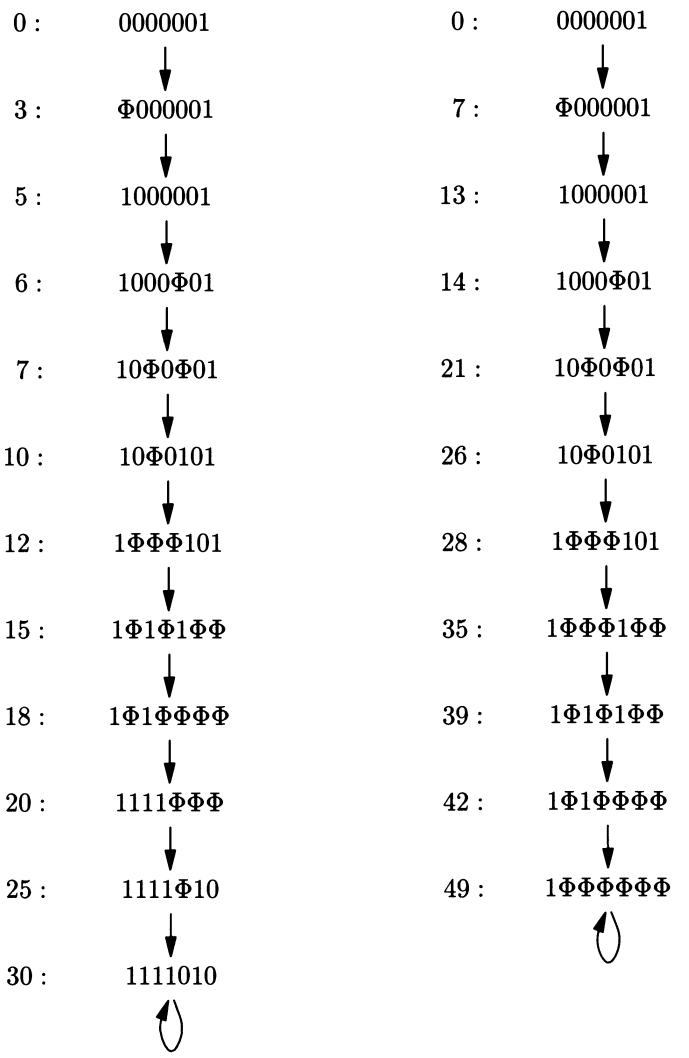


FIGURE 8.12. TBD analyses with ±25% and ±30% deviation.

when the input changes to $X = 1$ at time 0. In Figure 8.12 we have summarized the TBD analyses of this transition when the delay in each gate is bounded by $[3, 5)$ and $[7, 13)$, respectively. To clarify the picture, we only show $\mathbf{z}^h$ when $\mathbf{z}^h \neq \mathbf{z}^{h-1}$. If the base time is in ns, then, if the delay is bounded by $[7, 13)$, we are analyzing a situation where the gate delays are 10 ns ±30%. Similarly, if the base time corresponds to 2.5 ns, then, if the delays are bounded by $[3, 5)$, the gate delays are 10 ns ±25%.

We are now ready to characterize the results obtained by a TBD analysis. Let $\mathbf{z}^0, \tilde{\mathbf{z}}^1, \mathbf{z}^1, \ldots$ be the sequence of states computed by the TBD algorithm and let *Reach*(t) be the outcome according to the extended bi-bounded delay race model. The following theorem shows that the TBD algorithm can be used to get essentially the same information as that found by the XBD race model.

Theorem 8.3 $\mathbf{z}^r = lub\ Reach(r)$ *for* $r = 0, 1, \ldots$.

Proof: The proof of this result is too lengthy to include here, and we refer the reader to [122]. □

We are now in a position to interpret the results of the TBD analyses of $C_{8.11}$ shown in Figure 8.12. By Theorem 2, we can conclude that, according to the extended bi-bounded delay race model, the circuit can tolerate a $\pm 25\%$ variation of the gate delays but not a $\pm 30\%$ variation.

One of the main attractions of the TBD algorithm is its computational efficiency. A TBD analysis usually requires at most twice the computational effort required by a nominal delay analysis. If we start with a binary network and use its ternary extension as our network for the analysis, however, we obtain this computational efficiency at a price. In particular, if we compare with a (continuous) binary bounded delay analysis, we may get "false negatives," i.e., our analysis in the ternary domain may not be an exact summary of a binary analysis. Although the TBD analysis result is correct with respect to an extended bi-bounded race analysis, it is more conservative than the binary bi-bounded delay analysis. A possible multi-level approach to analyzing a transition would be to start with ternary simulation. If the result of ternary simulation is binary, then we are done. Otherwise, use the TBD algorithm to analyze the transition. Again, if this yields a binary final state, we are done. Finally, if all else fails, use the binary bi-bounded delay analysis algorithm to analyze the transition. By using the most efficient but most pessimistic analysis techniques first, we can reduce the number of transitions that need to be analyzed by the computationally very expensive methods. In Chapter 14 we return to this topic and show how we can analyze several transitions at once by using symbolic formulations of the algorithms discussed so far in the book.

Chapter 9

Complexity of Race Analysis

Two questions naturally arise in connection with the analysis of a transition caused by an input change: 1) Will the network eventually reach a unique (binary) stable state? 2) Will the network be in a unique (binary) state at time t? We call the first question the "stable-state reachability" (SSR) problem and the second the "limited reachability" (LR) problem. Both questions can be answered by using the race analysis algorithms presented in earlier chapters. Some of these algorithms are highly efficient, whereas others appear to require time exponential in the size of the network to be analyzed. In this chapter we explore the inherent computational complexity of these analysis problems. We assume the reader is familiar with the standard terminology for NP-completeness, as described in [55]. The work given here is based mainly on [122, 123].

How To Read This Chapter

This chapter can be omitted on first reading since no subseqeuent material depends directly on it. For a summary of the main results, the reader is directed to Tables 9.1 and 9.2 on pages 181 and 185, respectively.

9.1 Stable-State Reachability

Assume that N is a network started in stable total state $\hat{a}\cdot b$. The *stable-state reachability* (SSR) problem is: If the input changes to a at time 0 and is kept at that value, does the network eventually reach a unique stable binary state?

Before studying the computational complexity of this question, we need to make a rather fundamental assumption. Unless otherwise stated, we henceforth assume that the excitation function S_j, $1 \leq j \leq m$, can be computed using at most $O(m^k)$ space and in time $O(m^k)$ for some constant $k > 0$. This assumption is quite reasonable. Also, if it does not hold, even the problem of determining whether a given state is unstable is intractable. In fact, for many types of circuits it is reasonable to assume that the excitation functions can be computed in constant time. Gate circuits with bounded fan-in provide a typical example.

In the following we study the complexity of the SSR problem for different race models. We start with the simplest case—the unit-delay model.

Theorem 9.1 *The SSR problem is PSPACE-complete for the unit-delay race model.*

Proof: For the unit-delay model, SSR is clearly in PSPACE, since we can simulate the circuit and use a counter to keep track of the number of state changes. If the circuit has not reached a stable state within 2^m steps, it must have entered an oscillation and will never reach a stable state. Since the simulation requires only enough space to compute the excitation functions, i.e., space polynomial in m, and the counter needs only space linear in m, it follows that the unit-delay SSR (UD-SSR) problem is in PSPACE.

To prove the theorem, it is therefore sufficient to establish that a known PSPACE-complete problem can be reduced to the UD-SSR problem. The quantified Boolean formula (QBF) problem turns out to be appropriate. Following [65] a QBF is defined as follows:

1. If x is a variable, then it is a QBF. The occurrence of x is free.

2. If E_1 and E_2 are QBFs, then so are $\neg(E_1)$, $(E_1) \wedge (E_2)$, and $(E_1) \vee (E_2)$. An occurrence of x is free or bound, depending on whether the occurrence is free or bound in E_1 or E_2. Redundant parentheses can be omitted.

3. If E is a QBF, then $\exists x(E)$ and $\forall x(E)$ are QBFs. The scopes of $\exists x$ and $\forall x$ are all the free occurrences of x in E. Free occurrences of x in E are bound in $\exists x(E)$ and $\forall x(E)$. All other occurrences of variables in $\forall x(E)$ and $\exists x(E)$ are free or bound, depending on whether they are free or bound in E.

A QBF with no free variables has a value of either *true* or *false*, denoted by 1 and 0, respectively. The value of such a QBF is obtained by replacing each subexpression of the form $\exists x(E)$ by $E^0 \vee E^1$ and each subexpression of the form $\forall x(E)$ by $E^0 \wedge E^1$, where E^0 (E^1) is E with all the occurrences of x in the scope of the quantifier replaced by 0 (1). The *QBF problem* is to determine whether a QBF with no free variables has the value *true*. It was shown in [65, 131] that the QBF problem is PSPACE-complete.

Given a QBF E with no free variables, we design a circuit with one input; this circuit, started in a stable state (to be defined) and with the input changing from 0 to 1, reaches a new stable state if and only if E is *true*. If E is *false*, the circuit oscillates. Thus the basic idea is to design a sequential circuit that evaluates E. We must ensure, however, that the size of this circuit—and thus the size of the UD-SSR problem—remains polynomial in the size of the QBF problem. Hence, we cannot simply take two copies of a circuit that evaluates E, use them to evaluate E for $x = 0$ and $x = 1$, and compute the OR of the results in order to compute $\exists x(E)$,

since this could generate a circuit of size exponential in the size of E. We can design a small control circuit, however, that first evaluates E for $x = 0$, stores the result, and then evaluates E for $x = 1$ and combines the two results. This is the basic idea, but the construction we actually present is more complicated and far from minimal in size. We do this in order to simplify the correctness proof.

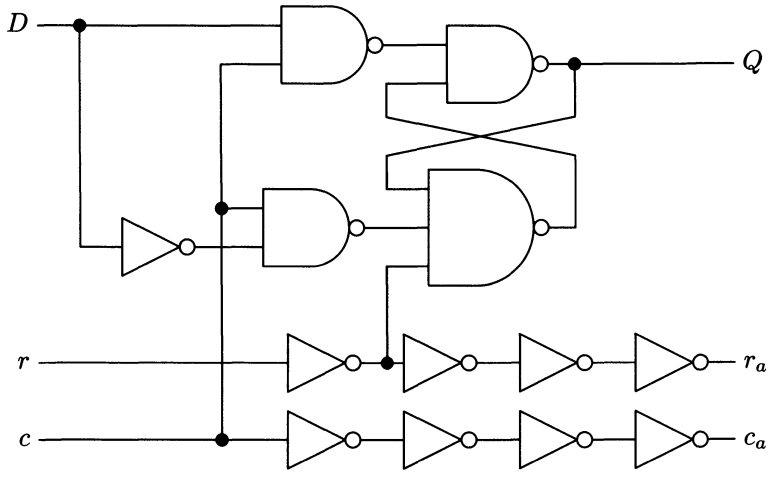


FIGURE 9.1. A D flip-flop with completion and reset signals.

Before we describe how to transform E into N, we present our basic building block—a D flip-flop (DFF) with completion signals and an asynchronous reset input. The circuit is shown in Figure 9.1; it is an ordinary D flip-flop with reset input, but it also has eight extra inverters. These inverters function as delays, making sure that the completion signal, called c_a, for the store signal, c, does not occur before the output Q has obtained the value of the input D. The reset signal r and the reset completion signal r_a are treated similarly. The signals that arrive on the store input and reset input consist of pulses $(0 \to 1 \to 0)$ 3 unit delays long. Furthermore, there is at most one pulse propagating through the flip-flop at any given time. The reader can easily verify that the circuit of Figure 9.1 does indeed behave as described under these assumptions, if every gate is assumed to have a unit delay.

Assume that E is a given QBF with no free variables, and let N denote the circuit to be constructed. We first define a subcircuit $\tilde{N}$ recursively as follows:

Basis: If $E = x$, then $\tilde{N}$ is the circuit shown in Figure 9.2(a).

Induction step: Assume that circuits $\tilde{N}_1$ and $\tilde{N}_2$ corresponding to the QBFs $E_1(x)$ and $E_2(x)$ have been constructed.

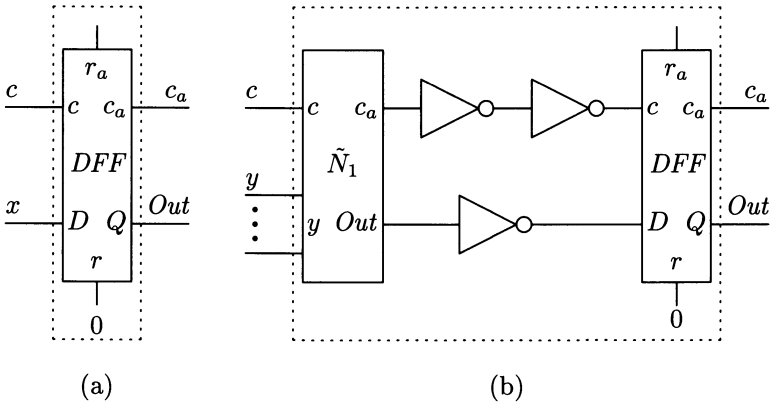


FIGURE 9.2. Construction for (a) a variable x; (b) $\neg E_1(y)$.

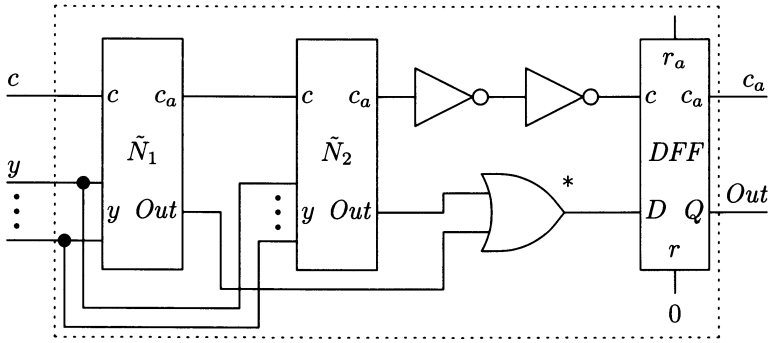


FIGURE 9.3. Construction for $E_1(y) \vee E_2(y)$.

1) If $E = \neg(E_1)$, then $\tilde{N}$ is the circuit shown in Figure 9.2(b).
2) If $E = (E_1) \vee (E_2)$, then $\tilde{N}$ is the circuit shown in Figure 9.3. Similarly, if $E = (E_1) \wedge (E_2)$, then $\tilde{N}$ is the circuit obtained by replacing the $*$-marked OR gate in Figure 9.3 by an AND gate.
3) If $E = \exists x(E_1)$, then $\tilde{N}$ is the circuit shown in Figure 9.4. Similarly, if $E = \forall x(E_1)$, then $\tilde{N}$ is the circuit obtained by replacing the $*$-marked OR gate in Figure 9.4 by an AND gate.

Finally, if $\tilde{N}$ is the circuit that corresponds to E, then N is the circuit shown in Figure 9.5.

The main idea of the construction of N is to separate the "control" path from the "data" path. Furthermore, we make sure that only one pulse propagates through the control path of the circuit, and that no signal from the data path can affect the control path. The "combinational" parts of E are rather obvious, and are shown in Figures 9.2–9.3. For example, the

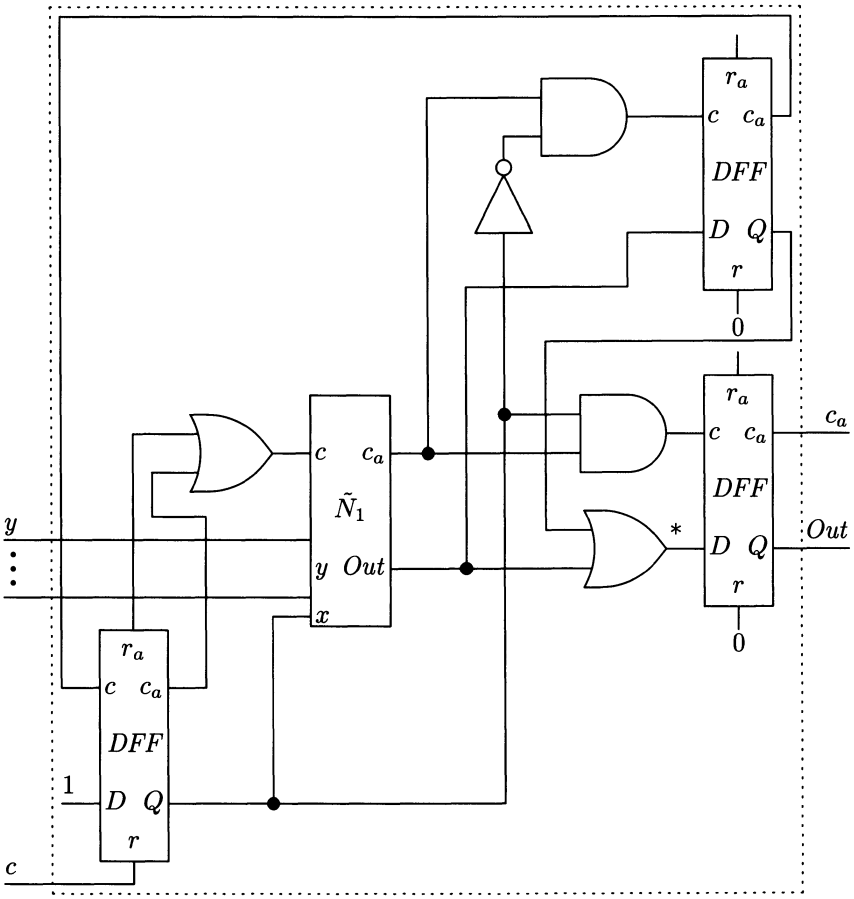


FIGURE 9.4. Construction for $\exists x.\, E_1(x, y)$.

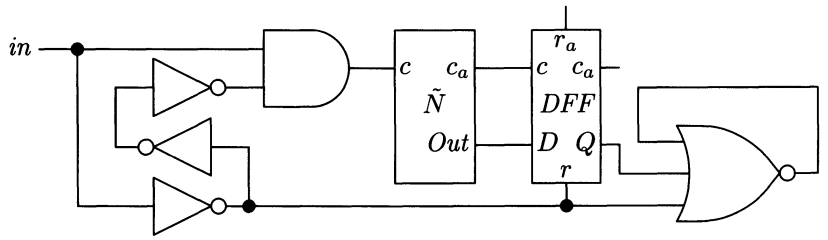


FIGURE 9.5. Complete construction for Theorem 9.1.

circuit of Figure 9.2(b) corresponds to a subexpression $\neg E_1(y)$, and works as follows: When a pulse arrives at the "start" input (c), it is immediately sent to the start input of the subcircuit that evaluates $E_1(y)$. When the

completion signal arrives from this subcircuit, a pulse is delayed 2 units of time in order to allow the inversion of the result in the data path. Finally, this inverted result is latched in the D flip-flop. When this is done, a pulse arrives at the completion output (c_a) of the entire circuit. The other combinational circuits can be verified using similar arguments.

The circuits corresponding to the expressions $\exists x.\, E_1(x,y)$ and $\forall x.\, E_1(x,y)$ are more complicated. We discuss only the circuit shown in Figure 9.4, which evaluates $\exists x.\, E_1(x,y)$. The circuit for $\forall x.\, E_1(x,y)$ works similarly. The basic idea is as follows: We want the circuit to evaluate $E_1(0,y)$, store the result, evaluate $E_1(1,y)$, OR the two results together, and latch the final result. The left-most D flip-flop serves as a "status register" keeping track of the current value of x. The topmost D flip-flop is used to remember the result of $E_1(0,y)$, and the right-most D flip-flop is used to latch the final result. When a pulse arrives at the start input c, the status register is reset. This causes x to be set to 0. Once the status register has been reset, the completion signal (r_a) is fed via the left-most OR gate to the start input (c) of the subcircuit $\tilde{N}_1$ evaluating $E_1(x,y)$. When this result $E(0,y)$ becomes available (i.e., when a pulse arrives at the c_a output), it is latched in the topmost D flip-flop. This follows because the completion output (c_a) from $\tilde{N}_1$ is directed to the store input of the topmost D flip-flop, when x is 0. After the value has been latched in the topmost D flip-flop, a pulse goes back to the status register. Since the D input of the status register is connected to 1, the output (Q) changes to 1, causing x to become 1. Now, when the output of the latch (Q) has obtained the value 1, the completion signal (c_a) is fed via the left-most OR gate to the c input of $\tilde{N}_1$. This time $x = 1$, and $\tilde{N}_1$ evaluates $E_1(1,y)$. When the result of $E_1(1,y)$ is available, it is combined with the stored value of $E_1(0,y)$ and latched in the right-most D flip-flop. The completion signal is then used to signal to the remaining circuit that the result of $\exists x.\, E_1(x,y)$ is available at the output. It is straightforward to convince oneself that the circuit of Figure 9.4 behaves as described if the unit-delay model is assumed. We leave the details to the interested reader.

The figures above have shown how to convert a QBF E with no free variables into a sequential circuit $\tilde{N}$. Note that, for each operator in the QBF, we need only a constant number of gates. For a QBF of length r, we get a circuit with at most $44r$ gates. We are not quite done, however. The final part of the circuit, shown in Figure 9.5, must be put "around" $\tilde{N}$ in order to correctly introduce a pulse into $\tilde{N}$ and also to interpret the final result. The idea is to use a NOR gate feeding back to itself as an oscillator. The only time a stable state can be reached in the complete circuit when the input changes from 0 to 1, occurs when the final latch gets the value 1. This can happen if and only if QBF E is *true*.

Our final task is to assign a starting state to this circuit. This is simply the state in which all the flip-flops are storing the value 0, and the input *in* is 0. The reader can easily verify that this total state is stable.

In summary, the construction described above takes an arbitrary QBF E of size r, with no free variables, and yields a gate circuit with at most $44r+19$ gates. If this circuit is started in the stable total state defined above, and the input changes from 0 to 1, then the circuit eventually reaches a stable state if and only if E is *true*. Hence, if we can solve the UD-SSR problem in polynomial time, then we can solve all problems in PSPACE in polynomial time. This, together with the fact that the UD-SSR problem is in PSPACE, implies that it is PSPACE-complete. □

Since the unit-delay version of SSR is PSPACE-complete and unit-delay analysis is a special case of nominal delay analysis, it follows trivially that the SSR problem for the nominal delay model is also PSPACE-complete.

In the models above we have knowledge of the exact sizes of the delays. In contrast to this, the GMW and XMW race models below use delays that are (almost) completely unknown. The complexity of the SSR problem for the GMW model depends on where we assume delays.

Theorem 9.2 *If the network has input-, gate-, and wire-delays, the SSR problem is solvable in polynomial time for the GMW race model.*

Proof: Since the network is complete, we can use the ternary simulation algorithm described in Chapter 7. To determine the answer to this SSR problem, it is sufficient to consider the result $\mathbf{s}^B$ of ternary simulation. If $\mathbf{s}^B$ is binary, then it follows by Theorem 7.2 that the circuit must eventually reach this state according to a GMW analysis. On the other hand, if $\mathbf{s}^B$ is not binary, then, again by Theorem 7.2, we can conclude that the circuit does not reach a unique binary state; thus it can oscillate and/or have a critical race. The polynomial time result follows from the fact that the ternary simulation algorithm produces, in the worst case, a sequence of $2m$ states, each requiring at most m excitation function evaluations. This, together with the assumption that the excitation functions can be evaluated in time polynomial in the size of the circuit, implies the required result. □

In view of Theorems 7.6 and 7.7, we can also conclude, using similar arguments, that the following theorem holds.

Theorem 9.3 *For any network with input delays, the SSR problem is solvable in polynomial time for the XMW race model.*

Surprisingly, if we gain more information about the delays—in particular, that there are no wire delays—the SSR problem for the GMW race model becomes intractable (assuming P $\neq$ NP) as the following theorem shows.

Theorem 9.4 *In the gate-delay model and in the input- and gate-delay model the SSR problem for the GMW race model is NP-hard.*

Proof: We prove the claim for the gate-delay model. The input- and gate-delay model result can be shown using virtually identical arguments and is left as an exercise for the interested reader. To prove NP-hardness, we

show how to transform the Boolean tautology problem to the GMW-SSR problem. The Boolean *nontautology* problem is defined as follows: Given a Boolean expression E over $\{x_1, \ldots, x_n\}$ using the connectives $\neg$, $\vee$, and $\wedge$, is E *not* a tautology, i.e., is there a truth assignment for the variables that makes E *false*. Cook [39] showed in 1971 that the nontautology problem is NP-complete. This result implies that the Boolean tautology problem, defined dually, is NP-hard. Hence, to prove that the GMW-SSR problem is NP-hard, it is sufficient to transform the Boolean tautology problem to the GMW-SSR problem.

Given any Boolean expression E over $\{x_1, \ldots, x_n\}$ using the connectives $\neg$, $\vee$, and $\wedge$, we show how to construct a circuit N with one input. When this circuit is started in a stable state (to be defined) and the input changes from 0 to 1, the outcome according to a GMW analysis consists of a single state if and only if E is a tautology. The basic idea is quite similar to the construction in [116]. We use a "race-generating" circuit, shown in Figure 9.6, for every input variable x_i. In Figure 9.7 we show a GMW

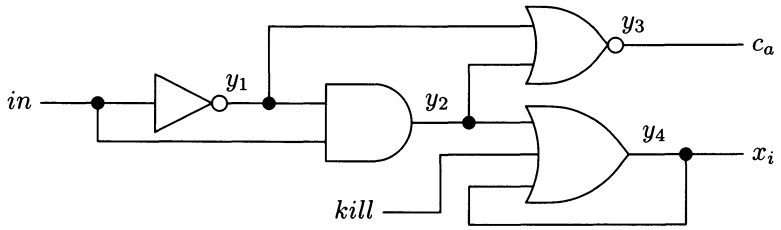


FIGURE 9.6. Critical-race generating circuit.

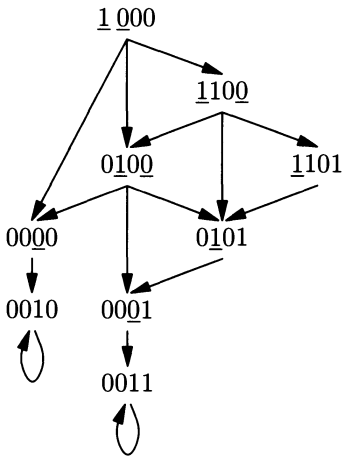


FIGURE 9.7. GMW analysis of the critical-race generating circuit.

analysis of the race-generating circuit when it is started in stable state $y = y_1 \ldots y_4 = 1000$, $kill = 0$ and in changes from 0 to 1. There are two important properties of the circuit: first, that there are two stable states reachable (0010 and 0011), and, second, that when the NOR gate changes from 0 to 1, all other gates in the circuit are stable. Thus, output c_a acts as a completion signal for this circuit.

Assume that E is a given Boolean expression. We first construct a subcircuit $\tilde{N}$ defined recursively as follows:

Basis: If $E = x_i$ then $\tilde{N}$ is the (trivial) circuit shown in Figure 9.8.

Induction step: Assume that circuits $\tilde{N}_1$ and $\tilde{N}_2$, corresponding to Boolean expressions $E_1(x)$ and $E_2(x)$, respectively, have been constructed.

1) If $E = \neg E_1(x)$ then $\tilde{N}$ is the circuit shown in Figure 9.9.
2) If $E = E_1(x) \wedge E_2(x)$ then $\tilde{N}$ is the circuit shown in Figure 9.10.
3) If $E = E_1(x) \vee E_2(x)$ then $\tilde{N}$ is the circuit shown in Figure 9.11.

The circuit $\tilde{N}$ has n inputs, labeled $x_1, \ldots, x_n$, a "compute" input c, an output *Out*, and a "compute-acknowledge" output c_a. We will design $\tilde{N}$ in such a way that the inputs $x_1, \ldots, x_n$ are stable by the time c changes. $\tilde{N}$ is started in the stable state implied by $c = 0$ and c changes to 1 at some

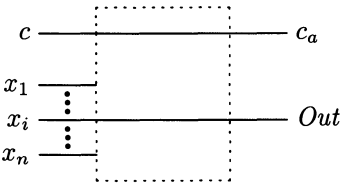


FIGURE 9.8. Circuit for $E = x_i$.

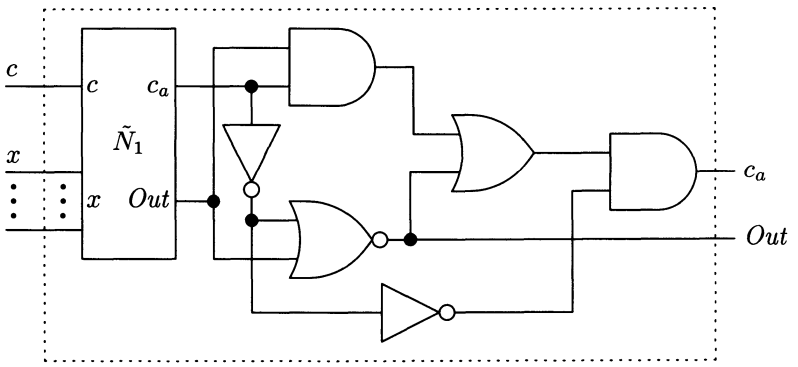


FIGURE 9.9. Circuit for $\neg E_1(x)$.

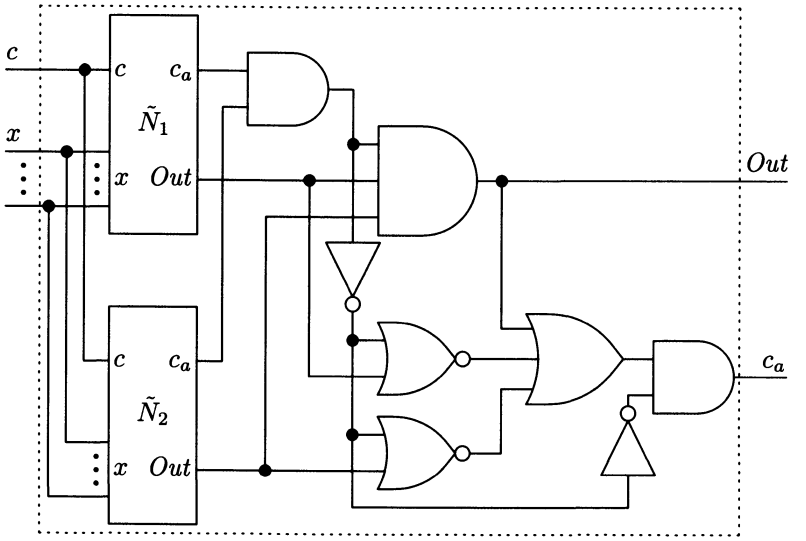


FIGURE 9.10. Circuit for $E_1(x) \wedge E_2(x)$.

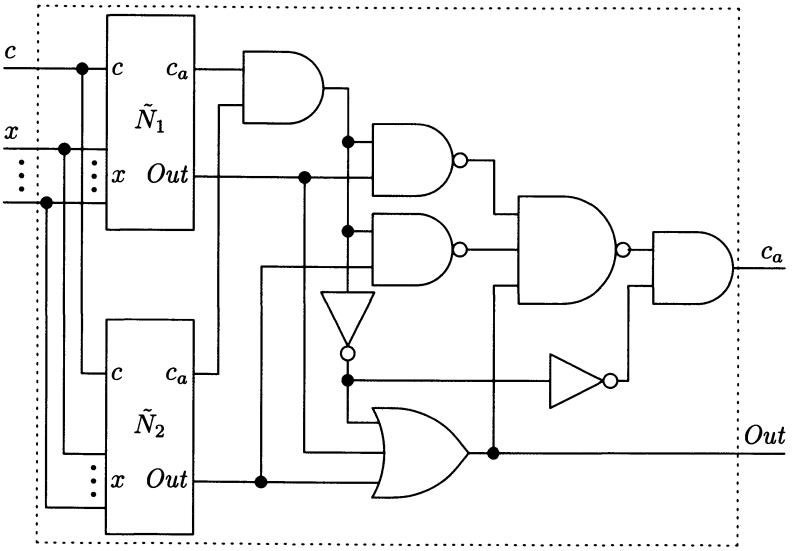


FIGURE 9.11. Circuit for $E_1(x) \vee E_2(x)$.

point in time. We claim that $\tilde{N}$ satisfies the following two properties: The output *Out* of $\tilde{N}$ has the value $E(x)$ when the c_a output changes from 0 to 1, and neither *Out* nor c_a changes again if the inputs are held fixed. We prove this claim by induction.

Basis: It is trivial to verify that the circuit of Figure 9.8 satisfies both properties.

Induction step: Assume that the claim holds for the circuits $\tilde{N}_1$ and $\tilde{N}_2$ corresponding to the Boolean expressions $E_1(x)$ and $E_2(x)$, respectively.

1) If $E = \neg E_1(x)$, we need to verify the claim for the circuit shown in Figure 9.9. Note that, if the c_a output of $\tilde{N}_1$ is 0, then the outputs of the gates shown in Figure 9.9 are uniquely determined. By the induction hypothesis, it is sufficient to perform a GMW analysis for the gates shown in Figure 9.9 for different values of *Out* from $\tilde{N}_1$ when the c_a signal of $\tilde{N}_1$ changes from 0 to 1. There are two cases to consider. First, suppose that *Out* of $\tilde{N}_1$ is 1. This implies that the NOR gate has the value 0 and remains stable no matter which other gates change. Hence, the output *Out* of $\tilde{N}$ is 0, i.e., has the same value as $\neg E_1(x)$. The output c_a of $\tilde{N}$ can change from 0 to 1 only when the c_a signal from $\tilde{N}_1$ has propagated through both inverters, the topmost AND gate, the OR gate, and the last AND gate. It follows that c_a changes only once, and the claim holds. Second, consider the case when *Out* from $\tilde{N}_1$ is 0. In this case the topmost AND gate is stable with the value 0. The output c_a of $\tilde{N}$ can change from 0 to 1 only when the c_a signal from $\tilde{N}_1$ has propagated through both inverters, the NOR gate, the OR gate, and the final AND gate. It follows that *Out* of $\tilde{N}$ has the value 1 (i.e., the value of $\neg E(x)$) before c_a of $\tilde{N}$ changes, and that c_a changes only once. Hence the claim follows.

2) If $E = E_1(x) \wedge E_2(x)$, we need to verify the claim for the circuit shown in Figure 9.10. In order for c_a of $\tilde{N}$ to change from 0 to 1, the c_a signals from both $\tilde{N}_1$ and $\tilde{N}_2$ must have changed to 1, and this change must have propagated through the topmost AND gate and both inverters. If at least one of the *Out* outputs of $\tilde{N}_1$ and $\tilde{N}_2$ is 0, then the three-input AND gate must be stable with the value $0 = E_1(x) \wedge E_2(x)$. If both *Out* outputs are 1, then for c_a of $\tilde{N}$ to become 1 requires that the output of the three-input AND gate changes to 1 first. With these observations, it is straightforward to verify the claim.

3) If $E = E_1(x) \vee E_2(x)$, we need to verify the claim for the circuit of Figure 9.11. This is similar to case 2), and we leave the details to the interested reader.

In summary, $\tilde{N}$ is a circuit that evaluates $E(x)$ when the "compute" signal c changes to high. When the correct value is available on the *Out* output, the c_a output changes to high. Note that $\tilde{N}$ works correctly no matter what the gate delays are (as long as they are finite).

Finally, in Figure 9.12, we show how to use n race-generating circuits and $\tilde{N}$ to construct N. The basic idea is as follows. The circuit is started in the stable state in which $in = 0$ and all OR gates with self-loops have the value 0. There is one race-generating circuit for each input variable to E. When in changes from 0 to 1, each of these race-generating circuits settles down with either the value 0 or 1 on its output x_i. Since the c_a outputs of all the race-generating circuits are connected via a chain of AND gates, the c input to $\tilde{N}$ does not change to 1 until all the race-generating circuits have reached a stable state. There are two cases to consider: If $E(x)$ is a tautology, then—irrespective of the state of each input variable—the output *Out* of $\tilde{N}$ is 1 when its c_a output changes to high. This implies that the last AND gate eventually changes to 1 causing the right-most OR gate also to change to 1. Once this happens, the OR gate remains stable. After the output of the OR gate changes to high, the oscillation in the NAND gate eventually stops. Furthermore, this sets all the input variables to 1 (since the *kill* signal causes the OR gates with self-loops in the race-generating circuits to change to 1). This uniquely determines the values of all the gates in $\tilde{N}$ and of the right-most AND gate. Hence, if $E(x)$ is a tautology, the circuit N eventually reaches a unique stable state.

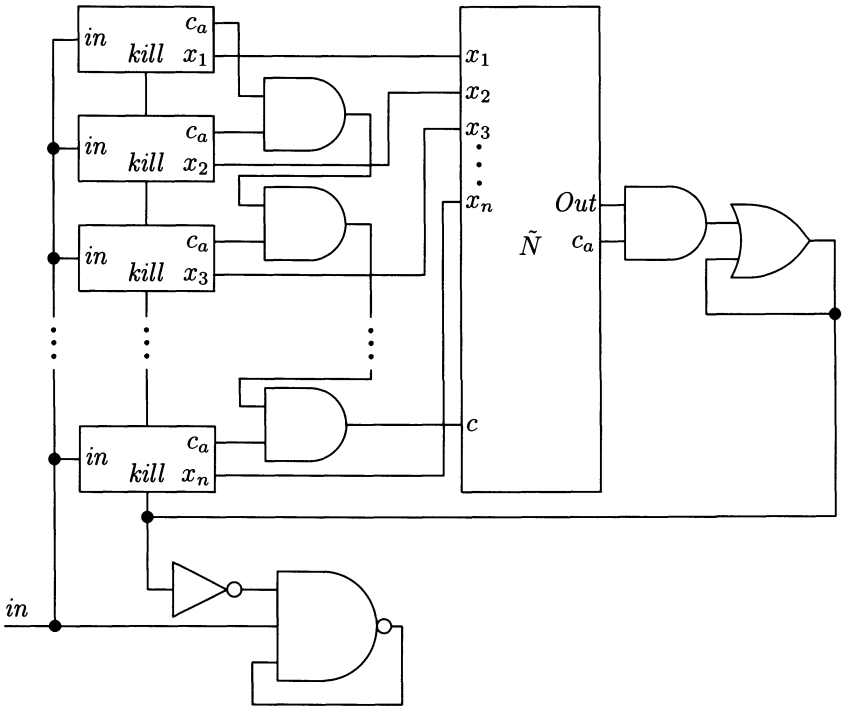


FIGURE 9.12. Complete circuit for Theorem 9.4.

On the other hand, assume $E(x)$ is *not* a tautology. Then, for some Boolean vector $\alpha \in \{0,1\}^n$, $E(\alpha)$ is *false*. Since each race-generating circuit can reach the state 0 or 1 independently, there must exist some delay assignment such that $x_1 = \alpha_1$, $x_2 = \alpha_2, \ldots$, $x_n = \alpha_n$. For this delay assignment, the result on *Out* from $\tilde{N}$ is 0 when its c_a output changes from 0 to 1. This implies that the right-most AND gate, and therefore also the right-most OR gate, will never change. Thus the circuit never reaches a stable state, since the NAND gate continues to oscillate. In summary, if $E(x)$ is not a tautology, then the outcome of the GMW analysis of N contains more than one state.

Because the circuit N contains $O(l(E))$ gates, where $l(E)$ denotes the length of the Boolean expression, E is a tautology if and only if the outcome of the GMW analysis of N contains a single state. Since the Boolean tautology problem is NP-hard, so is the GMW-SSR problem. □

Next we consider the inherent complexity of the SSR question for the bi-bounded race models of Chapter 8.

Theorem 9.5 *The SSR problem is NP-hard for the bi-bounded binary delay race model.*

Proof: The proof parallels the proof of Theorem 9.4 with the addition that each gate used in the construction is assumed to have delays bounded by $[1, 2)$. For these delay bounds, it is easy to see that the race-generating circuit of Figure 9.6 still satisfies the two crucial properties, i.e., that output y_4 can end up being either 0 or 1, and that output y_3 changes from 0 to 1 after all the other gates in the circuit have reached a stable state. Also, it is easy to verify that the construction used in the remainder of the proof ensures that the circuit reaches a unique stable binary state if and only if the expression E is a tautology. □

The complexity of the SSR problem for the extended bi-bounded delay race model is less clear. For the special case in which the delay bounds can grow with the size of the circuit, we have the following result:

Theorem 9.6 *If the lower delay bounds of the gates are of size $\Omega(3^m)$, where m is the size of the circuit, the SSR problem for the extended bi-bounded delay model is PSPACE-hard.*

Proof: We show how to transform the unit-delay SSR problem into the general extended bi-bounded delay SSR problem. In view of Theorem 9.1, the result then follows immediately. For convenience, we refer to the two SSR problems as the UD-SSR and XBD-SSR problems. Let N be the original network, containing m state variables, for which we try to answer the unit-delay SSR question. Transform N to $\tilde{N}$ by adding a wire delay in every edge of N. This can increase the circuit size by at most $O(m^2)$ state variables. Let the delay $\delta_j(t)$ in every variable be bounded as follows: $(D-1) \leq \delta_j(t) < D$, for $D = 2 \times 3^m + 2$. We now argue that the answer to

the XBD-SSR problem for $\tilde{N}$ is yes if and only if the answer to the UD-SSR problem for N is yes. Note that all the delays have the same bounds in $\tilde{N}$—thus they could be exactly the same—and there is a delay in every wire. It follows that, if there is an oscillation in the UD analysis of N, then the same oscillation must exist in an XBD analysis of $\tilde{N}$. Hence, if the answer to the UD-SSR question for N is no, then so is the answer to the XBD-SSR question for $\tilde{N}$. If the answer to the UD-SSR question for N is yes, then the sequence of states computed by the UD analysis of N must be of length less than or equal to 3^m (and the last state must be stable). This implies that the unit-delay analysis of $\tilde{N}$ also reaches the same stable state, but that the length of the sequence is less than or equal to 2×3^m—every odd state corresponding to an input or gate variable change and every even state corresponding to a wire variable change. On the other hand, it is easy to convince oneself that the XBD analysis of $\tilde{N}$ yields the same result as the UD analysis of $\tilde{N}$, as long as we do not carry out the analysis for too many steps. In fact, the two correspond exactly for r steps, if the time to complete $r+1$ of the fastest possible transitions is strictly greater than the time to complete r of the slowest possible transitions. In other words, the UD and XBD analyses of $\tilde{N}$ correspond exactly for r steps as long as $(r+1)(D-1) > rD$, i.e., as long as $r < D-1$. Since $D = 2 \times 3^m + 2$, we can conclude that the XBD analysis of $\tilde{N}$ corresponds exactly to the UD analysis of $\tilde{N}$, if the length of the UD sequence is at most 2×3^m. Altogether, we have shown that, if the answer to the UD-SSR question for N is yes, then so is the answer to the XBD-SSR question for $\tilde{N}$. □

The assumption that the maximum delay can grow exponentially fast with the size of the network was crucial in the proof of Theorem 9.6. This is a very unrealistic assumption. A more interesting question is the complexity of the XBD-SSR problem when the maximum delay is constant or can grow only polynomially in the size of the circuit. The complexity of this restricted XBD-SSR problem is still open.

In Table 9.1 we summarize the results of this section. The following observations can be made: When the delay in each component is known exactly or almost exactly, the SSR problem is intractable (assuming PSPACE$\neq$ PTIME, of course). On the other hand, when the delays can be arbitrary and changes can go through Φ (i.e., when the XMW model is used), the SSR problem can be solved very efficiently. It is interesting to note that, for the GMW model, the difficulty of the SSR problem depends on whether both wires and gates or only gates have delays. In the former case the GMW-SSR problem is solvable in polynomial time, whereas in the latter case it is NP-hard. For the binary bi-bounded delay models, the SSR problem is intractable. In general, the SSR problem is intractable for all binary race models, except the GMW model on complete networks.

The results of this section are rather negative, showing that the SSR problem is intractable for many realistic race and delay models. In the next

TABLE 9.1. Complexity of the stable-state reachability problem.

Race model	Complexity
UD	PSPACE-complete
BD	NP-hard
XBD	PSPACE-hard
restricted XBD	unknown
XMW	polynomial time
GMW[1]	polynomial time
GMW[2]	NP-hard
[1] assuming both gate and wire delays.	
[2] assuming gate delays only.	

section we study the more practical version of the SSR problem, where we impose a condition on the length of time required for the circuit to reach a stable state.

9.2 Limited Reachability

The *limited reachability* (LR) problem is to determine whether a network with m state variables reaches a unique binary stable state within time $r(m)$ for some function r. If r is exponential in m, the LR problem degenerates into essentially the SSR problem. Thus we henceforth assume that $r(m)$ is $O(m^k)$ for some constant $k > 0$.

Theorem 9.7 *The limited reachability problem is solvable in polynomial time for the unit-delay model and the extended bi-bounded delay model.*

Proof: For the unit-delay model, simulate the circuit for r steps. Each step requires at most m excitation-function evaluations, each using time at most polynomial in m. Since r is assumed to be bounded by some polynomial in m, the first claim follows immediately. For the extended bi-bounded delay model, carry out r steps of the TBD algorithm. By Theorem 8.3, the result of the TBD algorithm is equal to the *lub* of all the states the network can be in at time r. If this result is a binary stable state, the answer to the LR problem is yes, otherwise it is no. Each step of the TBD algorithm requires at most $2m$ excitation function evaluations plus some bookkeeping. Since r is assumed to be bounded by some polynomial in m, the claim follows. □

Our next two results are perhaps more of theoretical than practical interest, but they illustrate how different assumptions made about the network affect the complexity of the limited reachability problem. First we consider a constant fan-in network and a constant r.

Theorem 9.8 *If the maximum indegree of a state vertex is some constant c, and $r(m) = r_0$ for some constant r_0, then the limited reachability problem is solvable in time polynomial in the size of the circuit for the binary bi-bounded delay model.*

Proof: Assume that d is the minimum delay of any vertex in the network. The crucial observation is that an input change can propagate through at most $\frac{r_0}{d}$ vertices when each vertex has a delay of at least d units and we want to know the value of a vertex after r_0 time units. Hence, to determine whether a vertex i will have a unique binary value and be stable at time r_0, it is sufficient to consider the values on the vertices that are within distance $\frac{r_0}{d}$ from vertex i. Since we assumed a constant maximum indegree of the state vertices this implies that there are at most

$$\frac{c^{\frac{r_0}{d}+1} - 1}{c - 1}$$

vertices that can affect the value and excitation of vertex i at time r_0. In other words, to determine vertex i's value and excitation at time r_0, it is sufficient to perform an extended bounded delay analysis on a subnetwork containing a constant (though normally very large) number of vertices. Thus, the computational effort required to determine the value and excitation of vertex i at time r_0 does not depend on the size m of the complete network. Using this procedure for each vertex in N immediately gives the answer to the LR problem: If each vertex reaches a unique stable state then the answer to the LR problem is yes, otherwise it is no. We leave the details of the proof to the interested reader. □

In the following theorems we consider the complexity of the LR problem for the binary bi-bounded delay model with more liberal conditions on the maximum indegree of state vertices or on the number of time units we are willing to wait.

Theorem 9.9 *If the maximum indegree of a state vertex is $\Omega(m)$ and $r(m)$ is $\Omega(1)$, the limited reachability problem is NP-hard for the binary bi-bounded delay model.*

Proof: We transform the 3-satisfiability problem into the LR problem. Given a Boolean formula E in 3-conjunctive normal form (i.e., a formula that is a product of clauses, each clause containing three literals), we show how to construct a network N with one input. If N is started in a stable total state (to be defined) and the input changes from 0 to 1, then a binary bi-bounded delay analysis of this transition, carried out for at least 64 time units, indicates that N reaches a single stable state if and only if E is not satisfiable. In other words, the answer to the LR problem for N is yes if and only if E is not satisfiable. Since the size of N is polynomial in the size of E, and the 3-satisfiability problem is NP-complete [39], it follows that the LR problem is NP-hard.

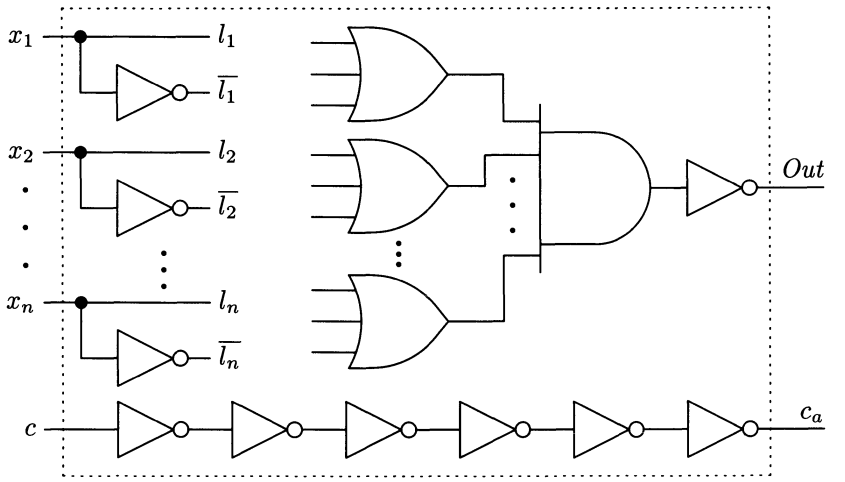


FIGURE 9.13. Construction of $\tilde{N}$ for Theorem 9.9.

The construction of N is very similar to the construction in the proof of Theorem 9.4. We use the race-generating circuit of Figure 9.6. However, we now assume that the delay in each gate is bounded by $[3, 4)$. The reader can easily verify that, if *in* changes from 0 to 1 at time 0, then, according to the bi-bounded delay model, the race-generating circuit can only reach stable states 0010 or 0011; in either case, the value of x_i is stable before the c_a signal changes from 0 to 1. Also, it is easy to verify that c_a and x_i are guaranteed to be stable at time 12 and 8, respectively.

Let E be a given Boolean expression in 3-conjunctive normal form. We first construct a subcircuit $\tilde{N}$ that evaluates the complement of E. In Figure 9.13 we outline how such a circuit can be constructed given that all the gates have delay bounds $[3, 4)$. First the inverters compute $\overline{l}_i$, then the OR gates compute the values of the clauses, the multi-input AND gate computes the product of all the clauses, and the topmost inverter complements the result. The row of inverters at the bottom is used to delay the "compute" signal c so that the c_a signal cannot change from 0 to 1 until the value of $\neg E(x)$ is available on *Out*. If the various x_is are stable by the time c changes from 0 to 1, it is easy to verify that c_a does not change from 0 to 1 before *Out* has taken on the value of $\neg E(x)$.

Finally, in Figure 9.14 we show how to use n race-generating circuits and $\tilde{N}$ to construct N. Again, all the gates are assumed to have delay bounds $[3, 4)$. The basic idea is as follows. The circuit is started in the stable state in which $in = 0$ and all the OR gates with self-loops have the value 0. There is one race-generating circuit for each input variable to E. When *in* changes from 0 to 1, each of these circuits settles down with either the value 0 or 1 on its output. Since the c_a outputs of all these circuits are connected to

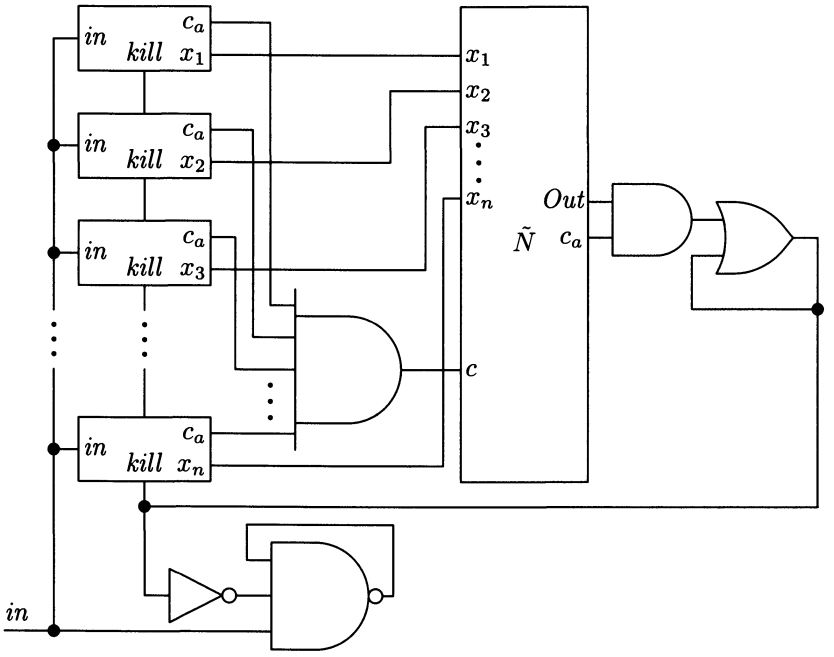


FIGURE 9.14. Complete circuit for Theorem 9.9.

an AND gate, it follows that the c input to $\tilde{N}$ does not change to 1 until all the race-generating circuits have reached a stable state. There are two cases to consider: If $E(x)$ is not satisfiable, then, irrespective of the state of each input variable, the output *Out* of $\tilde{N}$ is 1 when c_a rises in $\tilde{N}$. This implies that the last AND gate changes to 1, causing the right-most OR gate to change to 1 also. Once the OR gate changes to 1, it remains stable. When the output of this OR gate rises, the oscillation in the NAND gate stops. Furthermore, all the input variables will be set to 1 (since the *kill* signal causes the OR gates with self-loops in the race-generating circuits to change to 1). This uniquely determines the values of all the gates in $\tilde{N}$ and of the right-most AND gate. Hence, if $E(x)$ is not satisfiable, the circuit N terminates in a unique stable state. In fact, it is easy to verify that N takes at most 64 time units to reach this stable state.

In case $E(x)$ is satisfiable there must exist some Boolean vector $\alpha \in \{0,1\}^n$, such that $E(\alpha)$ is true. Since each race-generating circuit can reach the state 0 or 1 independently, there must exist some delay assignment such that $x_1 = \alpha_1, x_2 = \alpha_2, \ldots, x_n = \alpha_n$. For this delay assignment, the result on *Out* from $\tilde{N}$ is 0 when c_a changes from 0 to 1 on $\tilde{N}$. This implies that the right-most AND gate never changes; consequently, neither does the right-most OR gate. Since the NAND gate can continue to oscillate, the circuit never reaches a stable state. In summary, if $E(x)$ is not satisfiable, the

outcome of the binary bi-bounded race analysis carried out for at least 64 time steps contains more than one state.

We have shown that E is not satisfiable if and only if the outcome of the binary bi-bounded delay analysis of N, carried out for at least 64 time units, contains a single state, where the circuit N contains $O(l(E))$ gates, $l(E)$ being the size of the Boolean expression. Since the 3-satisfiability problem is NP-complete, it follows that the LR problem for the binary bi-bounded delay model is NP-hard. □

Although the unbounded indegree assumption above is not very realistic, it allowed us to compute the AND of $O(m)$ signals in one gate delay. A more realistic assumption is that the indegree of every state vertex is bounded by some relatively small constant. It is then quite reasonable to allow a circuit $\Omega(\log m)$ time units to settle down. Consider, for example, a one-output combinational network having m vertices. Such a network may take $O(\log m)$ time units to reach stability. If we assume bounded indegree and allow a (relatively slowly) growing $r(m)$ we get the following result:

Theorem 9.10 *If the maximum indegree of a state vertex is c for some constant c, and if r is $\Omega(\log m)$, then the limited reachability problem is NP-hard for the binary bi-bounded delay model.*

Proof: The proof is similar to the proof of Theorem 9.9, except that the big AND gates are replaced by trees of c-input AND gates. Since the heights of these trees are bounded by $O(\log m)$, it is easy to verify that, if E is not satisfiable, circuit N reaches a unique stable state within $O(\log m)$ time units. On the other hand, if E is satisfiable, then N can reach an oscillation. □

TABLE 9.2. Complexity of the limited reachability problem.

Race model	Complexity
UD	polynomial time
BD[1]	polynomial time
BD[2]	NP-hard
BD[3]	NP-hard
XBD	polynomial time
[1] assuming bounded indegree and $r(m) = r_0$.	
[2] assuming unbounded indegree and that $r(m)$ is $\Omega(1)$.	
[3] assuming bounded indegree and that $r(m)$ is $\Omega(\log m)$.	

In Table 9.2 we summarize the results of this section. The LR problem is solvable in time polynomial in the size of the circuit for the unit-delay model and the extended bi-bounded race model. For binary race models, even the LR problem is almost always intractable.

Chapter 10

Regular Languages and Finite Automata

This chapter provides an introduction to the theory of regular languages and finite automata. These concepts will be used in the following chapters for describing behaviors of asynchronous circuits and for specifying abstract behaviors. The theory of regular languages and finite automata is now well established as one of the important basic tools of computer science. We present this theory in a somewhat different way than is done in most textbooks, because we feel that our approach is more general and permits us to establish the relationship between regular languages and finite automata in a very natural way. For a general introduction to regular languages and finite automata, see one of the many texts on this subject, for example [65, 77, 100, 120]; for material closer to our treatment see [16, 17, 20].

10.1 Regular Languages

10.1.1 Semigroups

A *semigroup* is an algebraic system $S = \langle \mathcal{X}, \circ \rangle$, where $\mathcal{X}$ is a set and $\circ$ is a binary operation on $\mathcal{X}$ that satisfies the associative law

$$x \circ (y \circ z) = (x \circ y) \circ z,$$

for all $x, y, z \in \mathcal{X}$. The binary operation is often called multiplication, and the symbol $\circ$ is usually omitted. If $\mathcal{Y}$ is a subset of $\mathcal{X}$, then $\langle \mathcal{Y}, \circ \rangle$ is a *subsemigroup* of S if $\mathcal{Y}$ is closed under $\circ$, i.e., if $x \circ y$ is in $\mathcal{Y}$, for all $x, y \in \mathcal{Y}$.

A *monoid* is an algebraic system $M = \langle \mathcal{X}, \circ, 1_M \rangle$, where $\langle \mathcal{X}, \circ \rangle$ is a semigroup and 1_M is an element of $\mathcal{X}$, called *unit*, that satisfies the unit law

$$x \circ 1_M = 1_M \circ x = x,$$

for all $x \in \mathcal{X}$. It is easily verified that a semigroup can have only one element satisfying the unit law, i.e., the unit element is unique. If $\mathcal{Y}$ is a subset of $\mathcal{X}$, then $\langle \mathcal{Y}, \circ, 1_M \rangle$, is a *submonoid* of M if $1_M \in \mathcal{Y}$ and $\mathcal{Y}$ is closed under $\circ$.

As an example, consider any set $\mathcal{X}$. Then the systems $\langle \mathcal{P}(\mathcal{X}), \cup, \emptyset \rangle$ and $\langle \mathcal{P}(\mathcal{X}), \cap, \mathcal{X} \rangle$ are both monoids. Also, if $\mathcal{N}$ is the set of nonnegative integers, then $\langle \mathcal{N}, +, 0 \rangle$ and $\langle \mathcal{N}, \times, 1 \rangle$ are monoids, where $+$ and $\times$ denote ordinary addition and multiplication.

10.1.2 Languages

Let Σ be a nonempty set. We normally assume that the set Σ is a fixed, finite set; we refer to it as an *alphabet* and to the elements of Σ as *letters*. Any finite sequence of letters is called a *word*. For example, if $\Sigma = \{a, b\}$, then a and $babb$ are words. Note that we make no distinction between a letter and the word consisting of that letter; the meaning is clear from the context. The number of letters in a word x is called its *length* and is denoted by $|x|$. Thus $|a| = 1$ and $|babb| = 4$. A word of length n may be viewed as an ordered n-tuple of letters. It is also convenient to introduce the 0-tuple of letters, called the *empty word* and denoted by ε. Note that $|\varepsilon| = 0$.

Given two words $x = a_1 \dots a_n$ and $y = b_1 \dots b_m$, we define the *product* or *concatenation* of x and y to be the word $xy = a_1 \dots a_n b_1 \dots b_m$. It is clear that concatenation is associative, and that the empty word ε acts as a unit since $\varepsilon x = x\varepsilon = x$, for any word x. Let Σ^* be the set of all the words, including the empty word ε, over alphabet Σ. It follows that $\langle \Sigma^*, \mathit{concatenation}, \varepsilon \rangle$ is a monoid. This monoid of all the words over Σ is called the *free monoid generated by* Σ.

If w is a word, we denote the concatenation of n copies of w by w^n. If $n = 0$, we have $w^0 = \varepsilon$ for all w. Thus, for example, the infinite set of words $\mathcal{X} = \{b, ab, aab, \dots\}$ is conveniently denoted by $\{a^n b \mid n \geq 0\}$, i.e., it is the set of all the words of the form "zero or more a's followed by b."

A *language* over an alphabet Σ is any subset of Σ^*, i.e., any set of words. Given a family of languages over Σ, we can form new languages by applying certain operations. To begin with, we have the usual set operations such as union, intersection, and complement with respect to Σ^*. Other operations arise naturally because languages are subsets of a special universal set Σ^*, which is a monoid. Thus we can extend the operation of concatenation from Σ^* to $\mathcal{P}(\Sigma^*)$, the set of all languages over Σ, as follows: For $\mathcal{X}, \mathcal{Y} \subseteq \Sigma^*$,

$$\mathcal{X}\mathcal{Y} = \{w \mid w = xy, x \in \mathcal{X}, y \in \mathcal{Y}\}.$$

It is easily verified that concatenation of languages is associative and that the language $\{\varepsilon\}$ satisfies

$$\{\varepsilon\}\mathcal{X} = \mathcal{X}\{\varepsilon\} = \mathcal{X},$$

for all $\mathcal{X} \subseteq \Sigma^*$. Thus the system $\langle \mathcal{P}(\Sigma^*), \mathit{concatenation}, \{\varepsilon\} \rangle$ is again a monoid.

To illustrate concatenation of languages, let $\Sigma = \{a, b\}$, $\mathcal{X} = \{bab, baba\}$ and $\mathcal{Y} = \{\varepsilon, a, bb\}$; then $\mathcal{X}\mathcal{Y} = \{bab, baba, babaa, babbb, bababb\}$. Note that a word in a product may be generated in more than one way; for example,

$baba = (baba)(\varepsilon) = (bab)(a)$. Some basic properties of concatenation of languages are given below; the concatenation operator has precedence over union and intersection.

C1	$\mathcal{X}(\mathcal{Y}\mathcal{Z}) = (\mathcal{X}\mathcal{Y})\mathcal{Z}$,		
C2	$\mathcal{X}\{\varepsilon\} = \mathcal{X}$,	*C2'*	$\{\varepsilon\}\mathcal{X} = \mathcal{X}$,
C3	$\mathcal{X}\emptyset = \emptyset$,	*C3'*	$\emptyset\mathcal{X} = \emptyset$,
C4	$\mathcal{X}(\mathcal{Y} \cup \mathcal{Z}) = \mathcal{X}\mathcal{Y} \cup \mathcal{X}\mathcal{Z}$,	*C4'*	$(\mathcal{Y} \cup \mathcal{Z})\mathcal{X} = \mathcal{Y}\mathcal{X} \cup \mathcal{Z}\mathcal{X}$,
C5	$\mathcal{X}(\mathcal{Y} \cap \mathcal{Z}) \subseteq \mathcal{X}\mathcal{Y} \cap \mathcal{X}\mathcal{Z}$,	*C5'*	$(\mathcal{Y} \cap \mathcal{Z})\mathcal{X} \subseteq \mathcal{Y}\mathcal{X} \cap \mathcal{Z}\mathcal{X}$.

The following example shows that *C5* cannot be strengthened to an equality. Let $\mathcal{X} = \{\varepsilon, a\}$, $\mathcal{Y} = \{aa\}$, and $\mathcal{Z} = \{a\}$. Then $\mathcal{X}(\mathcal{Y} \cap \mathcal{Z}) = \emptyset$, but $\mathcal{X}\mathcal{Y} \cap \mathcal{X}\mathcal{Z} = \{aa\}$. A similar example shows that *C5'* cannot be strengthened.

We next define two closely related unary operations on languages. For $\mathcal{X} \subseteq \Sigma^*$, the language

$$\mathcal{X}^+ = \bigcup_{n \geq 1} \mathcal{X}^n$$

is the *subsemigroup of* Σ^* *generated by* $\mathcal{X}$. Thus $\mathcal{X}^+$ is the set of all the words of the form $w = x_1 \ldots x_n, n \geq 1, x_i \in \mathcal{X}, i = 1, \ldots, n$. Similarly, the language

$$\mathcal{X}^* = \bigcup_{n \geq 0} \mathcal{X}^n$$

is the *submonoid of* Σ^* *generated by* $\mathcal{X}$.

We refer to the $^+$ and * operations as *plus* and *star*, respectively. Some properties of these operations are listed below.

P1	$\mathcal{X}^+ = \mathcal{X}\mathcal{X}^*$,	*S1*	$\mathcal{X}^* = \mathcal{X}^+ \cup \{\varepsilon\}$,
P2	$(\mathcal{X}^+)^+ = \mathcal{X}^+$,	*S2*	$(\mathcal{X}^*)^* = X^*$,
P3	$\emptyset^+ = \emptyset$,	*S3*	$\emptyset^* = \{\varepsilon\}$,
P4	$\{\varepsilon\}^+ = \{\varepsilon\}$,	*S4*	$\{\varepsilon\}^* = \{\varepsilon\}$,
P5	$\mathcal{X}^+\mathcal{X} = \mathcal{X}\mathcal{X}^+$,	*S5*	$\mathcal{X}^*\mathcal{X} = \mathcal{X}\mathcal{X}^*$,
P6	$\mathcal{X} \subseteq \mathcal{Y}$ implies $\mathcal{X}^+ \subseteq \mathcal{Y}^+$,	*S6*	$\mathcal{X} \subseteq \mathcal{Y}$ implies $\mathcal{X}^* \subseteq \mathcal{Y}^*$.

10.1.3 Regular Languages

We now restrict our attention to a particular family of languages, namely to the family $\mathcal{REG}$ of "regular" languages; this turns out to be precisely the family of languages recognizable by finite automata.

First, given an alphabet Σ, we define a *letter language* to be any language $\mathcal{L}$ consisting of a single word of length one, i.e., $\mathcal{L} = \{a\}$, where $a \in \Sigma$. The family $\mathcal{REG}_\Sigma$ of regular languages over Σ is the smallest family of languages

containing the letter languages, and closed under union, complementation, concatenation, and star. Thus we have

Definition 10.1 *The family* $\mathcal{REG}_\Sigma$ *of* regular languages *over* Σ *is defined inductively as follows:*

Basis: $\{a\} \in \mathcal{REG}_\Sigma$ *for each* $a \in \Sigma$.

Induction step: *If* $\mathcal{X}$ *and* $\mathcal{Y}$ *are in* $\mathcal{REG}_\Sigma$, *then so are* $\mathcal{X} \cup \mathcal{Y}$, $\overline{\mathcal{X}}$, $\mathcal{XY}$, *and* $\mathcal{X}^*$.

Every language in $\mathcal{REG}_\Sigma$ *can be constructed by a finite number of applications of the two rules above.*

In view of the fact that $\mathcal{X} \cap \mathcal{Y} = \overline{\overline{\mathcal{X}} \cup \overline{\mathcal{Y}}}$, the family $\mathcal{REG}_\Sigma$ is closed under intersection. Similarly, $\mathcal{REG}_\Sigma$ is closed under the difference operation and the symmetric difference operation.

Suppose the alphabet is $\Sigma = \{a, b\}$; then the following are examples of regular languages over Σ:

- $\mathcal{R}_1 = (\{a\} \cup \{b\})^* = \{a\} \cup \overline{\{a\}} = \Sigma^*$—the set of all words over the alphabet Σ.
- $\mathcal{R}_2 = \overline{\Sigma^*} = \emptyset$—the empty language.
- $\mathcal{R}_3 = (\overline{\Sigma^*})^* = \emptyset^* = \{\varepsilon\}$—the language consisting of the empty word.
- $\mathcal{R}_4 = \{a\}\{b\}\{a\}(\{a\} \cup \{b\})^*$—the set of all words that begin with *aba*.
- $\mathcal{R}_5 = (\{a\} \cup \{b\})^*\{a\}\{a\}(\{a\} \cup \{b\})^*$—the set of all words that contain two consecutive a's.
- $\mathcal{R}_6 = \{a\}\Sigma^* \cap \Sigma^*\{b\} \cap \overline{\Sigma^*\{b\}\{b\}\Sigma^*}$—the set of all words that begin with a, end with b, and do not contain two consecutive b's.
- $\mathcal{R}_7 = (\{a\} \cup \{b\}\{a\}^*\{b\})^*$—the set of all words that have an even number of b's.

10.1.4 Quotients of Languages

We now introduce the notion of "left quotient," or simply "quotient," of a language by a word. This notion will play a key role in the characterization of regular languages.

