Testing Delay-Insensitive Circuits

Thesis by

Pieter Johannes Hazewindus

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California, USA

1992

Defended 20 May 1992

Aan mijn ouders

Acknowledgements

Niet over rozen gaat het pad
Maar zonder vrienden kan ik niet.
— Boudewijn de Groot

Thanking people is my favorite vice.
— Allan Gurganus

Now the fearful trip is done. I have many people to thank for shaping the almost seven years of my stay at Caltech, and its long-awaited completion. The following is but a partial list. Thanks to anyone whom I forgot to mention.

I want to express my gratitude to my advisor, Alain Martin, who invented the delay-insensitive synthesis method, and suggested I look into testing for delay-insensitive circuits. He has guided the development of my research. His suggestions, comments, and criticisms were invaluable. Thank you very much.

My thanks to my thesis committee, Alain Martin, Yaser Abu-Mostafa, Chuck Seitz, Jerry Sussman, and Jan van de Snepscheut, for the time spent reading my thesis, and for suggesting such useful corrections.

My thanks to DARPA for funding my research.

My thanks to Chuck Seitz, who introduced this mathematician to circuit design. He helped Steve Burns, Andy Fyfe, and me develop a standard-cell route-and-placement system, with which our first chips were designed. This system reduced design time considerably, and was a major step forward toward our proving the practicality of delay-insensitive circuit design. Thanks also for the use of the narrow-pitch pads.

My thanks to Mani Chandy, for radically changing my thinking on concurrency and parallel programming.

My thanks to Yaser Abu-Mostafa, for being my best teacher at Caltech. I always learned something from his lectures, even when I already knew the material.

My thanks to Martin Rem, for his help with preparing me for Caltech, and for his help and support during my studies.

My thanks to Steve Burns for the many fruitful discussions on circuit design, from the first day until (almost) the last, and for posing the difficult questions that needed to be answered.

My thanks to the other graduate students in my research group, Dražen Borković, Marcel van der Goot, Tony Lee, Christian Nielsen, and José Tierno, for the discussions and suggestions during many group meetings, and for proofreading this thesis.

My thanks to my officemates for long periods of absence, which gave me a quiet environment in which to work, and for eventually showing me how to complete the work.

My thanks to Rajiv Gupta for his patient ears, for his encouragement, and for his hospitality.

My thanks to David Schweizer, who would bristle at being called a bastion of sanity, but who was one for me. He knew where my towel was.

My thanks to Andy Fyfe for his early work on the route-and-placement system, for his efforts at coercing LaTeX into conforming to Caltech thesis standards, and for his understanding of the vagaries of the computer system.

My thanks to Jim Boyk for his music, for his epicurism, and for all the great conversations.

Finally, my thanks to my mother, my father, and my sister, Geertje, for their unwavering support while I took the road less traveled by. They have made all the difference.

> *"The thing you've got to realize", he said, "is that most of these guys, for all their computer wizardry, don't know very much about the English language. Some of them are positively subliterate."*
>
> — David Leavitt, *Equal Affections*

A special thanks to Dian De Sha, B.A., for diligently proofreading manuscripts, for sharing stories, and for moral support during difficult times. And yes, I know the comma ought to be inside the quotation marks.

Abstract

A method is developed to test delay-insensitive circuits, using the single stuck-at fault model. These circuits are synthesized from a high-level specification. Since the circuits are hazard-free by construction, there is no test for hazards in the circuit. Most faults cause the circuit to halt during test, since they cause an acknowledgement not to occur when it should. There are stuck-at faults that do not cause the circuit to halt under any condition. These are *stimulating* faults; they cause a premature firing of a production rule. For such a stimulating fault to be testable, the premature firing has to be propagated to a primary output. If this is not guaranteed to occur, then one or more test points have to be added to the circuit. Any stuck-at fault is testable, with the possible addition of test points. For combinational delay-insensitive circuits, finding test vectors is reduced to the same problem as for synchronous combinational logic. For sequential circuits, the synthesis method is used to find a test for each fault efficiently, to find the location of the test points, and to find a test that detects all faults in a circuit.

The number of test points needed to fully test the circuit is very low, and the size of the additional testing circuitry is small. A test derived with a simple transformation of the handshaking expansion yields high fault coverage. Adding tests for the remaining faults results in a small complete test for the circuit.

Contents

List of Figures

CHAPTER 1
Introduction

Delay is preferable to error
— Thomas Jefferson

In this thesis, I develop a method to test delay-insensitive circuits that have been synthesized from a high-level specification. The fault model is the single stuck-at fault model. A test is derived from the high-level specification of the circuit. Most, but not all, faults cause the circuit to halt during test. For faults that are not guaranteed to be detected by the environment, one or more test points have to be added. The number of test points needed to fully test the circuit is typically small, as is the added testing circuitry. Deriving a test from the high-level specification yields high fault coverage.

Most VLSI circuits currently produced are synchronous circuits; a central clock provides a synchronization mechanism for the computation performed in a system. As the feature size of circuits decreases, the complexity and speed increases. It is becoming difficult to translate this into improved performance. Since each clock pulse has to be distributed over the entire circuit, clock skew is a major obstacle to shortening of the clock cycle.

A way to avoid problems with clocks is to design circuits without clocks – asynchronous circuits. Few asynchronous circuits have been designed, since the design process was not well understood. It was felt that these circuits are necessarily too big and too slow – if one can even overcome the problem of hazards and critical races.

Alain Martin has constructed a simple and powerful method for the design of *delay-insensitive* circuits [46, 47]. This type of circuit is a subclass of asynchronous circuits, where propagation delays in wires and gates are assumed to be arbitrary and unbounded (but finite). The delay-insensitive circuits are synthesized from a high-level specification. The specification is proven correct, whereupon a series of semantics-preserving operations are performed, to obtain a delay-insensitive circuit. The resulting circuits are free of hazards, since the synthesis method guarantees the absence of hazards, by construction.

At Caltech, we have designed and fabricated a number of delay-insensitive circuits, including a stack, a mutual exclusion circuit, a multiplier, a router, and a microprocessor [42, 50, 51]. These circuits have been reasonably fast, very robust to variations in temperature, voltage, and fabrication parameters, and have been fully functional on "first silicon".

We have shown that it is possible to design delay-insensitive circuits; that it is possible to design them reliably; and that delay-insensitive circuits can be used for a wide range of applications. A subsequent problem is to show that they can be tested. In this thesis, I investigate how to test delay-insensitive circuits using the stuck-at fault model.

The stuck-at fault model is a widely used model to derive tests for VLSI circuits. Many methods are known to generate minimal tests efficiently for testing synchronous circuits; efforts to adapt these methods to delay-insensitive circuits have not been very successful. I think this is, again, because the discipline of designing delay-insensitive circuits was not well understood, and because most prior efforts consisted of adapting methods for testing synchronous circuits to asynchronous ones. Since the workings of a delay-insensitive circuit – and the methods to derive it – are so different from synchronous circuits, I believe that a different paradigm is needed for the test generation problem for delay-insensitive circuits. Rather than adapt any method for synchronous circuits, I investigate the testing problem based on the synthesis method with which the circuit was generated.

A traditional concern regarding testing of delay-insensitive circuits is the difficulty of testing for hazards and critical races. In order for a hazard to occur, there are constraints on some delays in the circuits; these delays vary with voltage, temperature, and fabrication variances. Moreover, when a hazard occurs in an internal node, it has to be propagated to a primary output, so that the result can be observed. This is exceedingly difficult.

I believe that the testing stage is the wrong level at which to analyze hazards. It is possible to design circuits that are hazard-free, using Martin's synthesis method. Such circuits function correctly regardless of voltage, temperature, and fabrication variances.

The traditional model for synchronous circuits is that of combinational logic to which clocked latches are added to provide feedback; the traditional method to test these circuits is to add test circuitry to the latches, so that the resulting circuit is feedback-free. The emphasis in testing synchronous circuits is on generating efficient tests to test combinational logic.

A similar approach has been proposed for delay-insensitive circuits [33, 61], but is impractical. The ratio of combinational gates to state-holding gates is much smaller than for synchronous circuits; to add test circuitry for each state-holding element in a delay-insensitive circuit is prohibitively expensive. Instead I analyze the circuit as is, and add test circuitry as necessary. The emphasis therefore is on testing sequential circuits.

The fault model I use is the *single stuck-at fault model.* In this model, a faulty circuit has one fault; an input or an output of a gate is either permanently at a high voltage (stuck-at-1), or at a low voltage (stuck-at-0). The stuck-at fault model is not a realistic model of actual faults in a circuit. It is widely used, however, since it is conceptually simple, and since there is a strong correlation between faulty circuits and circuits that are rejected with a test for stuck-at faults [5]. A discussion of the accuracy of the stuck-at fault model is beyond the scope of this thesis.

It is of great value in the fault analysis that each delay-insensitive circuit is synthesized from a high-level specification. It is possible to analyze a circuit from just the gate-level specification; the search for test vectors is greatly facilitated if the specification of the circuit is known. For example, the instruction fetch process of the asynchronous microprocessor [51] consists of an if-statement with two cases. A test derived from the specification, consisting of executing each case of the if-statement once, detects 93 of 100 stuck-at faults in the circuit. Deriving the same test from just the gate-level specification of the circuit is vastly more difficult. I use the *handshaking expansion* as the specification of the circuit. This notation describes the sequence of actions on variables in the specification.

Some authors claim that for a delay-insensitive circuit any stuck-at fault causes the circuit to halt [7]. Consequently, it is not necessary to test a delay-insensitive circuit, since even a faulty circuit will never exhibit faulty behavior, and no test circuitry needs to be added to the circuit. I show that this is not the case for the general stuck-at fault model; some faults on inputs of gates never cause the circuit to halt. It is therefore necessary to test delay-insensitive circuits, and for some circuits test points need to be added to make them fully testable [52].

1. Organization of the Thesis

In the next chapter, I outline the synthesis method for delay-insensitive circuits. The high-level description language is akin to Hoare's CSP (Communicating Sequential Processes). I describe the four-phase handshaking protocol with which processes communicate, and the resulting *handshaking expansion*, which is the notation from which tests are derived. The circuit itself is described in the form of a *production rule set.*

Chapter 3 is a description of the general problem of testing delay-insensitive circuits. I discuss the delay assumptions; I describe the two types of stuck-at faults, the inhibiting fault and the stimulating fault; I show that for each fault there is a state in the handshaking expansion where the fault causes a production rule to be inhibited or to fire prematurely; I derive conditions under which a fault inhibits a production rule, or causes a premature firing that is detected by the environment. From these conditions follow a series of simple theorems with which the testability of most faults in a circuit is trivially shown.

Chapter 4 concerns testing of datapaths. A delay-insensitive datapath differs from synchronous combinational logic, since it has state-holding elements; also, it is not implemented as a network of standard gates (AND, OR, inverter), but rather with fewer, more complex gates. The problem is the generation of a reasonably small test set with which to test all testable faults. I show that such a delay-insensitive circuit can be transformed into a circuit with only combinational gates; a test that detects all testable faults for this new circuit will also detect all testable faults in the delay-insensitive circuit, with one exception. I show that the D-algorithm can be adapted for use in delay-insensitive circuits.

In chapter 5 I explore some testing issues that are more technology-dependent, such as the initialization of the circuit, and faults causing interfering production rules. I also derive a design for test circuitry. This design does not use a clock, but it is necessarily not fully delay-insensitive.

Finally, chapter 6 describes heuristics on how to derive a small test set from the test vectors for each individual fault. I describe a refinement of the stuck-at fault model, a model with stuck-on and stuck-open faults, and explain how to test circuits under this new model.

In appendix A are algorithms to test delay-insensitive circuits. Appendix B is an introduction to circuit testing.

2. A Note on Notation

In the abstract reasoning about circuits, a low voltage is called 0 or **false**, and a high voltage 1 or **true**. The delay-insensitive circuit synthesis method is described entirely using **false** and **true**. I have followed this convention as much as possible. All boolean conditions are expressed with "$\wedge$" and "$\vee$", rather than "." and "+".

If I were consistent, I would call the different faults stuck-at-false and stuck-at-true; instead, I have used the standard terms stuck-at-0 and stuck-at-1. For the description of the D-algorithm and the adaptation for delay-insensitive circuits I have also used 0 and 1 (in addition to X, D, and $\overline{\mathrm{D}}$), because these notations are well-known.

In boolean expressions, $\wedge$ has higher binding power than $\vee$.

Throughout the thesis, I refer to circuits as delay-insensitive. For such circuits, delays (both for gates and for wires) are assumed to be unbounded (but finite) and arbitrary. An exception is the *isochronic fork*, for which the delays in all branches of the fork are assumed to be roughly the same. Such an assumption is necessary to be able to construct any interesting circuit. Because of the isochronic fork assumption, this class of circuits is sometimes called *quasi-delay-insensitive*.

CHAPTER 2
Synthesis of Delay-Insensitive Circuits

La chose importante, c'est la théorie, qui est mind-blowing. Et si j'ai raison, ma théorie va produire un crisis en world thinking et, avec luck, un Prix Nobel. Cela sera beau, n'est-ce pas?

—Miles Kington, *L'Origine des Species, Dans une Version Complètement Modernisée*

1. Introduction

In this chapter I explain parts of the high-level synthesis method for delay-insensitive circuits that was developed by Martin [41, 42, 44, 45, 46, 47]. The high-level specification for circuits is in a language that is based on Hoare's CSP (Communicating Sequential Processes) [31]. There are several other high-level synthesis methods [17, 55, 70].

The organization of this chapter is as follows. I define gates and circuits; I explain the constructs of the high-level specification language, and the steps of the synthesis method that lead to a delay-insensitive circuit. The first step is at the program level, namely a decomposition of processes into smaller subprocesses. The next step is the generation of a *four-phase handshaking expansion* that establishes a communication protocol between processes. For communications where no data are sent (*synchronization channels*) this is a straightforward transformation. For *communication channels*, where data are sent, I list several ways to encode data values into bits for use in delay-insensitive circuits. The third step is to transform the handshaking expansion so that each state is unique. This is done by *reshuffling* of actions and by the addition of *state variables.* The final transformation is from a handshaking expansion, where each state is unique, to a *production rule set.*

In addition to these steps of the synthesis method, I explain how a delay-insensitive circuit operates as a series of actions, where each action is *acknowledged* with the next action. For actions that are not acknowledged, the assumption that all delays are arbitrary and unbounded has to be restricted. I introduce *isochronic forks* as a way to insure correctness of circuits for which some actions are not acknowledged.

2. Gates and Circuits

A circuit is a network of gates that interacts with its environment.

DEFINITION 2.1 (GATE). *A* gate *is a circuit element with one or more inputs, and one output. It is described as a pair of* production rules

$$\begin{cases} B_u & \rightarrow & z\uparrow \\ B_d & \rightarrow & z\downarrow, \end{cases}$$

where B_u and B_d are boolean expressions on the inputs of the gate (known as guards*), and z is the output. If condition B_u holds, output z becomes* **true** *(denoted $z\uparrow$), and if condition B_d holds, output z becomes* **false** *(denoted $z\downarrow$).*

An execution of a production rule is called a *firing.* If $z\uparrow$ ($z\downarrow$) fires in a state where $\neg z$ (z) holds, then the firing is *effective*, otherwise it is *vacuous.* Unless otherwise noted, I shall only consider effective firings in the sequel.

FIGURE 2.1. Schematic representation of AND gate and C-element

A two-input AND gate with inputs a and b and output z has production rules:

$$\left\{ \begin{array}{rcrcl} a & \wedge & b & \rightarrow & z\uparrow \\ \neg a & \vee & \neg b & \rightarrow & z\downarrow . \end{array} \right.$$

A Muller C-element [57, 58] with inputs a and b and output z has production rules:

$$\left\{ \begin{array}{rcrcl} a & \wedge & b & \rightarrow & z\uparrow \\ \neg a & \wedge & \neg b & \rightarrow & z\downarrow . \end{array} \right.$$

See figure 2.1.

The guards of the production rules of a gate have to be mutually exclusive. Otherwise it is possible that both production rules can fire at the same time. In an actual implementation this causes a short circuit.

DEFINITION 2.2 (NON-INTERFERENCE). *For a gate with production rules*

$$\left\{ \begin{array}{lcl} B_u & \rightarrow & z\uparrow \\ B_d & \rightarrow & z\downarrow, \end{array} \right.$$

condition $\neg B_u \vee \neg B_d$ *has to hold at any time. This is known as* non-interference.

Note that $\neg B_u \vee \neg B_d$ is not necessarily a tautology. For instance an SR flip-flop with output q has production rules

$$\left\{ \begin{array}{lcl} s & \rightarrow & q\uparrow \\ r & \rightarrow & q\downarrow . \end{array} \right.$$

This flip-flop has to be used in such a way that s and r are not both **true** at the same time.

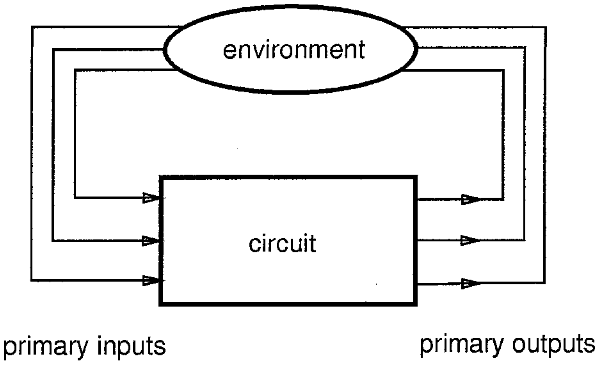

FIGURE 2.2. A circuit and its environment

If $B_u \equiv \neg B_d$, then the gate is a *combinational gate.* All other gates are *state-holding elements.* Examples of combinational gates are the AND gate, the OR gate, the XOR (exclusive-or) gate, and the inverter. The C-element and the SR flip-flop are both state-holding elements.

DEFINITION 2.3 (CIRCUIT). *A* circuit *is an interconnection of gates, interacting with its* environment. *Each input of a gate is either connected to the output of another gate, or to the environment. An output of a gate may be connected to any number of inputs of other gates, and to the environment.*

An input of a gate that is connected to the environment is a *primary input*; an output of a gate that is connected to the environment is a *primary output.* An output (of a gate) that is input to more than one gate is said to *fork*, as in the implementation there is a forking wire.

The environment of a circuit is also a circuit. The primary inputs of the environment of circuit C are the primary outputs of C; the primary outputs of the environment are the primary inputs of C (figure 2.2). The environment changes its primary outputs in such a way that circuit C operates according to specification. The environment never malfunctions. For most circuits, I shall not give an explicit description of the environment as a set of gates; rather, there is a spec-

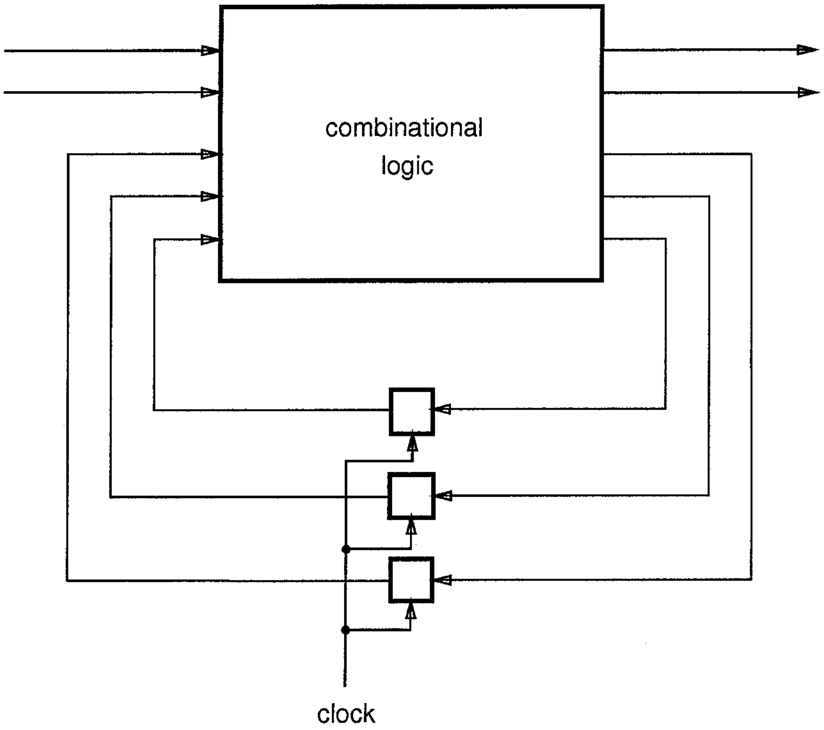

FIGURE 2.3. A synchronous sequential circuit, with clocked state-holding elements

ification of the environment, relating the primary outputs to the primary inputs of the environment. I assume that there is a gate-level implementation of such a specification.

A circuit can be represented as a directed graph G, where each node is a gate. If the output of gate g_0 is an input to gate g_1, then there is an arc from node g_0 to node g_1. If the graph representation is acyclic, then the circuit is *feedback-free.*

For *synchronous* (or *clocked*) circuits each state-holding element is a clocked memory element. See figure 2.3. Operation of a synchronous circuit is as follows. During each cycle, the value of the output of each memory element is set, and the environment sets the primary inputs. The circuit fires production rules until there are no more effective firings. Then the values of the primary outputs are sent to the environment, and the value of the output of each memory element is set to the value of its input. A convenient way to describe a synchronous sequential circuit is as a finite-state machine (FSM) [30, 38].

For *asynchronous* (or *delay-insensitive*) circuits there is no clocking mechanism to distinguish between cycles. Operation is as follows. Given the value of the output of each gate, and the value of each primary input, repeatedly fire any production rule that can fire. The primary inputs may be changed as long as the handshaking protocol is obeyed. For correct operation, each production rule has to be *stable.*

DEFINITION 2.4 (STABILITY). *Production rule* $B_u \rightarrow z\uparrow$ *is* stable *if, whenever* $B_u \wedge \neg z$ *holds,* B_u *remains* **true** *until* $z\uparrow$ *has fired, that is, until* z *is* **true**. *Similarly for* $z\downarrow$.

A common restriction on the usage of an asynchronous circuit is that it can only be used in *Fundamental Mode*, that is, a primary input may only change value if there is no production rule in the circuit that can fire [57]. All delay-insensitive circuits in this thesis operate correctly even when a primary input changes value concurrently with the firing of a production rule in the circuit, as long as stability of each production rule is guaranteed.

3. The High-Level Specification

The high-level specification, from which a delay-insensitive circuit is synthesized, is based on Hoare's CSP [31]. A program consists of one or more concurrent processes. Each process itself is described with a sequential program. There are no shared variables. Instead, processes communicate values via *channels.* Channels are also used as a means of synchronization between processes. A channel connects two processes, and a communication on a channel is slackless [32].

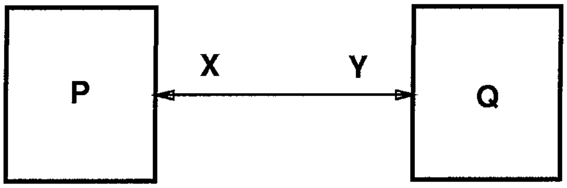

FIGURE 2.4. Channel (X, Y) between processes P and Q

Let (X, Y) be a channel between processes P and Q. See figure 2.4. In process P, X is known as a *port.* Since the communication is slackless, at any time the number of completed X actions in P is the same as the number of completed Y actions in Q. If process P attempts to do a communication X, while process Q cannot do a communication Y, then process P is *suspended* until Q does a communication Y. Both X and Y then complete at the same time. As a progress condition, X and Y cannot both be suspended at the same time.

In a digital circuit, each variable is either at a high voltage or at a low voltage. Therefore all variables in the language are boolean variables. It is sometimes advantageous to abbreviate a vector of boolean values with a single identifier, for instance when describing arithmetic operations. It is necessary to describe how each integer value is mapped onto a boolean vector. In section 5 I describe some of the systematic mappings from integers to booleans for delay-insensitive circuits.

3.1. Language Constructs. The constructs in the high-level specification are:

- Parallel composition for processes. The parallel execution of processes P and Q is denoted "$P||Q$". Parallel composition is associative and commutative.
- Sequential composition of statements S and T is denoted "S;T". Sequential composition is associative.

- Parallel composition of atomic statements s and t is denoted "s,t". Parallel composition is associative and commutative.
- Assignment statement. The assignment operator is ":=". For a boolean variable b the assignment to **true** is abbreviated to $b\uparrow$ and the assignment to **false** is abbreviated to $b\downarrow$.
- Selection. The selection command consists of a number of *guards* (boolean expressions on the variables in the program), $G_0, G_1, \ldots, G_{n-1}$, and an equal number of program parts, $S_0, S_1, \ldots, S_{n-1}$, denoted

  $$[G_0 \rightarrow S_0 | G_1 \rightarrow S_1 | \ldots | G_{n-1} \rightarrow S_{n-1}].$$

  A program part S_i is executed for which G_i evaluates to **true**. If more than one guard is **true**, a nondeterministic choice is made; if no guard evaluates to **true**, then the process is suspended until a guard is **true**.
- Repetition. The repetition command also consists of a number of guards and an equal number of program parts, and is denoted

  $$*[G_0 \rightarrow S_0 | G_1 \rightarrow S_1 | \ldots | G_{n-1} \rightarrow S_{n-1}].$$

  As long as there is a guard that evaluates to **true**, a program part S_i is executed for which G_i is **true**. If no guard holds, then the repetition terminates.
- Send and receive on a port. Let process P have a port X. Sending the value of s on port X is denoted $X!s$, and $X?t$ denotes a receiving communication on port X whereby the value received is stored in t.
- Probed communication [43]. Let (X, Y) be a channel between processes P and Q. For process P, the probe of X (denoted $\overline{X}$) is a boolean condition that holds when process Q is suspended on a communication Y.

In addition, the only form of recursion in the language is tail recursion.

There are a few abbreviations for constructs. The program part $*[\mathbf{true} \rightarrow S]$, for indefinite repetition of statement S, is abbreviated to $*[S]$, and the program part $[G \rightarrow \mathit{skip}]$ (where a process waits for condition G to hold) is abbreviated to $[G]$. A frequently used notation is $*[[G_0 \rightarrow S_0 | G_1 \rightarrow S_1]]$: wait until either G_0 or G_1 holds, then execute an S_i for which G_i holds, and repeat the program part.

Some channels are used solely to synchronize two processes. For process P with a port X of such a *synchronization channel* a communication is simply written "X". Since no data are transmitted, there is no difference between a send and a receive action for a synchronization channel.

3.2. Examples of Programs. A buffer is a FIFO queue. It consists of a number of processes. Each process has two ports, L to a channel connecting the process to its left neighbor, and R to a channel connecting it to its right neighbor. The buffer repeatedly communicates to its left neighbor, then communicates to its right neighbor. If each channel is a synchronization channel then a program for a buffer process is

$$*[L; R].$$

If the buffer sends and receives values, there is one variable, x:

$$*[L?x; R!x].$$

As an example of a probed communication, a buffer that sends the value stored to its right neighbor before it receives a new value has program:

$$*[[\overline{L} \rightarrow R!x, L?x]].$$

3.3. Program Decomposition. At the program level, it is often advantageous to split a large process up into two or more smaller processes. A number of channels are introduced to connect these subprocesses, while the existing channels to other processes remain the same.

Process decomposition is done using the

Decomposition Rule: A process, P, containing an arbitrary program part, S, is semantically equivalent to two processes, $P1$ and $P2$, where $P1$ is derived from P by replacing S with a communication action, C, on the newly introduced channel (C, D) between $P1$ and $P2$, and $P2$ is the process $*[[\overline{D} \rightarrow S; D]]$ [47].

If a process is decomposed into several processes using the decomposition rule, then these subprocesses are never active concurrently. Therefore the subprocesses may use shared variables.

4. The Handshaking Expansion

The next step in the synthesis of a delay-insensitive circuit from a high-level specification is the transformation from a program to a *handshaking expansion.* In the handshaking expansion, the communications with neighboring processes are specified by actions on ports, using the *handshaking variables.*

Let (X, Y) be a synchronization channel connecting processes P and Q. Port X in P consists of two variables, output xo and input xi, and port Y in Q consists of output yo and input yi. The channel connects the ports with two (directed) wires, from xo to yi, and from yo to xi.

The implementation of a communication on the channel is asymmetric. One port initiates the communication (it sends a "request"), and the other reacts (it sends an "acknowledgement"). The first is known as the *active* port, the latter as the *passive* port. Without loss of generality, assume that X is active, and Y passive. Then each communication X in the program is implemented as the sequence

$$xo\uparrow; [xi]; xo\downarrow; [\neg xi],$$

and each communication Y in the program is implemented as the sequence

$$[yi]; yo\uparrow; [\neg yi]; yo\downarrow .$$

Initially all variables of the channel are **false**. Process P initiates the communication by raising xo. After Q has detected this (when yi is **true**), it acknowledges by raising yo. Then process P resets xo to **false**, after which process Q resets yo to **false**. This communication protocol is known as a *four-phase handshake.*

Notice that there is no concurrency between the processes while a synchronization on the channel is executed. The sequence of events during a communication is:

$$xo\uparrow; [yi]; yo\uparrow; [xi]; xo\downarrow; [\neg yi]; yo\downarrow; [\neg xi].$$

An alternative implementation for the active port is as a *lazy-active* port [11, 47], where the final wait-action is postponed until the next communication on the channel:

$$X \equiv [\neg xi]; xo\uparrow; [xi]; xo\downarrow .$$

4.1. The Probe. If process P initiates a communication X, and Q does not immediately acknowledge with a communication Y, then P is suspended. Process Q detects that P is suspended when yi holds. Therefore an implementation of $\overline{Y}$ is the condition yi. If a port is probed, it must be implemented as a passive port.

4.2. Two-Phase Handshaking. In the four-phase handshaking expansion above, the final two actions in both X and Y are not necessary to achieve a synchronization between processes. These resetting actions can be useful, however, since they insure that for each communication the initial state of all variables is **false**.

It is possible to expand the odd-numbered communications as

$$X \equiv xo\uparrow; [xi]$$
$$Y \equiv [yi]; yo\uparrow,$$

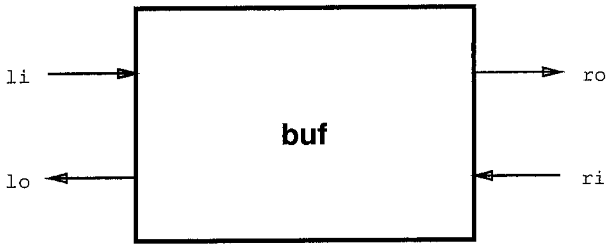

FIGURE 2.5. Buffer process with left and right ports

and the even-numbered communications as

$$X \equiv xo\downarrow; [\neg xi]$$
$$Y \equiv [\neg yi]; yo\downarrow .$$

This is known, for obvious reasons, as a *two-phase handshake.*

In the sequel I assume that the communication protocol is a four-phase handshake. The testing results are the same for each type of handshaking protocol.

4.3. Examples.

EXAMPLE 2.1. *For the buffer process* $*[L; R]$*, with* L *and* R *synchronization channels, let* L *be a passive, and* R *an active port. Port* L *has variables* lo *and* li*, port* R *has variables* ro *and* ri*. See figure* 2.5*. The handshaking expansion for this buffer is:*

$$*[[li]; lo\uparrow; [\neg li]; lo\downarrow; ro\uparrow; [ri]; ro\downarrow; [\neg ri]].$$

EXAMPLE 2.2. *For the process* $*[[\overline{L} \rightarrow R; L]]$ *port* L *is probed, and therefore has to be a passive port. If* R *is an active port the handshaking expansion is:*

$$*[[li \rightarrow ro\uparrow; [ri]; ro\downarrow; [\neg ri]; lo\uparrow; [\neg li]; lo\downarrow].$$

5. The Handshaking Expansion for Communication Channels

To implement a communication channel (which can transmit data values), more than two wires are needed. Data values are transmitted using communication channels by raising one or more wires of the channel. For instance, consider a one-bit wide data channel A that is implemented with two wires, $a0$ and $a1$, in one direction, and one acknowledgment wire in the other. To transmit value "0" over channel A, wire $a0$ is raised, and to transmit value "1" wire $a1$ is raised.

For any test that detects all stuck-at faults, every wire in the circuit has to be raised at least once. Therefore both $a0$ and $a1$ have to be raised during such a test (but not simultaneously). An equivalent formulation is to require that a "0" and a "1" be sent over channel A during the test. For an arbitrary communication channel a set of data values has to be sent such that each wire of the channel is raised at least once. I call such a set a complete set:

Definition 2.5 (Complete Set of Data Values). *Let A be a communication channel, where each value is encoded on N wires, and let S be the set of values that can be sent over channel A. Let $w_j(s)$ be the value of the jth wire in the encoding of s. Set S_c is called a* complete set of data values *if*

$$\forall_{0 \le j < N} \exists_{s \in S_c} : w_j(s).$$

For the one-bit wide channel above $S = S_c = \{0, 1\}$. In general, the smallest complete set of data values for a communication channel is much smaller than the set of all values that can be sent over that channel. I examine a few standard ways to encode data values.

5.1. Dual-rail Encoding. For the dual-rail encoding scheme two wires are used for each bit in the binary representation of a number. If the jth bit is 0, wire $2j$ is raised, if it is 1, then wire $2j + 1$ is raised. For N-bit numbers the channel consists of $2N$ data wires. With $S = \{0, 1, \ldots, 2^N - 1\}$ a complete set of data values is $\{0, 2^N - 1\}$. Another is $\{2^m - 1, 2^N - 2^m\}$, for $N \ge m$. Dual-rail encoding is a widely used coding scheme.

EXAMPLE 2.3. *For a four-bit wide dual-rail encoded channel eight wires are used. If codewords are represented in the form of a string, then 0 might be encoded as 01010101, 1 as 01010110, etc. There are 16 different codewords, but a minimal complete set of data values has only two codewords, for example* $S_c = \{0, 15\}$*, corresponding to codewords 01010101 and 10101010, or* $S_c = \{5, 10\}$*, corresponding to codewords 01100110 and 10011001.*

5.2. One-hot Encoding. If one wire is used per data value that can be transmitted, then the values are said to be one-hot encoded. The only complete set of data values is the set of all data values ($S_c = S$). One-hot coding is only practical for narrow channels.

EXAMPLE 2.4. *For a four-bit wide one-hot encoded channel sixteen wires are necessary, as there are sixteen different codewords. For instance 0 is encoded as 0000000000000001, 1 as 0000000000000010, etc. The complete set of data values is* $\{0, 1, \ldots, 15\}$.

5.3. k-out-of-N Encoding. The one-hot encoding scheme is a special case of so-called k-out-of-N codes [71]. To transmit a value over an N-bit wide channel, k wires are raised. A complete set of data values has at least $\lceil N/k \rceil$ elements. Another special case is an N-out-of-$2N$ code, where N wires are raised out of a total of $2N$ wires. A minimal complete set of data values has at least two elements, for instance the value corresponding to raising the first N wires and the value corresponding to raising the remaining N wires.

EXAMPLE 2.5. *To encode four-bit values on seven wires a 2-out-of-7 code might be used. This code has 21 possible codewords:* $\{0000011, 0000101, \ldots, 1100000\}$*. Another encoding uses six wires, with a 3-out-of-6 code. It has 20 different codewords:* $\{000111, 001011, 001101, \ldots, 111000\}$*. A complete set of data values to be sent over the channel depends on the mapping of values to codewords (there are* *more codewords than values to be encoded). For a 2-out-of-7 code a complete set of data values might be* $\{0000011, 0001100, 0110000, 1100000\}$*, and for a 3-out-of-6 code* $\{000111, 111000\}$.

For other encoding schemes, such as Berger codes [8, 23], it is easy to compute a minimal complete set of data values. For efficient encoding schemes [59], the minimal set is small, and its size independent of the size of the channel. For instance for dual-rail encoding and N-out-of-$2N$ encoding the size of a minimal complete set of data values is two.

For two-phase handshaking protocols, the encoding for data channels is similar to the schemes above, except that a wire is raised or lowered only once per communication. In the case of two-phase communications, there are different ways to interpret the encoded data. As an example, I describe two schemes for dual-rail encoded data.

Consider a one-bit wide channel, A, with data wires $a0$ and $a1$. The first scheme is analogous to the one used for the four-phase protocol. For each "0" sent wire $a0$ is toggled, and for each "1" sent wire $a1$ is toggled. This scheme is conceptually simple, but has the disadvantage that the values of $a0$ and $a1$ alone are not enough to decode the value sent. For instance, if $a0$ and $a1$ are both 1, then the last value sent on channel A is either 0, if $a0$ was raised last, or 1, if $a1$ was raised last.

For the second scheme, the receiving process does not have to store extra information to decode the value sent. To send a 0, set $a0$ to 0, and to send a 1, set $a0$ to 1. Furthermore, if a 0 is sent after a 0, or if a 1 is sent after a 1, then $a1$ is toggled. For each value sent, exactly one wire is toggled. The value sent is always $a0$.

The complete set of data values in the first scheme is $\{0, 1\}$. It is not possible to compute a complete set of data values for the second scheme, as the order in which values are sent over the channel is important. The channel is tested by sending either a 0 followed by another 0, or by sending a 1 followed by another 1.

6. Reshuffling and State Assignment

The goal of the synthesis method is to derive a delay-insensitive circuit, described as a production rule set. The production rule set should be equivalent to the handshaking expansion, that is, if a production rule can fire in the production rule set, then the corresponding transition in the handshaking expansion can occur, and vice versa. The main difference between the handshaking expansion and the production rule set is that there is explicit sequencing in the handshaking expansion (by means of semicolons), whereas the production rules may fire concurrently.

The task, then, is to remove the sequencing from the handshaking expansion to obtain an equivalent production rule set. Each state in the handshaking expansion must be unique. If two states are identical, one or more *state variables* have to be introduced.

EXAMPLE 2.6. *For the buffer of example* 2.2 *the handshaking expansion is:*

$$*[[li]; ro\!\uparrow; [ri]; ro\!\downarrow; [\neg ri]; lo\!\uparrow; [\neg li]; lo\!\downarrow].$$

There are two indistinguishable states: the state where transition $ro\!\uparrow$ *takes place, and the state where* $lo\!\uparrow$ *takes place. With the introduction of a state variable,* u*, each state is unique, with the following handshaking expansion:*

$$*[[li]; ro\!\uparrow; [ri]; u\!\uparrow; [u]; ro\!\downarrow; [\neg ri]; lo\!\uparrow; [\neg li]; u\!\downarrow; [\neg u]; lo\!\downarrow].$$

A second method to make each state in a handshaking expansion unique is to *reshuffle* actions in the handshaking expansion. For instance, in a buffer with L and R communications, the actions on port L and the actions on port R may be interleaved. A requirement is that the actions on each port not violate the four-phase handshaking protocol. In addition, for a set of concurrent processes, reshuffling of actions should not introduce a deadlock.

EXAMPLE 2.7. *In the buffer of example* 2.1, *there are two indistinguishable states: the initial state, and the state where* $ro\uparrow$ *takes place. If the first three actions on port* R *are reshuffled as follows:*

$$*[[li]; lo\uparrow; ro\uparrow; [\neg li]; [ri]; lo\downarrow; ro\downarrow; [\neg ri]],$$

then each state in the handshaking expansion is unique.

Reshuffling may drastically change the performance of the circuit [11, 13].

7. The Production Rule Set

After the introduction of state variables, and the reshuffling of actions, each state in the handshaking expansion is unique. It is now rather simple to derive a production rule set. Let x be a variable in the handshaking expansion, and consider a transition $x\uparrow$. There is a boolean expression B on the variables in the handshaking expansion, such that B only holds in a state where $x\uparrow$ occurs. The production rule

$$B \rightarrow x\uparrow$$

can then be added to the production rule set.

The next problem is how to derive condition B. If transition $x\uparrow$ is preceded in the handshaking expansion by a wait action, $[z]$ say, then $x\uparrow$ can only fire after z has been observed. Hence z has to be included as a term in B. Term z is said to be included in the production rule by *syntactic derivation.*

EXAMPLE 2.8. *For the buffer of example* 2.2*, with the state variable u of example* 2.6*, the production rules from the syntactic derivation are:*

$$\begin{array}{rcl} li & \rightarrow & ro\uparrow \\ ri & \rightarrow & u\uparrow \\ u & \rightarrow & ro\downarrow \\ \neg ri & \rightarrow & lo\uparrow \\ \neg li & \rightarrow & u\downarrow \\ \neg u & \rightarrow & lo\downarrow\,. \end{array}$$

Syntactic derivation usually is not sufficient for a correct production rule set. For instance, in the above example the production rule for $lo\uparrow$ may fire in the initial state (since all variables are **false** in the initial state). It is necessary to *strengthen* the guards of some production rules to prevent firings that are not in the specification.

EXAMPLE 2.9. *Continuing with the same buffer, the production rules for* $ro\uparrow$ *and for* $lo\uparrow$ *have to be strengthened, as follows:*

$$\begin{array}{rcccl} li & \wedge & \neg u & \rightarrow & ro\uparrow \\ \neg ri & \wedge & u & \rightarrow & lo\uparrow\,. \end{array}$$

After strengthening some guards, the resulting production rule set yields a circuit that is equivalent to the handshaking expansion. A final step that can be done is known as *symmetrization.* An example of symmetrization is to weaken one of a pair of production rules, so that the resulting pair is the specification of a combinational gate, rather than a state-holding element.

EXAMPLE 2.10. *To finish the buffer of example* 2.6*, weaken the production rule for* $ro\downarrow$ *with* $\neg li$*. This is allowed since it does not cause the production rule to fire*

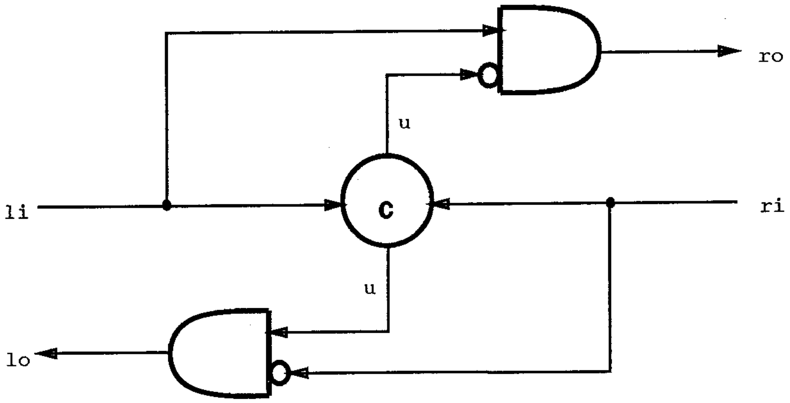

FIGURE 2.6. Circuit $C2$, the D-element buffer process

in a state where it could not fire before. The resulting pair of production rules is:

$$\begin{cases} li \wedge \neg u \rightarrow ro\uparrow \\ \neg li \vee u \rightarrow ro\downarrow . \end{cases}$$

This is an AND gate with one inverted input. Weakening the production rule for $lo\downarrow$ also changes the gate with output lo from a state-holding element to an AND gate with an inverted input:

$$\begin{cases} \neg ri \wedge u \rightarrow lo\uparrow \\ ri \vee \neg u \rightarrow lo\downarrow . \end{cases}$$

Finally, the production rules for u are the specification of a flip-flop. It is possible to strengthen both production rules so that the resulting pair is the specification of a C-element:

$$\begin{cases} li \wedge ri \rightarrow u\uparrow \\ \neg li \wedge \neg ri \rightarrow u\downarrow . \end{cases}$$

The circuit for this buffer is known as a D-element [12, 45]. See figure 2.6.