Let $\mathcal{X} \subseteq \Sigma^*$ be a language and let $w \in \Sigma^*$ be a word. The *(left) quotient* of $\mathcal{X}$ by w is denoted by $w^{-1}\mathcal{X}$, and is defined by

$$w^{-1}\mathcal{X} = \{x \mid wx \in \mathcal{X}\}.$$

The quotient $w^{-1}\mathcal{X}$ may be viewed as the set of all words that can "follow w in $\mathcal{X}$:" To construct $w^{-1}\mathcal{X}$ we can first take the set $\mathcal{Y}$ of all the words of $\mathcal{X}$ that begin with w. (Note that w itself also begins with w.) If w is then removed from each word wx in $\mathcal{Y}$ leaving the word x, the set of words so obtained is precisely $w^{-1}\mathcal{X}$. In a sense, this is a division of $\mathcal{X}$ by w on the left, hence the name "left quotient." As an example, consider $\mathcal{X} = \{ba, aba, abb\}$. Then $a^{-1}\mathcal{X} = \{ba, bb\}$ and $(ba)^{-1}\mathcal{X} = \{\varepsilon\}$.

We often need to determine whether a given regular language contains the empty word. This can be done with the aid of the δ operator defined by

$$\delta(\mathcal{X}) = \begin{cases} \{\varepsilon\} & \text{if } \varepsilon \in \mathcal{X}, \\ \emptyset & \text{otherwise.} \end{cases}$$

The use of the left quotient and the δ operator permits us to determine whether an arbitrary word w is in $\mathcal{X}$, because

$$w \in \mathcal{X} \text{ if and only if } \delta(w^{-1}\mathcal{X}) = \{\varepsilon\}.$$

Some basic properties of left quotients with respect to letters are given below. For all $a, b \in \Sigma, a \neq b$ and $\mathcal{X}, \mathcal{Y} \subseteq \Sigma^*$, we have

$$a^{-1}\emptyset = a^{-1}\{\varepsilon\} = a^{-1}\{b\} = \emptyset,$$

$$a^{-1}\{a\} = \{\varepsilon\},$$

$$a^{-1}\Sigma^* = \Sigma^*,$$

$$a^{-1}(\mathcal{X} \cup \mathcal{Y}) = (a^{-1}\mathcal{X}) \cup (a^{-1}\mathcal{Y}),$$

$$a^{-1}(\mathcal{X} \cap \mathcal{Y}) = (a^{-1}\mathcal{X}) \cap (a^{-1}\mathcal{Y}),$$

$$a^{-1}\overline{\mathcal{X}} = \overline{a^{-1}\mathcal{X}},$$

$$a^{-1}(\mathcal{X} - \mathcal{Y}) = (a^{-1}\mathcal{X}) - (a^{-1}\mathcal{Y}),$$

$$a^{-1}(\mathcal{X}\Delta\mathcal{Y}) = (a^{-1}\mathcal{X})\Delta(a^{-1}\mathcal{Y}),$$

$$a^{-1}(\mathcal{X}\mathcal{Y}) = (a^{-1}\mathcal{X})\mathcal{Y} \cup \delta(\mathcal{X})(a^{-1}\mathcal{Y}),$$

$$a^{-1}\mathcal{X}^* = (a^{-1}\mathcal{X})\mathcal{X}^*.$$

The verification of these properties is straightforward. The following observations permit us to calculate the quotients of a language with respect to arbitrary words:

$$\varepsilon^{-1}\mathcal{X} = \mathcal{X},$$

and for all $w \in \Sigma^*$ and $a \in \Sigma$:

$$(wa)^{-1}\mathcal{X} = a^{-1}(w^{-1}\mathcal{X}).$$

We will show that a language is regular if and only if it has a finite number of distinct quotients. Before we do this, however, we will simplify our notation for regular languages by introducing regular expressions.

10.2 Regular Expressions

10.2.1 Extended Regular Expressions

In view of Definition 10.1, the only way we have for representing a regular language $\mathcal{X}$ consists of specifying how $\mathcal{X}$ is formed from the letter languages by the application of union, complement, concatenation, and star. This amounts to writing an expression for $\mathcal{X}$. We formalize this as follows:

Definition 10.2 *The family $\mathcal{ERX}_\Sigma$ of* extended regular expressions *over an alphabet Σ is defined inductively as follows:*

Basis: *$\emptyset$, ε, and each $a \in \Sigma$ are in $\mathcal{ERX}_\Sigma$.*

Induction step: *If X and Y are in $\mathcal{ERX}_\Sigma$, then so are $(X \cup Y)$, $\overline{X}$, (XY), and X^*.*

Every extended regular expression in $\mathcal{ERX}_\Sigma$ can be constructed by a finite number of applications of the two rules above.

At this point, extended regular expressions are strings of symbols formed by starting with some basic symbols and combining these symbols with the use of a finite number of operations, as specified in the induction step. For example, $(((a \cup b^*)(aa))^* \cup \bar{b})$ is an extended regular expression, whereas $a \cup b$ and a^b are not. Of course, we intend to use extended regular expressions to denote regular languages. The two concepts are related below.

Definition 10.3 *The mapping $L : \mathcal{ERX}_\Sigma \to \mathcal{REG}_\Sigma$ is defined inductively as follows:*

Basis: $L(\emptyset) = \emptyset$, $L(\varepsilon) = \{\varepsilon\}$, *and for each* $a \in \Sigma$, $L(a) = \{a\}$.

Induction step:

$$
\begin{aligned}
&L(X \cup Y) = L(X) \cup L(Y),\\
&L(\overline{X}) = \overline{L(X)},\\
&L(XY) = L(X)L(Y), \text{ and}\\
&L(X^*) = (L(X))^*.
\end{aligned}
$$

The mapping L associates with each extended regular expression X the language $L(X)$. For example, for $\Sigma = \{a, b\}$,

$$L(((a \cup b)^* b)) = L((a \cup b)^*)L(b) = (L(a \cup b))^* L(b)$$

$$= (L(a) \cup L(b))^* L(b) = (\{a\} \cup \{b\})^* \{b\}.$$

Whereas each extended regular expression defines a unique regular language, each regular language may be represented by an infinite number of extended regular expressions. We say that extended regular expressions X and Y are *equivalent* if and only if $L(X) = L(Y)$. To simplify notation, we

denote this equivalence by equality and write $X = Y$ if $L(X) = L(Y)$. All the laws applicable to languages are now also applied to extended regular expressions. Thus we write $X \cup Y \cup Z$ instead of $(X \cup (Y \cup Z))$ since union is associative for languages, etc. Also, we write $x \in X$ and $X \subseteq Y$ instead of $x \in L(X)$ and $L(X) \subseteq L(Y)$. This simplifies the notation considerably. For example, for $(\{a\} \cup \{b\})^*\{a\}\{a\}$ we can now write $(a \cup b)^*aa$. It is also convenient to include $(X \cap Y), (X - Y), (X \Delta Y)$, and X^+ as extended regular expressions.

10.2.2 Quotients of Regular Expressions

We now define a quotient operator for extended regular expressions; the quotient of an extended regular expression by a word denotes the quotient of the language denoted by that expression. First we need a method for computing $\delta(L(X))$ from an extended regular expression X.

Definition 10.4 *If X is an extended regular expression, $\delta(X)$ is defined inductively as follows:*

Basis: $\delta(\emptyset) = \emptyset$; $\delta(\varepsilon) = \varepsilon$; $\delta(a) = \emptyset$, *for each* $a \in \Sigma$.

Induction step:

$$\delta(X \cup Y) = \delta(X) \cup \delta(Y),$$
$$\delta(\overline{X}) = \begin{cases} \varepsilon & \text{if } \delta(X) = \emptyset \\ \emptyset & \text{if } \delta(X) = \varepsilon, \end{cases}$$
$$\delta(XY) = \delta(X)\delta(Y),$$
$$\delta(X^*) = \varepsilon.$$

One easily verifies that $L(\delta(X)) = \delta(L(X))$.

The *quotient of an extended regular expression X with respect to a letter a* is defined as follows:

Basis: $a^{-1}\emptyset = a^{-1}\varepsilon = a^{-1}b = \emptyset$, where $a \neq b$; $a^{-1}a = \varepsilon$.

Induction step:

$$a^{-1}(X \cup Y) = (a^{-1}X) \cup (a^{-1}Y),$$

$$a^{-1}\overline{X} = \overline{a^{-1}X},$$

$$a^{-1}(XY) = (a^{-1}X)Y \cup \delta(X)(a^{-1}Y),$$

$$a^{-1}X^* = (a^{-1}X)X^*.$$

The *quotient of an extended regular expression with respect to a word* is defined as follows. If $|w| = 0$ then $w = \varepsilon$ and $w^{-1}X = X$. If $|w| = 1$, then w is a letter, and $w^{-1}X$ is computed as above. If $|w| > 0$, then $w = xa$, for some $x \in \Sigma^*, a \in \Sigma$, and

$$(xa)^{-1}X = a^{-1}(x^{-1}X).$$

One now verifies that

$$L(w^{-1}X) = w^{-1}(L(X)).$$

Definition 10.5 *Two extended regular expressions X and X' are said to be* similar *(and this is denoted by $X \sim Y$) if one can be obtained from the other using only a finite number of applications of the following rules, called* similarity laws*:*

$$X\emptyset = \emptyset X = \emptyset,$$

$$X \cup \emptyset = \emptyset \cup X = X,$$

$$X\varepsilon = \varepsilon X = X,$$

$$X \cup X = X,$$

$$X \cup Y = Y \cup X,$$

$$X \cup (Y \cup Z) = (X \cup Y) \cup Z,$$

where X, Y, and Z are any extended regular expressions.

The similarity relation $\sim$ is an equivalence relation on $\mathcal{ERX}_\Sigma$, and $X \sim Y$ implies $L(X) = L(Y)$. However, $L(X) = L(Y)$ does not imply $X \sim Y$. For example, this is the case if $\Sigma = \{a, b\}$, $X = \Sigma^* a$ and $Y = (b \cup aa^*b)^* aa^*$, or if $X = \overline{a^*}$ and $Y = \Sigma^* b \Sigma^*$. Even with the relatively weak set of similarity laws, we can prove that the number of quotients of a regular language is finite.

Theorem 10.1 *The number of dissimilar quotients of an extended regular expression is finite.*

Proof: The proof is beyond the scope of this book, and we refer the reader to [16, 17] for further details. However, the following sketch indicates the structure of the proof. We proceed by induction on the number $n(X)$ of regular operators (union, concatenation, complement, and star) in X. Let $q(X)$ be the number of dissimilar quotients of an extended regular expression X.

Basis: If $n(X) = 0$, then X is one of $\emptyset$, ε, or $a \in \Sigma$. One verifies that $q(\emptyset) = 1$, $q(\varepsilon) = 2$, and $q(a) = 3$.

Induction step: If $n(X) > 0$ then X must have one of the forms below.

If $X = Y \cup Z$, then $q(X) \leq q(Y)q(Z)$,
if $X = \overline{Y}$, then $q(X) = q(Y)$,
if $X = YZ$, then $q(X) \leq q(Y)2^{q(Z)}$,
if $X = Y^*$, then $q(X) \leq 2^{q(Y)}$.

From this it follows that the number of dissimilar quotients is always finite. □

Corollary 10.1 *If X has $q(X)$ dissimilar quotients, then they can all be found by taking quotients with respect to words of length $\leq q(X) - 1$.*

Proof: Arrange the words of Σ^* in order of increasing length, and alphabetically for words of the same length. Find the quotients of X in that order. If, for some n, all the quotients with respect to words of length n already have similar counterparts with respect to words of shorter length, then no new dissimilar quotients will ever be found, and the process terminates. Thus at least one new quotient must be found for each n or the process terminates. In the worst case, only one quotient is found for each $n \in \{0, 1, \ldots, q(X) - 1\}$. □

Corollary 10.2 *Every regular language has a finite number of distinct quotients.*

Proof: This follows, because similarity implies equivalence. □

We now present some examples of the process of quotient construction.

Example 1

In the divide-by-2 counter described in Chapter 1, an output change occurs for every two input changes. Suppose we let a and b represent input and output changes, respectively. Then the possible input-output sequences—such sequences are sometimes called *traces*—that may occur when the counter is started in a stable state are:

$$\varepsilon, a, aa, aab, aaba, aabaa, aabaab, \ldots,$$

i.e., at any particular time, there may have been no signal changes at all (ε), or there may have been exactly one input change (a), or two input changes (aa), or two input changes followed by an output change (aab), etc. One can verify that this set of sequences is conveniently represented by the extended regular expression $X = (aab)^*(\varepsilon \cup a \cup aa)$. For the purposes of illustrating precisely the inductive nature of the quotient construction process, we will assume that the expression is parenthesized as follows:

$$X = (((aa)b)^*((\varepsilon \cup a) \cup (aa))).$$

Suppose we wish to compute $a^{-1}X$. Since X is the concatenation of two expressions, $X = YZ$, where $Y = ((aa)b)^*$ and $Z = ((\varepsilon \cup a) \cup (aa))$, we use the law for concatenation first:

$$a^{-1}X = (a^{-1}Y)Z \cup \delta(Y)(a^{-1}Z) = (a^{-1}Y)Z \cup (a^{-1}Z),$$

since $\delta(Y) = \varepsilon$. Next, we use the law for star to obtain

$$a^{-1}Y = (a^{-1}((aa)b))Y.$$

The law for concatenation would then be applied to the subexpression $((aa)b)$. However, if we use the similarity laws, the computation can proceed much more quickly. It is clear that $a^{-1}((aa)b) = a^{-1}(aab) = ab$. Therefore, $a^{-1}Y = abY$. Similarly, it is easy to see that $a^{-1}Z = \varepsilon \cup a$. Altogether,

$$a^{-1}X = abYZ \cup \varepsilon \cup a = abX \cup \varepsilon \cup a.$$

Using similar reasoning, we find all the dissimilar quotients of X as follows:

$$\varepsilon^{-1}X = X = (aab)^*(\varepsilon \cup a \cup aa),$$

$$a^{-1}X = abX \cup \varepsilon \cup a,$$

$$b^{-1}X = \emptyset,$$

$$(aa)^{-1}X = a^{-1}(a^{-1}X) = bX \cup \varepsilon,$$

$$(ab)^{-1}X = b^{-1}(a^{-1}X) = \emptyset = b^{-1}X,$$

$$(ba)^{-1}X = a^{-1}(b^{-1}X) = \emptyset = b^{-1}X,$$

$$(bb)^{-1}X = b^{-1}(b^{-1}X) = \emptyset = b^{-1}X,$$

$$(aaa)^{-1}X = a^{-1}((aa)^{-1}X) = \emptyset = b^{-1}X,$$

$$(aab)^{-1}X = b^{-1}((aa)^{-1}X) = X.$$

Since no new quotients arise with respect to words of length three, the process terminates here. There are four dissimilar quotients: X, $abX \cup \varepsilon \cup a$, $\emptyset$, and $bX \cup \varepsilon$. It is easy to see that each quotient corresponds to a distinct language. We will return to this example shortly.

Example 2

Consider a set-reset latch in which s and r represent changes on the *set* and *reset* inputs, respectively. Suppose the latch starts in a stable state with both inputs at 0. We then wish to restrict the latch operation as follows: (a) the *set* and *reset* inputs are never to change at the same time; (b) only input pulses on the *set* and *reset* inputs are allowed, i.e., if the signal s occurs, it must be followed by another s before r can change, and vice versa; (c) the input sequence cannot begin with a reset. The first condition is automatically taken care of if we assume that the input alphabet is $\Sigma = \{r, s\}$. For the second condition we must restrict the input changes to the set $(ss \cup rr)^*$. The third condition can be expressed by $\overline{r\Sigma^*}$, where $\Sigma = s \cup r$. Altogether, the desired behavior may be described by

$$X = \overline{r\Sigma^*} \cap (ss \cup rr)^*.$$

The quotients are computed as shown below. To shorten the example, we use some obvious additional simplification rules like $X \cap \Sigma^* = X$.

$$\varepsilon^{-1}X = X = \overline{r\Sigma^*} \cap (ss \cup rr)^*,$$

$$r^{-1}X = \overline{\Sigma^*} \cap r(ss \cup rr)^* = \emptyset \cap r(ss \cup rr)^* = \emptyset,$$

$$s^{-1}X = \overline{\emptyset} \cap s(ss \cup rr)^* = \Sigma^* \cap s(ss \cup rr)^* = s(ss \cup rr)^*,$$

$$(rr)^{-1}X = \emptyset = r^{-1}X,$$

$$(rs)^{-1}X = \emptyset = r^{-1}X,$$

$$(sr)^{-1}X = \emptyset = r^{-1}X,$$

$$(ss)^{-1}X = (ss \cup rr)^*,$$

$$(ssr)^{-1}X = r(ss \cup rr)^*,$$

$$(sss)^{-1}X = s(ss \cup rr)^* = s^{-1}X,$$

$$(ssrr)^{-1}X = (ss \cup rr)^* = (ss)^{-1}X,$$

$$(ssrs)^{-1}X = \emptyset = r^{-1}X.$$

Altogether, there are five dissimilar quotients; one verifies that no two of them denote the same language. We will return to this example shortly.

10.3 Quotient Equations

A word in any language is either empty or must begin with some letter. The set of all words that begin with the letter a in a language $\mathcal{X}$ is clearly $\{a\}(a^{-1}\mathcal{X})$. Hence we have the following disjoint decomposition for any language:

$$\mathcal{X} = \bigcup_{a\in\Sigma} \{a\}(a^{-1}\mathcal{X}) \cup \delta(\mathcal{X}).$$

Each quotient can also be expressed in this form because

$$w^{-1}\mathcal{X} = \bigcup_{a\in\Sigma} \{a\}((wa)^{-1}\mathcal{X}) \cup \delta(w^{-1}\mathcal{X}).$$

If $\mathcal{X}$ is regular and is denoted by X, then the dissimilar quotients of X satisfy a finite set of equations derived from the form above.

Example 1 (continued)

Let X_w denote $w^{-1}X$. Then the quotient equations for $X = (aab)^*(\varepsilon \cup a \cup aa)$ are

$$X = aX_a \cup bX_b \cup \varepsilon,$$

$$X_a = aX_{aa} \cup bX_b \cup \varepsilon,$$

$$X_b = aX_b \cup bX_b,$$

$$X_{aa} = aX_b \cup bX \cup \varepsilon.$$

Example 2 (continued)

For $X = \overline{rA^*} \cap (ss \cup rr)^*$, we have the equations

$$X = rX_r \cup sX_s \cup \varepsilon,$$

$$X_r = rX_r \cup sX_r,$$

$$X_s = rX_r \cup sX_{ss},$$

$$X_{ss} = rX_{ssr} \cup sX_s \cup \varepsilon,$$

$$X_{ssr} = rX_{ss} \cup sX_r.$$

	r	s	
→ X	X_r	X_s	ε
X_r	X_r	X_r	$\emptyset$
X_s	X_r	X_{ss}	$\emptyset$
X_{ss}	X_{ssr}	X_s	ε
X_{ssr}	X_{ss}	X_r	$\emptyset$

FIGURE 10.1. Representation of quotient equations by a table.

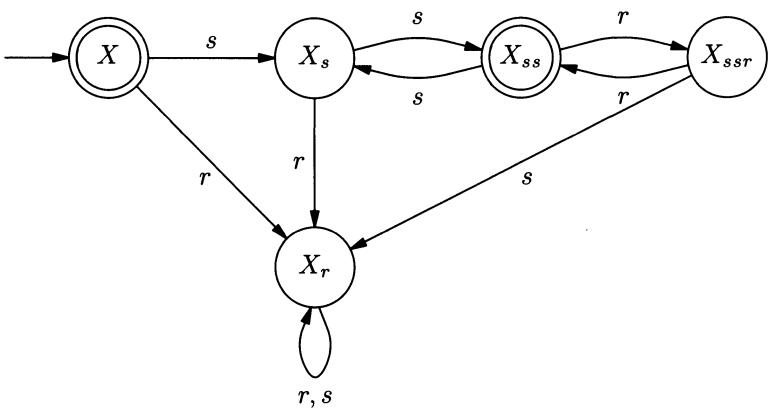


FIGURE 10.2. Representation of quotient equations by a graph.

It is useful to represent the set of quotient equations in two different forms. The *tabular* form for Example 2 is shown in Figure 10.1, where the correspondence is self-explanatory. The incoming arrow designates the given expression. The *graphical* form is shown in Figure 10.2, where quotients correspond to the vertices of the graph. An incoming arrow designates the expression X. A double circle denotes a quotient containing the empty word. The rest is self-explanatory.

Up to this point we have shown that every extended regular expression has a finite number of dissimilar quotients that satisfy a set of equations. We now show that every such set of equations can be solved, and that the solution is always a regular language.

Theorem 10.2 *Let $\mathcal{Y}$ and $\mathcal{Z}$ be arbitrary languages over some alphabet Σ. Suppose further that $\varepsilon \notin Y$. Then the equation*

$$\mathcal{X} = \mathcal{Y}\mathcal{X} \cup \mathcal{Z}$$

has the unique solution $\mathcal{X} = \mathcal{Y}^*\mathcal{Z}$.

Proof: First note that every solution must contain $\mathcal{Z}$. But, if $\mathcal{X} \supseteq \mathcal{Z}$, then $\mathcal{Y}\mathcal{X} \supseteq \mathcal{Y}\mathcal{Z}$; thus $\mathcal{X}$ also contains $\mathcal{Y}\mathcal{Z}$. Continuing this argument, we see that $\mathcal{X} \supseteq \mathcal{Z} \cup \mathcal{Y}\mathcal{Z} \cup \mathcal{Y}\mathcal{Y}\mathcal{Z} \cup \ldots = \mathcal{Y}^*\mathcal{Z}$. Thus every solution must contain $\mathcal{Y}^*\mathcal{Z}$. Now suppose that $\mathcal{Y}^*\mathcal{Z} \cup \mathcal{W}$ is a solution for some $\mathcal{W}$. Without loss of generality, we can assume that $\mathcal{Y}^*\mathcal{Z} \cap \mathcal{W} = \emptyset$. Substituting $\mathcal{Y}^*\mathcal{Z} \cap \mathcal{W}$ for $\mathcal{X}$ in the equation, we get

$$\mathcal{Y}^*\mathcal{Z} \cup \mathcal{W} = \mathcal{Y}(\mathcal{Y}^*\mathcal{Z} \cup \mathcal{W}) \cup \mathcal{Z} = \mathcal{Y}\mathcal{Y}^*\mathcal{Z} \cup \mathcal{Z} \cup \mathcal{Y}\mathcal{W} = \mathcal{Y}^*\mathcal{Z} \cup \mathcal{Y}\mathcal{W}.$$

Let w be a shortest word of $\mathcal{W}$ and let $|w| = r$. We must have $w \in \mathcal{Y}^*\mathcal{Z} \cup \mathcal{Y}\mathcal{W}$. Also, $w \notin \mathcal{Y}^*\mathcal{Z}$, since we have assumed that $\mathcal{Y}^*\mathcal{Z} \cap \mathcal{W} = \emptyset$. Thus $w \in \mathcal{Y}\mathcal{W}$. Because we have assumed that the empty word is not in $\mathcal{Y}$, a shortest word of $\mathcal{Y}\mathcal{W}$ is of length at least $r + 1$. This is a contradiction, showing that $\mathcal{W} = \emptyset$. Thus, if there is a solution, it cannot be smaller than $\mathcal{Y}^*\mathcal{Z}$ and it cannot be larger. One easily verifies that $\mathcal{Y}^*\mathcal{Z}$ is a solution, for

$$\mathcal{Y}(\mathcal{Y}^*\mathcal{Z}) \cup \mathcal{Z} = (\mathcal{Y}\mathcal{Y}^* \cup \varepsilon)\mathcal{Z} = (Y^+ \cup \varepsilon)\mathcal{Z} = \mathcal{Y}^*\mathcal{Z}. \qquad \square$$

Corollary 10.3 *Any set of quotient equations can be solved for all the quotients by repeated application of Theorem 10.2.*

Example 2 (continued)

First note that the equation for X_r has the form $X_r = rX_r \cup sX_r = (r \cup s)X_r \cup \emptyset$. Its solution is $(r \cup s)^*\emptyset = \emptyset$. Hence, the set of equations can be simplified to

$$X = sX_s \cup \varepsilon,$$

$$X_s = sX_{ss},$$

$$X_{ss} = sX_s \cup rX_{ssr} \cup \varepsilon.$$

$$X_{ssr} = rX_{ss}.$$

Substituting the second and fourth equations into the first and third yields

$$X = ssX_{ss} \cup \varepsilon,$$

$$X_{ss} = ssX_{ss} \cup rrX_{ss} \cup \varepsilon = (ss \cup rr)X_{ss} \cup \varepsilon.$$

Solving for X_{ss} we have $X_{ss} = (ss \cup rr)^*$. The remaining quotients are $X = ss(ss \cup rr)^* \cup \varepsilon$, $X_s = s(ss \cup rr)^*$, and $X_{ssr} = r(ss \cup rr)^*$.

We have shown that each extended regular expression X has a finite number of distinct quotients that satisfy the quotient equations. Conversely, the quotient equations can be solved to recover $L(X)$, although the new expression may be different from X. We can thus state

Theorem 10.3 *A language is regular if and only if it has a finite number of quotients.*

Using the theorem above, we can easily verify that the language $\{a^n b^n \mid n \geq 1\}$ is not regular. The quotient of this language with respect to the word a^i is $\{a^{n-i}b^n \mid n \geq i\}$. This quotient contains the word b^i and it contains no other word consisting of b's only. Hence all the quotients with respect to words of the form a^i are distinct, and there is an infinite number of them. Therefore, the language cannot be regular.

Define a *regular expression* to be an extended regular expression that does not use the complementation operator, but only union, concatenation, and star. From our method for solving quotient equations we have

Theorem 10.4 *A language is regular if and only if it can be denoted by a regular expression.*

This result shows that we could have defined regular languages without using complementation, and we would have obtained the same family. We have allowed complement to be used because it is often convenient.

We next consider the problem of deciding the equivalence of two extended regular expressions. The solution to this problem allows us to reduce the quotient equations to a set in which all quotients are distinct.

Theorem 10.5 *Let X, Y, and Z be extended regular expressions over Σ. Then*

- *$X = \emptyset$ if and only if no quotient of X contains ε.*
- *$X = \Sigma^*$ if and only if every quotient of X contains ε.*
- *$Y \supseteq Z$ if and only if $Y \cup \overline{Z} = \Sigma^*$.*
- *$Y = Z$ if and only if $Y \Delta Z = \emptyset$.*

The verification of these properties is straightforward. We now have the following procedures.

To test whether $X = \emptyset$ or $X = \Sigma^*$, construct the quotients of X. The equivalence of two quotients may not always be recognized, but we are assured that the number of dissimilar quotients is finite. Find $\delta(X_w)$ for each quotient X_w. Then use Part 1 or 2 of the theorem above. To test whether $Y \supseteq Z$, test whether the expression $Y \cup \overline{Z}$ is equivalent to Σ^*. To test whether $Y = Z$, test whether the expression $Y \Delta Z$ is equivalent to the empty set.

10.4 Finite Automata

10.4.1 Basic Concepts

We now define finite automata and characterize their behavior by regular expressions. Finite automata have numerous applications in computer science; we will use them for describing behaviors. A *finite automaton*, or simply *automaton*, is a system of the form $\alpha = \langle \Sigma, \mathcal{Q}, q_1, \mathcal{F}, f \rangle$, where

- Σ is a finite, nonempty *input alphabet*;
- $\mathcal{Q}$ is a finite, nonempty set, called the *state* set;
- $q_1 \in \mathcal{Q}$ is the *initial state*;
- $\mathcal{F} \subseteq \mathcal{Q}$ is the set of *accepting states* (the set $\mathcal{Q} - \mathcal{F}$ is the set of *rejecting states*);
- $f : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ is the *transition function.*

A typical step in the operation of a finite automaton is as follows: Suppose the automaton is in a state $q \in \mathcal{Q}$. If it receives an input $\sigma \in \Sigma$, it computes its next state as $f(q, \sigma)$ and moves to that state. We assume that it is possible for us to know whether the present automaton state is an accepting state or not. In fact, it is convenient to define an *output function* $g : \mathcal{Q} \to \{0, 1\}$, where $g(q) = 1$ if $q \in \mathcal{F}$ and $g(q) = 0$ otherwise.

The transition function is extended to a mapping from $\mathcal{Q} \times \Sigma^*$ to $\mathcal{Q}$ as described below; we use the same symbol f for the extended function.

$$f(q, \varepsilon) = q,$$

$$f(q, w\sigma) = f(f(q, w), \sigma),$$

for all $q \in \mathcal{Q}$, $w \in \Sigma^*$, and $\sigma \in \Sigma$. Thus the application of the empty word does not change the state and, for $w \in \Sigma^+$, $f(q, w)$ denotes the state reached from q after the letters of w have been applied in succession.

An *incomplete automaton* is the same as an automaton, except that its transition function is partial. This means effectively that for some pairs (q, σ) the transition function is not defined. An incomplete automaton α' is used as a simpler form of a (complete) automaton α. The automaton α has one additional rejecting "sink" state s. The transition function of α is the same as that of α', except that unspecified transitions of α' go to the sink state s in α and $f(s, \sigma) = s$ for all $\sigma \in \Sigma$.

The *language accepted* by a finite automaton α is denoted by $L(\alpha)$ and is defined as

$$L(\alpha) = \{w \in \Sigma^* \mid f(q_1, w) \in \mathcal{F}\}.$$

To illustrate these definitions, consider the automaton

$$\alpha = \langle \{0, 1\}, \{0, 1, 2\}, 0, \{2\}, f \rangle,$$

where $f(q,0) = q$ and $f(q,1) = q+1$ (modulo-3) for all $q \in \{0,1,2\}$. We can represent the automaton by the table of Figure 10.3. The states (0, 1, and 2) are listed as rows of the table. The arrow indicates that state 0 is the initial state. The input symbols (0 and 1) are listed as the first two columns. The entry in row q and column σ gives the next state $f(q,\sigma)$. The right-most column gives the value $g(q)$ of the output function, showing that state 2 is the only accepting state. The automaton accepts any input word in which the total number of 1's is 2 modulo-3.

q \ σ	0	1	$g(q)$
→ 0	0	1	0
1	1	2	0
2	2	0	1

FIGURE 10.3. State table of a modulo-3 counter.

An alternative, but equivalent, representation is shown in Figure 10.4. Here, states are represented as vertices. The initial state has an incoming short arrow, and accepting states are indicated by double circles. If $f(q,\sigma) = q'$, then there is a directed edge from vertex q to vertex q'; furthermore, this edge is labeled by σ.

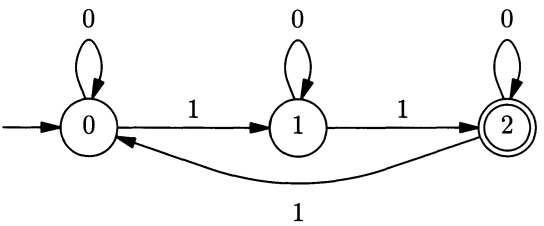


FIGURE 10.4. State graph of a modulo-3 counter.

It is evident that the three ways of describing an automaton (as a 5-tuple, a state table, and a state graph) are equivalent, and one description is easily reconstructed from another.

An automaton $\langle \Sigma', \mathcal{Q}', q_1, \mathcal{F}', f' \rangle$ is a *subautomaton* of the automaton $\langle \Sigma, \mathcal{Q}, q_1, \mathcal{F}, f \rangle$ if $\Sigma' \subseteq \Sigma$, $\mathcal{Q}' \subseteq \mathcal{Q}$, $\mathcal{F}' = \mathcal{F} \cap \mathcal{Q}'$, and f' is the restriction of f to $\mathcal{Q}' \times \Sigma'$. A state q of an automaton $\alpha = \langle \Sigma, \mathcal{Q}, q_1, \mathcal{F}, f \rangle$ is *accessible* if there exists a word $w \in \Sigma^*$ such that $f(q_1, w) = q$. The *connected subautomaton* α_{con} of α is the subautomaton $\alpha_{con} = \langle \Sigma, \mathcal{Q}', q_1, \mathcal{F}', f' \rangle$, where $\mathcal{Q}'$ is the set of all the accessible states, $\mathcal{F}'$ is the set of all the accessible accepting states, and f' is the restriction of f to $\mathcal{Q}' \times \Sigma$. Normally, we assume that the automata we are dealing with are *connected*, i.e., that $\alpha_{con} = \alpha$.

10.4.2 Recognizable Languages

A language $\mathcal{X} \subseteq \Sigma^*$ is called *recognizable* by a finite automaton if there exists a finite automaton α such that $\mathcal{X} = L(\alpha)$. We now show that the family of recognizable languages is the same as the family of regular languages.

Let $\alpha = \langle \Sigma, \mathcal{Q}, q_1, \mathcal{F}, f \rangle$ be a finite automaton, and let q_i be a state in $\mathcal{Q}$. Denote by α_i the automaton α with the initial state changed to q_i, i.e., let $\alpha_i = \langle \Sigma, \mathcal{Q}, q_i, \mathcal{F}, f \rangle$. (Note that α_i need not be connected, even if α is.) In this notation $\alpha = \alpha_1$. If the cardinality of $\mathcal{Q}$ is n, we have n automata of the form α_i. Let $\mathcal{X}_i = L(\alpha_i)$, for $i = 1, \ldots, n$. The set $\mathcal{X}_i$ can be thought of as the language accepted by state q_i of α; in fact, $L(\alpha_i) = \{w \mid f(q_i, w) \in \mathcal{F}\}$. These n languages are related as follows:

Proposition 10.1 *Let $\alpha = \langle \Sigma, \mathcal{Q}, q_1, \mathcal{F}, f \rangle$ be a connected automaton with $\mathcal{X} = L(\alpha)$ and, for each i, let $w_i \in \Sigma^*$ be such that $q_i = f(q_1, w_i)$. Then $\mathcal{X}_i = w_i^{-1}\mathcal{X}$.*

Proof: We have $w \in \mathcal{X}_i$ if and only if $f(q_i, w) \in \mathcal{F}$ if and only if $f(q_1, w_i w) \in \mathcal{F}$ if and only if $w \in w_i^{-1}\mathcal{X}$. □

Theorem 10.6 *A language is recognizable if and only if it is regular.*

Proof: Suppose $\mathcal{X} \subseteq \Sigma^*$ is recognizable by automaton α. Consider any $w \in \Sigma^*$ and the quotient $w^{-1}\mathcal{X}$. By the proposition above, if w takes q_1 to q_i, then $w^{-1}\mathcal{X} = \mathcal{X}_i$. Hence each quotient of $\mathcal{X}$ is equal to one of the languages of the form $\mathcal{X}_i$. Since the automaton is finite, it follows that $\mathcal{X}$ has a finite number of quotients. Hence it is regular. Conversely, suppose $\mathcal{X} \subseteq \Sigma^*$ is regular and let $\mathcal{X} = \mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_n$ be all the distinct quotients of $\mathcal{X}$. Consider the automaton $\alpha = \langle \Sigma, \{\mathcal{X}_1, \ldots, \mathcal{X}_n\}, \mathcal{X}_1, \mathcal{F}, f \rangle$, where $\mathcal{X}_i \in \mathcal{F}$ if and only if $\varepsilon \in \mathcal{X}_i$, and $f(\mathcal{X}_i, \sigma) = \sigma^{-1}\mathcal{X}_i$. One verifies that $f(\mathcal{X}_i, w) = w^{-1}\mathcal{X}_i$, for any $w \in \Sigma^*$. Now $w \in L(\alpha)$ if and only if $f(\mathcal{X}_1, w) \in \mathcal{F}$ if and only if $w^{-1}\mathcal{X}_1 \in \mathcal{F}$ if and only if $\varepsilon \in w^{-1}\mathcal{X}_1$ if and only if $w \in \mathcal{X}_1$. Hence $L(\alpha) = \mathcal{X}$, and $\mathcal{X}$ is recognizable. □

The proof above establishes a one-to-one correspondence between the quotients of a regular language and the states of the automaton recognizing that language. This provides very direct methods for constructing a finite automaton recognizing a given regular language, and for finding a regular expression for the language accepted by a finite automaton.

To illustrate the correspondence between states and quotients, consider the modulo-3 counter of Figure 10.3. If $\mathcal{X}_i$ corresponds to state i, we immediately obtain the following set of equations:

$$\mathcal{X}_0 = 0\mathcal{X}_0 \cup 1\mathcal{X}_1,$$

$$\mathcal{X}_1 = 0\mathcal{X}_1 \cup 1\mathcal{X}_2,$$

$$\mathcal{X}_2 = 0\mathcal{X}_2 \cup 1\mathcal{X}_0 \cup \varepsilon.$$

We can solve these equations for $\mathcal{X}_1 = L(\alpha)$. Thus,

$$\mathcal{X}_2 = 0^*1\mathcal{X}_0 \cup 0^*,$$

$$\mathcal{X}_1 = 0\mathcal{X}_1 \cup 10^*1\mathcal{X}_0 \cup 10^* = 0^*10^*1\mathcal{X}_0 \cup 0^*10^*,$$

$$\mathcal{X}_0 = 0\mathcal{X}_0 \cup 10^*10^*1\mathcal{X}_0 \cup 10^*10^* = (0 \cup 10^*10^*1)^*10^*10^*.$$

Conversely, given an extended regular expression, one needs only to construct its quotient equations to obtain a finite automaton recognizing the expression. We have already illustrated this with the example of Figure 10.2.

10.5 Equivalence and Reduction of Automata

Consider two automata $\alpha = \langle \Sigma, \mathcal{Q}, q_1, \mathcal{F}, f \rangle$ and $\alpha' = \langle \Sigma, \mathcal{Q}', q_1', \mathcal{F}', f' \rangle$ with output functions g and g', connected as shown in Figure 10.5. Apply the same input sequence to both automata and observe the outputs at the end of the sequence. If the outputs differ, then the two automata are said to be *distinguishable*; in fact, they are distinguishable by that sequence. If,

$$\text{for all } w \in \Sigma^*, g(f(q_1, w)) = g'(f'(q_1', w)),$$

then $L(\alpha) = L(\alpha')$ and α and α' are said to be *indistinguishable* or *equivalent.*

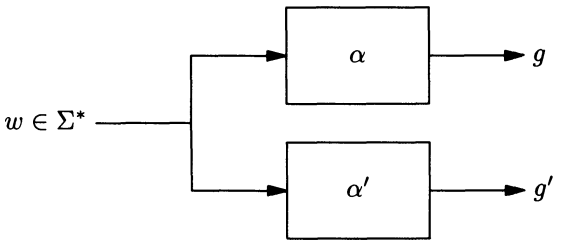


FIGURE 10.5. Parallel connection of automata.

The test described above involves an infinite number of words, namely, all the words in Σ^*. We can reduce this test to one involving a finite number of steps. Define the automaton

$$\alpha\Delta\alpha' = \langle \Sigma, \mathcal{Q} \times \mathcal{Q}', (q_1, q_1'), \mathcal{H}, h \rangle,$$

where $h((q, q'), \sigma) = (f(q, \sigma), f'(q', \sigma))$, and $(q, q') \in \mathcal{H}$ if and only if ($q \in \mathcal{F}$ and $q' \notin \mathcal{F}'$) or ($q \notin \mathcal{F}$ and $q' \in \mathcal{F}'$). The automaton $\alpha\Delta\alpha'$ corresponds to the automaton obtained from the parallel connection of α and α' by combining the outputs g and g' with a XOR gate to obtain output h. It is clear that α and α' are indistinguishable if and only if the output h is

always 0. This is equivalent to testing whether $L(\alpha \Delta \alpha') = \emptyset$. If α has n states and α' has n' states, then $\alpha \Delta \alpha'$ has at most nn' states. It follows that, if two automata are distinguishable, then they can be distinguished by a word of length not greater that nn'.

Consider now the problem of testing whether two states q_i and q_j of a given automaton $\alpha = \langle \Sigma, \mathcal{Q}, q_1, \mathcal{F}, f \rangle$ are equivalent. If they are equivalent, i.e., indistinguishable, then we will write $q_i \sim q_j$. Indistinguishability is an equivalence relation on $\mathcal{Q}$. Testing for indistinguishability amounts to testing whether $L(\alpha_i) = L(\alpha_j)$. (Recall that α_i is α with initial state changed to q_i.) We could test for the equivalence of every pair of states by the method described above. A more efficient algorithm will be described below in a somewhat more general setting.

A *Moore machine* [104] is a 6-tuple $\mu = \langle \Sigma, \Gamma, \mathcal{Q}, q_1, f, g \rangle$, where Σ, $\mathcal{Q}$, q_1, and f are as in a finite automaton, Γ is a finite, nonempty *output alphabet*, and $g : \mathcal{Q} \to \Gamma$ is the *output function*. A Moore machine is therefore a generalization of the concept of finite automaton. An automaton can have at most two output values, whereas a Moore machine may have any finite number of output values.

Two states q_i and q_j of a Moore machine are *k-distinguishable* if there exists a word $w \in \Sigma^*$ of length $\leq k$ such that $g(f(q_i, w)) \neq g(f(q_j, w))$. In particular, q_i and q_j are 0-distinguishable if and only if $g(q_i) \neq g(q_j)$. Two states are *distinguishable* if they are k-distinguishable for some $k \geq 0$. They are *indistinguishable* or *equivalent*, written $q_i \sim q_j$, if they are not distinguishable. Call q_i and q_j *k-equivalent* if they are not k-distinguishable; in that case we write $q_i \sim_k q_j$. The relation k-equivalence is indeed an equivalence relation on the set $\mathcal{Q}$ of states, and defines a partition P_k on $\mathcal{Q}$; the blocks of this partition are the k-equivalence classes. Since $q_i \sim_{k+1} q_j$ implies $q_i \sim_k q_j$, it follows that P_{k+1} is a refinement of P_k.

Theorem 10.7 *Let $\mu = \langle \Sigma, \Gamma, \mathcal{Q}, q_1, f, g \rangle$, be a Moore machine with n states. If two states q_i and q_j are distinguishable, then they are distinguishable for some $k \leq n - 2$.*

Proof: Consider the partitions $P_0, P_1, \ldots$, etc. Suppose, for some m, $P_0, \ldots, P_m$ are all distinct, i.e., P_i is a proper refinement of P_{i-1} for $1 \leq i \leq m$, but $P_{m+1} = P_m$. We claim that $q_i \sim_m q_j$ implies $q_i \sim q_j$. Suppose, to the contrary, that there exists $w \in \Sigma^*$ with $|w| = r > m$ such that $g(f(q_i, w)) \neq g(f(q_j, w))$. Without loss of generality, suppose that w is the shortest such word. Let $w = uv$ with $|v| = m + 1$, and let $p_i = f(q_i, u)$ and $p_j = f(q_j, u)$. Then p_i and p_j are $(m + 1)$-distinguishable. By the assumption that $P_{m+1} = P_m$, p_i and p_j are m-distinguishable. Hence q_i and q_j are $(r - 1)$-distinguishable. This contradicts the fact that w was a shortest distinguishing word.

We have assumed that q_i and q_j are distinguishable; hence the partition P_0 must have at least two nonempty blocks. For $i > 0$, if P_i is a proper refinement of P_{i-1}, then P_i has at least $i + 2$ blocks. In particular, P_{n-2}

has at least n blocks if it is a proper refinement of P_{n-3}. Thus the longest possible sequence of partitions is $P_0, \ldots, P_{n-2}$. Hence any two states that are distinguishable are in different blocks of P_{n-2}, and the claim follows.□

To illustrate the construction above, consider the Moore machine of Figure 10.6. The partition P_0 is $P_0 = \{1,2,3\}\{4,5\}$. Examine the transitions from block $\{1,2,3\}$ under input a; the resulting states are $\{2,3,2\}$. Since 2 and 3 are 0-equivalent, no new information is found. Next examine transitions under b; now the states are $\{3,4,4\}$. Since 3 and 4 are 0-distinguishable, the pairs $\{1,2\}$ and $\{1,3\}$ are 1-distinguishable. A similar examination of block $\{4,5\}$ yields no new information. Hence $P_1 = \{1\}\{2,3\}\{4,5\}$. In the next step, block $\{4,5\}$ can be refined and $P_2 = \{1\}\{2,3\}\{4\}\{5\}$. One verifies that $P_2 = P_3$, and the process stops here.

	a	b	
1	2	3	1
2	3	4	1
3	2	4	1
4	5	3	0
5	4	1	0

FIGURE 10.6. Illustrating equivalent states.

A Moore machine is *reduced* if no two of its states are equivalent. A reduced machine corresponding to a given machine can always be constructed by using the equivalence classes (blocks of the final partition) of the relation $\sim$ on $\mathcal{Q}$ as states of the reduced machine. The reduced machine for our example above is shown in Figure 10.7.

	a	b	
{1}	{2,3}	{2,3}	1
{2,3}	{2,3}	{4}	1
{4}	{5}	{2,3}	0
{5}	{4}	{1}	0

FIGURE 10.7. Illustrating a reduced machine.

10.6 Nondeterministic Automata

The automata and Moore machines defined so far were *deterministic* in the sense that the next state was uniquely determined by the present state

and input. The notion of transition function can be generalized to permit a choice of several next states. Such a concept arises naturally in the context of quotient equations. For example, consider the equations

$$\mathcal{X}_1 = a\mathcal{X}_2 \cup b\mathcal{X}_1 \cup c\mathcal{X}_3 \cup \varepsilon,$$

$$\mathcal{X}_2 = a\mathcal{X}_2 \cup b\mathcal{X}_2 \cup c\mathcal{X}_2 \cup \varepsilon,$$

$$\mathcal{X}_3 = a\mathcal{X}_3 \cup b\mathcal{X}_2 \cup c\mathcal{X}_1.$$

Since $\mathcal{X}_2 = (a \cup b \cup c)\mathcal{X}_2 \cup \varepsilon$, we have $\mathcal{X}_2 = \Sigma^*$. Also,

$$\begin{aligned} \mathcal{X}_1 \cup \mathcal{X}_3 &= a(\mathcal{X}_2 \cup \mathcal{X}_3) \cup b(\mathcal{X}_1 \cup \mathcal{X}_2) \cup c(\mathcal{X}_1 \cup \mathcal{X}_3) \cup \varepsilon \\ &= a\Sigma^* \cup b\Sigma^* \cup c(\mathcal{X}_1 \cup \mathcal{X}_3) \cup \varepsilon. \end{aligned}$$

Since every quotient of $\mathcal{X}_1 \cup \mathcal{X}_3$ contains ε, we also have $\mathcal{X}_1 \cup \mathcal{X}_3 = \Sigma^*$, i.e., $\mathcal{X}_2 = \mathcal{X}_1 \cup \mathcal{X}_3$. The three quotient equations can be replaced by the following two equations:

$$\mathcal{X}_1 = a(\mathcal{X}_1 \cup \mathcal{X}_3) \cup b\mathcal{X}_1 \cup c\mathcal{X}_3 \cup \varepsilon,$$

$$\mathcal{X}_3 = a\mathcal{X}_3 \cup b(\mathcal{X}_1 \cup \mathcal{X}_3) \cup c\mathcal{X}_1.$$

The state graph corresponding to the new set of equations is shown in Figure 10.8.

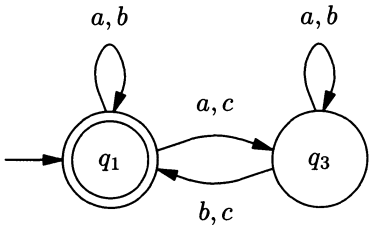


FIGURE 10.8. A nondeterministic automaton.

A *nondeterministic finite automaton* ν is a 5-tuple $\nu = \langle \Sigma, \mathcal{Q}, \mathcal{I}, \mathcal{F}, f \rangle$, where Σ, $\mathcal{Q}$, and $\mathcal{F}$ are as in a deterministic automaton, $\mathcal{I} \subseteq \mathcal{Q}$ is the *set of initial states*, and $f : \mathcal{Q} \times \Sigma \to \mathcal{P}(\mathcal{Q})$ is the transition function specifying a set (possibly empty) of next states.

As in the deterministic case, we extend the transition function to words. For all $q \in \mathcal{Q}$, $w \in \Sigma^*$, and $\sigma \in \Sigma$, $f(q, \varepsilon) = \{q\}$, and

$$f(q, w\sigma) = \bigcup_{q' \in f(q,w)} f(q', \sigma).$$

We also extend the transition function to sets of states. For $\mathcal{Q}' \subseteq \mathcal{Q}$, $f(\mathcal{Q}', w) = \bigcup_{q \in \mathcal{Q}'} f(q, w)$. The language accepted by a nondeterministic automaton is $L(\nu) = \{w \mid f(\mathcal{I}, w) \cap \mathcal{F} \neq \emptyset\}$.

Theorem 10.8 *Let ν be a nondeterministic automaton. Then $L(\nu)$ is a regular language.*

Proof: As in the deterministic case, we associate a nondeterministic automaton ν_i with each state q_i of ν, where $\nu_i = \langle \Sigma, \mathcal{Q}, \{q_i\}, \mathcal{F}, f \rangle$. Let $\mathcal{X}_i = L(\nu_i)$, as before. Then one can write a set of quotient equations for the $\mathcal{X}_i$ and solve them as in the deterministic case. □

A deterministic automaton equivalent to a given nondeterministic automaton can be found using the so-called subset construction given in the following theorem.

Theorem 10.9 *Let $\nu = \langle \Sigma, \mathcal{Q}, \mathcal{I}, \mathcal{F}, f \rangle$ be a nondeterministic automaton. Let $\alpha = \langle \Sigma, \mathcal{P}(\mathcal{Q}), \mathcal{I}, \mathcal{G}, f' \rangle$ be the deterministic automaton with $\mathcal{Q}' \in \mathcal{G}$ if and only if $\mathcal{Q}' \cap \mathcal{F} \neq \emptyset$ for all $\mathcal{Q}' \subseteq \mathcal{Q}$, and $f'(\mathcal{Q}', \sigma) = f(\mathcal{Q}', \sigma)$. Then $L(\alpha) = L(\nu)$.*

Proof: The verification of this is routine. □

The concept of nondeterministic automaton is easily generalized to Moore machines. We leave the details to the reader.

10.7 Expression Automata

Expression automata are useful generalizations of nondeterministic automata. In particular, we use them here in order to describe an easy algorithm for finding regular expressions accepted by finite automata. The following is based on [20] and an improvement introduced in [144].

An *expression automaton* is a quintuple $\eta = \langle \Sigma, \mathcal{Q}, \mathcal{I}, \mathcal{F}, R \rangle$, where Σ, $\mathcal{Q}$, $\mathcal{F}$, and $\mathcal{I}$ are as in a nondeterministic automaton, and R, the *transition relation*, is a finite subset of $\mathcal{Q} \times \mathcal{RE}\mathcal{X}_\Sigma \times \mathcal{Q}$, where $\mathcal{RE}\mathcal{X}_\Sigma$ is the family of regular expressions over Σ. If $(q, X, q') \in R$, then we draw a directed edge from q to q' and label it with X. We interpret such an edge as follows: The automaton η can move from q to q' whenever any word $w \in X$ is applied to it. A word $w \in \Sigma^*$ is *accepted* by η if and only if there are the following items: a state $q \in \mathcal{I}$; a state $q' \in \mathcal{F}$; words $w_1, w_2, \ldots, w_k$ in Σ^* such that $w = w_1 w_2 \ldots w_k$; states $q_1 = q, q_2, \ldots, q_k, q_{k+1} = q'$ in $\mathcal{Q}$; and expressions $X_1, X_2, \ldots, X_k$ such that $w_i \in X_i$ and $(q_i, X_i, q_{i+1}) \in R$ for $1 \leq i \leq k$. In case all these conditions are satisfied, we say that there is a successful path spelling w from q to q'. The *language accepted* by η is the set of all words accepted by η.

An expression automaton is said to be *normalized* if there is at most one transition of the type (q, X, q') for every pair (q, q') of states from $\mathcal{Q}$. Given any expression automaton η we can easily find a normalized expression automaton η' that accepts the same language as η. Suppose $(q, X_1, q'), \ldots, (q, X_m, q')$ are all the transitions from q to q' in η. We remove

them all, and add the transition (q, X, q'), where $X = X_1 \cup \ldots \cup X_m$. Clearly, the language accepted is not changed by this transformation.

We now describe an algorithm for finding a regular expression for the language accepted by a normalized expression automaton. Given a normalized expression automaton $\eta = (\Sigma, \mathcal{Q}, \mathcal{I}, \mathcal{F}, R)$, we first enlarge η by adding two states to it. The resulting expression automaton η' is defined by

$$\eta' = (\Sigma, \mathcal{Q} \cup \{i, t\}, \{i\}, \{t\}, R'),$$

where

$$R' = R \cup \{(i, \varepsilon, q) \mid q \in \mathcal{I}\} \cup \{(q, \varepsilon, t) \mid q \in \mathcal{F}\}.$$

It is clear that the language accepted by η' is the same as that accepted by η. We have merely introduced a new initial state i from which every state of $\mathcal{I}$ can be reached by an empty-word transition, and a new terminal state t that is reachable from each accepting state in $\mathcal{F}$ by an empty-word transition. There is a successful path spelling w from i to t in η' if and only if there is a successful path spelling w from one of the initial states in $\mathcal{I}$ to one of the final states in $\mathcal{F}$. This can be easily formalized.

Next we show that any state of η' that belongs to $\mathcal{Q}$ can be removed without changing the language accepted. States i and t are not removed. We apply this procedure of removing states from $\mathcal{Q}$ until none is left; the

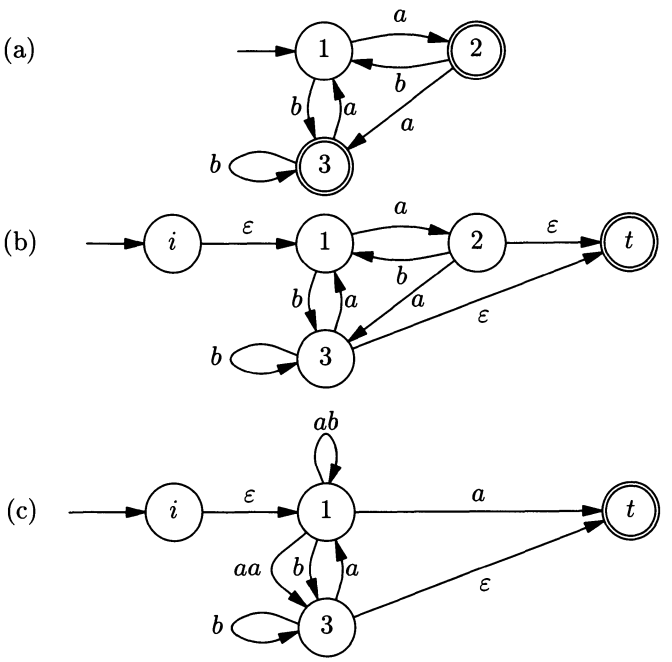


FIGURE 10.9. Finding a regular expression by state elimination.

final expression automaton so obtained has only the two states i and t and only one entry (i, X, t) in R. The language accepted by the original automaton η is precisely $L(X)$.

The removal of a single state p from $\mathcal{Q}$ is done as follows: For each triple of transitions of the type $((q, X, p), (p, Y, p), (p, Z, q'))$, we remove the transitions (q, X, p) and (p, Z, q') and add the transition (q, XY^*Z, q'). Finally, state p and the transition (p, Y, p) are also removed. If there are no transitions of the type (p, Y, p), then, for each pair $((q, X, p), (p, Z, q'))$ of transitions, we remove (q, X, p) and (p, Z, q') and add (q, XZ, q'). We claim that we have not changed the language accepted by doing this removal of state p. First, any path from q to q' that does not go through p still exists after the modification. Second, any path that does go through p has been accounted for by the expression XY^*Z, or by the expression XZ. No paths have been added unless equivalent paths existed in the original automaton. Hence the language accepted is the same.

We illustrate this construction with the example of Figure 10.9 and Figure 10.10. The original automaton of Figure 10.9(a) is deterministic. First, we add the initial and terminal states i and t as in Figure 10.9(b). The

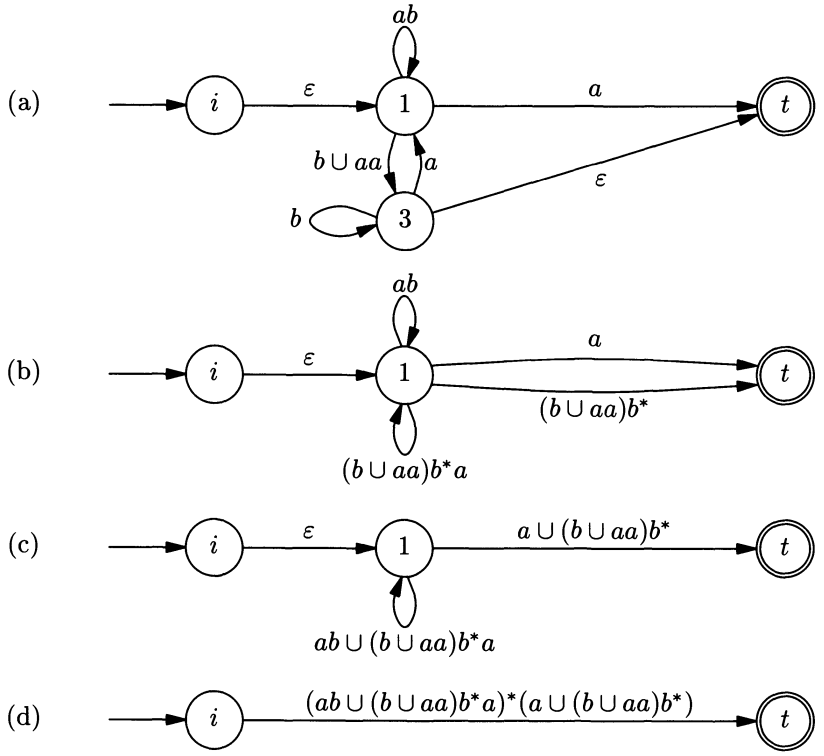


FIGURE 10.10. Figure 10.9 continued.

elimination of state 2 yields the graph of Figure 10.9(c). The normalized version of Figure 10.9(c) is shown in Figure 10.10(a). Next, we eliminate state 3, normalize, and finally eliminate state 1, as shown in Figure 10.10. Thus a regular expression for the language accepted by the automaton is

$$(ab \cup (b \cup aa)b^*a)^*(a \cup (b \cup aa)b^*).$$

Chapter 11

Behaviors and Realizations

In previous chapters we have discussed the analysis of a network when it is started in a stable state and some inputs are changed, and then held constant at their new values. This is only part of the analysis problem. A complete analysis also involves the network behavior in response to a *sequence* of input changes, some of which may occur while the network is unstable, if fundamental-mode operation is not used. We consider such behaviors in this chapter.

We define the concept of "realization" of a specification behavior by an implementation behavior; this includes deadlock and livelock phenomena in candidate implementations, and choice in specifications. The concept of realization is needed in Chapter 13, where we show that some specifications are not realizable under certain delay assumptions. This concept is also required in Chapter 14, where we discuss the verification process.

The early work on asynchronous circuits used the so-called primitive flow tables to describe behaviors [66, 67]. The problem of realizing a flow table by a logic circuit has been studied by many researchers; see, for example, [135] and the more recent work of [38, 109]. Many modern asynchronous design techniques do not use flow tables; see Chapter 15 for references to these techniques. Our approach has been influenced by the work of [19] and [146, 147]. Some related work can also be found in [62] and [101], but there are considerable differences in our approach. We develop a model closely related to finite automata; this allows us to exploit some well-known ideas from automaton and language theory. Also, we remove some of the possible ambiguities of flow tables by making transitions between states explicit. This will be discussed further in Chapter 12.

How To Read This Chapter

Section 11.3 involves relatively simple ideas, but is rather technical. It can be omitted on first reading; instead, the reader may assume that the specification and implementation have identical input alphabets and identical output alphabets. Section 11.7 deals with choice in specifications. This topic requires further research; consequently, this section is primarily of interest to the researcher.

11.1 Motivation

The following intuitive distinction is made between implementations and specifications: An implementation describes what *might* happen in a circuit, whereas a specification states what *should* happen.

To motivate our definitions of specification and implementation, we begin with some very simple informal examples from combinational logic design. We have in mind a scenario where a designer is given some sort of specification and is to design a circuit satisfying this specification.

We assume that any combinational circuit is adequately modeled by a Boolean function; we call this a (proposed) implementation (function) f_B. In the simplest case, the specification is also modeled by a Boolean function f_A. To check whether an implementation f_B realizes a specification f_A, we need to compare the two functions. Thus "realizes" is a binary relation on Boolean functions; in this case the relation is simply equality. For example, suppose the specification requires that there be a binary input X and a binary output O, and that the function f_A be complementation, i.e., O is to be equal to $\overline{X}$. An inverter circuit is an acceptable implementation, because its Boolean function f_B is also complementation.

Consider now a parallel situation where we are to design an asynchronous circuit to meet some specification. We assume that any asynchronous circuit is adequately represented by a formal model we call "behavior";[1] we call this a (proposed) implementation (behavior) B. In the simplest case, the specification is also modeled by a behavior A. To check whether an implementation B realizes a specification A, we need to compare the two behaviors. Thus "realizes" is a binary relation on behaviors; in general, this relation is much more complex than equality.

The next example raises some issues concerning the inputs and outputs of an implementation as they relate to a specification. For every input of a specification, there must be a corresponding input in the implementation. In general, however, the implementation B may have more inputs than the specification A. Such inputs of B are simply "not used" if B is to realize A. More precisely, each such input is fixed at either 0 or 1. Similarly, for every output of a specification, there must be a corresponding output in the implementation, but the implementation may also have some other outputs. We simply "do not look" at the unused outputs.

To illustrate this, suppose the specification is the complementation function, and the proposed implementation is a half-adder. The half-adder has inputs X_1 and X_2 and produces outputs $O_1 = X_1 \oplus X_2$ (sum) and $O_2 = X_1X_2$ (carry). If we set the input X_1 to 1 permanently and ignore

[1] In general, it is not a simple problem to determine an appropriate behavior from a circuit. Such a behavior depends on the network model, on the race model, and on the restrictions placed on the environment. In Chapters 12 and 14 we show how to derive certain types of behaviors.

the output O_2, then the output $O_1 = 1 \oplus X_2 = \overline{X_2}$ realizes the complement of X_2. Similar ideas are introduced for asynchronous circuits.

The remainder of the chapter is structured as follows. Our formal concept of behavior is introduced in Section 11.2; it models all the signal transitions that might occur in a network. In Section 11.3 we handle "unused" inputs and outputs of one behavior (considered as the implementation) when it realizes another behavior (considered as the specification). In Sections 11.4–11.6 we formalize the concept of realization. In Section 11.7, we briefly discuss specifications with choice.

11.2 Behaviors

From now on, the word "behavior" means the formal mathematical object defined below, unless explicitly stated otherwise. This definition is intended to model all the possible signal transitions that might occur in a network. It is consistent with the formal definition given in Chapter 4 of a network with a binary domain, but (a) it represents the network states by an abstract set rather than by a set of binary vectors, and (b) it explicitly introduces the external outputs of the network.[2]

Example 1

The following running example is used to motivate and explain our definitions. Suppose we wish to describe the possible changes that might occur in the binary network model of an inverter. Figure 11.1(a) shows the inverter circuit, together with a set of variables that permit us to define its circuit graph. Since we need to deal with external outputs now, we modify the definition of circuit graph slightly by adding output vertices and output wires. Thus the circuit graph of the inverter consists of a single input vertex X_1, an input-delay vertex x_1, gate vertex s_1, wire vertices w_1 and w_2, and an output vertex O_1.

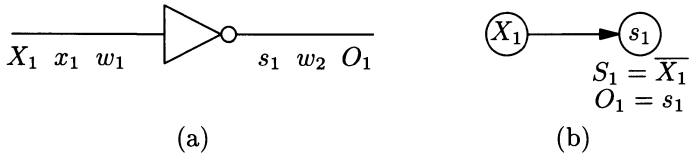


FIGURE 11.1. Inverter circuit and its network model.

[2]Previously, the external outputs did not play a major role and were not mentioned very often. Any gate or wire variable in a circuit graph of Chapter 4 may be considered as an external output. Its dependence on the input excitations and state variables is given by one of the circuit equations in the network model.

Suppose we select only vertex variable s_1 to be the state variable. Then we have the binary network model of Figure 11.1(b):

$$N = \langle \{0,1\}, \{X_1\}, \{s_1\}, \mathcal{E}, F \rangle,$$

where the excitation function ($\mathcal{E}$ in the network model) is $S_1 = \overline{X_1}$ and the output O_1 is given by the circuit equation (F in the model) $O_1 = s_1$.

A *behavior* is a 7-tuple $B = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$, where

- $X = (X_1, \ldots, X_n)$, $n \geq 0$, is a *vector of input excitation variables*, and the corresponding *set* of variables is $\mathcal{X} = \{X_1, \ldots, X_n\}$;
- $\mathcal{R}$ is a finite, nonempty set of *internal states*;
- $O = (O_1, \ldots, O_p)$, $p \geq 0$, is a *vector of output variables* and the corresponding *set* of variables is $\mathcal{O} = \{O_1, \ldots, O_p\}$;
- $\mathcal{Q} = \{0,1\}^n \times \mathcal{R}$ is the set of *total states*,[3] where the first component of a total state is of the form $(a_1, \ldots, a_n)$ and the binary value a_i is associated with the variable X_i, for $i = 1, \ldots, n$;
- $q_1 \in \mathcal{Q}$ is the *initial (total) state*;
- $\mathcal{T} \subseteq (\mathcal{Q} \times \mathcal{Q}) - \{(q,q) \mid q \in \mathcal{Q}\}$ is the set of *transitions*;
- ψ is the *output function*, $\psi : \mathcal{Q} \to \{0,1\}^p$, where, for any $q \in \mathcal{Q}$, $\psi(q)$ is of the form $(a_1, \ldots, a_p)$, and the binary value a_i is associated with the variable O_i, for $i = 1, \ldots, p$.

Example 1 (continued)

Let $B = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$, where $X = (X_1)$, $\mathcal{R} = \{0,1\}$, $O = (O_1)$, $\mathcal{Q} = \{0,1\} \times \{0,1\}$, $q_1 = (0,1)$, the transitions are as shown in Figure 11.2, and the output function ψ is given by the expression $\psi((a,b)) = b$, i.e., the output value is equal to the state value. In Figure 11.2, we show the initial state by an incoming arrow. An edge between q and q' with two arrowheads represents two transitions: from q to q' and from q' to q.

The behavior B may be derived from the inverter network of Figure 11.1(b) operated in an unrestricted environment. The following is an informal description of that behavior: The behavior state $(0,1)$ corresponds to a stable state of the network. When the input changes

[3]Strictly speaking, the concept of total state is redundant, since it is derived from the size of X and the set $\mathcal{R}$. We retain this concept for notational convenience.

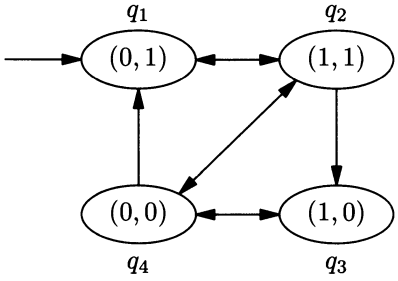


FIGURE 11.2. Behavior of inverter.

in this state, the state becomes $(1,1)$. Now the internal state may change since the inverter is unstable, and state $(1,0)$ can be reached. In state $(1,1)$, the environment may change the input excitation. If that happens, the state can become $(0,1)$ again—if the input pulse was very short and was ignored by the inertial delay of the inverter—or the input excitation and the internal state may both change—if the pulse was long enough to be recognized; this would result in state $(0,0)$. The remaining transitions are similarly explained.

Let $\mathcal{V} = \mathcal{X} \cup \mathcal{O}$ be the set of *(external) variables* of the behavior. The set $\Sigma = \mathcal{P}(\mathcal{V}) - \{\emptyset\}$, where $\mathcal{P}(\mathcal{V})$ is the power set of $\mathcal{V}$, is called the *alphabet* of B.

Let $l : \mathcal{Q} \to \{0,1\}^n \times \{0,1\}^p$ be a function that associates a *state label* $l(q) \in \{0,1\}^n \times \{0,1\}^p$ with each state $q = (a,r) \in \mathcal{Q}$, where $a \in \{0,1\}^n$ and $r \in \mathcal{R}$. The label of $q = (a,r)$ is the input excitation state a together with the output state $\psi(q)$; we may think of the label of a total state as the externally visible information about that state. A label is denoted by a vector $c = a{\cdot}b = a_1 \ldots a_n{\cdot}b_1 \ldots b_p$, where $a_i, b_j \in \{0,1\}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, p$. We also write the label of q as $l(q) = X(q){\cdot}\psi(q)$, where $X(q)$ is the input component of q and $\psi(q)$ is the output associated with q. Note that two or more distinct total states may have the same label.