8. The Role of the Environment

I now investigate the role of the environment of a circuit in the execution of the handshaking expansion. For the D-element, the initial action of the circuit, after it is set to its initial state, is to wait for li to hold. The environment can hold the circuit in this state for an indefinite time by holding li **false**. By contrast, after the environment has set li to **true**, transition $ro\uparrow$ will follow eventually. The environment cannot delay $ro\uparrow$ at that point. In the former case, the circuit is in a *controllable* state, whereas in the latter case it is in a *transient* state.

DEFINITION 2.6 (CONTROLLABLE AND TRANSIENT STATES). *Let C be a circuit implementing a handshaking expansion for a single sequential process. A state in the handshaking expansion in which no production rule can fire, and the environment does not change any primary inputs, is a* controllable *state. Any other state is a* transient *state.*

DEFINITION 2.7 (CONTROLLABLE AND TRANSIENT CONDITIONS). *Let C be a circuit implementing a single sequential process, and B a boolean expression on variables of C. If there is a controllable state for which B holds, then B is a* controllable *condition. If B holds only in transient states, it is a* transient *condition.*

The disjunction of two controllable conditions is a controllable condition, but the conjunction of two controllable conditions is not necessarily controllable. The negation of a transient condition is controllable, if there is at least one controllable state in the handshaking expansion.

For the D-element, $\neg ro$ holds in the initial state, and is therefore a controllable condition. Also, li and $\neg u$ are controllable conditions, as is $li \wedge \neg u$, but $\neg ro \wedge li \wedge \neg u$ is a transient condition. The reason is that there is a production rule $li \wedge \neg u \rightarrow ro\uparrow$ in the circuit.

9. Acknowledgments and Isochronic Forks

In a delay-insensitive circuit there is a strict sequencing of actions. An action can only take place after the previous action has been completed; it *acknowledges* the completion of the previous action. After the environment changes the value of some primary inputs, there is a number of transitions in the circuit and some transitions of primary outputs. The transitions of the primary outputs acknowledge the transitions of the primary inputs. Once the environment observes the changes of the primary outputs, the necessary changes of the internal variables are guaranteed to have taken place. Consequently, there is no need to make the assumption that the circuit operates in Fundamental Mode.

DEFINITION 2.8 (ACKNOWLEDGEMENT). *Let C be a circuit, and consider a gate with input s and output t. If there is a transition $t\uparrow$, following a transition $s\uparrow$ ($s\downarrow$), in the handshaking expansion such that this transition only occurs when s ($\neg s$) holds, then $t\uparrow$* acknowledges *transition $s\uparrow$ ($s\downarrow$). Likewise for transition $t\downarrow$.*

It follows that if s is included in the production rule for $t\uparrow$ by syntactic derivation or strengthening, then there is a transition $t\uparrow$ in the handshaking expansion that acknowledges transition $s\uparrow$.

EXAMPLE 2.11. *Consider the following production rules for the D-element:*

$$\left\{\begin{array}{ccccc} li & \wedge & \neg u & \rightarrow & ro\uparrow \\ \neg li & \vee & u & \rightarrow & ro\downarrow . \end{array}\right.$$

Since $[\neg ri]$ precedes transition $ro\uparrow$ in the handshaking expansion, $ro\uparrow$ acknowledges $ri\downarrow$. Condition $\neg u$ was added to the guard for $ro\uparrow$ by strengthening, therefore $ro\uparrow$ also acknowledges transition $u\downarrow$. Transition $ro\downarrow$ acknowledges transition $u\uparrow$ (syntactic derivation), but not transition $li\downarrow$, since $\neg li$ does not hold when $ro\downarrow$ fires.

Each transition of a variable (corresponding to the output of a gate) in a delay-insensitive circuit, that is not a primary output, is acknowledged with another transition. A transition of a variable is not necessarily acknowledged by a transition of the output of each gate for which the variable is an input.

All delays are assumed to be arbitrary and unbounded. In case a transition of an input of a gate is not acknowledged with a transition of the output of the gate, the circuit may malfunction [48]. For instance, if a transition $li\downarrow$ is propagating much slower to the gate with output ro than to the C-element, then a transition $ro\uparrow$ may occur before transition $li\downarrow$ has been observed by the gate with output ro. For the circuit to operate correctly, the propagation delay of transition $li\downarrow$ to the gate with output ro must be shorter than the propagation delay of $li\downarrow$ to the C-element, plus the propagation delay of a down-transition of the C-element, plus the propagation delay of transition $u\downarrow$ to the gate with output ro.

Because of this delay assumption, the fork of variable li is known as an *isochronic fork*. Circuits with isochronic forks are sometimes referred to as quasi-delay-insensitive circuits [48].

DEFINITION 2.9 (ISOCHRONIC FORK). *Let C be a circuit with a variable s that forks to several gates, including a gate with output t. If there is a transition $s\uparrow$ or $s\downarrow$ in the handshaking expansion that is not acknowledged with a transition of t, then the fork of s is an* isochronic fork.

For all isochronic forks a delay assumption has to be made to insure correctness of the circuit. For some, the delay assumption is "one-sided". For instance, whereas there is an upper bound on the delay of transition $li\downarrow$ to the gate with output ro, no such delay assumption is necessary for the propagation delay of any transition of li to the C-element. I therefore refine the concept of an isochronic fork, and call the former branch of the fork an *isochronic branch.*

DEFINITION 2.10 (ISOCHRONIC BRANCH). *Let C be a circuit with a variable s that forks to several gates, including a gate with output t. If there is a transition*

$s\uparrow$ or $s\downarrow$ in the handshaking expansion that is not acknowledged with a transition of t, then the branch of s to the gate with output t is an isochronic branch*.*

In addition to the branch of *li* to the gate with output *ro*, the D-element also has an isochronic branch for variable *ri* as input to the gate with output *lo*.

CHAPTER 3

A Method to Test Delay-Insensitive Circuits

But there was another source of errors in PETER: unreliable soldering points which are very difficult to detect because 99.9 percent of the time they function properly. In the course of time an efficient method was developed to find them. A fist or hammer was used and by banging on the racks numerous weak soldering points were discovered.

— N. C. de Troye, *From Arra to Apple*

1. Introduction

In this chapter I discuss a method to test delay-insensitive circuits. First I discuss problems associated with testing. In particular the assumption that all delays are finite but unbounded means that no fault is testable. I therefore have to make some assumption on propagation delays.

I show that a stuck-at fault in a delay-insensitive circuit may either cause a production rule to fire when it should not, or not to fire when it should. In the former case the fault is *stimulating*, in the latter it is *inhibiting.* Some faults are both stimulating and inhibiting. For each inhibiting fault there is a state in the handshaking expansion where the fault causes a transition not to take place when it should; for each stimulating fault there is a state in the handshaking expansion where the fault causes a transition to occur when it should not. The testing problem is to bring the faulty circuit in such a state, and to propagate the result to a primary output, which can be observed by the environment. As a larger example I analyze the faults in a one-bit queue element.

Let a and b be variables in a circuit, where a is an input to the gate with output b. There is a difference between a fault on variable a, that is a primary input or the output of a gate, and the input a of the gate with output b. Since a production rule set typically does not include production rules for forks, this difference is implicit in a production rule set. When it is not clear from the context which variable is being discussed, I denote the input a to the gate with output b as $a[b]$.

In a delay-insensitive circuit there are sequences of transitions, where each transition is acknowledged with the next one. In theorems about faults I frequently have to refer to a "next" transition. It does not matter whether such an acknowledging transition is an up- or a down-transition. For the sake of clarity I shall always assume that it is an up-transition.

Finally, most inhibiting faults are detected when a sequence of acknowledgements does not take place. The environment then detects the fault, as there is a missing transition of a primary output. I have to assume that there is always a next transition of a primary output in each state of the handshaking expansion. In other words, either a program is terminating, and ends with a transition of a primary output, or the program is non-terminating and it is always possible to have a transition of a primary output later.

2. Problems with the Delay Model

The purpose of testing chips is to separate the fully functional ones from the faulty ones. The goal is twofold: first, to accept only those chips that have no defects, and second, to reject as few as possible (ideally: none) of the chips without defects. The latter objective is typically the easiest to meet; any non-defective chip ought to pass any test. It is, however, impossible to guarantee that only non-defective chips be accepted. Even if we can prove that all faults are detectable by some test, such a proof is given with the assumption of some fault model. There can always be faults that cannot be described in the fault model.

DEFINITION 3.1 (TEST). *A* test *for circuit C is a finite sequence of actions by the environment of C. An action is either setting the value of a primary input of C, or observing the value of a primary output of C.*

Executing a test consists of setting the primary inputs of the circuit, and observing the primary outputs. After the correct primary output transitions have been observed, one has to wait for some time before changing the primary inputs again; it is possible that there is another transition of a primary output pending, in a faulty circuit.

I use a straightforward definition of testable faults: a fault is testable if there is a sequence of inputs to the circuit, such that the fault is *guaranteed* to be detected.

DEFINITION 3.2 (DETECTED FAULT). *Let T be a test for circuit C. Let circuit C' be the same circuit, but with a single stuck-at fault. If, during execution of test T, there is a point at which a primary output of C' has a value that cannot occur at the same point in the test for circuit C, then the fault is* detected.

DEFINITION 3.3 (DETECTABLE FAULT). *A fault in circuit C is* detectable *with test T if the fault is guaranteed to be detected, regardless of the propagation delays in the circuit.*

DEFINITION 3.4 (TESTABLE FAULT). *A fault in circuit C is* testable *if there is a test that detects the fault.*

For a discussion of these definitions, see appendix B. Unfortunately, with the model for asynchronous circuits, the definition of a testable fault implies that *no* fault is guaranteed to be detected. Recall that for self-timed circuits any gate delay and wire propagation delay is arbitrary and unbounded.

We test a circuit by setting the value of its inputs, and observing the values of its outputs. A circuit is faulty when the value of its outputs is not the same as the value of the outputs for a correct circuit. This can happen when either for the faulty circuit an output changes value when the correct circuit does not, or when for the correct circuit an output changes value when the faulty circuit does not. When there is an unexpected change of an output, then the circuit, obviously, is a faulty one. But if a change of output value does not take place while a change is expected, can we conclude that the circuit is faulty? Since wire delays are assumed to be arbitrary and unbounded, it is possible, at any time, that there will be a change of the output value some time later. I show in this chapter that there is a large number of faults in any circuit that lead to the circuit halting in some state. In an actual test, there is no point in time, at which we can conclude that a circuit with such a fault is to be rejected.

Likewise, if, in some state, a correct circuit does not change its outputs, then the observation that a circuit under test, in the same state, does not change its value either, does not imply that the circuit behaves correctly. It may be the case that there is a change of an output pending, that is not yet observed because of the unbounded wire delay.

An even worse scenario is the following: consider a circuit, delete all internal circuitry, and connect each output pad to a ring oscillator. Assume that each ring oscillator can be reset so that the outputs in the initial state for both the original circuit and the one with the ring oscillators are the same. Then no test will guarantee that the ring oscillators can be distinguished from the original circuit.

With the delay assumptions made here, there is no test that will accept only correct circuits, and there is no test that will reject only defective circuits, since the delay assumptions require that a circuit be observed for an infinite time before any conclusions can be made. From a theoretical point of view, the testing problem cannot be solved.

The problems described above are, however, merely theoretical problems. In any actual test, the difference between any circuit and a ring oscillator will be immediately obvious from the behavior of the circuits. With modern technologies, switching times are on the order of nanoseconds, or even picoseconds, and variations in switching times are relatively minor. A ring oscillator will change the value of the output frequently, so if we do not change the value of the inputs, the original circuit will not change its outputs, whereas the ring oscillator will. Also, if a circuit under test does not switch within, say, a microsecond after a correct circuit switches, then we may safely assume that the circuit is faulty.

There are two types of faults. Some faults cause the circuit to halt entirely, and some faults result in an unspecified output change. I consider such occurrences detectable. I could extend the definition of detectability: in the presence of some faults an output of a circuit will switch faster than expected. However, it may be hard to tell whether such an occurrence takes place because of a fault or because the chip happens to be relatively fast.

To the above observations I add an important caveat, concerning testing of arbitration devices. As is well-known, any arbitration device has a metastable state, when arbitrating between two alternatives. The time that the arbiter takes to leave such a state is arbitrary and unbounded [16]. Therefore it is not enough to wait a microsecond or so for outputs to change. Typically it is possible to test an arbiter by having only one request valid at any time. The arbiter is then considered equivalent to two wires, where the wires are not both high at the same time. If the actual arbitration is tested, then the methods described here do not apply. The simplest way to test an arbiter is to add test circuitry with which it can be functionally separated from the remainder of the circuit.

3. A Classification of Stuck-At Faults

For a given production rule a fault may reduce the number of states in which the production rule may fire, or it may increase the number of such states. In the former case, the fault *inhibits* the production rule, and in the latter case it *stimulates* the production rule.

I assume that each circuit is non-redundant. A variable is redundant if each occurence of the variable in the production rules can be replaced with **true** or **false**. A gate is redundant if its output is redundant. A circuit is redundant if it has a redundant gate, or a gate with a redundant input. See Appendix B. Note that a production rule in a non-redundant circuit may still have a redundant literal, or a redundant term. (A literal is a boolean variable, or its negation.) In particular, weakening production rules introduces redundant literals.

DEFINITION 3.5 (INHIBITING FAULT). *Let C be a circuit that has a gate with input s and output t. Consider a production rule for t. If it contains a non-redundant term with literal s ($\neg s$), then fault s stuck-at-0 (s stuck-at-1) is* inhibiting *the production rule.*

DEFINITION 3.6 (STIMULATING FAULT). *If a production rule for t contains a non-redundant term with literal s ($\neg s$), then fault s stuck-at-1 (s stuck-at-0) is* stimulating *the production rule.*

An inhibiting fault reduces the set of states in which a production rule may fire. Such a fault may be tested by bringing the circuit in a state where the correct production rule fires, and the production rule with the fault does not. A stimulating fault augments the set of states in which a production rule may fire. Such a fault may be tested by bringing the circuit in a state where the correct production rule does not fire, but the production rule with the fault does. The following theorems state that, both for inhibiting and stimulating faults, there is always such a state in the handshaking expansion.

THEOREM 3.1. *Let C be a non-redundant circuit that has a gate with input s and output t. Let s stuck-at-0 be an inhibiting fault. Then there is a state in the handshaking expansion where the fault causes a production rule for t not to fire when it should. Similarly for fault s stuck-at-1.*

Proof: The general form of the production rules for t is:

$$\begin{cases} s \wedge B_0 & \vee \quad \neg s \wedge B_1 \quad \rightarrow \quad t\uparrow \\ s \wedge C_0 & \vee \quad \neg s \wedge C_1 \quad \rightarrow \quad t\downarrow, \end{cases}$$

where each B_i and C_i is a boolean expression not containing s. Consider fault s stuck-at-0. With this fault, the guard for $t\uparrow$ reduces to B_1. The fault inhibits a transition $t\uparrow$ if there is a state in the handshaking expansion where

$$\neg t \wedge (s \wedge B_0 \vee \neg s \wedge B_1) \wedge \neg B_1,$$

that is

$$\neg t \wedge s \wedge B_0 \wedge \neg B_1.$$

If s is not redundant in the guard for $t\uparrow$, then there is a state in the handshaking expansion where $t\uparrow$ fires, so that $t\uparrow$ is an acknowledgement for transition $s\uparrow$, and $s \wedge B_0$ holds when $t\uparrow$ fires. If B_1 holds in this state, then $t\uparrow$ may fire before the gate observes transition $s\uparrow$, that is, $t\uparrow$ does not acknowledge $s\uparrow$. Hence $\neg B_1$ must hold. Therefore there is a state in the handshaking expansion where a transition $t\uparrow$ is inhibited, if s is not redundant in the up-guard. Similarly if s not redundant in the down-guard. □

THEOREM 3.2. *Let C be a non-redundant circuit that has a gate with input s and output t. Let s stuck-at-0 be a stimulating fault. Then there is a state in the handshaking expansion where the fault may cause a production rule for t to fire when it should not. Similarly for fault s stuck-at-1.*

Proof: The general form of the production rules for t is:

$$\begin{cases} s \wedge B_0 & \vee & \neg s \wedge B_1 & \rightarrow & t\uparrow \\ s \wedge C_0 & \vee & \neg s \wedge C_1 & \rightarrow & t\downarrow, \end{cases}$$

where each B_i and C_i is a boolean expression not containing s. Consider fault s stuck-at-0. With this fault, the guard for $t\uparrow$ reduces to B_1. The fault may cause a premature transition of $t\uparrow$ if there is a state in the handshaking expansion where

$$\neg t \wedge B_1 \wedge \neg(s \wedge B_0 \vee \neg s \wedge B_1),$$

that is

$$\neg t \wedge s \wedge \neg B_0 \wedge B_1.$$

If $\neg s$ is not redundant in the guard for $t\uparrow$, then there is a state in the handshaking expansion where $t\uparrow$ cannot fire, and where $s \wedge \neg B_0 \wedge B_1$ holds; in this state the fault may cause a premature firing. Similarly for $\neg s$ in the down-guard. □

The next theorem states that any fault on an output of a gate is both stimulating and inhibiting.

THEOREM 3.3. *Let C be a delay-insensitive circuit. Let s be a variable in C that is either a primary input, or the output of a gate, but not a primary output. Then faults s stuck-at-0 and s stuck-at-1 are both inhibiting and stimulating.*

Proof: Consider, without loss of generality, the fault s stuck-at-0. Since s is not a redundant variable in C, there is a transition $s\uparrow$ in the handshaking expansion (or $[s]$ if s is a primary input). Let $u\uparrow$ be a transition directly following this transition $s\uparrow$. Then s must be an input to the gate with output u, and transition $u\uparrow$ acknowledges transition $s\uparrow$. The production rule for $u\uparrow$ is of the form

$$s \wedge B_0 \vee B_1 \rightarrow u\uparrow,$$

where $s \wedge B_0$ is not redundant. With a fault s stuck-at-0 the same production rule reduces to

$$B_1 \rightarrow u\uparrow .$$

Then $u\uparrow$ will not fire if $s \wedge B_0$ holds; the fault is inhibiting.

Now consider a transition $s\downarrow$ in the handshaking expansion. Let the transition immediately following it be $v\uparrow$. Then $v\uparrow$ acknowledges $s\downarrow$, and s is an input to the gate with output v. The production rule for $v\uparrow$ is of the form

$$\neg s \wedge C_0 \vee C_1 \rightarrow v\uparrow,$$

where $\neg s \wedge C_0$ is not redundant. With a fault s stuck-at-0 the same production rule reduces to

$$C_0 \vee C_1 \rightarrow v\uparrow .$$

This production rule may fire *before* transition $s\downarrow$; the fault is stimulating. □

A fault that is only inhibiting, or only stimulating, is necessarily a fault on an input of a gate. For a primary output, the terms inhibiting and stimulating are not defined. It should be clear, that for any fault on a non-redundant primary output, there is a state in the handshaking expansion where the primary output has the wrong value.

4. Faults Causing the Circuit to Halt During Test

Most stuck-at faults in a delay-insensitive circuit will cause the circuit to halt during some test. I investigate under what circumstances a circuit halts.

THEOREM 3.4. *Let C be a circuit implementing program P. Let s be a primary input or the output of a gate of C. Let circuit C' be identical to C, except for a single stuck-at fault on s. If there is a test such that each variable has the same value in C and in C' after execution of the test, then the fault is testable.*

Proof: Assume, without loss of generality, that the fault is s stuck-at-1. Execute a test such that C and C' are in the same state. This implies that s is **true** in circuit C. Continue to execute the handshaking expansion from this point until the first transition $s\downarrow$. As long as s is **true**, both circuits are identical. Because of the fault, C' will not have any transition $s\downarrow$.

Continue to execute the handshaking expansion until the first transition of a primary output after $s\downarrow$, say $u\uparrow$. This part of the handshaking expansion is of the form

$$\ldots ; s\downarrow; [\neg s \wedge B]; \ldots ; u\uparrow; \ldots$$

Transition $u\uparrow$ is an acknowledgement of transition $s\downarrow$. In circuit C such a transition $u\uparrow$ will eventually take place. For circuit C', however, there is no transition $s\downarrow$ to be acknowledged. Therefore there will not be a transition $u\uparrow$ in circuit C'. The fault is testable. □

From this theorem a number of simple statements follow.

THEOREM 3.5. *A fault on a primary output is testable.*

Proof: Let so be a primary output of circuit C. Let C' be the same circuit, but for a fault so stuck-at-0. Then the initial state for C and C' is the same. By theorem 3.4, the fault is testable.

Let C'' be identical to C, except for a fault so stuck-at-1. The fault is detectable in the initial state, since so will be observed by the environment to be **true**. □

THEOREM 3.6. *Let s be either a primary input, or the output of a gate in circuit C. Then fault s stuck-at-0 is testable.*

Proof: Let C' be the same circuit as C, except for a fault s stuck-at-0. Then circuits C and C' will reset to the same initial state. Variable s is not redundant, therefore by theorem 3.4 the fault is testable. □

Because of this theorem, the only faults on outputs of gates that need more analysis are stuck-at-1 faults.

THEOREM 3.7. *Let si be a primary input of circuit C, where si is an input of a passive channel or a lazy-active channel. Then fault si stuck-at-1 is testable.*

Proof: The initial state of si is **false**. If si is an input of a passive channel this input may have a transition to **true** at any time. Therefore si **true** must also be allowed in the initial state. By theorem 3.4 the fault is testable. Likewise, if si is an input of a lazy-active channel, then si may initially be **true**. □

Theorem 3.4 is a testability result for faults on primary inputs and on outputs of gates. Upon closer inspection of the proof, it also extends to some faults on inputs of gates. In the proof, a fault s stuck-at-1 is shown to be testable since it causes a transition $s\downarrow$ not to take place in circuit C'. As a consequence, there is a series of acknowledgements that do not take place, resulting in a primary output, u, not having a transition when it should.

If s is not a primary output, then the transition $s\downarrow$ is acknowledged with another transition, say $v\uparrow$ (variable v is not necessarily the same variable as u). This means that s is an input to the gate with output v. Now consider circuit C'', which is the same as the correct circuit C, except the input s to the gate with output v is stuck-at-1. Then the test that detected fault s stuck-at-1 in circuit C' also detects this fault in circuit C'', since the fault prevents an acknowledgement (namely, $v\uparrow$) of the transition $s\downarrow$ to occur. This leads to the following

COROLLARY 3.8. *Let s be a primary input, or the output of a gate, in circuit C. Let T be a test for the fault s stuck-at-0 (stuck-at-1) that results in a transition* *of a primary output not taking place in the presence of the fault. Let s be input to a gate with output v, such that the last transition $s\uparrow$ $(s\downarrow)$ in T is acknowledged with a transition $v\uparrow$. Then test T also detects a fault if input s to the gate with output v is stuck-at-0 (stuck-at-1).*

Theorem 3.4 proves the testability of an inhibiting fault, by constructing a test such that the fault causes the circuit to halt. There are a few other theorems on inhibiting faults on inputs of gates.

THEOREM 3.9. *Let s be an input to a gate with output v. If a stuck-at fault on the input s causes the guard for $v\uparrow$ to be* **false**, *then this fault is testable.*

Proof: Without loss of generality, consider fault s stuck-at-1. The production rule for $v\uparrow$ is of the form

$$\neg s \wedge B \rightarrow v\uparrow,$$

where B is a boolean expression. With fault s stuck-at-1 this reduces to

$$\textbf{false} \rightarrow v\uparrow .$$

Therefore variable v is invariantly **false** during any test. Fault v stuck-at-0 is testable, therefore input s stuck-at-1 is also testable. □

THEOREM 3.10. *Let s be an input to a gate with output v, where v is a primary output. If a stuck-at fault on the input s causes the guard for $v\downarrow$ to be* **false**, *then this fault is testable.*

Proof: If v is **true** initially, as a result of the fault, then by theorem 3.5 the fault is detectable. Otherwise apply theorem 3.4. □

This theorem can be extended to any fault that causes a sequence of variables that are stuck at some value, resulting in a primary output being stuck-at-1.

The above theorems can be used to prove the testability of many inhibiting faults in a circuit. For the remaining inhibiting faults it is necessary to consider the handshaking expansion. In the next section I derive conditions under which these inhibiting faults cause the circuit to halt during a test.

5. A Classification of Inhibiting Faults

The following is a classification of faults that are inhibiting. A fault that is only inhibiting and not stimulating is always testable, provided that the circuit resets to a well-defined state. For such a fault, there is a test that will cause the circuit to halt. If a fault is both inhibiting and stimulating, then it may be detected either if the circuit halts, or if there is a premature firing of a primary output. I derive when such a fault causes the circuit to halt.

The formulae for the testability of an inhibiting fault depend on whether the fault is stuck-at-0 or stuck-at-1, and on whether the corresponding literal is in the guard for the up-transition or the down-transition of a gate, or both. In the first subsection I derive these formulae for one of these cases in some detail; the other subsections can be omitted on first reading.

5.1. Fault stuck-at-0 Inhibits Down-transition. Consider a gate in circuit C with input u and output v, and production rules:

$$\left\{ \begin{array}{rcccl} \neg u \wedge B_0 & \vee & B_1 & \rightarrow & v\uparrow \\ u \wedge C_0 & \vee & C_1 & \rightarrow & v\downarrow, \end{array} \right.$$

where B_0, B_1, C_0, and C_1 are boolean conditions that include neither u nor $\neg u$.

Let circuit C' be identical to C, except for a stuck-at-0 fault on this input u. The production rules for v in C' are:

$$\left\{ \begin{array}{rcccl} B_0 & \vee & B_1 & \rightarrow & v\uparrow \\ & & C_1 & \rightarrow & v\downarrow\,. \end{array} \right.$$

The guard for $v\uparrow$ has weakened, whereas the guard for $v\downarrow$ has strengthened. If the circuit is to halt during a test, then $v\uparrow$ should not fire prematurely, and the circuit should be brought in a state where $v\downarrow$ is inhibited.

If u is redundant in the guard for $v\downarrow$, then the fault is not inhibiting, and only stimulating, which is dealt with in section 6. If $\neg u$ is redundant in the guard for $v\uparrow$, then the fault is only inhibiting, and will always cause the circuit to halt, provided that v resets to the correct value.

The remaining case is when neither u nor $\neg u$ is redundant. Consider a state in the handshaking expansion where $v\downarrow$ fires as a result of condition $u \wedge C_0 \wedge \neg C_1$. If circuit C' can be brought into such a state from the initial state, without a premature firing of $v\uparrow$, then the fault is testable. With fault u stuck-at-0, condition $u \wedge C_0 \wedge \neg C_1$ (when the guard for $v\downarrow$ is **true**) reduces to $\neg C_1$ (when the guard for $v\downarrow$ is **false**). As a result, $v\downarrow$ will not fire, and there will not be an acknowledgement for $v\downarrow$, so that there is a primary output that fails to have a transition.

There is a possible premature firing of $v\uparrow$ because of the fault when

$$\neg v \wedge u \wedge B_0 \wedge \neg B_1.$$

Even if this condition is transient, there could be a premature firing of $v\uparrow$.

In conclusion, the stuck-at-0 fault on input u is testable and causes the circuit to halt if there is a test such that

$$v \vee \neg u \vee \neg B_0 \vee B_1$$

holds from the initial state until a state where

$$v \wedge u \wedge C_0 \wedge \neg C_1.$$

Note that the first condition holds in the initial state, since $\neg u$ holds in the initial state. Therefore the circuit will reset correctly even in the presence of this fault, except for variable u.

5.2. Fault stuck-at-1 Inhibits Up-transition. Consider the same circuit C, where C' has input u stuck-at-1. In C the production rules for u are:

$$\begin{cases} \neg u \wedge B_0 & \vee \quad B_1 \quad \rightarrow \quad v\uparrow \\ u \wedge C_0 & \vee \quad C_1 \quad \rightarrow \quad v\downarrow, \end{cases}$$

In C' these production rules are:

$$\begin{cases} & \quad B_1 \quad \rightarrow \quad v\uparrow \\ C_0 & \vee \quad C_1 \quad \rightarrow \quad v\downarrow . \end{cases}$$

Again assume that neither u nor $\neg u$ is redundant in the production rules for v. The fault may cause a premature firing of $v\downarrow$ when

$$v \wedge \neg u \wedge C_0 \wedge \neg C_1$$

holds.

The stuck-at-1 fault on input u is testable and causes the circuit to halt if

$$\neg v \vee u \vee \neg C_0 \vee C_1$$

holds from the initial state until a state where

$$\neg v \wedge \neg u \wedge B_0 \wedge \neg B_1.$$

Note that the first condition holds in the initial state, since $\neg v$ holds in the initial state. Therefore the circuit will reset correctly even in the presence of this fault.

5.3. Fault stuck-at-0 Inhibits Up-transition. A second kind of gate is one where u occurs in the guard for $v\uparrow$, and $\neg u$ in the guard for $v\downarrow$, and neither literal is redundant. The production rules for variable v in circuit C are:

$$\begin{cases} u \wedge B_0 & \vee \quad B_1 \quad \rightarrow \quad v\uparrow \\ \neg u \wedge C_0 & \vee \quad C_1 \quad \rightarrow \quad v\downarrow . \end{cases}$$

Let circuit C' be identical to C, except for a stuck-at-0 fault on input u. The production rules for v in C' are:

$$\begin{cases} \qquad\qquad B_1 \rightarrow v\uparrow \\ C_0 \;\vee\; C_1 \rightarrow v\downarrow . \end{cases}$$

Circuit C' will halt if there is a transition $v\uparrow$ that fails to occur, before there is a premature firing of $v\downarrow$.

The conditions to cause the circuit to halt are derived in the same way as before. Circuit C' will halt if

$$\neg v \vee \neg u \vee \neg C_0 \vee C_1$$

holds from the initial state until a state where

$$\neg v \wedge u \wedge B_0 \wedge \neg B_1.$$

Note that the first condition holds in the initial state, since both $\neg v$ and $\neg u$ hold in the initial state. Therefore the circuit will reset correctly even in the presence of this fault.

5.4. Fault stuck-at-1 Inhibits Down-transition. This is the same gate as in the previous case. In circuit C' input u is now stuck-at-1. The production rules for v in C are:

$$\begin{cases} u \wedge B_0 \;\vee\; B_1 \rightarrow v\uparrow \\ \neg u \wedge C_0 \;\vee\; C_1 \rightarrow v\downarrow . \end{cases}$$

For C' the production rules for v are:

$$\begin{cases} B_0 \;\vee\; B_1 \rightarrow v\uparrow \\ \qquad\quad C_1 \rightarrow v\downarrow . \end{cases}$$

In order for the circuit to halt, there must be an inhibited transition $v\downarrow$ before a premature firing of $v\uparrow$. Circuit C' will halt if

$$v \vee u \vee \neg B_0 \vee B_1$$

holds from the initial state until a state where

$$v \wedge \neg u \wedge C_0 \wedge \neg C_1.$$

Since $\neg u \wedge \neg v$ holds in the initial state of the handshaking expansion, $\neg B_0 \vee B_1$ must hold in the initial state to avoid a premature firing.

5.5. Fault stuck-at-0 Inhibits and Stimulates Both Transitions. Finally I examine the general case of production rules for a gate with input u and output v. This is when u and $\neg u$ are included in both the guard for $v\uparrow$ and the guard for $v\downarrow$. Therefore either production rule may fire prematurely and may cause the circuit to halt.

The previous gates are special cases of this gate. I have analyzed them separately, since the conditions that cause halting are less complicated, and since those cases generally comprise the vast majority of gates.

For an arbitrary gate with input u and output v the production rules are:

$$\begin{cases} u \wedge B_0 & \vee \quad \neg u \wedge B_1 \quad \vee \quad B_2 \quad \rightarrow \quad v\uparrow \\ u \wedge C_0 & \vee \quad \neg u \wedge C_1 \quad \vee \quad C_2 \quad \rightarrow \quad v\downarrow, \end{cases}$$

where $B_0, B_1, B_2, C_0, C_1,$ and C_2 are boolean expressions that contain neither u nor $\neg u$ as literals. With input u stuck-at-0, the production rules reduce to

$$\begin{cases} B_1 \quad \vee \quad B_2 \quad \rightarrow \quad v\uparrow \\ C_1 \quad \vee \quad C_2 \quad \rightarrow \quad v\downarrow. \end{cases}$$

Transition $v\uparrow$ will not take place in the presence of the fault if there is a transition $v\uparrow$ in a state in the handshaking expansion where $u \wedge B_0 \wedge \neg B_1 \wedge \neg B_2$ holds. A transition $v\downarrow$ does not take place if there is a state where a transition $v\downarrow$ occurs because $u \wedge C_0 \wedge \neg C_1 \wedge \neg C_2$ holds. Either production rule may also fire prematurely. For $v\uparrow$ this can occur when $u \wedge B_1 \wedge \neg B_0 \wedge \neg B_2$ holds, and for $v\downarrow$ this can occur when $u \wedge C_1 \wedge \neg C_0 \wedge \neg C_2$ holds.

In a formula, the circuit will halt as a result of a stuck-at-0 fault on the input u of the gate with output v if

$$(v \vee \neg u \vee B_0 \vee \neg B_1 \vee B_2) \wedge (\neg v \vee \neg u \vee C_0 \vee \neg C_1 \vee C_2)$$

hold from the initial state until a state where

$$(\neg v \wedge u \wedge B_0 \wedge \neg B_2) \vee (v \wedge u \wedge C_0 \wedge \neg C_2).$$

5.6. Fault stuck-at-1 Inhibits and Stimulates Both Transitions. Consider the same general production rules:

$$\begin{cases} u \wedge B_0 & \vee & \neg u \wedge B_1 & \vee & B_2 & \rightarrow & v\uparrow \\ u \wedge C_0 & \vee & \neg u \wedge C_1 & \vee & C_2 & \rightarrow & v\downarrow . \end{cases}$$

Now let input u be stuck-at-1, then the production rules reduce to

$$\begin{cases} B_0 & \vee & B_2 & \rightarrow & v\uparrow \\ C_0 & \vee & C_2 & \rightarrow & v\downarrow . \end{cases}$$

As before, either production rule may cause the circuit to halt, or may cause a premature firing. In order for the fault to be detectable by an inhibited firing,

$$(v \vee u \vee \neg B_0 \vee B_1 \vee B_2) \wedge (\neg v \vee u \vee \neg C_0 \vee C_1 \vee C_2)$$

should hold from the initial state until a state where

$$(\neg v \wedge \neg u \wedge B_1 \wedge \neg B_2) \vee (v \wedge \neg u \wedge C_1 \wedge \neg C_2).$$

5.7. Primary Input or Output of Gate stuck-at-1. Let u be either a primary input, or the output of a gate, and consider the fault u stuck-at-1. If u is a primary output, then the fault is immediately detectable. Otherwise, u is input to one or more gates. If fault u stuck-at-1 causes the circuit to halt during a test, then none of the faults on the inputs should cause a premature firing, until there is an inhibited transition for one of the gates to which u is connected. The same formulae can be used as for the cases above. The conjunction of the conditions to prevent a premature firing has to hold, until one of the conditions for an inhibited firing holds.

6. Stimulating Faults

A fault that does not cause an inhibited firing (and a halting circuit) for any test must be tested by having the fault cause a production rule fire prematurely. The case of such a stimulating fault is more complicated than the case of an inhibiting fault. Suppose, for an inhibiting fault, that there is a test such that the fault causes an inhibited firing before any premature firing. Then subsequent firings are also inhibited, causing a primary output not to have a transition when one is expected. The difference between the correct and the faulty circuit is that there are fewer transitions in the faulty circuit.

In case of a premature firing, however, there may be *more* transitions in the faulty circuit, possibly introducing more concurrency. There are several problems associated with prematurely firing production rules. The first problem is one of stability of the production rule set of the faulty circuit. Since the premature firing is not specified in the handshaking expansion, it may be that the condition for the premature firing is transient. The premature transition may or may not occur, depending on the propagation delays in the circuit, hence such a fault may or may not be detected with a test. In that case the fault is not testable.

To find the state (or states) where the production rule may fire prematurely, derive a boolean condition, from the guard, for the premature firing to take place. Then scan the handshaking expansion, and try to match this condition with the value of variables in each state of the handshaking expansion. In each state the value of all internal variables and primary outputs are known; only the values of some primary inputs may be unknown. The primary inputs are controlled by the environment, hence such unknown values may be defined by the user. Once a state is found where the production rule fires prematurely, check whether the firing is guaranteed to take place, then check (in the handshaking expansion) whether there is a test sequence from the initial state to this state, without reaching a transient state where the premature firing can occur.

Suppose a production rule fires prematurely, and the condition for this premature firing to occur is not transient. If the variable firing prematurely is a primary output, then the fault is obviously detected. Otherwise, the variable (v, say) is an internal variable, and the premature firing has to be propagated to a primary output. There are three possibilities at this point.

- There is a sequence of transitions, from the premature transition of v to a premature transition of a primary output, and no condition for any of these premature firings is transient. Then the fault is testable.
- There is a sequence of transitions, from the premature transition of v to a premature transition of a primary output, and one of these premature transitions is unstable. Then the premature transition of v may or may not propagate to a primary output. The fault is sometimes detectable, but it is not testable.
- The premature transition of v does not cause a sequence of premature firings leading to the premature firing of a primary output. Then the fault is never detectable, and not testable.

Only in the first case is the fault testable. In all other cases, the fault can only be made testable with the addition of test circuitry. This is the subject of chapter 5.

The derivation of a condition for a premature firing of a production rule is quite similar to the derivations for inhibited production rules. I only derive the general case.

Let C be a delay-insensitive circuit that has a gate with input u and output v, and production rules:

$$\begin{cases} u \wedge B_0 & \vee & \neg u \wedge B_1 & \vee & B_2 & \to & v\uparrow \\ u \wedge C_0 & \vee & \neg u \wedge C_1 & \vee & C_2 & \to & v\downarrow, \end{cases}$$

where B_0, B_1, B_2, C_0, C_1, and C_2 are boolean expressions that contain neither u nor $\neg u$ as literals. With input u stuck-at-0, the production rules reduce to

$$\begin{cases} B_1 & \vee & B_2 & \to & v\uparrow \\ C_1 & \vee & C_2 & \to & v\downarrow. \end{cases}$$

Now $v\uparrow$ may fire prematurely, since the conjunction $\neg u \wedge B_1$ in the correct circuit is weakened to B_1 for the faulty circuit. The premature firing will take place if there is a controllable state in the handshaking expansion where

$$\neg v \wedge u \wedge B_1 \wedge \neg B_2.$$

This state must be reachable from the initial state without having an transient intermediate state where the same condition holds. This ensures that the premature firing is stable, and that there is no unstable premature firing previously.

Similarly, for $v\downarrow$ to fire prematurely there has to be a controllable state in the handshaking expansion where

$$v \wedge u \wedge C_1 \wedge \neg C_2.$$

In case input u is stuck-at-1, $v\uparrow$ fires prematurely if there is a controllable state in the handshaking expansion where

$$\neg v \wedge \neg u \wedge B_0 \wedge \neg B_2,$$

and $v\downarrow$ fires prematurely if there is a controllable state where

$$v \wedge \neg u \wedge C_0 \wedge \neg C_2.$$

In all these cases, there must not be a transient intermediate state where a production rule can fire prematurely, until the controllable state has been reached.

Once a production rule has fired prematurely, the result has to be propagated to a primary output. Hence, if v is not a primary output, there has to be a sequence of premature firings, resulting in the premature firing of a primary output. The condition for each subsequent premature firing to take place is the same as the condition derived above for the firing caused directly by the fault. There has to be a controllable state where the next production rule will fire prematurely, and there may not be a transient state between the premature firing of the previous production rule until this controllable state, and so forth until there is a stable premature firing of a primary output.

There is another way for a fault causing a prematurely firing production rule to be detectable. After one or more production rules have fired prematurely, the circuit is not in a correct state any more. It is then possible that a subsequent transition is inhibited. In other words, the sequence of premature transitions may cause another transition not to take place, so that the circuit halts. In such a case, the fault is obviously detected. The conditions for production rules to fire prematurely remain the same as above, as do the conditions for the inhibited production rules. For the inhibited production rules it is not necessary to check the conditions from the initial state (if the fault caused the circuit to halt *before* the premature firing, then it would already be detected), but only from the state where the premature firing takes place.

7. Fault Analysis of a One-bit Queue Element

I derive a test for all single stuck-at faults for a circuit implementing a one-bit queue element. The queue element has two one-bit channels, L and R. It repeatedly receives a binary value over channel L from its left neighbor, then sends the same value over channel R to its right neighbor. The CSP specification of such a queue element is

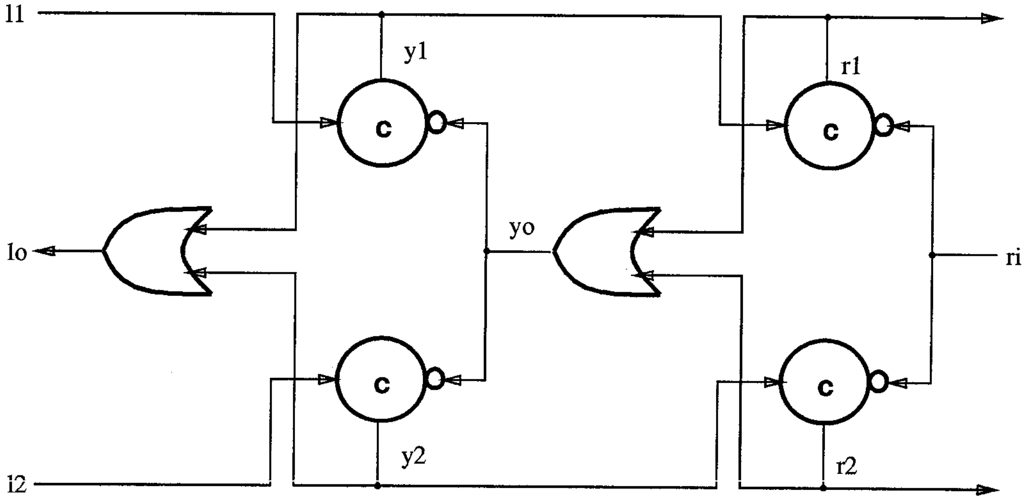

FIGURE 3.1. One-bit wide queue element

$$*[L?x; R!x].$$

In the handshaking expansion I reshuffle some actions, and I include internal variables $y1$, $y2$, and yo, yielding

```
*[[  l1 → y1↑; [y1]; lo↑; [¬ri]; r1↑; yo↑; [yo ∧ ¬l1];
          y1↓; [¬y1]; lo↓; [ri]; r1↓; yo↓; [¬yo]
   | l2 → y2↑; [y2]; lo↑; [¬ri]; r2↑; yo↑; [yo ∧ ¬l2];
          y2↓; [¬y2]; lo↓; [ri]; r2↓; yo↓; [¬yo]
  ]].
```

If the left neighbor sends a "1" value to the queue element, it raises signal $l1$, and if it sends a "0" it raises signal $l2$. (I use the suffix "2" to avoid confusion with the suffix for outputs, "o".) Similarly, the element sends a "1" to the right neighbor by raising $r1$, and a "0" by raising $r2$.