The *expanded state* of a behavior is an element of $\mathcal{Q} \times \{0,1\}^p = (\{0,1\}^n \times \mathcal{R}) \times \{0,1\}^p$. We frequently use the expanded state in order to have the output vector associated with a total state easily available. For convenience, we write the expanded state $((a,b),c)$ as $a{\cdot}b{\cdot}c$. The label of an expanded state $a{\cdot}b{\cdot}c$ consists of its first and third components, i.e., it is $a{\cdot}c$.

A change in the input component or in the internal-state component (or in both) of a total state is represented by a transition to another total state. We stress that each transition (q, q') involves a change in at least one of the two components of q; thus transitions of the form (q, q) are not allowed. In a transition, we may have an input change, an internal state change, or both. Note, however, that the output cannot change by itself in a transition; either the input or the internal state must change if the output changes.

A transition is said to be an *invisible transition* if only the internal state changes but not the input nor the output, i.e., if $l(q) = l(q')$; otherwise, it is a *visible transition*. We define the "tag mapping" τ from $\mathcal{T}$ to $\Sigma \cup \{\varepsilon\}$ as follows. The *tag* of a visible transition (q, q') is denoted by $\tau(q, q')$, and is that element $\sigma \in \Sigma$ that consists of all the external variables that change in that transition. The value of $\tau(q, q')$ for an invisible transition (q, q') is the empty word ε. Note that ε is not a letter of Σ.

Example 1 (continued)

In Figure 11.3 we show the expanded-state behavior corresponding to the behavior of Figure 11.2. The transition tags are redundant, since they can be deduced from the two states connected by each edge; however, these tags are shown for convenience.

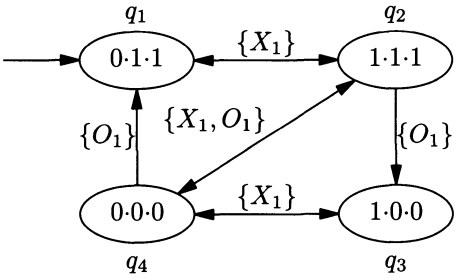


FIGURE 11.3. Expanded-state behavior of inverter.

A transition (q, q') is said to be an *input transition* if the input changes in going from q to q', but the internal state does not change. (The output may or may not change in an input transition.) A transition is called an *internal-state transition* if the internal state changes but not the input. (The output may or may not change in an internal-state transition.) We also talk about *X transitions*, *O transitions*, *XO transitions*, and *ε transitions*—depending on the tag of the transition. In summary, the transitions can be classified as shown in Table 11.1.

TABLE 11.1. Types of transitions.

what changes:			transition	tag	visible
X	$\mathcal{R}$	O	type	type	
yes	no	no	input	X	yes
yes	no	yes	input	XO	yes
no	yes	no	internal-state	ε	no
no	yes	yes	internal-state	O	yes
yes	yes	no	mixed	X	yes
yes	yes	yes	mixed	XO	yes

A state of a behavior is said to be *static* if it has no outgoing internal-state transitions; otherwise, it is *dynamic*. Note that a behavior cannot leave a static state unless the input changes. In contrast to this, a dynamic state has at least one transition not involving any input change.

Example 1 (continued)

In Figure 11.3, the transition (q_1, q_2) is an input transition of type X, (q_2, q_3) is an internal-state transition of type O, and (q_4, q_2) is a mixed transition of type XO. States q_1 and q_3 are static, and q_2 and q_4 are dynamic.

Given any behavior $B = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$, we associate with it a nondeterministic finite automaton β with empty-word transitions; this automaton is called the *behavior automaton* of B and is defined as follows: $\beta = \langle \Sigma, \mathcal{Q}, q_1, \mathcal{F}, f \rangle$, where

- Σ, the behavior's alphabet, is the input alphabet of β;
- $\mathcal{Q}$, the behavior's state set, is the state set[4] of β;
- $q_1 \in \mathcal{Q}$, the behavior's initial state, is the initial state of β;
- $\mathcal{F}$ is the set of accepting states and it is always equal to $\mathcal{Q}$;
- $f \subseteq \mathcal{Q} \times (\Sigma \cup \{\varepsilon\}) \times \mathcal{Q}$ is the automaton's transition relation: $(q, \sigma, q') \in f$ if and only if $(q, q') \in \mathcal{T}$ and $\tau(q, q') = \sigma$.

Example 1 (continued)

The behavior of Figure 11.3 redrawn as a behavior automaton is shown in Figure 11.4. Since all the states are accepting, they are not marked in any special way.

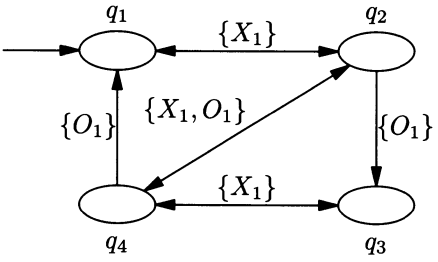


FIGURE 11.4. A behavior automaton.

[4]Here $\mathcal{Q}$ is viewed as an abstract set of elements, i.e., the fact that $\mathcal{Q} = \{0,1\}^n \times \mathcal{R}$ in the behavior is irrelevant in its automaton.

The set of all the words accepted by a behavior automaton β, i.e., its language, is denoted by $L(\beta)$. This language is obviously regular since $\mathcal{Q}$ is finite. Because each state is an accepting state, this language is *prefix-closed*, i.e., satisfies $L = prefL$, where $prefL$ is the set of all prefixes of words in L.

The *language of a behavior* B is denoted by $L(B)$ and is defined to be the language $L(\beta)$ of the corresponding automaton β.

11.3 Projections of Implementations to Specifications

In this section we formalize the concepts of "unused" inputs and outputs of an implementation behavior with respect to a given specification behavior. This section may be omitted on first reading.

If an implementation B' is to realize a specification A, then each input of A must be represented by an input of B'. Thus, we need to map a subset of the set of inputs of B' to the set of inputs of A. We need a similar mapping for the outputs. Also the starting state q'_1 of B' must represent the starting state q_1 of A, i.e., we require that $l'(q'_1)$ projected to A should equal $l(q_1)$. In case these conditions are satisfied, we say that there is a *projection of* B' *to* A.

Once we have selected the appropriate inputs of an implementation B', we fix the unused inputs at the value they have in the initial total state; clearly, we are not allowed to change these inputs as long as B' is realizing specification A. This operation is formalized as follows. Let $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$ be a specification behavior, where $X = (X_1, \ldots, X_h), h \geq 0$, and $O = (O_1, \ldots, O_k), k \geq 0$. Let $B' = \langle X', \mathcal{R}', O', \mathcal{Q}', q'_1, \mathcal{T}', \psi' \rangle$ be an implementation behavior, where $X' = (X'_1, \ldots, X'_n)$, $n \geq h$, and $O' = (O'_1, \ldots, O'_p)$, $p \geq k$.

Without loss of generality, we suppose that input X'_i of the implementation represents input X_i of the specification, for $i = 1, \ldots, h$. Then we may assume that $X' = XX''$, where $X = (X_1, \ldots, X_h)$, and $X'' = (X'_{h+1}, \ldots, X'_n)$. Thus X'' represents the unused inputs. Now each state $q' \in \mathcal{Q}'$ has the form $q' = aa''\cdot r'$, where $a \in \{0,1\}^h$, $a'' \in \{0,1\}^{n-h}$ and $r' \in \mathcal{R}'$. In this notation, the initial state is denoted by $q'_1 = bb''\cdot r'_1$.

The *input projection* of an implementation behavior

$$B' = \langle X' = XX'', \mathcal{R}', O', \mathcal{Q}', q'_1 = bb''\cdot r'_1, \mathcal{T}', \psi' \rangle$$

to the subvector X of X' is now defined as follows:

$$B' \Downarrow_X = \langle X, \mathcal{R}', O', \mathcal{Q}' \Downarrow_X, q'_1 \Downarrow_X, \mathcal{T}' \Downarrow_X, \psi' \Downarrow_X \rangle$$

where

- $\mathcal{Q}' \Downarrow_X = \{a\cdot r' \mid ab''\cdot r' \in \mathcal{Q}'\}$,

- $q'_1 \Downarrow_X = b{\cdot}r'_1$,
- $\mathcal{T}' \Downarrow_X = \{(a{\cdot}r, \hat{a}{\cdot}\hat{r}) \mid (ab''{\cdot}r, \hat{a}b''{\cdot}\hat{r}) \in \mathcal{T}'\}$,
- $(\psi' \Downarrow_X)(a{\cdot}r') = \psi'(ab''{\cdot}r')$.

Basically, all the inputs not in $\mathcal{X}$ are removed. The second part (the last $n - h$ components) of the input vector is first fixed at the value b'' that this part has in the initial state q'_1. All the states that differ from b'' in the second part of the input are removed, along with all the transitions from and to these states. Since the second part of the input vector is now fixed at b'' in all the states that remain, the components from the second part can be dropped. Thus the new state label is the old label with all the components in $\mathcal{X}' - \mathcal{X}$ removed.

TABLE 11.2. Transition table for NOR latch.

q	$X_1X_2{\cdot}r{\cdot}O$	$\{X_1\}$	$\{X_2\}$	$\{X_1, X_2\}$	$\{O_1\}$	$\{O_2\}$	$\{O_1, O_2\}$
q_1	00·01·01	q_4	q_9	q_{12}	-	-	-
q_2	00·10·10	q_{11}	q_3	q_{13}	-	-	-
q_3	01·10·10	q_{13}	q_2	q_{11}	-	-	-
q_4	10·01·01	q_1	q_{12}	q_9	-	-	-
q_5	11·00·00	q_8	q_{10}	q_6	-	-	-
q_6	00·00·00	-	-	-	q_2	q_1	q_7
q_7	00·11·11	-	-	-	q_1	q_2	q_6
q_8	01·00·00	-	-	-	q_3	-	-
q_9	01·01·01	-	-	-	-	q_8	-
q_{10}	10·00·00	-	-	-	-	q_4	-
q_{11}	10·10·10	-	-	-	q_{10}	-	-
q_{12}	11·01·01	-	-	-	-	q_5	-
q_{13}	11·10·10	-	-	-	q_5	-	-

Example 2

To illustrate input projection, consider the behavior of Table 11.2. (In the next chapter, we show how this behavior is derived from a network model; for now its meaning is of no significance.) This is an implementation behavior represented—because of its size—by a table, rather than a graph. The total state consists of two input variables and two internal-state variables, which are also the output variables. (Thus the output component is always equal to the internal-state component.) The initial state is q_6.

To illustrate the removal of unused inputs, suppose that only input X_1 is needed for some specification. Then we would remove all the transitions in columns $\{X_2\}$ and $\{X_1, X_2\}$. We also remove all the

TABLE 11.3. Illustrating input projection.

q	$X_1 \cdot r \cdot O$	$\{X_1\}$	$\{O_1\}$	$\{O_2\}$	$\{O_1, O_2\}$
q_1	0·01·01	q_4	-	-	-
q_2	0·10·10	q_{11}	-	-	-
q_4	1·01·01	q_1	-	-	-
q_6	0·00·00	-	q_2	q_1	q_7
q_7	0·11·11	-	q_1	q_2	q_6
q_{10}	1·00·00	-	-	q_4	-
q_{11}	1·10·10	-	q_{10}	-	-

states in which X_2 has the value 1, because they are not reachable from the initial state q_6 in which X_2 has the value 0. Finally, we remove the X_2 component from the labels, obtaining Table 11.3.

We may also "erase" unused outputs, since we never look at them. Without loss of generality, suppose that output O_i of the specification is represented by output O'_i of the implementation, for $i = 1, \ldots, k$. Consequently, we may assume that $O' = OO''$, where $O = (O_1, \ldots, O_k)$, and $O'' = (O'_{k+1}, \ldots, O'_p)$. Thus O'' represents the unused outputs. Each output vector now has the form $O = cc''$, where $c \in \{0,1\}^k$, and $c'' \in \{0,1\}^{p-k}$ The *output projection* of an implementation behavior

$$B' = \langle X', \mathcal{R}', O' = OO'', \mathcal{Q}', q'_1, \mathcal{T}', \psi' \rangle$$

to the subvector O of O' is now defined as follows:

$$B' \downarrow_O = \langle X', \mathcal{R}', O, \mathcal{Q}', q'_1, \mathcal{T}', \psi' \downarrow_O \rangle,$$

where $(\psi' \downarrow_O)(q) = c$ if $\psi'(q) = cc''$.

Note that the state labels are now changed: the new label is the old label with all the components in $\mathcal{O}' - \mathcal{O}$ removed. Also, the transition tags will be modified. The outputs not in $\mathcal{O}$ are removed from each tag; if the resulting set is empty, the transition becomes invisible, and the new tag is ε.

Example 2 (continued)

Consider the behavior of Table 11.3. Suppose that only output O_2 is used for some specification. Consequently, we remove the O_1 component from the labels, and erase O_1 from the transition tags. This results in one ε column and a second column labeled $\{O_2\}$ as shown in Table 11.4.

It is easily verified that the input and output projections can be done in either order and produce the same result. Let $B' \Downarrow_{X \cdot O}$—the behavior resulting from B' after the appropriate removal of unused inputs and outputs—be defined as

$$B' \Downarrow_{X \cdot O} = (B' \Downarrow_X) \downarrow_O = (B' \downarrow_O) \Downarrow_X . \tag{11.1}$$

TABLE 11.4. Illustrating output projection.

q		$\{X_1\}$	ε	$\{O_2\}$	$\{O_2\}$
q_1	0·01·1	q_4	-	-	-
q_2	0·10·0	q_{11}	-	-	-
q_4	1·01·1	q_1	-	-	-
q_6	0·00·0	-	q_2	q_1	q_7
q_7	0·11·1	-	q_1	q_2	q_6
q_{10}	1·00·0	-	-	q_4	-
q_{11}	1·10·0	-	q_{10}	-	-

For later use, we also define the removal of certain letters from a language. The *restriction of a letter* σ of an alphabet $\Sigma = \mathcal{P}(\mathcal{V}) - \{\emptyset\}$ to a subset $\mathcal{U}$ of $\mathcal{V}$ is defined as follows:

$$\sigma\dagger\mathcal{U} = \begin{cases} \varepsilon & \text{if } \sigma \cap \mathcal{U} = \emptyset, \\ \sigma \cap \mathcal{U} & \text{otherwise.} \end{cases}$$

This operation is extended to words as follows: $\varepsilon\dagger\mathcal{U} = \varepsilon$, and, for $w \neq \varepsilon$, $(w\sigma)\dagger\mathcal{U} = (w\dagger\mathcal{U})(\sigma\dagger\mathcal{U})$. Finally, the operation is also extended to languages by $L\dagger\mathcal{U} = \{w\dagger\mathcal{U} \mid w \in L\}$.

The restriction operation has the following interpretation. Suppose we have a network with variable set $\mathcal{V}$, but we only look at the variables in $\mathcal{U}$ and ignore all the others. Then, if the network history is represented by some word $w \in \mathcal{V}^*$, we would only see the word $w\dagger\mathcal{U}$.

11.4 Relevant Words

In defining the notion of realization of A by B', we need to check, among other things, that B' does not produce any outputs not permitted by A. For this purpose, we need not consider *all* the words of $L(B')$, but only those that are "relevant" to A. In general, a specification behavior imposes some constraints on the environment in which the implementation is operated. To illustrate this, suppose $X_2O_1X_1O_2$ is in $L(B')$, but no word in $L(A)$ allows X_2 to precede X_1. Then the word $X_2O_1X_1O_2$ is not relevant to A.

11.4.1 Same Input and Output Alphabets

Let $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi\rangle$ be a specification behavior, and let $B' = \langle X', \mathcal{R}', O', \mathcal{Q}', q_1', \mathcal{T}', \psi'\rangle$ be an implementation behavior. In the interest of clarity, we first define the concept of relevant words under the assumption that A and B' have the same input and output alphabets, i.e., that $X' = X$ and $O' = O$. Thus $B' = \langle X, \mathcal{R}', O, \mathcal{Q}', q_1', \mathcal{T}', \psi'\rangle$, and also $\Sigma = \Sigma'$. This restriction is removed later.

Let $x \in \Sigma^*$ be a word in $L(A)$. A set $\mathcal{Y}$ of input variables, $\mathcal{Y} \subseteq \mathcal{X}$, is *applicable* in A after x if $\mathcal{Y} = \emptyset$ or if there exists a $\sigma \in \Sigma$ such that $x\sigma \in L(A)$ and $\sigma \cap \mathcal{X} = \mathcal{Y}$. In other words, suppose an input/output word x occurs in A. Suppose further that it is possible to apply the input change $\mathcal{Y}$ after x according to the specification A; this input change may or may not be accompanied by some output change. Then the input change $\mathcal{Y}$ is applicable in A after x. Note that the empty subset of $\mathcal{X}$ is applicable after x; in that case there may or may not be a $\sigma \subseteq \mathcal{O}$ such that $x\sigma \in L(A)$.

We say that $w' \in L(B')$ is *relevant* to A if $w' \in L(A)$, or $w' = x'\sigma'$, where $x' \in L(A)$, $\sigma' \in \Sigma$, and $\sigma' \cap \mathcal{X}$ is applicable to A after x'. Let $L(B'/A)$ be the set of all the words of B' that are relevant to A.

The motivation for this definition is as follows. Suppose we change the inputs in the set $\mathcal{Y}$ (possibly empty) in both A and B' after input/output word x has occurred in A and in B'. Consider now any σ' such that $\sigma' \cap \mathcal{X} = \mathcal{Y}$. We need to consider $x\sigma'$, because the outputs accompanying the input change $\mathcal{Y}$ in B' may not be permitted by A; in that case B' would not be a good realization of A. Note that we are looking for the first violation of the specification; any word with $x\sigma'$ as prefix violates the specification if $x\sigma'$ does.

11.4.2 Different Input and Output Alphabets

The definition of relevant words is now generalized to the case where the alphabets of A and B' are not equal. The reader who omitted Section 11.3 may now proceed to Section 11.5. Our main goal in the remainder of this section is to show that input and output projections preserve relevant words in a certain sense. Therefore, we can perform the input and output projections first and then consider relevant words.

Let $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$ be a specification behavior, and let $B' = \langle X', \mathcal{R}', O', \mathcal{Q}', q_1', \mathcal{T}', \psi' \rangle$ be an implementation behavior. We say that a word $x' \in L(B')$ is *consistent* with A if the word $x = x' \dagger_{(\mathcal{X}' \cup \mathcal{O})}$ obtained from x' by erasing the unused outputs (i.e., the outputs in $\mathcal{O}' - \mathcal{O}$) is in $L(A)$. Under these circumstances, the input/output word x of the specification is represented by the word x' of the implementation. Note that if a word w is consistent with A, then every prefix of w is also consistent with A.

We say that $w' \in L(B')$ is *relevant* to A if $w' = \varepsilon$ or $w' = x'\sigma'$, where x' is consistent with A, and either $\sigma' = \varepsilon$ or $\sigma' \in \Sigma'$ and $\sigma' \cap \mathcal{X}'$ is applicable to A after $x' \dagger_{(\mathcal{X}' \cup \mathcal{O})}$. Note that every word of $L(B')$ that is consistent with A is relevant to A.

Observe that the set of words of B' that are relevant to A is preserved under input projection, i.e.,

$$L(B'/A) = L(B' \Downarrow_X /A). \tag{11.2}$$

Certainly, the right-hand side is a subset of the left-hand side, since the input projection can only remove paths in the graph of B'. On the other

hand, every word in $L(B'/A)$ uses only inputs from $\mathcal{X}$. Hence it is not removed by the projection operation. One also verifies that

$$L(B')\dagger_{\mathcal{O}} = L(B' \downarrow_O)$$

and

$$(L(B'/A))\dagger_{\mathcal{O}} = L(B' \downarrow_O /A). \tag{11.3}$$

The language $(L(B'/A))\dagger_{\mathcal{O}}$ is the set of all words of B' relevant to A, from which the unused outputs have been erased. The following proposition shows that this set of words is preserved by the input and output projections.

Proposition 11.1 $(L(B'/A))\dagger_{\mathcal{O}} = L(B' \Downarrow_{X \cdot O} /A)$.

Proof: By 11.2 we have

$$(L(B'/A))\dagger_{\mathcal{O}} = (L(B' \Downarrow_X /A))\dagger_{\mathcal{O}}.$$

By 11.3

$$(L(B' \Downarrow_X /A))\dagger_{\mathcal{O}} = L((B' \Downarrow_X) \downarrow_O /A).$$

Finally, by 11.1

$$L((B' \Downarrow_X) \downarrow_O /A) = L(B' \Downarrow_{X \cdot O} /A). \qquad \square$$

From now on, we assume that the appropriate input and output projections have been performed. Consequently, *we assume that the input excitation and output vectors of B' are identical to those of A.*

11.5 Proper Behaviors

So far we have not imposed any restrictions on behaviors, but we are about to do so now.

A behavior $B = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$ is *deterministic* if and only if its associated automaton is (incomplete) deterministic.[5] Note that the word "deterministic" is used here in the automaton-theoretic sense. Being deterministic does not prevent a behavior from producing any one of several outputs in a given state.

We write $q \xrightarrow{w} q'$ if state q' can be reached from state q by a path spelling w in B. A behavior is said to be *proper* if, whenever $q_1 \xrightarrow{w} q_2$ and $q_1 \xrightarrow{w} q_2'$, and $q_2 \xrightarrow{\sigma} q_3$, for some $\sigma \in \Sigma$, then we also have $q_2' \xrightarrow{\sigma} q_3'$, for some $q_3' \in \mathcal{Q}$. Note that every deterministic behavior is proper.

---

[5]Recall that an incomplete deterministic automaton has no ε transitions and for each state $q \in \mathcal{Q}$ and each letter $\sigma \in \Sigma$ there is at most one transition (q, q') with $\tau(q, q') = \sigma$.

If a behavior B is proper, we can construct a deterministic behavior D such that $L(D) = L(B)$. This is done using a sort of subset construction. For any $w \in \Sigma^*$, let $[w]$ denote the set of all total states reachable by w from q_1, i.e., let $[w] = \{q \in \mathcal{Q} \mid q_1 \xrightarrow{w} q\}$. Note that all the states in $[w]$ have the same input component, i.e., $q, q' \in [w]$ implies $X(q) = X(q')$. This is true because we start in the initial state and change exactly the same inputs in order to get to q as we do to get to q'. By the same argument, the output vectors associated with q and q' are the same. Altogether, q and q' must have the same label. Let $X([w])$ denote the input component of each state in $[w]$, and let $\psi([w])$ be the output component of each state in $[w]$. Furthermore, let $\mathcal{R}([w]) = \{r \in \mathcal{R} \mid q_1 \xrightarrow{w} X([w])\cdot r\}$. This is the set of all internal states of B that are reachable by w from q_1. Let

$$D = \langle X, \mathcal{R}', O, \mathcal{Q}', q_1', \mathcal{T}', \psi' \rangle,$$

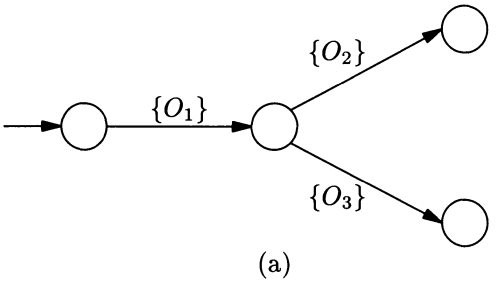
where

- $\mathcal{R}' = \{\mathcal{R}([w]) \mid w \in L(B)\}$;
- $\mathcal{Q}' = \{[w] \mid w \in L(B)\}$;
- $q_1 = [\varepsilon]$;
- $\mathcal{T}' = \{([w], [w\sigma]) \mid \text{for some } (q, q') \in \mathcal{T}, q \in [w], q' \in [w\sigma] \text{ and } \tau(q, q') = \sigma\}$;
- $\psi'([w]) = \psi([w])$.

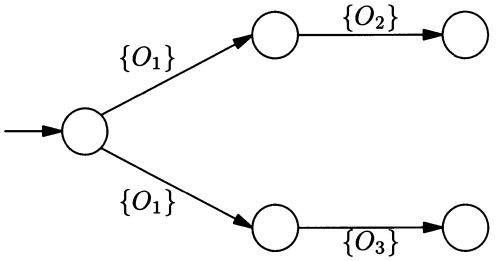
One verifies that D is well defined and that $L(D) = L(B)$. In view of this, every proper behavior can be replaced by a deterministic behavior with the same language.

We require that all specification behaviors be deterministic (and therefore proper) for the following reasons. When we describe the behavior of a circuit, we treat it as a "black box"; consequently, we can only observe a sequence of symbols from Σ. Consider the behaviors in Figure 11.5. One can certainly design a circuit in which first output O_1 is produced, and then the circuit makes the decision whether to produce O_2 or O_3, as in Figure 11.5(a). One can also design a circuit in which the decision whether O_2 or O_3 will eventually be produced is made before O_1 appears, as in Figure 11.5(b). However, in any history of operation of either circuit, one can only have the following words: ε, O_1, O_1O_2, and O_1O_3. Thus the distinction made by the two behaviors of Figure 11.5 cannot be tested by any input/output experiment. Therefore such properties are structural, not behavioral, and we consider the behaviors of Figure 11.5 equivalent.

The specification of Figure 11.6 is not proper. According to this specification, the circuit could produce the output word $\{O_1\}\{O_2\}$ and stop, or it could produce just the word $\{O_1\}$ and stop. There would then be two



(a)



(b)

FIGURE 11.5. Indistinguishable behaviors.

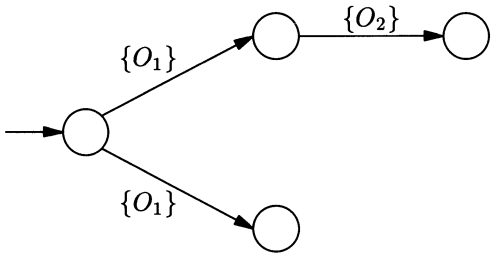


FIGURE 11.6. An improper specification.

distinct states reached by $\{O_1\}$, and these two states would have different outgoing transitions. We will be able to represent such situations by specifications with choice, which are the treated in Section 11.7.

We place a somewhat weaker restriction on implementation behaviors. A behavior is said to be *input-proper* if, whenever $q_1 \stackrel{w}{\rightarrow} q_2$ and $q_1 \stackrel{w}{\rightarrow} q_2'$, and there exists $(q_2, q_3) \in \mathcal{T}$ with $\tau(q_2, q_3) \cap \mathcal{X} \neq \emptyset$, then there also exists $q_3' \in \mathcal{Q}$ such that $(q_2', q_3') \in \mathcal{T}$ and $\tau(q_2', q_3') \cap \mathcal{X} = \tau(q_2, q_3) \cap \mathcal{X}$. In other

words, whether or not an input change is permitted in a given state should depend only on the word w leading to that state.

We argue that implementation behaviors should be input-proper. (This should apply to words relevant to a specification; we need not place any restrictions on words that are not relevant.) If an implementation does not satisfy this condition, then, with the same word, it can reach a state q in which an input change is permitted, or a state q' in which such a change is not permitted. However, the actions of the environment cannot depend on the invisible internal state of a circuit. Consequently, implementations that are not input-proper (with respect to words relevant to the specification) will not be considered.

On the other hand, implementations need not be "output-proper," because distinctions like those of Figure 11.5 can be made by some internal variables.

We illustrate the definitions with some examples. The following behavior is input-proper.

$$A_1 : q_1 \xrightarrow{\{X_1\}} q_2 \xrightarrow{\{O\}} q_3 \xrightarrow{\{X_1\}} q_4 \xrightarrow{\{X_2\}} q_5.$$

In contrast to this, the behavior below is not input-proper.

$$A_2 : q_1 \xrightarrow{\varepsilon} q_2 \xrightarrow{\{X\}} q_3.$$

The environment is not permitted to apply X in state q_1, but it is allowed to do so in state q_2. However, no information is provided by the circuit whether it is in state q_1 or q_2.

We remark that a behavior like that in the last example can be made input-proper by providing some timing information. Thus, the behavior

$$A_3 : q_1 \xrightarrow{\{10ns\}} q_2 \xrightarrow{\{X\}} q_3$$

is acceptable. Here, the environment waits for 10 ns; after that, it is free to change the input X. We will not be introducing any special notation for such timing information, but we may simply view this information as an additional input symbol.

In summary, *we assume from now on that all specification behaviors are deterministic and all implementation behaviors are input-proper (with respect to relevant words).*

We have the following order among the families of behaviors that we have defined:

behaviors

$\supset$ input-proper behaviors

$\supset$ proper behaviors

$\supset$ deterministic behaviors.

The following examples show that all the inclusions are indeed proper. The behavior

$$q_1 \xrightarrow{\varepsilon} q_2 \xrightarrow{\{X\}} q_3$$

is not input-proper. The behavior

$$q_1 \xrightarrow{\varepsilon} q_2 \xrightarrow{\{O\}} q_3$$

is input-proper, but not proper. The behavior with the transitions

$$q_1 \xrightarrow{\{X\}} q_2 \quad \text{and} \quad q_1 \xrightarrow{\{X\}} q_3$$

is proper, but not deterministic.

11.6 Realization

We now consider only deterministic specifications. The main question studied in this section is how to make precise the notion that an implementation behavior realizes a specification behavior.

11.6.1 Safety and Capability

Assuming that a projection of an implementation to a specification exists, we now introduce the second condition for an implementation to realize a specification. Suppose w' is a word of $L(B')$ relevant to A. Intuitively, w' is an input/output word that may occur in the implementation when an allowable sequence of input changes is applied to B'. If w' is not in $L(A)$, the implementation is capable of producing a sequence of transitions that does not exist in the specification. We say that such an implementation is "unsafe" for A.

Definition 11.1 *An implementation B' is* safe *for a specification A if* $L(B'/A) \subseteq L(A)$.

For our third condition, we need to ensure that every word in the language of A is in $L(B')$. Otherwise, the specification would have input/output words that the implementation is not capable of producing.

Definition 11.2 *An implementation B' has the* capability *of a specification A if* $L(A) \subseteq L(B')$.

Note that the condition $L(A) \subseteq L(B')$ is equivalent to the condition $L(A) \subseteq L(B'/A)$. Thus the basic requirement for an implementation B' to realize a specification A is the following language equality:

$$L(B'/A) = L(A).$$

As we shall see, this condition—which is based solely on language properties—is not sufficient. We add two more conditions defined on the behaviors, rather than just on their languages.

11.6.2 Deadlock

An implementation may fail to be a realization of a specification because a "deadlock" situation may arise, as we now illustrate.

Example 3

Consider the network of Figure 11.7 and its implementation behavior B' derived in the general single-winner model (for simplicity), shown in Figure 11.8. First the input changes, and then there is a critical race. If s_1 wins the race, an output is produced. However, if s_2 is faster, no output is produced. Here, the language of B' is $\{\varepsilon, \{X\}, \{X\}\{O\}\}$. Suppose A is defined by

$$q_1 \xrightarrow{\{X\}} q_2 \xrightarrow{\{O\}} q_3.$$

Then B' is safe for A and has the same capability. However, should the network take the path in which s_2 wins the race, the required output would never be produced.

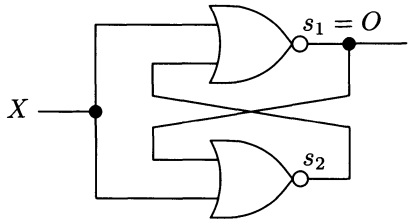


FIGURE 11.7. Network for deadlock example.

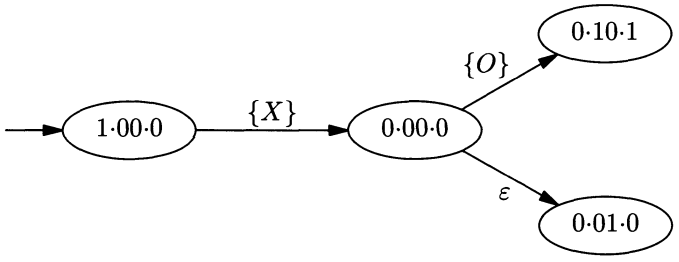


FIGURE 11.8. Behavior with deadlock.

Example 4

Consider now the network of Figure 11.9 and its behavior in Figure 11.10, derived in the general single-winner model. Is this a realization of the behavior in Figure 11.11? In state 0·010·00 there is no possibility of output O_1. Should this be considered as deadlock? We can state the condition given by the specification of Figure 11.11 as follows: After the environment applies the input X, the circuit should produce either O_1 or O_2, and both responses should be possible. It is clear that the implementation does what is intended. It just happens that, in case O_2 is produced, an internal-state change takes place first.

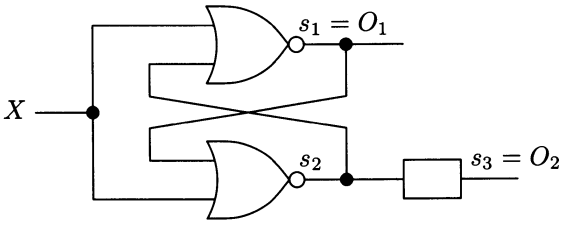


FIGURE 11.9. A network with two outputs.

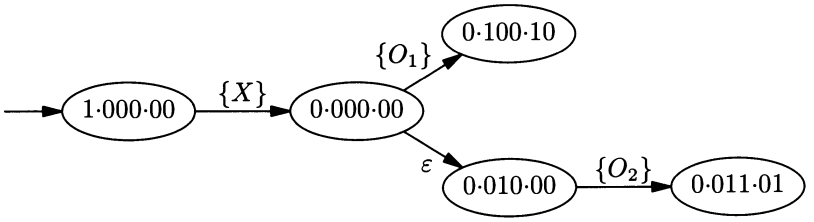


FIGURE 11.10. Behavior of network with two outputs.

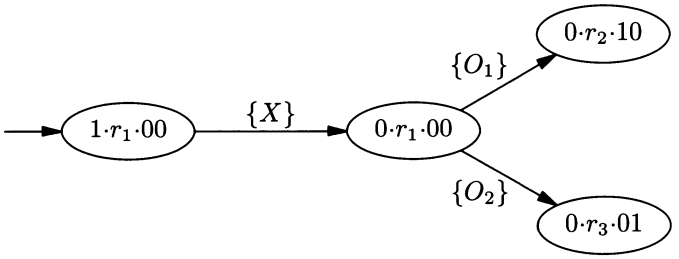


FIGURE 11.11. A specification with two outputs.

To discover the appropriate definition of deadlock, consider a specification $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$. We would like to describe the output words that need to be produced by A from each state. For this purpose we consider the input projection $A \Downarrow_{X=()}$ of A to the empty vector of input variables. The alphabet of this projection is $\Omega = \mathcal{P}(O) - \{\emptyset\}$. Let $L_{\mathcal{O}}(q)$ be the language accepted by state q in $A \Downarrow_{X=()}$, i.e., let $L_{\mathcal{O}}(q)$ be the language of the behavior $A \Downarrow_{X=()}$ with initial state changed to q.

Since A is deterministic, each word $w \in L(A)$ takes A to a unique state q_w. The language $L_{\mathcal{O}}(q_w)$ can be thought of as the response of A to w.

Suppose now that B' is an implementation with the same input and output alphabets as A. Consider a fixed word w of $L(A)$; then $wL_{\mathcal{O}}(q_w)$ describes the set of all words in $L(A)$ that begin with w and involve no input transitions other than those present in w itself. Note that w may take B' to several different states from q'_1. As we did for A, we also define the input projection $B' \Downarrow_{X=()}$ of B', and the languages $L'_{\mathcal{O}}(q')$ for all the states of B'.

A state q' of an implementation behavior B' is said to be *terminal* if $L'_{\mathcal{O}}(q') = \{\varepsilon\}$. A similar definition applies to specification behaviors.

Definition 11.3 *An implementation behavior B' has* deadlock *with respect to a specification behavior A if there exists a word $w \in L(B') \cap L(A)$ that leads to a terminal state in B', but to a nonterminal state in A. Otherwise, B' is* deadlock-free *with respect to A.*

11.6.3 Livelock

A problem may also arise if there is "livelock," as illustrated below.

Example 5

Consider the behavior A_1 shown in Figure 11.12, the network N_1 of Figure 11.13 in the gate-state model, and the behavior of N_1 shown

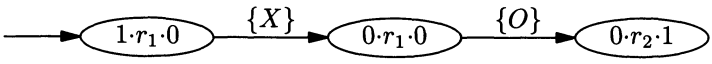


FIGURE 11.12. Behavior A_1.

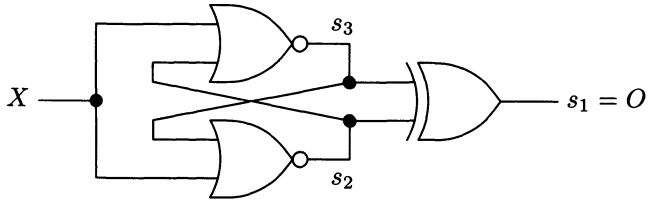


FIGURE 11.13. Network N_1.

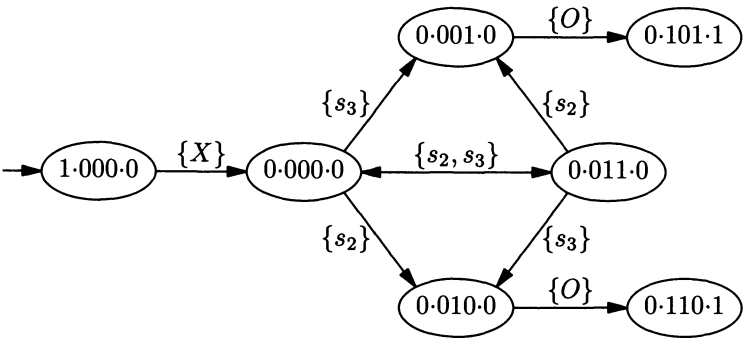


FIGURE 11.14. Behavior of N_1.

in Figure 11.14. We would like to determine whether N_1 can realize A_1, when the state variable s_1 is used as the output O. The initial state of N_1 that is to represent the initial state of A_1 is the stable state (1·000·0). The first transition $(1{\cdot}r_1{\cdot}0, 0{\cdot}r_1{\cdot}0)$ corresponds to (1·000·0, 0·000·0). The labels of N_1 and A_1 agree in the new states, so this transition is properly implemented. Next, there should occur the transition $(0{\cdot}r_1{\cdot}0, 0{\cdot}r_2{\cdot}1)$ in A_1. In Figure 11.14, the network has a nontransient (match-dependent) oscillation and a critical race. If s_3 wins the race, the network moves to (0·001·0); this change is externally invisible. The network will then move to (0·101·1), and an output change will be observed, as required. Similarly, if s_2 wins the race from state 0·000·0, the output will change in the next step, as required. There is, however, a third possibility, namely, the match-dependent oscillation. Should the network enter that oscillation and remain in it indefinitely, the output change would not occur. This is an example of livelock.

We define livelock as follows:

Definition 11.4 *An implementation B' has* livelock *with respect to a specification A if there is a word $w \in L(B') \cap L(A)$, leading to a nonterminal state in A, and to a state in B' that has a cycle spelling ε around it. Otherwise, B' is* livelock-free *with respect to A.*

Note that the definition applies well to Example 5 above. The following example, however, illustrates a difficulty with the general multiple-winner model with respect to livelock.

Example 6

The GMW analysis of the network of Figure 11.15 when it is started in state 1·00·0 and the input changes to 0 is shown in Figure 11.16. There is a nontransient oscillation starting in the initial state; this oscillation is invisible. If the input changes, the network moves to either

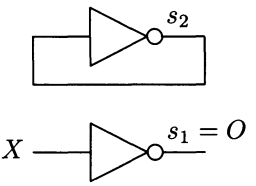


FIGURE 11.15. Network N_2.

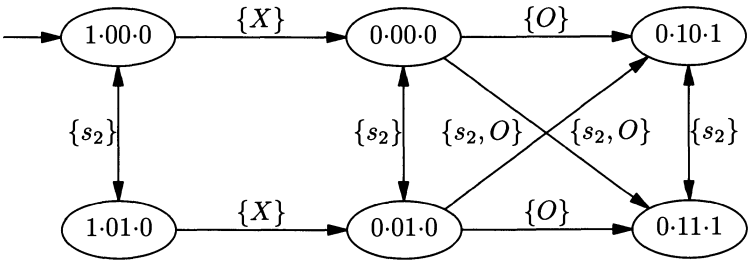


FIGURE 11.16. Behavior of N_2.

0·00·0 or 0·01·0, and the oscillation involving s_2 continues. However, the output is also unstable and will change eventually; thus the cycle (0·00·0, 0·01·0) is transient. If the delay of variable s_1 were infinite, then livelock would occur and the output change would not be produced. In reality, however, every delay has an upper bound, and the network must leave the transient cycle. When that happens, an output change occurs as required. In fact, the output may change in four different ways, as shown in the figure. The visible behavior is then just the word $\{X\}\{O\}$. Consequently, N_2 realizes A_1. Of course, it is clear from Figure 11.15 that this should be so.

The inconsistency above is caused by the fact that the GMW model does not assume any *specific upper bound* on the delays in the network, but only that such a bound *exists*. In a more accurate model, one would have to take the elapsed time into account in each state. This would lead to a more complicated model, which we do not pursue.

11.6.4 Definition of Realization

We are finally ready to define the concept of realization.

Definition 11.5 *An implementation behavior B'* realizes *a specification behavior A if*

- *There exists a projection of B' to A such that each variable of A is represented by a distinct variable of B', and the projection of q'_1 is q_1.*

Assuming now that the unused inputs and outputs have been removed, we require four conditions for realization.

- *B' is safe for A,*
- *B' has the capability of A,*
- *B' is deadlock-free with respect to A, and*
- *B' is livelock-free with respect to A.*

Example 2 (continued)

The behavior B' of Table 11.4 is shown in Figure 11.17. Consider the specification A with $L(A)$ equal

$$\{\varepsilon, \{O_2\}, \{O_2\}\{O_2\}, \{O_2\}\{O_2\}\{X_1\}, \{O_2\}\{O_2\}\{X_1\}\{O_2\}\}.$$

The proposed implementation B' is not safe for A, because, for example, $\{O_2\}\{O_2\}\{O_2\}$ is relevant to A, but it is not in $L(A)$. On the other hand, B' has the capability of A, because every word of $L(A)$ is also in $L(B')$. The implementation B' has deadlock; for example, the word ε leads to state q_2, which is terminal. The same word, however, leads to a nonterminal state in A. The same applies also to the word O_2. There is no livelock present.

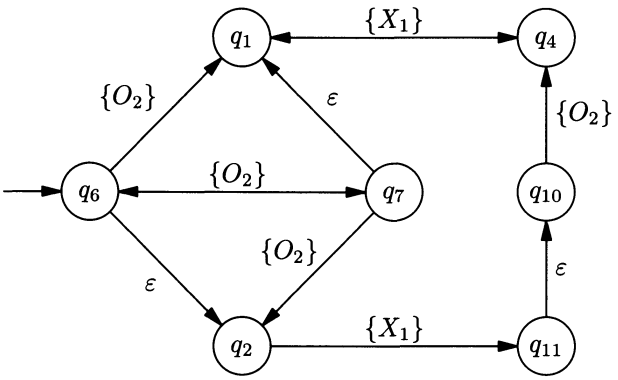


FIGURE 11.17. Behavior of Table 11.4.

11.7 Behavior Schemas

To motivate this section, let us return to our earlier example from combinational logic design. Frequently, a specification of a combinational circuit is not a Boolean function but a partial Boolean function. For example,

we may have a specification for a one-input, one-output circuit, where the output should be 0 when the input is 1, but the output is a "don't care" when the input is 0. Here, the specification is no longer a function, but could be viewed as a *set* of Boolean functions. In our example we would have two Boolean functions: $f_A = \overline{X}$ (if the value 1 is chosen for the don't care entry), and $g_A = 0$ (if the value 0 is chosen). We view a partial function specification as a "schema" describing a set of acceptable Boolean functions, called "options." Consider the inverter circuit again as a possible implementation. This implementation realizes the specification because the inverter function f_B realizes one of the options f_A.

Return now to asynchronous circuits. We want to permit some choice in the specification. In general, any set of behaviors could be used as a specification, and the designer would choose to implement one of them. It is convenient, however, to use a compact representation for such a set of behaviors. As in the combinational circuit example, a specification is a schema describing a set of behaviors called options. An implementation realizes such a schema provided that it realizes one of its options.

Before we proceed with the formal definition, we contrast the concept of choice with the concept of nondeterminism. If a state of a behavior representing a network has a single outgoing transition, and that transition has a tag of type O, then we expect the network to produce output O after entering that state. Here the output is chosen deterministically. On the other hand, we also need the ability to specify arbitration. Consider the case of specifying an arbiter with inputs X_1 and X_2 and outputs O_1 and O_2. A request is represented by an input change, and a grant by an output change. If the two requests arrive simultaneously, one of them should be granted. The request-granting aspect of the specification is illustrated in Figure 11.18(a). We do not accept as a valid implementation of such an arbiter a behavior in which only output O_1 is produced in the state corresponding to q. An acceptable implementation would be provided, for example, by a network (single-winner model) in which there is a two-way critical race between two gates, resulting in O_1 if one gate wins and in O_2 if the other gate wins.

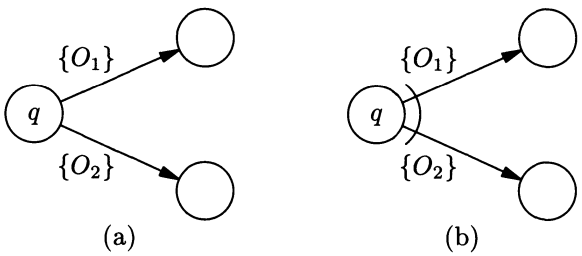


FIGURE 11.18. Arbitration and choice.

In contrast to the arbitration example above, a specification with "choice" does not require that both outputs be possible. Choice is indicated by an arc across outgoing transitions, as illustrated Figure 11.18(b). Here the designer, rather than the network, has the freedom to choose only output O_1, only output O_2, or both. In the last case, where both outputs are retained, the designer is in fact leaving the choice to the network, which, in turn, realizes the arbitration by a critical race each time the situation arises.

We now present a rather general definition of choice in specifications. A *behavior schema* is an 8-tuple

$$A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi, \mathcal{C} \rangle,$$

where

- $\langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$ is a behavior, and
- $\mathcal{C} = \{\mathcal{C}_q \mid q \in \mathcal{Q}\}$, is a set of *choice sets* $\mathcal{C}_q$, where $\mathcal{C}_q = \{\mathcal{B}_1, \ldots, \mathcal{B}_{j_q}\}$ is a set of nonempty subsets, called *blocks*, of the set

  $$\mathcal{T}_q = \{t \in \mathcal{T} \mid t = (q, q') \text{ for some } q' \in \mathcal{Q}\}$$

  of transitions leaving state q; furthermore, the union of all the blocks $\mathcal{B}_i$ of $\mathcal{Q}$ is $\mathcal{T}_q$.

Note that the blocks of a choice set need not be disjoint. A behavior schema is said to be *choice-free* if, for every $q \in \mathcal{Q}$, each block $\mathcal{B}_i$ of $\mathcal{C}_q$ has exactly one transition; in effect, such a schema degenerates to a behavior.

Example 7

The idea behind the choice set is illustrated by the example of Figure 11.19, which is a specification of a "fork" with input X and outputs O_1 and O_2. In static states, the two outputs have the same value as the input. In Figure 11.19, blocks of the choice set are denoted by arcs across the corresponding transitions. Blocks consisting of single transitions are not marked in any way. Starting with static state $0{\cdot}r_1{\cdot}00$, we can change the input and reach the dynamic state $1{\cdot}r_1{\cdot}00$. Here, the output variables are both required to change, but they may do so in any order whatsoever. The second half of the specification schema is similar.

A possible implementation for this specification would be one that always gives preference to O_1. In that case, the implementation would have the transitions

$$0{\cdot}r_1{\cdot}00 \longrightarrow 1{\cdot}r_1{\cdot}00 \longrightarrow 1{\cdot}r_3{\cdot}10 \longrightarrow 1{\cdot}r_4{\cdot}11 \longrightarrow$$

$$0{\cdot}r_4{\cdot}11 \longrightarrow 0{\cdot}r_2{\cdot}01 \longrightarrow 0{\cdot}r_1{\cdot}00.$$

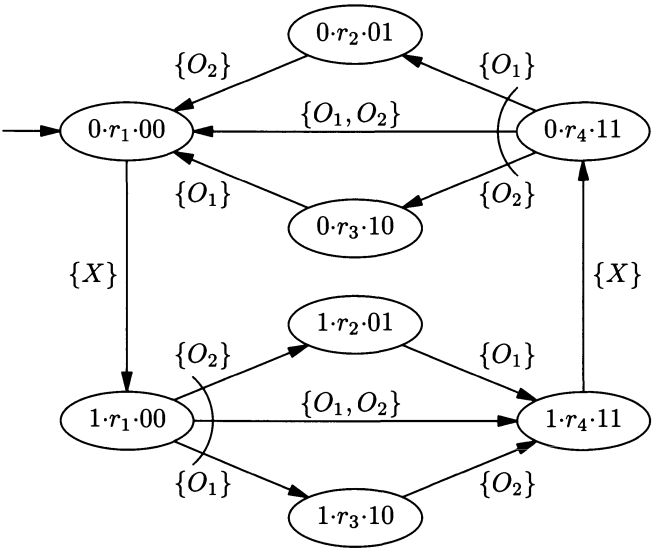


FIGURE 11.19. A specification for a fork.

An equally acceptable solution would be provided by the following set of transitions, where O_1 rises first but falls second:

$$0{\cdot}r_1{\cdot}00 \longrightarrow 1{\cdot}r_1{\cdot}00 \longrightarrow 1{\cdot}r_3{\cdot}10 \longrightarrow 1{\cdot}r_4{\cdot}11 \longrightarrow$$

$$0{\cdot}r_4{\cdot}11 \longrightarrow 0{\cdot}r_3{\cdot}10 \longrightarrow 0{\cdot}r_1{\cdot}00.$$

A third choice would be to implement the transitions by allowing a race between the two variables. In that case, the implementation behavior would be exactly like the specification schema, but without the choice-set arcs.

In the examples above, some transitions from each block are selected at design time, and these transitions are the ones implemented. The same set of transitions is used every time a particular state is visited. One could also envision an implementation that selects transitions dynamically, and may use different selections for different visits to the same state. Consider the following implementation behavior for the fork of Example 7:

$$0{\cdot}r_1{\cdot}00 \longrightarrow 1{\cdot}r_1{\cdot}00 \longrightarrow 1{\cdot}r_3{\cdot}10 \longrightarrow 1{\cdot}r_4{\cdot}11 \longrightarrow 0{\cdot}r_4{\cdot}11 \longrightarrow$$

$$0{\cdot}r_2{\cdot}01 \longrightarrow 0{\cdot}r_1'{\cdot}00 \longrightarrow 1{\cdot}r_1'{\cdot}00 \longrightarrow 1{\cdot}r_2'{\cdot}01 \longrightarrow 1{\cdot}r_4'{\cdot}11 \longrightarrow$$

$$0{\cdot}r_4'{\cdot}11 \longrightarrow 0{\cdot}r_3'{\cdot}10 \longrightarrow 0{\cdot}r_1{\cdot}00.$$

Here, variable O_1 changes first during the first "pass" through the specification, but it changes second during the second pass.

Given a behavior schema $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi, \mathcal{C} \rangle$ and a behavior $A' = \langle X, \mathcal{R}', O, \mathcal{Q}', q_1', \mathcal{T}', \psi' \rangle$, we say that A' is an *option* of A if there exists a function $\rho : \mathcal{Q}' \to \mathcal{Q}$ satisfying the following conditions:

- $\rho(q_1') = q_1$, and $l(q_1') = l(q_1)$. This ensures that both A and A' start in states with the same input/output label.
- For every state $q' \in \mathcal{Q}'$ and for every block $\mathcal{B}$ of $\mathcal{C}_{\rho(q')}$, there exists a transition $(q', p') \in \mathcal{T}'$ such that $(\rho(q'), \rho(p'))$ is a transition in $\mathcal{B}$, and $\tau(q', p') = \tau(\rho(q'), \rho(p'))$. This ensures that every block of the choice set of the state $\rho(q')$ is represented by at least one transition of A'.
- For every transition $(q', p') \in \mathcal{T}'$, we have $(\rho(q'), \rho(p'))$ in $\mathcal{T}$, and $\tau(q', p') = \tau(\rho(q'), \rho(p'))$.

Example 7 (continued)

Consider Figure 11.19 again. An option for the fork specification is shown in Figure 11.20. The first time we change the two variables sequentially in either order, but we do not allow a simultaneous change. The second and third times we select the simultaneous change. The fourth time we select only O_1 followed by O_2.

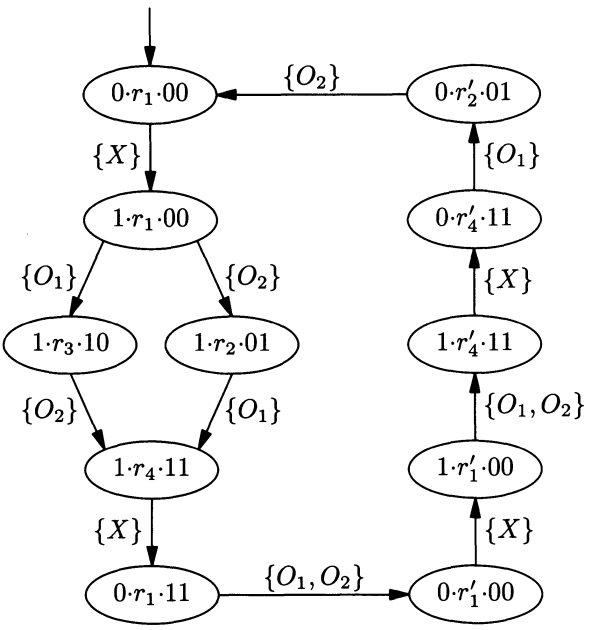


FIGURE 11.20. An implementation of a fork.

With each schema $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi, \mathcal{C} \rangle$, we associate a behavior $B = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$. The language of a schema A is defined as the language of its associated behavior B. One can verify that options satisfy the following properties:

Proposition 11.2 *Let A be a behavior schema and let B be one of its options. Then*

- $L(A) = \bigcup_B L(B)$, *where the union is over all options B of A;*
- $L(B) \subseteq L(A)$;
- *If A is choice-free, then $L(B) = L(A)$.*

We can now reduce the problem of realizing specifications with choice to a problem of realizing choice-free specifications.

Definition 11.6 *A behavior B realizes a behavior schema A if there exists an option A' of A such that B realizes A'.*

11.8 Concluding Remarks

In this chapter we have defined some important concepts, which will be applied in later chapters. In particular, in Chapter 12 we discuss a number of behaviors that can be associated with a given network. We also compare our definitions of schemas and behaviors with the classical flow table techniques. In Chapter 13 we use our formal notions of behaviors and realizations to prove that certain specifications are not realizable. In Chapter 14, we deal with symbolic behaviors and the verification problem. Finally, in Chapter 15 we use some of the notation developed here.

Chapter 12

Types of Behaviors

In this chapter, we consider several types of behaviors that may be used for specifying and analyzing asynchronous circuits. Section 12.1 shows how a number of behaviors may be associated with a simple OR gate. In Section 12.2, we consider the classical primitive flow table specifications. We point out some deficiencies of the flow table approach, and we compare it with our formal model of behaviors. In Section 12.3 we discuss the derivation of behaviors of networks operated in fundamental mode. General fundamental-mode behaviors can be abbreviated to the "direct" behaviors described in Section 12.4, if transient states are of no interest. Moreover, many behaviors encountered in practice are even more restricted. For this reason, we introduce in Section 12.5 the class of "serial" behaviors. These behaviors will be used again in Chapters 13 and 14.

12.1 Introductory Examples

The most general environment one could provide for a network is the unrestricted environment. We will use the GMW model as the underlying race model for the analysis of a network N; thus, let R_a be the GMW relation of N. To derive the network behavior in an unrestricted environment we define a binary relation R_u on the set of total states of N as follows: Let $a, a' \in \{0,1\}^n$ and $b, b' \in \{0,1\}^m$ be such that $a{\cdot}b \neq a'{\cdot}b'$. Then

$$a{\cdot}bR_u a'{\cdot}b' \text{ if and only if } b = b' \text{ or } bR_a b'.$$

This means that the input excitation vector may change arbitrarily at any time, and the state component may change according to the R_a relation, where a is the present input vector.

Throughout this section, we use an OR gate as the running example. We will introduce various types of behaviors by using this example. The network model of the OR gate is $N = \langle \{0,1\}, \{X_1, X_2\}, \{s\}, \mathcal{E}, F \rangle$, where X_1 and X_2 are the two input excitation variables, s is the state variable with excitation function $S = X_1 + X_2$, and O is the output variable given by the circuit equation $O = s$.

Example 1

We construct the unrestricted behavior for the OR gate. State 00·1 has seven possible successor states. Since this state is unstable, the state variable may change; this would lead to state 00·0. If input X_1 changes, the state could become 10·1 or 10·0. Similarly, if X_2 changes,

TABLE 12.1. Unrestricted behavior of OR gate.

q	$X{\cdot}r$	$\{X_1\}$	$\{X_2\}$	$\{X_1, X_2\}$	$\{O\}$	$\{X_1\ O\}$	$\{X_2, O\}$	$\{X_1, X_2, O\}$
q_1	00·0	10·0	01·0	11·0	-	-	-	-
q_2	01·1	11·1	00·1	10·1	-	-	-	-
q_3	10·1	00·1	11·1	01·1	-	-	-	-
q_4	11·1	01·1	10·1	00·1	-	-	-	-
q_5	00·1	10·1	01·1	11·1	00·0	10·0	01·0	11·0
q_6	01·0	11·0	00·0	10·0	01·1	11·1	00·1	10·1
q_7	10·0	00·0	11·0	01·0	10·1	00·1	11·1	01·1
q_8	11·0	01·0	10·0	00·0	11·1	01·1	10·1	00·1

we might have state 01·1 or 01·0. Finally, both inputs might change, leading to state 11·1 or 11·0. The complete transition table for the OR gate is shown in Table 12.1. The table is divided into two parts, with the first four states being stable and the last four unstable. Since the output variable has the same value as the state variable, we don't use expanded states.

Example 2

The fundamental-mode behavior of the OR gate is shown in Table 12.2. The stable states have exactly the same transitions as they do in the unrestricted mode. Each unstable state, however, has only a single transition, because the input is not allowed to change. Since no XO transitions exist, we have deleted the last three columns.

Example 3

In some applications, only one input excitation is permitted to change at a time. Our third example shows the behavior of the OR gate under this restriction; the behavior is given in Table 12.3.

TABLE 12.2. Fundamental-mode behavior of OR gate.

q	$X{\cdot}r$	$\{X_1\}$	$\{X_2\}$	$\{X_1, X_2\}$	$\{O\}$
q_1	00·0	10·0	01·0	11·0	-
q_2	01·1	11·1	00·1	10·1	-
q_3	10·1	00·1	11·1	01·1	-
q_4	11·1	01·1	10·1	00·1	-
q_5	00·1	-	-	-	00·0
q_6	01·0	-	-	-	01·1
q_7	10·0	-	-	-	10·1
q_8	11·0	-	-	-	11·1

TABLE 12.3. Single-input-change behavior of OR gate.

q	$X{\cdot}r$	$\{X_1\}$	$\{X_2\}$	$\{O\}$	$\{X_1, O\}$	$\{X_2, O\}$
q_1	00·0	10·0	01·0	-	-	-
q_2	01·1	11·1	00·1	-	-	-
q_3	10·1	00·1	11·1	-	-	-
q_4	11·1	01·1	10·1	-	-	-
q_5	00·1	10·1	01·1	00·0	10·0	01·0
q_6	01·0	11·0	00·0	01·1	11·1	00·1
q_7	10·0	00·0	11·0	10·1	00·1	11·1
q_8	11·0	01·0	10·0	11·1	01·1	10·1

Example 4

Table 12.4 shows a single-input-change behavior with some input changes permitted in unstable states [44]. The approach is illustrated as follows. Suppose the OR gate starts in stable state 10·1. If the input X_1 now changes, it is the environment's intention to change the state of the OR gate. Suppose state 00·1 is now reached. Applying any input change here would negate the original intention because the new state would be stable. Hence, no input changes are permitted in state 00·1. In contrast to this, consider stable state 00·0 and a change in X_2. The new state 01·0 is reached with the intention of changing the output. Here, the input X_1 can change without affecting the original intention: the gate remains unstable. Consequently, the change in X_1 is permitted. Similarly, in state 11·0, the intention is to change the output. Changing either input preserves this intention; hence both input changes are permitted.

TABLE 12.4. Dill's single-input-change behavior of OR gate.

q	$X{\cdot}r$	$\{X_1\}$	$\{X_2\}$	$\{O\}$
q_1	00·0	10·0	01·0	-
q_2	01·1	11·1	00·1	-
q_3	10·1	00·1	11·1	-
q_4	11·1	01·1	10·1	-
q_5	00·1	-	-	00·0
q_6	01·0	11·0	-	01·1
q_7	10·0	-	11·0	10·1
q_8	11·0	01·0	10·0	11·1

TABLE 12.5. Single-input-change fundamental-mode behavior of OR gate.

q	$X{\cdot}r$	$\{X_1\}$	$\{X_2\}$	$\{O\}$
q_1	00·0	10·0	01·0	-
q_2	01·1	11·1	00·1	-
q_3	10·1	00·1	11·1	-
q_4	11·1	01·1	10·1	-
q_5	-	-	-	00·0
q_6	-	-	-	01·1
q_7	-	-	-	10·1
q_8	-	-	-	11·1

Example 5

Our last example of Table 12.5 shows the fundamental-mode single-input-change behavior of the OR gate. This is more restricted than Dill's single-input-change behavior.

12.2 Fundamental-Mode Specifications

We now consider the classical primitive flow tables that have been used for fundamental-mode specifications [66, 67, 135]. We will show that such flow tables have some shortcomings; consequently, we will use our behavior schema model instead.

Example 6

The flow table of Table 12.6 illustrates many of the concepts from classical theory [135]. The rows correspond to internal states $r_1, \ldots, r_4$. The columns list the four values of two binary inputs X_1 and X_2. Each nonblank entry, except the one in column 11, row r_1, contains the next internal state and the present output. The output value is assumed to be uniquely determined by the total state. The double entry in column 11, row r_1 denotes choice: either entry could be implemented and the specification would be satisfied. If the internal state of an entry

TABLE 12.6. A primitive flow table.

	00	01	11	10
r_1	(⓵), 0	r_2, 0	r_3, 0 | r_4, 0	r_2, 1
r_2	r_3, 1	(⓶), 0	-	r_3, 0
r_3	r_1, 0	-	(⓷), 1	r_4, 1
r_4	r_1, 1	-	(⓸), 0	r_1, 0

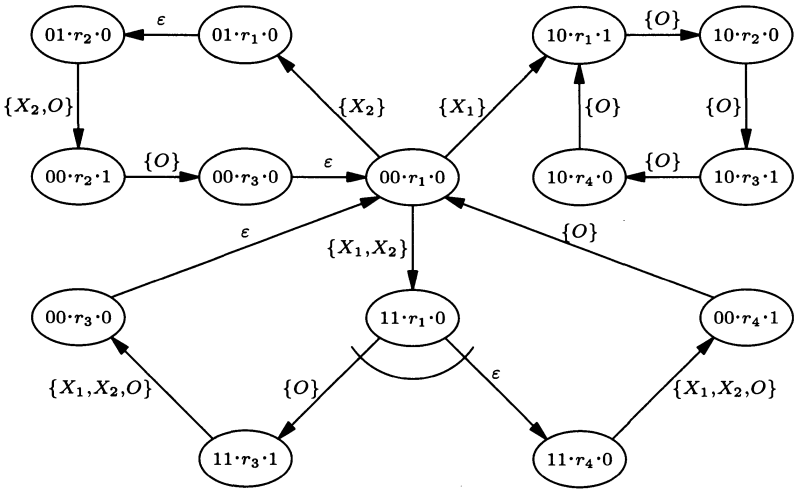


FIGURE 12.1. Schema of primitive flow table.

agrees with the row state, then the internal state will not change, i.e., it is stable. Stable states are indicated by circling the corresponding internal-state entries. A flow table is called *primitive* if there is at most one stable state per row.

Let us illustrate the operation of the flow table by some typical transitions, and also compare the flow table notation to our behavior schema model. For example, in column 00, row r_1, entry $r_1, 0$ indicates that total state $X_1X_2{\cdot}r = 00{\cdot}r_1$ is stable; the flow table remains in this state until an input change occurs. In our notation, column 00, row r_1 entry corresponds to expanded static state $q_1 = 00{\cdot}r_1{\cdot}0$; see Figure 12.1. Next, consider the entry in $(01, r_1)$; it indicates that the next internal state should be r_2 and the output should not change. Thus the "operating point" that we might associate with the flow table starts in $(00, r_1)$, and moves horizontally to $(01, r_1)$, when X_2 is changed by the environment. Since fundamental-mode operation does not permit any input changes in unstable states, the operating point can only move vertically to $(01, r_2)$ as the internal state changes. In our notation, we also have two transitions: the input transition from $00{\cdot}r_1{\cdot}0$ to $01{\cdot}r_1{\cdot}0$, where only the input X_2 has changed, and an internal-state transition from $01{\cdot}r_1{\cdot}0$ to $01{\cdot}r_2{\cdot}0$.

Consider now total state $X_1X_2{\cdot}r = 11{\cdot}r_4$, which is stable. If the input changes to 00, we have the entry $r_1, 1$, showing that both the internal state and the output must change. We can model this motion of the operating point from $(11, r_4)$ through $(00, r_4)$ to $(00, r_1)$ by an input transition from $11{\cdot}r_4{\cdot}0$ to $00{\cdot}r_4{\cdot}1$, followed by an internal-state transition to $00{\cdot}r_1{\cdot}0$.

Location $(11, r_1)$ has two entries. This represents choice: Either the operating point can move to state r_4—in which case the output does not need to change—or it can move to r_3—in which case an output change is eventually required. We model this by transition $(00{\cdot}r_1{\cdot}0, 11{\cdot}r_1{\cdot}0)$, followed by transitions $(11{\cdot}r_1{\cdot}0, 11{\cdot}r_3{\cdot}1)$ and $(11{\cdot}r_1{\cdot}0, 11{\cdot}r_4{\cdot}0)$ joined by a choice arc,[1] as shown in Figure 12.1. Note that there are ε transitions in this schema.

The transition from $(00, r_1)$ to $(10, r_1)$ leads to an oscillation through states $r_1, r_2, r_3, r_4, r_1, r_2, \ldots$. Now the question arises whether the entry in $(10, r_3)$ is there only because it is necessary for the oscillation, or is it possible to reach this state by changing input X_2, while the flow table is in stable state r_3 with input 11? (In Figure 12.1 we assume that there is no transition from $(11, r_3)$ to $(10, r_3)$.) There is no way of answering such questions in the flow table representation, whereas these ambiguities do not arise in our behavior schema representation, since each permitted transition has an explicit representation; hence we prefer our definition for fundamental-mode specifications.

A *fundamental-mode behavior schema* is defined as a schema $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi, C \rangle$, where q_1 is static, and each dynamic state has only internal-state transitions. This satisfies the fundamental-mode requirement that inputs can change only if the present state is stable.

12.3 Fundamental-Mode Network Behaviors

In this section we describe a method for finding behaviors of networks operated in fundamental mode.

Let $N = \langle \{0,1\}, \mathcal{X}, \mathcal{S}, \mathcal{O}, \mathcal{E}, F \rangle$ be a network with n inputs and m state variables. Note that we have added the set $\mathcal{O}$ of output variables of N and the corresponding output vector $O = O_1, \ldots, O_p$. Each output is a Boolean function of the input excitations and state variables, as specified by the vector F of circuit equations. Let R_a be a race relation for N that specifies a set of possible next states for a given unstable state. The exact nature of this relation is not important here, as long its outcome is well defined. In all of our examples we use the GMW relation for convenience. We remind the reader of the difficulty with livelock representation: transient cycles must be removed from consideration.

---

[1] We point out that, if we change the flow table slightly by replacing the $r_3, 0$ entry in state $(11, r_1)$ with $r_3, 1$, this situation can be modeled by transitions $(00{\cdot}r_1{\cdot}0, 11{\cdot}r_1{\cdot}1)$ and $(00{\cdot}r_1{\cdot}0, 11{\cdot}r_1{\cdot}0)$ joined by a choice arc, and the additional transitions $(11{\cdot}r_1{\cdot}1, 11{\cdot}r_3{\cdot}1)$ and $(11{\cdot}r_1{\cdot}0, 11{\cdot}r_4{\cdot}0)$.