The production rule set is

$$\begin{cases} l1 \wedge \neg yo & \rightarrow & y1\uparrow \\ \neg l1 \wedge yo & \rightarrow & y1\downarrow \end{cases}$$

$$\begin{cases} l2 \wedge \neg yo & \rightarrow & y2\uparrow \\ \neg l2 \wedge yo & \rightarrow & y2\downarrow \end{cases}$$

$$\begin{cases} y1 \vee y2 & \rightarrow & lo\uparrow \\ \neg y1 \wedge \neg y2 & \rightarrow & lo\downarrow \end{cases}$$

$$\begin{cases} y1 \wedge \neg ri & \rightarrow & r1\uparrow \\ \neg y1 \wedge ri & \rightarrow & r1\downarrow \end{cases}$$

$$\begin{cases} y2 \wedge \neg ri & \rightarrow & r2\uparrow \\ \neg y2 \wedge ri & \rightarrow & r2\downarrow \end{cases}$$

$$\begin{cases} r1 \vee r2 & \rightarrow & yo\uparrow \\ \neg r1 \wedge \neg r2 & \rightarrow & yo\downarrow\,. \end{cases}$$

A schematic of this circuit is in figure 3.1. There are three primary inputs, three primary outputs, and three outputs of gates that are not primary outputs, as well as six forking wires, all with an out-degree of two. Therefore there are 21 fault locations, and 42 different single stuck-at faults. I analyze the testability of the faults, by category.

A fault on a primary output is trivially testable (theorem 3.5):

$lo, r1$ [primary output]$, r2$ [primary output] stuck-at-0 and stuck-at-1.

The stuck-at-1 faults are testable in the initial state. Primary output $r1$ stuck-at-0 is testable with test

$$l1\uparrow; [lo \wedge r1]$$

and primary output $r2$ stuck-at-0 with test

$$l2\uparrow; [lo \wedge r2].$$

Either test also detects fault lo stuck-at-0.

A stuck-at-0 fault on a primary input or the output of a gate is testable by theorem 3.6: The circuit resets correctly, but an acknowledgement does not take place. I list each fault and a sequence that detects it.

$$\begin{aligned}
&l1, y1, r1 \text{ stuck-at-0}: && l1\uparrow; [lo \wedge r1]\\
&l2, y2, r2 \text{ stuck-at-0}: && l2\uparrow; [lo \wedge r2]\\
&yo \text{ stuck-at-0}: && l1\uparrow; [lo \wedge r1]; l1\downarrow; [\neg lo]\\
&ri \text{ stuck-at-0}: && l1\uparrow; [lo \wedge r1]; l1\downarrow; [\neg lo]; ri\uparrow; [\neg r1].
\end{aligned}$$

A stuck-at-1 fault on the input of a passive or lazy-active channel is testable, by theorem 3.7. Since channel L is passive, $l1$ and $l2$ stuck-at-1 are testable. For either fault, lo will be **true** after reset. Channel R is lazy-active, hence ri stuck-at-1 is testable. A test is an execution of the handshaking expansion until the first $[\neg ri]$ action, for instance

$$l1\uparrow; [lo \wedge r1].$$

A fault that causes the guard for an up-transition to be false is testable, since as a result the output of the gate is permanently **false** (theorem 3.9. For instance, $yo[y1]$ stuck-at-1 inhibits any transition $y1\uparrow$. This fault can be tested with any test for $y1$ stuck-at-0. Likewise for faults:

$$\begin{gathered}
y1[r1], y2[r2] \text{ stuck-at-0}\\
yo[y2], ri[r1], ri[r2] \text{ stuck-at-1}.
\end{gathered}$$

A fault that causes a primary output to be true is testable (theorem 3.10). In the initial state the following faults are therefore testable:

$$y1, y1[lo], y1[r1], y2, y2[lo], y2[r2] \text{ stuck-at-1}.$$

A fault that causes the guard for a down-transition to be false, and that does not change the up-guard, is testable; the initial state of the circuit is a valid state, and once the output of the gate becomes **true**, it remains **true**. In this circuit there are no faults in this category.

All faults above either cause the circuit to halt during a test, or result in a primary output being **true** after the circuit is reset. Some of the remaining faults may cause a premature firing of a production rule.

A stuck-at-0 fault that inhibits an acknowledgement is testable (theorem 3.8). Such a fault causes the circuit to reset to the correct initial state, but interrupts a sequence of acknowledgements. For instance, fault $y1[lo]$ stuck-at-0 will inhibit an acknowledgement $lo\uparrow$ for action $y1\uparrow$. A test for this fault is therefore

$$l1\uparrow; [lo \wedge r1].$$

Likewise, the following faults are testable:

$y2[lo]$ stuck-at-0 :	$l2\uparrow; [lo \wedge r2]$
$yo[y1], r1[yo]$ stuck-at-0 :	$l1\uparrow; [lo \wedge r1]; l1\downarrow; [\neg lo]$
$yo[y2], r2[yo]$ stuck-at-0 :	$l2\uparrow; [lo \wedge r2]; l2\downarrow; [\neg lo]$
$ri[r1]$ stuck-at-0 :	$l1\uparrow; [lo \wedge r1]; l1\downarrow; [\neg lo]; ri\uparrow; [\neg r1]$
$ri[r2]$ stuck-at-0 :	$l2\uparrow; [lo \wedge r2]; l2\downarrow; [\neg lo]; ri\uparrow; [\neg r2]$.

These are tests that cause the circuit to halt. Fault $yo[y1]$ stuck-at-0 may also cause a premature firing. This occurs in any state where

$$yo \wedge l1 \wedge \neg y1$$

holds, for instance if there is a transition $l1\uparrow$ immediately after the environment observes a transition $lo\downarrow$. Likewise for fault $yo[y2]$ stuck-at-0. Fault $ri[r1]$ stuck-at-0 also may cause a premature firing, in any state where

$$ri \wedge y1 \wedge \neg r1$$

holds. A state where this occurs is after the environment sets $l1\uparrow$, and ri is **true**. Likewise for fault $ri[r2]$ stuck-at-0.

The remaining five faults do not fall in any of the above categories. I derive a test for each.

- Fault yo stuck-at-1 is both inhibiting and stimulating. If it is tested as an inhibiting fault, there has to be a test, such that

  $$\neg(\neg l1 \wedge \neg yo \wedge y1) \wedge \neg(\neg l2 \wedge \neg yo \wedge y2)$$

  holds until a state where

  $$(l1 \wedge \neg yo \wedge \neg y1) \vee (l2 \wedge \neg yo \wedge \neg y2)$$

  holds (subsection 5.7). It is easy to check that test

  $$l1\uparrow; [lo \wedge r1]$$

  fulfills this condition. Furthermore, faults $r1[yo]$ and $r2[yo]$ stuck-at-1 imply that yo is invariantly **true**. These faults are testable with the same test.

- Finally, fault $y1[r1]$ stuck-at-1 is testable as an inhibiting fault if there is a test, such that

  $$\neg(\neg y1 \wedge \neg ri \wedge \neg r1)$$

  holds until a state where

  $$\neg y1 \wedge ri \wedge r1$$

  holds (subsection 5.4). The first condition does not hold in the initial state. But then there is a premature firing of $r1\uparrow$ after the circuit is reset.

This firing is obviously detectable. Likewise fault $y2[r2]$ stuck-at-1 causes a premature firing of $r2\uparrow$ in the initial state.

This concludes the analysis of each individual fault in the circuit. If N one-bit queue elements are concatenated into an N-bit queue, the results are the same. An N bit queue is fully testable, and a complete test consists of sending to, and receiving from, the queue a one and a zero.

CHAPTER 4

Testing Delay-Insensitive Combinational Logic

The ALU was sitting outside Gallifrey's frame, on the extender. Gallifrey was running a low-level program. Carman said, "Hmm-mmm". He walked over to the computer and, to the engineers' horror, he grasped the ALU board by its edges and shook it. At that instant, Gallifrey failed.

They knew where the problem lay now.

– Tracy Kidder, *The Soul of a New Machine*

1. Introduction

An efficient way of synthesizing a delay-insensitive circuit is to split it into control and datapath [47]. Whereas the control part enforces the sequencing between different parts of the computation, the datapath performs the computation of the data.

In synchronous systems, combinational logic is a feedback-free network of combinational gates that computes a function of the inputs. There are similar feedback-free delay-insensitive circuits that compute such a function, without buffering the result. Strictly speaking, such a circuit is not combinational, as a state-holding element is necessary for the implementation of all but the simplest functions. Following Seitz [66], I still call a feedback-free network that computes a function without buffering – for lack of a better term – combinational logic.

In this chapter, I show that any delay-insensitive combinational logic circuit can be reduced to a standard combinational logic circuit, for testing purposes. This combinational logic is monotone, and *any* test that detects all testable faults in this circuit will also detect all testable faults in the delay-insensitive circuit, with one exception.

Consider a process, P, computing a function f, with input x and output y, such that $y = f(x)$. Input values x are encoded on port L, and output values y on port R, using a delay-insensitive code (see chapter 2). P repeatedly waits for a valid input to occur on port L (denoted $v(L)$), then computes function f, and outputs result y on port R by raising some wires (denoted $R \Uparrow$). The environment then lowers the inputs of port L, and P lowers the outputs of port R (denoted $R \Downarrow$). Between receiving valid inputs, the environment sends the all-0 vector (for which all variables are **false**) on L; this vector is known as the *neutral value* [49]. A program for P is

$$*[[v(L)]; R \Uparrow; [n(L)]; R \Downarrow],$$

A formulation of the program as a handshaking expansion is as follows. Let L be implemented with wires $l_0, l_1, \ldots, l_{n-1}$, and R with wires $r_0, r_1, \ldots, r_{m-1}$. Then

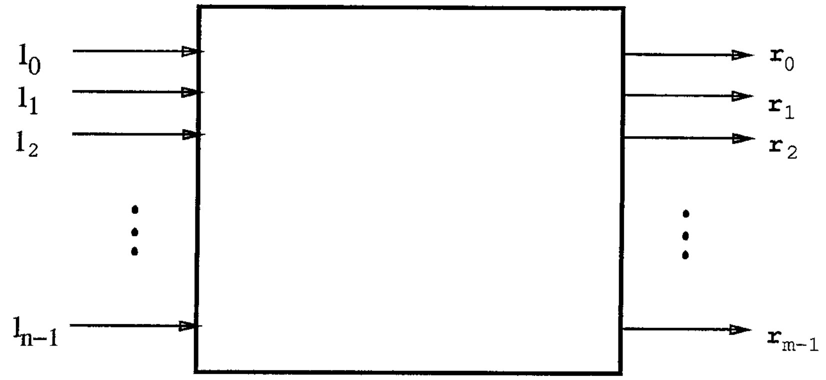

FIGURE 4.1. Model of delay-insensitive combinational logic

the handshaking expansion for P is of the form:

$$\begin{array}{ll}
*[& [l_{00} \wedge l_{01} \wedge \ldots \wedge l_{0p_0} \rightarrow r_{00}\uparrow, r_{01}\uparrow, \ldots, r_{0q_0}\uparrow \\
 & |l_{10} \wedge l_{11} \wedge \ldots \wedge l_{1p_1} \rightarrow r_{10}\uparrow, r_{11}\uparrow, \ldots, r_{1q_1}\uparrow \\
 & |\vdots \\
 & |l_{s0} \wedge l_{s1} \wedge \ldots \wedge l_{sp_s} \rightarrow r_{s0}\uparrow, r_{s1}\uparrow, \ldots, r_{sq_s}\uparrow \\
 &]; \\
 & [\neg l_0 \wedge \neg l_1 \wedge \ldots \wedge l_{n-1}]; r_0\downarrow, r_1\downarrow, \ldots, r_{m-1}\downarrow \\
], &
\end{array}$$

where

$$l_{ij} \in \{l_0, l_1, \ldots, l_{n-1}\} \text{ for } 0 \leq i \leq s, 0 \leq j \leq p_i$$

and

$$r_{ij} \in \{r_0, r_1, \ldots, r_{m-1}\} \text{ for } 0 \leq i \leq s, 0 \leq j \leq q_i,$$

and if $(l_{i0}, l_{i1}, \ldots, l_{ip_i})$ is an encoding of x, then $(r_{i0}, r_{i1}, \ldots, r_{iq_i})$ is an encoding of $f(x)$. See figure 4.1.

EXAMPLE 4.1 (SIMPLE DUAL-RAIL AND OPERATOR). *I derive an implementation for the computation of the AND-operator for two dual-rail encoded variables. The primary inputs are* a0 *and* a1, *and* b0 *and* b1, *and the result is encoded* *in primary outputs $z1$ and $z0$. Output $z1$ is raised if both $a1$ and $b1$ are* **true**, *otherwise $z0$ is raised. A program for this computation is:*

```
$*[\ \ [(ao \lor a1) \land (bo \lor b1)]; [a1 \land b1 \rightarrow z1\uparrow\ | a0 \lor b0 \rightarrow z0\uparrow];$
      $[\neg a1 \land \neg a0 \land \neg b1 \land \neg b0]; z1\downarrow, z0\downarrow$
$].$
```

Transform this program by combining the first wait-action with the guards of the if-statement:

```
$*[[\ \ a1 \land b1 \rightarrow z1\uparrow$
 $|\ \ (a1 \land b0) \lor (a0 \land b1) \lor (a0 \land b0) \rightarrow z0\uparrow$
 $]\ \ ; [\neg a1 \land \neg a0 \land \neg b1 \land \neg b0]; z1\downarrow, z0\downarrow$
$].$
```

A production rule set derived from this program is:

$$\left\{\begin{array}{ccccc} a1 & \land & b1 & \rightarrow & z1\uparrow \\ \neg a1 & \land & \neg b1 & \rightarrow & z1\downarrow \end{array}\right.$$

$$\left\{\begin{array}{rcl} (a1 \land b0) \lor (a0 \land b1) \lor (a0 \land b0) & \rightarrow & z0\uparrow \\ \neg a1 \land \neg a0 \land \neg b1 \land \neg b0 & \rightarrow & z0\downarrow . \end{array}\right.$$

It is possible that some of the wires of R are raised *before* all the necessary wires of L are raised. For instance, for a full adder, in some cases the carry can be computed when only two of the three operands have been received [66]. The sum, however, can only be computed when all three values are known (see section 6).

For any delay-insensitive implementation of program P, I make the following assumptions:

(1) The circuit has no feedback at the gate level;

(2) Each production rule for an up-transition (down-transition) has only positive (negative) literals in its guard.

In other words, there is an induced partial order among the gates of the circuit, and there are no possible pending transitions of internal variables between $R \Uparrow$ and $[n(L)]$, and between $R \Downarrow$ and the following $[v(L)]$.

2. Monotone Circuits

The state-holding elements of a delay-insensitive combinational logic circuit are solely used to distinguish the up-going phase (that sets the outputs and, possibly, internal variables) from the down-going phase (that resets the outputs and any internal variables that were previously set). During each phase, there is at most one transition for each variable: a possible transition from **false** to **true** in the up-going phase, and a transition from **true** to **false** in the down-going phase for each variable that had a transition in the preceding up-going phase.

Consider a delay-insensitive combinational logic circuit C. For each state-holding element in C, replace the guard for the down-transition with the negation of the guard for the up-transition; call the resulting circuit C_u. Circuit C_u combinational. It is important to note that during the up-phase, the behaviour of circuit C cannot be distinguished from the behavior of circuit C_u, both for the primary outputs and for all internal nodes. The only production rules that can fire are production rules that set a variable to **true**. These production rules are identical for C and C_u.

Circuit C_u is a monotone combinational circuit [25]. Much research has been done on finding test vectors for monotone combinational logic; I use this to derive test vectors for delay-insensitive combinational logic.

EXAMPLE 4.2. *I derive another implementation for the dual-rail AND program:*

$$
\begin{array}{ll}
*[[& a1 \wedge b1 \rightarrow z1\!\uparrow \\
\;\;| & (a1 \wedge b0) \vee (a0 \wedge b1) \vee (a0 \wedge b0) \rightarrow z0\!\uparrow \\
\;\;] & ;[\neg a1 \wedge \neg a0 \wedge \neg b1 \wedge \neg b0]; z1\!\downarrow, z0\!\downarrow \\
]. &
\end{array}
$$

In this implementation, the AND function is computed with an AND-gate with inputs $a1$ *and* $b1$*, and its complement with an OR-gate with inputs* $a0$ *and* $b0$*. To insure that the primary outputs are only changing* after *the primary inputs have changed, I add a signal that detects when* $a0$ *or* $a1$*, and* $b0$ *or* $b1$ *have changed. This "completion signal" is then input to two C-elements, one for each primary output. See figure* 4.2*. Call this circuit* C*.*

By changing each C-element in this figure to an AND-gate, the circuit of figure 4.3 *results. This circuit,* C_u*, is monotone. During any up-phase circuits* C *and* C_u *are equivalent. Note that circuit* C_u *is redundant. For instance, the top input (the "completion signal") to the AND-gate with output* $z1$ *can be replaced with* **true** *without changing the functionality of* C_u*. Note that no signal in* C_u *can be replaced with* **false** *without changing the functionality of the circuit.*

This implementation of the AND operator is larger than the previous. However, it is an example of a general method to implement an arbitrary function as a delay-insensitive circuit, using a completion signal. For a description, see section 5.

3. Testing Synchronous Combinational Logic

This section is a synopsis of a method to find test vectors to test all faults in monotone combinational logic. It is based on the work of Betancourt [9]. A function f is monotone if $X \leq Y$ implies $f(X) \leq f(Y)$, for all X and Y; a monotone circuit is a circuit that is equivalent to combinational logic consisting of only AND and OR gates (that is, no negative logic).

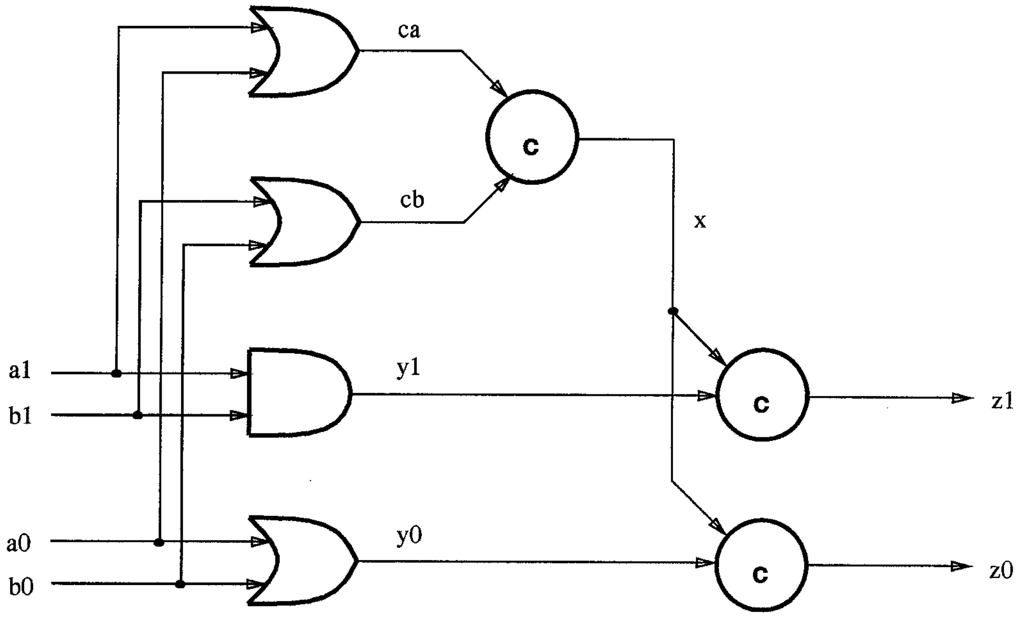

FIGURE 4.2. Dual-rail delay-insensitive circuit for AND operator

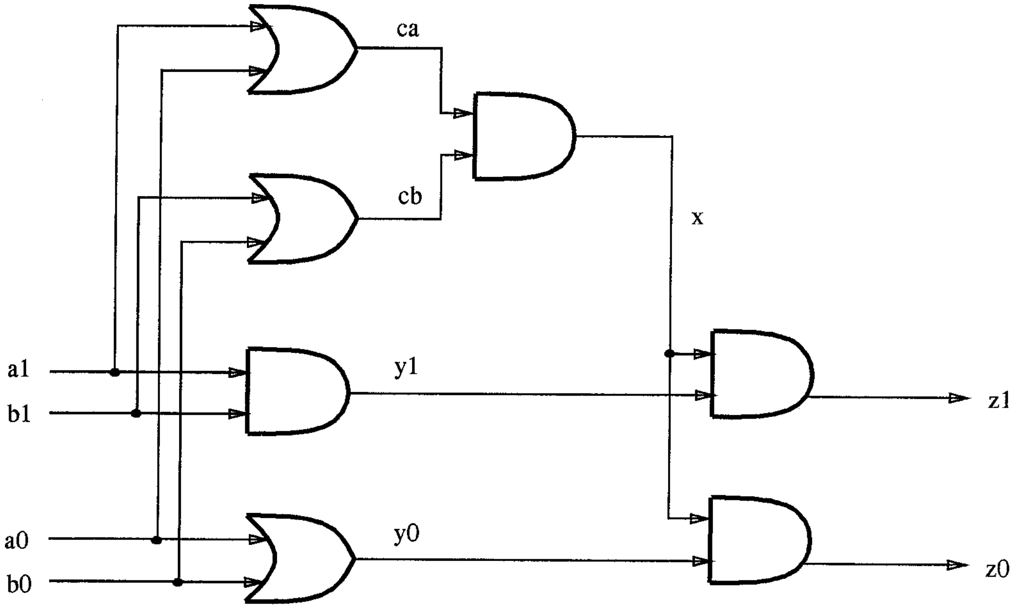

FIGURE 4.3. Equivalent synchronous circuit for AND operator

Let circuit C be a monotone circuit that implements the monotone function f. The circuit has n primary inputs, $x_0, x_1, \ldots, x_{n-1}$, and one primary output, y, where

$$y = f(x_0, x_1, \ldots, x_{n-1}).$$

Define the function h_0^i as follows:

$$h_0^i(X) = f(x_0, \ldots, x_{i-1}, x_i, \ldots, x_{n-1}) \oplus f(x_0, \ldots, x_{i-1}, \mathbf{false}, \ldots, x_{n-1}).$$

Then the set $\tau_0^i = \{t | h_0^i(t) = \mathbf{true}\}$ is the set of all test vectors that will detect if primary input x_i is stuck-at-0. Likewise, defining function h_1^i as

$$h_1^i(X) = f(x_0, \ldots, x_{i-1}, x_i, \ldots, x_{n-1}) \oplus f(x_0, \ldots, x_{i-1}, \mathbf{true}, \ldots, x_{n-1}),$$

the set $\tau_1^i = \{t | h_1^i(t) = \mathbf{true}\}$ is the set of all test vectors that will detect if primary input x_i is stuck-at-1.

Define

$$f_0^i(X) = f(x_0, \ldots, x_{i-1}, \mathbf{false}, \ldots, x_{n-1})$$

and

$$f_1^i(X) = f(x_0, \ldots, x_{i-1}, \mathbf{true}, \ldots, x_{n-1}).$$

For a monotone function f, the functions h_0^i and h_1^i reduce to

$$h_0^i(X) = x_i \wedge \neg f_0^i(X) \wedge f_1^i(X)$$

and

$$h_1^i(X) = \neg x_i \wedge \neg f_0^i(X) \wedge f_1^i(X)$$

for any input vector X.

THEOREM 4.1. *The test set*

$$\bigcup_{i=0}^{n-1} \tau_0^i \cup \bigcup_{j=0}^{n-1} \tau_1^j$$

will detect any detectable single stuck-at fault in circuit C implementing f, including faults on internal variables.

Proof: See Betancourt [9] □

This is a strong theorem, as it holds for any monotone implementation of the monotone function f.

Now define set S_0 as follows:

$$S_0 = \{p | f(p) = \textbf{true} \text{ and } p > q \implies f(q) = \textbf{false}\},$$

where $p > q$ iff for all i the ith coördinate of p is **true** when the ith coördinate of q is **true**. Each element of S_0 is known as a *minimal true vertex.*

THEOREM 4.2. *The set S_0 is sufficient to detect all testable single stuck-at-0 faults in any monotone circuit that implements f.*

Proof: Let C be a monotone circuit implementing f, and let C have a single stuck-at-0 fault. Denote the output of C for input x by $C(x)$. Then $C(x) \leq f(x)$ for all x.

Assume there is an input x such that $C(x) \neq f(x)$. Then $f(x) = \textbf{true}$ and $C(x) = \textbf{false}$. Pick a vector $y \in S_0$ such that $y \leq x$. Then $f(y) = \textbf{true}$. Apply input y to circuit C. Subsequently apply x to C. Since $y \leq x$, the only transitions that can occur in the circuit then are from **false** to **true**. If there is a transition of the output of C, it must be a transition from **false** to **true**. But since $C(x) =$ **false**, it must be that $C(y) =$ **false**. Therefore the fault will be detected by applying y to the circuit. □

Similarly, define the set S_1 as follows:

$$S_1 = \{p | f(p) = \textbf{false} \text{ and } q > p \implies f(q) = \textbf{true}\}.$$

The elements of S_1 are the *maximal false vertices.*

THEOREM 4.3. *The set S_1 is sufficient to detect all testable single stuck-at-1 faults in any monotone circuit that implements f.*

Proof: Analogous to the previous theorem. □

Corollary 4.4. *The set of all minimal true vertices and all maximal false vertices is sufficient to detect all testable single stuck-at faults in any monotone circuit implementing f.*

Reddy [62] has shown that the same result holds even if the implementation of f is redundant.

The set S_0 may not be a minimal set to detect all stuck-at-0 faults for an arbitrary implementation of f. However, for an important class of circuits each element of S_0 is necessary to test such faults.

Theorem 4.5. *All the elements of S_0 (S_1) are necessary to detect all stuck-at-0 (stuck-at-1) faults in a two-level non-redundant AND-OR (OR-AND) realization of f.*

Proof: See Betancourt [9]. □

4. Sufficient Tests for Delay-Insensitive Combinational Logic

For any delay-insensitive combinational logic circuit C, there is a monotone combinational logic circuit C_u, such that C and C_u are equivalent during any up-phase. A test for C_u can be used to test for faults in C also.

For combinational logic, a test is a sequence of test vectors that is applied to the circuit. In order to be able to apply the same sequence of test vectors to the delay-insensitive combinational logic circuit, the all-0 vector has to be applied between consecutive test vectors. In this manner the environment of the delay-insensitive circuit behaves according to the four-phase (return-to-zero) handshaking protocol. Note that during an up-phase there are only transitions from **false** to **true**, and during a down-phase only from **true** to **false**.

THEOREM 4.6. *Let t be an input vector that detects a single stuck-at fault on gate G_u of circuit C_u. Then the equivalent fault on gate G of circuit C is also detectable by applying t from the initial state.*

Proof: Assume, without loss of generality, that an input, y, of G_u is stuck-at-0. Input vector t detects fault y stuck-at-0. After applying t, variable y is **true** for the correct circuit, and **false** for the faulty circuit. The fault is detectable, so there is a primary output, z, whose value is **false** in the faulty circuit and **true** in the correct circuit. In addition, there is a series of variables, $y_0, y_1, \dots, y_n$, with $y_0 = y$, $y_n = z$, and, for $0 \leq i < n$, y_i is an input to the gate with output y_{i+1}, such that y_i is **true** in the correct circuit, and **false** for the faulty one.

Therefore there is a series of variables for which an up-transition does not take place in circuit C_u with fault y stuck-at-0. Now apply vector t to circuit C in the initial state (where all variables are **false**). Recall that C and C_u are identical during the up-phase. If y is stuck-at-0 in circuit C, then there is the same chain $(y_0, y_1, \dots, y_n)$ of variables, all of which are **true** in the correct circuit and **false** in the faulty circuit. In particular, primary output z is **true** for the correct circuit, and **false** for the faulty one. Vector t detects fault y stuck-at-0 in C.

If y is stuck-at-1, then there similarly is a series of variables from y to a primary output z, such that these variables are **false** in the correct circuit, and **true** in the faulty one. If vector t is applied to the delay-insensitive circuit from the initial state, then for the correct circuit z remains **false**, and there is a premature firing of $z\uparrow$ in the faulty circuit. □

Note that the order in which the inputs are raised in the delay-insensitive circuit is irrelevant. Any test that can detect all testable faults in C_u will also detect the equivalent faults in C. In general, circuit C_u is redundant, so that not all its faults are testable. It is possible, however, that the equivalent faults in C *are* testable. I investigate when a variable in C_u is redundant, and if the corresponding fault in

C is testable.

EXAMPLE 4.3. *Consider again the delay-insensitive circuit C, the implementation of the dual-rail AND operator in figure 4.2. In the equivalent monotone circuit C_u of figure 4.3, the expression that $z1$ evaluates is $a1 \wedge b1$. A minimal true vertex for this expression is $a1 \wedge b1$. When $a1 \wedge b1$ is supplied to C_u as a test vector, $z1$ remains* **false** *if $a1$, $b1$, $y1$, or $z1$ is stuck-at-0, as these variables are not redundant in C_u. Consequently, if test vector $a1 \wedge b1$ is supplied to circuit C, it detects if $a1$, $b1$, $y1$, or $z1$ is stuck-at-0.*

After $a1 \wedge b1$ is supplied to C, $z1$ is **true**. *In the following down-phase a transition $z1\downarrow$ will follow transitions $a1\downarrow$ and $b1\downarrow$, in the correct circuit. If the input x of the C-element with output $z1$ is stuck-at-1, then there is no transition $z1\downarrow$. Hence this fault is detectable in circuit C. In circuit C_u, however, the input x to the AND gate with output $z1$ is redundant (it can be replaced with* **true***), therefore no test vector detects that fault.*

The following theorem states that C_u has no redundant primary outputs. This trivial fact is necessary for the next theorem.

THEOREM 4.7. *If circuit C is not redundant, then circuit C_u has no redundant primary outputs.*

Proof: Let z be a primary output of C, and let z_u be the corresponding primary output of C_u. Variable z_u cannot be replaced with **true**, as all primary outputs are **false** in the initial state.

If z_u can be replaced with **false**, then for all input vectors to C_u primary output z_u is **false**. Therefore z is **false** during every up-phase of C. Since there are no transitions from **false** to **true** in any down-phase, z is redundant in C. A contradiction. □

I now show that no output of a gate in C_u may be replaced with **false** without changing the functionality of C_u.

THEOREM 4.8. *Let x be a primary input or the output of a gate in a non-redundant delay-insensitive circuit C, and let x_u be the corresponding variable in circuit C_u. Then replacing x_u with* **false** *alters the functionality of C_u.*

Proof: By contradiction. Suppose there is a set of primary inputs, and outputs of gates that may be replaced with **false** without altering the functionality of C_u. Let x_u be a member of this set, such that in the graph induced by the circuit there is no path from x_u to any other member of the set. Such an element exists, since the graph is acyclic.

The corresponding variable x in C is not redundant. Therefore, there is an input vector that results in a transition $x\uparrow$. Since x is not a primary output, there is a series of acknowledgements, resulting in a transition of a primary output that is an acknowledgement of $x\uparrow$. Since C and C_u are identical during any up-phase, there is a transition of a primary output that acknowledges transition $x_u\uparrow$. For a fault x_u stuck-at-0, this acknowledgement does not take place; x_u can not be replaced with **false**. □

As a consequence, a stuck-at fault on a primary input or on the output of a gate in C is testable, as shown next.

THEOREM 4.9. *Let x be a primary input or the output of a gate in circuit C. Then a test set that detects all testable faults in circuit C_u also detects faults x stuck-at-0 and x stuck-at-1.*

Proof: Consider the variable x_u in circuit C_u that is equivalent to variable x in C. Variable x_u cannot be replaced with **false** without altering the functionality of C_u. Therefore fault x_u stuck-at-0 is testable, say with input vector **t**. The same vector **t** also tests fault x stuck-at-0, by theorem 4.6.

To detect fault x stuck-at-0, there has to be a transition $x\uparrow$ during the test. If x is stuck-at-1, then there will not be a corresponding transition $x\downarrow$ during a down-phase of the test. Then there is no acknowledgement for this transition $x\downarrow$; the down-phase ends with at least one primary output **true**. □

The only case left is the testability of faults on inputs of gates. Let y be the input of a gate with output z in circuit C, and let y_u and z_u be the corresponding variables in C_u. If y_u is not redundant, then there is a test that detects a fault on y_u; the same test detects the corresponding fault on y.

If y_u can be replaced with **true** without altering the functionality of C_u, and $z\downarrow$ is not an acknowledgement for $y\downarrow$, then fault y stuck-at-1 is not testable. Input y is not necessarily a redundant input for the gate with output z; y is added to the guard for $z\uparrow$ to avoid a hazard on z. An example of this is in section 5. If a variable is added to a production rule not for functionality but for the avoidance of a hazard, then a fault on such a variable is not testable. A test point has to be added to the circuit to make it fully testable.

If y_u can be replaced with **false** without altering the functionality of C_u, then no transition $y\uparrow$ in C is ever acknowledged; variable y is redundant in the guard for $z\uparrow$, and is used in the guard for $z\downarrow$ to insure correctness of a down-phase. Then an additional test vector has to be generated for this fault, for instance using the D-algorithm (see section 7).

The latter case occurs in the adder example of section 6. It is possible to redesign the adder so that any variable that occurs in a down-guard also occurs non-redundantly in an up-guard; this significantly reduces the performance, however.

5. Application: Combinational Logic in AND-OR Form

The following is a method to implement any function as a delay-insensitive combinational logic circuit [19]:

(1) Create a two-level AND-OR network of gates implementing the desired function, such that for any legal input vector each OR gate has at most one input **true** at any time;

(2) Create a completion signal that detects that a legal input vector has been received, and that all inputs have been reset to **false**. For a set of dual-rail encoded signals, each pair is merged with an OR gate, and the output of each such OR gate is input to a C-element that is the completion signal;

(3) Finally, the completion signal is input to a number of C-elements, each of which has an output of the AND-OR network as the other input.

EXAMPLE 4.4. *I derive a circuit that computes the majority function of three inputs. The output is 1 if at least two inputs are 1, and 0 otherwise. In the implementation there are three input channels, A, B, and C, and one output channel, Z. Each channel is dual-rail encoded, with variables* $a1$ *and* $a0$*;* $b1$ *and* $b0$*;* $c1$ *and* $c0$*; and* $z1$ *and* $z0$*. To derive an AND-OR network for this function, I write the outputs as a disjunction of conjunctions of inputs:*

$$z1 = a1 \wedge b1 \vee a1 \wedge c1 \vee b1 \wedge c1$$
$$z0 = a0 \wedge b0 \vee a0 \wedge c0 \vee b0 \wedge c0.$$

This description cannot be transformed directly into an AND-OR network, since the terms in the disjunctions are not mutually exclusive. A description with mutually exclusive terms is:

$$z1 = a1 \wedge b1 \vee a1 \wedge b0 \wedge c1 \vee a0 \wedge b1 \wedge c1$$
$$z0 = a0 \wedge b0 \vee a0 \wedge b1 \wedge c0 \vee a1 \wedge b0 \wedge c0.$$

With the addition of the completion signal, the circuit is in figure 4.4

For the following fault analysis I assume that each variable is dual-rail encoded. The problem of testing all stuck-at faults in such a circuit is greatly reduced with the following observations:

- Consider a C-element whose output is a primary output (step 3). Faults on the inputs and outputs of the C-element are testable if there is a test vector that raises the primary output, that is, if the primary output is not redundant.
- Consequently, any fault on the inputs and outputs of the C-element whose output is the completion signal (step 2) is testable.
- Consider an OR gate that merges the two signals of a dual-rail encoded variable (step 2). The inputs to this gate are primary inputs. The output of the OR gate is the input to the C-element generating the completion signal, hence faults on this output are testable. If an input is stuck-at-1, then the output of the OR gate is stuck-at-1, so that fault is also testable. Finally, consider the case when an input to the OR gate is stuck-at-0. This fault is testable if there is an input vector for which the variable is raised. In other words, if no primary input is redundant, then any stuck-at fault on the inputs of the OR gate is testable.

The faults that remain to be considered are faults in the AND-OR network (step 1). If there are no redundant primary inputs and outputs, then all other faults in the circuit are testable with any test that tests all faults in the AND-OR network. Therefore the problem of finding a test set for the circuit is reduced to finding a test set for the AND-OR network. Since such a network does not contain any state-holding elements, any technique to find test vectors for combinational logic may be used.

A test consisting of all minimal true vertices is necessary (theorem 4.5) and sufficient (theorem 4.2) to test all stuck-at-0 faults in the AND-OR network. During

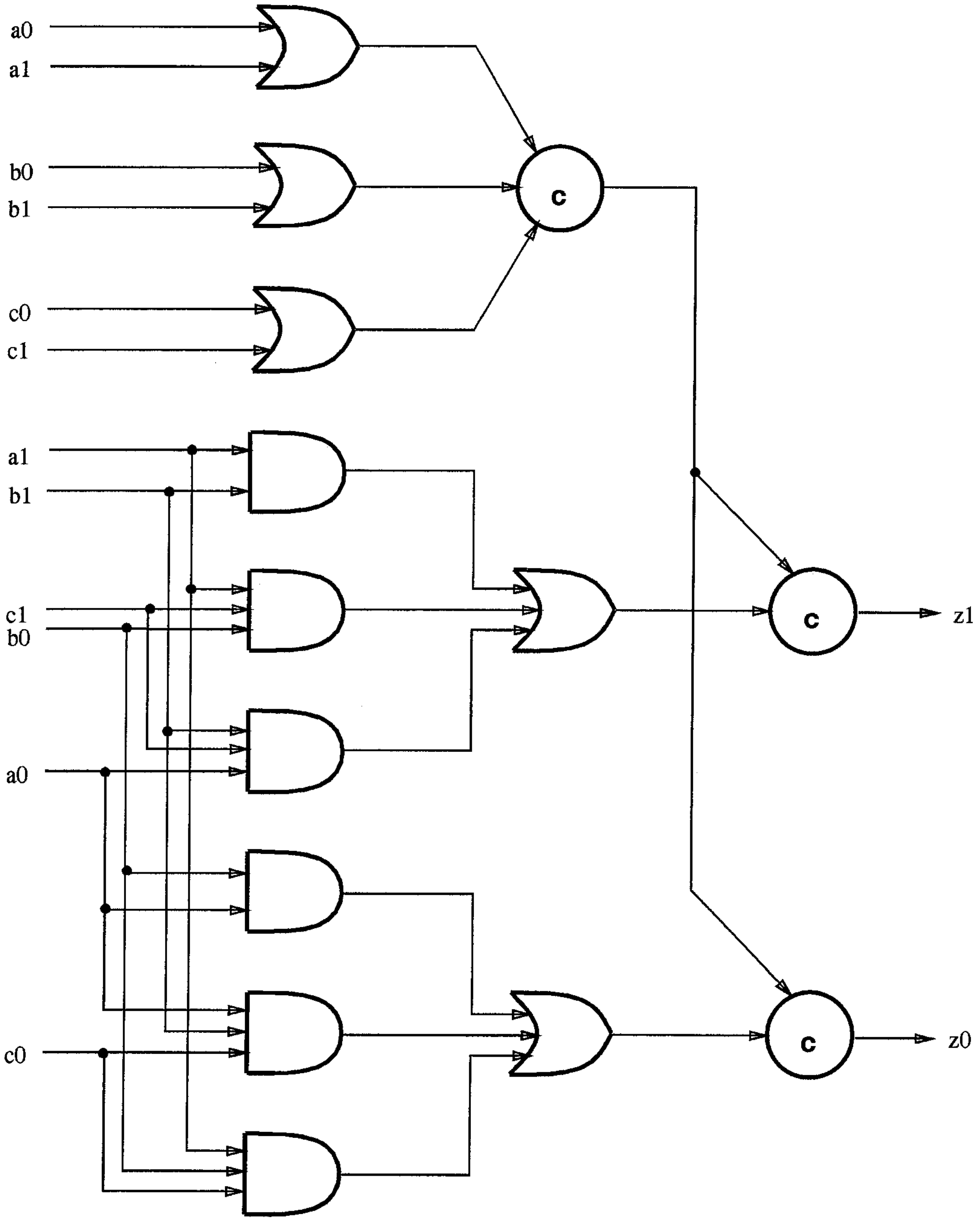

FIGURE 4.4. Majority circuit

this test each variable in the network will be raised at least once. If there is a stuck-at-1 fault on the output of an AND gate, or on an input or the output of an OR gate, then there is a down-phase that does not complete during this test (that is, a primary output will not have a down-transition). Consequently, such faults are also testable with the same test.

Finally, I consider the faults where an input of an AND gate is stuck-at-1. Consider an AND gate with inputs a, b, and c, and output u:

$$\left\{\begin{array}{ccccccc} a & \wedge & b & \wedge & c & \to & u\uparrow \\ \neg a & \vee & \neg b & \vee & \neg c & \to & u\downarrow . \end{array}\right.$$

(The analysis generalizes to AND gates with an arbitrary number of inputs). Let u be used as part of the implementation of primary output z. Fault a stuck-at-1 is testable if there is a test vector t such that

(1) the production rule for $u\uparrow$ fires prematurely. This can only be in a state where $\neg a \wedge b \wedge c \wedge \neg u$; and

(2) primary output z will have a premature up-transition after transition $u\uparrow$, so that the fault is detectable. Therefore t must cause the completion signal to be raised.

I prove that there is a minimal true vertex for a variable other than z that will test fault a stuck-at-1, if a is not redundant in the AND-OR network. Variable z is part of an output channel. For the proof below I assume that this channel is a dual-rail encoded bit. The proof generalizes to other encoding schemes. Let $z = z1$. In the proof I show that there is a minimal true vertex for $z0$ that detects fault a stuck-at-1.

THEOREM 4.10. *Let a be a non-redundant input to an AND gate of an AND-OR network, as above. Then there is a minimal true vertex of a primary output that tests fault a stuck-at-1.*

Proof: Variables a, b, and c are primary inputs. If only b and c are raised from the initial state, then neither $z1$ nor $z0$ can have an up-transition: if $z1$ can have an up-transition, then u is redundant in the implementation of $z1$; if $z0$ has an up-transition, then an additional transition $a\uparrow$ cannot cause a transition $z1\uparrow$ (since $\neg z0 \vee \neg z1$ is invariant for a dual-rail encoded channel), so that u is again redundant.

The only remaining proof obligation is to show that there is a minimal true vertex for $z0$ that includes $\neg a$, b, and c. For such a true vertex the completion signal has an up-transition, and, since it raises $z0$, $z1$ must be **false**.

Observe that there is no true vertex for $z0$ that includes a, b, and c all **true**. Since $a \wedge b \wedge c$ causes $u\uparrow$, any such vertex would also cause a transition $z1\uparrow$. Suppose that there is no true vertex for $z0$ that includes $\neg a$, b, and c. Then no true vertex for $z0$ includes both b and c **true**, only true vertices of $z1$ have both b and c **true**. But then a is a redundant variable in the AND gate with output u. A contradiction.