The *fundamental-mode relation* R_{fm} of a network N on the set $\{0,1\}^{n+m}$ of all the total states of N is defined as follows:

$$(a{\cdot}b)R_{fm}(a'{\cdot}b')$$

if and only if

$$(a{\cdot}b) \neq (a'{\cdot}b'),\ bR_ab',\ \text{and either } a = a' \text{ or } b = b'.$$

Thus two states are related by R_{fm} if either the first is stable and the second differs from it only in the input vector, or the first is unstable and the second differs from it only in the state vector.

A *fundamental-mode behavior* B_{fm} of a network N with fundamental-mode relation R_{fm} is a behavior defined as follows:

$$B_{fm} = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle, \text{ where}$$

- $X = (X_1, \ldots, X_n)$;
- $\mathcal{R} = \{0,1\}^m$, where the internal state has the form $(b_1, \ldots, b_m)$, and the binary value b_i is associated with the network state variable s_i, for $i = 1, \ldots, m$;
- $O = (O_1, \ldots, O_p)$;
- $\mathcal{Q} = \{0,1\}^n \times \{0,1\}^m$, where the total state has the form $((a_1, \ldots, a_n), (b_1, \ldots, b_m))$, with a_i associated with X_i for $i = 1, \ldots, n$, and b_j associated with s_j, for $j = 1, \ldots, m$;
- $q_1 \in \mathcal{Q}$ is a stable initial state of N;
- $\mathcal{T} = R_{fm}$;
- $\psi : \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^p$ is the circuit output function.

Note that, when the fundamental-mode behavior of a network is used to realize a specification behavior, one of its stable states is chosen as the initial state. Thus we associate with a network as many behaviors as there are different stable states.

Example 7

To illustrate fundamental-mode behaviors, consider the NOR latch in the gate-state model. From the excitation functions $S_1 = \overline{X_1 + s_2}$ and $S_2 = \overline{X_2 + s_1}$, we obtain the R_a relations shown in Figure 12.2. Using these relations (and some routine work) we construct the behavior for the latch as shown in Table 12.7. We assume that both gate outputs are also external outputs. The table is divided into three parts. First we list the five stable states. According to the fundamental-mode operation [66, 67, 93, 135], we are allowed to change any set of inputs

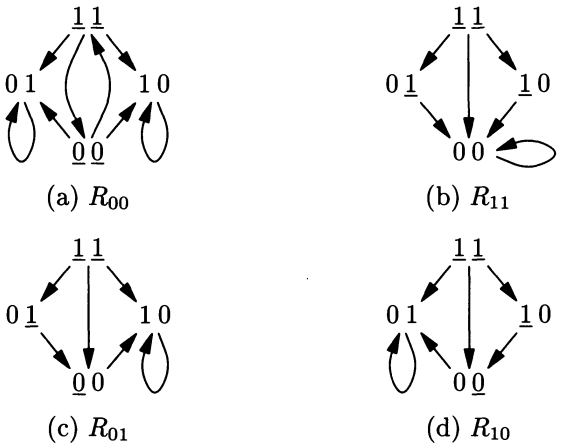


FIGURE 12.2. R_a relations for the NOR latch.

in any stable state. Dashed entries indicate that there are no transitions with the corresponding tags. Next, we list the unstable states reachable from the stable states. The successors of unstable states are determined by the R_{fm} relation—here based on the GMW relation. Finally, we list the unstable states that are not reachable from the stable states. Since we assume that the network always starts in a stable state, these three states can be omitted.

TABLE 12.7. Transition table for NOR latch.

q	$X{\cdot}r{\cdot}O$	$\{X_1\}$	$\{X_2\}$	$\{X_1, X_2\}$	$\{O_1\}$	$\{O_2\}$	$\{O_1, O_2\}$
q_1	00·01·01	q_4	q_9	q_{12}	-	-	-
q_2	00·10·10	q_{11}	q_3	q_{13}	-	-	-
q_3	01·10·10	q_{13}	q_2	q_{11}	-	-	-
q_4	10·01·01	q_1	q_{12}	q_9	-	-	-
q_5	11·00·00	q_8	q_{10}	q_6	-	-	-
q_6	00·00·00	-	-	-	q_2	q_1	q_7
q_7	00·11·11	-	-	-	q_1	q_2	q_6
q_8	01·00·00	-	-	-	q_3	-	-
q_9	01·01·01	-	-	-	-	q_8	-
q_{10}	10·00·00	-	-	-	-	q_4	-
q_{11}	10·10·10	-	-	-	q_{10}	-	-
q_{12}	11·01·01	-	-	-	-	q_5	-
q_{13}	11·10·10	-	-	-	q_5	-	-
q_{14}	01·11·11	-	-	-	q_9	q_3	q_8
q_{15}	10·11·11	-	-	-	q_4	q_{11}	q_{10}
q_{16}	11·11·11	-	-	-	q_{12}	q_{13}	q_5

12.4 Direct Behaviors

In a many situations we are interested in specifying only the outcome of a transition but *not* in the many ways by which a state in the outcome can be reached. In such cases it is sufficient to consider a restricted class of fundamental-mode behaviors in which emphasis is placed on the outcome of a transition, and details concerning transient states are suppressed. The resulting concise model is useful for describing many practical circuits.

A behavior is called *direct* if it is a fundamental-mode behavior and, for every dynamic state q, in every transition (q, q'), the state q' is static. Note that, in a fundamental-mode behavior, all transitions from a dynamic state must be internal-state transitions. Thus, in a direct behavior we specify the "final destination" the behavior should reach. Of course, a circuit for which a direct behavior is computed may go through a number of intermediate states before reaching a circuit state corresponding to the final destination. This, however, is not taken into account in a direct-behavior description of the circuit.

Example 8

In the behavior of Figure 12.3, two output changes (constituting an output pulse) are produced for every input change. This behavior is fundamental-mode, but it is not direct.

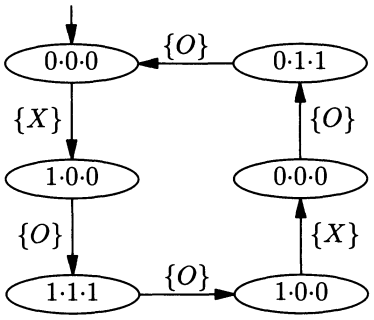


FIGURE 12.3. A behavior that is not direct.

Proposition 12.1 *If B is a direct implementation behavior, then it is livelock-free for every specification behavior A.*

Proof: There are no outgoing ε transitions from a static state. Hence, every path spelling ε must begin in a dynamic state. Since each dynamic state leads directly to a static state, a direct behavior can have only ε paths of length one. Since no cycle in a behavior can have length one, there can be no cycles spelling ε. Thus a direct behavior is livelock-free with respect to every specification. □

We now describe a construction that permits us to find the direct behavior of a network N. We assume that its fundamental-mode behavior $B_{fm} = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$ has already been constructed.

An unstable state $q = a{\cdot}r$ of N is called *fresh* if there exists an input vector a' such that $a'{\cdot}r$ is a stable state. Thus a state is fresh if it is unstable and can be reached from a stable state in one step of the relation R_{fm}.

We assume here that we are interested in designing a class of circuits in which an input change applied to a stable state results in a "reliable" transition to another stable state. Thus oscillations are not permitted. Furthermore, we want to ensure that no hazards are present in any of the circuit outputs; otherwise, other circuits using the outputs of the circuit being designed might reach incorrect states. A state $q = a{\cdot}r$ is *stabilizing* if $out(R_a(r))$ contains only stable states. A state $q = a{\cdot}r$ is O_i*-hazard-free* if, in any R_a-sequence of states leading from r to a state r' in $out(R_a(r))$, the output variable O_i changes at most once. A state q is *output-hazard-free* if it is O_i-hazard-free for every $O_i \in \mathcal{O}$.

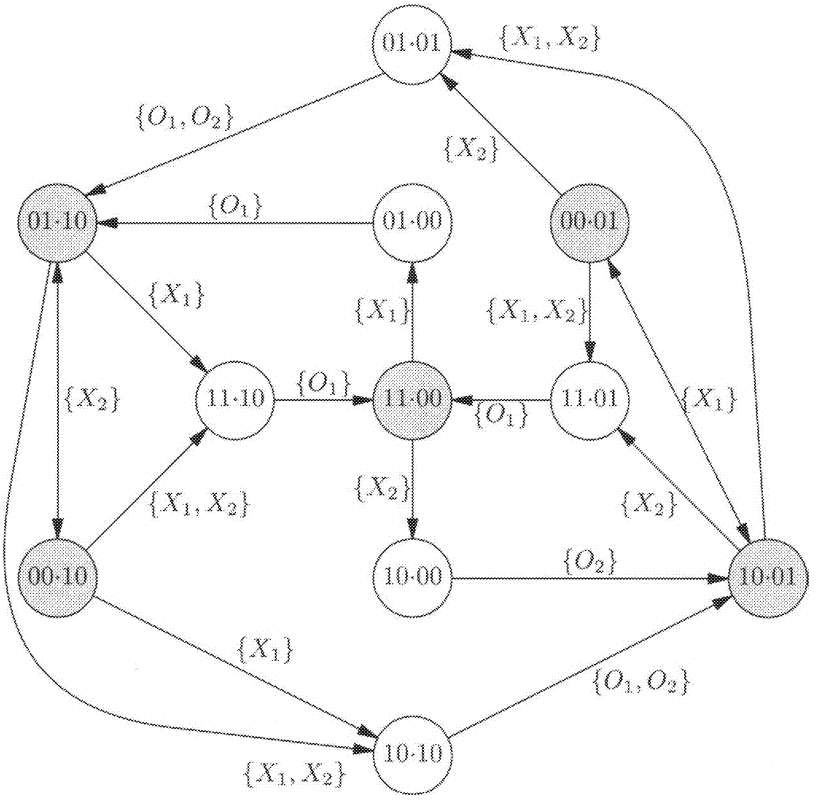


FIGURE 12.4. Direct behavior for the NOR latch.

The direct behavior of network N with fundamental mode behavior B_{fm} is $B_{dir} = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}_{dir}, \psi \rangle$, where $\mathcal{T}_{dir} \subseteq (\mathcal{Q} \times \mathcal{Q})$ is defined as follows: If $q = a \cdot r$ is stable, $qR_{fm}q'$, and q' is both stabilizing and output-hazard-free, then $(q, q') \in \mathcal{T}_{dir}$ and $(q', q'') \in \mathcal{T}_{dir}$ for all $q'' \in out(R_{fm}(q'))$. No other pairs of states are related.

Example 9

We illustrate the concept of direct behavior for the NOR latch with the fundamental-mode behavior of Table 12.7. The direct behavior is shown in Figure 12.4. Since the outputs are the same as the state variables, they are not shown. Static states are shown shaded. Note that a double output change occurs in the transition from state 10·10 to state 10·01. This does not imply that the two outputs change simultaneously in the network; in fact, from the relation R_{10} it is clear that O_2 must change before O_1. No transition is included for state 11·00 under $\{X_1, X_2\}$ because the resulting state is not stabilizing.

12.5 Serial Behaviors

The types of behaviors that we have defined above are rather general; in many practical cases the model can be even simpler than the direct behaviors. In this section we study a class of restricted direct behaviors, called "serial behaviors." These behaviors have some desirable properties with respect to deadlock, and have simple representations.

A behavior is *serial* if it is direct and satisfies the following three additional conditions:

1. It has no XO transitions;
2. For each $\mathcal{X}' \subseteq \mathcal{X}$ and for each state q, there is at most one transition leaving q with tag $\mathcal{X}'$;
3. Each dynamic state has exactly one outgoing transition.

Consider the possible transitions in a serial behavior. The initial state is static and can only have input or mixed transitions of type X. Consider a transition (q, q') from any static state. If q' is static, it also can only have X transitions. On the other hand, if q' is dynamic, we must have exactly one (internal-state) transition, say (q', q''), because of Condition 3. This transition (q', q'') may be an ε transition or an O transition. In both cases q'' is static, because the behavior is direct. In summary, we can describe the behavior as follows. It starts in a static state. In every static state, the environment may supply an input change. There is exactly one input transition corresponding to this input change. The circuit then responds by either "doing nothing" (if the state reached by the transition is static)

or by moving to a new static state with or without an output change (if the state reached is dynamic). Note that "doing nothing" permits the circuit to change its internal state.

Example 10

The behavior in Figure 12.5 has one input, two state variables and two outputs. In the initial state 0·00·00, the input may change. The behavior then randomly selects one of the two outputs and changes it. A second input change may then occur, causing the second output to change. No further actions are permitted in state 0·11·11. This behavior is direct, but it is not serial.

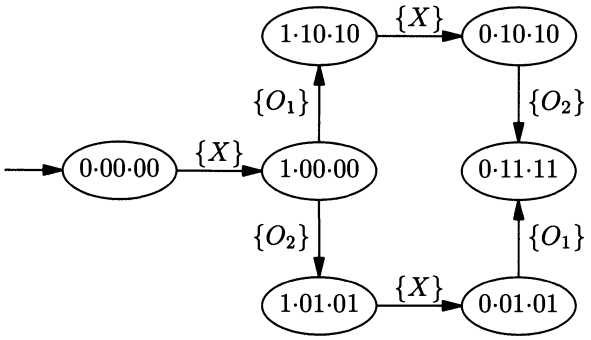


FIGURE 12.5. A behavior that is direct but not serial.

Example 11

The direct behavior of the latch of Figure 12.4 is serial provided the input $\{X_1, X_2\}$ is not used in state 11·00.

Proposition 12.2 *Let B be a serial implementation behavior, and A a specification behavior. If B has the capability of A, then it is deadlock-free for A.*

Proof: We claim that at most two states can be reached in B by any word w. Furthermore, we claim that, if two distinct states of B are reached by the same word w, then one of them, say q, is dynamic, and the other, called q', is static; furthermore, there is a transition (q, q') with tag ε. We prove our claim by induction on the length $|w|$ of w.

If $|w| = 0$, the only state reached by w is the initial state q_1. Hence our claim holds.

Suppose that only one state q can be reached by w, and consider $w\sigma$. If q is static, then σ must be a subset of $\mathcal{X}$, because B has no XO transitions. Only one state can be reached by $w\sigma$, because of Condition 2. If q is dynamic, then σ must be a subset of $\mathcal{O}$, because of Condition 3. In case

σ is nonempty, again, only one state can be reached by $w\sigma$. In case σ is ε, two states can be reached by $w\sigma$: q, and the static state q' that is the destination of the single transition from q. Hence the induction step goes through, if only one state can be reached by w.

Suppose now that both q and q' can be reached by w and satisfy the inductive assumption, with q being dynamic. Consider $w\sigma$. Since the single transition from the dynamic state q is already used to get to static state q', the only state reachable by $w\sigma$ is the one that can be reached from q', and σ must necessarily be a subset of $\mathcal{X}$. Therefore the induction step goes through in this case also.

Suppose now that deadlock occurs for the word w. If only one state is reached in B by w, then that state must be terminal, and the state reached in A by w must be nonterminal. Thus there is some word $w\sigma$, $\sigma \subseteq \mathcal{O}$, in $L(A)$, but there is no such word in $L(B)$. Therefore B does not have the capability of A, and we have a contradiction.

In case two states q and q' can be reached by w, it follows from the claims that $L_{\mathcal{O}}(q) = L_{\mathcal{O}}(q') = \{\varepsilon\}$. Thus both states are terminal, and the same argument applies as above. □

Suppose B is a serial behavior realizing some specification A. It is possible to remove the ε transitions without affecting $L(B)$ and without introducing deadlock or livelock by the following process. If q is static, (q, q') is an input transition, and (q', q'') is an ε transition, remove both of these transitions from B and add the transition (q, q''). Clearly, such a construction preserves $L(B)$. Since the resulting behavior is still serial, it is also livelock-free and deadlock-free. Therefore, the modified B with ε transitions removed still realizes A. Note that ε-free serial behaviors are deterministic.

Let $B = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$ be a serial ε-free behavior. A word of $L(B)$ is said to be *complete* if it leads from the initial state to a static state. In serial behaviors we can simplify the notation by eliminating dynamic states entirely. The *complete-word* behavior of a serial behavior B is defined as follows:

$$\hat{B} = \langle X, \mathcal{R}, O, \mathcal{Q}_{static}, q_1, \hat{\mathcal{T}}, \hat{\psi} \rangle,$$

where

- if $(q, q') \in \mathcal{T}$ and q' is static, then $(q, q') \in \hat{\mathcal{T}}$;
- if $(q, q') \in \mathcal{T}$ and q' is dynamic, and if $(q', q'') \in \mathcal{T}$, then $(q, q'') \in \hat{\mathcal{T}}$.
- There are no other transitions in $\hat{\mathcal{T}}$.
- $\hat{\psi}$ is the restriction of ψ to $\mathcal{Q}_{static}$.

It is clear that the original ε-free serial behavior can be uniquely reconstructed from a complete-word behavior, by reversing the construction, i.e., by introducing a dynamic state whenever there is an XO transition.

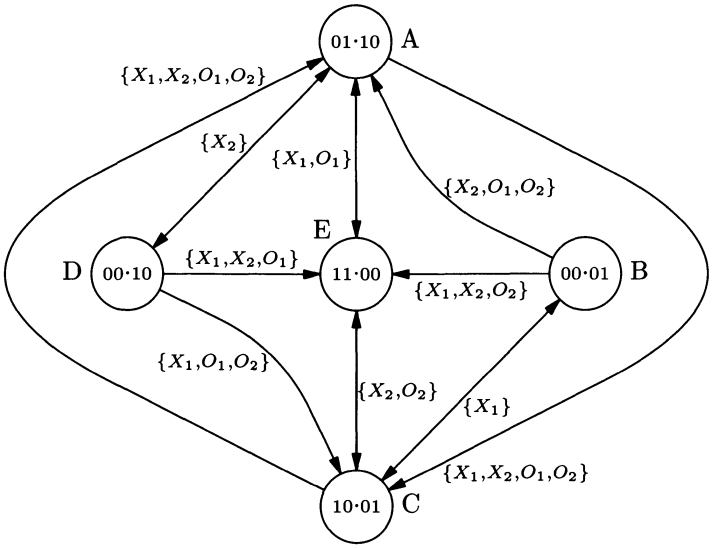


FIGURE 12.6. Complete-word behavior of NOR latch.

Example 12

The complete-word behavior corresponding to the direct behavior of Figure 12.4 for the latch is shown in Figure 12.6. It is assumed that the input $\{X_1, X_2\}$ is not used in state 11·00.

Chapter 13

Limitations of Up-Bounded Delay Models

In this chapter we study the behavior of the so-called delay-insensitive (DI) circuits in response to sequences of input changes. The correctness of the behaviors of such circuits is independent of the relative delays in their components and wires. It will be shown that the class of behaviors realizable by delay-insensitive circuits is quite restricted.

The basic result concerning fundamental-mode circuits dates from 1959 and is due to Unger [134, 135]. Unger considered circuits with single input changes. He defined an "essential hazard" in a flow table as follows: Suppose a circuit is in a stable state $\hat{a}{\cdot}b$ and the input changes first to a, then back to $\hat{a}$ and again to a. The circuit has an essential hazard if the state reached after the first input change is different from that reached after three input changes. Unger showed that no flow table with an essential hazard can have a delay-insensitive realization. In this chapter, we give a new proof of Unger's theorem. Our proof is based on the equivalence of the results of ternary simulation and GMW analysis in the input-, gate-, and wire-state model. This proof originally appeared in [122]; see also [124].

In modern design approaches, asynchronous circuits are not operated in fundamental mode. Several such approaches use some sort of "input/output mode" [18, 19, 44, 103]. We will not be defining any precise notion of input/output mode, but will later use one very simple version of this mode. Roughly speaking, in this mode the environment does not have to wait until the network has stabilized completely to give the next input change; a new input can be applied as soon as the network has given an appropriate output response. Thus the input/output mode is more "demanding" than fundamental mode. It is not surprising, therefore, that even fewer behaviors have delay-insensitive realizations when operated in this mode.

Some results about the limitations of the input/output mode circuits seem to have been "folk theorems" for quite a long time. For example, one such folk theorem is that the C-ELEMENT has no delay-insensitive input/output-mode realization. To the best of our knowledge the first proof of this was given in [19]. Here we present a somewhat more general definition of input/output-mode operation and a somewhat more general version of that result. We also show that it is impossible to design a delay-insensitive

arbiter, thus providing a new proof of a result of [2]. The results concerning the input/output mode are also based on the equivalence of ternary simulation and GMW analysis.

For some related work concerning other limitations of delay-insensitive circuits, we refer the reader to [90]. That work deals with a different set of components rather than gates.

13.1 Delay-Insensitivity in Fundamental Mode

Delay-insensitive circuits are highly desirable, since they are very robust with respect to manufacturing variations, changes in operating conditions, etc. In this section we define delay-insensitive network behaviors in fundamental mode.

In a delay-insensitive network, each gate and wire must have a state variable associated with it. Thus every output variable coincides with some state variable. For this reason, it is not possible to have an input transition in which the output also changes, and every letter associated with an input transition is a subset of $\mathcal{X}$.

A transition of a network N, from a stable state $\hat{a}\cdot b$ under new input vector a, is said to be *delay-insensitive in fundamental mode* if and only if $out(R_a(b))$ contains a single state, where $out(R_a(b))$ is the outcome of the GMW analysis in the gate-and-wire-delay model. In terms of our definitions in the previous chapter, a delay-insensitive fundamental-mode network will be represented by a serial behavior, or the corresponding complete-word behavior.

In general, not all of the transitions of a network are delay-insensitive. To illustrate this—and also to give an example of a nontrivial delay-insensitive behavior—we introduce the circuit of Figure 13.1. It is clear from Chapter 7 that ternary simulation is the correct tool for determining whether a transition is delay-insensitive. In fact, ternary simulation of an input- and feedback-state model of the circuit is sufficient. We choose to carry out the reduction procedure using $\{y_3, y_7\}$ as the feedback-vertex set. This yields

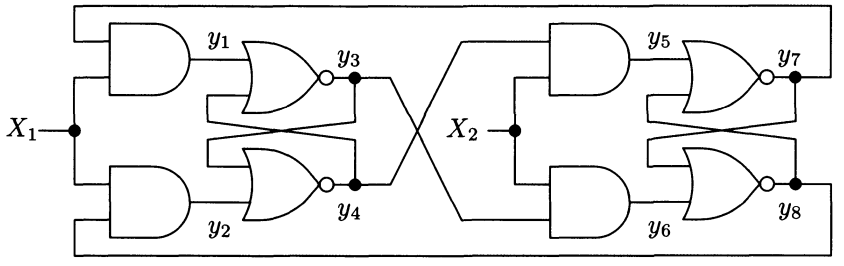


FIGURE 13.1. Gate circuit C with nontrivial delay-insensitive behavior.

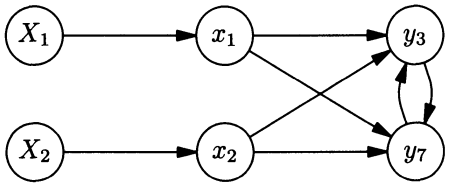


FIGURE 13.2. Reduced network corresponding to circuit C.

the reduced network of Figure 13.2 with excitation functions $\mathbf{X}_1$, $\mathbf{X}_2$ and

$$\mathbf{Y}_3 = \overline{(\mathbf{x}_1\mathbf{y}_7 + \overline{(\mathbf{x}_1\overline{(\mathbf{x}_2\mathbf{y}_3 + \mathbf{y}_7)} + \mathbf{y}_3)}}),$$

$$\mathbf{Y}_7 = \overline{(\mathbf{x}_2\overline{(\mathbf{x}_1\overline{(\mathbf{x}_2\mathbf{y}_3 + \mathbf{y}_7)} + \mathbf{y}_3)} + \overline{(\mathbf{x}_2\mathbf{y}_3 + \mathbf{y}_7)})}.$$

We summarize the delay-insensitive transitions of a network in the form of a serial behavior. The complete-word behavior for the serial behavior of the network of Figure 13.2 is shown in Figure 13.3. Since all the states involved are stable, and there is a state variable x_i associated with each input X_i, the first two components of the total state are identical to the second two components. For this reason, we omit the input excitation part X_1X_2 of total state $X_1X_2{\cdot}x_1x_2y_3y_7$, and show only the internal state $x_1x_2y_3y_7$. Since we will not be concerned with outputs for a while, we do not show them either. This is equivalent to assuming that $\mathcal{O}$ is empty. Although the transition tags on the edges are redundant, we show the set of inputs that change, for convenience.

In Figure 13.3 we show all the transitions that are delay-insensitive for our example network when it is started in stable state 0011. Note that no transition caused by a multiple-input change is delay-insensitive in this example. It is also interesting to note that, for any state reachable from the initial state for which an input change σ is allowed, any odd number of σ's takes the machine to the same state. For example, $0011 \xrightarrow{\{X_1\}} 1001$, $1001 \xrightarrow{\{X_1\}} 0001$, and $0001 \xrightarrow{\{X_1\}} 1001$. We will show that this is not a coincidence but a fundamental property of delay-insensitive networks.

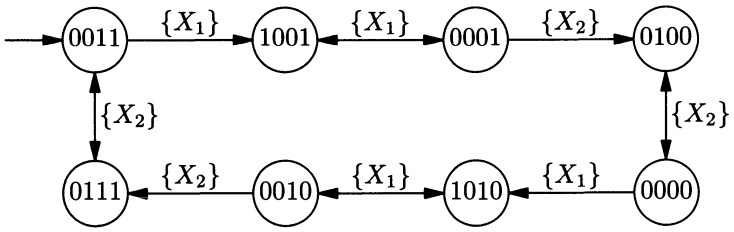


FIGURE 13.3. Delay-insensitive complete-word behavior.

13.2 Composite Functions

Before deriving certain properties that are common to all delay-insensitive networks, we introduce a technique that significantly simplifies the proofs of our results. When analyzing synchronous circuits it is quite common to use a method called "unfolding." The basic idea is to replicate the combinational part of the circuit a number of times instead of feeding back the output values; we then iteratively compute the next state. For our application, we do the unfolding using ternary extensions of the excitation functions rather than the binary functions. The idea is to form a ternary function that directly computes the results of Algorithm $\tilde{A}$ and Algorithm B. (Since the network always starts in a stable total state in fundamental mode, Algorithm $\tilde{A}$ is indeed applicable.) Since Algorithms $\tilde{A}$ and B require at most m iterations, we only need to unfold the circuit m times.

More formally, given a ternary network $\mathbf{N}$, define its *composite function* $\mathbf{F} : \{0, \Phi, 1\}^{n+m} \rightarrow \{0, \Phi, 1\}^m$ as $\mathbf{F}(\mathbf{X}\cdot\mathbf{s}) = \mathbf{S}^{(m)}(\mathbf{X}\cdot\mathbf{s})$, where $\mathbf{S}^{(h)}$ is defined recursively as follows:

$$\mathbf{S}^{(h)}(\mathbf{X}\cdot\mathbf{s}) = \begin{cases} \mathbf{s} & \text{if } h = 0, \\ \mathbf{S}(\mathbf{X}\cdot\mathbf{S}^{(h-1)}(\mathbf{X}\cdot\mathbf{s})) & \text{if } h \geq 1. \end{cases}$$

To illustrate this idea, consider the network of Figure 13.4, where $\mathbf{S}_1 = \mathbf{X}$ and $\mathbf{S}_2 = \overline{\mathbf{s}_1}(\mathbf{s}_1 + \mathbf{s}_2)$. Since the network has two state variables, we unfold it into two levels. This yields

$$\mathbf{S}_1^{(0)} = \mathbf{s}_1,$$

$$\mathbf{S}_2^{(0)} = \mathbf{s}_2,$$

$$\mathbf{S}_1^{(1)} = \mathbf{X},$$

$$\mathbf{S}_2^{(1)} = \overline{\mathbf{s}_1}(\mathbf{s}_1 + \mathbf{s}_2),$$

$$\mathbf{S}_1^{(2)} = \mathbf{X},$$

$$\mathbf{S}_2^{(2)} = \overline{\mathbf{X}}(\mathbf{X} + \overline{\mathbf{s}_1}(\mathbf{s}_1 + \mathbf{s}_2)).$$

Thus, for example, $\mathbf{F}_2(\mathbf{X}, \mathbf{s}) = \mathbf{S}_2^{(2)}$ above.

From the definition of the composite function, it follows that it is monotonic, i.e., if $\mathbf{a}\cdot\mathbf{b} \sqsubseteq \mathbf{c}\cdot\mathbf{d}$ then $\mathbf{F}(\mathbf{a}\cdot\mathbf{b}) \sqsubseteq \mathbf{F}(\mathbf{c}\cdot\mathbf{d})$. Furthermore, it is trivial to show that if $\mathbf{a}\cdot\mathbf{b}$ is a stable state of $\mathbf{N}$, i.e., if $\mathbf{b} = \mathbf{S}(\mathbf{a}\cdot\mathbf{b})$, then $\mathbf{F}(\mathbf{a}\cdot\mathbf{b}) = \mathbf{b}$. Finally, assume that network $\mathbf{N}$ is started in stable total state $\hat{a}\cdot\mathbf{b}$ and the

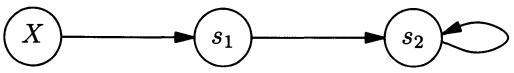


FIGURE 13.4. Network $\mathbf{N}$.

input is changed to a. Let $\mathbf{s}^A$ and $\mathbf{t}^B$ be the results of Algorithms Ã and B for this input change, respectively, and let $\mathbf{a} = lub\{\hat{a}, a\}$. Then, by the definition of $\mathbf{F}$ and Propositions 7.3 and 7.4, it can be shown that

(i)	$\mathbf{F}(\hat{a}\cdot\mathbf{b}) = \mathbf{b}$	(stability)
(ii)	$\mathbf{F}(\mathbf{a}\cdot\mathbf{b}) = \mathbf{s}^A$	(result of Alg. Ã)
(iii)	$\mathbf{F}(\mathbf{a}\cdot\mathbf{s}^A) = \mathbf{s}^A$	(stability)
(iv)	$\mathbf{F}(a\cdot\mathbf{s}^A) = \mathbf{t}^B$	(result of Alg. B)
(v)	$\mathbf{F}(a\cdot\mathbf{t}^B) = \mathbf{t}^B$	(stability)

For example, if $\mathbf{s}^h$ denotes the hth state of the network reached during algorithm Ã, establishing property (ii) is accomplished by showing that $\mathbf{S}^{(h)}(\mathbf{a}\cdot\mathbf{b}) = \mathbf{s}^h$ for $0 \leq h \leq A$.

13.3 Main Theorem for Fundamental Mode

We are now ready to derive some (quite restrictive) properties that are common to all delay-insensitive circuits. Our main result is summarized in the following theorem:

Theorem 13.1 *Let N be any network and let $\hat{a}\cdot\hat{b}$ be a stable state of N. If $(\hat{a}\cdot\hat{b}, a\cdot b)$, and $(a\cdot b, \tilde{a}\cdot\tilde{b})$ are delay-insensitive transitions of N, then so is the transition from $\hat{a}\cdot\tilde{b}$ under input a, and the result of this transition is again the state $a\cdot b$. See Figure 13.5. Furthermore, if some vertex j has the same value in state b as in state $\hat{b}$, i.e., $\hat{b}_j = b_j = \alpha$, and there is no static hazard on this vertex during this transition, then the vertex will have the same value in $\tilde{b}$, i.e., $\tilde{b}_j = \alpha$.*

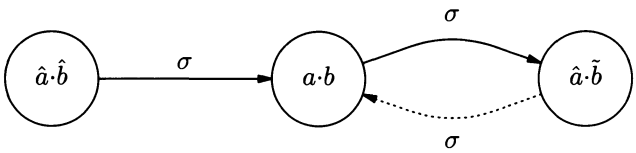


FIGURE 13.5. Illustrating the main theorem.

The theorem states that any odd number of changes of the same set of inputs must leave a delay-insensitive network in the same state. Furthermore, suppose we only consider transitions that are free of static hazards; if an output does not change value for some input change σ, then it will not change for any sequence of σ's.

An interesting special case of the theorem occurs when the network has only one input. From the theorem it follows that the state graph showing all the delay-insensitive transitions for such a network can have at most three states, assuming that we consider only the component of the state graph

that is reachable from some initial state. Since the value of the input vertex (stored in the input-delay variable) is part of the state of the network, any such graph must have at least two states. Hence, it is easy to see that the only possible graphs are the ones shown in Figure 13.6. From this we can conclude, for example, that there does not exist a delay-insensitive divide-by-2 counter such as the one discussed in Chapter 1. In fact, there does not exist a delay-insensitive mod-k counter for any $k > 1$.

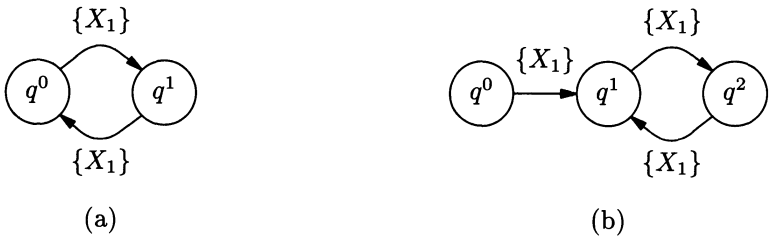


FIGURE 13.6. Delay-insensitive transitions for one-input circuit.

We prove the theorem with the aid of a series of lemmas. The following assumptions are used for Lemmas 13.1–13.3 below. Let $\mathbf{N}$ be the ternary network obtained from any Boolean network N. Assume $\mathbf{N}$ is operated according to the fundamental-mode assumption. Let $\mathbf{F}$ denote the composite network functions as defined above. Furthermore, assume that the input sequence is given by $a^0, a^1, a^2, a^3, \ldots = \hat{a}, a, \hat{a}, a, \ldots$, i.e., that the input is cycled between the binary input vectors $\hat{a}$ and a. Assume that $\hat{a}\cdot\mathbf{b}^0$ is a stable total state of $\mathbf{N}$. Let $\mathbf{b}^{i,i+1}$ denote the result of Algorithm $\tilde{\text{A}}$ for the transition from the stable state $a^i\cdot\mathbf{b}^i$ when the input changes to a^{i+1}. Similarly, let $\mathbf{b}^{i+1}$ denote the result of Algorithm B for the same transition. Note that we do not assume that $\mathbf{b}^{i,i+1}$ and $\mathbf{b}^{i+1}$ are binary.

The following lemma is the key lemma to all subsequent results. The lemma states that if, at some point, a vertex with a binary value does not react to an input change, it will never react thereafter.

Lemma 13.1 *If there exists an integer $k \geq 1$ such that $\mathbf{b}_j^{k-1} = \mathbf{b}_j^{k-1,k} = \mathbf{b}_j^k = \alpha \in \{0,1\}$, then $\mathbf{b}_j^{i-1,i} = \mathbf{b}_j^i = \alpha$ for all $i \geq k$.*

Proof: We prove this by induction on i. The basis, $i = k$, holds trivially by the assumptions in the lemma. Thus assume inductively that $\mathbf{b}_j^{i-1,i} = \mathbf{b}_j^i = \alpha$ for some $i \geq k$. First note that $lub\{a^{i-1}, a^i\} = lub\{a^i, a^{i+1}\} = lub\{\hat{a}, a\} = \mathbf{a}$. By the monotonicity of Algorithm B (Proposition 7.4), it follows that $\mathbf{b}^{i-1,i} \sqsupseteq \mathbf{b}^i$ and hence, by the monotonicity of the composite network function, that $\mathbf{F}_j^B(a\cdot\mathbf{b}^{i-1,i}) \sqsupseteq \mathbf{F}_j^B(a\cdot\mathbf{b}^i)$. By Property (iii) of the composite network functions, $\mathbf{F}^A(a\cdot\mathbf{b}^{i-1,i}) = \mathbf{b}^{i-1,i}$ and, in particular, $\mathbf{F}_j^A(a\cdot\mathbf{b}^{i-1,i}) = \mathbf{b}_j^{i-1,i}$ which is equal to α by the induction hypothesis. Hence, $\alpha = \mathbf{b}_j^{i-1,i} \sqsupseteq \mathbf{b}_j^i = \mathbf{F}_j^B(a\cdot\mathbf{b}^i)$, and thus $\mathbf{F}_j^B(a\cdot\mathbf{b}^i) = \alpha$. Furthermore, by Property (ii) of

the composite network functions, it follows that $\mathbf{b}_j^{i,i+1} = \mathbf{F}_j^A(a{\cdot}\mathbf{b}^i)$ and hence $\mathbf{b}_j^{i,i+1} = \alpha$. In other words, the value of vertex j after Algorithm Ã for the input change a^i to a^{i+1} will be α. Finally, by the monotonicity of Algorithm B (Proposition 7.4), it follows immediately that $\mathbf{b}^{i,i+1} \sqsupseteq \mathbf{b}^{i+1}$ and therefore that $\mathbf{b}_j^{i+1} = \alpha$. Hence, the induction step goes through and the lemma follows. □

From Lemma 13.1 we get the following corollary.

Corollary 13.1 *(Monotonicity for change sequences) For all* $k \geq 1$,

$$\mathbf{b}^{k-1,k} \sqsupseteq \mathbf{b}^{k,k+1}.$$

Proof: It suffices to show that whenever $\mathbf{b}_j^{k-1,k}$ is binary, then $\mathbf{b}_j^{k,k+1}$ has the same value. Suppose $\mathbf{b}_j^{k-1,k} = \alpha \in \{0,1\}$. From the monotonicity of Algorithm Ã (Proposition 7.1 and proof of Lemma 7.1) and the monotonicity of Algorithm B (Proposition 7.4), it follows that $\mathbf{b}_j^{k-1} = \mathbf{b}_j^k = \alpha$. Hence, $\mathbf{b}_j^{k-1} = \mathbf{b}_j^{k-1,k} = \mathbf{b}_j^k = \alpha \in \{0,1\}$ and Lemma 13.1 applies. Thus, $\mathbf{b}_j^{i-1,i} = \mathbf{b}_j^i = \alpha$ for all $i \geq k$ and, in particular, $\mathbf{b}_j^{k,k+1} = \alpha$. □

The following two lemmas give conditions on the values of a vertex after an odd and an even number of input changes, respectively. The first lemma states that if a vertex has a binary value after one input change, then it has the same value after any odd number of input changes. The second lemma is similar, but for an even number of changes.

Lemma 13.2 *If* $\mathbf{b}_j^1 = \alpha \in \{0,1\}$, *then* $\mathbf{b}_j^{2i-1} = \alpha$ *for all* $i \geq 1$.

Proof: We show this by induction on i. The basis ($i = 1$) holds trivially by the assumption in the lemma. Thus assume inductively that $\mathbf{b}_j^{2i-1} = \alpha$ for some $i \geq 1$. Since $i \geq 1$, and thus $2i - 2 \geq 0$, the state $\mathbf{b}^{2i-2,2i-1}$ is well defined. By Property (iv) of the composite network function, it follows that $\mathbf{b}^{2i-1} = \mathbf{F}(a^{2i-1}{\cdot}\mathbf{b}^{2i-2,2i-1})$ and, in particular, that $\mathbf{b}_j^{2i-1} = \mathbf{F}_j(a^{2i-1}{\cdot}\mathbf{b}^{2i-2,2i-1})$. By the same arguments, $\mathbf{b}_j^{2i+1} = \mathbf{F}_j(a^{2i+1}{\cdot}\mathbf{b}^{2i,2i+1})$. However, by Corollary 13.1 it follows that $\mathbf{b}^{2i-2,2i-1} \sqsupseteq \mathbf{b}^{2i-1,2i} \sqsupseteq \mathbf{b}^{2i,2i+1}$. Also, by assumption, $a^{2i-1} = a^{2i+1} = a$ and thus $a^{2i-1}{\cdot}\mathbf{b}^{2i-2,2i-1} \sqsupseteq a^{2i+1}{\cdot}\mathbf{b}^{2i,2i+1}$. This, together with the monotonicity of $\mathbf{F}$, shows that

$$\mathbf{F}(a^{2i-1}{\cdot}\mathbf{b}^{2i-2,2i-1}) \sqsupseteq \mathbf{F}(a^{2i+1}{\cdot}\mathbf{b}^{2i,2i+1}).$$

Thus,

$$\mathbf{b}_j^{2i-1} = \mathbf{F}_j(a^{2i-1}{\cdot}\mathbf{b}^{2i-2,2i-1}) \sqsupseteq \mathbf{F}_j(a^{2i+1}{\cdot}\mathbf{b}^{2i,2i+1}) = \mathbf{b}_j^{2i+1},$$

and since $\mathbf{b}_j^{2i-1} = \alpha$ by the induction hypothesis, it follows that $\mathbf{b}_j^{2i+1} = \alpha$ and the induction step goes through. □

Lemma 13.3 *If* $\mathbf{b}_j^2 = \alpha \in \{0,1\}$, *then* $\mathbf{b}_j^{2i} = \alpha$ *for all* $i \geq 1$.

Proof: The arguments are similar to those in the proof of Lemma 13.2. □

We are now in a position to prove Theorem 13.1.

Proof of Theorem 13.1: There are two cases to consider. If $\tilde{b} = \hat{b}$, the theorem follows immediately. Hence consider the case where $\tilde{b} \neq \hat{b}$. The dotted edge in Figure 13.5 illustrates this case. Now consider the ternary simulation of the transition caused by the input changing from $\hat{a}$ to a when the network is started in stable state $\hat{a}\cdot\hat{b}$. Since this transition is assumed to be delay-insensitive, $out(R_a(\hat{b}))$ contains a single state. In view of Theorem 7.2, it therefore follows that the ternary simulation of this transition must yield $b \in \{0,1\}^{m+n}$. Hence, Lemma 13.2 applies for each vertex of the circuit establishing the first claim of the theorem.

For the second half of the theorem, consider again the first transition, i.e., the case where the network is started in stable state $\hat{a}\cdot\hat{b}$ and the input changes to a. First, using the same arguments as above, the ternary simulation of the transition yields a binary state. In particular, $b_j = \alpha \in \{0,1\}^{m+n}$. Second, by Theorem 7.4 and the fact that the transition is hazard-free, the value on vertex j after Algorithm Ã is α too. This, together with Lemma 13.1, implies that vertex j remains α for any sequence of input changes between $\hat{a}$ and a, and in particular that $\tilde{b}_j = \alpha$. □

Using the results above, it is easy to verify that the following six types of vertex behaviors are the only ones possible for a vertex in a delay-insensitive network when the input alternates between the two binary input vectors $\hat{a}$ and a :

1. The vertex never reacts.

2. The vertex changes value on the first input change and keeps this value from then on.

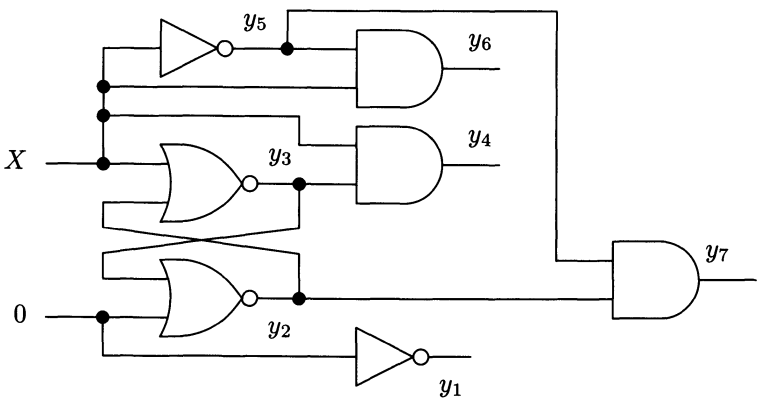


FIGURE 13.7. Gate circuit G.

3. The vertex changes value for every input change.

4. The vertex keeps the same value, although there may be a short pulse during every input change.

5. The vertex keeps the same value, although there may be a short pulse during the first input change.

6. The vertex keeps the same value for the first input change, except that there may be a short pulse during this change. For the remaining changes, the vertex changes value for every input change.

Note that only behaviors 1-3 are normally acceptable for an output vertex.

The gate circuit G of Figure 13.7 contains gates of all the above types if it is started in stable state $X = 0$, $y = y_1 \dots y_7 = 1010100$, and the input oscillates between 1 and 0. In particular, gate 1 is of type 1, gates 2 and 3 are of type 2, gate 4 is of type 5, gate 5 is of type 3, gate 6 is of type 4, and finally gate 7 is of type 6.

13.4 Delay-Insensitivity in Input/Output Mode

In this section, we show that delay-insensitive networks operated in input/output mode are even more restricted than those in fundamental mode.

13.4.1 The Main Lemma

We will show that the very simple behavior of Figure 13.8 does not have a delay-insensitive realization operated in the input/output mode, although it has a delay-insensitive realization operated in the fundamental mode. To prove this result, it suffices to consider a very limited class of behaviors.

Recall that a behavior is serial if it is direct, has no XO transitions, has at most one transition with tag $\mathcal{X}'$ for each subset $\mathcal{X}'$ of $\mathcal{X}$, and each dynamic state has at most one outgoing transition. Furthermore, we may assume that it has no ε transitions, i.e., that it is deterministic. For the purposes of this section, we restrict such behaviors even further. A *simple deterministic* behavior is a deterministic serial behavior $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$, with one additional condition: At most one external variable changes in each transition.

We need to define how a network N in the input-, gate-, and wire-state model is operated in the *input/output mode with respect to a specification* A. Assume that we have performed the appropriate projection of N to A, i.e., that N has the input vector X and the output vector O. The network must have an initial state q_1' representing q_1. Rather than finding the general behavior under all possible input sequences—as we have done for fundamental mode—we will attempt to simulate the specification by

applying to the network only the relevant input sequences. We define the *input/output-mode behavior* of N to be $B' = \langle X, \mathcal{R}, O, \mathcal{Q}', q_1', \mathcal{T}', \psi' \rangle$, where $\mathcal{R}' \subseteq \{0,1\}^m$ and $\mathcal{T}'$ will be defined later.

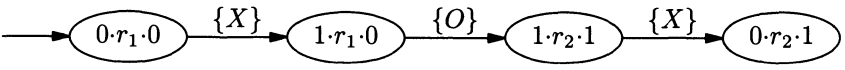


FIGURE 13.8. Behavior A_1.

We begin by an example that illustrates the main ideas and also points out the differences between the fundamental mode and the input/output mode. Consider the behavior A_1 of Figure 13.8. The simulation of A_1 by a network N is illustrated in Figure 13.9. In the fundamental mode, the initial state of a network N realizing A_1 must be stable. No such condition is imposed on N if it is operated in the input/output mode. Thus N could start in any state $q_1' = 0{\cdot}b$ such that $\psi'(q_1') = 0$. Note, however, that any state $q' = 0{\cdot}c$ with $c \in \mathit{reach}(R_0(b))$ would also have to have $\psi'(q') = 0$. This has to hold because the network output is not permitted to change, if the environment chooses not to apply any input changes for a while.

It follows that every state $c \in \mathit{reach}(R_0(b))$ must have an input transition to represent the first transition of A_1. The state reached by this transition must be of the form $1{\cdot}d$ with $\psi'(1{\cdot}d) = 0$. Observe that the internal state d may differ from c: Since $0{\cdot}c$ need not be stable, it is conceivable that the internal state can change at the instant the input change occurs. However, the output cannot change instantly because the output wire has a delay associated with it.

Every network state $1{\cdot}d$ as above must be unstable, because the specification now expects an output change. Furthermore, in any R_1-sequence

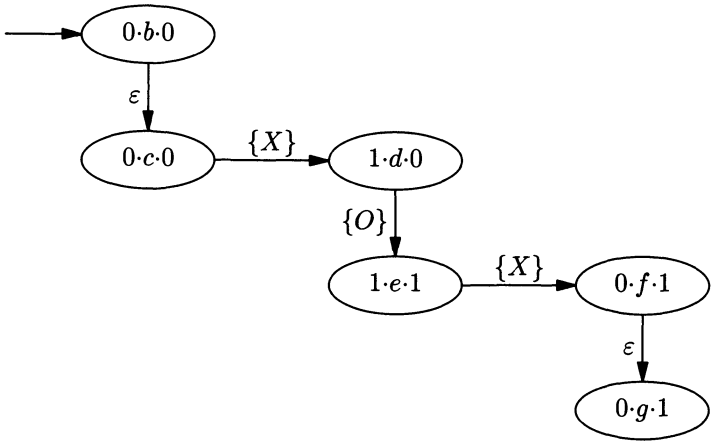


FIGURE 13.9. Simulation of A_1 by a network.

starting in d and terminating in some state in $out(R_1(d))$, the output must change exactly once. In such a sequence, every state with the old output value 0 still represents state $1{\cdot}r_1{\cdot}0$ of the specification, and every state with the new output value 1 represents $1{\cdot}r_2{\cdot}1$.

Consider now any state e that is reachable from d by an R_1-sequence and that has $\psi'(1{\cdot}e) = 1$. In fundamental mode, the environment would have to wait until a stable state is reached. There is no such condition here, and the input is allowed to change again as soon as the new output value appears. Let $0{\cdot}f$ be the state immediately after the input change; then $\psi'(0{\cdot}f) = 1$.

Finally, we must ensure that the output does not change again. Thus, every state g reachable from f by an R_0-sequence must have $\psi'(0{\cdot}g) = 1$.

The properties of any network N realizing A_1 are now summarized for convenience:

P_1 If $q_1' = 0{\cdot}b$, every state c (including b) reachable by an R_0-sequence from b must have $\psi'(0{\cdot}c) = 0$.

P_2 The input is allowed to change in any state $0{\cdot}c$, defined as above, and the state $1{\cdot}d$ reached after this input change must be unstable and must satisfy $\psi'(1{\cdot}d) = 0$.

P_3 In every R_1-sequence starting with d and ending with a state in $out(R_1, d)$, O changes exactly once.

P_4 Let e be any state that can be reached by an R_1-sequence from d and that has $\psi'(1, e) = 1$. Then the input is allowed to change again. The state $0{\cdot}f$ so reached must have $\psi'(0, f) = 1$.

P_5 Every state g reached from f by an R_0-sequence must have $\psi'(0{\cdot}g) = 1$.

In general, we define a set $\mathcal{Q}'$ of states of N that is used in the simulation of a simple deterministic behavior A; the definition is by induction. We also define inductively a function $\phi : \mathcal{Q} \to \mathcal{P}(\mathcal{Q}')$; this function specifies, for each q, the set $\phi(q)$ of all the states of the network that have to behave like q.

Basis: $\mathcal{Q}' = \{q_1'\}$, $\phi(q_1) = \{q_1'\}$.

Induction step: Suppose $q' = a{\cdot}b \in \mathcal{Q}'$, and $q' \in \phi(q)$, where $q = a{\cdot}r$.

- If q is static, we have two cases:

  1. If $p' = a{\cdot}\tilde{b}$ and $bR_a\tilde{b}$, add p' to $\mathcal{Q}'$ and to $\phi(q)$.
  2. If q is static, $p = \tilde{a}{\cdot}r$, and $(q, p) \in \mathcal{T}$, let $\tilde{b}$ be any state R_a-related to b in N, and let $p' = \tilde{a}{\cdot}\tilde{b}$. Add p' to $\mathcal{Q}'$ and to $\phi(p)$.

- If q is dynamic, then there is a unique transition (q,p) in which one output variable changes. If $p' = a{\cdot}\tilde{b}$, $bR_a\tilde{b}$, we have the following two cases:
  1. If $\psi'(p') = \psi(p)$, add p' to $\mathcal{Q}'$ and to $\phi(p)$.
  2. If $\psi'(p') \neq \psi(p)$, then N is rejected, since it is unable to realize A.

This construction is continued until N is rejected as above, or $\mathcal{Q}'$ cannot be enlarged any further.

Assuming that N has not been rejected, we are now in a position to state the basic rule for the input/output mode of operation: *The environment may change the input only if the network N is in a state q' representing a static state of the behavior A, i.e., if $q' \in \phi(q)$ and q is static.* More precisely, we define the transition set $\mathcal{T}'$ of B':

- If $q' \in \phi(q)$, q is dynamic, and $q' = a{\cdot}b$, then $(q',p') \in \mathcal{T}'$ for all $p' = a{\cdot}\tilde{b}$, where $bR_a\tilde{b}$.
- If $q' \in \phi(q)$, q is static, $q' = a{\cdot}b$, $q = a{\cdot}r$, $p = \tilde{a}{\cdot}r$ and (q,p) is an allowed input transition in A, then $(q',p') \in \mathcal{T}'$, where $p' = \tilde{a}{\cdot}\tilde{b}$ and $bR_a\tilde{b}$.

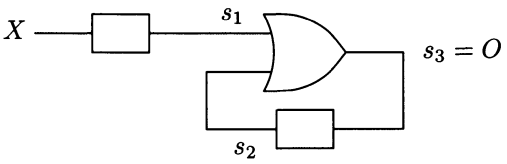


FIGURE 13.10. Network $C_{13.10}$.

We illustrate the construction using the network[1] of Figure 13.10. First, we show that this network operated in the fundamental mode realizes behavior A_1. We use stable state $X{\cdot}s_1s_2s_3 = X{\cdot}s_1s_2O = 0{\cdot}000$ as the initial state. If the input changes to 1, we have the following sequence of states:

$$0{\cdot}000 \xrightarrow{\{X\}} 1{\cdot}000 \xrightarrow{\varepsilon} 1{\cdot}100 \xrightarrow{\{O\}} 1{\cdot}101 \xrightarrow{\varepsilon} 1{\cdot}111.$$

The last state reached is stable. Now the second input change can be applied; this results in the sequence

$$1{\cdot}111 \xrightarrow{\{X\}} 0{\cdot}111 \xrightarrow{\varepsilon} 0{\cdot}011.$$

Thus we see that the specification behavior A_1 is indeed implemented.

---

[1]Formally, in the input-, gate-, and wire-state model, we should have one delay for the input X and another for the wire from the input delay to the OR gate. One can easily verify that the extra delay would not change any of the conclusions that are about to be made.

Now consider the input/output mode. First, we have $\mathcal{Q}' = \{0{\cdot}000\}$, and $0{\cdot}000 \in \phi(0{\cdot}r_1{\cdot}0)$. Next, since $(0{\cdot}r_1{\cdot}0, 1{\cdot}r_1{\cdot}0) \in \mathcal{T}$, we have $\mathcal{Q}' = \{0{\cdot}000, 1{\cdot}000\}$, and $1{\cdot}000 \in \phi(1{\cdot}r_1{\cdot}0)$. We then add $1{\cdot}100, 1{\cdot}101$, and $1{\cdot}111$ to $\mathcal{Q}'$. We also add $1{\cdot}100$ to $\phi(1{\cdot}r_1{\cdot}0)$, and $1{\cdot}101$ and $1{\cdot}111$ to $\phi(1{\cdot}r_2{\cdot}1)$. The input is permitted to change in states $1{\cdot}101$ and $1{\cdot}111$, since they represent the static state $1{\cdot}r_2{\cdot}1$. Hence we also add the states $0{\cdot}101$ and $0{\cdot}111$, to $\mathcal{Q}'$ and to $\phi(0{\cdot}r_2{\cdot}1)$. From state $0{\cdot}101$, we can reach $0{\cdot}001$ and hence $0{\cdot}000$. Here the output is incorrect. Hence $C_{13.10}$ fails to realize A_1.

We are now in a position to prove that there does not exist any network that realizes A_1 when operated in the input/output mode.

Lemma 13.4 *The behavior A_1 does not have a DI input/output-mode realization.*

Proof: If gate network N with initial state q_1' is a delay-insensitive input/output realization of behavior A_1, then it must have the properties $P_1 - P_5$ listed above.

We show that if a delay-insensitive input/output-mode realization (N, q_1') of A_1 existed, then we could construct a network $C_{13.11}$ that would have contradictory properties.

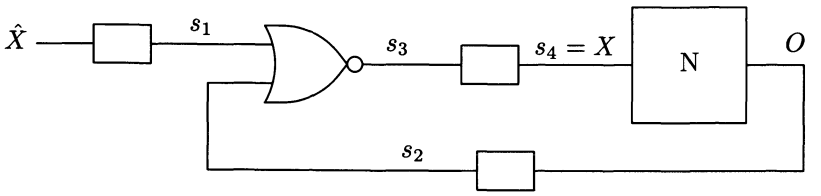


FIGURE 13.11. Network $C_{13.11}$.

Consider the network $C_{13.11}$ that uses N as shown in Figure 13.11. Notice that a delay element is introduced for every wire.[2] Since Network N also contains a delay element for each wire, we have an input-, gate-, and wire-state network model for $C_{13.11}$. Let s' denote the vector of internal state variables of N, except for the output variable, which is denoted by O.

The initial state of $C_{13.11}$ is $\hat{X}{\cdot}ss'O = \hat{X}{\cdot}s_1s_2s_3Xs'O = 0{\cdot}1000b0$. We then have the following $\hat{R}_0$-sequence:

$$1000b0 \longrightarrow 0000b'0 \longrightarrow 0010c0 \longrightarrow 0011d0.$$

Note that, in all the steps above, the output O of N has been stable, as guaranteed by conditions P_1 and P_2. Condition P_2 also requires state $1{\cdot}d0$

[2] As before we omit the input delay since it has no effect on the conclusions we will draw.

of N to be unstable. By P_3, N eventually reaches a state $1{\cdot}e1$, for some vector e. Thus we must have an $\hat{R}_0$-sequence

$$0011d0 \longrightarrow^* 0011e1.$$

From P_4 and P_5 it now follows that O cannot change any more, even if X becomes 0 again; this has to hold for all possible values that s' may reach. Thus, the s'-component of the state of $C_{13.11}$ becomes irrelevant, and we replace it by # from now on. We have the following $\hat{R}_0$-sequence:

$$0011e1 \longrightarrow 0111\#1 \longrightarrow 0101\#1 \longrightarrow 0100\#1.$$

In the last state, the variables s_1, s_2, s_3, X, and O are stable and will not become unstable again. It follows that the outcome of the GMW analysis of $C_{13.11}$ started in state $0{\cdot}1000b0$, always yields states of the form $0{\cdot}0100\#1$, i.e.,

$h \in out(\hat{R}_0(1000b0))$ implies the O component of h is 1.

Consequently, even in the presence of arbitrary gate and wire delays, the final outcome of the transition yields $O = 1$. We also observe that, in the analysis above, N is operated in input/output mode with respect to behavior A_1.

Next we show that ternary simulation of $C_{13.11}$ contradicts the conclusion reached above. Note that, by condition P_1, as long as the input X of N is 0, the excitation of the output delay must be 0. Hence the output delay is initially stable. Algorithm A produces the following sequence:

$$0{\cdot}1000b0 \longrightarrow 0{\cdot}\Phi000\#0 \longrightarrow 0{\cdot}\Phi0\Phi0\#0 \longrightarrow 0{\cdot}\Phi0\Phi\Phi\#0,$$

where the # here indicates that we don't know the values of the s' portion of the state. We trivially have $1000b0\hat{R}_0^*1000b0$, and we have shown above that $1000b0\,\hat{R}_0{}^*\,0011e1$, i.e., both $1000b0$ and $0011e1$ are reachable from $1000b0$ (in zero or more steps). Consequently, the output O can take the values 0 and 1 in the GMW analysis of the network. But then, by Proposition 7.2, Algorithm A of the ternary simulation must produce $O = \Phi$. Subsequently, s_2 becomes Φ, and the final result of Algorithm A has the form $0{\cdot}\Phi\Phi\Phi\Phi\mathbf{t}\Phi$ for some vector $\mathbf{t}$ of ternary values.

Applying Algorithm B to state $0{\cdot}\Phi\Phi\Phi\Phi\mathbf{t}\Phi$, we find that it terminates in the second step with state $0{\cdot}0\Phi\Phi\Phi\mathbf{t}\Phi$. Consequently, Algorithm B predicts that O has the value Φ. But then, by Theorem 7.2, there exists a state in the outcome of the GMW analysis where $O = 0$. This contradicts the GMW analysis above. Therefore, the network N with the postulated properties cannot exist, and we have proved that behavior A_1 does not have a delay-insensitive gate realization operated in the input/output mode. □

13.4.2 Some Behaviors Without DI Realizations

With the lemma above it is easy to verify that the behaviors of the NOR latch, and two basic components of delay-insensitive design, namely, the JOIN and the TOGGLE [46, 107], do not have delay-insensitive realizations in the input/output mode.

First, the latch of Figure 12.4 has the following sub-behavior:

$$00{\cdot}10{\cdot}10 \xrightarrow{\{X_1\}} 10{\cdot}10{\cdot}10 \xrightarrow{\{O_1,O_2\}} 10{\cdot}01{\cdot}01 \xrightarrow{\{X_1\}} 00{\cdot}01{\cdot}01.$$

Thus, simply by setting X_2 to 0 and ignoring the output O_1, we obtain the behavior

$$0{\cdot}10{\cdot}0 \xrightarrow{\{X_1\}} 1{\cdot}10{\cdot}0 \xrightarrow{\{O_2\}} 1{\cdot}01{\cdot}1 \xrightarrow{\{X_1\}} 0{\cdot}01{\cdot}1,$$

which is isomorphic to A_1, if we identify X with X_1 and O with O_2.

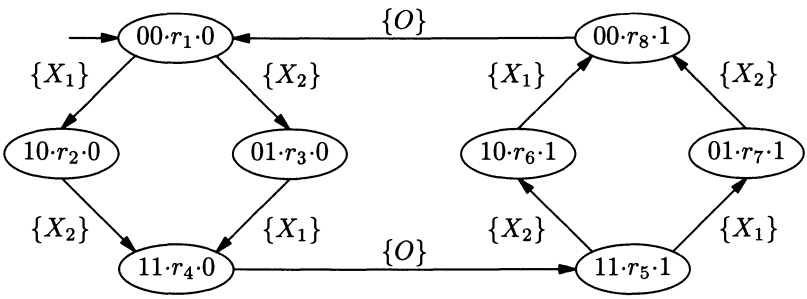


FIGURE 13.12. Behavior of JOIN.

In Figure 13.12 we show a behavior of the JOIN with inputs X_1 and X_2 and output O. Starting from state $00{\cdot}r_1{\cdot}0$ or $11{\cdot}r_5{\cdot}1$, the JOIN produces no output when only one input changes. When the second input changes, the JOIN then changes its output to agree with the two inputs. The JOIN has the sub-behavior:

$$10{\cdot}r_2{\cdot}0 \xrightarrow{\{X_2\}} 11{\cdot}r_4{\cdot}0 \xrightarrow{\{O\}} 11{\cdot}r_5{\cdot}1 \xrightarrow{\{X_2\}} 10{\cdot}r_6{\cdot}1.$$

If we set the input X_1 to 1 and associate X_2 with X, we obtain a behavior isomorphic to A_1.

Figure 13.13 shows a behavior of a TOGGLE with input X and outputs O_1 and O_2. If we count the input changes starting from the initial state $0{\cdot}r_1{\cdot}00$, each odd input change causes a change in output O_1 and each even input change causes a change in output O_2. The TOGGLE contains the behavior

$$0{\cdot}r_1{\cdot}00 \xrightarrow{\{X\}} 1{\cdot}r_1{\cdot}00 \xrightarrow{\{O_1\}} 1{\cdot}r_2{\cdot}10 \xrightarrow{\{X\}} 0{\cdot}r_2{\cdot}10 \xrightarrow{\{O_2\}} 0{\cdot}r_3{\cdot}11,$$

and we obtain a behavior with the same language as A_1 by ignoring the second output.

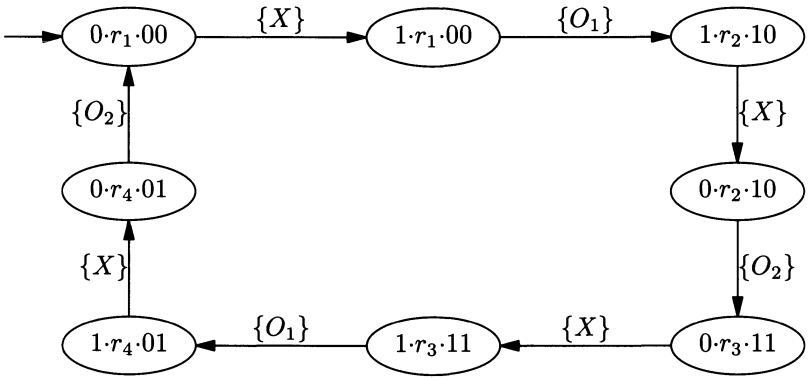


FIGURE 13.13. Behavior of TOGGLE.

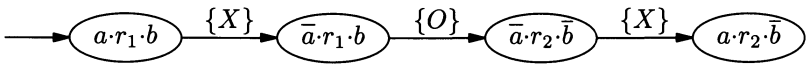


FIGURE 13.14. A generalized version of A_1.

By means of slight modifications in the proof of Lemma 13.4, we can show that three other behaviors also lack delay-insensitive input/output-mode realizations.

Lemma 13.5 *Any behavior having the form shown in Figure 13.14, where* $a, b \in \{0, 1\}$*, does not have a DI gate realization in input/output mode.*

Proof: In case $ab = 10$, repeat the arguments of Lemma 13.4, but with network $C_{13.11}$ modified as follows. Insert an inverter in series with a delay in the wire leading to the input X of network N.

In case $ab = 01$, modify network $C_{13.11}$ by the addition of an inverter in series with a delay in the wire leaving output O of network N.

In case $ab = 11$, modify network $C_{13.11}$ by the addition of two inverters with delays as indicated in the two cases above. □

13.4.3 Nontrivial Sequential Behaviors

An example of a simple deterministic behavior that does have a delay-insensitive input/output-mode realization is shown below. It is realizable by an inverter with input X and output O. The behavior is rather trivial, however, since every input vector uniquely determines the static state eventually reached by the behavior.

$$0{\cdot}r_1{\cdot}1 \xrightarrow{\{X\}} 1{\cdot}r_1{\cdot}1 \xrightarrow{\{O\}} 1{\cdot}r_2{\cdot}0 \xrightarrow{\{X\}} 0{\cdot}r_2{\cdot}0 \xrightarrow{\{O\}} 0{\cdot}r_1{\cdot}1.$$

To eliminate such trivial cases, we impose the following condition on simple deterministic behaviors:

A behavior $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$ is *nontrivial* if there exists at least one input vector $X = a$ for which there are at least two static states $a \cdot r_1 \cdot b$ and $a \cdot r_2 \cdot b'$ where $b \neq b'$ and one of the states is reachable from the other.

We have the following result:

Theorem 13.2 *No nontrivial simple deterministic behavior A with a binary input has a delay-insensitive gate realization in the input/output mode.*

Proof: Suppose $a \cdot r_2 \cdot b'$ is reachable from $a \cdot r_1 \cdot b$. Since $b \neq b'$, they must differ in at least one component. Without loss of generality, assume that these two output vectors differ in their last component, i.e., that $b = cd$ and $b' = c'\overline{d}$, where $d \in \{0, 1\}$.

In case $ad = 00$, we can apply the following reasoning: Start in state $0 \cdot r_1 \cdot c0$, which is stable. For $0 \cdot r_2 \cdot c'1$ to be reachable from $0 \cdot r_1 \cdot c0$, we must change X to 1 and then back to 0 some finite number of times. At some point in this sequence we must have a state $0 \cdot r_3 \cdot e0$, where the output has not yet changed, but from which we can reach state $0 \cdot r_2 \cdot c'1$ with two input changes. Thus, we must have the sub-behavior

$$0 \cdot r_3 \cdot e0 \xrightarrow{\{X\}} 1 \cdot r_3 \cdot e0 \longrightarrow 1 \cdot r_4 \cdot fg \xrightarrow{\{X\}} 0 \cdot r_4 \cdot fg \longrightarrow 0 \cdot r_2 \cdot c'1.$$

We can now consider two subcases.

Case 1: If $g = 1$, then the sequence above projects to the behavior A_1, if we ignore all but the first and the last components. By Lemma 13.4, A cannot be realized.

Case 2: $g = 0$. We now have the following sequence:

$$0 \cdot r_3 \cdot e0 \xrightarrow{\{X\}} 1 \cdot r_3 \cdot e0 \longrightarrow 1 \cdot r_4 \cdot f0 \xrightarrow{\{X\}} 0 \cdot r_4 \cdot f0 \longrightarrow 0 \cdot r_2 \cdot c'1,$$

where state $1 \cdot r_4 \cdot f0$ represents a static state in behavior A, because the output is not changing, by assumption. By Theorem 13.1, since the output does not change in the first step, it cannot change in the second step. Hence, this behavior is not realizable by any network, even if it is operating in the fundamental mode.

The other cases, where $ad = 10$ or $ad = 01$ or $ad = 11$ are all dealt with similarly, using Lemma 13.5. □

We close by proving that it is impossible to construct a delay-insensitive gate circuit that would act as an arbiter. This problem was considered by [2], where a proof of this was given in a totally different formalism. Consider the behavior of Figure 13.15. It represents the essential function in the arbitration process. The two inputs represent requests for the use of a single resource. When X_1 and X_2 are both 1, only one of them can be served. The arbiter then has to decide whether to grant the resource to X_1

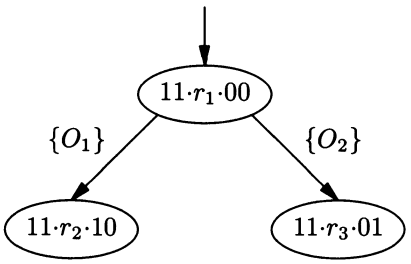


FIGURE 13.15. An arbiter behavior.

by setting $O_1 = 1, O_2 = 0$, or to X_2 by setting $O_1 = 0, O_2 = 1$. The arbiter is not allowed to always give preference to one of its inputs but must, in fact, implement the critical race. Note that the behavior of Figure 13.15 satisfies the condition that there are two distinct static states for one input vector. However, one state need not be reachable from the other; thus we are unable to apply Theorem 13.2. Nevertheless, we are able to prove the following result.

Theorem 13.3 *Suppose a direct behavior A has an initial dynamic state $q_0 = a{\cdot}r_0{\cdot}bc$, where b and c are the values of one-bit output vectors O_1 and O_2. Suppose further that $q_1 = a{\cdot}r_1b\bar{c}$ and $q_2 = a{\cdot}r_2{\cdot}\bar{b}c$ are two static states such that $(q_0, q_1), (q_0, q_2) \in \mathcal{T}$ and that no output sequences other than $\varepsilon, \{O_1\}$, and $\{O_2\}$ are allowed. Then this behavior does not have a delay-insensitive realization in the input/output mode.*

Proof: Assume that a delay-insensitive realization N exists. This means that we can use the input-, gate-, and wire-state network model to analyze N. By Theorem 7.2, we know that the outcome of Algorithm B must yield $O_1 = O_2 = \Phi$. By Lemma 7.9, there exists a nontransient cycle reachable from the network state representing r_0 in which the variables O_1 and O_2 take both values. Thus, N is capable of producing output sequences that are not allowed. □

13.5 Concluding Remarks

In this chapter, we have confined ourselves to networks of *gates*. Consequently, basic elements like the JOIN, C-ELEMENT,[3] TOGGLE, and ARBITER were not allowed. This choice was made because we wanted to investigate the basic limitations of gate circuits, in view of the well-established use of gates as primitive elements for the design of synchronous circuits.

[3]The C-ELEMENT is similar to the JOIN, but permits the environment to "withdraw" an input change; it will be treated in more detail in Chapter 15.

Delay-insensitivity needs to be redefined for switch-level models of CMOS circuits, since the "components" and "wires" are not as easily identified. One could consider the components to be transistors, but it may be more practical to consider complex cells as components. Similarly, which wires should be taken into account and which can be ignored is arguable. However, the most important decision is which race model should be used. Since CMOS circuits may produce intermediate voltage levels during operation, it is natural to require that the XMW race model be used. In view of Theorems 7.6 and 7.7 it thus follows that our results about nonrealizability apply to these more more modern technologies as well. Furthermore, since these theorems apply both to feedback delay models as well as to more elaborate network models, it follows that these results are quite insensitive to the exact definition of "components" and "wires" in a CMOS network.

We may draw the following two conclusions from the present chapter:

- If one wants to realize components like JOINs, C-ELEMENTs, TOGGLEs, or latches by circuits using only gates, then one has to make some assumptions about the gate and wire delays.
- A set of components different from the set of logic gates is needed for the realization of any significant class of delay-insensitive behaviors.

Sets of primitive components for particular classes of delay-insensitive behaviors have been suggested in [46, 146], for example. We will return to this in Chapter 15.

Chapter 14

Symbolic Analysis

The race analysis algorithms and the behaviors introduced so far in this book all use the tacit assumption that the states of a network are represented explicitly. Since the state space grows exponentially with the size of the network, such a representation can only be used for relatively small circuits. In this chapter we consider representing states and other similar objects symbolically.

In Section 14.1 we discuss a method, based on ordered binary decision diagrams (OBDDs), for representing Boolean functions. This method overcomes, in practice, many of the drawbacks of the more traditional ways of representing such functions, since it permits a compact representation for complex Boolean functions. In Section 14.2 we show how mathematical objects like sets, relations, and behaviors can be represented concisely by OBDDs. We also demonstrate how traditional algorithms can be rephrased in terms of OBDD manipulations.

In Sections 14.3 and 14.4 we derive a symbolic representation of a behavior directly from a network. In particular, in Section 14.4 we convert some of the race analysis algorithms discussed earlier in the book to symbolic form. As a side effect, we derive an efficient and practical algorithm for computing the minimum and maximum delays in a combinational network, finally providing a solution to the problem introduced in Section 1.2.

We next explore algorithms for determining whether a symbolic behavior realizes another symbolic behavior. In Section 14.5 we give an algorithm which, if successful, guarantees that a behavior realizes another one. However, there are situations in which our algorithm can be overly cautious, reporting failure when the realization relation actually holds.

Finally, in Section 14.6 we discuss a method, called "model checking," which can be used to determine the validity of a temporal-logic formula with respect to a behavior. This technique is invaluable for checking that a specification satisfies some desirable properties. Furthermore, for systems in which the only specification consists of a collection of properties that the system should satisfy, model checking can be used directly to verify that an implementation behavior extracted from a network satisfies these properties.

14.1 Representing Boolean Functions

An efficient method for representing and manipulating symbolic expressions constitutes a corner-stone in symbolic analysis. In the context of digital circuits, such a method usually involves representing and manipulating Boolean functions. For this reason, we begin with a brief digression into methods of representing Boolean functions.

Ideally, a representation of Boolean functions should satisfy the following requirements:

1. It should allow an efficient test for the equality of two functions.
2. It should be able to represent Boolean functions of a large number of variables. To be of practical significance, the representation must be able to handle functions of at least 30–40 variables.
3. It should permit efficient computation of common operations, such as complement, product, and sum of Boolean functions. Also, functional composition and quantification over a set of variables is often needed.
4. It should not require excessive amounts of storage for common functions.

Unfortunately, the existence of a representation satisfying these requirements would imply the existence of an efficient solution to the Boolean tautology problem—a well-known NP-hard problem. Therefore, unless P = NP, no such representation exists. Consequently, we have to be satisfied with some heuristic method that works well in practice but has an exponential worst-case behavior.

A common way of representing a Boolean function is by a Boolean expression, such as a sum-of-products. This representation has the disadvantage that a sum-of-products denoting the complement of a sum-of-products E can be exponentially larger than E. Other common representations, like expression trees or expression DAGs (directed acyclic graphs), solve the problem of exponential blowup when performing a single operation, but pay the price of making comparisons extremely time-consuming.

The OBDD is a Boolean function representation that satisfies many of the requirements listed above. OBDDs were originally proposed by [82] and [1], further refined by [53], and made practical and popular by [11]. An OBDD represents a collection of Boolean functions as a forest of rooted DAGs. This is illustrated by the example of Figure 14.1, where four distinct Boolean functions are represented. There are two leaf vertices (represented as squares): one labeled 0 and the other labeled 1. Each internal vertex (represented as a circle) is labeled with a variable. It has two outgoing edges, one corresponding to 1 and the other to 0. The 1-edge represents the Boolean function for the case where the variable is 1, and the 0-edge corresponds to the case where the variable is 0. In Figure 14.1 the 1-edges

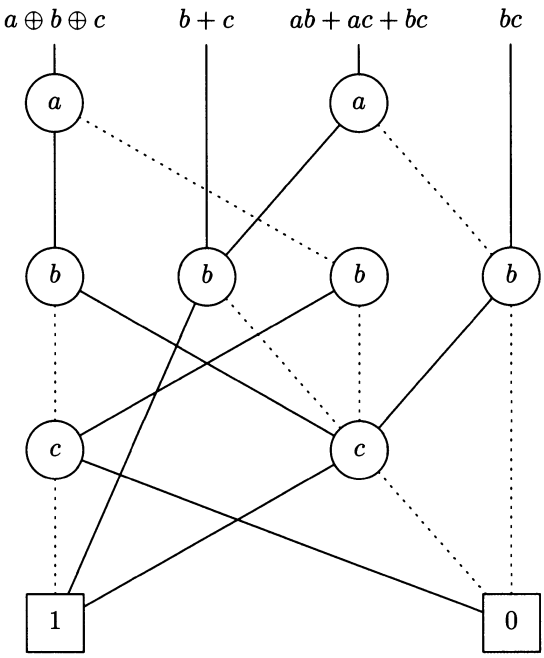


FIGURE 14.1. Example of ordered binary decision diagram.

are shown as solid lines, whereas the 0-edges are dotted. For a given assignment of binary values to the variables, the value of the function is found by following the path (corresponding to the assignment) from its root to one of the leaves.

The order in which variables are tested in the OBDD follows a global total order in which each variable occurs only once. In Figure 14.1 the order is a, b, c. If we ensure that no OBDD vertex is created with its 1-edge leading to the same successor vertex as its 0-edge and that no two OBDD vertices with the same variable label have the same 0-successors (vertices reached by the 0-edges) and the same 1-successors, we can show that no two vertices define the same Boolean function. Consequently, testing two Boolean functions for equality is trivial: the functions are equal if and only if they are represented by the same OBDD vertex.

Computing the product or sum of two Boolean functions represented by OBBDs can be done efficiently by a recursive procedure. The size of the resulting OBDD is bounded by the product of the sizes of the OBDDs representing the two functions. In practice, the resulting OBDD is often significantly smaller. The size of the OBDD representing the complement of a Boolean function is exactly the same as that representing the function.

In the OBDD representation, it is straightforward to perform universal and existential quantification of some of the variables of a Boolean function.

Also, the operation of substitution of functions for some of the variables of a given function can be performed efficiently. We use the following notation:[1]

$$\forall v_i.\, f(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n) = \\ f(v_1, \ldots, v_{i-1}, 0, v_{i+1}, \ldots, v_n) * f(v_1, \ldots, v_{i-1}, 1, v_{i+1}, \ldots, v_n)$$

and

$$\exists v_i.\, f(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n) = \\ f(v_1, \ldots, v_{i-1}, 0, v_{i+1}, \ldots, v_n) + f(v_1, \ldots, v_{i-1}, 1, v_{i+1}, \ldots, v_n).$$

If it is understood that v is one of the variables of f, we write $\forall v.\, f$ and $\exists v.\, f$ for short. We also extend this notation to quantifications over vectors of Boolean variables. For example, if v is the vector (v_1, v_2) then we write

$$\begin{aligned} \exists v.\, f(v_1, v_2, v_3, \ldots, v_n) \quad = \quad & f(0, 0, v_3, \ldots, v_n) + \\ & f(0, 1, v_3, \ldots, v_n) + \\ & f(1, 0, v_3, \ldots, v_n) + \\ & f(1, 1, v_3, \ldots, v_n). \end{aligned}$$

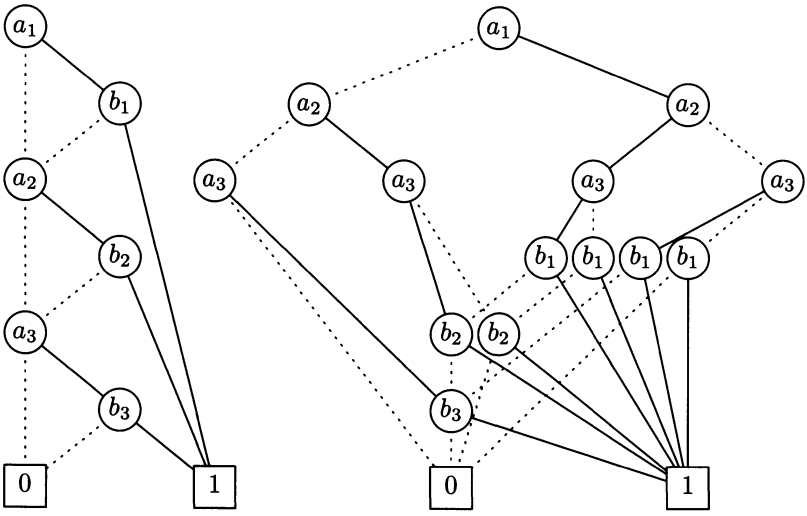


FIGURE 14.2. OBDDs for $a_1b_1 + a_2b_2 + a_3b_3$ using different orderings.

The variable order can have a significant effect on the size of the OBDD of a function. For example, in Figure 14.2 we show the function $a_1b_1 + a_2b_2 + a_3b_3$ for two different variable orderings. If we generalize this expression to $a_1b_1 + a_2b_2 + \cdots + a_nb_n$, we see that the OBDD using the ordering

[1] To avoid ambiguity, in this chapter we return to the use of the explicit symbol $*$ for Boolean multiplication.

$a_1, b_1, a_2, b_2, \ldots, a_n, b_n$ is of size $2n+2$, whereas the OBDD using the ordering $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n$ is of size 2^{n+1}. In general, finding an optimal ordering is an NP-hard problem; thus heuristics must be used. Fortunately, in most OBDD implementations, it is possible to change the order of adjacent variables efficiently. By repeating this procedure, one can devise reordering schemes that find local minima. In practice, such dynamic schemes are quite useful, although they are slow. Finally, it should be pointed out that there are Boolean functions for which no variable ordering yields a small OBDD. The classical example here is the nth output of an n-bit by n-bit binary multiplier. It has been shown [14] that the OBDD representing this function is of size exponential in n for every variable ordering. Surprisingly, although most Boolean functions have OBDDs of exponential size (as can be seen by a simple counting argument), most functions encountered in computer-aided design algorithms have small OBDD representations. It is this pragmatic observation that has lead to the widespread use of OBDDs.

In summary, as a practical way of representing Boolean functions, OBDDs meet most of the requirements listed in the beginning of this section. For a more thorough treatment of OBDDs, including more efficient representations and various applications, the reader is referred to the survey article [15]. In the remainder of this chapter, we develop algorithms under the assumption that all Boolean functions are represented by OBDDs.

14.2 Symbolic Representations

Given that we can represent, manipulate, and compare Boolean functions efficiently, an attractive way of solving many problems is to represent the objects involved as Boolean functions and to phrase the relevant algorithms in terms of operations on Boolean functions. In this section we describe some of the common encoding techniques that we use in the remainder of the chapter. It should be emphasized that the techniques we discuss do not rely on the OBDD representation of Boolean functions. Any representation will suffice, as long as it supports efficient equality checking, composition, and quantification.

14.2.1 Finite Domains

A common approach to mapping a problem concerning objects drawn from some finite domain into a problem phrased in terms of Boolean functions is to encode the domain as a vector of Boolean functions. If the domain $\mathcal{D}$ has n elements, we can encode its elements as $\lceil \log_2 n \rceil$-bit binary numbers. Sometimes the encoding is obvious—as is the case of numbers drawn from some finite subset of the integers. In other cases, the mapping can be chosen arbitrarily. For simplicity, the mapping is usually a one-to-one function, i.e., there is a unique binary number associated with every element $d \in \mathcal{D}$.

Let $\pi: \mathcal{D} \rightarrow \{0,1\}^n$ denote such a function, and let π_i denote the ith bit of the encoding. A function $F: \mathcal{D} \rightarrow \mathcal{D}$ is encoded in the obvious way, i.e., as a function $f: \{0,1\}^n \rightarrow \{0,1\}^n$ such that $f = (f_1, f_2, \ldots, f_n)$ and $f_i(\pi(d)) = \pi_i(F(d))$, for all $d \in \mathcal{D}$ and $1 \leq i \leq n$.

To illustrate the process of encoding a finite domain—and also to introduce an important encoding used in the remainder of the chapter—consider representing the domain $\{0, \Phi, 1\}$. Since there are three elements, the encoding must use at least two bits. One possibility is a "dual-rail" encoding in which the ternary value $\mathbf{a}$ is encoded as pair $(a.1, a.0)$, as shown in Table 14.1. Note that pair $(0,0)$ is not used. We use the convention

TABLE 14.1. Dual-rail encoding of ternary values.

Ternary value $\mathbf{a}$	Encoded value $(a.1, a.0)$
0	$(0,1)$
Φ	$(1,1)$
1	$(1,0)$

that variables like $\mathbf{a}_i$, $\mathbf{b}_i$, etc., have encoded versions $(a_i.1, a_i.0)$, $(b_i.1, b_i.0)$, etc., and that ternary functions, like $\mathbf{S}()$, $\mathbf{T}()$, etc., have encoded versions $(S.1(), S.0())$, $(T.1(), T.0())$, etc. In Table 14.2 we show the encoded versions of the ternary extensions of the Boolean functions ID, NOT, AND, OR, and *lub* of two ternary values. One verifies that, if $\pi(\mathbf{a}) = (a.1, a.0)$ and $\pi(\mathbf{b}) = (b.1, b.0)$, then $\mathbf{a} \sqsubseteq \mathbf{b}$ if and only if $b.1 * b.0 + \overline{a.1} * b.0 + \overline{a.0} * b.1 = 1$.

TABLE 14.2. Dual-rail versions of ternary operations.

Ternary operation $\mathbf{f}()$	Encoded version $f.1()$	$f.0()$
$\mathbf{a}$	$a.1$	$a.0$
$\overline{\mathbf{a}}$	$a.0$	$a.1$
$\mathbf{a} * \mathbf{b}$	$a.1 * b.1$	$a.0 + b.0$
$\mathbf{a} + \mathbf{b}$	$a.1 + b.1$	$a.0 * b.0$
$lub\{\mathbf{a}, \mathbf{b}\}$	$a.1 + b.1$	$a.0 + b.0$

14.2.2 Sets

Given a finite domain $\mathcal{D}$ and a binary encoding π of $\mathcal{D}$, we have two natural ways of representing subsets of $\mathcal{D}$: as characteristic functions or as parametric representations [71]. For our purposes, the former representation suffices. The *characteristic function*, $\chi_{\mathcal{S}}$, of a set $\mathcal{S} \subseteq \mathcal{D}$ is a *membership predicate*, i.e., given an element $s \in \mathcal{D}$, we have $\chi_{\mathcal{S}}(\pi(s)) = 1$ if and only if $s \in \mathcal{S}$. Characteristic functions are useful, because operations on sets have

operations on characteristic functions as natural counterparts, as is shown in Table 14.3. Recall that $|0|$ and $|1|$ denote the Boolean functions that are identically 0 and 1, respectively.

TABLE 14.3. Set operations on characteristic function representation.

Set and operations	Corresponding function and operation
$\emptyset$	$\|0\|$
$\mathcal{D}$	$\|1\|$
$\mathcal{S} \cup \mathcal{T}$	$\chi_{\mathcal{S}} + \chi_{\mathcal{T}}$
$\mathcal{S} \cap \mathcal{T}$	$\chi_{\mathcal{S}} * \chi_{\mathcal{T}}$
$\mathcal{S} - \mathcal{T}$	$\chi_{\mathcal{S}} * \overline{\chi_{\mathcal{T}}}$
$\mathcal{S} \subseteq \mathcal{T}$	$\forall z.\ \overline{\chi_{\mathcal{S}}(z)} + \chi_{\mathcal{T}}(z)$

14.2.3 Relations

Since relations can be viewed as sets, characteristic functions can also be used to represent relations compactly and to perform efficiently such operations as intersection, union, and difference of relations. By using function composition and quantification, we can also compute the composition of relations efficiently. For example, suppose that $P \subseteq \mathcal{S} \times \mathcal{T}$ and $R \subseteq \mathcal{T} \times \mathcal{U}$ are two binary relations with characteristic functions χ_P and χ_R, respectively. Then the characteristic function $\chi_{P \circ R}$ of the composition of the two relations can be computed as

$$\chi_{P \circ R}(x, y) = \exists z.\ \chi_P(x, z) * \chi_R(z, y).$$

Thus, as long as substitution and quantification can be performed efficiently, so can the composition of relations.

Another useful operation is the reflexive and transitive closure of a relation. The algorithm for this computation illustrates a common technique in symbolic computations—the use of fixed-point calculations. Here we introduce only the basic ideas of fixed-point calculations; for a more complete treatment of the underlying theory, the reader is referred to [61].

The system $\langle \mathcal{P}(\mathcal{S}), \subseteq \rangle$, where $\mathcal{S}$ is any set, is a partially ordered set. Because the least upper bound and the greatest lower bound of any set $\mathcal{T}$ of elements from $\mathcal{P}(\mathcal{S})$ are both defined,[2] the system is also a complete lattice.

A function $f: \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})$ is said to be *continuous* if for every nondecreasing sequence $\mathcal{S}_1 \subseteq \mathcal{S}_2 \subseteq \ldots$, where $\mathcal{S}_i \in \mathcal{P}(\mathcal{S})$, we have $f(\cup_{i \geq 1} \mathcal{S}_i) = \cup_{i \geq 1} f(\mathcal{S}_i)$ and $f(\cap_{i \geq 1} \mathcal{S}_i) = \cap_{i \geq 1} f(\mathcal{S}_i)$. For finite sets, every monotonic function is continuous [61]. For a continuous function $f: \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})$

[2]The least upper bound is the union of the sets in $\mathcal{T}$ and the greatest lower bound is their intersection.

1. (a) f has a unique *least fixed point* $\mathcal{L} \in \mathcal{P}(\mathcal{S})$, i.e., there exists an element $\mathcal{L}$ that satisfies $f(\mathcal{L}) = \mathcal{L}$ and, if $f(\mathcal{C}) = \mathcal{C}$ for some $\mathcal{C} \in \mathcal{P}(\mathcal{S})$, then $\mathcal{L} \subseteq \mathcal{C}$.

   (b) The least fixed point of the function f, written *lfp* $\mathcal{S}.f(\mathcal{S})$, is defined by $\cup_{i \geq 0} f^i(\emptyset)$, where f^i is the composition of i copies of f.

2. (a) f has a unique *greatest fixed point* $\mathcal{G} \in \mathcal{P}(\mathcal{S})$, i.e., there exists an element $\mathcal{G}$ that satisfies $f(\mathcal{G}) = \mathcal{G}$ and, if $f(\mathcal{C}) = \mathcal{C}$ for some $\mathcal{C} \in \mathcal{P}(\mathcal{S})$, then $\mathcal{C} \subseteq \mathcal{G}$.

   (b) The greatest fixed point of the function f, written *gfp* $\mathcal{S}.f(\mathcal{S})$, is defined by $\cap_{i \geq 0} f^i(S)$.

If $\mathcal{S}$ is finite and f is monotonic, the least fixed point can be derived by iteratively computing $\mathcal{S}^0 = \emptyset$, and $\mathcal{S}^{i+1} = f(\mathcal{S}^i)$ for $i \geq 0$. Eventually some iteration step yields $\mathcal{S}^i = \mathcal{S}^{i-1}$; this value is the least fixed point [61]. Similarly, starting with $\mathcal{S}^0 = \mathcal{S}$, the sequence of sets $\mathcal{S}^i$ converges to a fixed point, which is the greatest fixed point.

To illustrate the use of fixed-point calculation, consider any binary relation $R \subseteq \mathcal{D} \times \mathcal{D}$, where $\mathcal{D}$ is a finite domain. Let $I = \{(a, a) \mid a \in \mathcal{D}\}$ and define the function $f: \mathcal{P}(\mathcal{D} \times \mathcal{D}) \rightarrow \mathcal{P}(\mathcal{D} \times \mathcal{D})$ by

$$f(\mathcal{S}) = I \cup R \circ \mathcal{S}.$$

One verifies that f is monotonic. Consequently, f is continuous and has a unique least fixed point. We claim that this fixed point is the relation R^*. To see this, first note that R^* is a fixed point of f. Thus we need only show that it is the least fixed point. For suppose that R^* is not the least fixed point. Then there is an element $(x, y) \in R^*$ such that $(x, y) \notin$ *lfp* $\mathcal{S}.f(\mathcal{S})$. Note that $x \neq y$, since $I = \{(a, a) \mid a \in \mathcal{D}\} \subseteq$ *lfp* $\mathcal{S}.f(\mathcal{S})$. On the other hand, if $x \neq y$, there must exist a finite sequence $x^0, x^1, \ldots, x^k$, such that $x^0 = x$, $x^k = y$, and $(x^i, x^{i+1}) \in R$ for $0 \leq i < k$. It is a routine exercise to show, by induction on the length of the sequence, that every pair (x, y) such that y is reachable from x by a finite sequence as above is in *lfp* $\mathcal{S}.f(\mathcal{S})$. Thus, $R^* =$ *lfp* $\mathcal{S}.f(\mathcal{S})$.

For the function f above, if the cardinality of $\mathcal{D} \times \mathcal{D}$ is N, then the fixed point will be reached in at most N steps. By modifying the iteration slightly, a significantly faster convergence can be achieved. The technique is usually referred to as *iterative squaring* [28] and is based on the following definition:

$$f(\mathcal{S}) = I \cup R \cup \mathcal{S} \circ \mathcal{S}.$$

Again, it can be shown that $R^* =$ *lfp* $\mathcal{S}.f(\mathcal{S})$, and that the fixed point is now reached in at most $\lceil \log_2 N \rceil$ steps. For symbolically encoded sets, this reduction in the number of iterations can sometimes significantly speed up the fixed-point calculation.

The reflexive and transitive closure of a relation can now be computed using characteristic functions to perform relation composition and fixed-point calculations. The success of this method relies heavily on the ability to perform substitutions, quantifications, and equality checking efficiently. We shall return to other fixed-point calculations later in this chapter.

14.2.4 Behaviors

Behaviors were formally defined in Chapter 11 as 7-tuples. Since all the components of a behavior are finite, we can encode them like any other finite domain. For example, consider a behavior $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$, where X has n components and R has cardinality h. Let $k = \lceil \log_2 h \rceil$, and define a mapping $\pi: \mathcal{Q} \rightarrow \{0,1\}^{n+k}$. We can now define the characteristic function $t: \{0,1\}^{n+k} \times \{0,1\}^{n+k} \rightarrow \{0,1\}$ for the transition set as follows. For every $q, r \in \mathcal{Q}$,

$$t(\pi(q), \pi(r)) = \begin{cases} 1 & \text{if } (q,r) \in \mathcal{T}, \\ 0 & \text{otherwise.} \end{cases} \tag{14.1}$$

Note that, in general, t is not uniquely defined, since π may not be onto the set $\{0,1\}^{n+k}$. As a result, the user is free to choose any Boolean function that satisfies (14.1). If the function is represented as an OBDD, choosing a function with a compact representation is important. For an automatic procedure that often works quite well in practice, the reader is referred to [40].

In summary, the symbolic representation of a behavior is also a 7-tuple, but the internal state set is encoded as a set of binary vectors and the transition set is replaced by a Boolean function serving as the characteristic function of the set. More formally, a *symbolic behavior* is a 7-tuple

$$B = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, t, \psi \rangle,$$

where

- $X = (X_1, \ldots, X_n)$, $n \geq 0$, as in a behavior;
- $\mathcal{R} \subseteq \{0,1\}^k$ for some $k \geq 0$;
- $O = (O_1, \ldots, O_p)$, $p \geq 0$, as in a behavior;
- $\mathcal{Q} \subseteq \{0,1\}^n \times \{0,1\}^k$;
- $q_1 \in \mathcal{Q}$;
- $t: \{0,1\}^{n+k} \times \{0,1\}^{n+k} \rightarrow \{0,1\}$ is the *transition predicate*;
- $\psi : \{0,1\}^n \times \{0,1\}^k \rightarrow \{0,1\}^p$.

14.3 Deriving Symbolic Behaviors

For the symbolic representation of a behavior to be practical, we need some way of deriving it directly from a given network, race model, and environment assumptions, without having to compute the conventional behavior first. In this section we show how various types of behaviors can be derived when the underlying race model is GMW or GSW. In particular, we illustrate how to derive the unrestricted behavior and how this behavior can be restricted to yield, for example, the fundamental-mode behavior. Finally we show how a direct behavior can be derived from the fundamental-mode behavior.

To simplify the notation in the remainder of the chapter, we introduce some shorthand. First, for any Boolean values a and b, let $a \Rightarrow b$ denote the expression $\overline{a} + b$. Similarly, let $a \equiv b$ denote the expression $\overline{a \oplus b}$, and let $a \not\equiv b$ denote $\overline{a \equiv b} = a \oplus b$. Thus $\equiv$ denotes the complement of the XOR function; this is sometimes called the *equivalence* function or the *exclusive* NOR function. We extend this notation to vectors in the obvious way, i.e., for vectors $a, b \in \{0,1\}^n$, we write $a \equiv b$ and $a \not\equiv b$ instead of $\prod_{i=1}^{n}(a_i \equiv b_i)$ and $\overline{\prod_{i=1}^{n}(a_i \equiv b_i)}$, respectively. Similarly, for $a, c \in \{0,1\}^n$ and $b, d \in \{0,1\}^m$, we write $a{\cdot}b \equiv c{\cdot}d$ and $a{\cdot}b \not\equiv c{\cdot}d$ rather than $(a \equiv c) * (b \equiv d)$ and $\overline{(a \equiv c) * (b \equiv d)}$.

Assume that we are given a binary network $N = \langle \{0,1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, F \rangle$ with n input excitations and m state variables. Let $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$ denote the unrestricted behavior of N under the GMW race model. According to the definition of unrestricted behavior, we have $(a{\cdot}b, c{\cdot}d) \in \mathcal{T}$ if and only if $a{\cdot}b \neq c{\cdot}d$ and $b = d$ or $b R_a d$, where R_a is the GMW relation. We first derive the characteristic function r for the union over all a of the R_a relations as follows:

$$\begin{aligned} r(a{\cdot}b, c{\cdot}d) \quad = \quad & (a \equiv c) * \\ & ((b \not\equiv S(a{\cdot}b)) \Rightarrow (d \not\equiv b)) * \\ & \left(\prod_{i=1}^{m} ((d_i \equiv b_i) + (d_i \equiv S_i(a{\cdot}b))) \right). \end{aligned}$$

Using r, we can express the characteristic predicate t_U of the unrestricted behavior's transition set $\mathcal{T}$ as

$$t_U(a{\cdot}b, c{\cdot}d) = (a{\cdot}b \not\equiv c{\cdot}d) * ((b \equiv d) + r(a{\cdot}b, a{\cdot}d)),$$

where the first product corresponds to the constraint $a{\cdot}b \neq c{\cdot}d$, and the second product asserts that the next internal state value is either the same as the current one or is R_a-related with the current one.

Example 1

The unrestricted behavior of the network of Figure 14.3 illustrates the efficiency of the symbolic representation. Suppose we are using

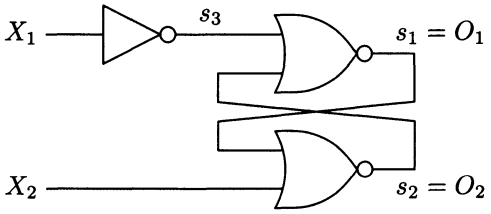


FIGURE 14.3. Network $C_{14.3}$.

the gate-state network model with excitation functions $S_1 = \overline{s_3 + s_2}$, $S_2 = \overline{X_2 + s_1}$, and $S_3 = \overline{X_1}$ and output functions $O_1 = s_1$ and $O_2 = s_2$. We use the obvious encoding of internal states, i.e., we assume that $\mathcal{Q} = \{0,1\}^5$, where the first two values correspond to the input excitations, the next two to the outputs s_1 and s_2 and the last one to the (internal) state variable s_3. After expanding the shorthand notation and performing some simplifications, we can write the characteristic predicate $t_U(a_1a_2{\cdot}b_1b_2b_3, c_1c_2{\cdot}d_1d_2d_3)$ as

$$\begin{aligned} t_U = &((a_1 \oplus c_1) + (a_2 \oplus c_2) + (b_1 \oplus d_1) + (b_2 \oplus d_2) + (b_3 \oplus d_3)) * \\ &((\overline{b_1} \oplus d_1) + (d_1 \oplus (b_2 + b_3))) * \\ &((\overline{b_2} \oplus d_2) + (d_2 \oplus (a_2 + b_1))) * \\ &((\overline{b_3} \oplus d_3) + (d_3 \oplus a_1)). \end{aligned}$$

This Boolean expression can be represented by an OBDD with 65 vertices. This should be compared with the total number, 412, of elements in $\mathcal{T}$.

Example 2

For larger circuits, the difference between an explicit representation and the implicit symbolic OBDD representation of the transition set $\mathcal{T}$ is even more striking. For example, the symbolic behavior for the unrestricted behavior of the gate-state model of the network in

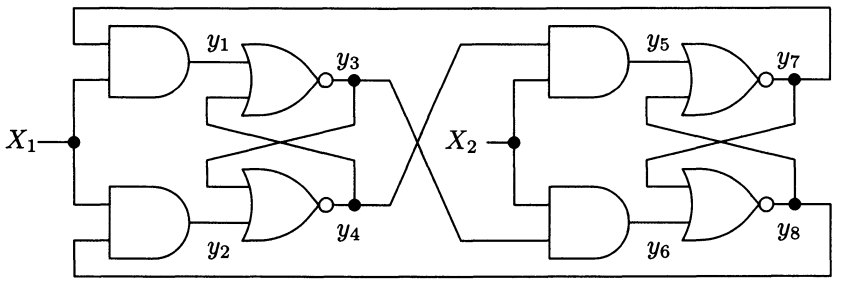


FIGURE 14.4. Network $C_{14.4}$.

Figure 14.4 requires 645 OBDD vertices representing a set containing 108,768 elements. Clearly, representing behaviors symbolically as OBDDs makes it possible to deal with much larger circuits than those that can be handled by the traditional explicit representations.

From the characteristic function for the unrestricted behavior, we can derive other types of behaviors. For example, consider finding the *unrestricted single-winner* (*USW*) behavior. The additional constraint is that at most one internal state variable can change in each transition. We can derive the USW behavior by intersecting the transition set of the unrestricted behavior with the set of transitions in which at most one internal state changes. Since the transition set is represented by its characteristic function, intersection corresponds to Boolean product. Thus we get

$$t_{\mathrm{USW}}(a{\cdot}b, c{\cdot}d) = t_U(a{\cdot}b, c{\cdot}d) * \left(\sum_{i=1}^{m} \prod_{\substack{j=1 \\ j \neq i}}^{m} (b_j \equiv d_j) \right).$$

Performing this operation on the OBDD-based transition predicates for Examples 1 and 2 above is straightforward; the resulting OBDDs have 66 and 462 vertices, respectively. In a similar way, one can derive the required operations to compute other types of behaviors, for example, those that only allow single input changes, no mixed transitions, etc.

There are two possible avenues for finding the fundamental-mode behavior of a network. The most straightforward approach is to derive that behavior directly from the network, without first deriving the unrestricted behavior. This can be accomplished by adding the constraint that, if $a{\cdot}b$ is not stable, then the input is not allowed to change. Thus

$$\begin{aligned} t_{\mathrm{FM}}(a{\cdot}b, c{\cdot}d) \;=\; & (a{\cdot}b \not\equiv c{\cdot}d) * \\ & ((b \equiv d) + r(a{\cdot}b, c{\cdot}d)) * ((a \equiv c) + (b \equiv S(a{\cdot}b))), \end{aligned}$$

where the first part ensures freedom from self-loops, the second part ensures that state variables either keep their values or change according to the R_a relation, and the final part ensures that either the inputs do not change or the state $a{\cdot}b$ is stable.

The second approach to computing the fundamental-mode behavior follows from the observation that a state of a network is stable if and only if it is a static state in the unrestricted behavior, i.e., has no internal-state transitions. Given this observation, we can compute the characteristic function for the fundamental-mode transition set as

$$t_{\mathrm{FM}}(a{\cdot}b, c{\cdot}d) = t_U(a{\cdot}b, c{\cdot}d) * \left((a \equiv c) + \overline{\exists f.\, t_U(a{\cdot}b, a{\cdot}f)} \right).$$

Note that this computation requires existential quantification over a set of Boolean variables. This is a straightforward operation with OBDDs, but

whether this added complexity is worthwhile depends on whether the unrestricted environment is also needed. If the goal is to compute only the fundamental-mode behavior, the first approach is simpler and computationally more efficient.

For some applications, the fundamental-mode behavior is unnecessarily detailed, and the direct behavior may be preferable. To find this behavior, we introduce a set of Boolean functions that allow us to write the final computation succinctly. Let r^* denote the characteristic function for the reflexive and transitive closure of the relation r defined earlier. Note that $r^*(a{\cdot}b, c{\cdot}d) = 1$ implies that $a = c$. Given r, we can compute this function using the fixed-point algorithm discussed in Section 14.2.3. Also, let r^+ denote the transitive closure of r. From r^* and r, we can find r^+ as follows:

$$r^+(a{\cdot}b, c{\cdot}d) = ((b \not\equiv d) * r^*(a{\cdot}b, c{\cdot}d)) + r(a{\cdot}b, c{\cdot}d).$$

Next, define the Boolean predicate $nontrans_i(a{\cdot}b)$ as

$$\begin{aligned} nontrans_i(a{\cdot}b) \quad = \quad & (b_i \equiv S_i(a{\cdot}b)) + \\ & \exists d.\ [(r^*(a{\cdot}b, a{\cdot}d) * r^*(a{\cdot}d, a{\cdot}b)) \\ & \qquad \Rightarrow ((d_i \not\equiv b_i) + (S_i(a{\cdot}d) \not\equiv S_i(a{\cdot}b)))]. \end{aligned}$$

This predicate can be interpreted informally as follows. State variable s_i is "nontransient" in total state $a{\cdot}b$ (formally, $nontrans_i(a{\cdot}b) = 1$) if either that variable is stable in state $a{\cdot}b$, or, if there exists a cycle, say C_i, of the R_a relation containing b, and d is any state C_i, then either d_i is different from b_i or the excitation $S_i(a{\cdot}d)$ is different from $S_i(a{\cdot}b)$. Now, if every variable is nontransient in state $a{\cdot}b$, consider the cycle C constructed by following the cycles C'_1, then $C'_2, \ldots$, then C'_m, of the individual variables, where C'_i is the empty cycle if variable s_i is stable in state $a{\cdot}b$, and $C'_i = C_i$ if there is a cycle C_i of nonzero length involving s_i, as above. Clearly, the cycle C so constructed is a nontransient cycle. Hence, $a{\cdot}b$ appears in a nontransient cycle, if and only if each of the state variables satisfies the predicate $nontrans_i(a{\cdot}b)$. Let $nontrans(a{\cdot}b) = \prod_{i=1}^{m} nontrans_i(a{\cdot}b)$. The outcome relation is now defined as

$$out(a{\cdot}b, c{\cdot}d) = \exists(e{\cdot}f).\ r^*(a{\cdot}b, e{\cdot}f) * nontrans(e{\cdot}f) * r^*(e{\cdot}f, c{\cdot}d).$$

Here, state $c{\cdot}d$ is *out*-related to state $a{\cdot}b$ if and only if $a = c$ and internal state d is in $out(R_a(b))$. Finally, define the Boolean predicates *stable* and *fresh* as

$$stable(a{\cdot}b) = (b \equiv S(a{\cdot}b)),$$

and

$$fresh(a{\cdot}b) = \exists c.\ stable(c{\cdot}b).$$

We are now ready to define a "stabilizing" predicate, i.e., the characteristic function of the set of states that only have stable states in their outcomes

$$stabilizing(a{\cdot}b) = \forall d.\ [out(a{\cdot}b, a{\cdot}d) \Rightarrow stable(a{\cdot}d)].$$

To define an O_j-hazard-free predicate, we define two relations: The first one ($unch_j$) relates states b and d if output O_j has the same value in every state in every path from b to d. The second relation ($diff$) relates b and d if d can be reached from b by an R_a relation and output O_j is different in the two states. Formally, let

$$\begin{aligned} unch_j(a{\cdot}b, c{\cdot}d) \;=\; & r^*(a{\cdot}b, c{\cdot}d) * \\ & \forall e.\,[(r^*(a{\cdot}b, a{\cdot}e) * r^*(a{\cdot}e, a{\cdot}d)) \\ & \qquad \Rightarrow (\psi_j(a{\cdot}b) \equiv \psi_j(a{\cdot}e))], \end{aligned}$$

$$diff_j(a{\cdot}b, c{\cdot}d) = r^*(a{\cdot}b, c{\cdot}d) * (\psi_j(a{\cdot}b) \not\equiv \psi_j(c{\cdot}d)),$$

$$hazfree_j(a{\cdot}b) = \forall d.\; r^*(a{\cdot}b, a{\cdot}d) \Rightarrow (unch_j(a{\cdot}b, a{\cdot}d) + diff_j(a{\cdot}b, a{\cdot}d)),$$

$$hazardfree(a{\cdot}b) = \prod_{j=1}^{p} hazfree_j(a{\cdot}b),$$

and

$$valid(a{\cdot}b) = stabilizing(a{\cdot}b) * hazardfree(a{\cdot}b).$$

We now define the transition predicate, $t_{\text{DIR}}(a{\cdot}b, c{\cdot}d)$, for the direct behavior as

$$\begin{aligned} t_{\text{DIR}}(a{\cdot}b, c{\cdot}d) \;=\; & (a{\cdot}b \not\equiv c{\cdot}d) * \\ & [(b \equiv d) * stable(a{\cdot}b) * valid(c{\cdot}d) \\ & + \\ & (a \equiv c) * fresh(a{\cdot}b) * valid(a{\cdot}b) * out(a{\cdot}b, c{\cdot}d)]\,. \end{aligned}$$

Intuitively, total states $a{\cdot}b$ and $c{\cdot}d$ are related in the direct behavior of a network if and only if either $b = d$, $a{\cdot}b$ is stable, and $c{\cdot}d$ is stabilizing and output hazard-free, or $a = c$, $a{\cdot}b$ is a fresh state that is stabilizing and output hazard-free and $d \in out(R_a(b))$.

14.4 Symbolic Race Analysis

In the previous section we showed how to derive symbolic representations of various behaviors directly from a network. A network's serial behavior can be found from its direct behavior using techniques similar to the ones used for direct behaviors. However, it is more efficient to compute the serial behavior directly, using a symbolic version of the race analysis algorithms given in Chapters 7 and 8. In this section we first describe how the symbolic serial behavior can be derived for an input-, gate-, and wire-state network using symbolic ternary simulation. We then show how the serial behavior can be computed when the network is represented by the ternary bi-bounded delay model. We accomplish this by devising a symbolic version

of the TBD algorithm. As a side effect, we also show how this algorithm can be used to determine the minimum and maximum delays of a combinational network, finally giving an algorithm for solving the problem discussed in Section 1.2.

All the race analysis algorithms that we discuss in this section are based on ternary networks. Let $N = \langle \{0,1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, F \rangle$ be a binary network, and $\mathbf{N} = \langle \{0, \Phi, 1\}, \mathcal{X}, \mathcal{S}, \mathcal{E}, \mathbf{F} \rangle$ be its ternary extension. Using the dual-rail encoding of Section 14.2.1, we find encoded versions of the excitation functions; let $S_j.1$ and $S_j.0$ denote the high and low rails corresponding to ternary excitation function $\mathbf{S}_j$.

We derive the serial behavior of a network by first finding its complete-word behavior t_{CW}. The serial behavior can then be computed as follows:

$$\begin{aligned} t_{\text{SER}}(a{\cdot}b, c{\cdot}d) \;=\; & [(b \equiv d) * \exists d'.\ t_{\text{CW}}(a{\cdot}b, c{\cdot}d')] + \\ & [(a \equiv c) * \exists a'.\ t_{\text{CW}}(a'{\cdot}b, c{\cdot}d)]. \end{aligned}$$

14.4.1 Symbolic Ternary Simulation

If $s = (s_1, \ldots, s_n)$ is a binary vector, let $\overline{s}$ denote its bit-by-bit complement, i.e., $\overline{s} = (\overline{s_1}, \ldots, \overline{s_n})$. Also, the vector $(s_1.1, \ldots, s_n.1)$ is denoted by $s.1$, and $(s_1.0, \ldots, s_n.0)$ by $s.0$.

Reformulating Algorithms A and B to the symbolic (dual-rail-encoded) domain is straightforward; we replace all ternary operations by the encoded versions, as given in Table 14.2. More precisely, let $c{\cdot}b$ be any binary total state of the network N. The symbolic version of Algorithm A is given by

Algorithm A
$h := 0;$
$s.1 := b;$
$s.0 := \overline{b};$
repeat
  $h := h + 1;$
  $s^h.1 := s^{h-1}.1 + S.1(c{\cdot}(s^{h-1}.1, s^{h-1}.0));$
  $s^h.0 := s^{h-1}.0 + S.0(c{\cdot}(s^{h-1}.1, s^{h-1}.0));$
until $(s^h.1 = s^{h-1}.1)$ and $(s^h.0 = s^{h-1}.0);$

and the symbolic version of Algorithm B is defined by

Algorithm B
$h := 0;$
$t^0.1 := s^A.1;$
$t^0.0 := s^A.0;$
repeat
  $h := h + 1;$
  $t^h.1 := S.1(c{\cdot}(t^{h-1}.1, t^{h-1}.0));$
  $t^h.0 := S.0(c{\cdot}(t^{h-1}.1, t^{h-1}.0));$
until $(t^h.1 = t^{h-1}.1)$ and $(t^h.0 = t^{h-1}.0);$

where $(s^A.1, s^A.0)$ is the dual-rail-encoded version of the fixed point reached in Algorithm A. Note that only product, sum, complement, and equality checking between Boolean functions are needed to carry out this symbolic ternary simulation algorithm. If we represent the Boolean functions by OBDDs, all these operations can be performed efficiently. Hence, we can compute the vectors $t^B.1$ and $t^B.0$ of Boolean functions representing the final result $\mathbf{t}^B$ of ternary simulation when the initial state of the network is $c{\cdot}b$.

The complete-word behavior of an input-, gate-, and wire-state network analyzed according to the GMW model can now be derived as follows:

$$\begin{aligned} t_{\mathrm{CW}}(a{\cdot}b, c{\cdot}d) \quad = \quad & (a \not\equiv c) * (b \equiv S(a{\cdot}b)) * (d \equiv S(c{\cdot}d)) * \\ & (t^B.1 \equiv d) * (t^B.0 \equiv \overline{d}), \end{aligned}$$

where $(t^B.1, t^B.0)$ is the dual-rail-encoded result of symbolic ternary simulation, given $c{\cdot}b$ as initial state. In other words, two total states are complete-word related if they differ in the input vector, they are both stable, and the second state is the result of (symbolic) ternary simulation applied to state $c{\cdot}b$. The serial behavior of a network can be computed from the complete-word behavior as described earlier.

14.4.2 Symbolic Bounded-Delay Analysis

The reformulation of standard ternary simulation into a symbolic algorithm required only that ternary operations be replaced by pairs of binary operations. Creating symbolic versions of other race analysis algorithms can be more difficult. The main problem is the dependence of the algorithms on data. For example, the TBD algorithm of Section 8.5 contains tests whether a variable is currently Φ or binary, etc. Consequently, it is not possible to use this algorithm directly in a symbolic environment. In this subsection we first restate the TBD algorithm in a format that makes it more amenable to symbolic analysis. We then show how the new formulation can be converted to symbolic notation. We assume here that there is an input delay for each input-excitation variable in the network. The reason for this assumption will become clear when we describe the transformation technique used to make the TBD algorithm amenable to symbolic manipulation.

Example 3

To illustrate the motivation behind the revised TBD (rTBD) algorithm, consider the circuit shown in Figure 14.5 with delay bounds as illustrated. If this circuit is started in total (stable) state 100·1000000, and the input excitations change to 011, we obtain a TBD analysis as shown in Figure 14.6. In Figure 14.6 we have also named each state reached in the algorithm by $\dot{\mathbf{z}}^i$, for consecutive i's. We make the following observations: First, an unstable input vertex j with delay

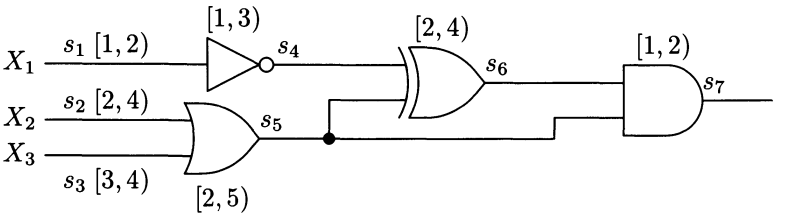


FIGURE 14.5. Circuit to illustrate rTBD algorithm.

$$
\begin{aligned}
\dot{\mathbf{z}}^0 &: \mathbf{z}^0 = \langle 1_0\,0_1\,0_1\,0\;\;0\;\;0\;\;0\;\;, (1,1,1,0,0,0,0), (1,1,1,0,0,0,0)\rangle\\
\dot{\mathbf{z}}^1 &: \tilde{\mathbf{z}}^1 = \langle 1_0\,0_1\,0_1\,0\;\;0\;\;0\;\;0\;\;, (1,1,1,0,0,0,0), (1,1,1,0,0,0,0)\rangle\\
\dot{\mathbf{z}}^2 &: \mathbf{z}^1 = \langle \Phi_0 0_1\,0_1\,0_\Phi 0\;\;0\;\;0\;\;, (0,2,2,1,0,0,0), (2,2,2,0,0,0,0)\rangle\\
\dot{\mathbf{z}}^3 &: \tilde{\mathbf{z}}^2 = \langle 0\;\;0_1\,0_1\,0_1\,0\;\;0\;\;0\;\;, (0,2,2,1,0,0,0), (0,2,2,0,0,0,0)\rangle\\
\dot{\mathbf{z}}^4 &: \mathbf{z}^2 = \langle 0\;\;\Phi_1 0_1\,\Phi_1 0_\Phi 0_\Phi 0\;\;, (0,0,3,0,1,1,0), (0,3,3,1,0,0,0)\rangle\\
\dot{\mathbf{z}}^5 &: \tilde{\mathbf{z}}^3 = \langle 0\;\;\Phi_1 0_1\,\Phi_1 0_\Phi 0_\Phi 0\;\;, (0,0,3,0,1,1,0), (0,3,3,1,0,0,0)\rangle\\
\dot{\mathbf{z}}^6 &: \mathbf{z}^3 = \langle 0\;\;\Phi_1\Phi_1\Phi_1 0_\Phi 0_\Phi 0\;\;, (0,0,0,0,2,2,0), (0,4,4,2,0,0,0)\rangle\\
\dot{\mathbf{z}}^7 &: \tilde{\mathbf{z}}^4 = \langle 0\;\;1\;\;1\;\;\Phi_1 0_1\,0_\Phi 0\;\;, (0,0,0,0,2,2,0), (0,0,0,2,0,0,0)\rangle\\
\dot{\mathbf{z}}^8 &: \mathbf{z}^4 = \langle 0\;\;1\;\;1\;\;\Phi_1\Phi_1\Phi\;\;0_\Phi, (0,0,0,0,0,0,1), (0,0,0,3,1,0,0)\rangle\\
\dot{\mathbf{z}}^9 &: \tilde{\mathbf{z}}^5 = \langle 0\;\;1\;\;1\;\;1\;\;\Phi_1\Phi\;\;0_\Phi, (0,0,0,0,0,0,1), (0,0,0,0,1,0,0)\rangle\\
\dot{\mathbf{z}}^{10} &: \mathbf{z}^5 = \langle 0\;\;1\;\;1\;\;1\;\;\Phi_1\Phi\;\;\Phi\;\;, (0,0,0,0,0,0,0), (0,0,0,0,2,0,0)\rangle\\
\dot{\mathbf{z}}^{11} &: \tilde{\mathbf{z}}^6 = \langle 0\;\;1\;\;1\;\;1\;\;\Phi_1\Phi\;\;\Phi\;\;, (0,0,0,0,0,0,0), (0,0,0,0,2,0,0)\rangle\\
\dot{\mathbf{z}}^{12} &: \mathbf{z}^6 = \langle 0\;\;1\;\;1\;\;1\;\;\Phi_1\Phi\;\;\Phi\;\;, (0,0,0,0,0,0,0), (0,0,0,0,3,0,0)\rangle\\
\dot{\mathbf{z}}^{13} &: \tilde{\mathbf{z}}^7 = \langle 0\;\;1\;\;1\;\;1\;\;\Phi_1\Phi\;\;\Phi\;\;, (0,0,0,0,0,0,0), (0,0,0,0,3,0,0)\rangle\\
\dot{\mathbf{z}}^{14} &: \mathbf{z}^7 = \langle 0\;\;1\;\;1\;\;1\;\;\Phi_1\Phi\;\;\Phi\;\;, (0,0,0,0,0,0,0), (0,0,0,0,4,0,0)\rangle\\
\dot{\mathbf{z}}^{15} &: \tilde{\mathbf{z}}^8 = \langle 0\;\;1\;\;1\;\;1\;\;\Phi_1\Phi\;\;\Phi\;\;, (0,0,0,0,0,0,0), (0,0,0,0,4,0,0)\rangle\\
\dot{\mathbf{z}}^{16} &: \mathbf{z}^8 = \langle 0\;\;1\;\;1\;\;1\;\;\Phi_1\Phi\;\;\Phi\;\;, (0,0,0,0,0,0,0), (0,0,0,0,5,0,0)\rangle\\
\dot{\mathbf{z}}^{17} &: \tilde{\mathbf{z}}^9 = \langle 0\;\;1\;\;1\;\;1\;\;1\;\;\Phi_0\Phi\;\;, (0,0,0,0,0,0,0), (0,0,0,0,0,0,0)\rangle\\
\dot{\mathbf{z}}^{18} &: \mathbf{z}^9 = \langle 0\;\;1\;\;1\;\;1\;\;1\;\;\Phi_0\Phi\;\;, (0,0,0,0,0,0,0), (0,0,0,0,0,1,0)\rangle\\
\dot{\mathbf{z}}^{19} &: \tilde{\mathbf{z}}^{10} = \langle 0\;\;1\;\;1\;\;1\;\;1\;\;\Phi_0\Phi\;\;, (0,0,0,0,0,0,0), (0,0,0,0,0,1,0)\rangle\\
\dot{\mathbf{z}}^{20} &: \mathbf{z}^{10} = \langle 0\;\;1\;\;1\;\;1\;\;1\;\;\Phi_0\Phi\;\;, (0,0,0,0,0,0,0), (0,0,0,0,0,2,0)\rangle\\
\dot{\mathbf{z}}^{21} &: \tilde{\mathbf{z}}^{11} = \langle 0\;\;1\;\;1\;\;1\;\;1\;\;\Phi_0\Phi\;\;, (0,0,0,0,0,0,0), (0,0,0,0,0,2,0)\rangle\\
\dot{\mathbf{z}}^{22} &: \mathbf{z}^{11} = \langle 0\;\;1\;\;1\;\;1\;\;1\;\;\Phi_0\Phi\;\;, (0,0,0,0,0,0,0), (0,0,0,0,0,3,0)\rangle\\
\dot{\mathbf{z}}^{23} &: \tilde{\mathbf{z}}^{12} = \langle 0\;\;1\;\;1\;\;1\;\;1\;\;\Phi_0\Phi\;\;, (0,0,0,0,0,0,0), (0,0,0,0,0,3,0)\rangle\\
\dot{\mathbf{z}}^{24} &: \mathbf{z}^{12} = \langle 0\;\;1\;\;1\;\;1\;\;1\;\;\Phi_0\Phi\;\;, (0,0,0,0,0,0,0), (0,0,0,0,0,4,0)\rangle\\
\dot{\mathbf{z}}^{25} &: \tilde{\mathbf{z}}^{13} = \langle 0\;\;1\;\;1\;\;1\;\;1\;\;0\;\;\Phi_0, (0,0,0,0,0,0,0), (0,0,0,0,0,0,0)\rangle\\
\dot{\mathbf{z}}^{26} &: \mathbf{z}^{13} = \langle 0\;\;1\;\;1\;\;1\;\;1\;\;0\;\;\Phi_0, (0,0,0,0,0,0,0), (0,0,0,0,0,0,1)\rangle\\
\dot{\mathbf{z}}^{27} &: \tilde{\mathbf{z}}^{14} = \langle 0\;\;1\;\;1\;\;1\;\;1\;\;0\;\;\Phi_0, (0,0,0,0,0,0,0), (0,0,0,0,0,0,1)\rangle\\
\dot{\mathbf{z}}^{28} &: \mathbf{z}^{14} = \langle 0\;\;1\;\;1\;\;1\;\;1\;\;0\;\;\Phi_0, (0,0,0,0,0,0,0), (0,0,0,0,0,0,2)\rangle\\
\dot{\mathbf{z}}^{29} &: \tilde{\mathbf{z}}^{15} = \langle 0\;\;1\;\;1\;\;1\;\;1\;\;0\;\;0\;\;, (0,0,0,0,0,0,0), (0,0,0,0,0,0,0)\rangle\\
\dot{\mathbf{z}}^{30} &: \mathbf{z}^{15} = \langle 0\;\;1\;\;1\;\;1\;\;1\;\;0\;\;0\;\;, (0,0,0,0,0,0,0), (0,0,0,0,0,0,0)\rangle
\end{aligned}
$$

FIGURE 14.6. Example of TBD analysis.

bound $[d_j, D_j)$ changes to Φ in state $\dot{\mathbf{z}}^{2d_j}$ and to its (binary) excitation in state $\dot{\mathbf{z}}^{2D_j-1}$. A state vertex j with delay bound $[d_j, D_j)$ that becomes unstable in state $\dot{\mathbf{z}}^{2k}$, for some k, and stays unstable until at least state $\dot{\mathbf{z}}^{2k+2d_j-1}$, changes to Φ in state $\dot{\mathbf{z}}^{2k+2d_j}$. Also, if a state vertex j is unstable and gets a binary excitation in state $\dot{\mathbf{z}}^{2k-1}$, for some $k > 0$, and the excitation remains at this value for the next $2D_j - 1$ states, then the vertex changes to this excitation in state $\dot{\mathbf{z}}^{2k-1+2D_j}$. Given the behavior of the input vertices, it is easy to see that vertices only change to Φ in even states and to binary values in odd states. Thus, one can view the input vertices as creating a "change-to-Φ wave" and a "change-to-binary wave." The first wave propagates on even time steps; the second on odd time steps. Note that the "speed" of the Φ-wave is strictly greater than the speed of the binary wave.

The example above provides an intuitive motivation for the revised TBD algorithm, called the *rTBD* algorithm for short. The rTBD algorithm also uses two vectors of counters, in addition to the current state of the network, to keep track of how long a vertex has been unstable and how long it has had a binary excitation. However, the algorithm uses the odd/even changes instead of computing intermediate states, as was done in the original TBD algorithm. More formally, the rTBD algorithm is defined inductively as follows:

Basis: $\langle \dot{\mathbf{z}}^0, \dot{U}^0, \dot{V}^0 \rangle = \langle b, (0, \ldots, 0), (0, \ldots, 0) \rangle$

Induction Step: Given $\langle \dot{\mathbf{z}}^h, \dot{U}^h, \dot{V}^h \rangle$, state $\langle \dot{\mathbf{z}}^{h+1}, \dot{U}^{h+1}, \dot{V}^{h+1} \rangle$ is computed as follows:

$$\dot{U}_j^{h+1} = \begin{cases} \dot{U}_j^h + 1 & \text{if } s_j \in \mathcal{U}(a, \dot{\mathbf{z}}^h) \cap \mathcal{B}(\dot{\mathbf{z}}^h), \\ 0 & \text{otherwise;} \end{cases}$$

$$\dot{V}_j^{h+1} = \begin{cases} \dot{V}_j^h + 1 & \text{if } s_j \in \mathcal{U}(a, \dot{\mathbf{z}}^h) \cap \mathcal{BE}(a, \dot{\mathbf{z}}^h), \\ 0 & \text{otherwise;} \end{cases}$$

$$\dot{\mathbf{z}}_j^{h+1} = \begin{cases} S_j(a, \dot{\mathbf{z}}^h) & \text{if } 1 \le j \le n \text{ and } \dot{V}_j^{h+1} = 2D_j - 1, \\ S_j(a, \dot{\mathbf{z}}^h) & \text{if } n < j \le n+m \text{ and} \dot{V}_j^{h+1} = 2D_j, \\ \Phi & \text{if } \dot{U}_j^{h+1} = 2d_j, \\ \dot{\mathbf{z}}_j^h & \text{otherwise.} \end{cases}$$

The following theorem shows that this algorithm produces the same results as the TBD algorithm of Chapter 8.

Theorem 14.1 *Suppose that* $(\langle \mathbf{z}^0, U^0, V^0 \rangle, \langle \tilde{\mathbf{z}}^1, \tilde{U}^1, \tilde{V}^1 \rangle, \langle \mathbf{z}^1, U^1, V^1 \rangle, \ldots)$ *and* $(\langle \dot{\mathbf{z}}^0, \dot{U}^0, \dot{V}^0 \rangle, \langle \dot{\mathbf{z}}^1, \dot{U}^1, \dot{V}^1 \rangle, \langle \dot{\mathbf{z}}^2, \dot{U}^2, \dot{V}^2 \rangle, \ldots)$ *are the sequences of states computed by the TBD and the rTBD algorithms, respectively. Then* $\dot{\mathbf{z}}^{2i-1} = \tilde{\mathbf{z}}^i$ *for* $i = 1, 2, \ldots$, *and* $\dot{\mathbf{z}}^{2i} = \mathbf{z}^i$ *for* $i = 0, 1, \ldots$.

Proof: The proof of this result is rather technical and too lengthy to be included here. We refer the reader to [125]. □

Consider now the rTBD algorithm when the ternary values and excitation functions are represented in the dual-rail encoding of Table 14.1. In [122] it was shown that a vertex can change only from a binary value to Φ or from Φ to a binary value in the TBD algorithm. Furthermore, except for the input vertices, the excitation of a vertex can only change from a binary value to Φ or from Φ to a binary value. In view of Theorem 14.1, the same statements hold for the rTBD algorithm. The signal transitions possible in the rTBD algorithm are shown in Table 14.4, together with the corresponding encoded transitions.

TABLE 14.4. Possible signal transitions in the rTBD algorithm.

Ternary values	Encoded values
$0 \rightarrow \Phi$	$(0,1) \rightarrow (1,1)$
$1 \rightarrow \Phi$	$(1,0) \rightarrow (1,1)$
$\Phi \rightarrow 0$	$(1,1) \rightarrow (0,1)$
$\Phi \rightarrow 1$	$(1,1) \rightarrow (1,0)$

Note that changing from a binary value to Φ requires changing one of the rails from 0 to 1, and changing from Φ to a binary value requires changing one of the rails from 1 to 0. This observation can be used to derive a "dual-rail version" of the rTBD algorithm. In Figure 14.7 we show graphically

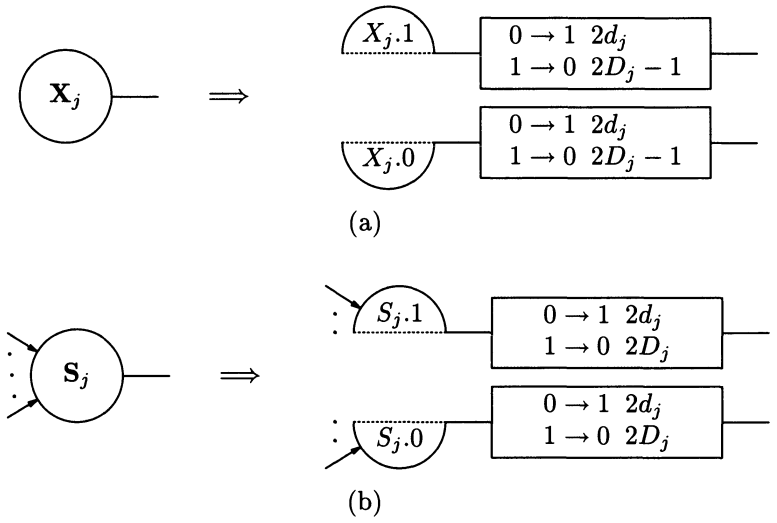


FIGURE 14.7. Dual-rail rTBD algorithm: (a) input vertex; (b) state vertex.

how such a transformation can be performed. Each vertex in the TBD algorithm is split into two vertices, each with a "delay box" showing the delays associated with changing the vertex variable from 0 to 1 and from 1 to 0. Note that the input vertices are treated somewhat differently from the state vertices.

The restatement of the rTBD algorithm transforms the original TBD algorithm into two simpler binary-delay modeling problems. However, it does not solve the data dependence problem—the delay boxes must still test for specific values. What is needed is some way of modifying the delay boxes of Figure 14.7 so that they need not test explicitly for 0's or 1's. To introduce the technique of symbolic TBD analysis, we derive a delay box for the case where the $0 \rightarrow 1$ transition should take two steps, and the $1 \rightarrow 0$ transition should take six steps. To aid the reader's intuition, we develop the ideas in terms of a "circuit" that is analyzed using a unit-delay race algorithm. The basic construction can be easily generalized and formalized.

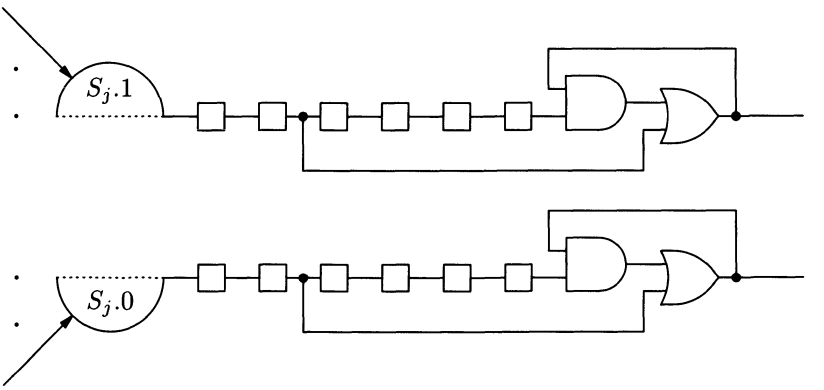


FIGURE 14.8. First attempt to design a delay circuit.

In Figure 14.8 we show two small "delay circuits" that almost achieve the desired data independence. Since the high-rail and the low-rail delay circuits are identical, we discuss only one of them. The delay circuit consists of a series of unit-delay elements, an AND gate, and an OR gate. Note that we assume that the evaluations of the excitation function and the AND and OR gates are instantaneous (i.e., have zero delay). Thus, we are really performing a "unit/zero-delay analysis." Assume that the delay circuit is started in a stable state, i.e., all element and gate outputs are 0 (1). If the excitation function changes to 1 (0), the output will change to its new value after 2 (6) unit delays. In other words, if changes of the excitation occur so seldom that the delay circuits are always stable when the change occurs, the delay circuit delays a $0 \rightarrow 1$ transition two steps, and a $1 \rightarrow 0$ transition six steps—exactly as required.

The delay circuit of Figure 14.8 works correctly as long as changes of the excitation do not occur too often. However, in a real circuit this cannot always be guaranteed. Consider for example what would happen if the excitation had been 0 for a long time and then changed to 1 at time t and back to 0 again at time $t + 1$. This "glitch" propagates unchanged through the delay circuit and appears on the output of the delay circuit at time $t + 3$. On the other hand, in the rTBD algorithm, such a pulse would be suppressed. What is needed is a delay circuit that can exhibit inertia, i.e., that ignores such short pulses. Since pulses in the rTBD algorithm are always of integer length, it is sufficient to find a circuit that can filter out glitches of integer length. In Figure 14.9 we show such a delay circuit. Again, it is a chain of unit delays and some delay-free AND gates and OR gates. For the output of this delay circuit to change from 0 to 1, the excitation must be 1 for at least two consecutive steps. Similarly, for the output to change from 1 to 0, the excitation function must be 0 for at least six consecutive steps. Hence, the delay circuit of Figure 14.9 exhibits the desired inertia. Furthermore, it does not contain any data-dependent tests, and requires only standard unit-/zero-delay simulation.

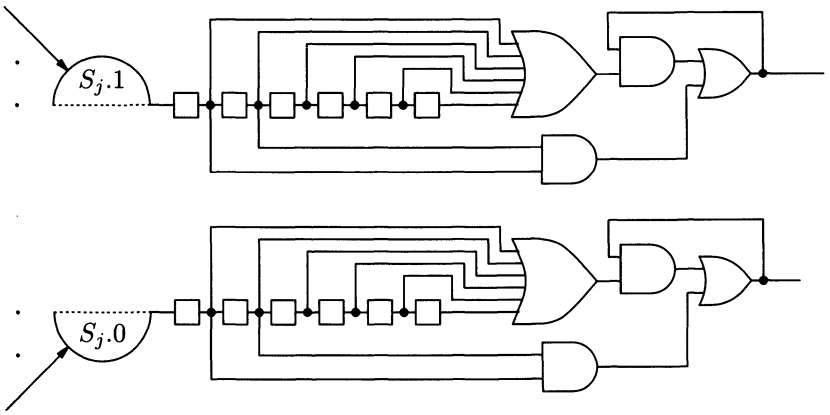


FIGURE 14.9. Final version of delay circuit.