There is a true vertex for $z0$ that includes $\neg a$, b, and c, and that will detect a fault a stuck-at-1. □

From the preceding analysis it follows that

THEOREM 4.11. *For an implementation of delay-insensitive combinational logic with an AND-OR network to which a completion signal is added, all testable faults are testable with a test set consisting of all minimal true vertices for the outputs of the AND-OR network.*

If this method is used to derive delay-insensitive combinational logic, there are some single stuck-at-1 faults that are not testable without the addition of test circuitry. Consider the case of the following function to be implemented:

$$z = f(a,b,c) = \neg c \vee \neg a \wedge \neg b.$$

In a delay-insensitive implementation, all variables are dual-rail encoded. A diagram of function f in terms of the dual-rail encoded variables is:

f	$a0 \wedge b0$	$a1 \wedge b0$	$a1 \wedge b1$	$a0 \wedge b1$
$c0$	1	1	1	1
$c1$	1	0	0	0

An easy – but wrong – implementation of $z1$ and $z0$ as an AND-OR network is

$$z1 = c0 \vee a0 \wedge b0$$
$$z0 = c1 \wedge a1 \vee c1 \wedge b1,$$

consisting of an AND gate followed by an OR gate, and two AND gates followed by an OR gate. This is a direct conversion into a dual-rail code of the original formula for z. For $z1$, in case $a0 \wedge b0 \wedge c0$ holds, both inputs of the OR-gate can become **true**. This may cause a hazard [12]. Similarly for $z0$. To avoid this from occurring, a term can be added to the AND-gate, yielding the following implementation:

$$z1 = c0 \vee c1 \wedge a0 \wedge b0$$
$$z0 = c1 \wedge a1 \vee c1 \wedge a0 \wedge b1.$$

Consider this implementation of $z1$. Since $\neg(c0 \wedge c1)$ is an invariant, the two inputs of the OR gate can never be **true** at the same time. Note that the implementation is not redundant. Figure 4.5 shows the circuit for $z1$, including the completion signal (variable $z0$ is omitted).

Consider the fault stuck-at-1 for the input $c1$ to the AND-gate. To test this fault, $c1$ must be **false**, $a0$ and $b0$ **true**, and $c0$ **false**. Then the output of the AND-OR network is **false** in the correct circuit, and **true** in the faulty one. There are two ways to propagate an erroneous value of the output of the AND-OR network to the output $z1$:

(1) The completion signal is **true**, and $z1$ is **false** for the correct circuit. This occurs during the up-phase. In the presence of the fault, a transition $z1\uparrow$

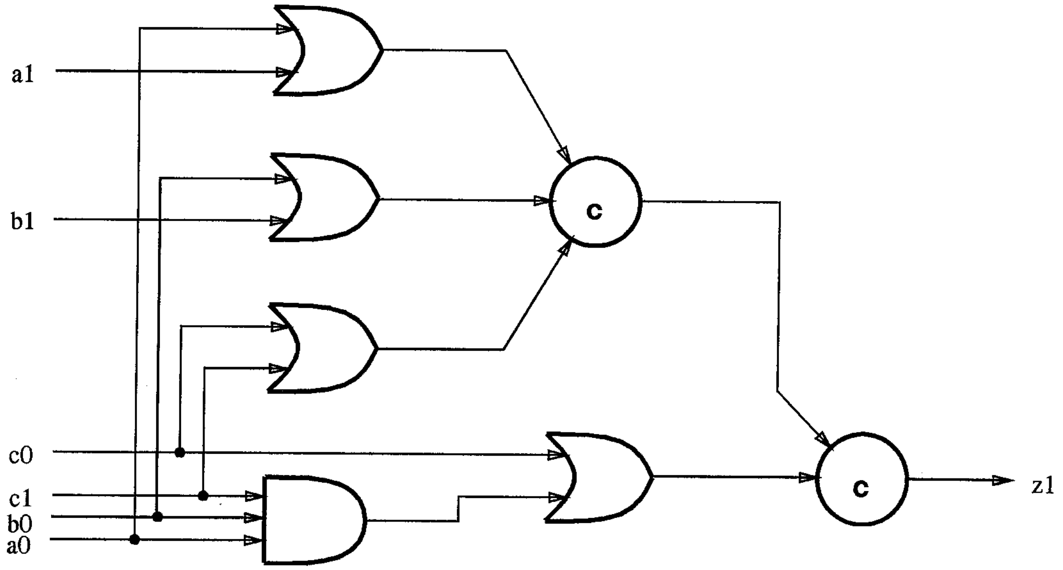

FIGURE 4.5. Implementation of $z1$

occurs. For the completion signal to be **true** during an up-phase, however, either $c0$ or $c1$ must be **true**, which is not the case;

(2) The completion signal is **false**, and $z0$ is **true** for the correct circuit. This occurs during the down-phase. Then a transition $z1 \downarrow$ takes place for the correct circuit, but not for the faulty circuit. For the completion signal to be **false** during a down-phase all primary inputs must be **false**. However, $a0$ and $b0$ are **true**.

In either case it is not possible to propagate the erroneous signal to the primary output $z1$. Therefore the fault is not testable.

The reason that a fault stuck-at-1 on the input $c1$ of the AND-gate is not testable, is that it is, in some sense, redundant. The signal itself is not redundant, according to the definition of redundancy, and there is no smaller AND-OR network that will correctly implement the function. The problem is that $c1$ was added to the AND-gate not because of functionality, but to avoid a possible hazard.

There are three ways to approach this problem.

(1) Disregard faults stuck-at-1 for inputs to AND gates when the input is added to avoid a possible hazard. A complete test will detect all testable stuck-at faults, but the fault coverage is not 100 percent.

(2) Do not add terms to avoid hazards. In the example above, the implementation of $z1$ would be $c0 \vee a0 \wedge b0$. This may cause a hazard, and the resulting circuit is not delay-insensitive. For such a hazard to occur, however, the propagation delays must be widely different. We can extend the concept of an isochronic region, and assume that the propagation delays of the combinational gates are roughly the same, or that the propagation delays of the combinational gates are smaller than the propagation delays through any feedback circuitry.

(3) Add test circuitry. A test point to observe each output of the AND-OR network is sufficient to make all faults testable. These test points make it possible to test the AND-OR network independently of the remainder of the circuit. The AND-OR network is not redundant, and consists solely of combinational gates; it is fully testable. Any method to derive a test set for combinational logic can be used. The remainder of the circuit consists of the completion signal and C-elements that generate the primary outputs. Faults in this part of the circuit are testable as before.

Of these three alternatives, only the last yields a fully testable delay-insensitive circuit. It is, however, the slowest solution, as well as the costliest in area. If we can make assumptions on propagation delays, then the second alternative yields the smallest circuits, as well as the fastest of these three possibilities.

Alternatively, it is possible to derive logic in a different manner, without adding terms to avoid hazards, or employing hazard-avoidance algorithms. An example is the ripple-carry adder in the following section, which is derived directly from the program it implements; function evaluation and completion detection are not separated. The ripple-carry adder has no untestable faults.

6. Example: Ripple-carry Adder

As an example of testing delay-insensitive combinational logic, I analyze a circuit for a ripple-carry adder [49]. The one-bit adder element has inputs *a0*,*a1*, *b0*, and *b1*, for both operands, and *c0* and *c1* for carry-in; it has outputs *d0* and *d1* for carry-out, and *s0* and *s1* for the sum (result).

The production rule set is

$$\left\{\begin{array}{rcl} (((a1 \wedge b1) \vee (a0 \wedge b0)) \wedge c1) & \vee & \\ (((a1 \wedge b0) \vee (a0 \wedge b1)) \wedge c0) & \rightarrow & s1\uparrow \\ \neg c1 \wedge \neg c0 & \rightarrow & s1\downarrow \end{array}\right.$$

$$\left\{\begin{array}{rcl} (((a1 \wedge b1) \vee (a0 \wedge b0)) \wedge c0) & \vee & \\ (((a1 \wedge b0) \vee (a0 \wedge b1)) \wedge c1) & \rightarrow & s0\uparrow \\ \neg c1 \wedge \neg c0 & \rightarrow & s0\downarrow \end{array}\right.$$

$$\left\{\begin{array}{rcl} (a1 \wedge b1) \vee ((a1 \vee b1) \wedge c1) & \rightarrow & d1\uparrow \\ \neg a1 \wedge \neg a0 \wedge \neg b0 \wedge \neg b1 & \rightarrow & d1\downarrow \end{array}\right.$$

$$\left\{\begin{array}{rcl} (a0 \wedge b0) \vee ((a0 \vee b0)) \wedge c0) & \rightarrow & d0\uparrow \\ \neg a1 \wedge \neg a0 \wedge \neg b0 \wedge \neg b1 & \rightarrow & d0\downarrow . \end{array}\right.$$

There are six primary inputs, four primary outputs, and 22 inputs of gates; the total number of fault locations is 32, and there are 64 possible single stuck-at faults.

All faults are tested if each input combination is used once (eight vectors), with the exception of faults on variables that occur only in the guard for a down-transition. These are *a0* and *b0* in $d1\downarrow$ and *a1* and *b1* in $d0\downarrow$. Fault *a0*[*d1*] stuck-at-1 is testable, since it causes the guard for $d1\downarrow$ to be **false**; the fault is detected with any test vector for which *d1* is raised. Likewise, faults *a1*[*d0*], *b0*[*d1*], and *b1*[*d0*] stuck-at-1 are testable.

Fault *a0*[*d1*] stuck-at-0 is only detected if the inputs are changed in a particular order. Since *a0* occurs only in the guard for $d1\downarrow$, the fault has to be tested during a down-phase. For the corresponding up-phase, *a0* and *d1* have to be **true**, hence *b1* and *c1* are **true**. Subsequently there will be a premature transition $d1\downarrow$ after a transition $b1\downarrow$. A test for this fault is therefore

$$a0\uparrow, b1\uparrow, c1\uparrow; [d1 \wedge s0]; b1\downarrow .$$

For faults *a1*[*d0*], *b0*[*d1*], and *b1*[*d0*] stuck-at-0 we get tests

$$\begin{gathered} a1\uparrow, b0\uparrow, c0\uparrow; [d0 \wedge s1]; b0\downarrow \\ a1\uparrow, b0\uparrow, c1\uparrow; [d1 \wedge s0]; a1\downarrow \\ a0\uparrow, b1\uparrow, c0\uparrow; [d0 \wedge s1]; a0\downarrow . \end{gathered}$$

The size of the test can be reduced to six test vectors. Consider only the up-transitions, and transform the circuit into a combinational logic by having as down-guards the negation of the up-guards.

The outputs as a function of the inputs are

$$
\begin{aligned}
d0 &= a0 \wedge b0 && \vee \\
&\ (a0 \vee b0) \wedge c0 \\
d1 &= a1 \wedge b1 && \vee \\
&\ (a1 \vee b1) \wedge c1 \\
s0 &= a0 \wedge b0 \wedge c0 && \vee \\
&\ a0 \wedge b1 \wedge c1 && \vee \\
&\ a1 \wedge b0 \wedge c1 && \vee \\
&\ a1 \wedge b1 \wedge c0 \\
s1 &= a1 \wedge b1 \wedge c1 && \vee \\
&\ a1 \wedge b0 \wedge c0 && \vee \\
&\ a0 \wedge b1 \wedge c0 && \vee \\
&\ a0 \wedge b0 \wedge c1.
\end{aligned}
$$

Consider output *s0*. Fault *a0*[*s0*] stuck-at-0 is testable with either of two test vectors:

$$a0 \wedge b0 \wedge c0$$

$$a0 \wedge b1 \wedge c1.$$

Fault *a1*[*s0*] stuck-at-0 is also testable with either of two test vectors:

$$a1 \wedge b0 \wedge c1$$

$$a1 \wedge b1 \wedge c0.$$

Since the function for *s0* is symmetric in *a0*, *b0*, and *c0*, and in *a1*, *b1*, and *c1*, the test vectors for stuck-at-0 faults on other inputs follow easily. Similarly for faults on the gate with output *s1*.

Next, consider output *d0*. Fault *a0*[*d0*] stuck-at-0 is testable with either of the following test vectors:

$$a0 \wedge b0 \wedge c1$$

$$a0 \wedge b1 \wedge c0.$$

Likewise for faults $b0[d0]$ stuck-at-0 and $c0[d0]$ stuck-at-0, and for faults on the gate with output $d1$.

It follows that the minimal number of test vectors for this adder is six, for instance

$$a0 \wedge b0 \wedge c1$$

$$a0 \wedge b1 \wedge c0$$

$$a0 \wedge b1 \wedge c1$$

$$a1 \wedge b0 \wedge c0$$

$$a1 \wedge b0 \wedge c1$$

$$a1 \wedge b1 \wedge c0.$$

This test set will test all faults (stuck-at-0 and stuck-at-1) for variables that occur in up-guards of the production rule set, as well as on the primary inputs and primary outputs.

In addition, to test faults $a0[d1]$, $b0[d1]$, $a1[d0]$, and $b1[d0]$ stuck-at-0 some more tests are needed, as analyzed previously. It is easy to see, that these tests can be combined with the six test vectors above. All that is needed is that the order in which signals are changing during the down-phases be specified.

A test to test all faults in the complete circuit is

$$a0\uparrow, b0\uparrow, c1\uparrow; [d0 \wedge s1]; a0\downarrow, b0\downarrow, c1\downarrow; [\neg d0 \wedge \neg s1];$$

$$a0\uparrow, b1\uparrow, c0\uparrow; [d0 \wedge s1]; a0\downarrow; b1\downarrow, c0\downarrow; [\neg d0 \wedge \neg s1];$$

$$a0\uparrow, b1\uparrow, c1\uparrow; [d1 \wedge s0]; b1\downarrow; a0\downarrow, c1\downarrow; [\neg d1 \wedge \neg s0];$$

$$a1\uparrow, b0\uparrow, c0\uparrow; [d0 \wedge s1]; b0\downarrow; a1\downarrow, c0\downarrow; [\neg d0 \wedge \neg s1];$$

$$a1\uparrow, b0\uparrow, c1\uparrow; [d1 \wedge s0]; a1\downarrow; b0\downarrow, c1\downarrow; [\neg d1 \wedge \neg s0];$$

$$a1\uparrow, b1\uparrow, c0\uparrow; [d1 \wedge s0]; a1\downarrow, b1\downarrow, c0\downarrow; [\neg d1 \wedge \neg s0].$$

For an array of N one-bit adder elements, the carry-out ($d0$ and $d1$) of an element is connected to the carry-in ($c0$ and $c1$) of the next. The number of different test vectors to test such an array is still six, regardless of the size of the array. With the six test vectors above, the same vector can be supplied to each element, with the exception of vectors

$$a0 \wedge b0 \wedge c1$$

and

$$a1 \wedge b1 \wedge c0,$$

which have to be alternated between elements.

The only problem, again, is the stuck-at-0 faults for variables that occur in a down-guard only. These cause a premature transition, either $d0\downarrow$ or $d1\downarrow$. These signals may not be directly observable by the environment, since they encode the carry-out. Such a premature transition will, however, cause a premature transition of either $s1\downarrow$ or $s0\downarrow$ for the next element; this is observed by the environment.

Note that the sequencing between actions needed to detect these faults still does not cause the testing time for the array of full adders to increase; all elements can be tested in parallel.

In the remainder of this chapter, I investigate how to test delay-insensitive combinational logic by adapting algebraic testing methods for combinational logic.

7. The D-algorithm for Delay-insensitive Circuits

The D-algorithm (see Appendix) can be extended to find a test vector for a stuck-at fault in a delay-insensitive combinational logic circuit. Since such a circuit is feedback-free, the regular forward and backward propagation methods can be used. The difference with combinational logic circuits is that there are state-holding elements. As seen previously, the purpose of the state-holding elements is to have the circuit obey the handshaking protocol. In propagating a fault through a state-holding element, it is important to note whether the circuit is in an up-going or a down-going phase.

Recall that for the D-algorithm there are five different logic values: 1 (**true**), 0 (**false**), X (don't care), D (**true** for the correct circuit, and **false** for the faulty circuit), and $\overline{\mathrm{D}}$ (**false** for the correct circuit, and **true** for the faulty one).

Both during any up-phase and during any down-phase a delay-insensitive combinational logic circuit is equivalent to a monotone combinational logic circuit. This simplifies the propagation algorithm somewhat. If there is a signal D, then there will not be a signal $\overline{\mathrm{D}}$; if there is a signal $\overline{\mathrm{D}}$, then there will not be a signal D. For arbitrary combinational logic, it is possible that two faulty signals cancel each other. For instance, if an OR-gate has an input D and an input $\overline{\mathrm{D}}$, then the output is 1, and neither faulty signal can be propagated. This situation does not occur with delay-insensitive circuits.

The following is an overview of forward and backward propagation of faults. The forward and backward propagation of faulty signals through combinational gates is the same as for the standard D-algorithm.

7.1. Forward Propagation. For a state-holding element, the output depends not only on the inputs, but also on the state of the gate. The faulty signal may be propagated either during an up-phase, or during a down-phase.

Let G be a state-holding element. Transform G into gate G_u by replacing the guard for the down-transition with the negation of the guard for the up-transition. Gate G_u, a combinational gate, is equivalent to G during the up-phase. Therefore the propagation of D and $\overline{\mathrm{D}}$ is the same for G as for G_u. For instance, if G is a C-element, then G_u is an AND gate; a D ($\overline{\mathrm{D}}$) input causes a D ($\overline{\mathrm{D}}$) output if all other inputs are 1 or D ($\overline{\mathrm{D}}$).

During a down-phase, G only propagates a faulty signal if the output of the gate is 1 after the up-phase. Transform G into G_d by replacing the guard for the up-transition with the negation of the guard for the down-transition. Then G_d is equivalent to G during the down-phase, if the output of G is 1 after the up-phase. The propagation of D and $\overline{\mathrm{D}}$ is the same for G as for G_d. In addition an input vector has to be computed for which the output of G is 1 after the up-phase. This is done by backward propagation.

If G is a C-element, then G_d is an OR gate. Propagate the D ($\overline{\mathrm{D}}$) signal by setting all X inputs to 0, so that the output of the C-element is D ($\overline{\mathrm{D}}$).

7.2. Backward Propagation. For backward propagation, transform G again into G_u for the up-phase, and G_d for the down-phase. The backward propagation for these combinational gates is as before.

If G is a C-element, and the output is D, then all inputs must be 1 for detection during an up-phase; at least one input is 1 for detection during a down-phase. Likewise, if the output is $\overline{\mathrm{D}}$, then during an up-phase at least one input is 0; during a down-phase all inputs are 0.

7.3. An Example. As an example, I reëxamine the circuit of example 4.2, which is an implementation of an AND-operator. See figure 4.6.

Consider fault input x of the C-element with output $z1$ stuck-at-1. To apply the D-algorithm, set all variables to X, except this input, which is $\overline{\mathrm{D}}$. Assume that the fault is detectable during an up-phase. For the forward propagation, $z1$ is $\overline{\mathrm{D}}$ if $y1$ is 1. What remains is backward propagation. Since $y1$ is 1, both $a1$ and $b1$ have to be 1. Since x is $\overline{\mathrm{D}}$, one of the inputs of the C-element generating signal x must be 0. But since $a1$ and $b1$ are both 1, both inputs of that C-element are 1. This is a contradiction. Therefore the fault is *not* detectable during any up-phase.

Now assume the fault is detectable during a down-phase. With forward propagation, $z1$ is $\overline{\mathrm{D}}$, so $y1$ is 0. In addition, $z1$ must be 1 after the up-phase. It is easy

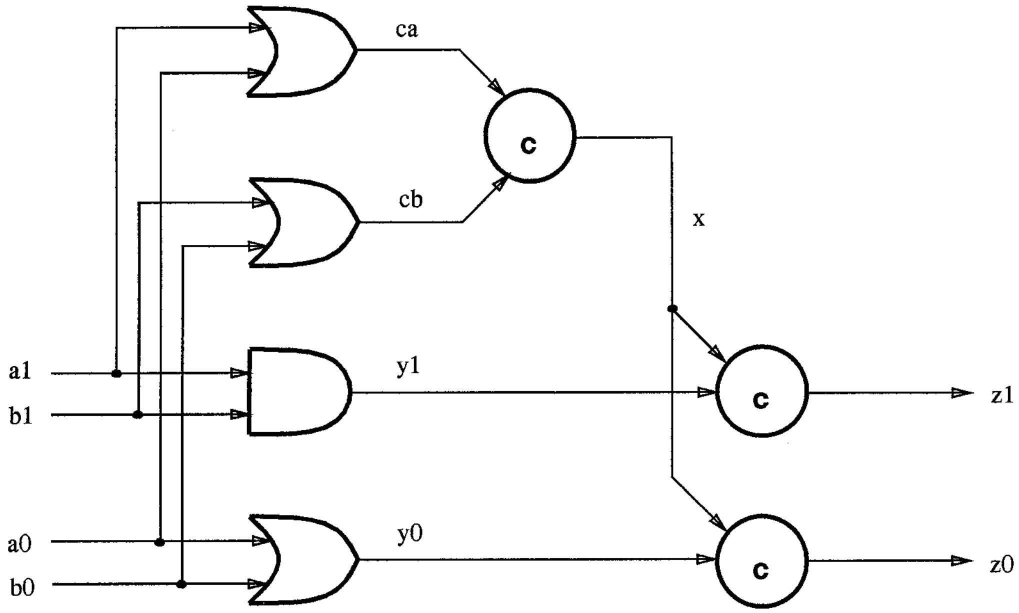

FIGURE 4.6. AND operator

to check that this occurs when $a1$ and $b1$ are both 1. Then, during the down-phase $y1$ is 0, so at least one of $a1$ and $b1$ is 0. Since x is $\overline{\mathrm{D}}$, both inputs to the C-element with output x must be 0. Propagating backward through the two OR-gates, $a1$, $b1$, $a0$, and $b0$ are all 0. It follows that a test to detect the fault is

$$a1\uparrow, b1\uparrow; [z1]; a1\downarrow, b1\downarrow; [\neg z1].$$

An efficient way to use the D-algorithm to find tests for all faults is

(1) For each primary input, find a test for the stuck-at-0 fault and a test for the stuck-at-1 fault. In applying the D-algorithm, each fault is propagated to a primary output. If a test is found for an input that is D ($\overline{\mathrm{D}}$), then the same test will detect faults on all other variables that are D ($\overline{\mathrm{D}}$). Hence the D-algorithm finds a series of equivalent faults for each fault on a primary input.

(2) Repeat the process for any remaining variables, starting with any variables

that are “closest” to the primary inputs, until for each fault there is either a test that detects it, or a contradiction occurs (in which case the fault is not testable).

I apply this method for the same circuit for an AND operator.