In summary, by reformulating the TBD algorithm and using the dual-rail encoding of Table 14.1 for ternary values, we have shown how a ternary network can be transformed into a binary network in such a way that the unit-delay analysis of the transformed network corresponds exactly to the TBD analysis of the original network. The same transformation technique can be used for other delay models. First of all, it is straightforward to design a delay circuit that delays a $0 \rightarrow 1$ transition e time units and a $1 \rightarrow 0$ transition E time units (where e is not necessarily less than E), and exhibits inertia. Now assume the high rail has a $0 \rightarrow 1$ delay equal to r (for minimum rise delay) and a $1 \rightarrow 0$ delay equal to F (for maximum

fall delay), and the low rail has a $0 \rightarrow 1$ delay equal to f (for minimum fall delay) and a $1 \rightarrow 0$ delay equal to R (for maximum rise delay). Note that we must have $r \leq R$ and $f \leq F$ for proper operation. Otherwise the output of the vertex could possibly get the value 0 on both the low and the high rails—an illegal situation. In Table 14.5 we show the different race models we obtain by using different values of the delay bounds. Note that the nominal delay (ND) model can be simplified by using only half as many unit-delay elements. However, by using the numbers shown, it is possible to have nominal delays on certain vertices and bounded delays on others.

TABLE 14.5. Race models obtained for different delay circuits.

Input vertices	State vertices	Corresponding race model
$r = f = 2d_j$ $R = F = 2D_j - 1$	$r = f = 2d_j$ $R = F = 2D_j$	TBD analysis with delays $d_jT \leq \Delta_j(t) < D_jT$
$r = R = 2r_j$ $f = F = 2f_j$	$r = R = 2r_j$ $f = F = 2f_j$	ND analysis with rise delays r_jT and fall delays f_jT
$r = 2r_j$ $R = 2R_j - 1$ $f = 2f_j$ $F = 2F_j - 1$	$r = 2r_j$ $R = 2R_j$ $f = 2f_j$ $F = 2F_j$	TBD analysis with rise delays $r_jT \leq \Delta_j^r(t) < R_jT$ and fall delays $f_jT \leq \Delta_j^f(t) < F_jT$

With the ability to perform bi-bounded-delay and nominal-delay race analysis symbolically, it is easy to derive the serial behavior of a network. There is one difficulty remaining. In the nominal-delay and TBD algorithms, the only guarantee we can give, in general, on the maximum length of any cycle in the outcome is that it contains fewer than $2^m D$ ($3^m D$) states, where D is the maximum delay associated with any vertex. This implies that the symbolic versions of these algorithms may have to be carried out for exponentially many steps in order to compute the outcome of a transition. A pragmatic solution is to impose an upper limit on the number of steps the circuit is allowed to take to reach a stable state. Hence, we can compute a vector of Boolean functions representing the final result of symbolic TBD (or ND) simulation when the initial state of the network is $\hat{a}{\cdot}b$ and the input is changed to a. Consequently, the computation of the complete-word behavior of a network can be performed as follows:

$$\begin{aligned} t_{\mathrm{CW}}(a{\cdot}b, c{\cdot}d) \;=\; & (a \not\equiv c) * (b \equiv S(a{\cdot}b)) * (d \equiv S(c{\cdot}d)) * \\ & (r.1 \equiv d) * (r.0 \equiv \overline{d}\,), \end{aligned}$$

where $(r.1, r.0)$ represents the final result of symbolic TBD (or ND) simulation, depending on the delay/race model used. Intuitively, two total states are complete-word-related if they are both stable, they differ in their inputs, and the second state is the result obtained by a (symbolic) race analysis algorithm. From the symbolic complete-word behavior we can derive the serial behavior as discussed earlier.

We have demonstrated how nominal-delay and bi-bounded-delay race analysis can be performed by carrying out a unit-delay analysis of a transformed network. The transformation of the original network consisted of two steps. First a binary dual-rail-encoded network was obtained from the ternary network. Next, each input- and next-state vertex in the dual-rail network was replaced by a delay circuit that was determined by the race model and the delay associated with the vertex. By using this transformational technique, we were able to carry out these race analysis algorithms symbolically. One important side effect of this is that we can use symbolic-unit-delay analysis to compute the minimum and maximum delays a network may exhibit for any input change. In particular, this is a powerful technique for determining the minimum and maximum propagation delays in a combinational network. We begin by applying the (fully symbolic) input vector a. We then simulate the (transformed) network until it reaches a stable state. Since the network is combinational, the circuit is guaranteed to reach such a state. At this point, we apply a completely new set of (symbolic) input values and start simulating. For each step of the (unit-delay) simulation, we also compute the XOR of the old and new values of the output functions. Given the results above, one verifies that the first time this Boolean function differs from $|0|$ determines the minimum delay for any input change, and the last time the function differs from $|0|$ determines the maximum delay for any input change. In essence, by running two symbolic unit-delay simulations, we obtain the same amount of information as we would get had we simulated the original network using a nominal or bi-bounded race analysis algorithm for every possible input change. Moreover, symbolic simulation can often be done in a tiny fraction of the time that such an exhaustive simulation would require.

14.5 Symbolic Verification of Realization

In the previous sections we have shown how to represent specification behaviors in symbolic form, and we have derived symbolic representations of several implementation behaviors from a given network, environmental assumptions, and race model. We now consider algorithms for determining whether a (symbolic) specification behavior is realized by a (symbolic) implementation behavior.

In Chapter 11 five conditions were given for an implementation behavior B' to realize a specification behavior A. Here we assume that the first condition has been satisfied, i.e., that the initial mapping and removal of unused inputs and outputs have been done. The reason for this is our purely pragmatic aim to simplify the notation. Thus, following Section 11.3, we assume that the input excitation and output vectors of B' are identical to those of A, and we focus on the remaining four conditions. Unfortunately, these conditions include both language containment properties and

structural properties of the behaviors' graphs; thus, a direct adaptation to a symbolic domain presents some problems. In particular, the language containment conditions are difficult to translate into efficient symbolic algorithms, especially when the implementation behavior is nondeterministic. Consequently, we first develop a new set of conditions that are amenable to symbolic analysis and that imply the original conditions. Unfortunately, our decision procedure is not perfect: there are cases where an implementation realizes a specification, but our decision procedure fails to discover this.

Let $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$ be a deterministic (and hence proper) specification behavior, and let $B' = \langle X, \mathcal{R}', O, \mathcal{Q}', q_1', \mathcal{T}', \psi' \rangle$ denote an input-proper implementation behavior.[3]

Recall that an empty-word path (or ε-path) in B' is a sequence $s_0', s_1', \ldots, s_k'$ of states such that $(s_i', s_{i+1}') \in \mathcal{T}'$ for $0 \leq i < k$, and $l'(s_i') = l'(s_0')$, for $0 < i \leq k$. In other words, such a path "spells" the empty word ε. For any states $s', t' \in \mathcal{Q}'$, define the predicate $\varepsilon\text{-}path(s', t')$ to hold if and only if there is an ε-path from s' to t'. Let $\mathcal{T}'_{\varepsilon^*}$ denote the transition set obtained by adding transitions that "bypass" empty-word transitions in $\mathcal{T}'$, i.e., let

$$\mathcal{T}'_{\varepsilon^*} = \mathcal{T}' \cup \{(q', r') \mid \exists s'.\ \varepsilon\text{-}path(q', s') \text{ and } (s', r') \in \mathcal{T}'\ \}.$$

Finally, let $\mathcal{T}'_{\varepsilon\text{-free}}$ denote the set obtained from $\mathcal{T}'_{\varepsilon^*}$ by removing all empty-word transitions, i.e., let

$$\mathcal{T}'_{\varepsilon\text{-free}} = \{(q', r') \mid l'(q') \neq l'(r') \text{ and } (q', r') \in \mathcal{T}'_{\varepsilon^*}\}.$$

The *ε-free behavior* $B'_{\varepsilon\text{-free}}$ of an implementation behavior B' is identical to B' except the transition set $\mathcal{T}'$ is replaced by $\mathcal{T}'_{\varepsilon\text{-free}}$. The following proposition follows from the construction of $\mathcal{T}'_{\varepsilon\text{-free}}$.

Proposition 14.1 $L(B') = L(B'_{\varepsilon\text{-free}})$.

The ε-free behavior $B'_{\varepsilon\text{-free}}$ is more convenient than B' for language properties, but does not preserve the deadlock and livelock characteristics of B'. Consequently, we need to keep track of these characteristics separately. We do this by defining a set, called *trap'*, that contains all the states in B' that can reach, through a sequence of empty-word transitions, either an empty-word cycle or a terminal state. Intuitively, a *trap state* is a state from which the implementation might never produce an output, unless an input change is applied. Formally, let *trap'* be defined as

$$\{q' \mid \exists r'.\ (q', r') \in \mathcal{T}'_{\varepsilon^*}, l'(q') = l'(r'),\ \text{and } (r', r') \in \mathcal{T}'_{\varepsilon^*} \text{ or } r' \in term'\},$$

[3] Determining that a symbolic behavior is (input-)proper can be done using techniques similar to those of this section. However, the algorithms that we normally use to derive the symbolic behaviors from networks guarantee that the behaviors already have the desired property.

where $term'$ is the set of terminal states of B'. One verifies that the terminal states of B' are the same as those of $B'_{\varepsilon\text{-free}}$. In view of the fact that $B'_{\varepsilon\text{-free}}$ has no empty-word transitions, the set $term'$ can be found more conveniently from $\mathcal{T}'_{\varepsilon\text{-free}}$ as follows:

$$term' = \{q' \mid \forall p'.\ (q', p') \in \mathcal{T}'_{\varepsilon\text{-free}} \text{ implies } X(q') \neq X(p')\}.$$

The basic idea behind the decision procedure that tests for realization is to find a binary relation between the states in the specification behavior and the states in the implementation behavior. Informally, this relation identifies "equivalent" states in the sense that words that are relevant to the specification take the two behaviors to related states, and related states have similar livelock and deadlock behaviors. More formally, we say that binary relation $M \subseteq \mathcal{Q} \times \mathcal{Q}'$ is a *state-realization relation* if and only if

1. $q_1 M q'_1$.
   (Initial states correspond.)

2. If qMq', then $l(q) = l'(q')$.
   (Corresponding states have the same labels.)

3. If $(q, r) \in \mathcal{T}$ and qMq', then there exists $r' \in \mathcal{Q}'$ such that $(q', r') \in \mathcal{T}'_{\varepsilon\text{-free}}$ and rMr'.
   (For each transition in the specification there is a corresponding transition in the implementation.)

4. Suppose that $(q', r') \in \mathcal{T}'_{\varepsilon\text{-free}}$, qMq', and either (a) $X(q') \neq X(r')$, and there exists a state $\tilde{r} \in \mathcal{Q}$ such that $(q, \tilde{r}) \in \mathcal{T}$ and $X(\tilde{r}) = X(r')$, or (b) $X(q') = X(r')$. Then there exists $r \in \mathcal{Q}$, such that $(q, r) \in \mathcal{T}$ and rMr'.
   (For each transition of the implementation that is relevant to the specification there is a corresponding transition in the specification.)

5. If $q' \in trap'$, and qMq', then q must be a terminal state.
   (Implementation states that might never produce any outputs can only be related to specification states that are terminal.)

Lemma 14.1 *Assume that M is a state-realization relation between A and B' and that $w \in L(A) \cap L(B'_{\varepsilon\text{-free}})$. If $q \in \mathcal{Q}$ is the state reachable by w from q_1 in A and $q' \in \mathcal{Q}'$ is any state reachable by w from q'_1 in $B'_{\varepsilon\text{-free}}$, then qMq'.*

Proof: We prove the claim by induction on the length of the word w. The basis, $|w| = 0$, follows trivially from Property 1 of the state-realization relation, since, by construction of $B'_{\varepsilon\text{-free}}$, the only state reachable from q'_1 by the empty word is q'_1. Assume now that the claim holds for all words of length less than n for some $n \geq 1$. Consider a word $w = u\sigma \in L(A) \cap L(B'_{\varepsilon\text{-free}})$ such that $|w| = n$. Since $w \in L(A)$, it follows that there must exist a

state sequence $q_1, \ldots, q_{n+1} \in \mathcal{Q}$ that spells w. Note that the sequence $q_1, \ldots, q_n$ spells u. Now consider any state q'_{n+1} reachable by w in $B'_{\varepsilon\text{-free}}$. Since $w = u\sigma$, there exists a state q'_n reachable by u in $B'_{\varepsilon\text{-free}}$ such that $(q'_n, q'_{n+1}) \in \mathcal{T}'_{\varepsilon\text{-free}}$ and $\tau(q'_n, q'_{n+1}) = \sigma$. By the induction hypothesis, $q_n M q'_n$; by Property 2 of the state-realization relation, $l(q_n) = l'(q'_n)$. Since $(q_n, q_{n+1}) \in \mathcal{T}$ and $\tau(q_n, q_{n+1}) = \sigma$, it follows that $l(q_{n+1}) = l'(q'_{n+1})$. By Property 4 of the state-realization relation, there must exist a state $r \in \mathcal{Q}$ such that $(q_n, r) \in \mathcal{T}$ and $r M q'_{n+1}$. Because A is deterministic, r must be q_{n+1}. Altogether, we have shown that $q_{n+1} M q'_{n+1}$. Thus the induction goes through and the claim follows. □

Lemma 14.2 *If M is a state-realization relation between A and B', then B' realizes A.*

Proof: Assume that M is a state-realization relation between A and B'. We need to verify that

i. $L(B'/A) \subseteq L(A)$ (safety).

ii. $L(A) \subseteq L(B'/A)$ (capability).

iii. If $w \in L(B') \cap L(A)$ leads to a terminal state in B', then it also leads to a terminal state in A (deadlock-freedom).

iv. If $w \in L(B') \cap L(A)$, leads to a state in B' that has a cycle spelling ε around it, then w leads to a terminal state in A (livelock-freedom).

By Proposition 14.1 we know that $L(B') = L(B'_{\varepsilon\text{-free}})$. From the definition of relevant words we have $L(B'/A) = L(B'_{\varepsilon\text{-free}}/A)$. Hence, for the language containment properties (i) and (ii), we can substitute $B'_{\varepsilon\text{-free}}$ for B'.

We prove (i) by showing, by induction on the length of w, that $w \in L(B'_{\varepsilon\text{-free}}/A)$ implies $w \in L(A)$. If $|w| = 0$, i.e., $w = \varepsilon$, then $w \in L(A)$, because the empty word is accepted by every specification behavior. Now assume inductively that, for some $n \geq 0$, we have $u \in L(B'_{\varepsilon\text{-free}}/A)$ and $|u| \leq n$ implies that $u \in L(A)$. Consider any word $w = u\sigma \in L(B'_{\varepsilon\text{-free}}/A)$ such that $|u| = n$ and $\sigma \in \Sigma$. Since $w \in L(B'_{\varepsilon\text{-free}}/A)$, we can find states q' and $r' \in \mathcal{Q}'$ such that q' is reachable from q'_1 by u, $(q', r') \in \mathcal{T}'_{\varepsilon\text{-free}}$, and $\tau(q', r') = \sigma$. We now claim that there are corresponding states q and $r \in \mathcal{Q}$ such that q is reachable from the initial state q_1 by the word u, $(q, r) \in \mathcal{T}$, and $\tau(q, r) = \sigma$. By the induction hypothesis, it follows that $u \in L(A)$; let $q \in \mathcal{Q}$ be the state reachable from q_1 by u. Because $u \in L(A) \cap L(B'_{\varepsilon\text{-free}}/A) = L(A) \cap L(B'_{\varepsilon\text{-free}})$, it follows by Lemma 14.1 that $q M q'$. Since w is relevant to A, either $\sigma \cap \mathcal{X} = \emptyset$ or σ is applicable in A after u. In either case, Property 4 of the state-realization relation guarantees that there is a state $r \in \mathcal{Q}$ such that $(q, r) \in \mathcal{T}$ and $r M r'$. By Property 2 it follows that $l(r) = l(r')$ and $\tau(q, r) = \tau(q', r') = \sigma$. Altogether, we have shown that $w \in L(A)$; thus the induction step goes through and the claim follows.

Claim (ii) follows by an inductive argument on the length of a word w in $L(A)$ and by Properties 1, 2, and 3 of the state-realization relation. Finally, (iii) and (iv) follow from Property 5 of the state-realization relation and from the definition of *trap'*. □

We now turn our attention to determining whether a state-realization relation exists. Our approach is to compute the most general relation satisfying Properties 2–5, and then determine whether this relation also satisfies Property 1. This most general relation is found by a fixed-point calculation. Intuitively, we compute a sequence of approximations to the relation, refining each approximation until we reach the most general relation satisfying Properties 2–5.

Formally, suppose we are given a deterministic specification behavior $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$ and an input-proper implementation behavior $B' = \langle X, \mathcal{R}', O, \mathcal{Q}', q_1', \mathcal{T}', \psi' \rangle$. We start with the set $\mathcal{M}_0 = \mathcal{Q} \times \mathcal{Q}'$. This includes all possible pairs of states that are candidates to be related by a state-realization relation. To ensure that corresponding states have the same label, we define the set

$$label = \{(q, q') \mid q \in \mathcal{Q}, q' \in \mathcal{Q}', \text{ and } l(q) = l'(q')\}.$$

To enforce Property 5 of a state-realization relation we define

$$live = \{(q, q') \mid q \in \mathcal{Q}, q' \in \mathcal{Q}', \text{ and } q' \in trap' \text{ implies } q \in term\}.$$

The next two definitions deal with the language properties of capability and safety. For any set $\mathcal{M} \subseteq \mathcal{Q} \times \mathcal{Q}'$, let

$$cap(\mathcal{M}) = \{(q, q') \mid q \in \mathcal{Q}, q' \in \mathcal{Q}', \text{ and } \forall r \in \mathcal{Q}. [(q, r) \in \mathcal{T} \text{ implies } \exists r' \in \mathcal{Q}'. [(q', r') \in \mathcal{T}'_{\varepsilon\text{-free}} \text{ and } (r, r') \in \mathcal{M}]]\}$$

and

$$safe(\mathcal{M}) = \{(q, q') \mid q \in \mathcal{Q}, q' \in \mathcal{Q}', \text{ and } \forall r' \in \mathcal{Q}'. [(q', r') \in \mathcal{T}'_{\varepsilon\text{-free}} \text{ and } (q, q', r') \in appl \text{ implies } \exists r \in \mathcal{Q}. [(q, r) \in \mathcal{T} \text{ and } (r, r') \in \mathcal{M}]]\},$$

where *appl* is the set

$$\{(q, q', r') \mid X(q') = X(r') \text{ or } \exists \tilde{r} \in \mathcal{Q}. [(q, \tilde{r}) \in \mathcal{T} \text{ and } X(\tilde{r}) = X(r')]\}.$$

Now define

$$f(\mathcal{M}) = label \cap cap(\mathcal{M}) \cap safe(\mathcal{M}) \cap live.$$

Note that the sets in the expression above correspond directly to Properties 2–5 of the state-realization relation. Consequently, the following proposition holds:

Proposition 14.2 *If $\mathcal{M}$ is a solution to the equation $\mathcal{M} = f(\mathcal{M})$ and $(q_1, q_1') \in \mathcal{M}$, then $\mathcal{M}$ is a state-realization relation.*

It is easily verified that $f(\mathcal{M})$ is monotonic; thus the greatest fixed-point of the equation $\mathcal{M} = f(\mathcal{M})$ is well defined. Altogether, we get the main result of this section:

Theorem 14.2 *Let* $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$ *be a proper specification behavior, let* $B = \langle X, \mathcal{R}', O, \mathcal{Q}', q_1', \mathcal{T}', \psi' \rangle$ *an input-proper implementation behavior, and let* $\widehat{\mathcal{M}} = gfp\ \mathcal{M}.\, f(\mathcal{M})$. *If* $(q_1, q_1') \in \widehat{\mathcal{M}}$, *then* B' *realizes* A.

Proof: Assume $(q_1, q_1') \in \widehat{\mathcal{M}}$. Since $\widehat{\mathcal{M}} = gfp\ \mathcal{M}.\, f(\mathcal{M})$, it follows trivially that $\widehat{\mathcal{M}}$ is a solution to $\mathcal{M} = f(\mathcal{M})$. By Proposition 14.2, $\widehat{\mathcal{M}}$ is a state-realization relation. Hence Lemma 14.2 applies and the claim follows. □

The theorem above provides a straightforward fixed-point algorithm for determining whether a behavior is a realization of another behavior. However, it should be noted that the condition of the theorem is only a sufficient condition. There are behaviors related by the realization relation for which this procedure fails, since we require each specification state to correspond to some implementation state. For example, there is no state-realization relation between the behavior of Figure 11.5 (a) and the behavior of Figure 11.5 (b). Consequently, our decision procedure would incorrectly claim that the behavior of Figure 11.5 (b) is not a realization of the behavior of Figure 11.5 (a). It is difficult to guess how common this type of state splitting is in practice; thus the degree of applicability of our algorithm is still unknown.

We now turn our attention to a symbolic version of the fixed-point algorithm above. We denote all the sets and relations needed in the algorithm by their characteristic functions. Assume that $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, t, \psi \rangle$ is a deterministic symbolic specification behavior and that $B' = \langle X, \mathcal{R}', O, \mathcal{Q}', q_1', t', \psi' \rangle$ is an input-proper symbolic implementation behavior. We introduce some shorthand to improve the notation. First, we write $l(a{\cdot}b) \equiv l(c{\cdot}d)$ to denote the expression $(a \equiv c) * (\psi(a{\cdot}b) \equiv \psi(c{\cdot}d))$. We also write $l'(a'{\cdot}b') \equiv l'(c'{\cdot}d')$ to denote the corresponding expression in which ψ is replaced by ψ'. We write $X(a{\cdot}b) \equiv X(c{\cdot}d)$ instead of $(a \equiv c)$.

We first need to compute the characteristic functions t'_{ε^*} and $t'_{\varepsilon\text{-free}}$ for the relations $\mathcal{T}'_{\varepsilon^*}$ and $\mathcal{T}'_{\varepsilon\text{-free}}$, respectively. The function t'_{ε^*} is defined by the fixed-point equation

$$t'_{\varepsilon^*} = lfp\ m.\, \alpha[m],$$

where

$$\alpha[m](a', b') = t'(a', b') + (\exists c'.\, t'(a', c') * (l'(a') \equiv l'(c')) * m(c', b')).$$

To better understand the fixed-point formulation, consider computing t'_{ε^*} by the fixed-point iteration

$$m^i = \begin{cases} |0| & \text{if } i = 0, \\ \alpha[m^{i-1}] & \text{otherwise.} \end{cases}$$

It is easy to verify that, for $i > 0$, $m^i(a', b') = 1$ if and only if there is a state c' ($a' = c'$ is possible) such that there is an empty-word path of length at most $i - 1$ that takes B' from a' to c' and $t'(c', b')$ holds.

Now $t'_{\varepsilon\text{-free}}$ can be computed from t'_{ε^*}:

$$t'_{\varepsilon\text{-free}}(a', b') = t'_{\varepsilon^*}(a', b') * (l'(a') \not\equiv l'(b')).$$

If we define *term* and *term'* as

$$term(a) = \forall b.\, t(a, b) \Rightarrow (X(a) \not\equiv X(b)),$$

and

$$term'(a') = \forall b'.\, t'(a', b') \Rightarrow (X(a') \not\equiv X(b')),$$

we can define *trap'* as

$$trap'(a') = \exists b'.\, t'_{\varepsilon^*}(a', b') * (l'(a') \equiv l'(b')) * (term'(b') + t'_{\varepsilon^*}(b', b'))\,.$$

Finally, the most general state-realization relation (excluding Property 1), is given by the fixed-point equation

$$\hat{m} = gfp\ m.\, \beta[m],$$

where

$$\beta[m](a, a') = label(a, a') * cap[m](a, a') * safe[m](a, a') * live(a, a'),$$

$$label(a, a') = (l(a) \equiv l(a')),$$

$$cap[m](a, a') = \forall b.\, [t(a, b) \Rightarrow (\exists b'.\, [t'_{\varepsilon\text{-free}}(a', b') * m(b, b')])],$$

$$safe[m](a, a') = \forall b'.\, [(t'_{\varepsilon}(a', b') * appl(a, a', b')) \Rightarrow (\exists b.\, [t(a, b) * m(b, b')])],$$

$$appl(a, a', b') = (X(a') \equiv X(b')) + (\exists c.\, [t(a, c) * (X(c) \equiv X(b'))])\,,$$

and $live(a, a') = (trap'(a') \Rightarrow term(a))$.

In view of Theorem 14.2, if $\hat{m}(q_1, q'_1) = 1$, then B' is a realization of A.

14.6 Symbolic Model Checking

In the previous section, we discussed methods for testing whether one behavior is a realization of another. Such methods cannot be applied, however, if the correctness of the specification itself is in doubt. One way to improve the "quality" of a specification is to check that it satisfies some desired properties. For example, we may want to ensure that an arbiter specification never allows both requests to be granted at the same time. In other circumstances, we may not have a complete specification, but only a collection of properties a design should satisfy. Such properties can be verified using *model checking*—an algorithm that can be used to determine

the validity of some temporal-logic formulas with respect to a behavior. In this section we discuss a simple temporal logic and a decision procedure that checks the validity of a formula in a given symbolic behavior.

We begin with a brief introduction to *temporal logic*. Propositional logic deals with absolute truths in a domain, i.e., with propositions that are either true or false. Predicate logic extends this notion of truth to relative truth, depending on the actual arguments involved. Modal logic—a special case of which is temporal logic—generalizes this concept of truth even further by making it dependent on the "world" currently being examined. Thus, within a world, predicate logic is used, whereas between worlds, modal operators are introduced. In the hardware domain, the worlds represent different states of a system, and the movement from one world to another represent the dynamic behavior of the system. For this reason, we use the word *state* rather than *world* in the sequel.

There are several types of temporal logics; for a comprehensive discussion of the temporal logics used in hardware verification, the reader is referred to [57]. In this section we highlight only one such logic—*computational tree logic* (*CTL*)—together with its associated decision procedure [37]. CTL is particularly appropriate for stating properties of asynchronous systems.

A CTL formula is defined with respect to a set of atomic formulas. These atomic formulas should be viewed as basic properties of individual states. In our context, the atomic formulas constrain the inputs and/or outputs to be 1 or 0. Formally, the syntax of a *CTL formula* is defined as follows:

1. (a) Input i is 1 (where $1 \leq i \leq n$),
   (b) Input i is 0 (where $1 \leq i \leq n$),
   (c) Output i is 1 (where $1 \leq i \leq p$),
   (d) Output i is 0 (where $1 \leq i \leq p$).
2. If f is a CTL formula, then so are
   (a) $\neg f$ (not f),
   (b) AXf (for all paths f holds in the next state),
   (c) EXf (there is a path in which f holds in the next state),
   (d) AGf (for all paths, f holds in every state),
   (e) EGf (for some path, f holds in every state),
   (f) AFf (for all paths, eventually f holds),
   (g) EFf (for some path, eventually f holds).
3. If f and g are CTL formulas, then so are
   (a) $f \wedge g$ (f and g),
   (b) $A(fUg)$ (for all paths: f until g),
   (c) $E(fUg)$ (for some path: f until g),

where we have included in parentheses the common way to read the various formulas. Other logical connectives like $\vee$, $\Rightarrow$, are defined in the usual way in terms of $\wedge$ and $\neg$.

Suppose $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$ is a behavior. If $q \in \mathcal{Q}$ and there does not exist a $p \in \mathcal{Q}$ such that $(q, p) \in \mathcal{T}$ we say that q is a *sink* in A. With A and $s \in \mathcal{Q}$ we associate an infinite *computation tree*, with root s and with an edge from vertex t to vertex u if and only if $(t, u) \in \mathcal{T}$ or t is a sink and $u = t$. An infinite path of the tree starting at the root s is an *s-path* of A. In Figure 14.10 we illustrate the construction of the computation tree from a simple behavior. Note that the sink $(10{\cdot}r_1{\cdot}10)$ is repeated indefinitely.

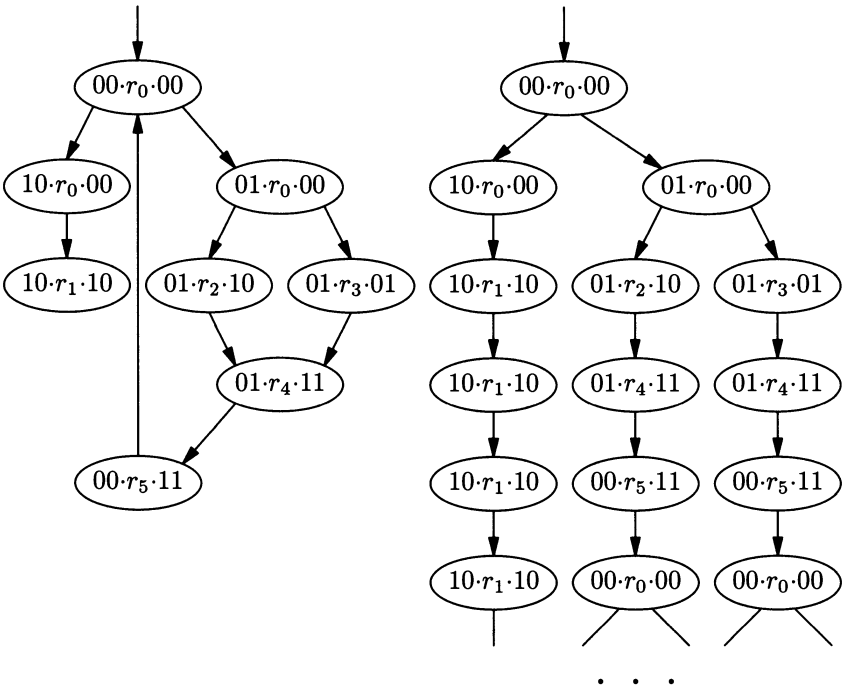


FIGURE 14.10. Simple behavior and corresponding computation tree.

Given a CTL formula f, we write $A, s \models f$ (or $s \models f$ if A is understood), to state that the formula f holds in the computation tree derived from A and rooted at s. Formally, the semantics of a CTL formula f is defined recursively as

1. (a) $s \models$ (input i is 1) holds if and only if $X_i(s) = 1$.
   (b) $s \models$ (input i is 0) holds if and only if $X_i(s) = 0$.
   (c) $s \models$ (output i is 1) holds if and only if $\psi_i(s) = 1$.
   (d) $s \models$ (output i is 0) holds if and only if $\psi_i(s) = 0$.

2. (a) $s \models \neg f$ if and only if $s \models f$ does not hold.
   (b) $s \models AXf$ if and only if $t \models f$ for every s-path $(s, t, \ldots)$ of A.
   (c) $s \models EXf$ if and only if $t \models f$ for some s-path $(s, t, \ldots)$ of A.
   (d) $s \models AGf$ if and only if f holds in every state in every s-path of A.
   (e) $s \models EGf$ if and only if f holds in every state in some s-path of A.
   (f) $s \models AFf$ if and only if for every s-path of A there is at least one state in which f holds.
   (g) $s \models EFf$ if and only if for some s-path of A there is at least one state in which f holds.

3. (a) $s \models f \wedge g$ if and only if $s \models f$ and $s \models g$.
   (b) $s \models A(fUg)$ if and only if for every s-path $(s_0, s_1, \ldots)$ of A there exists some $j \geq 0$ such that $s_j \models g$ and $s_i \models f$ for $0 \leq i < j$.
   (c) $s \models E(fUg)$ if and only if for some s-path $(s_0, s_1, \ldots)$ of A there exists some $j \geq 0$ such that $s_j \models g$ and $s_i \models f$ for $0 \leq i < j$.

The temporal logic above is useful for describing properties of behaviors. To illustrate this, consider the behavior of Figure 14.11, where the (expanded) initial state is $000{\cdot}r_0{\cdot}0$, the inputs are a, b, and c, and the output is *Out*. The behavior specifies a three-input Muller C-ELEMENT operated in an environment that may withdraw a request, but only if the behavior is in a static state. We can divide the properties we would like the behavior

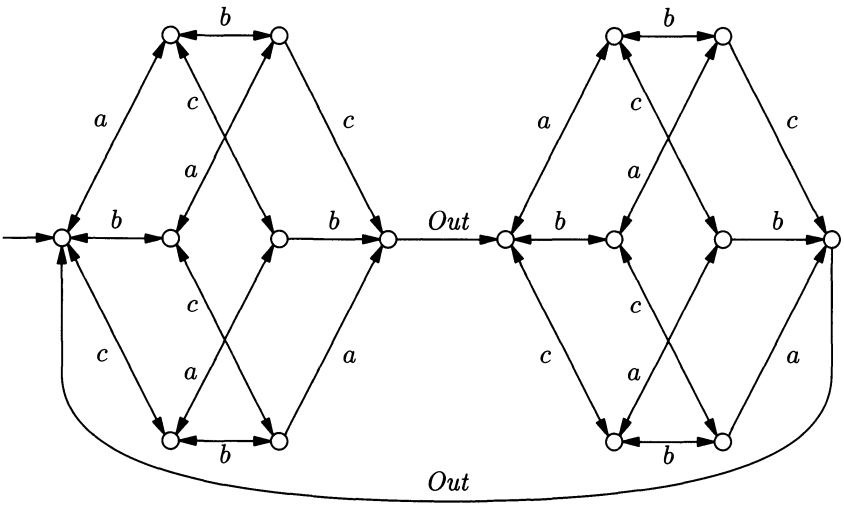


FIGURE 14.11. Specification behavior for three-input C-ELEMENT.

to exhibit into two types: *liveness* (something good will eventually happen) and *safety* (nothing bad will ever happen). We give some examples of each type of property.

For a C-ELEMENT to function properly, we should ensure that, if we keep all three inputs at the same value, then eventually the output should change to this value. We can express this liveness property by the following two CTL formulas:

$$AG(A((a = 0 \wedge b = 0 \wedge c = 0)\, U\, (Out = 0 \vee a = 1 \vee b = 1 \vee c = 1))),$$

and

$$AG(A((a = 1 \wedge b = 1 \wedge c = 1)\, U\, (Out = 1 \vee a = 0 \vee b = 0 \vee c = 0))),$$

where we have used the shorthand $a = 1$ for "input a is 1," etc. The CTL formulas state that, in any state s that is reachable from the initial state and in which all the inputs are equal, in every path leaving s, either one of the inputs must eventually change or the output must eventually change to agree with the common input value.

The following safety condition should hold: If all three inputs and the output have the same value, then the output should not change until all three inputs have changed to the complementary value. This condition can be expressed by the two CTL formulas:

$$\begin{aligned} AG(\, &(a = 0 \wedge b = 0 \wedge c = 0 \wedge Out = 0) \\ &\Rightarrow A(Out = 0\, U\, (a = 1 \wedge b = 1 \wedge c = 1))) \end{aligned}$$

and

$$\begin{aligned} AG(\, &(a = 1 \wedge b = 1 \wedge c = 1 \wedge Out = 1) \\ &\Rightarrow A(Out = 1\, U\, (a = 0 \wedge b = 0 \wedge c = 0))). \end{aligned}$$

We are interested in CTL not only because it is a concise and powerful specification language for desirable properties of a system, but also because there is a very efficient algorithm for determining whether a CTL formula holds for the initial state of a behavior. The basic algorithm, called the model checking algorithm, was introduced in [37]. The original algorithm was described in terms of *Kripke structures*,[4] which include behaviors, and requires an explicit representation of the state space. Here we present the algorithm in terms of fixed-point calculations, making it amenable to a symbolic formulation. Our formulation is similar to the one described in [28].

---

[4] A Kripke structure is a triple $\langle S, R, L \rangle$, where S is a set of states, R is a successor relation, and L is a labeling function associating with each state a subset of a given fixed set of atomic formulas.

Given a behavior $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, \mathcal{T}, \psi \rangle$ and a CTL formula f, the model checking algorithm computes the set $\mathcal{H}(f) \subseteq \mathcal{Q}$ of states that satisfy f. Let

$$\mathcal{T}' = \mathcal{T} \cup \{(q, q) \mid q \in \mathcal{Q} \text{ and } q \text{ is a sink } \}.$$

This ensures that for every q there exists a q' such that $(q, q') \in \mathcal{T}'$ by adding the self-loops needed to make all computation paths infinite. The model checking algorithm is now defined recursively as follows:

1. (a) $\mathcal{H}(\text{ input } i \text{ is } 1\ ) = \{q \mid X_i(q) = 1\}$.
   (b) $\mathcal{H}(\text{ input } i \text{ is } 0\ ) = \{q \mid X_i(q) = 0\}$.
   (c) $\mathcal{H}(\text{ output } i \text{ is } 1\ ) = \{q \mid \psi_i(q) = 1\}$.
   (d) $\mathcal{H}(\text{ output } i \text{ is } 0\ ) = \{q \mid \psi_i(q) = 0\}$.
2. (a) $\mathcal{H}(\neg f) = \mathcal{Q} - \mathcal{H}(f)$.
   (b) $\mathcal{H}(AXf) = \{s \mid \forall t.\ (s,t) \in \mathcal{T}' \Rightarrow t \in \mathcal{H}(f)\}$.
   (c) $\mathcal{H}(EXf) = \mathcal{H}(\neg(AX(\neg f)))$.
   (d) $\mathcal{H}(AGf) = gfp\ \mathcal{U}.\ \mathcal{H}(f) \cap \{s \mid \forall t.\ (s,t) \in \mathcal{T}' \Rightarrow t \in \mathcal{U}\}$.
   (e) $\mathcal{H}(EGf) = \mathcal{H}(\neg(AF\neg f))$.
   (f) $\mathcal{H}(AFf) = lfp\ \mathcal{U}.\ \mathcal{H}(f) \cup \{s \mid \forall t.\ (s,t) \in \mathcal{T}' \Rightarrow t \in \mathcal{U}\}$.
   (g) $\mathcal{H}(EFf) = \mathcal{H}(\neg(AG\neg f))$.
3. (a) $\mathcal{H}(f \wedge g) = \mathcal{H}(f) \cap \mathcal{H}(g)$.
   (b) $\mathcal{H}(A(fUg)) = lfp\ \mathcal{U}.\ \mathcal{H}(g) \cup (\mathcal{H}(f) \cap \{s \mid \forall t.\ (s,t) \in \mathcal{T}' \Rightarrow t \in \mathcal{U}\}$.
   (c) $\mathcal{H}(E(fUg)) = \mathcal{H}(\neg(A(\neg gU(\neg f \wedge \neg g)) \vee AG(\neg g)))$.

To verify that CTL formula f holds in behavior A, we simply ensure that the initial state of A is in $\mathcal{H}(f)$.

The fixed-point calculations above are used to compute the set of states that satisfy some of the "global" CTL formulas. For example, consider finding the set of states that satisfy the CTL formula AGf when given the set $\mathcal{H}(f)$ that contains all the states that satisfy f. As stated above, this set can be found by computing the greatest fixed point of the function $g(\mathcal{U}) = \mathcal{H}(f) \cap \{s \mid \forall t.\ (s,t) \in \mathcal{T}' \Rightarrow t \in \mathcal{U}\}$. First, note that g is monotone. Since $\mathcal{Q}$ is finite, it follows that the greatest fixed point is well defined. Intuitively, the fixed-point calculation consists of finding the largest subset $\mathcal{U}$ of $\mathcal{Q}$ such that 1) f holds in every state in $\mathcal{U}$, and 2) for every state $u \in \mathcal{U}$, every successor of u is also in $\mathcal{U}$. Clearly, every element $u \in \mathcal{U}$ satisfies AGf. Conversely, if an element v is not in $\mathcal{U}$, then either f does not hold in v or there is some state reachable from v in which f does not hold. (Otherwise v would have been in $\mathcal{U}$.) Both cases imply that AGf does not hold in v. Altogether, it is straightforward to prove that $s \models AGf$ if

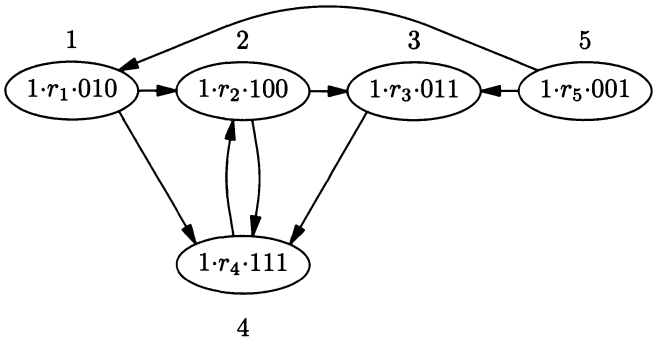


FIGURE 14.12. Example of behavior for model checking.

and only if $s \in gfp\ \mathcal{U}.\ \mathcal{H}(f) \cap \{s \mid \forall t.\ (s,t) \in \mathcal{T}' \Rightarrow t \in \mathcal{U}\}$. The other fixed-point computations can be motivated using similar arguments.

To illustrate the model checking algorithm, consider the behavior of Figure 14.12, where the outputs are a, b, and c. Consider the CTL formula $AG((a = 1) \vee (c = 1))$. We would like to compute the subset of $\mathcal{Q}$ for which this formula holds, i.e., we would like to find the set $\mathcal{U}$ such that

1. every state in $\mathcal{U}$ has $a = 1$ or $c = 1$,
2. for every $u \in \mathcal{U}$ and for every infinite path starting in u, every state in this path has $a = 1$ or $c = 1$.

First, we rewrite the formula as one using only the connectives $\wedge$ and $\neg$; the new version is $AG(\neg((\neg(a = 1)) \wedge (\neg(c = 1))))$. Since the behavior has no sinks, we have $\mathcal{T}' = \mathcal{T}$. Using the definition of $\mathcal{H}()$ we obtain

$$\mathcal{H}(AG(\neg((\neg(a = 1)) \wedge (\neg(c = 1))))) = gfp\ \mathcal{U}.\ \mathcal{H}(\neg((\neg(a = 1)) \wedge (\neg(c = 1)))) \cap \{s \mid \forall t.\ (s,t) \in \mathcal{T}' \Rightarrow t \in \mathcal{U}\}.$$

Now, since

$$\begin{aligned}
&\mathcal{H}(\neg((\neg(a = 1)) \wedge (\neg(c = 1)))) \\
&\quad = \mathcal{Q} - \mathcal{H}((\neg(a = 1)) \wedge (\neg(c = 1))) \\
&\quad = \mathcal{Q} - (\mathcal{H}((\neg(a = 1))) \cap \mathcal{H}((\neg(c = 1)))) \\
&\quad = \mathcal{Q} - ((\mathcal{Q} - \mathcal{H}((a = 1))) \cap (\mathcal{Q} - \mathcal{H}((c = 1)))) \\
&\quad = \mathcal{Q} - ((\mathcal{Q} - \{2,3\}) \cap (\mathcal{Q} - \{3,4,5\})) \\
&\quad = \mathcal{Q} - ((\{1,4,5\}) \cap (\{1,2\})) \\
&\quad = \{2,3,4,5\},
\end{aligned}$$

we get

$$\mathcal{H}(AG(\neg((\neg(a = 1)) \wedge (\neg(c = 1))))) = gfp\ \mathcal{U}.\ \{2,3,4,5\} \cap \{s \mid \forall t.\ (s,t) \in \mathcal{T}' \Rightarrow t \in \mathcal{U}\}.$$

The fixed point can now be computed as

$$\begin{aligned}
\mathcal{U}_0 &= \mathcal{Q}, \\
\mathcal{U}_1 &= \{2,3,4,5\} \cap \{s \mid \forall t.\ (s,t) \in \mathcal{T}' \Rightarrow t \in \mathcal{U}_0\} \\
&= \{2,3,4,5\} \cap \{1,2,3,4,5\} = \{2,3,4,5\}, \\
\mathcal{U}_2 &= \{2,3,4,5\} \cap \{s \mid \forall t.\ (s,t) \in \mathcal{T}' \Rightarrow t \in \mathcal{U}_1\} \\
&= \{2,3,4,5\} \cap \{1,2,3,4\} = \{2,3,4\}, \\
\mathcal{U}_3 &= \{2,3,4,5\} \cap \{s \mid \forall t.\ (s,t) \in \mathcal{T}' \Rightarrow t \in \mathcal{U}_2\} \\
&= \{2,3,4,5\} \cap \{1,2,3,4\} = \{2,3,4\} = \mathcal{U}_2.
\end{aligned}$$

Thus

$$\begin{aligned}
&\mathcal{H}(AG(\neg((\neg(a=1)) \wedge (\neg(c=1))))) \\
&= \mathit{gfp}\ \mathcal{U}.\ \{2,3,4,5\} \cap \{s \mid \forall t.\ (s,t) \in \mathcal{T}' \Rightarrow t \in \mathcal{U}\} \\
&= \{2,3,4\}.
\end{aligned}$$

Hence the formula $AG((a=1) \vee (c=1))$ holds if and only if the initial state of the behavior is 2, 3, or 4.

We now convert the model checking algorithm to symbolic form. For a more complete treatment of symbolic model checking, the reader is referred to [28]. Given a symbolic behavior $A = \langle X, \mathcal{R}, O, \mathcal{Q}, q_1, t, \psi \rangle$ and a CTL formula f, the symbolic model checking algorithm computes the characteristic function, $h[f]$, for the set of states that satisfy f. As before, we need to add self-loops to sinks to make the transition predicate complete. Let

$$t'(a,b) = t(a,b) + \left(\left(\overline{\exists c.\ t(a,c)} \right) * (a \equiv b) \right).$$

The symbolic model checking algorithm is now defined recursively as follows:

1. (a) $h[$ input i is 1 $](a) = X_i(a)$.
   (b) $h[$ input i is 0 $](a) = \overline{X_i(a)}$.
   (c) $h[$ output i is 1 $](a) = \psi_i(a)$.
   (d) $h[$ output i is 0 $](a) = \overline{\psi_i(a)}$.
2. (a) $h[\neg f](a) = \overline{h[f](a)}$
   (b) $h[AXf](a) = \forall b.\ ((t'(a,b)) \Rightarrow (h[f](b)))$.
   (c) $h[EXf] = h[\neg(AX(\neg f))]$.
   (d) $h[AGf] = \mathit{gfp}\ s.\ \alpha[f,s]$, where
   $$\alpha[f,s](a) = (h[f](a)) * (\forall b.\ (t'(a,b) \Rightarrow s(b))).$$

(e) $h[EGf] = h[\neg(AF\neg f)]$.

(f) $h[AFf] = \mathit{lfp}\ \ s.\ \beta[f, s]$, where

$$\beta[f, s](a) = (h[f](a)) + (\forall b.\ (t'(a, b) \Rightarrow s(b)))\,.$$

(g) $h[EFf] = h[\neg(AG\neg f)]$.

3. (a) $h[f \wedge g](a) = (h[f](a)) * (h[g](a))$.

(b) $h[A(fUg)] = \mathit{lfp}\ \ s.\ \gamma[f, g, s]$, where

$$\gamma[f, g, s](a) = h[g](a) + (h[f](a) * (\forall b.\ (t'(a, b) \Rightarrow s(b))))\,.$$

(c) $h[E(fUg)] = h[\neg(A(\neg gU(\neg f \wedge \neg g)) + AG(\neg g))]$.

To determine whether f is satisfied in the symbolic behavior, we check whether $h[f](q_1) = 1$.

One of the main strengths of CTL model checking is the fact that the decision procedure is completely automated. The user of the procedure need not be aware of the details of the given behavior, but can interact with the model checker to determine whether the behavior satisfies some desired properties. Also, one can modify the model checking algorithm above to produce *counterexamples* to a formula, i.e., sequences of states starting in the initial state and leading to states that do not satisfy the formula. This capability makes model checking an extremely valuable checking procedure in practical applications.

Temporal logics and model checking have the following drawbacks: In this approach to verification, the specification must be a list of desired properties. It may be difficult to judge whether the temporal formulas used completely characterize the desired behavior of the system. Also, it is easy to forget to check some property that one might take for granted. Finally, temporal logic formulas can be difficult to understand; this creates a danger of misunderstanding the properties that have been verified.

Although the symbolic approach significantly increases the usefulness of model checking, the size of behaviors that can be checked is too small to model circuits that include nontrivial data paths. Hence, model checking is primarily useful in verifying the control parts of a design. Other methods must be used to verify the data path as well as the interactions between the data path and the control parts. Alternatively, design techniques that guarantee correct operation can also be employed. We return to this topic in Chapter 15.

Chapter 15

Design of Asynchronous Circuits

Janusz A. Brzozowski, Scott Hauck, and Carl-Johan H. Seger

Most of this chapter is based on an article by Scott Hauck.[1] However, we have adapted this material to the style of the book, omitted certain topics, significantly changed other topics, and added new material.

The main theme of this book has been the analysis of circuit behavior. We have also considered the verification problem—whether an implementation realizes a given specification. However, we have not yet discussed the vitally important question of design. Systematic and efficient methods for designing asynchronous circuits are needed if such circuits are to be more widely used. As we have already pointed out, the classical methods for designing asynchronous circuits are often inadequate. The interest in asynchronous circuit design has increased dramatically in the past ten years, and today there are several methodologies that have been used on nontrivial design projects. In this chapter we give a brief survey of this exciting and fast moving field. Because of space and time limitations, we do not consider these design methods in depth, but only give the reader a flavor of the approaches together with references that can be pursued for additional detail.

How To Read This Chapter

In the introductory Section 15.1 we discuss the potential advantages of asynchronous design, and also some of the drawbacks. In the remainder of the chapter we briefly describe a number of design techniques. The relations among these techniques are discussed below. However, most sections are relatively independent and could be read separately in order to get a first impression of the methods described.

First, we describe four methods that use the bounded-delay assumption. Both gates and wires are assumed to have delays, but such delays are bibounded. We begin in Section 15.2 with the classical method of Huffman,

---

[1] Scott Hauck, "Asynchronous Design Methodologies: An Overview," *Proceedings of the IEEE*, Vol. 83, No. 1, January 1995. Copyright ©1995 by the Institute of Electrical and Electronic Engineers, Inc.

in which behaviors are specified by flow tables and circuits are assumed to operate in the fundamental mode. Next, some of the fundamental-mode restrictions are removed in Section 15.3, where Hollaar's approach is described, and in Section 15.4, where the more recent work on "burst-mode" circuits is presented. Section 15.5 discusses the design of modules for delay-insensitive networks to be discussed later in the chapter. Since these modules are designed using the bounded-delay assumption, they are included here. The modules are specified by a type of Petri net.

Second, we present two methodologies for designing speed-independent circuits. Here, component delays are permitted to be arbitrary, but wire delays are assumed to be negligible. Section 15.6 discusses "signal transition graphs," which constitute a specification formalism closely related to Petri nets. In Section 15.7 we briefly mention a somewhat different formalism called "change diagrams."

Third, we turn to the design of delay-insensitive circuits. General problems of data representation and synchronization in systems operated without a clock are discussed in Section 15.8. We then describe Ebergen's design method, which is based on "trace theory"—a language derived from regular expressions by the addition of some operators, such as a parallel composition ("weave"), and by the introduction of a distinction between input and output alphabets. This method is presented in Section 15.9.

Fourth, we consider three methodologies that involve a high-level specification language and compilation to lower-level constructs. Section 15.10 describes Martin's methodology, which has been used in several relatively large asynchronous designs, including a microprocessor. Martin uses a high-level language related to communicating sequential processes (CSP) and Dijkstra's "guarded command language." Programs are transformed into collections of simple statements, which are next expanded into "handshake protocols." These protocols are further refined into "production rules," which, in turn, are converted directly to CMOS circuits. This methodology leads to "quasi-delay-insensitive" circuits, in which some forks are assumed to have equal delays (are "isochronic"). Section 15.11 discusses van Berkel's Tangram language and the intermediate architecture of "handshake circuits." Tangram is also related to CSP and to the guarded command language. Brunvand's approach is briefly sketched in Section 15.12. Here, a subset of the language OCCAM, also related to CSP, is used. Delay-insensitive modules are designed to correspond to language constructs, such as a *while* loop. The compilation of programs to networks of modules is then straightforward.

Finally, we discuss two methodologies that combine several different features. In Section 15.13, we mention the design style of Jacobs, Broderson, and Meng, which results in "self-timed" circuits. The design uses logic blocks implemented in DCVS logic with dual-rail inputs and completion signals, connected by "interconnection blocks" that provide control. The interconnection blocks are specified by signal transition graphs. Last, but not

least, in Section 15.14 we discuss "micropipelines"—a design style in which the control functions are implemented delay-insensitively, but the data-path circuits obey the "bundled-data" convention. Micropipelines were the topic of the Turing Award lecture by Sutherland. This approach has been used for the design of a number of special purpose circuits, including microprocessors [54, 114, 130].

Section 15.15 concludes the chapter. As we have mentioned earlier, time limitations prevented us from including other promising asynchronous design techniques. The interested reader may wish to refer to [72] for an overview of algebraic approaches to the specification of safety and progress properties of delay-insensitive circuits. Such approaches make it possible to specify circuits concisely and facilitate verification of designs.

15.1 Introduction

Much of today's synchronous logic design is based on two major assumptions: all signals are binary and time is discrete. Both of these assumptions are made in order to simplify logic design. The assumption that signals are binary permits us to use Boolean algebra to describe and manipulate logic circuits. The assumption that time is discrete permits us to ignore hazards to a large extent.

Asynchronous digital circuits keep the assumption that signals are binary but remove the assumption that time is discrete. This has the following potential benefits:

No clock skew

Clock skew is the difference in arrival times of the clock signal at different parts of the circuit. Since asynchronous circuits—by definition—have no global clock, there is no need to worry about clock skew. In contrast, the designer of a synchronous circuit must often slow down its operation in order to accommodate the skew. As VLSI feature sizes decrease, clock skew becomes a much greater concern.

Lower power

In synchronous circuits, clock lines have to be toggled and circuit nodes have to be precharged and discharged even in parts unused in the current computation.[2] For example, if a floating-point unit in a processor is not used in a given instruction stream, it must still be operated by the clock. Although asynchronous circuits often require more signal transitions in a given computation than do synchronous circuits, these transitions usually occur only in areas involved in the current computation.

---

[2]In fairness it should be pointed out that in some synchronous designs the clock is selectively turned off and on in different subsystems as needed.

Average-case instead of worst-case performance

Synchronous circuits must wait until the slowest possible computation has been completed before latching the results; this yields worst-case performance. Many asynchronous circuits sense when a computation has ended; this gives average-case performance. For circuits such as ripple-carry adders—where the worst-case delay is significantly larger than the average-case delay—this can result in substantial savings.

Easing of global timing issues

In circuits such as synchronous microprocessors, the clock rate, and thus performance, is dictated by the slowest (critical) path. Thus, the design must be carefully optimized to achieve the highest clock rate; this optimization must be applied also to rarely used portions of the circuit. Since many asynchronous circuits operate at the speed of the path currently in operation, rarely used portions of the circuit can be left unoptimized without adversely affecting overall performance.

Better technology migration potential

Circuit functions are often implemented in several different technologies during their lifetime. Early circuits might be implemented with mask- or field-programmable gate arrays (MPGAs or FPGAs), while later production runs might migrate to semi-custom or custom ICs. Often, greater performance for synchronous circuits can be achieved only by migrating all components to a new technology, since the overall performance is based on the longest path. In many asynchronous circuits, migration of only the more critical components can improve average performance, since performance depends only on the currently active path. Also, since many asynchronous circuits sense computation completion, components with different delays may be substituted for older components without altering the rest of the circuit.

Automatic adaptation to physical properties

The delay through a circuit can change with variations in fabrication, temperature, and power-supply voltage. In synchronous circuits one must assume that the worst possible combination of factors is present and one must clock the circuit accordingly. Asynchronous circuits that sense computation completion run as quickly as the current physical properties allow.

Robust mutual exclusion and external input handling

Elements that guarantee mutual exclusion of independent signals or synchronize external signals with a clock are subject to metastability [31]. A metastable state is a state of unstable equilibrium in which a circuit can remain for an unbounded amount of time [96]. For example, as we have mentioned in Chapter 6, a NOR latch can exhibit this

type of behavior. Synchronous circuits require all elements to have bounded response time. Thus, there is some chance that mutual exclusion elements will fail in a synchronous circuit. Most asynchronous circuits can wait an arbitrarily long time for the mutual exclusion element to leave the metastable state; thus mutual exclusion is robustly implemented. Also, since there is no clock with which signals must be synchronized, asynchronous circuits accommodate external inputs, which are by nature asynchronous, more gracefully.

With all of the advantages of asynchronous circuits, one might wonder why synchronous circuits predominate. The reason is that asynchronous circuits have several problems as well. They are more difficult to design in an ad hoc fashion than are synchronous circuits. In a synchronous circuit, a designer can simply define the combinational logic necessary to compute given functions, and add latches to store the results of these computations. By providing a long enough clock period, one removes all worries about hazards and dynamic states of the circuit. In contrast to this, designers of asynchronous circuits must pay a great deal of attention to the dynamic states of the circuit. Hazards must be removed from the circuit (or not introduced in the first place) to avoid incorrect results. The ordering of operations—fixed by the placement of latches in a synchronous circuit—must be carefully ensured by the asynchronous control logic. For complex circuits, these issues become too difficult to handle manually. Unfortunately, in general, the existing CAD tools and implementation alternatives available for synchronous circuits cannot be used in asynchronous design. For example, some asynchronous methodologies severely limit the transformations permitted for logic decomposition. Placement, routing, partitioning, logic synthesis, and most other CAD tools either cannot be used at all or require extensive modifications in order to be applicable to asynchronous circuits.

Finally, even though many of the advantages of asynchronous circuits relate to higher performance, it is not clear that asynchronous circuits are actually faster in practice. Asynchronous circuits generally require extra time because of their communication protocols; this increases the average-case delay. It is unclear whether this cost is greater or smaller than the benefits listed previously, and more research in this area is necessary.

Because of all the problems listed above, asynchronous design is an important research area. Regardless of how successful synchronous designs are, there will always be a need for asynchronous logic in interface circuits. Also, although ad hoc design of asynchronous circuits is impractical, there exist methodologies and CAD algorithms developed specifically for asynchronous design.

Several of the main asynchronous design approaches are surveyed in this chapter. Because of space limitations, we do not attempt to include all of the existing methodologies, nor do we explore all the subtleties of the methodologies that are included. Instead, we discuss the essential aspects of

some classical methods and some of the more promising modern methods. This should provide the reader with a solid framework on which to base further study. Likewise, we do not cover many of the related areas—such as testing—which are important to any design, yet too complex to be handled adequately here. For an introduction to asynchronous testing, see [69].

15.2 Fundamental-Mode Huffman Circuits

The classical model for asynchronous circuits is quite similar to that used for synchronous circuits, except that delays are used in place of clocked latches or flip-flops; see Figure 15.1. It is assumed that the delays of all the circuit elements and wires are bi-bounded. Circuits designed with this model (usually coupled with the fundamental-mode assumption) are generally referred to as Huffman circuits, after D. A. Huffman, who developed many of the early concepts [66, 67].

The circuit to be synthesized is usually specified by a flow table [135], such as those we have used in Chapters 1 and 12. Normally it is assumed that each unstable state leads directly to a stable state (i.e., the behavior is direct), with at most one transition occurring on each output variable. As in synchronous sequential circuit design, the flow table is first reduced, and the reduced table is then encoded. Finally, expressions for the excitation functions for all state variables are found with the aid of some minimization programs. The reader may wish to refer to the divide-by-2 counter in Chapter 1 to recall some of these design steps.

Several special concerns not occurring in synchronous circuits arise in asynchronous designs. Since there is no clock to synchronize input arrivals, the circuit must behave properly in intermediate states caused by multiple input changes. For example, suppose the input changes from 00 to 11; then it may briefly pass through either 01 or 10. One must ensure that the entries in columns 01 and 10 of the flow table are appropriately selected so that the transition accompanying the double input change is independent of the transient inputs.

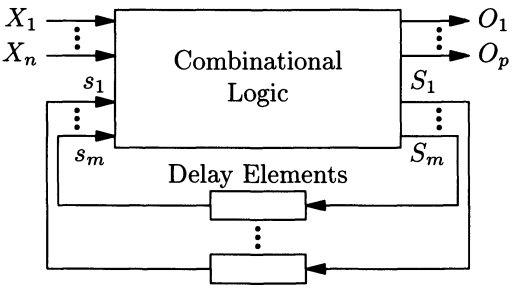


FIGURE 15.1. Classical asynchronous sequential circuit structure.

We must also deal with hazard removal. As we have shown in Chapter 7, hazards on circuit outputs may cause unexpected short pulses that may result in incorrect behavior of circuits that receive these outputs. All static and dynamic hazards corresponding to a single input change can be eliminated by adding certain redundant products to a sum-of-products realization of a circuit [135]. Unfortunately, this procedure cannot guarantee correct operation when several inputs are allowed to change simultaneously. The solution generally adopted is to impose the single-change restriction, although it may be difficult to impose such a restriction on the environment.

An important point needs to be made about the sum-of-products form. As the number of circuit inputs increases, the number of inputs to the AND and OR gates increases. Since most technologies either restrict the number of inputs to a gate, or involve long delays in gates with large fan-ins, it is important to have methods for decomposing large gates. As proven in [135], many algebraic transformations, including the associative, distributive, and DeMorgan's laws, do not introduce any new hazards. Thus, a sum-of-products form can be changed by these transformations into a multilevel expression involving smaller fan-in. The ability to use logic transformations is an important advantage of this methodology, for some other methodologies do not allow them.

Note that some transformations, such as $ab + ac + \bar{b}c = ab + \bar{b}c$ can introduce hazards. In the circuit using the expression on the right-hand side, there is a static hazard when $a = b = c = 1$ and b changes to 0. Before the change, the output is 1 because of the product ab. After the change, the product $\bar{b}c$ holds the output at 1. But, if the b and $\bar{b}$ inputs are both 0 during the change, because of a slow inverter, the output might become 0 temporarily. This hazard is prevented in the expression on the left-hand side by the presence of the product term ac.

To extend the combinational circuit methodology to sequential circuits, we use a model similar to that used for synchronous circuits. See Figure 15.1 and compare it to Figure 1.2. Since we made the restriction that only one input to the combinational logic can change at a time, this forces several requirements on the asynchronous circuit. First, we must ensure that the combinational logic has settled in response to a new input before a state variables changes. This is done by placing delay elements in the feedback lines. Also, the same restriction dictates that only one state variable can change at a time. State encoding can be done in such a way that only a single state bit changes in each state transition; however, these encodings sometimes require multiple representations of each state [135], and complicate the combinational logic. "One-hot" encodings—in which each state q_i is represented by a vector y with $y_i = 1$ and with $y_j = 0$ for $i \neq j$—require two transitions, but simplify the associated logic. A state transition from q_i to q_j is accomplished by first setting y_j and then resetting y_i. The final requirement is that the next external input transition cannot occur until the entire circuit settles in a stable state, i.e., fundamental-mode operation

is used. For a one-hot encoding, this means that a new input must be delayed long enough for three change propagations through the combinational logic and two through the delay elements. With a one-hot encoding one can implement each state variable with the same type of module; state transitions are then realized by appropriate connections between modules. The bounded-delay design method proposed in [41] illustrates this approach.

15.3 Hollaar Circuits

The fundamental-mode assumption, while simplifying logic design, increases computation time. As has been mentioned above, the fundamental mode is often coupled with the single-input-change assumption. There are certain cases in which both restrictions can be removed. For example, suppose an output O is determined by some function of the form $X_1 + f(X_2, X_3, X_4)$. When $X_1 = 0$, changes in the remaining inputs could produce hazards in O. However, when $X_1 = 1$ the remaining inputs can change in an arbitrary fashion without affecting the output, and the total state need not be stable when such changes occur. Clearly, this observation can eliminate the fundamental-mode assumption only in certain special cases.

A method due to Hollaar [63] uses detailed knowledge of the implementation to allow new transitions to arrive earlier than the fundamental-mode assumption would allow. A one-hot state assignment is used in this approach, and the value of each state variable is stored in a set-reset NAND latch. Suppose first that we have a "straight-line" sequence q_1, q_2, q_3 of transitions as shown in Figure 15.2, and the transition into state q_i occurs when certain conditions defined by the function c_i hold. In Figure 15.2, NAND gates 5 and 6 form a set-reset latch for state q_2. Suppose that the circuit is in state q_1; then $s_1 = 1$ and the other state variables are all 0. The latch with output s_2 can be set only if the latch of the previous state q_1 is set, and the conditions c_2 for a transition from q_1 to q_2 are satisfied. Once s_2 becomes 1, gate 6 becomes 0 and causes a 0 to 1 transition in gate 3. This, in turn, causes s_1 to become 0. Thus, the "present-state" latch is reset after the "next-state" latch has been set. This basic scheme is extended to more general types of transitions by the use of FORK and JOIN modules.

Careful analysis of the implementation in Figure 15.2 shows that the fundamental-mode assumption can be relaxed. Suppose that each gate has delay δ. The fundamental-mode assumption requires that 6δ time units must elapse between transitions. For example, after c_2 becomes 1 (causing a state change from q_1 to q_2), the sequence of gate changes is 4, 5, 6, 3, 2, 4, before the circuit stabilizes. However, after time 3δ (the sequence 4, 5, 6), state bit s_2 is properly set, and c_2 can safely change again. This delay is half of the delay required by the fundamental-mode assumption. Also, after gate 5 becomes 1, gate 7 can change and begin a transition to state q_3. The time between a change on c_2 and a change on c_3 would then be

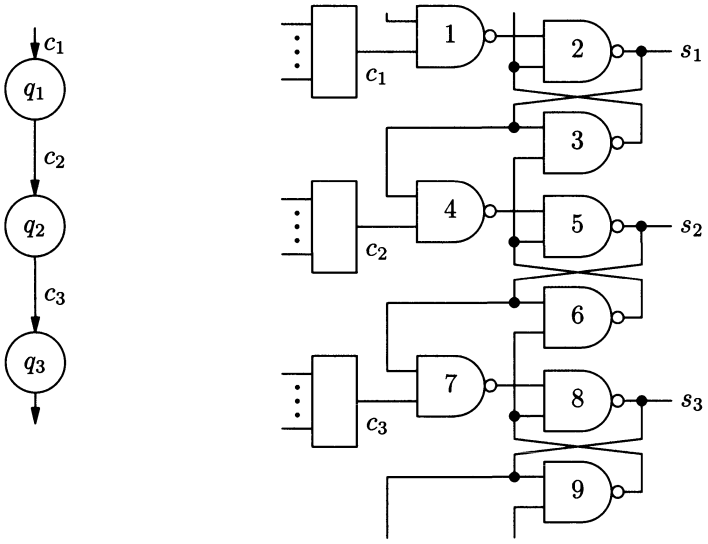


FIGURE 15.2. Hollaar's implementation.

only 2δ, since only gates 4 and 5 would have changed. This is three times faster than the fundamental-mode operation. As long as gate delays are approximately equal, state bits s_1 and s_2 will be eventually 0, and state bit s_3 will be 1. Although there is a possibility that three state bits might be 1 at the same time, the final state will be correct for each transition.

Unfortunately, Hollaar's method has some disadvantages. First, the general case where a state can have multiple successors requires five gate delays for proper operation. Second, some hazards can occur and the circuit can enter an incorrect state. Imagine that the circuit is in state q_1, and that $c_2 = X_1$ and $c_3 = \overline{X_1} f(X_2, X_3, X_4)$. Further assume that input X_1 is changing from 0 to 1, and $f(X_2, X_3, X_4) = 1$. As expected, c_2 becomes 1 and eventually causes s_2 to be 1. However, c_3 might take longer to become 0 than it took c_2 to become 1. If this difference is greater than the delays in gates 4 and 5, gate 7 might produce a hazard pulse that might cause state bit s_3 to be set. Thus, the circuit would not correctly implement the specified transitions. Nevertheless, Hollaar's method permits the relaxation of the fundamental-mode requirements in some circuits.

15.4 Burst-Mode Circuits

A design methodology called burst-mode attempts to move closer to synchronous design styles than Huffman's method. Like Huffman's method, it is based on sequential machines, but the circuits designed are hazard-free (at the gate level) by construction, and inertial delays are not needed for hazard elimination.

As shown in Figure 15.3, circuits are specified by state graphs in which each transition is labeled by a nonempty set of inputs (an *input burst*), and a set of outputs (an *output burst*). The state labels show the values of the input vector, an internal state variable, and the output vector. More details about this graph will be given later. When the circuit is in a given state, the inputs in one of the input bursts leaving this state can change. Such changes of inputs in a burst are allowed to occur in any order, and the circuit does not react until the entire input burst has occurred. No input burst can be a subset of another input burst leaving the same state. This is required so that the circuit can unambiguously determine when a complete input burst has occurred, and can react accordingly. For example, in the state graph in Figure 15.3, an edge with input burst $\{X_1, X_2, X_3\}$ could not be added from state q_1 to state q_4, because the other input bursts leaving state q_1 would be subsets of this input burst.

Once an input burst is complete, the circuit activates the specified output burst and enters the specified next state. A new input change is allowed only after the circuit has completely reacted to the previous input burst. Thus, burst-mode systems still require the fundamental-mode assumption, but only between transitions in different input bursts.

As described in [149, 150], burst-mode circuits can be implemented by techniques similar to those used for Huffman circuits. Since burst-mode circuits allow multiple input changes, one would expect to have the same hazard problems that motivated the single-input-change restriction in Huffman circuits. However, the burst-mode specification allows outputs to change only after an entire input burst, as we now describe. Given two input vectors A and B, we say that an input vector C is "between" A and B if, for all i, C_i is either equal to A_i or to B_i. A given Boolean function f is said to have a *burst-mode input transition* from input vector A to input vector

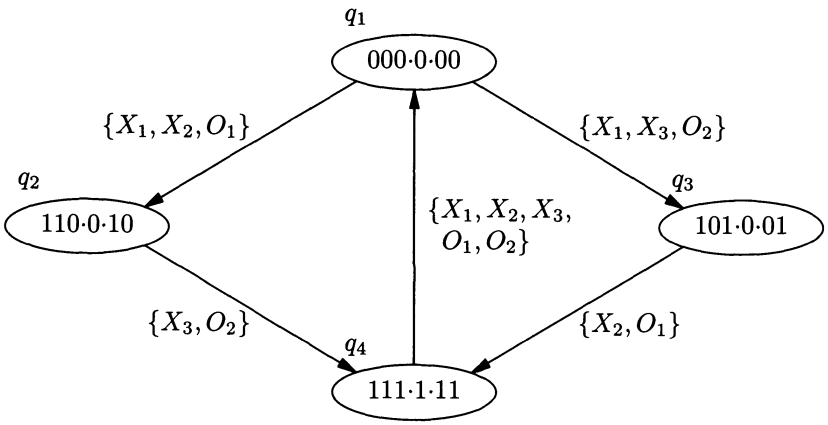


FIGURE 15.3. Burst-mode specification.

B if $f(A) = f(C)$ for every vector C, $C \neq B$, between A and B. Thus the output of a circuit implementing f is allowed to change only after every input (that participates in the transition) has changed. In this synthesis method all combinational functions are guaranteed to have burst-mode input transitions.

It is a constraint on the environment that transitions from the next input burst are not allowed to arrive until the circuit has finished reacting to the previous burst. A technique, similar to that used in Huffman circuits, of adding redundant product terms to a sum-of-products form to remove hazards is sufficient to implement burst-mode circuits [112]. Finally, burst-mode circuits must use the same special state encodings and delays on the feedback lines as do Huffman circuits.

A different method of implementing burst-mode specifications is described in [110, 111]. As shown in Figure 15.4, a clock is generated locally in each module being designed; this clock is independent of the local clock in any other module. This is intended to avoid some of the hazards found in the Huffman design style discussed earlier.

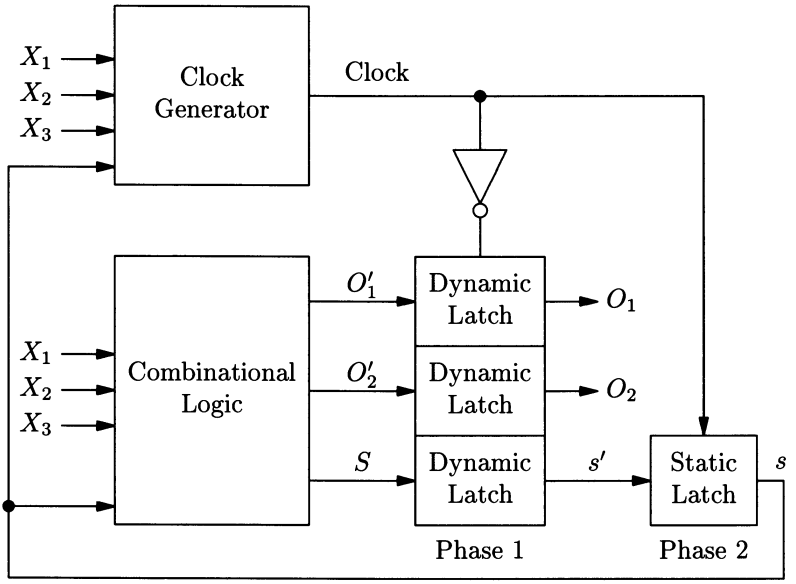


FIGURE 15.4. Circuit schematic for a locally clocked implementation.

To understand how a locally clocked module works, consider the example of Figure 15.3. This specification requires one state bit s, which is 1 when the machine is in state q_4, and 0 otherwise. A complete table of combinations for this specification is shown in Table 15.1. Assume that the circuit is stable in state q_1 with all the inputs, all the outputs, and s set to 0. In a stable state, the local clock is 0, and data can pass through the phase-1

TABLE 15.1. Table of combinations for burst-mode specification.

X_1	X_2	X_3	s	O_1	O_2	S
0	-	-	0	0	0	0
1	1	0	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	1	1	1
1	-	-	1	1	1	1
-	1	-	1	1	1	1
-	-	1	1	1	1	1
0	0	0	1	0	0	0

latches. The first transitions to occur are X_1 and X_2 being set to 1. This case is simple because there is no state change, since s is still stable at 0. Consequently, the local clock remains 0. The only effect is that, once both the X_1 and X_2 transitions occur, the combinational logic generating O_1' changes its output to 1, and that value propagates through the phase-1 latch to output O_1. A more interesting case occurs when the input X_3 then changes to 1 as well. In this case, the state variable s must change. First, the combinational logic for the output O_2' and for the state bit excitation S change in response to the change in X_3, resulting in $O_2' = 1$, and $S = 1$. The conditions for enabling the local clock are now satisfied, but the clock is delayed to ensure that the output and state changes propagate through the phase-1 latches (i.e., until O_2 and s' become 1) before the clock becomes 1. Once the clock becomes 1, the phase-1 latches are disabled, and the phase-2 latches are allowed to pass their values through. This permits the new value of the state bit to reach the combinational logic and the clock generator. However, since the phase-1 latches are disabled, any new values, including hazards, are not passed through. The local clock is then reset by the arrival of the new state, the phase-2 latches are disabled, and the phase-1 latches are again allowed to pass data. This completes the reaction of the module to the new data, and the module is now ready for another input burst.

The major advantage claimed for the locally clocked implementation is the avoidance of the hazards encountered by normal Huffman circuits. Also, standard synchronous state assignment techniques can be employed. However, not all hazards can be ignored. In all transitions the outputs are generated directly in response to the inputs, and the local clock offers no hazard protection. Thus, the redundant products necessary in Huffman circuits are also needed for the output logic, and special care must be taken to avoid dynamic hazards [112]. The local clock logic may also contain hazards. Although the clock signal is not directly seen by the environment, a hazard on the clock line could cause the state to change partially or completely when no change was intended.

15.5 Module Synthesis Using I-Nets

In contrast to bounded-delay models, delay-insensitive circuit design (to be discussed later) assumes that the delays in basic modules as well as in wires are unbounded. To make delay-insensitive circuit design practical for general computations, we must have a set of basic modules that both work properly under the delay-insensitive assumption and provide sufficient functionality to implement a wide class of circuits. From Chapter 13 and [90], we know that standard gates are not suitable. Consequently, we must abandon the delay-insensitivity goal when designing basic modules, i.e., we must design such modules using the bounded-delay assumption. This is not an unreasonable compromise, since a basic module usually involves a relatively small area on a chip and its delays can be controlled to a large extent. Once the modules are designed, however, we can carry out the rest of the design—of networks constructed using such modules—with delay-insensitive methods.

A methodology for the design of modules for use in delay-insensitive networks has been proposed in [35, 36, 102, 103, 129]. This methodology is founded on I-nets (for *interface nets*), a model based on Petri nets [108, 113]. Note that a second methodology based on Petri nets, namely that of signal transition graphs (STGs), is discussed in Section 15.6. STGs and I-nets have many similarities, which we discuss in Section 15.6.

I-nets are used as a formalism for specifying behaviors[3] of modules. An *I-net* is a directed graph with two types of nodes: *places*, denoted by circles, and *transitions*, denoted as bars. The graph is bipartite, with places connected by directed edges only to transitions and transitions connected only to places. A place may hold a finite number of *tokens*, denoted by small black dots inside the place's circle. If there is an edge from a place p to a transition t, then p is an *input place* of t. Similarly, if there is an edge from t to p, then p is an *output place* of t. A *marking* of an I-net is an assignment of tokens to places. Two simple I-nets are shown in Figure 15.5.

A transition is *enabled* when each of its input places contains at least one token. An enabled transition may *fire* by removing a token from each of its input places, and putting a token in each of its output places. A sequence of firings of single transitions in an I-net is called an *execution* of the I-net.

The left part of Figure 15.5(a) shows the symbol for a JOIN element with inputs X_1 and X_2 and output O. Starting from a stable state, the element produces an output change only after both of its inputs change. Since the inputs may change at different times, the JOIN provides a very basic synchronization function. The right part of Figure 15.5(a) shows an I-net for the JOIN and a simple environment. Similarly, Figure 15.5(b) shows the

[3]Here we use the word "behavior" in its intuitive sense and not in the formal sense of Chapter 11.

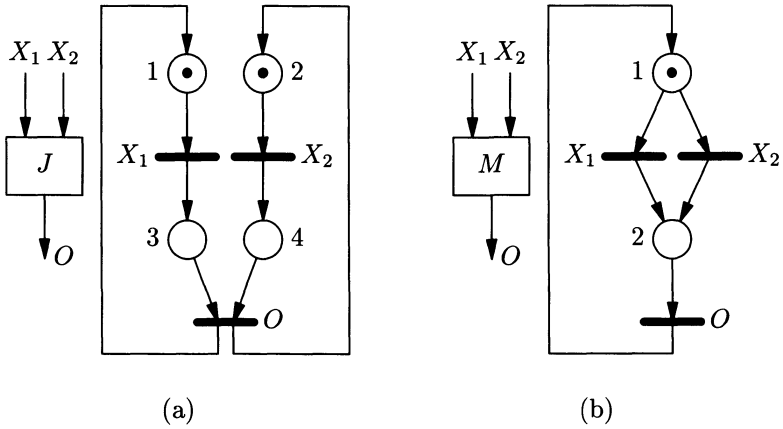


FIGURE 15.5. Examples of I-nets: (a) JOIN, (b) MERGE.

symbol for a MERGE element along with an I-net for the MERGE and a simple environment. In the MERGE element a change in either input produces a change in the output. More details about the operation of these I-nets are provided below.

In Figure 15.5 the transitions labeled X_1 and X_2 are enabled in both I-nets. For the JOIN element, once transitions X_1 and X_2 have fired, there are tokens only in places 3 and 4. At this point the O transition is enabled; once it fires, the I-net returns to the pictured state. For the MERGE element, either X_1 or X_2 can fire, but not both, because the firing of either transition removes the input token, disabling the other transition. After X_1 or X_2 fires, O becomes enabled. Its firing returns the graph to the pictured state.

To relate I-nets to circuits, we associate signal labels with the I-net transitions. Thus every firing of an I-net transition corresponds to a signal transition on the corresponding signal wire. Note that a given label may appear on several transitions. An I-net not only determines the proper functioning of the module being specified, but also determines how the environment of the module must behave. For example, the environment for the JOIN element is required to produce exactly one transition on both the X_1 and the X_2 wires between any two transitions on the O output. The MERGE element restricts the environment to exactly one transition on one of X_1 and X_2 between any two O transitions. Of course, the I-net is not a complete specification of the environment but only a description of its interaction with the module in question. Any environment that fulfills the specified sequences on the module inputs and outputs may be used. For example, one input of a MERGE element could be connected to the output of a JOIN, the other input could be kept fixed, and the output of the MERGE could be connected to the X_1 input of the JOIN, without violating the environment specification.

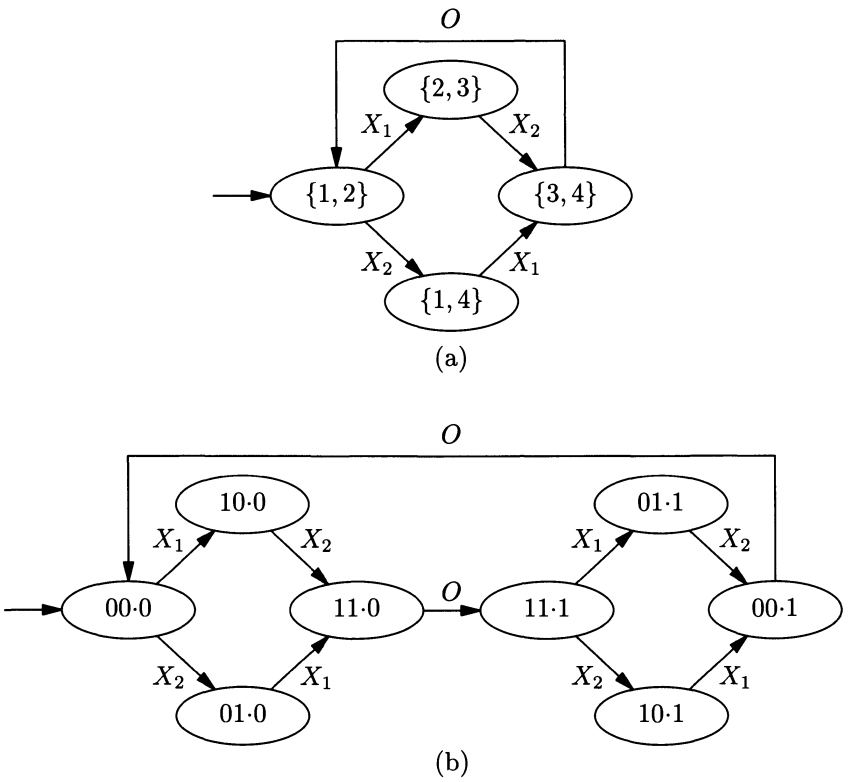


FIGURE 15.6. State graphs for JOIN: (a) ISG, (b) EISG.

To find a circuit corresponding to an I-net specification, an *interface state graph* (ISG) is first derived from the I-net. This graph shows the states the interface can assume and describes the allowed state changes. For example, in the JOIN, the initial state of the ISG corresponds to the marking of places 1 and 2; this is denoted as state $\{1,2\}$; see Figure 15.6. If X_1 fires, we reach state $\{2,3\}$, and, if X_2 fires, we reach $\{1,4\}$. These two states have only one possible successor state $\{3,4\}$ reached after both X_1 and X_2 have fired. In turn, state $\{3,4\}$ can only be followed by state $\{1,2\}$. Each edge between two states in an ISG is marked by the transition that causes the corresponding state change. Thus, we have the edge labeled X_1 from state $\{1,2\}$ to state $\{2,3\}$, etc.

The ISG describes all possible transitions of interface signals. This information must be converted to a representation based on logic levels that are used in digital circuits. This is done by constructing an *encoded ISG* (*EISG*), from the ISG. The designer has to choose an initial state for all the signals. In the case of the JOIN, we can use state $X_1X_2{\cdot}O = 00{\cdot}0$. From here on, we follow the ISG in order to develop the state transitions in the

TABLE 15.2. Table of combinations for JOIN.

X_1X_2O	O'
000	0
001	0
010	0
011	1
100	0
101	1
110	1
111	1

EISG. Thus, if a transition of signal X_1 occurs, we reach EISG state 10·0. Similarly, if X_2 changes, we have state 01·0. The complete EISG is shown in Figure 15.6(b).

The EISG is an expanded version of the ISG. Each state of the ISG corresponds to one or more states in the EISG. This "state-splitting" is often necessary, because the EISG must make a distinction between a rising transition on a wire and a falling transition on the same wire, whereas no such distinction is needed in the ISG. In the example of Figure 15.6, every state in the ISG is split into two states in the EISG. Finally, there are ISGs for which no valid EISG can be created. We will return to this topic shortly.

Note that, in general, if a valid EISG can be created, it is a restricted type of behavior (where we now use the word in the formal sense of Chapter 11). The total state is completely determined by the state label. Furthermore, only single input changes are permitted. (Concurrent input changes are represented as interleaved changes, as demonstrated by the JOIN element.) A table of combinations specifying the next value of the output in terms of the present values of the inputs and the output can be constructed directly from an EISG as follows. In any static state, the output should keep its old value. In any dynamic state, the output should change. If an input-output combination does not occur in the EISG, a "don't care" value is assigned to the next output. For the case of the JOIN, we find the function of Table 15.2. The next value O' of the output should be the same as its present value O in the static states 00·0, 01·0, 10·0, 11·1, 01·1, and 10·1. For those states, we simply copy the value of O to O'. For the two dynamic states, we complement the value of O to obtain the value of O'. From such a table, we can find a Boolean expression for the output, using standard methods of simplification. In the case of the JOIN we find

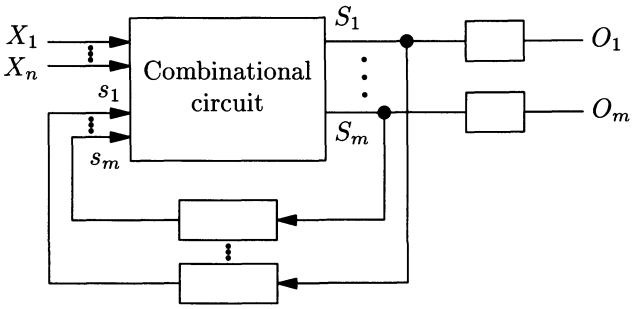
$$O' = X_1X_2 + X_1O + X_2O.$$

The module can now be implemented by the two-level combinational circuit corresponding to this expression, if we connect the O' output to the O input of the circuit.

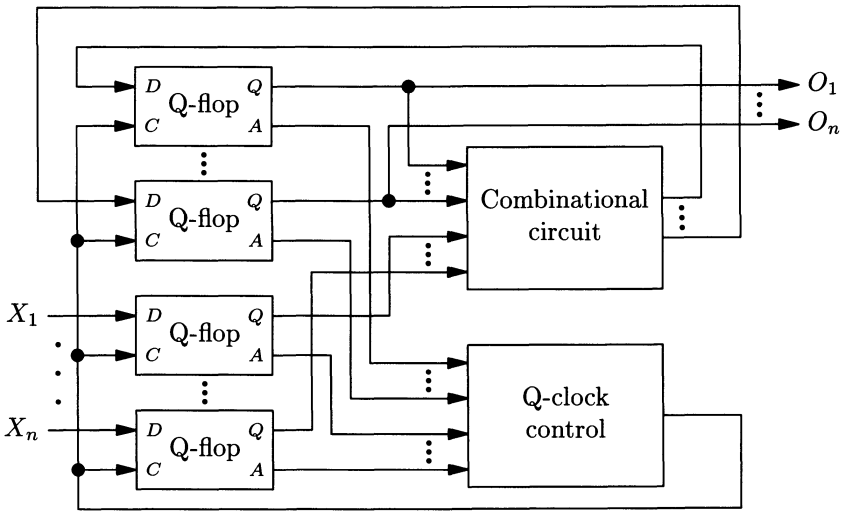
Complete algorithms describing the design steps illustrated above are given in [129]. Note that the algorithm for constructing an ISG from an I-net can be exponential in the number of places, since all the markings might have to be enumerated. This means that this synthesis procedure is inappropriate for large circuits with complex I-nets. It is, however, quite adequate for designing small modules.

Before we discuss specific implementation structures, we point out that not all I-nets properly represent delay-insensitive circuits. For example, an I-net can have two consecutive firings of a transition, and the corresponding two consecutive signal transitions on a single wire. This constitutes a hazard on that wire, because the second signal transition could "catch up" to the first one and cancel its effect. One has to admit such possibilities, if no assumptions are made about component and wire delays. A module described by an I-net with such a hazard would not be delay-insensitive. Consequently, the class of I-nets must be restricted to avoid such problems. The so-called *foam rubber wrapper* constraint [103] provides an appropriate restriction. It states that we must be able to attach arbitrary delays to the input and output wires of any delay-insensitive circuit, and the new interface so created must behave like the original module, with no hazards introduced by the added wire delays. If the introduction of these delays allows signal-transition sequences not present in the original circuit, the circuit is not delay-insensitive. Note that the same requirement can be expressed as local constraints on ISGs, as described in [133]. Different formalizations have also been given by [47, 121, 140].

While the foam rubber wrapper constraint helps identify a specification as delay-insensitive, there are other problems that might preclude a proper implementation. First, as shown in the MERGE specification, an I-net can include a mutually exclusive choice. This presents no difficulties in the MERGE I-net, since the decision between firing X_1 and X_2 is made by the environment, and the environment might have extra information with which to make this decision deterministically. However, modules in which some form of arbitration is needed are not properly handled by this methodology and must be handled differently [129]. The second difficulty arises because the I-net structures are too powerful and can define languages that are not regular. Such I-nets cannot be implemented as finite-state machines. For example, an I-net can be constructed [108] to accept the language $\{a^i b^i \mid i \geq 0\}$. Care must be taken to avoid such specifications. Third, the EISG generated by the algorithm may have more than one state with the same input/output label. Unless they are equivalent, such states must be distinguished by the addition of some binary internal state variables. An ad hoc method for doing this is presented in [129], but no automatic method for handling these situations is available. Note, however, that some of the techniques of state variable insertion developed for STGs (discussed in Section 15.6) are also applicable to I-nets.



(a)



(b)

FIGURE 15.7. Implementation structures: (a) clock-free, (b) locally clocked.

The function obtained for the module output from the EISG is implemented by a circuit structure similar to those presented in Sections 15.2 and 15.4. Two basic structures are used. The clock-free structure of Figure 15.7(a) does not follow the restrictions of fundamental-mode operation and permits some hazards. The effects of the hazards are eliminated in two steps [51]. The first step deals with hazards caused by transitions that are sequential in the specification but become concurrent because of circuit delays. For example, suppose a signal transition at an input of a MERGE element causes a signal transition at its output; because of wire delays these two transitions may arrive in reverse order in some other part of the circuit. The solution is to add delays to both the output and feedback

lines [51, 135]. In the second step, hazards due to concurrent transitions (i.e., transitions that are unordered in the specification) are removed. Static hazards can be eliminated by adding redundant product terms in the two-level realization of Boolean function [135]. Dynamic hazards are handled by the insertion of inertial-delay elements [135] at the outputs of certain gates. Note that these inertial delays can be combined with the delays necessary to eliminate hazards due to sequential transitions. While this approach allows multiple input changes (which are necessary in most delay-insensitive elements, including the JOIN), it significantly increases the total delay in the system.

The second implementation strategy [103, 117] is similar to the locally clocked burst-mode circuits discussed earlier. However, instead of generating a clock only when inputs arrive, a locally clocked module, *Q-module*, constantly clocks its latches, but the clock is totally internal to the module and is "pausable." The system behaves like a standard synchronous system. Inputs and state variables are latched on a regular basis, and enough time is allowed between clock pulses to permit the combinational logic to completely settle after each input change. This structure could have the same problems with asynchronous inputs as does a standard synchronous circuit. These problems are overcome as follows: First, all flip-flops, called *Q-flops*, are built with synchronizers—elements that can reliably latch asynchronous inputs. Second, since synchronizers can take an unbounded amount of time, the Q-flops must inform the Q-clock control when they have completed their operation (using the wires labeled A in Figure 15.7(b)), so that the clock can be sufficiently delayed. In this way, a completely synchronous state machine can be reliably embedded in an asynchronous environment.

Each of the two methods above has some disadvantages. The clock-free structure adds delays to the system, but it is simple. Since combinational modules require only small delays to be added, this model seems to be suitable for such modules. On the other hand, the locally clocked module has the added logic and delay of a rather complex latching structure.

15.6 Signal Transition Graphs

Signal transition graphs (*STGs*), were introduced in [33, 34]; *signal graphs*, a model almost identical to STGs, were introduced independently in [118]. These models have received considerable attention. Like I-nets, STGs specify asynchronous circuits by Petri nets [108, 113] with transitions labeled by signal names. When a labeled transition fires, the corresponding signal changes in the circuit. In contrast to I-nets, many STG methodologies attempt to achieve greater automation of the synthesis process and avoid exponential complexity by restricting the types of Petri nets allowed. It should be pointed out that by limiting the class of Petri nets one may also limit the class of circuits that can be designed.

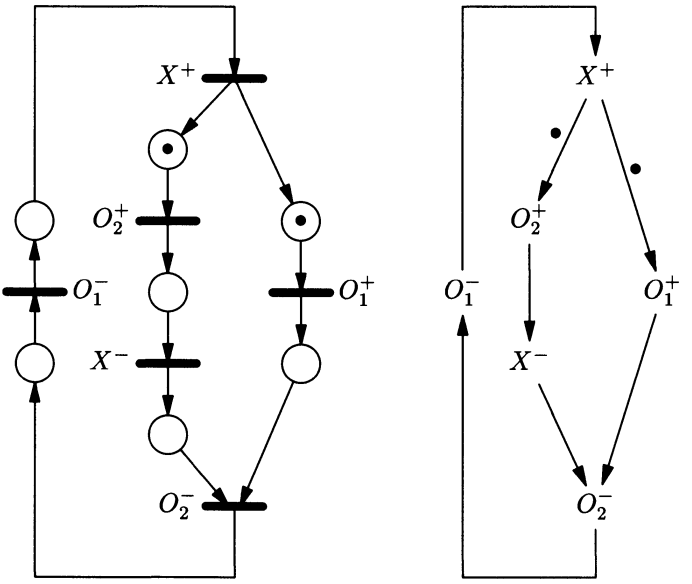


FIGURE 15.8. Illustrating marked graphs: (a) a marked graph; (b) STG/MG.

The simplest major class of STGs is the class STG/MG corresponding to Petri nets that are marked graphs [64], where a *marked graph* is a Petri net in which each place has at most one input transition and at most one output transition. In such a graph, tokens can be removed from a place only by firing its one output transition. Therefore, once a transition is enabled, it can be disabled only by firing. Consequently, choice cannot be modeled, where by choice we mean a situation in which either event A or event B, but not both, can occur.

Consider the marked graph of Figure 15.8(a). The following conventions are used when such graphs are treated as STGs. The transition labels indicate not just the signal names, but also the transition types, either rising $(+)$ or falling $(-)$. Thus, when a transition labeled X^+ (respectively X^-) fires, the signal X changes from 0 to 1 (respectively from 1 to 0). Transitions on input signals are also distinguished by underlining; however, we need not use this notation, since our inputs are called X_i and our outputs are O_j.

In the graphical representation of an STG, a labeled transition is replaced by its label, and places with one input and one output are omitted. Tokens in places that are so omitted are placed on the corresponding edges. Thus, the Petri net (here also a marked graph) of Figure 15.8(a) is redrawn as an STG in Figure 15.8(b).

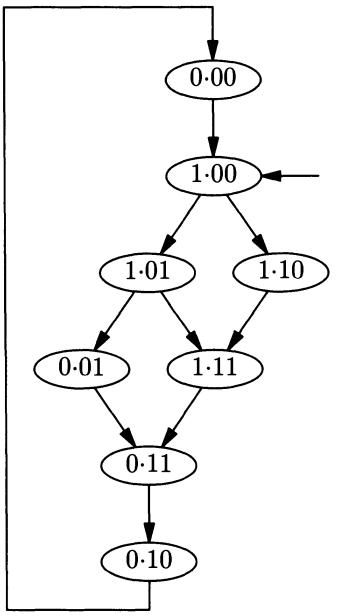


FIGURE 15.9. Illustrating marked graphs, continued: state graph.

State graphs can be associated with STGs in a manner similar to the one we have described for I-nets. The initial state corresponding to the marking shown in Figure 15.8(b) is $X{\cdot}O_1O_2 = 1{\cdot}00$. The state graph of this STG is shown in Figure 15.9.

To permit the modeling of choice, STGs can be extended in two ways. In an *input-choice STG* (*STG/IC*), places are allowed to have multiple input and output transitions; however, if two transitions share the same input place, then they cannot have any other input places, and they must be labeled by input signals. These STGs are also known as *free-choice* STGs. An example of an STG/IC is given in Figure 15.10(a). The transitions labeled X_1^+ and X_2^+ share a place. Note that we are not allowed to change either of these labels to an output label. Also, we are not permitted to add another edge leading from another place to X_1^+ or X_2^+. A *non-input-choice STG* (*STG/NC*) allows all of the constructs of STG/ICs, as well as "non-input" choice. As shown in Figure 15.10(b), two output-signal or internal-signal transitions can share a common input place, but must then have no other input places. Moreover, the model is extended by the addition of labels on the outgoing edges of places with choice. The label for an edge leaving a given place with choice is either a signal name (C in the figure) or its complement ($\overline{C}$). The transition reached by an edge labeled C can fire only if $C = 1$. In an STG/NC we must ensure that, when a place with non-input choice has a token, exactly one of the outgoing edge labels is 1,

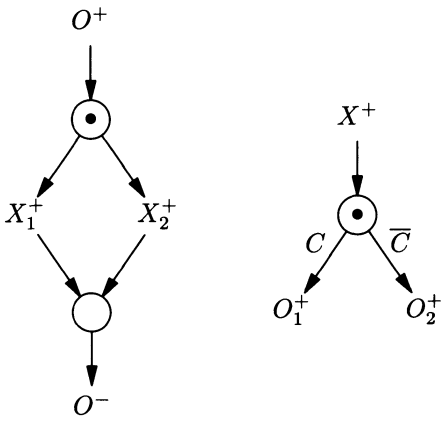


FIGURE 15.10. Choice in STGs: (a) input choice; (b) non-input choice.

and none of the edge signals can change before the choice is made. Thus, it is always clear which of the two transitions should fire. Also, input-signal transitions are not allowed as part of a non-input choice, except as edge labels, though input choice is allowed in the form described for STG/ICs.

To design useful circuits from STGs we often impose some restrictions on the STGs. An STG is *live* if it is possible to fire every transition from every reachable marking. The STG in Figure 15.11(a) is not live because once transition O_1^+ has fired it can never fire again. An STG is *safe* if no place or edge can ever contain more than one token. The STG in Figure 15.11(b) is not safe [145], because the edge from X_2^+ to O_1^+ has two tokens after the firing sequence $O_2^+, X_2^-, O_2^-, X_2^+$. An STG is *persistent* if for each edge $A^* \rightarrow B^*$, where S^* denotes either S^+ or S^-, there must be other edges that ensure that B^* fires before the opposite transition of A^*. The STG in Figure 15.11(c) is not persistent, since there is an edge $X_1^+ \rightarrow O_2^+$, yet X_1^- can fire before O_2^+ does. Note that input signals to an STG are generally not required to satisfy the persistency condition in the following sense. If $A^* \rightarrow B^*$, where B is an input, we do not care if the transition opposite to A^* fires before B^*; it is assumed that the environment guarantees the persistency of B^*. An STG has a *consistent state assignment* if the transitions of a signal S strictly alternate between S^+ and S^-, i.e., the STG attempts to neither raise a signal that is already high nor lower a signal that is already low. Consistent state assignment is a necessary condition for realizability. The STG in Figure 15.11(d) does not have a consistent state assignment, since two O^+ transitions can occur without an intervening O^-. An STG has a *unique state assignment* if no two of its markings have identical values for all the signals. Note that the value of a signal can be determined by

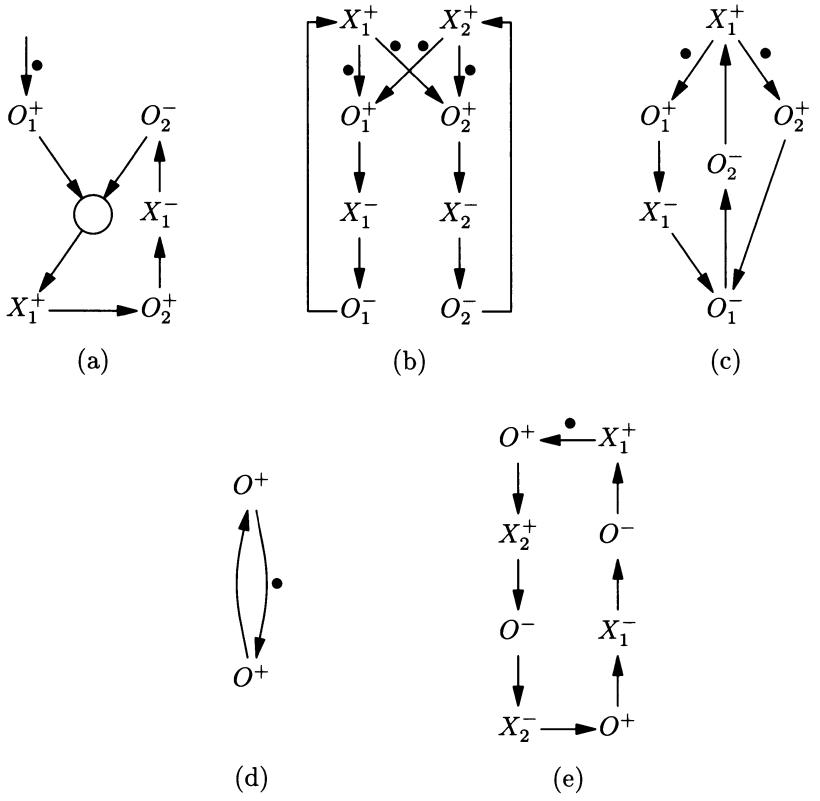


FIGURE 15.11. STGs violating various properties: (a) liveness; (b) safety; (c) persistency; (d) consistency of state assignment; (e) uniqueness of state assignment and single-cycle transitions.

looking at the next reachable transition of that signal. If the next transition is S^+ (respectively S^-), the signal's current value is 0 (respectively 1). The STG in Figure 15.11(e) does not have a unique state assignment, since the signal values are $X_1X_2{\cdot}O = 10{\cdot}0$ both in the initial marking and in the marking with a token on the edge $X_2^- \rightarrow O^+$. Finally, an STG has *single-cycle transitions* if each signal appears in exactly one rising and exactly one falling transition. The STG in Figure 15.11(e) does not satisfy this condition, since both O^+ and O^- appear twice.

Except for the consistent state assignment property, the restrictions listed above are not necessary for realizability. However, they often lead to efficient circuit generation. As we will see, efficient algorithms for ensuring most of these requirements have been developed.

The substantial restrictions imposed on the allowable Petri-net constructs—the restricted classes STG/MG, STG/IC, and STG/NC, as well

as the conditions illustrated in Figure 15.11—are justified only if they lead to better design algorithms. We have seen that I-nets constitute an example of automatic generation of circuits from rather general Petri-net specifications, as long as we are willing to pay the cost of potentially exponential-size state graphs. The same technique, of generating the underlying state graph from a Petri net, then finding the implied functions, and realizing the functions represented in it, applies also to STGs. Moreover, techniques have been developed to implement STGs without exponential state blowup.

One of the most intuitive approaches is *contraction* [33]. From an STG/IC we generate a circuit that is live, safe, persistent, and has a consistent unique state assignment, and single-cycle transitions. We do this by removing from the STG/IC all the transitions that do not directly impact the signal being synthesized. For example, consider the STG of Figure 15.12(a) and its state diagram shown in Figure 15.12(b). The incoming arrow at the top of Figure 15.12(b) designates the initial state, which is $X_1X_2{\cdot}O_1O_2 =$ 00·00. To save space, the state labels are not shown; however, they are easily reconstructed.

To reduce the exponential blowup in the size of the state diagram, we will synthesize a circuit for each of the two outputs from contracted STGs, as described below. First, we synthesize the logic for output O_1; in that case,

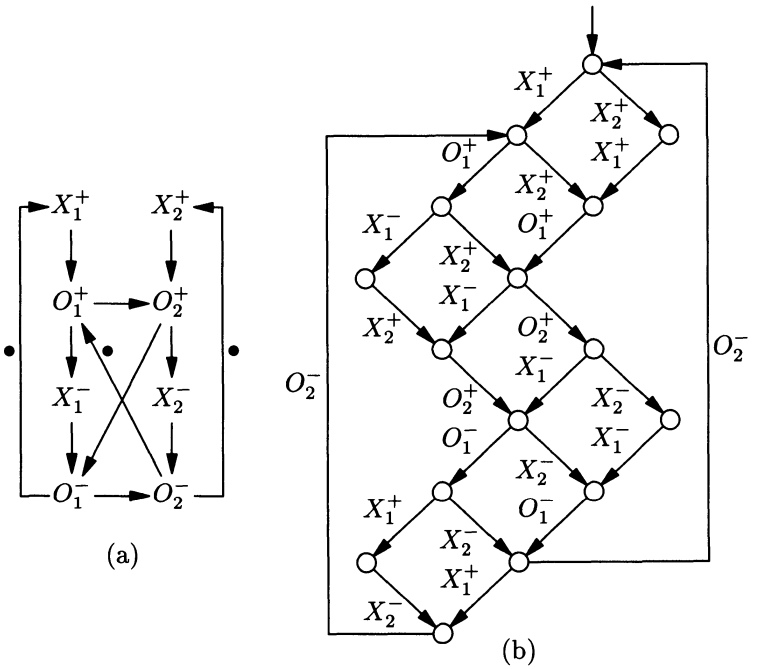


FIGURE 15.12. Contraction: (a) STG A; (b) state diagram of A.

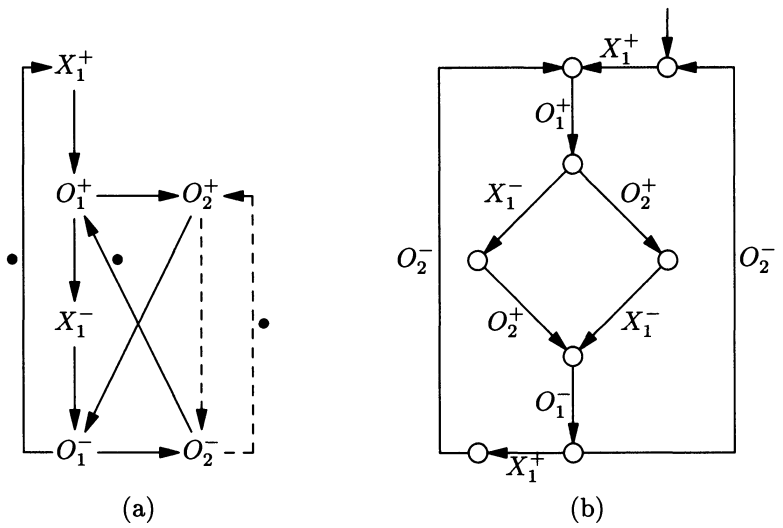


FIGURE 15.13. Contraction: (a) contracted STG for O_1; (b) state diagram.

output O_2 can be ignored, except to the extent that it affects O_1. Since the transitions X_2^+ and X_2^- do not affect O_1 directly, they are removed. Because there are some sequencing constraints between O_1 and O_2, the transitions O_2^+ and O_2^- are kept; they are shown connected by dashed edges in Figure 15.13(a). The state diagram corresponding to the contracted STG is shown in Figure 15.13(b). Note that there are only eight states, while the original state diagram has 16.

Next, we contract the original STG with respect to O_2. The result is shown in Figure 15.14(a), along with the corresponding state diagram. Again, we have eight states instead of 16. In more complex examples one can expect more significant savings. The complete circuit implementation for the original STG consists of the network of components obtained by the contractions.

An algorithm for converting an STG/NC into an STG/IC has been developed in [33]. This method both requires and preserves all of the restrictions listed above, except that single-cycle transitions are no longer guaranteed. In cases where transitions remain single-cycle, the contraction strategy can be used to implement STG/NC circuits efficiently.

Several other researchers have developed efficient algorithms for STG transformation and synthesis. As described above, the algorithm of [33] requires that the STG to be synthesized not only be persistent, but also obey several other restrictions, including unique state assignment. In [137] algorithms are included to transform a live STG/MG with single-cycle transitions to achieve both persistency and unique state encoding. In [87] a method is described for transforming a live, safe STG/MG with single-

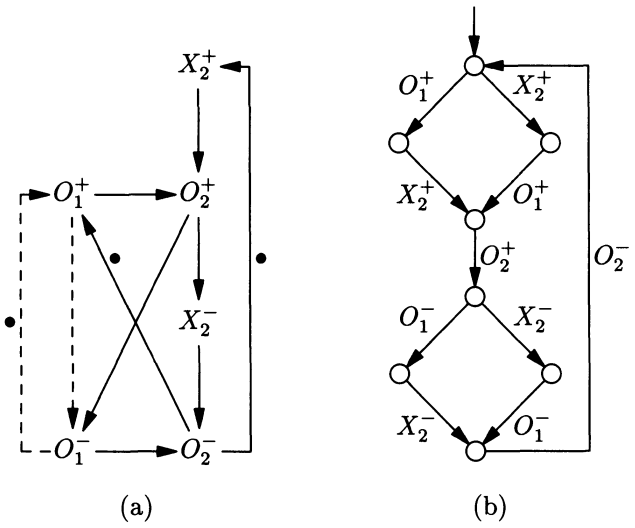


FIGURE 15.14. Contraction: (a) contracted STG for O_2; (b) state diagram.

cycle transitions to a persistent STG with unique state assignment, and then generating an implementation by an efficient algorithm that requires no state graph construction. This theory is extended in [86] to test for "realizable" state encoding in live, safe STG/ICs. Instead of requiring that no two states have identical values for all signals—as is the case with unique state assignment—realizable state encoding allows two states to have the same signal values, as long as the same non-input transitions can occur from both states. However, the input transitions can differ, since the environment is assumed to have additional information to resolve the ambiguity.

If the exponential cost of state graph construction can be tolerated, several other algorithms are of interest. In [3] speed-independent circuits are designed from state graphs using simple gates, such as ANDs, ORs, and C-ELEMENTs. This removes the difficulty in [33] of computing an arbitrarily complex function without internal hazards. The gates used in [3] may have high fan-in; this problem is addressed in [4]. Instead of speed-independence, [78] presents an approach to implementing live, safe STG/ICs with unique state assignments in a bounded-delay model. Note that these STGs do not have to be persistent and can have non-single-cycle transitions. These circuits have a structure similar to the Huffman circuits described earlier, with sum-of-products expressions implemented with AND and OR gates. These sum-of-products circuits are used by RS flip-flops, which are assumed to be somewhat immune to dynamic hazards on their inputs. Delays are added to avoid some hazards; [80] uses a linear-programming algorithm for optimal insertion of delays. The problem of delay-fault testing is addressed in [75]. Both [79] and [138] handle state-variable insertion, the former per-

mitting live, safe STG/ICs, the latter allowing any state graph that is finite, connected, and has a consistent state assignment. Finally, we refer the interested reader to [81] for a comprehensive treatment of the STG design methodology.

15.7 Change Diagrams

Change diagrams (*CDs*) [76] are similar to STGs. Like an STG, a CD has vertices labeled by signal transitions and edges that define the allowed sequences of transition firings. However, as shown in Figure 15.15, there are two types of edges: "strong-precedence" (solid lines) and "weak-precedence" (dashed lines). Furthermore, strong precedence edges can be either ordinary or "disengageable" (solid lines with crosses). Like an STG, a CD has an initial marking of tokens on its edges. Note the incoming edge to the vertex labeled X_1^+. It is assumed that such an edge has a token initially, and this token is used only to start the operation of the CD.

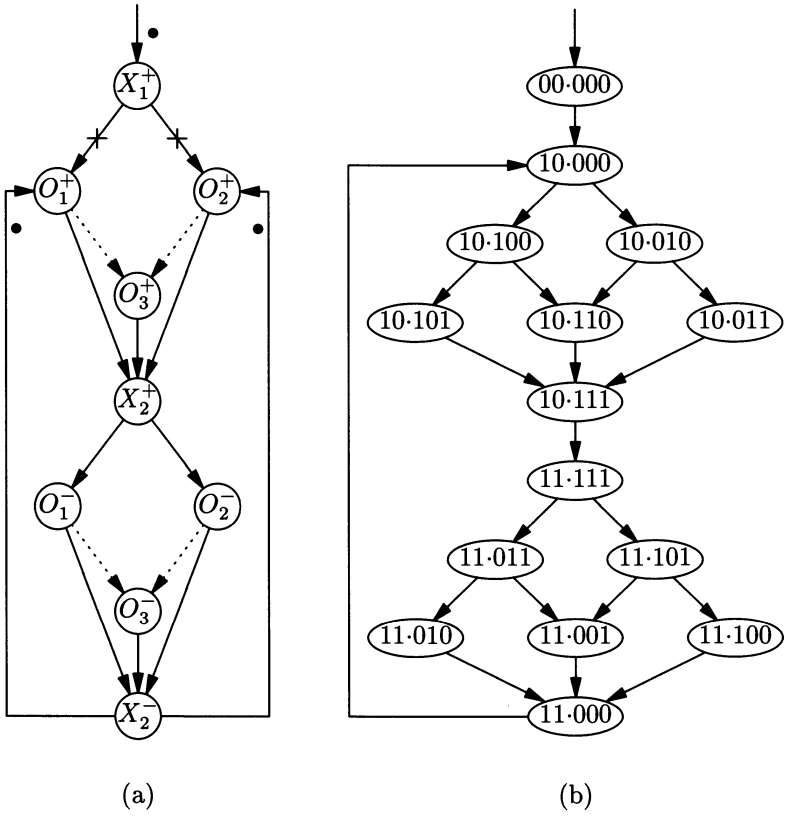


FIGURE 15.15. Illustrating change diagrams: (a) a CD; (b) its state graph.

The input edges to any node are either all strong-precedence or all weak-precedence. Strong edges can be thought of as AND-edges, since a transition with strong-precedence input edges cannot fire until *all* of these edges have tokens. Thus, strong-precedence edges are like the edges in STGs. Weak edges are OR-edges, in that a transition with weak-precedence input edges can fire whenever *any one* of these edges has a token. When a transition fires, a token is removed from each of its input edges and a token is placed on each of its output edges. Since a transition with weak-precedence input edges can fire before all these edges have tokens, tokenless edges may have negative tokens assigned to them; this is indicated by small open circles. When a (positive) token arrives at an edge with a negative token, cancellation occurs. Disengageable edges can fire only once; after such an edge fires, it is considered removed from the CD. Thus, disengageable edges can be used to connect an initial, nonrepeating set of transitions to an infinitely repeating cycle.

We illustrate the operation of a CD by the example of Figure 15.15. Initially, only transition X_1^+ is enabled. When it fires, X_1 will remain 1 for as long as the CD is in operation. When the CD is no longer needed, X_1 will be reset by some means that are not part of the model. After X_1^+ fires, the two disengageable edges have one token each. We see that O_1^+ and O_2^+ are both enabled. If O_1^+ fires and then O_2^+, the firing sequence can proceed just as it would in an STG. Thus, O_3^+ can fire and remove both tokens from the dashed edges. The transition X_2^+ is now enabled, and so on. On the other hand, consider the situation after O_1^+ has fired, but O_2^+ has not. Because O_3^+ has weak-precedence input edges, it can now fire, removing the token on the edge $O_1^+ \rightarrow O_3^+$ and creating a negative token on the edge $O_2^+ \rightarrow O_3^+$. Now, when O_2^+ fires, a positive token is introduced on the edge $O_2^+ \rightarrow O_3^+$, canceling the negative token. The CD specifies that O_3^+ may fire after O_1^+ has fired, or O_2^+ has fired, or both O_1^+ and O_2^+ have fired. But, in all these cases, O_3^+ should fire only once. If negative tokens were not used, the sequence $(X_1^+, O_1^+, O_3^+, O_2^+)$ could be followed by another firing of O_3^+. The complete state graph corresponding to the CD of Figure 15.15(a) is shown in Figure 15.15(b). The state is $X_1X_2 \cdot O_1O_2O_3$, and the edge tags are not shown.

Disengageable edges are an improvement over the STG model, since they allow the modeling of initial, nonrepeating transitions; this is not possible in STGs. Also, many of the restrictions that are placed on STGs are not present in CDs. Thus, liveness—which requires all transitions to potentially fire infinitely often—is replaced by the requirement that all transitions must be able to fire at least once. Persistency—which requires that the opposite transition of a given transition t not fire until all transitions enabled by t have fired—is replaced by the requirement that an enabled transition can only be disabled by its firing. Other constraints include certain connectedness properties and alternation of positive and negative transitions on every signal. All of these CD correctness constraints can be checked in

time polynomial in the size of the CD. The method used to verify that these conditions hold is to unroll a cyclic CD into an acyclic, infinite CD. It can be shown that only the first n periods in this unrolling need to be considered, where n is the number of vertices in the original CD.

Unfortunately, the CD model has some disadvantages. For example, it cannot represent a XOR gate. Weak-precedence edges can model the fact that either input to a gate can cause its output to change, but then the other input has to change twice before it can have an effect on the output, because the first change serves only to cancel the negative token. Also, CDs are unable to specify the type of choice where two different transitions can remove a token from a shared place.

15.8 Protocols in DI Circuits

As has been stated earlier, delay-insensitive circuit design assumes that the delays in both components and wires are unbounded. It should be no surprise that this assumption has a great impact on the resulting circuit structure. In bounded-delay models, we assume that, given enough time after an input change, a circuit will reach a stable state, and a new input change can then be safely applied. With a delay-insensitive model, no matter how long one waits, there is no guarantee that the input change has been properly received and processed by the circuit. This forces the receiver of a signal to inform the sender, by an *acknowledge* signal, that the information has been received. The sender, in turn, is required to wait until it gets the acknowledge signal before sending a new signal.

In the so-called *two-phase handshaking* protocol, a request transition is sent from the sender to the receiver, and then an acknowledge transition is returned by the receiver to the sender. Assuming the request (r) and acknowledge (a) wires are both 0 initially, they both become 1 after one cycle of the protocol. Thus the first cycle results in the following $r{\cdot}a$ states: 0·0, 1·0, 1·1. The next cycle returns both wires to low: 1·1, 0·1, 0·0.

Some methodologies use a *four-phase handshaking* protocol in order to return the request and acknowledge wires to their original values after every cycle. Thus one cycle results in the following $r{\cdot}a$ states: 0·0, 1·0, 1·1, 0·1, 0·0. Although four-phase handshaking appears to require twice as much time, because twice as many transitions are sent, in most cases computation time dominates communication time. In addition, the second half of the four-phase handshaking can often be done concurrently with computations, thus improving performance. Finally, since only a rising edge initiates a communication, four-phase circuit structures can be simpler than their two-phase counterparts. Altogether, these properties makes four-phase handshaking competitive.

Delay-insensitive design also requires a new way of passing data. In synchronous circuits, the value of a wire is assumed to be correct by a given time (for example, when the clock pulse arrives), and can be safely used

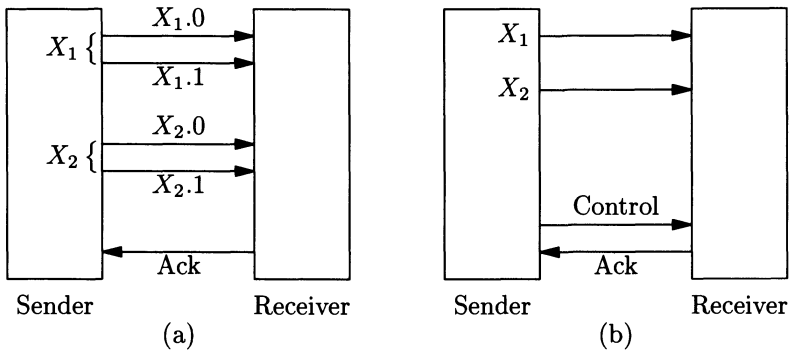


FIGURE 15.16. Data transfer: (a) transition signaling, (b) bundled-data method.

at that time. In delay-insensitive circuits, there is no guarantee that a wire will reach its proper value by any specific time. Thus, a transition must be sent to inform the receiver of the new value. With transition signaling, a bit of data cannot be transferred by a single wire, because the opposite of a transition—no transition—cannot be distinguished from a delayed transition. Thus two wires are required from sender to receiver to transfer a data bit, with the wire on which a transition occurs determining the value being transmitted. For example, the two wires could be labeled $X.0$ and $X.1$, with a transition on $X.0$ indicating the data bit is a 0, and a transition on $X.1$ indicating the data bit is 1; see Figure 15.16(a). Other variations on this theme are possible [139], but are beyond the scope of this discussion. Both two-phase and four-phase protocols can be implemented with a two-wire scheme. A two-phase communication requires a single transition on one of the two wires, whereas a four-phase protocol requires two. In both cases an additional wire is required to send acknowledgments back to the sender, though only one such wire is needed for multi-bit communications.

The so-called *bundled data* method of data transfer allows fewer wires to be used, but violates the delay-insensitive model. It allows a single wire for each data bit, and one extra control line for each data word. It is assumed that the delay in the extra control wire is longer than the delay in each of the data wires. Thus, when a transition appears on its control wire, the receiver knows that the values on the data lines have already arrived; see Figure 15.16(b).

As we have seen, the assumption that element and wire delays are unbounded leads to complications in the signaling protocols. However, these methodologies do overcome some of the problems found in bounded-delay models. The unbounded-delay assumption also has the desirable effect of separating circuit correctness concerns from concerns about specific delay values. Consequently, timing improvements, such as delay optimization by transistor sizing, can be applied without affecting circuit correctness.

In Section 15.5 we have described a methodology for designing delay-insensitive modules. Once a set of such modules is available, the design of asynchronous networks consisting of such modules is free of timing constraints. Of course, the modules have to be properly operated. For example, a JOIN element cannot be used in the same place as a MERGE, since the first requires two input transitions between any two output transitions, while the second requires only one. However, such restrictions are much simpler than those of most other methodologies, and it is usually clear which module needs to be used from the functionality required.

In the next three sections we describe techniques for designing delay-insensitive networks, assuming that suitable modules are available.

15.9 Ebergen's Trace Theory Method

A method for delay-insensitive circuit design has been proposed by Ebergen [46, 47]. The method uses a unified model for both module specification and circuit design, and is based on trace theory, a model similar to regular expressions. In the following, we do not use Ebergen's notation, but one that is closer to the notation in this book.

A *trace structure* is a triple $T = \langle \mathcal{X}, \mathcal{O}, L \rangle$, where $\mathcal{X}$ is the *input alphabet* of T, $\mathcal{O}$ is the *output alphabet* of T, and the language $L \subseteq (\mathcal{X} \cup \mathcal{O})^*$, called the *trace set* of T, describe a desired circuit functionality. The set $\mathcal{A} = \mathcal{X} \cup \mathcal{O}$ is the *alphabet* of T. Each symbol in the alphabet corresponds to a signal in the circuit, and the appearance of the symbol in a word represents a transition on that signal.

The following operations are defined on trace structures. Given trace structures $T = \langle \mathcal{X}, \mathcal{O}, L \rangle$ and $T' = \langle \mathcal{X}', \mathcal{O}', L' \rangle$, we define

- *concatenation:* $TT' = \langle \mathcal{X} \cup \mathcal{X}', \mathcal{O} \cup \mathcal{O}', LL' \rangle$;
- *union:* $T \cup T' = \langle \mathcal{X} \cup \mathcal{X}', \mathcal{O} \cup \mathcal{O}', L \cup L' \rangle$;
- *star:* $T^* = \langle \mathcal{X}, \mathcal{O}, L^* \rangle$;
- *pref:* $pref T = \langle \mathcal{X}, \mathcal{O}, pref L \rangle$, where $pref L$ is the set of all prefixes of words in L;
- *restriction to a subalphabet*, also called *projection*:

$$T\dagger_{\mathcal{B}} = \langle \mathcal{X} \cap \mathcal{B}, \mathcal{O} \cap \mathcal{B}, \{w\dagger_{\mathcal{B}} \mid w \in L\} \rangle,$$

  where $w\dagger_{\mathcal{B}}$ is w with all the letters that are not in $\mathcal{B}$ removed;

- and *weave*: $T \| T' = \langle \mathcal{X} \cup \mathcal{X}', \mathcal{O} \cup \mathcal{O}', L_{\|} \rangle$, where

$$L_{\|} = \{w \in (\mathcal{A} \cup \mathcal{A}')^* \mid w\dagger_{\mathcal{A}} \in L \text{ and } w\dagger_{\mathcal{A}'} \in L' \rangle.$$

The operations concatenation, union, and star on languages are the same as those used in regular expressions. We have already used the prefix operation and the restriction to a subalphabet in Chapter 11. Restriction to a subalphabet is an important tool for hierarchical construction of circuits. When a circuit is constructed from components, some symbols are used internally to interconnect the components, but are not associated with any external symbols. Such symbols become *internal* symbols, and are removed when the external properties of the network are described. The weave represents "synchronization on common symbols." To illustrate this, consider the trace structures $T = \langle\{a\},\{c\},\{ac\}\rangle$ and $T' = \langle\{b\},\{c\},\{bc\}\rangle$. The weave of T and T' is then $T\|T' = \langle\{a,b\},\{c\},\{abc,bac\}\rangle$. Note that the traces ac from T and bc from T' are both consistent with the trace abc of $T\|T'$, and that they are synchronized by the occurrence of c.

To have a convenient finite representation of the set of all possible input/output sequences, Ebergen uses the language of *commands*. Commands are similar to regular expressions. The *atomic commands* are: $\emptyset$, ε, $a?$, $a!$, and a, for each $a \in \mathcal{A}$. They represent the trace structures: $\langle\emptyset,\emptyset,\emptyset\rangle$, $\langle\emptyset,\emptyset,\{\varepsilon\}\rangle$, $\langle\{a\},\emptyset,\{a\}\rangle$, $\langle\emptyset,\{a\},\{a\}\rangle$, and $\langle\{a\},\{a\},\{a\}\rangle$, respectively. General commands are constructed from atomic commands with the use of the trace-structure operators above. We use the following order of operator precedence to simplify the notation: star, followed by *pref*, followed by concatenation, followed by union and weave at the lowest level.

We illustrate the command language with the JOIN element of Figure 15.5(a). The JOIN can be described by the command $pref(X_1?O!\|X_2?O!)^*$. The concatenation operations enforce that an input precedes the output. The fact that output O is shared between two commands in the weave synchronizes them, ensuring that both inputs occur before the output can occur. The star allows the JOIN to repeat this protocol an arbitrary number of times. The *pref* permits any prefix of the complete protocol to be defined as a valid trace. Note that the JOIN element can also be described by the commands $pref((X_1?\|X_2?)O!)^*$ and $pref(X_1?X_2?O! \cup X_2?X_1?O!)^*$.

As we have stated earlier, one of the advantages of the trace methodology is that both the circuit to be synthesized and the basic modules used to implement it are represented in the same model. Most of the basic elements used in this methodology are shown in Figure 15.17. WIRE and IWIRE represent connections between two terminals. In a WIRE the environment must produce the first transition, whereas in the IWIRE the first transition comes from the component. Otherwise, the two components are very similar in that their input and output transitions alternate. The WIRE and IWIRE are specified by the commands $pref(X?O!)^*$ and $pref(O!X?)^*$, respectively. The FORK is self-explanatory; it can be described by the command $pref(X?(O_1!\|O_2!))^*$. We have already discussed the JOIN above. The TOGGLE element, introduced in Chapter 13, can be specified by the command $pref(X?O_1!X?O_2!)^*$. In Figure 15.17, the small dot indicates the output that changes in response to the first input transition. We have dis-

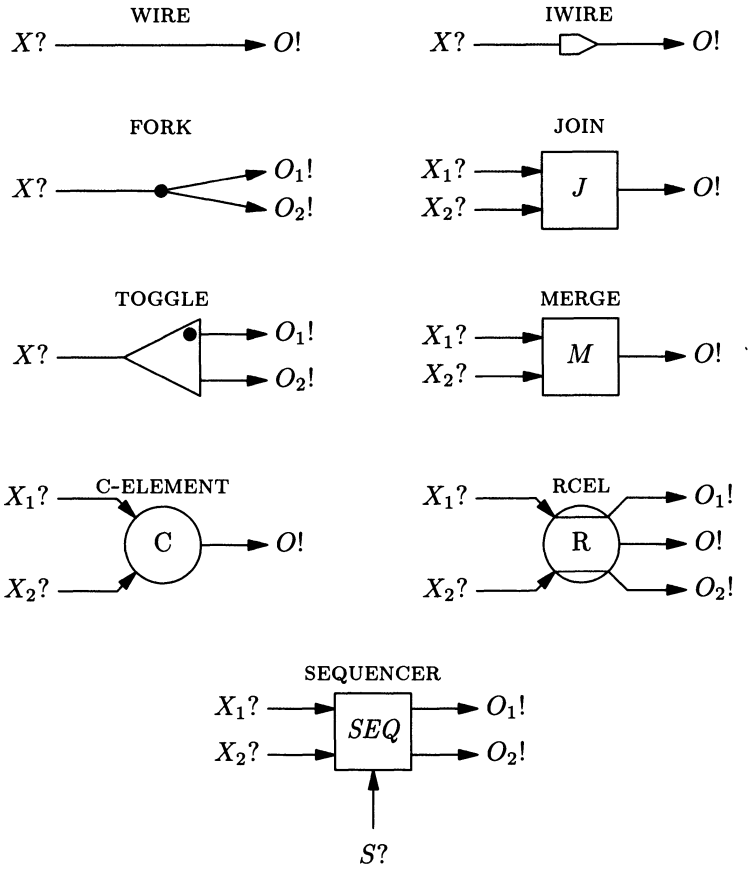


FIGURE 15.17. Basic elements.

cussed the MERGE earlier in connection with I-nets; it can be specified by $pref((X_1?\cup X_2?)O!)^*$. Notice that the trace theory representations of components are more compact than those of our behavior model, since only transitions are recorded in trace theory, whereas we also include the signal levels in behaviors. Compare, for example, the command specifications of the JOIN and TOGGLE elements with the corresponding behaviors in Chapter 11.

The C-ELEMENT is similar to the JOIN, except for the fact that an input transition can be "withdrawn" by a second transition on the same input. The C-ELEMENT can be specified by the command[4]

$$pref((X_1?)^2 \cup (X_2?)^2 \cup (X_1?\|X_2?)O!)^*.$$

[4]Recall that w^2 is a shorthand for ww.

Note that the trace set of the JOIN is a subset of the trace set of the C-ELEMENT. The astute reader will note that the C-ELEMENT is not delay-insensitive. This issue is discussed later. The RCEL is a delay-insensitive replacement for the C-ELEMENT. It has the same functionality as the C-ELEMENT but also acknowledges all inputs to X_1 and X_2 by outputs O_1 and O_2, respectively. It can be specified by

$$pref((X_1?O_1!)^2 \cup (X_2?O_2!)^2 \cup (X_1?(O_1!\|O!) \parallel X_2?(O_2!\|O!)))^*.$$

The SEQUENCER is a mutual exclusion element that passes a single input transition from X_1 to O_1 or from X_2 to O_2 for each transition on S. Thus, a single output transition is generated for each S transition, and there must always be at least as many X_1 transitions as O_1 transitions, and at least as many X_2 transitions as O_2 transitions. The following is a specification for the SEQUENCER:

$$pref(X_1?O_1!)^* \parallel pref(X_2?O_2!)^* \parallel pref(S?(O_1! \cup O_2!))^*.$$

To synthesize a circuit using trace theory, we first specify the desired behavior by a set of traces denoting its input-output sequences. Note that trace sets do not necessarily correspond to delay-insensitive behaviors. For example, $pref(O!O!)$ has an output hazard. To prevent this, Ebergen has proposed a test for delay-insensitivity that is similar to the foam rubber wrapper property; he also defined a command grammar that generates only delay-insensitive trace sets. Note that, although it is conjectured that this grammar cannot represent all possible delay-insensitive circuits (the RCEL has not been successfully represented [46]), it seems to handle most circuits. Once the desired circuit is specified by a delay-insensitive trace structure, it can be realized using the components shown in Figure 15.17 with the aid of a syntax-directed translation scheme. In this approach, a complex trace structure is decomposed into a network of several simpler trace structures. Successive applications of such decomposition eventually yield a set of trace structures directly implementable by the basic components.

To illustrate this approach, we describe a half-adder that would normally have 1-bit inputs X and Y and 1-bit outputs S (sum) and C (carry). Each of these signals is represented in the two-wire scheme; thus, X is represented by $X.0$ and $X.1$, etc. The half-adder can be specified by the trace structure

$$\begin{aligned} pref((&X.0?\|Y.0?)(S.0!\|C.0!) \cup \\ &(X.0?\|Y.1?)(S.1!\|C.0!) \cup \\ &(X.1?\|Y.0?)(S.1!\|C.0!) \cup \\ &(X.1?\|Y.1?)(S.0!\|C.1!))^*. \end{aligned}$$

In this specification, both the two inputs and the two outputs occur in parallel. We can decompose this trace structure into a weave $A\|B$ of two trace structures A and B (defined below), one without parallel outputs

and one without parallel inputs. This is done by introducing the auxiliary signals $q_0, \ldots, q_3$ as follows:

$$A = pref((X.0?\|Y.0?)q_0! \cup (X.0?\|Y.1?)q_1! \cup (X.1?\|Y.0?)q_2! \cup (X.1?\|Y.1?)q_3!)^*;$$

$$B = pref(q_0?(S.0!\|C.0!) \cup q_1?(S.1!\|C.0!) \cup q_2?(S.1!\|C.0!) \cup q_3?(S.0!\|C.1!))^*.$$

The command A is recognized as a specification for a 2×2 JOIN (also called a "decision wait" element); see the left component of Figure 15.18. Informally, the 2×2 JOIN has two "horizontal" inputs $X.0?$ and $X.1?$ and two "vertical" inputs $Y.0?$ and $Y.1?$. It expects one horizontal and one vertical input. It has four outputs $q_0!, \ldots, q_3!$; upon receiving $X.i?$ and $Y.j?$, the 2×2 JOIN produces output $q_k!$, where k is the decimal integer represented by the binary pair (i, j). The 2×2 JOIN can be decomposed further into simpler components; this decomposition is nontrivial, and we refer the reader to [46].

It can be verified that the command B can be implemented by three MERGE elements, as shown in Figure 15.18.

While trace structures provide a theoretical basis for the design of delay-insensitive circuits, they have some disadvantages. The first has been alluded to earlier—one of the elements commonly used in the synthesis procedure (the C-ELEMENT) is not delay-insensitive, since two transitions on a single input wire may occur without an intervening output. One solution proposed by Ebergen is to replace this with an RCEL, which is similar to a C-ELEMENT except that it has two extra outputs to acknowledge all input transitions. It is not clear whether the added complexity of this element would be justified in practical designs. The second solution is to add *isochronic forks* to the model; these are forks in which the difference in the

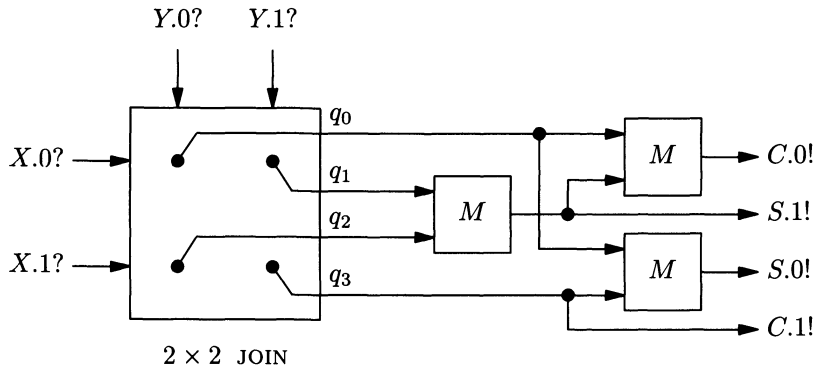


FIGURE 15.18. A decomposition of a half-adder.

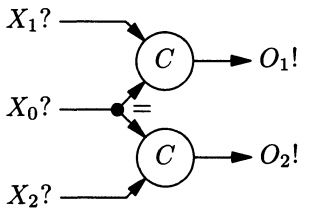


FIGURE 15.19. Illustrating the use of an isochronic fork.

delays between the two output branches is negligible. For example, consider the command

$$pref(((X_0?\|X_1?)O_1!)^2 \cup ((X_0?\|X_2?)O_2!)^2)^*.$$

It can be implemented by a relatively complex circuit consisting of several components [46]. It can also be implemented by two C-ELEMENTs and a fork, as shown in Figure 15.19. The fork must be isochronic; otherwise, the transition on the X_0? wire going to the C-ELEMENT that has not received its second input would not be acknowledged, thus creating the possibility of a hazard. With isochronic forks, we can safely use C-ELEMENTs; however, circuits so designed are no longer fully delay-insensitive. We return to isochronic forks in Section 15.10.

The second issue is the low-level nature of the command language, which makes command specifications difficult to understand. Most of this difficulty can be attributed to the weave operator. The reader should note, however, that without the use of some operator like the weave, the size of a specification would be exponentially bigger than the size of the corresponding specification with weaves. Also, any parallel operator is likely to be seen as difficult initially. In any case, the specification of a large circuit, such as a microprocessor, at the level of individual transitions is rather impractical. Adding a more understandable high-level language above the trace structure methodology would significantly increase its attractiveness.

15.10 Compilation of Communicating Processes

Martin's methodology [89, 91] starts with a high-level specification in the source language of *communicating hardware processes (CHP)*, similar to Hoare's communicating sequential processes [62] and Dijkstra's guarded commands [43]. The language describes a behavior by specifying the required sequences of communications. Several large asynchronous circuits have been designed and implemented using this technique [89, 91].

A program in CHP consists of a collection of concurrent processes communicating over named channels. The channels have no storage capacity; thus communication over a channel serves as a synchronization. Each

process is described in terms of simple programming language constructs that include variables, assignments, conditional branching, looping, and sequencing. The only basic data type is Boolean and the only type constructors are records and (fixed-size) arrays.

For a Boolean variable a, the assignment $a := 1$ ($a := 0$) is abbreviated as $a\uparrow$ ($a\downarrow$). There are two general composition operators: the *sequential operator* ";" and the *parallel operator* "||". Intuitively, $S_1; S_2$ is interpreted as "first execute S_1 and then S_2," whereas $S_1 \| S_2$ is viewed as "execute S_1 and S_2 concurrently." There are two types of *selection* operators: deterministic and nondeterministic. The syntax is $[\ (G_1 \rightarrow S_1) \,[\!]\, \cdots \,[\!]\, (G_n \rightarrow S_n)\ ]$ for the deterministic choice, and $[\ (G_1 \rightarrow S_1) \mid \cdots \mid (G_n \rightarrow S_n)\ ]$ for the nondeterministic choice. The G_i's are called *guards* and $G_i \rightarrow S_i$ is a *guarded command.* If a guard evaluates to true in the current state, the guarded statement may be executed. If we can guarantee by some means (for example, by environmental assumptions) that at most one guard is true in any state, then deterministic choice can be used. Otherwise, nondeterministic choice must be specified. Consequently, the introduction of arbitration is explicit in the high-level specification. We often write $[G]$, instead of $[G \rightarrow \textbf{skip}]$, where "skip" represents no operation. The statement $[G]$ simply stands for "wait until G holds." The last programming construct is *repetition*, which is written as $*[S]$, and denotes an infinite iteration of S.

Processes communicate with each other by communication commands on *ports*. A port on one process is paired with a port on another process to form a *channel.* There are one-to-one, one-to-many, and many-to-many channels. Normally, communications on channels serve as synchronizations. For example, a reader is blocked until a sender sends a message on their common channel. However, a *probe* command can be used by a process to determine, without blocking, whether there are data to be read. A probe of port A is written as $\overline{A}$. Finally, the communication of data over channels is specified by input and output commands. An *input command* on port A is written as $A?$, whereas an *output command* on port B is written as $B!$.

Although the language constructs are more primitive than those used in most programming languages, they provide a higher-level abstraction than many of the other approaches for describing asynchronous circuits. Furthermore, they appear to provide enough flexibility to handle a large class of circuits [89, 91]. Unfortunately, although superficially similar to CSP and guarded commands, CHP does not have a formal semantics. Consequently, the process decomposition and translation rules used in translating a CHP program to an asynchronous circuit are not formally proved to be correct. This is a current drawback of CHP and the associated design method.[5]

[5]This deficiency has been addressed to some extent in [127], where a semantics for a small subset of CHP and the associated transformation techniques has been given. However, more work is needed before all of CHP is properly defined.

The design methodology proceeds as follows: First the abstract specification program is refined using "semantics-preserving" transformations to increase concurrency, etc. For example, a purely sequential specification may be rewritten to introduce pipelining. The program is then rewritten by syntax-directed transformations into a collection of simple statements. Once a desirable intermediate-level program has been derived, the communication primitives of CHP are expanded into handshaking protocols using dual-rail-encoded signals. These protocols are then further refined to "production rules," which are guarded assignments. By imposing certain constraints on the production rules, one can implemented them directly as networks of CMOS transistors. Thus, there is a direct path from a CHP program to a (custom) CMOS circuit. We now illustrate this process by a simple example.

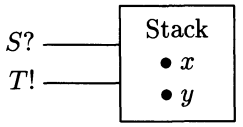


FIGURE 15.20. Abstract stack circuit.

The task is to design a one-bit-wide stack of depth two, as illustrated in Figure 15.20. We do not carry out a complete design, but we do show how parts of the design progress from an abstract program to a transistor circuit. For simplicity, we assume that competing requests for pushing and popping the stack never arise.[6] Note that in this example, the user of the stack must determine whether the stack is empty or full.

The following is a CHP program describing the stack:

$$
\begin{array}{l}
\mathit{Stack} \equiv \textbf{process}\ (S?\ \textbf{bool},\ T!\ \textbf{bool}) \\
\qquad x, y : \textbf{bool} \\
\qquad\qquad *[\,[\,(\overline{S} \rightarrow (y := x; S?x)) \parallel (\overline{T} \rightarrow (T!x; x := y))\,]\,] \\
\qquad \textbf{end}
\end{array}
$$

Intuitively, the CHP program declares a stack process that communicates on two ports: the input port $S?$ and the output port $T!$. It uses the two variables x and y to store the content of the stack. The probe construct is used to determine whether a push, a pop, or no operation is currently requested. If a push is requested, i.e., the probe of S becomes true, the process copies the current value in x to y. It then reads the value to be pushed and stores it in x. The pop works in a similar fashion.

The first step in the compilation process is called *process decomposition* and uses a divide-and-conquer strategy. Complex CHP processes are

[6] The reason for this assumption is to avoid the complexity of arbitration. The methodology deals quite well with this issue, but the general problem is too complicated to discuss here.

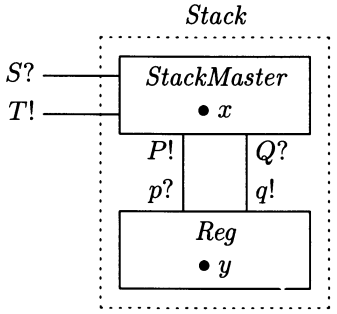


FIGURE 15.21. Stack process decomposed into StackMaster and Reg.

decomposed into smaller processes. By repeatedly applying such transformations, one can derive a collection of processes, each in one of only four distinct formats. This greatly simplifies the remaining steps in the compilation process. In our example, as illustrated in Figure 15.21, we decompose *Stack* into the processes *StackMaster* and *Reg*, which communicate over two new channels $(P!, p?)$ and $(Q?, q!)$. Let

$$\begin{aligned} \mathit{StackMaster} \equiv\ & \textbf{process}(S?\ \textbf{bool}, T!\ \textbf{bool}, Q?\ \textbf{bool}, P!\ \textbf{bool}) \\ & x : \textbf{bool} \\ & \quad *[\,[\,(\overline{S} \rightarrow (P!x; S?x)) \parallel (\overline{T} \rightarrow (T!x; Q?x))\,]\,] \\ & \textbf{end} \end{aligned}$$

and

$$\begin{aligned} \mathit{Reg} \equiv\ & \textbf{process}(p?\ \textbf{bool}, q!\ \textbf{bool}) \\ & y : \textbf{bool} \\ & \quad *[\,[\,(\overline{p} \rightarrow p?y) \parallel (\overline{q} \rightarrow q!y)\,]\,] \\ & \textbf{end} \end{aligned}$$

Altogether, we get

$$\begin{aligned} \mathit{Stack} \equiv\ & \textbf{process}(S?\ \textbf{bool}, T!\ \textbf{bool}) \\ & \quad \mathit{StackMaster}(S, T, Q, P) \parallel \mathit{Reg}(p, q) \\ & \quad \textbf{channel}\ (Q, q), (P, p) \\ & \textbf{end} \end{aligned}$$

To keep the example simple, from now on we focus on the compilation of the *Reg* process only.

The next step in the compilation process is *handshake expansion.* Here, all commands of CHP are implemented in terms of actions on wires carrying Boolean signals. A four-phase protocol is used for all communication, and all data items transmitted through channels are dual-rail-encoded. Since Boolean values are to be received on port $p?$, two data wires and one acknowledgment wire are needed, as illustrated in Figure 15.22. The data wire pi_1 is used to receive the value 1, whereas the wire pi_0 is used to receive

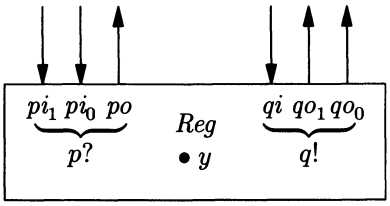


FIGURE 15.22. Handshake expansion of *Reg* process.

the value 0. Similarly, for port $q!$, we get request wire qi and (output) data wires qo_1 and qo_0. At this point we have a choice as to which process is the *active* agent (the initiator of communication) and which one is the *passive* agent. If the *Reg* process is active on port $q!$, then sending a 1 is replaced by $qo_1\uparrow; [qi]; qo_1\downarrow; [\neg qi]$ and sending a 0 is replaced by $qo_0\uparrow; [qi]; qo_0\downarrow; [\neg qi]$. Intuitively, the process first raises the appropriate qo wire, waits for the receiving process to acknowledge the data, lowers the data signal, and waits for the receiver to complete the handshake protocol. On the other hand, if the *Reg* process is passive on $q!$, then sending a 1 uses the following protocol: $[qi]; qo_1\uparrow; [\neg qi]; qo_1\downarrow$, and sending a 0 is similarly treated. If *Reg* is passive on $p?$, then receiving a 1 is replaced by $[pi_1]; po\uparrow; [\neg pi_1]; po\downarrow$, and receiving 0 is replaced by $[pi_0]; po\uparrow; [\neg pi_0]; po\downarrow$. Finally, a probe of p for value 1 (respectively, for value 0) is simply replaced by pi_1 (respectively, pi_0).

In our case, we assume that *Reg* is passive for communication on both ports $p?$ and $q!$. Thus, the handshake expansion of the body of *Reg* yields:

$$
\begin{array}{l}
*[[\quad (pi_1 \rightarrow y\uparrow; [y]; po\uparrow; [\neg pi_1]; po\downarrow) \\
\quad\ \| (pi_0 \rightarrow y\downarrow; [\neg y]; po\uparrow; [\neg pi_0]; po\downarrow) \\
\quad\ \| ((qi \wedge y) \rightarrow qo_1\uparrow; [\neg qi]; qo_1\downarrow) \\
\quad\ \| ((qi \wedge \neg y) \rightarrow qo_0\uparrow; [\neg qi]; qo_0\downarrow) \\
]].
\end{array}
$$

Intuitively, the first guarded command can be interpreted as follows: The process continually checks the value on the pi_1 wire. If pi_1 becomes 1, the process assigns 1 to y. It then waits for y to become 1. Once this happens, the process raises the acknowledgment signal po and waits for the data input signal to return to 0, after which the process concludes by lowering po. The other guarded commands can be interpreted similarly.

After the handshake expansion, further optimizations are often possible. The most important one is the *reshuffling* of communication actions. Intuitively, reshuffling moves the beginning of some four-phase handshake protocols to positions located earlier in the process. Thus processes can be started earlier and several processes can work concurrently; this improves the performance of the circuit. For brevity, we perform no further optimizations of the handshake expansion of *Reg*.

The final transformation of a CHP program translates the handshake program into a set of production rules. A *production rule*, written as $G \mapsto S$, is a guarded assignment. The guard G is a Boolean expressions and the assignment S is either of the form $x\uparrow$ or $x\downarrow$, for some variable x. The behavior of a set of production rules can be summarized as follows:

1. Nondeterministically select a production rule.
2. Evaluate its guard. If the guard is true, update the state of all signals according to the assignment. Otherwise, retain the current state.
3. Go to (1).

Note that the nondeterministic selection is *weakly fair* in the sense that each production rule is selected infinitely often [89, 91].

To ensure proper operation, any valid set of production rules must satisfy two basic requirements: stability and noninterference. A production rule $G \mapsto x\uparrow$ is said to be *stable* if, whenever G holds in s, then either G or x must hold in each possible successor state of s. The stability of a production rule $G \mapsto x\downarrow$ is defined similarly. Altogether, a production rule is said to be stable if, when its guard becomes valid, the guard continues to hold in every valid computation path until the assignment is performed. A set of production rules is stable if every production rule in the set is stable. The stability requirement implies freedom from hazards in the circuit.

The second requirement, "noninterference," states that no inconsistent production rules are allowed. More formally, two production rules $G_1 \mapsto x\uparrow$ and $G_2 \mapsto x\downarrow$ that assign complementary values to the same variable are said to be *complementary*. Two complementary production rules are *noninterfering* if no state can be reached in which both G_1 and G_2 hold. A set of production rules is said to satisfy the noninterference requirement if every pair of complementary production rules in the set is noninterfering.

The compilation of a handshaking protocol to production rules is the most difficult part in the compilation process. Part of the difficulty stems from the fact that we are trying to implement a sequential process (albeit a simple one) by a collection of very simple concurrent processes. Thus, the sequencing must be performed explicitly, and efficiency is difficult to achieve. The compilation proceeds as follows. First, the handshake process is syntactically translated into a set of production rules. To ensure proper sequencing, additional state variables are introduced, along with production rules involving these variables. This is done to distinguish different states with identical signal values. This step is referred to as *state assignment*, although it is quite different from the traditional state assignment. Once the state assignment has been performed, the guards of the production rules are strengthened until the set satisfies both the stability and noninterference requirements. Finally, the production rules assigning values to the same variable are grouped together, and a (complex) transistor cell is derived.

This last step is referred to as the *operator reduction* step. Here, further introduction of state variables and production rules can occur to make the implementation more efficient.

In the case of the *Reg* handshake expansion, one verifies that each handshake sequence results in a different variable- and wire-state. Thus, no further state variables are needed for the state assignment. The syntax-directed translation of the handshake process yields the following set of production rules (listed in the order in which they are produced from the program):

$$
\begin{array}{lrcl}
(1) & pi_1 & \mapsto & y\uparrow \\
(2) & y & \mapsto & po\uparrow \\
(3) & \neg pi_1 & \mapsto & po\downarrow \\
(4) & pi_0 & \mapsto & y\downarrow \\
(5) & \neg y & \mapsto & po\uparrow \\
(6) & \neg pi_0 & \mapsto & po\downarrow \\
(7) & qi \wedge y & \mapsto & qo_1\uparrow \\
(8) & \neg qi & \mapsto & qo_1\downarrow \\
(9) & qi \wedge \neg y & \mapsto & qo_0\uparrow \\
(10) & \neg qi & \mapsto & qo_0\downarrow
\end{array}
$$

If the circuit is started in the state in which all signals are 0, then the complementary Rules (3) and (5) are interfering. Thus, this set of production rules is not free of interference. Note that Rules (1) and (4) are not interfering since we assume the environment never raises both pi_1 and pi_0. There are several ways of solving these problems, but one of the simplest solutions is to strengthen the guard of Rule (2) to $pi_1 \wedge y$, the guard of Rule (5) to $pi_0 \wedge \neg y$, and the guards of Rules (3) and (6) to $\neg pi_1 \wedge \neg pi_0$. Note that these are indeed valid strengthenings since the corresponding states in the handshake process all satisfy these stronger conditions. After also merging Rules (3) and (6) (since they are now identical) we obtain the set

$$
\begin{array}{lrcl}
(1) & pi_1 & \mapsto & y\uparrow \\
(2') & pi_1 \wedge y & \mapsto & po\uparrow \\
(3') & \neg pi_1 \wedge \neg pi_0 & \mapsto & po\downarrow \\
(4) & pi_0 & \mapsto & y\downarrow \\
(5') & pi_0 \wedge \neg y & \mapsto & po\uparrow \\
(7) & qi \wedge y & \mapsto & qo_1\uparrow \\
(8) & \neg qi & \mapsto & qo_1\downarrow \\
(9) & qi \wedge \neg y & \mapsto & qo_0\uparrow \\
(10) & \neg qi & \mapsto & qo_0\downarrow
\end{array}
$$

One can show—using model checking for example—that this set satisfies both the noninterference and stability requirements.

The only remaining step is to design some components that implement the production rules derived above. If we can ensure that every guard of a rising assignment ($x\uparrow$) is a conjunction of negated terms and every guard

of a falling assignment $(x\downarrow)$ is a conjunction of (positive) terms, then a one-output CMOS cell can be used for each variable in the program. In our case, we can add five additional variables that take on the complemented values of some signals, to arrive at the production rule set given by

$$\begin{aligned}
\neg pi_1 &\mapsto \widetilde{pi}_1\uparrow \\
pi_1 &\mapsto \widetilde{pi}_1\downarrow \\
\\
\neg pi_0 &\mapsto \widetilde{pi}_0\uparrow \\
pi_0 &\mapsto \widetilde{pi}_0\downarrow \\
\\
\neg x &\mapsto y\uparrow \\
x &\mapsto y\downarrow \\
\\
\neg \widetilde{qo}_0 &\mapsto qo_0\uparrow \\
\widetilde{qo}_0 &\mapsto qo_0\downarrow \\
\\
\neg \widetilde{qo}_1 &\mapsto qo_1\uparrow \\
\widetilde{qo}_1 &\mapsto qo_1\downarrow \\
\\
x \wedge qi &\mapsto \widetilde{qo}_1\downarrow \\
\neg x \vee \neg qi &\mapsto \widetilde{qo}_1\uparrow \\
\\
y \wedge qi &\mapsto \widetilde{qo}_0\downarrow \\
\neg y \vee \neg qi &\mapsto \widetilde{qo}_0\uparrow \\
\\
\neg \widetilde{pi}_1 &\mapsto x\uparrow \\
pi_0 &\mapsto x\downarrow \\
\\
(\neg \widetilde{pi}_1 \wedge \neg y) \vee (\neg \widetilde{pi}_0 \wedge \neg x) &\mapsto po\uparrow \\
\widetilde{pi}_1 \wedge \widetilde{pi}_0 &\mapsto po\downarrow
\end{aligned}$$

where we have deliberately grouped together the production rules that set and reset the same variable. We see that the first five pairs of production rules can be implemented as inverters, and the next two pairs as two-input NAND gates. The last two pairs of production rules are nonstandard. They can be implemented as shown in Figure 15.23. Note that there are states in which the output nodes are isolated. If we ensure that the stack is operated frequently enough (so that charge leakage does not cause any problem in the dynamic gates), one can verify that the CMOS cells shown in the figure work properly. On the other hand, if we cannot guarantee the operating frequency, the cell can be modified by the addition of a "staticizer" as was discussed in Chapter 5 in the circuit of Figure 5.9. The final circuit implementing *Reg* is shown in Figure 15.24.

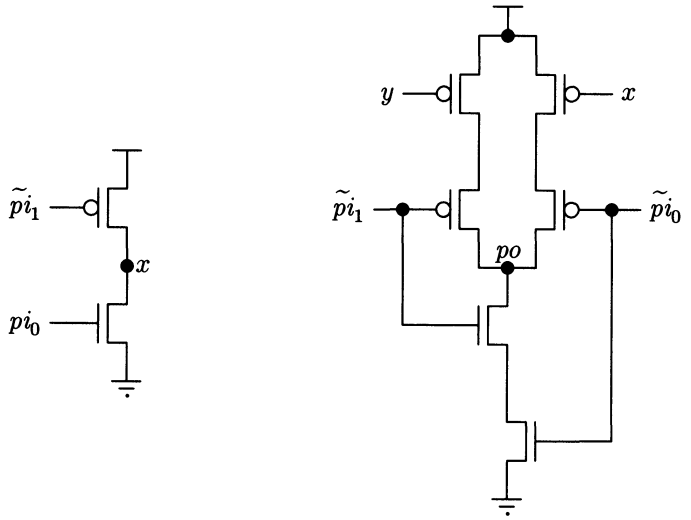


FIGURE 15.23. CMOS cells for production rules: (a) cell A, (b) cell B.

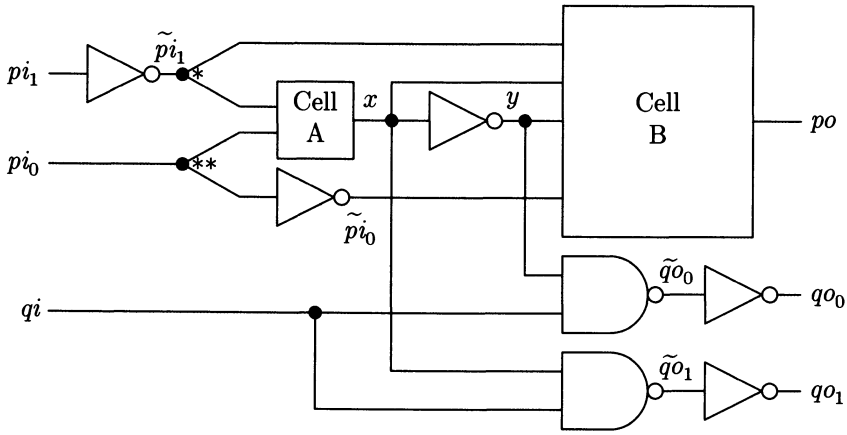


FIGURE 15.24. Circuit derived for the *Reg* process.

It should be emphasized that the design methodology creates speed-independent, but not delay-insensitive, designs, since it is assumed that a signal connected to several cells arrives at these cells at the same time, i.e., that forks are isochronic. One can first assume that all the wires have delays and analyze the circuit under this assumption. Thus, one can determine which wires actually need the isochronic assumption. In this case, it turns out that only the forks labeled $*$ and $**$ must be isochronic. In

practical terms, even those forks can use a more relaxed delay assumption. Basically, as long as the delay from pi_0 to the N-transistor in cell A is less than the combined delay of the lower inverter, the complex final cell B, the environment, the $\widetilde{pi}_1$ inverter, and the wire to the P-transistor in cell A, the circuit functions properly. A similar argument can be made for the upper fork. Thus, in practice, the circuit is very close to being delay-insensitive.

It is important to realize that many of the steps involved in the compilation process require subtle choices that may have significant impact on circuit area and delay. Although heuristics have been suggested for many of the choices, much of the effort is directed toward aiding a skilled designer instead of creating automatic tools. This is advantageous, because better decisions can often be made by humans than by programs. It does, however, require more informed designers than do other methods. Another source of problems is that circuits resulting from this synthesis process require complex custom gates, and these gates cannot be easily broken down into simpler components.

Work related to Martin's approach has been reported in [29], where a compiler was written to automatically transform concurrent programs written in a language similar to CHP to circuits consisting of standard building blocks, rather than custom CMOS cells. Another approach using standard building blocks is described in Section 15.11.

15.11 Handshake Circuits

A design methodology developed by van Berkel and his research group at Philips Research [136] uses a high-level programming language called Tangram and an intermediate architecture of "handshake circuits." Tangram is based on Hoare's CSP [62], and Dijkstra's guarded-command language [43]. Tangram programs are automatically compiled to handshake circuits. A handshake circuit is a delay-insensitive network of special components connected by communication channels. A component communicates with other components only through Request/Acknowledge messages along the channels. About 20 types of components are used; they are chosen to be in close correspondence with operations definable in Tangram. Thus, the translation of a Tangram program to a handshake circuit is "syntax-directed," i.e., the structure of a Tangram program is reflected in the structure of the corresponding handshake circuit. To complete the design, handshake components are implemented as VLSI circuits.

An important advantage of using components that correspond to Tangram statements is the ability to estimate the area, speed, and energy consumption of the final VLSI circuit from the Tangram program. A given functional specification can be represented by several functionally equivalent, but structurally different, Tangram programs. By analyzing these programs, the designer may be able to select the most appropriate struc-

ture to match the design goals. For example, in designing a portable CD player, one may wish to minimize the energy consumption in order to maximize the lifetime of the battery.

We now give some examples to illustrate the flavor of the design method. A Tangram program, $BUF_1(a, b)$, for a one-place buffer capable of storing a Boolean value is as follows:

$$(a?bool \ \& \ b!bool) \cdot [\![\ x : \textbf{var} \ bool \mid \#[a?x; b!x] \ ]\!],$$

where the statement in parentheses is a declaration of port and channel variables, and the statement after the dot $\cdot$ represents the program behavior. The declaration states that a is an input port of the buffer, and b is its output port. Both the input and output ports are of type Boolean. The centered dot $\cdot$ separates the declaration from the behavior. The behavior, in this case, is defined by a "block" command, enclosed in the brackets '[[' and ']]'. Within the command, x is declared as a local variable of type Boolean, the bar '|' is a separator, and the remaining Tangram statement is a command describing the program actions. The symbol # denotes unbounded repetition. The command $a?x$ denotes the storing of a (Boolean) value received on input port a in internal variable x. The command $b!x$ denotes the sending of the value stored in x through port b. The semicolon denotes that the second command follows the first.

Our next example illustrates the modularity of Tangram. A two-place buffer, $BUF_2(a, c)$, can be specified by the Tangram program

$$(a?bool \ \& \ c!bool) \cdot [\![\ b : \textbf{chan} \ bool \mid BUF_1(a, b) \ || \ BUF_2(b, c) \ ]\!],$$

where b is an internal channel, and $BUF_1(a, b)$ and $BUF_2(b, c)$ are two instances of the one-place buffer. The output of the first buffer is connected to the input of the second through channel b. The communication along b follows CSP rules [62], in that it requires the simultaneous participation by receiver and sender, and has the effect of copying the value stored in the local variable of the first buffer into the local variable of the second. The fact that the two buffers operate in parallel is denoted by ||.

The following Tangram program, $WAG(a, c)$ (for "wagging"buffer), is functionally equivalent to the two-place buffer:

$$(a?bool \ \& \ c!bool) \cdot [\![\ x, y : \textbf{var} \ bool \mid a?x; \#[(a?y \ || \ c!x); (a?x \ || \ c!y)] \ ]\!].$$

Here, the first data value is placed in variable x and then successive values are placed alternately into y and x. Similarly, the output is taken first from x, then from y, etc. The two designs, $BUF_2(a, c)$ and $WAG(a, c)$, have the same input/output behaviors but differ in structure. Depending on the design goals, one may prefer one design over another. For example, it has been shown [136] that a shift register designed using the "wagging principle" is faster that one using the "ripple structure."

A handshake circuit for $BUF_1(a, b)$ is shown in Figure 15.25. There are five handshake components depicted by large circles. Each component may

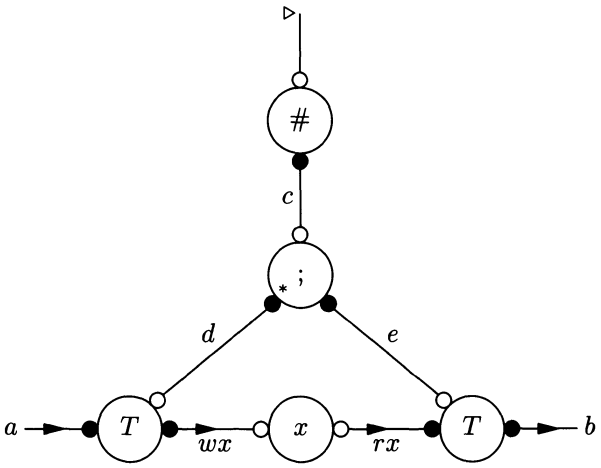


FIGURE 15.25. A handshake circuit for $BUF_1(a, b)$.

have active ports, denoted by black dots, and passive ports, denoted by small circles. The circuit has three (external) ports, labeled $\triangleright$, a, and b. There are five channels, labeled c, d, e, wx, and rx. The operation of the buffer is started by a request from the environment appearing on the channel $\triangleright$. Since the buffer will operate forever, no acknowledgment will ever be sent along $\triangleright$. The *repeater* component (#) sends out an initial request on its active port through channel c after the receipt of a communication on $\triangleright$ and, subsequently, after each communication received along c; this corresponds to the unbounded repetition in the Tangram command for $BUF_1(a, b)$. The component marked with ; is a *sequencer*. After the receipt of a request along c, it first engages in an exchange of handshakes through the port marked $*$, and then through the other port. After these handshakes, it sends an acknowledge signal through channel c. Each of the two components marked T is a *transferrer*. In response to a request along d, the left transferrer requests and receives a value along a and passes it along wx. Upon receipt of an acknowledgment along wx, it sends an acknowledgment along d. The component marked x is a *variable*. A value passed along wx is stored and then acknowledged along wx. A request received by variable x along rx results in the sending of the data stored in x along rx. Finally, the right transferrer, upon receipt of a request along e, requests a value from x, passes that value along b, and reports completion along e.

For further examples of handshake circuits, the reader is referred to [136]. To improve the efficiency of a handshake circuit, one may perform "peephole" optimization. Some techniques have been developed for the estimation of area, speed, and power of handshake circuits. Once the "best" handshake circuit is selected, the design is completed by implementing the handshake components involved and interconnecting them as dictated by the

handshake circuit. Many of the handshake components, like the repeater, are fixed and simple. The implementation of components like variables and transferrers depends on the data types involved.

A discussion of the pros and cons of this methodology, along with a number of interesting design examples, can be found in [136].

15.12 Module-Based Compilation Systems

The main advantage of module-based systems is that their use can be coupled with a high-level language and automatic translation software. A subset of OCCAM—a language invented to describe communicating sequential processes—has been used by [7]. The approach here is to provide a delay-insensitive module for each of the language constructs. For example, a *while* loop in the language would require a WHILE element, which has connection terminals for a conditional test, a loop body, and an interface to the surrounding environment. It is then a straightforward process to convert parse trees for the input language into circuit structures built out of delay-insensitive modules. Techniques similar to peep-hole optimization in software compilers can be applied to the circuit to reduce area and delays. For example, a WHILE element with its condition always true can be replaced by an infinite loop element. Finally, the circuit can be implemented by interconnecting the modules as specified by the program translation.

This approach is very similar to standard cell synthesis and has similar advantages and disadvantages. Since modules are standardized, they can be precertified. In designing circuits that use such modules, we can safely assume the modules are correct, and we need to worry only about the logical correctness of the overall network of modules. Also, typical modules tend to be simple and can be manually developed by skilled designers. Consequently, the methods used to synthesize the modules need not be efficient. For example, the exponential algorithm for converting I-nets to ISGs is acceptable for module synthesis. On the other hand, since we are required to use preset modules, we usually cannot perform optimizations on the module structures themselves. Thus, some possible optimizations are ruled out, because we do not have the required simpler modules. Also, for each implementation technology, we may need to generate a new set of modules. While the specific design rules of a different process may not introduce so many changes as to require new modules, technologies such as mask- and field-programmable gate arrays, and even specific architectures within these technologies, may require their own module sets. Finally, while strict delay-insensitive designs encapsulate timing issues within modules, some methodologies (including that of [7]) use bundled data protocols. Such protocols require timing constraints between modules, thus complicating circuit implementation.

15.13 DCVSL and Interconnection Modules

In the design methodology of [70, 98, 97], a digital system is composed of two types of blocks: computation blocks and interconnection blocks. Computation blocks include such functional units as shifters, multipliers, arithmetic logic units (ALUs), and other combinational circuits, and also random-access memories (RAMs) and read-only memories (ROMs). Interconnection blocks provide the required handshaking protocols between computation blocks and ensure proper timing for data transfers. Thus, they may include such data transfer circuits as pipeline registers and multiplexers. The control blocks are generated from STG specifications described in Section 15.6. The computation blocks generate completion information, in addition to performing computation. For this reason, the authors refer to these systems as *self-timed*—a term introduced by C. L. Seitz in [96], Chapter 7.

One implementation of the computation blocks uses *differential cascode voltage switch logic* (*DCVSL*) [60]. The DCVS logic is a precharge logic that uses two-rail complementary inputs and provides complementary outputs. The circuits use a four-phase protocol, and can easily be made to generate completion signals. A typical DVCS cell (with completion signal generation) is shown in Figure 15.26. When the request line *Req* is 0, the cell is precharged, the two outputs are set to 0, and the *Done* signal is 0. When

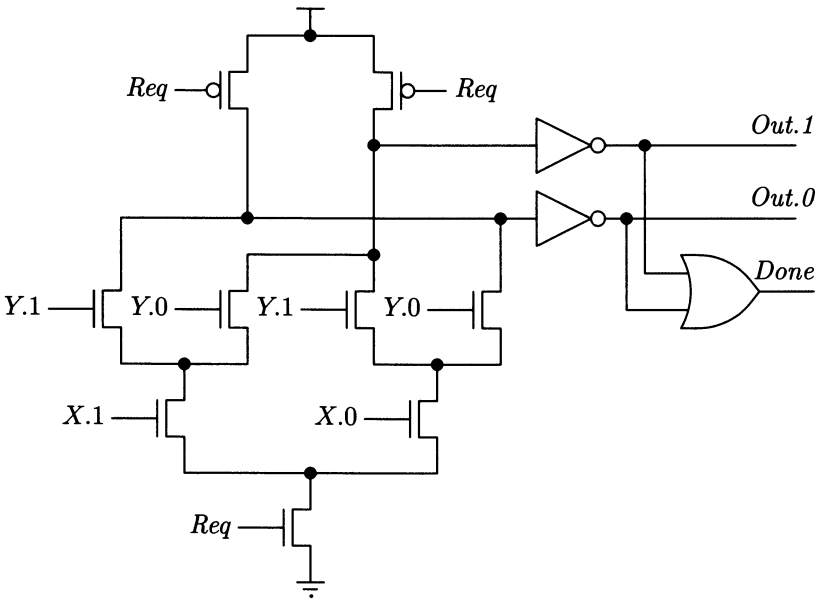


FIGURE 15.26. A DCVS logic block.

the *Req* line becomes 1 and the inputs have taken on their values, the tree of N-transistors implementing the desired function—in this case, the XOR function of two variables—pulls one, but only one, of the precharged nodes down to 0, causing one of the outputs to become 1. This in turn causes the *Done* signal to become 1. Eventually, *Req* becomes 0, causing *Done* to follow, and the pattern repeats. Altogether, the (*Req*,*Done*) sequence is $((0,0),(1,0),(1,1),(0,1),(0,0))$, as required by the four-phase protocol. Note that the succeeding circuit must have time to accept the new result before *Req* returns to 0.

An example of a pipeline constructed with this method is shown in Figure 15.27. A typical DCVS logic block consists of a several DCVSL cells connected in an acyclic network to compute some complex Boolean functions. Only the final DCVSL cells in the DCVS logic block need to compute their completion (*Done*) signals. These separate completion signals are then combined to form a single completion signal, R_{in}, from the DCVSL block. The signal R_{out} is connected to the individual *Req* lines in the cells.

The operation of the logic blocks must be controlled by the interconnect block. Since individual logic blocks have different and sometimes varying delays, the control must ensure that no data is overwritten or used more than once. In the methodology described in [97] these control circuits are designed using STGs. For more details, we refer the reader to [97].

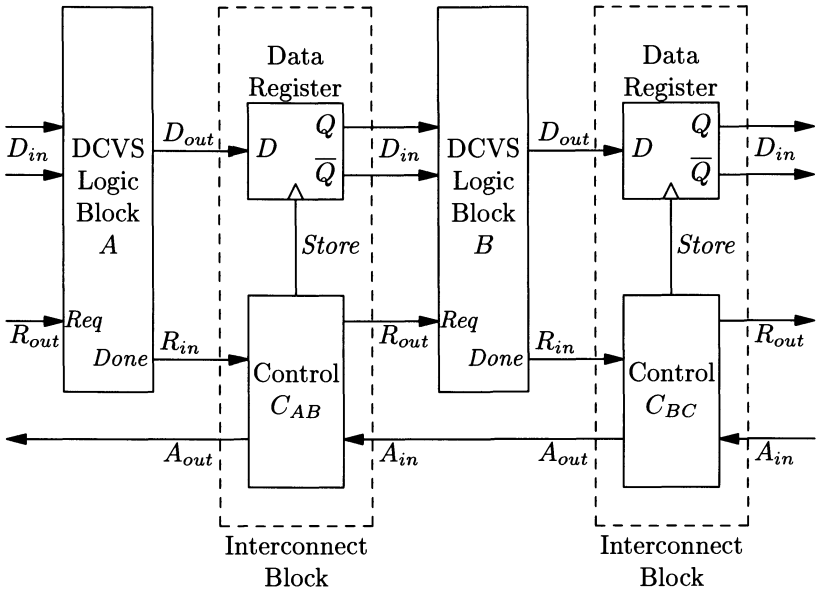
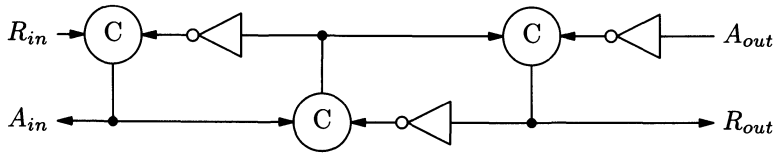


FIGURE 15.27. A pipeline with DCVSL blocks.

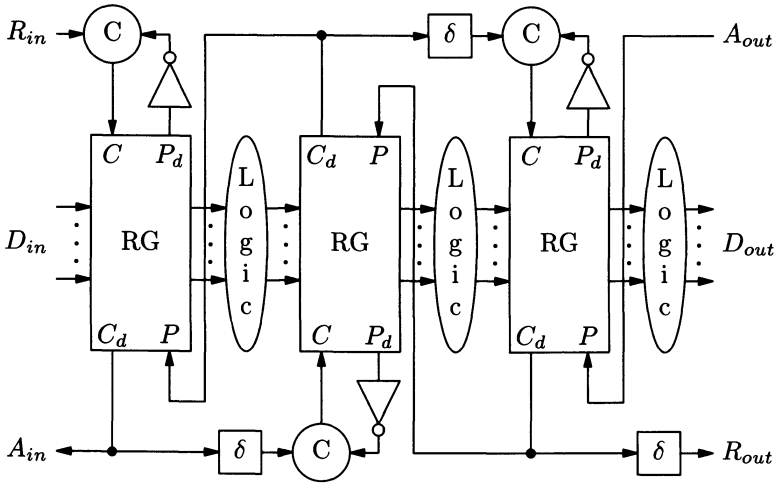
15.14 Micropipelines

Micropipelines were introduced in Ivan Sutherland's Turing Award lecture [132] primarily as an asynchronous alternative to synchronous "elastic" pipelines, i.e., pipelines in which the amount of data contained can vary. However, they also serve as a powerful method for implementing general computations. Although often categorized as a delay-insensitive methodology, they are actually composed of a bounded-delay data path moderated by a delay-insensitive control circuit. Note that the timing constraints in this system are not simply the bundled-data constraints, since the timing of all computation elements is important.

The basic structure of a micropipeline consists of the control FIFO ("first-in, first-out" buffer) shown in Figure 15.28(a), where the gates labeled C are Muller C-ELEMENTs. The FIFO stores transitions sent to it through R_{in}, shifts them to the right, and eventually outputs them through R_{out}. To understand how the circuit works, consider the initial state, in which the FIFO is empty and all the wires, except the inverter outputs,



(a)



(b)

FIGURE 15.28. Micropipeline structure: (a) control, (b) computation.

are 0. A transition from 0 to 1 at R_{in} is able to pass through all the C-ELEMENTs in series, and emerges on R_{out}. During this process a transition moves completely around each of the small cycles consisting of two C-ELEMENTs and an inverter. Except for A_{out}, the levels of all the signals in the circuit change. The next transition, which is from 1 to 0, is able to pass through the first two C-ELEMENTs, but not through the third one, which is waiting for a transition on A_{out}. This represents the case where the output side of the FIFO is not yet ready to accept a new transition. Note that new transitions may enter through R_{in} before previous transitions leave the FIFO, and they will be held up at successively earlier C-ELEMENTs, one transition per C-ELEMENT. Since the sender must wait for a transition on A_{in} before sending the next transition on R_{in}, when a transition on A_{in} does appear the sender knows that its previous transition has passed through the first C-ELEMENT. If a transition appears on A_{out}, a transition will be able to leave through R_{out}, freeing up a space in the pipeline. We require transitions on the receiver side to alternate between A_{out} and R_{out} to make sure the transitions sent on A_{out} actually pass through the first C-ELEMENT from the right. With these restrictions, the pipeline acts like a FIFO for transitions. Note that the structure repeats—there are three stages in the pipeline shown, with adjacent stages flipped around the horizontal axis—and could be extended by simply connecting additional stages to the front or back.

We can take the simple transition FIFO described above and use it as the basis for a complete computation pipeline, as shown in Figure 15.28(b). The blocks labeled RG are registers. The register output C_d is a delayed version of input C, and output P_d is a delayed version of input P. Thus, the transition FIFO of Figure 15.28(a) is embedded in Figure 15.28(b), with delays added to some of the lines. The registers are similar to level-sensitive latches from synchronous design, except that they respond to transitions on two inputs instead of a single clock input. They are initially active, passing data directly from data inputs to data outputs. When a transition occurs on the C (capture) wire, data are no longer allowed to pass and the current values of the outputs are statically maintained. Then, once a transition occurs on the P (pass) input, data are again allowed to pass from input to output, and the cycle repeats. As mentioned earlier, C_d and P_d are copies of the control signals C and P, delayed so that the register completes its response to the control signal transitions before they are sent back out. Refer to the figure; if we ignore the logic blocks and the explicit delay element, we have a simple data FIFO. Data are first supplied by the sender, and then a transition occurs on the R_{in} wire. Because of the delays associated with the control wires passing through the registers, the data advance ahead of the control transition. If the control transition is forced to wait at any C-ELEMENT, the data wait in the preceding register, which is in the capture mode. Thus, the transitions are buffered in the FIFO control, and the data are buffered in the registers.

Computation on the data stored in a micropipeline is accomplished by adding logic blocks between the register stages. Since these blocks slow down the data moving through them, the accompanying control transition must also be delayed; this is done by the added delay elements, labeled δ, whose delay must be at least as large as the worst-case delay of the logic block. The major benefit of the micropipeline structure is that, since the registers moderate the flow of data through the pipeline, they also "absorb" hazards. Thus, any logic structures, including the straightforward structures used in synchronous designs, can be used in the logic blocks. This means that a micropipeline can be constructed from a synchronous pipeline by simply replacing the clocked level-sensitive latches with the micropipeline control structure. Since the micropipeline removes the requirement of operating in lock-step with a global clock, an added benefit of a micropipelined version of a FIFO is that it is automatically elastic, in that data can be sent to and received from the FIFO at arbitrary times.

Although micropipelines are a powerful implementation strategy that elegantly implements elastic pipelines, they are not without some problems. While the micropipeline removes the hazard considerations of other bounded-delay models, it still delivers worst-case performance by adding delay elements to the control path to match worst-case computation times. Also, since delay assumptions are made, the circuits must be tested for delay faults. The final, and probably most significant, problem with micropipelines is the present lack of systematic methods for their use in complex systems. Although simple straight-line pipelines without feedback can be implemented easily by micropipelines, few designs conform to this simple model. Many applications, like digital signal processing, involve highly repetitive computations. Typically, the computation performed depends on earlier inputs and previously calculated values. Unfortunately, it is not clear how to implement feedback in micropipelines.

A variation of micropipelines are *self-timed rings* [142], which are essentially micropipelines whose output is connected directly to its input. If such a ring contains an odd number of inversions in the control path, it will operate indefinitely. Each complete cycle through the pipeline performs a step in an iterative computation. If completion signals and handshake protocols are used, there can be several computation "waves" progressing through the ring simultaneously. Such a ring circuit can achieve very high performance. For example, a very fast 54-bit divider circuit was described in [143]. In general, however, the design of self-timed ring circuits is not straightforward.

Although the control structure of a micropipeline can be enhanced by using additional elements, this is a fairly complex activity. While several micropipelined solutions using specific circuit structures have been developed [73, 85, 128], including complete asynchronous microprocessors [54, 114, 130], a general, higher-level method for designing micropipeline control circuits is yet to be developed.

15.15 Concluding Remarks

In this chapter we have discussed the following synthesis approaches: Huffman circuits, Hollaar circuits, burst-mode circuits, I-nets, STGs, change diagrams, trace theory, communicating process compilation, handshake circuits, module-based compilation, DCVSL-based circuits, and micropipelines. Making a thorough comparison of the different approaches, especially in the critical issues of performance, area, and power usage, is difficult, and very few such comparisons have been done. Moreover, in spite of the fact that several impressive asynchronous designs have been carried out, there has not been any compelling evidence that asynchronous circuits are better than synchronous. The fundamental issue as to which of the asynchronous design styles is best in performance, or area, or power, as well as the question whether any asynchronous approach is preferable over the prevalent synchronous model, is still open.

As we have seen, asynchronous design is a rich area of research, with many different approaches to circuit synthesis. We stress that only some of the results in this area have been surveyed in this chapter, since our goal has been to present an overview of several representative approaches. Many interesting techniques have been omitted, important areas such as verification and testing largely ignored, and the methodologies that were discussed have not been explored in depth. Our hope is, however, that this chapter gives sufficient background to put further readings in proper context.

Bibliography

[1] S. B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6), pp. 509–516, 1978.

[2] J. H. Anderson and M. G. Gouda. A New Explanation of the Glitch Phenomenon. *Acta Informatica*, 28, pp. 297–309, 1991.

[3] P. A. Beerel and T. H. Meng. Automatic Gate-Level Synthesis of Speed-Independent Circuits. In *Proceedings of International Conference on Computer-Aided Design*, IEEE Press, pp. 581–586, 1992.

[4] P. A. Beerel and T. H. Meng. Logic Transformations and Observability Don't Cares in Speed-Independent Circuits. In *Proceedings of Tau 93.* Participant's Proceedings, 1993.

[5] G. Birkhoff and T. C. Bartee. *Modern Applied Algebra.* McGraw-Hill Book Company, 1970.

[6] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* American Elsevier Publishing Company, 1976.

[7] E. Brunvand and R. F. Sproull. Translating Concurrent Programs into Delay-Insensitive Circuits. In *Proceedings of International Conference on Computer-Aided Design*, IEEE Press, pp. 262–265, 1989.

[8] R. E. Bryant. Race Detection in MOS Circuits by Ternary Simulation. In F. Anceau and E. J. Aas, editors, *Proceedings of the IFIP International Conference on Very Large Scale Integration*, North-Holland Publishing Company, pp. 85–95, 1983,

[9] R. E. Bryant. *Toward a Proof of the Brzozowski-Yoeli Conjecture on Ternary Simulation.* Unpublished Manuscript, 1983.

[10] R. E. Bryant. A Switch-Level Model and Simulator for MOS Digital Systems. *IEEE Transactions on Computers*, C-33(2), pp. 160–177, 1984.

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

[12] R. E. Bryant. Algorithmic Aspects to Symbolic Switch Network Analysis. *IEEE Transactions on Computer-Aided Design*, CAD-6(4), pp. 618–633, 1987.

[13] R. E. Bryant. Boolean Analysis of MOS Circuits. *IEEE Transactions on Computer-Aided Design*, CAD-6(4), pp. 634–649, 1987.

[14] R. E. Bryant. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Applications to Integer Multiplication. *IEEE Transactions on Computers*, C-40(2), pp. 205–213, 1991.

[15] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3), pp. 292–318, 1992.

[16] J. A. Brzozowski. *Regular Expression Techniques for Sequential Circuits.* PhD thesis, Department of Electrical Engineering, Princeton University, Princeton, New Jersey, USA, 1962.

[17] J. A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4), pp. 481–494, 1964.

[18] J. A. Brzozowski and J. C. Ebergen. Recent Developments in the Design of Asynchronous Circuits. In J. Csirik, J. Demetrovics, and F. Gécseg, editors, *Proceedings of Fundamentals of Computation Theory, Lecture Notes in Computer Science*, 380, Springer-Verlag, pp. 78–94, 1989.

[19] J. A. Brzozowski and J. C. Ebergen. On the Delay-Sensitivity of Gate Networks. *IEEE Transactions on Computers*, C-41(11), pp. 1349–1360, 1992.

[20] J. A. Brzozowski and E. J. McCluskey. Signal Flow Graph Techniques for Sequential Circuit State Diagrams. *IEEE Transactions on Electronic Computers*, EC–12(2), pp. 67–76, 1963.

[21] J. A. Brzozowski and C-J. H. Seger. A Characterization of Ternary Simulation of Gate Networks. *IEEE Transactions on Computers*, C-36(11), pp. 1318–1327, 1987.

[22] J. A. Brzozowski and C-J. H. Seger. A Unified Framework for Race Analysis of Asynchronous Networks. *Journal of the ACM*, 36(1), pp. 20–45, 1989.

[23] J. A. Brzozowski and C-J. H. Seger. Advances in Asynchronous Circuit Theory Part I: Gate and Unbounded Inertial Delay Models. *Bulletin of the European Association for Theoretical Computer Science*, 1990(42), pp. 198–249, 1990.

[24] J. A. Brzozowski and C-J. H. Seger. Advances in Asynchronous Circuit Theory Part II: Bounded Inertial Delay Models, MOS Circuits, Design Techniques. *Bulletin of the European Association for Theoretical Computer Science*, 1991(43), pp. 199–263, 1991.

[25] J. A. Brzozowski and M. Yoeli. *Digital Networks*. Prentice-Hall, 1976.

[26] J. A. Brzozowski and M. Yoeli. On a Ternary Model of Gate Networks. *IEEE Transactions on Computers*, C-28(3), pp. 178–184, 1979.

[27] J. A. Brzozowski and M. Yoeli. Combinational Static CMOS Networks. *Integration, The VLSI Journal*, 5, pp. 103–122, 1987.

[28] J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2), pp. 142–170, 1992.

[29] S. M. Burns. *Automated Compilation of Concurrent Programs into Self-timed Circuits*. Master's thesis, Department of Computer Science, California Institute of Technology, Pasadena, California, USA, 1987.

[30] P. K. Chan and K. Karplus. Computing Signal Delay in General RC Networks by Tree/Link Partitioning. *IEEE Transactions on Computer-Aided Design*, CAD-9(8), pp. 898–902, 1990.

[31] T. J. Chaney and C. E. Molnar. Anomalous Behavior of Synchronizer and Arbiter Circuits. *IEEE Transactions on Computers*, C-22(4), pp. 421–422, 1973.

[32] S. G. Chappell and S. S. Yau. Simulation of Large Asynchronous Logic Circuits Using an Ambiguous Gate Model. *AFIPS Conference Proceedings*, 39, pp. 651–661, 1971.

[33] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Massachusetts, USA, 1987.

[34] T.-A. Chu, C. K. C. Leung, and T. S. Wanuga. A Design Methodology for Concurrent VLSI Systems. In *Proceedings International Conference on Computer Design*, IEEE Press, pp. 407–410, 1985.

[35] W. A. Clark. Macromodular Computer Systems. In *Proceedings of the Spring Joint Computer Conference*, AFIPS, 1967.

[36] W. A. Clark and C. E. Molnar. The Promise of Macromodular Systems. In *Digest of Papers, Compcon 72*, 1972.

[37] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. In *Proceedings of the 10th ACM Symposium on the Principles of Programming Languages*, ACM, 1983.

[38] W. Coates, A. Davis, and K. Stevens. The Post Office Experience: Designing a Large Asynchronous Chip. *Integration, The VLSI Journal*, 15(3), pp. 341–366, 1993.

[39] S. A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on the Theory of Computing*, pp. 151–158, ACM, 1971.

[40] O. Coudert, J.-C. Madre, and C. Berthet. Verifying Temporal Properties of Sequential Machines Without Building Their State Diagrams. In E. Clarke and R. Kurshan, editors, *Proceedings of Computer-Aided Verification '90*, American Mathematical Society, pp. 75–84, 1990.

[41] R. David. Modular Design of Asynchronous Circuits Defined by Graphs. *IEEE Transactions on Computers*, C-26(8), pp. 727–737, 1977.

[42] B. S. Davie and G. J. Milne. The Role of Behaviour in VLSI Design Languages. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, North-Holland Publishing Company, 1987.

[43] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivations of Programs. *Communications of the ACM*, 18(8), pp. 453–457, 1975.

[44] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, USA, 1988.

[45] D. L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In J. Sifakis, editor, *Proceedings of International Workshop on Automatic Verification Methods for Finite State Systems, Lecture Notes in Computer Science*, 407, Springer-Verlag, pp. 197–212, 1990.

[46] J. C. Ebergen. *Translating Programs into Delay-Insensitive Circuits.* CWI Tract 56, Centre for Mathematics and Computing Science, Amsterdam, The Netherlands, 1989.

[47] J. C. Ebergen. A Formal Approach to Designing Delay-Insensitive Circuits. *Distributed Computing*, 5(3), pp. 107–119, 1991.

[48] J. C. Ebergen and J. A. Brzozowski. *On Network and Race Models for Asynchronous Circuits.* Unpublished Manuscript, 1990.

[49] E. B. Eichelberger. Hazard Detection in Combinational and Sequential Switching Circuits. *IBM Journal of Research and Development*, 9, pp. 90–99, 1965.

[50] E. B. Eichelberger and T. W. Williams. A Logic Design Structure for LSI Testability. *Journal of Design Automation and Fault Tolerant Computing*, 2(2), pp. 165–178, 1978.

[51] T. P. Fang and C. E. Molnar. *Synthesis of Reliable Speed-Independent Circuit Modules: II. Circuit and Delay Conditions to Ensure Operation Free of Problems from Races and Hazards*. Technical Memorandum 298, Computer Science Laboratory, Washington University, St. Louis, Missouri, USA, 1983.

[52] G. Fantauzzi. Theory and Design of Switching Circuits. *IEEE Transactions on Computers*, C-23(6), pp. 576–581, 1974.

[53] S. Fortune, J. Hopcroft, and E. M. Schmidt. The Complexity of Equivalence and Containment for Free Single Variable Program Schemes. In G. Ausiello and C. Boehm, editors, *Proceedings of the Fifth International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science*, 62 Springer-Verlag, pp. 227–240, 1978.

[54] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A Micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *Proceedings of the IFIP International Conference on Very Large Scale Integration*. North-Holland Publishing Company, 1993.

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

[56] S. Gingras and J. A. Brzozowski. *Unbounded Finite Delay Models in Asynchronous Circuit Analysis*. Unpublished Manuscript, 1991.

[57] A. Gupta. Formal Hardware Verification Methods: A Survey. *Formal Methods in System Design*, 1(2/3), pp. 151–238, 1992.

[58] M. A. Harrison. *Introduction to Switching and Automata Theory*. McGraw-Hill Book Company, 1965.

[59] S. Hauck. Asynchronous Design Methodologies: An Overview. *Proceedings of the IEEE*, 83(1), 1995.

[60] L. G. Heller, W. R. Griffin, J. W. Davis, and N. G. Thoma. Cascode Voltage Switch Logic: A Differential CMOS Logic Family. In *Proceedings of 1984 IEEE International Solid-State Circuits Conference*, IEEE Press, pp. 16–17, 1984.

[61] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[62] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[63] L. A. Hollaar. Direct Implementation of Asynchronous Control Units. *IEEE Transactions on Computers*, C-31(12), pp. 1133–1141, 1982.

[64] A. Holt and F. Commoner. Events and Conditions. In *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, ACM, pp. 3–52, 1970.

[65] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computations.* Addison-Wesley Publishing Company, 1979.

[66] D. A. Huffman. The Synthesis of Sequential Switching Circuits. *IRE Transactions on Electronic Computers*, 257(3), pp. 161–190, 1954.

[67] D. A. Huffman. The Synthesis of Sequential Switching Circuits. *IRE Transactions on Electronic Computers*, 257(4), pp. 275–303, 1954.

[68] D. A. Huffman. The Design and Use of Hazard-Free Switching Circuits. *Journal of the ACM*, 4(1), pp. 47–62, 1957.

[69] H. Hulgaard, S. M. Burns, and G. Borriello. *Testing Asynchronous Circuits: A Survey.* Technical Report FR-35, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, USA, 1994.

[70] G. M. Jacobs and R. W. Brodersen. Self-Timed Integrated Circuits for Digital Signal Processing Applications. In R. W. Brodersen and H. S. Moscovitz, editors, *VLSI Signal Processing, III*, IEEE Press, 1988.

[71] P. Jain and G. Gopalakrishnan. Hierarchical Constraint Solving in the Parametric Form with Applications to Efficient Symbolic Simulation Based Verification. In *Proceedings International Conference on Computer Design*, IEEE Press, pp. 304–307, 1993.

[72] M. B. Josephs and J. T. Udding. An Overview of D-I Algebra. In T. N. Mudge, V. Milutinovic, and L. Hunter, editors, *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, pp. 329–338, 1993.

[73] S. Karthik, I. de Souza, J. T. Rahmeh, and J. A. Abraham. Interlock Schemes for Micropipelines: Application to a Self-Timed Rebound Sorter. In *Proceedings International Conference on Computer Design*, IEEE Press, pp. 393–396, 1991.

[74] W. H. Kautz. The Necessity of Closed Circuit Loops in Minimal Combinational Circuits. *IEEE Transactions on Computers*, C-19, pp. 162–166, 1970.

[75] K. Keutzer, L. Lavagno, and A. Sangiovanni-Vincentelli. Synthesis for Testability Techniques for Asynchronous Circuits. In *Proceedings of International Conference on Computer-Aided Design*, IEEE Press, pp. 326–329, 1991.

[76] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware, The Theory and Practice of Self-Timed Design.* John Wiley & Sons, 1994.

[77] Z. Kohavi. *Switching and Finite Automata Theory*, Second Edition. McGraw-Hill Book Company, 1978.

[78] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for Synthesis of Hazard-Free Asynchronous Circuits. In *Proceedings of the Design Automation Conference*, IEEE Press, pp. 302–308, 1991.

[79] L. Lavagno, C. W. Moon, R. K. Brayton, and A. Sangiovanni-Vincentelli. Solving the State Assignment Problem for Signal Transition Graphs. In *Proceedings of the Design Automation Conference*, IEEE Press, pp. 568–572, 1992.

[80] L. Lavagno and A. Sangiovanni-Vincentelli. Linear Programming for Optimum Hazard Elimination in Asynchronous Circuits. In *Proceedings International Conference on Computer Design*, IEEE Press, pp. 275–278, 1992.

[81] L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits.* Kluwer Academic Publishers, 1993.

[82] C. Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell System Technical Journal*, 38(4), pp. 985–999, 1959.

[83] T. Lengauer and S. Naher. An Analysis of Ternary Simulation as a Tool for Race Detection in Digital MOS Circuits. *Integration, The VLSI Journal*, 4, pp. 309–330, 1986.

[84] D. L. Lewis. *Finite-State Analysis of Asynchronous Circuits with Bounded Temporal Uncertainty.* Technical Report TR-15-89, Department of Computer Science, Harvard University, Cambridge, Massachusetts, USA, 1989.

[85] A. Liebchen and G. Gopalakrishnan. Dynamic Reordering of High Latency Transactions Using a Modified Micropipline. In *Proceedings International Conference on Computer Design*, IEEE Press, pp. 336–340, 1992.

[86] K. J. Lin and C. S. Lin. On the Verification of State-Coding in STGs. In *Proceedings of International Conference on Computer-Aided Design*, IEEE Press, pp. 118–122, 1992.

[87] K. J. Lin and C. S. Lin. A Realization Algorithm of Asynchronous Control Circuits from STG. In *Proceedings of EDAC*, IEEE Press, pp. 322–326, 1992.

[88] M. M. Mano. *Digital Design*, Second Edition. Prentice-Hall, 1991.

[89] A. J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. In C. A. R. Hoare, editor, *UT Year of Programming Institute on Concurrent Programming.* Addison-Wesley Publishing Company, 1989.

[90] A. J. Martin. The Limitations to Delay-Insensitivity in Asynchronous Circuits. In *Proceedings of the 6th MIT Conference on Advanced Research in VLSI*, The MIT Press, pp. 263–278, 1990.

[91] A. J. Martin. Synthesis of Asynchronous VLSI Circuits. Lecture Notes from the Summer School in Marktoberdorf, July 26–August 7, 1994. To appear in *Lecture Notes in Computer Science*, Springer-Verlag, 1994.

[92] E. J. McCluskey. Transients in Combinational Logic Circuits. In R. H. Wilcox and W. C. Mann, editors, *Redundancy Techniques for Computing Systems*, Spartan Books, pp. 9–46, 1962.

[93] E. J. McCluskey. Fundamental Mode and Pulse Mode Sequential Circuits. In C. M. Popplewell, editor, *Proceedings of the IFIP Congress 62*, North-Holland Publishing Company, pp. 725–730, 1963.

[94] E. J. McCluskey. *Logic Design Principles.* Prentice-Hall, 1986.

[95] P. C. McGeer and R. K. Brayton. *Integrating Functional and Temporal Domains in Logic Design.* Kluwer Academic Publishers, 1991.

[96] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley Publishing Company, 1980.

[97] T. H. Meng. *Synchronization Design for Digital Systems.* Kluwer Academic Publishers, 1991.

[98] T. H. Meng, R. W. Brodersen, and D. G. Messerschmitt. Automatic Synthesis of Asynchronous Circuits from High-Level Specifications. *IEEE Transactions on Computer-Aided Design*, CAD-8(11), pp. 1185–1205, 1989.

[99] G. A. Metze. An Application of Multi-Valued Logic Systems to Circuits. In *Proceedings of the Symposium on Circuit Analysis*, pp. 11-1–11-14, 1955.

[100] R. E. Miller. *Switching Theory, Volume 2: Sequential Circuits and Machines.* John Wiley & Sons, 1965.

[101] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[102] C. E. Molnar. Macromodular Computer Systems. In B. D. Waxman and R. W. Stacy, editors, *Computers in Biomedical Research*, Academic Press, pp. 45–85, 1974.

[103] C. E. Molnar, T. P. Fang, and F. U. Rosenberger. Synthesis of Delay-Insensitive Modules. In H. Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on VLSI*, Computer Science Press, pp. 67–86, 1985.

[104] E. F. Moore. Gedanken Experiments on Sequential Machines. In C. E. Shannon and J. McCarthy, editors, *Automata Studies, Annals of Mathematics Study 34*, Princeton University Press, Princeton NJ, pp. 129–153, 1956.

[105] M. Mukaidono. Regular Ternary Logic Functions—Ternary Logic Functions Suitable for Treating Ambiguity. In *Proceedings of the 13th Annual Symposium on Multiple-Valued Logic*, IEEE Press, pp. 286–291, 1983.

[106] D. E. Muller. *A Theory of Asynchronous Circuits.* Technical Report 66, Digital Computer Laboratory, University of Illinois, Urbana-Champaign, Illinois, USA, 1955.

[107] D. E. Muller and W. S. Bartky. A Theory of Asynchronous Circuits. In *Proceedings of an International Symposium on the Theory of Switching*, Annals of the Computation Laboratory of Harvard University, Harvard University Press, pp. 204–243, 1959.

[108] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4), pp. 541–580, 1989.

[109] S. M. Nowick. *Automated Synthesis of Burst-Mode Asynchronous Controllers.* PhD thesis, Department of Computer Science, Stanford University, Stanford, California, USA, 1993.

[110] S. M. Nowick and D. L. Dill. Automatic Synthesis of Locally-Clocked Asynchronous State Machines. In *Proceedings of International Conference on Computer-Aided Design*, IEEE Press, pp. 318–321, 1991.

[111] S. M. Nowick and D. L. Dill. Synthesis of Asynchronous State Machines Using a Local Clock. In *Proceedings of International Conference on Computer Design*, IEEE Press, pp. 192–197, 1991.

[112] S. M. Nowick and D. L. Dill. Exact Two-Level Minimization of Hazard-Free Logic with Multiple Input Changes. In *Proceedings of International Conference on Computer-Aided Design*, IEEE Press, pp. 626–630, 1992.

[113] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[114] D. Pountain. Computing without Clocks. *BYTE*, 18(1), pp. 145–150, 1993.

[115] F. P. Preparata and R. T. Yeh. *Introduction to Discrete Structures for Computer Science and Engineering.* Addison-Wesley Publishing Company, 1973.

[116] V. Ramachandran. Algorithmic Aspects of MOS VLSI Switch-Level Simulation with Race Detection. *IEEE Transactions on Computers*, C-35(5), pp. 462–475, 1986.

[117] F. U. Rosenberger, C. E. Molnar, T. J. Chaney, and T. P. Fang. Q-Modules: Internally Clocked Delay-Insensitive Modules. *IEEE Transactions on Computers*, C-37(9), pp. 1005–1018, 1988.

[118] L. Y. Rosenblum and A. V. Yakovlev. Signal Graphs: From Self-Timed to Timed Ones. In *Proceedings of International Workshop on Petri Nets*, pp. 199–206, 1985.

[119] J. Rubinstein, Jr., P. Penfield, and M. A. Horowitz. Signal Delay in RC Tree Networks. *IEEE Transactions on Computer-Aided Design*, CAD-2(3), pp. 202–211, 1983.

[120] A. Salomaa. *Theory of Automata.* Pergamon, 1969.

[121] H. M. J. L. Schols. *A Formalization of the Foam Rubber Wrapper Principle.* Master's thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1985.

[122] C-J. H. Seger. *Models and Algorithms for Race Analysis in Asynchronous Circuits.* PhD thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1988.

[123] C-J. H. Seger. The Complexity of Race Detection in VLSI Circuits. In C. L. Seitz, editor, *Proceedings of the 1989 Decennial Caltech Conference on VLSI*, Advanced Research in VLSI, The MIT Press, pp. 335–350, 1989.

[124] C-J. H. Seger. On the Existence of Speed-Independent Circuits. *Theoretical Computer Science*, 86(2), pp. 343–364, 1991.

[125] C-J. H. Seger. *Symbolic Bounded Delay Simulation.* Unpublished Manuscript, 1992.

[126] C-J. H. Seger and J. A. Brzozowski. Generalized Ternary Simulation of Sequential Circuits. *Theoretical Informatics and Applications*, 28(3/4), pp. 159–186, 1994.

[127] S. F. Smith and A. E. Zwarico. Provably Correct Synthesis of Asynchronous Circuits. In J. Staunstrup and R. Sharp, editors, *Proceedings of the Second IFIP Workshop on Designing Correct Circuits*, North-Holland Publishing Company, pp. 237–260, 1992.

[128] J. Sparsø, C. D. Nielsen, L. S. Nielsen, and J. Staunstrup. Design of Self-Timed Multipliers: A Comparison. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies, IFIP Transactions*, A-28, Elsevier Science Publishers, pp. 165–179, 1993.

[129] R. F. Sproull and I. E. Sutherland. *Asynchronous Systems, Volume I: Introduction*. Technical Report 4706, Sutherland, Sproull & Associates, Inc., Palo Alto, California, USA, 1986.

[130] R. F. Sproull, I. E. Sutherland, and C. E. Molnar. The Counterflow Pipeline Processor Architecture. *IEEE Design and Test of Computers*, 11(3), pp. 48–59, 1994.

[131] L. J. Stockmeyer. *The Complexity of Decision Problems in Automata Theory and Logic*. PhD thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 1974.

[132] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6), pp. 720–738, 1989.

[133] J. T. Udding. A Formal Model for Defining and Classifying Delay-Insensitive Circuits and Systems. *Distributed Computing*, 1(4), pp. 197–204, 1986.

[134] S. H. Unger. Hazards and Delays in Asynchronous Sequential Switching Circuits. *IRE Transactions on Circuit Theory*, CT–6, pp. 12–25, 1959.

[135] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, 1969.

[136] K. van Berkel. *Handshake Circuits*. Cambridge University Press, 1993.

[137] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. De Man. Time and Area Performant Synthesis of Asynchronous Control Circuits. In *Proceedings of Tau 90*, Participant's Proceedings, 1990.

[138] P. Vanbekbergen, B. Lin, G. Goossens, and H. De Man. A Generalized State Assignment Theory for Transformations on Signal Transition Graphs. In *Proceedings of International Conference on Computer-Aided Design*, IEEE Press, pp. 112–117, 1992.

[139] T. Verhoeff. Delay-Insensitive Codes—An Overview. *Distributed Computing*, 3(1), pp. 1–8, 1988.

[140] T. Verhoeff. *Characterizations of Delay-Insensitive Communication Protocols*. Computing Science Notes 89/06, Department of Mathematics and Computer Science, Eindhoven Univ. of Technology, Eindhoven, The Netherlands, 1989.

[141] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley Publishing Company, 1985.

[142] T. E. Williams. *Self-Timed Rings and their Application to Division.* PhD thesis, Department of Computer Science, Stanford University, Stanford, California, 1991.

[143] T. E. Williams and M. A. Horowitz. A Zero-Overhead Self-Timed 160ns 54b CMOS Divider. *IEEE Journal of Solid-State Circuits*, 26(11), pp. 1651–1661, 1991.

[144] D. Wood. *Theory of Computation.* Harper and Row, Publishers, 1987.

[145] A. V. Yakovlev. On Limitations and Extensions of STG Model for Designing Asynchronous Control Circuits. In *Proceedings of International Conference on Computer Design*, IEEE Press, pp. 396–400, 1992.

[146] M. Yoeli. *Net-Based Synthesis of Delay-Insensitive Circuits*. Technical Report 609, Department of Computer Science, Technion, Haifa, Israel, 1990.

[147] M. Yoeli and I. Reicher. *Synthesis of Delay-Insensitive Circuits Based on Marked Graphs*. Technical Report 543, Department of Computer Science, Technion, Haifa, Israel, 1989.

[148] M. Yoeli and S. Rinon. Application of Ternary Algebra to the Study of Static Hazards. *Journal of the ACM*, 11(1), pp. 84–97, 1964.

[149] K. Yun and D. L. Dill. Automatic Synthesis of 3D Asynchronous State Machines. In *Proceedings of International Conference on Computer-Aided Design*, IEEE Press, pp. 576–580, 1992.

[150] K. Yun, D. L. Dill, and S. M. Nowick. Synthesis of 3D Asynchronous State Machines. In *Proceedings of International Conference on Computer Design*, IEEE Press, pp. 346–350, 1992.

List of Figures

List of Tables

List of Mathematical Concepts

List of Algorithms

List of Corollaries

List of Definitions

List of Lemmas

List of Propositions

List of Theorems

Index

An entry for a name refers to a page on which either the name is mentioned or an author's work is cited.