Let primary input $a1$ be D. I choose to propagate through the AND-gate. Then $y1$ is D if $b1$ is 1. Following that, $z1$ is D if x is 1 (detected during an up-phase). With backward propagation ca and cb are 1. This is a consistent assignment. Also assign 0 to $a0$ and $b0$. Note that if the D of $a1$ is propagated through the OR-gate, then the same test suffices. Raising $a1$ and $b1$ will therefore detect the following stuck-at-0 faults: primary input $a1$; input $a1$ of the OR-gate and the AND-gate; ca; x; $y1$; input x to the C-element with output $z1$; and primary output $z1$. By symmetry, the same test vector also detects these stuck-at-0 faults: primary input $b1$; input $b1$ of the OR-gate and the AND-gate; and cb.

Similarly for $a0$ stuck-at-0 the test raises $a0$ and $b1$, and for $b0$ the test raises $b0$ and $a1$. These two test vectors detect the remaining stuck-at-0 faults.

For the stuck-at-1 faults, let $a1$ be $\overline{\mathrm{D}}$. Then $y1$ is $\overline{\mathrm{D}}$ if $b1$ is 1, and $z1$ is $\overline{\mathrm{D}}$ if x is 1 (during an up-phase). For backwards propagation, ca and cb are both 1. Therefore $a0$ is 1. Furthermore, let $b0$ be 0. Note that, unlike for the stuck-at-0 fault, signal $\overline{\mathrm{D}}$ for $a1$ is not propagated through the OR-gate. Raising $b1$ and $a0$ will detect the following stuck-at-1 faults: primary input $a1$; input $a1$ to the AND-gate; $y1$; and $z1$.

For $b1$ inputs $a1$ and $b0$ have to be raised; and $a0$ and $b0$ signals $a1$ and $b1$. Now the following stuck-at-1 faults remain: all inputs to the OR-gates with outputs ca and cb; x; and input x to both C-elements. Applying the D-algorithm on these variables, starting with the inputs to the OR-gates, it turns out that all these stuck-at-1 faults are detectable during the down-phases of the previous tests. The D-algorithm also comes up with some tests to detect some of these faults during an up-phase. For instance, raising only $a0$ will detect a fault stuck-at-1 for either input of the OR-gate with output *cb*.

With the D-algorithm I have derived the following test, that detects *all* faults in the circuit:

$$a1\uparrow, b1\uparrow; [z1]; a1\downarrow, b1\downarrow; [\neg z1];$$
$$a1\uparrow, b0\uparrow; [z0]; a1\downarrow, b0\downarrow; [\neg z0];$$
$$a0\uparrow, b1\uparrow; [z0]; a0\downarrow, b0\downarrow; [\neg z0].$$

In this section I have shown that the D-algorithm can be adapted for use in delay-insensitive combinational logic circuits. Other non-algebraic methods may be adapted in the same manner. The only additional thing that has to be examined is the fault propagation for state-holding gates.

CHAPTER 5
Design for Testability

[ENIAC] was very reliable and very few hours were lost per week in tracing down the defective tubes. The greatest difficulty came about when on rare occasions two tubes would fail simultaneously. Then the observed symptoms were always highly anomalous.

— Herman H. Goldstine, *The Computer, from Pascal to von Neumann*

1. Introduction

In this chapter I explain how to improve the fault coverage of a circuit with the addition of test points. First, however, I discuss some details of the testing problem that have not been addressed before. In particular, I discuss how to test faults that cause interference in the production rules of a gate. With the strictest definition of testability, most such faults are not testable. It is possible to redesign the gate, with a simple transformation of a guard, so that no fault causes interference.

Another problem is the initialization of the circuit. For most of the proofs on testability of faults, I have assumed that the circuit initializes to a well-defined state. Whether this is the case in practice depends on the technology with which the circuit is implemented, and on how the initialization of the circuit is performed. I discuss the case of CMOS circuits, to which a global reset signal is added.

I show that each fault in a circuit can be made testable with the addition of test points. These can be *control points* or *observation points.* I explain when and where such test points need to be inserted into the circuit.

If the circuit is implemented on a chip, then each test point adds another pad to the design. If the design is pad-limited, then it is advantageous to string the test points together into a queue, so that the overhead for the test points, in terms of the number of pads, is constant. I derive a design for such a test queue, and show that it is fully testable.

2. Non-interference

Consider the production rules for a gate in a delay-insensitive circuit:

$$\begin{cases} B_u & \rightarrow \quad z\uparrow \\ B_d & \rightarrow \quad z\downarrow . \end{cases}$$

For correct operation of the circuit, $\neg B_u \vee \neg B_d$ has to hold invariantly, so that a gate cannot have an up-transition and a down-transition simultaneously. This is the non-interference requirement. It may be violated in the presence of a fault.

If $\neg B_u \vee \neg B_d$ is a tautology, then the expression evaluates to **true** for any value of the inputs. It still evaluates to **true** if there is a stuck-at fault on one of the inputs. A fault can therefore not violate the non-interference requirement. Gates in this category are all combinational gates, the C-element, and most generalizations thereof. For a state-holding element where both guards are conjunctions, it is necessary and sufficient that there be an input that is included as a literal in one guard, and its negation in the other guard.

One of the few gates that can have interfering production rules in the presence of a fault is the set-reset flip-flop:

$$\begin{cases} s & \rightarrow & z\uparrow \\ r & \rightarrow & z\downarrow . \end{cases}$$

If input s is stuck-at-1, the production rules reduce to

$$\begin{cases} \textbf{true} & \rightarrow & z\uparrow \\ r & \rightarrow & z\downarrow . \end{cases}$$

The production rules for z now interfere when r holds. In the implementation of the circuit, there is a short circuit. The value of the output z depends on how strongly the fault pulls z up, as compared to how strongly r pulls z down.

(1) If the fault causes a strong pull-up of z, then z is, effectively, stuck-at-1. Any test that detects z stuck-at-1 also detects s stuck-at-1.

(2) If the fault causes a weaker pull-up than the pull-down of r, the gate effectively changes into an inverter:

$$\begin{cases} \neg r & \rightarrow & z\uparrow \\ r & \rightarrow & z\downarrow . \end{cases}$$

The fault may be tested by bringing the circuit in a state where $\neg s \wedge \neg r \wedge \neg z$, where the faulty circuit will have a premature $z\uparrow$. In addition, $\neg r$ should hold from the initial state until this premature transition.

(3) Otherwise, the fault causes z to have an intermediate value (neither **true** nor **false**) when r holds. How such a state influences the rest of the circuit, and in particular, how this value can be propagated to a primary output so that the fault is detected, is not clear.

The third alternative is the most realistic: the fault is untestable if r holds at some point during a test.

It is always possible to redesign a gate so that a stuck-at fault on an input does not cause the production rules to interfere, by strengthening one or both of the guards. For instance, for the gate with production rules

$$\left\{\begin{array}{lcl} B_u & \rightarrow & z\uparrow \\ B_d & \rightarrow & z\downarrow, \end{array}\right.$$

condition $\neg B_u \vee \neg B_d$ is an invariant. Therefore the guard for $z\uparrow$ can be strengthened with $\neg B_d$, and the guard for $z\downarrow$ can be strengthened with $\neg B_u$. If both these changes are made, the resulting production rules are:

$$\left\{\begin{array}{lcl} B_u \wedge \neg B_d & \rightarrow & z\uparrow \\ B_d \wedge \neg B_u & \rightarrow & z\downarrow . \end{array}\right.$$

The set-reset flip-flop, can be changed into a C-element without changing the functionality of the circuit:

$$\left\{\begin{array}{lcl} s \wedge \neg r & \rightarrow & z\uparrow \\ r \wedge \neg s & \rightarrow & z\downarrow . \end{array}\right.$$

Another case of interference is a fault on a reset variable. This is the subject of the next section.

3. Initializing a Faulty Circuit

In chapter 3 I have stated a number of theorems on the testability of single stuck-at faults in a delay-insensitive circuit. In each I have assumed that, even in the presence of a fault, the circuit can be initialized to a well-defined state, that is, a state where each wire is at a high or a low voltage. The way a circuit is initialized depends on the technology with which it is implemented. I explain initialization for a CMOS process.

The output of a combinational gate depends solely on its inputs. For each input combination, there is a production rule with a **true** guard. This is not always the case for a state-holding element. In particular, the output may not be well-defined as a function of the initial value of its inputs. In that case the state of the state-holding element has to be explicitly initialized.

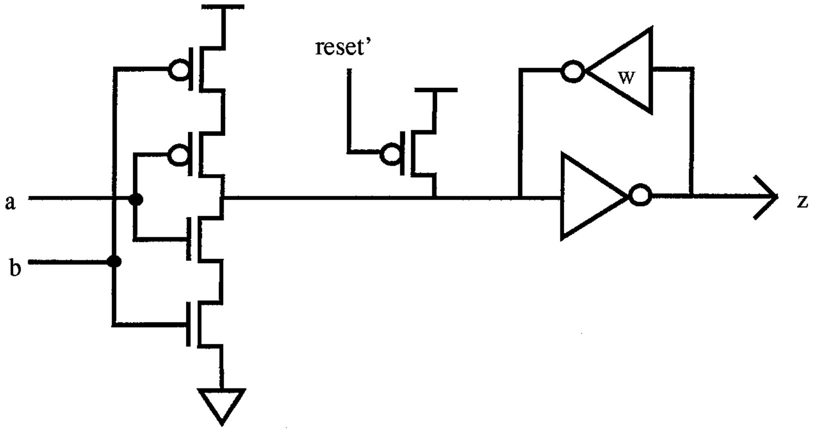

FIGURE 5.1. C-element with added reset transistor

A common way to initialize a state-holding element in a CMOS process is to add a *reset transistor* to the gate. Consider a C-element with inputs a and b and output z. Adding a reset signal to the circuit is equivalent to adding a term in the production rules for z:

$$\left\{\begin{array}{llllll} a & \wedge & b & & \rightarrow & z\uparrow \\ \neg a & \wedge & \neg b & \vee\ \ reset & \rightarrow & z\downarrow. \end{array}\right.$$

A standard implementation of these production rules is in figure 5.1.

To initialize the circuit, there is a transition *reset* ↑, and all primary inputs are set to their initial value. All state-holding elements then have a well-defined output. After a transition *reset* ↓ the circuit is in its initial state. Note that the reset signal does not obey a handshaking protocol. There is no acknowledgement for any transition of *reset*.

I postpone the problem of faults on the *reset* variable. First I investigate how a fault on any other variable may interfere with the initialization of the circuit.

Since the initial value of each variable is **false**, a stuck-at-0 fault cannot cause the circuit to initialize to an incorrect state. Consider a stuck-at-1 fault on input s of a gate with output t. If the fault does not cause the production rule for $t\uparrow$

to evaluate to **true**, then the faulty circuit will initialize correctly (that is, each variable has the correct initial value, except for the fault). Otherwise, the fault causes interfering production rules. See the previous section. There are three ways to approach the problem of interference during initialization due to a fault, two of which involve changing the circuit.

(1) Assume that the signal that initializes t to **false** is stronger than the signal that causes t to have an up-transition as a result of the fault. Each variable is initialized to the right value, after which a $t\uparrow$ transition occurs. Although this requires no alteration of the circuit, and allows the fault analysis to proceed as before, the realism of this assumption is shaky at best.

(2) In case of a production rule that may fire in an initial state, an extra reset transistor is usually included to prevent this firing during the initialization. Similarly, an extra reset transistor can be included to prevent transition $t\uparrow$ from taking place in the presence of the stuck-at-1 fault on input s.

(3) The problem of having $t\uparrow$ as well as $t\downarrow$ fire in the initial state is, that the result (possibly an intermediate voltage) may propagate to other gates, thereby impeding the initialization of other gates. A solution to this problem, then, is to interrupt propagation of variable t. In that case, t is a test point (a control point). See section 4.

Consider the C-element with the following production rules:

$$\left\{\begin{array}{lllllll} s & \wedge & \neg r & & & \rightarrow & t\uparrow \\ \neg s & \wedge & r & \vee & reset & \rightarrow & t\downarrow . \end{array}\right.$$

Let input s be stuck-at-1. Then transition $t\uparrow$ is enabled when the circuit is initialized, as is $t\downarrow$. Adding an extra reset transistors to prevent transition $t\uparrow$

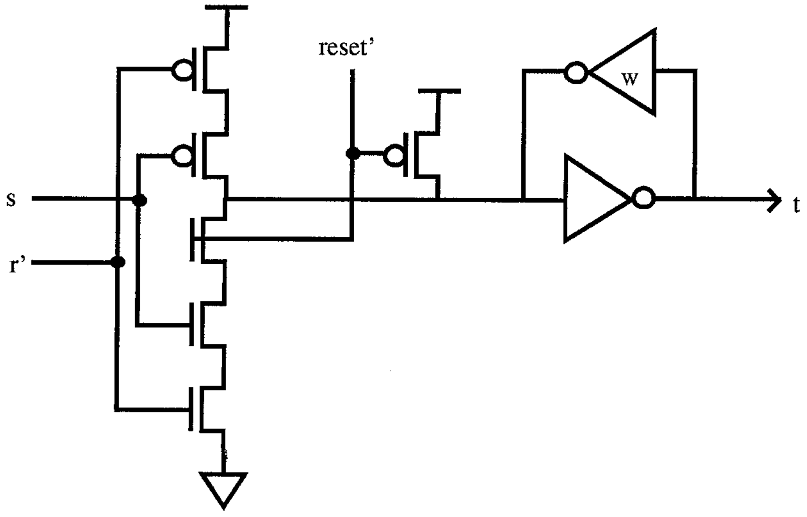

FIGURE 5.2. C-element with additional reset transistor, to insure non-interference

from firing when *reset* is **true** is equivalent to changing the production rules to:

$$\left\{\begin{array}{lllllll} s & \wedge & \neg r & \wedge & \neg reset & \rightarrow & t\uparrow \\ \neg s & \wedge & r & \vee & reset & \rightarrow & t\downarrow . \end{array}\right.$$

An implementation is in figure 5.2

A second possible alteration of the circuit is to have t as a test point. Variable t is transformed into variables *to* and *ti*, where *to* is t as the output of this gate, and *ti* is t as input to other gates. Variable *to* may be observed by the environment (an *observation point*), and *ti* may be set by the environment (a *control point*). See figure 5.3. In case of a fault s stuck-at-1, variable t may not reset to the correct state, but the transition $t\uparrow$ after initialization will be observed directly by the environment. Also, an incorrect value of t is not propagated, since the environment controls the value of variable *ti*.

3.1. Cascaded Resets. If the output of a state-holding element is uniquely determined by the initial values of its inputs, then no reset transistors are needed to initialize it. The state-holding element has a correct initial output because

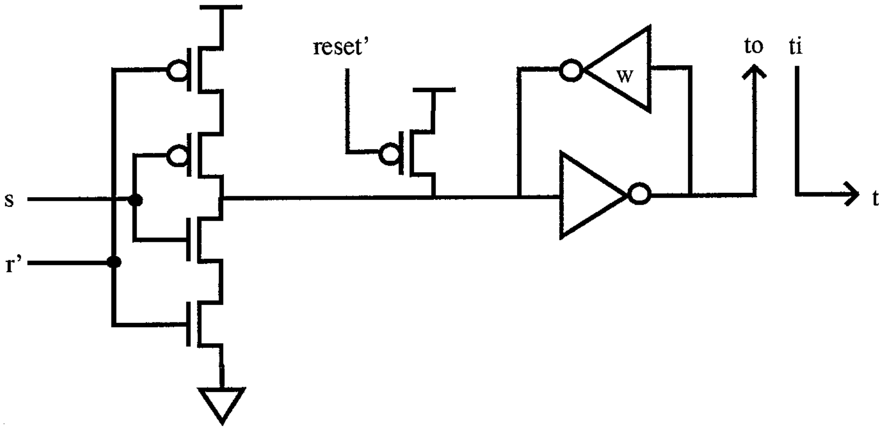

FIGURE 5.3. C-element with output as a test point

its inputs are known to have the correct initial value. Such an implicit way of initializing state-holding elements is known as a *cascaded reset.*

In the presence of a fault, a cascaded reset may not work. If the output of any gate in the cascade is not initialized to the right value, then subsequent gates may also not be initialized to the right value. It is possible that the circuit is not initializeable to a well-defined state. For fault analysis, either the circuit is left as is, and assumptions have to be made about the initial state of a faulty circuit, or reset transistors are added to guarantee a correct initialization. In the latter case, of course, the initialization is no longer a cascaded reset.

A cascaded reset causes unnecessary complications for the analysis of stuck-at faults in a circuit. This is not surprising. Indeed, Abramovici et al. [3] recommend that for any circuit the initialization procedure be as simple as possible, to facilitate fault analysis.

3.2. Faults on Reset Variables. The reset signal is another input for a state-holding element. I investigate if, and how, faults on this input can be tested.

In the absence of a *reset* input to a state-holding element, its output may be at a low, high, or intermediate voltage, after initialization of the circuit. In particular, it is always possible that this output is initially at a low voltage, and is interpreted as **false**. Then there is no need for any reset transistor. Therefore a fault *reset* stuck-at-0 is not testable.

Whether a fault *reset* stuck-at-1 is testable depends on the implementation of the gate. Consider the C-element with one reset transistor (figure 5.1):

$$\left\{\begin{array}{ccccccc} a & \wedge & b & & & \rightarrow & z\uparrow \\ \neg a & \wedge & \neg b & \vee & reset & \rightarrow & z\downarrow . \end{array}\right.$$

With fault *reset* stuck-at-1 the guard for $z\downarrow$ is invariantly **true**. The circuit will reset to the correct state, but as soon as $a \wedge b$ holds, both transitions are enabled. The production rules are interfering, and in the implementation there is a short circuit. As in section 2, either the short circuit is detected, or the $z\uparrow$ transition does not take place, in which cases the fault is detected, or there is a regular transition $z\uparrow$, and the fault is not detected.

If a second reset transistor is added to a state-holding element, then fault *reset* stuck-at-1 is testable. Consider again the circuit of figure 5.2:

$$\left\{\begin{array}{ccccccc} s & \wedge & \neg r & \wedge & \neg reset & \rightarrow & t\uparrow \\ \neg s & \wedge & r & \vee & reset & \rightarrow & t\downarrow . \end{array}\right.$$

With fault *reset* stuck-at-1 t is invariantly **false**. This is testable.

In conclusion, a fault *reset* stuck-at-0 is never testable, and a fault *reset* stuck-at-1 is not testable for the simplest (and most common) implementation.

4. Control and Observation Points

A stimulating stuck-at fault in a delay-insensitive circuit causes a production rule to fire prematurely. In order for this premature firing to be detectable, it is necessary that the premature firing is guaranteed to take place. Furthermore, there has to be a sequence of premature firings, resulting in a premature firing of a primary output or in an inhibited firing (and a halting circuit), so that the environment detects the fault. This is not necessarily possible for all faults in the circuit. If there is a fault in a circuit that is not testable, then one or more *test points* can be added to make the fault testable.

There are two types of test points. A *control point* is a wire in the circuit that the environment can directly set to a certain value. It transforms an internal variable into a primary input, for testing purposes. An *observation point* is a wire in the circuit whose value the environment can directly observe. It transforms an internal variable into a primary output.

For a circuit with test points, there are two modes of operation. The first is the regular mode, when the circuit operates as before, and the test points are ignored. The second is operation in test mode, when the control and observation points are used by the environment to test previously untestable faults.

It is easy to see that with the addition of test points each fault in a circuit can be made testable.

THEOREM 5.1. *Let C be a delay-insensitive circuit, and let G be a gate in C. Then any stuck-at fault on G is testable with the addition of test points.*

Proof: Transform C so that the output of each gate in the circuit is both a control point and an observation point. Then each input of G is either a primary input, or a control point. Therefore the environment can set each input of G to an arbitrary value, and it can hold these values for an unbounded time. The output of G is either a primary output or an observation point. Therefore the environment can always observe the value of the output of G.

In other words, with these test points, the environment can hold each gate in an arbitrary state for an unbounded time, and it can observe this state. If the gate has a stuck-at fault, the environment can detect it. □

Fortunately, for most circuits far fewer test points are needed to make the circuit fully testable. For instance, in the control part of the asynchronous microprocessor [51, 50], without the memory unit, there are 174 gates, and three test points are needed to make it fully testable. I investigate where control and observation points are necessary.

4.1. Control Points. Let C be a delay-insensitive circuit, and consider a stuck-at fault in C that may cause a premature firing of variable t, say $t\uparrow$. There is a state in the handshaking expansion where this premature firing will take place. Consider a test, T, that will bring the faulty circuit in a state where the premature firing may take place. If the premature firing is unstable, a control point has to be added to the circuit.

Consider the state where premature firing $t\uparrow$ takes place. Since the firing is unstable, the guard of $t\uparrow$ is evaluated to **true**, and then may be evaluated to **false** before transition $t\uparrow$ is acknowledged. There are transitions of one or more of the inputs that falsify the guard of $t\uparrow$, say $x_0\uparrow, x_1\uparrow, \ldots, x_{n-1}\uparrow$. If these transitions are delayed by a sufficient amount of time, then the firing of $t\uparrow$ will occur. If $x_0, x_1, \ldots, x_{n-1}$ are control points, then the environment can hold these variables to **false** until the premature firing of $t\uparrow$ occurs.

It is possible to have another set of control points. If y is an input to the gate with output x_0, and the transition $x_0\uparrow$ is an acknowledgement for transition $y\uparrow$, then y can be a control point instead of x_0. If the environment keeps y **false**, through the control point, then $y\uparrow$ and $x_0\uparrow$ will not occur, and the premature firing of $t\uparrow$ will take place.

This process of substituting control points may continue to the inputs of y, and so forth, as long as the circuit is guaranteed to reach the state where the premature firing of $t\uparrow$ occurs. Note that the purpose of a control point is to delay, by a sufficient amount of time, a transition in the circuit. It is therefore not strictly necessary that the environment be able to set the value of the control point directly.

4.2. Observation Points. Once a premature firing has occurred, it has to be propagated to a primary output. Either the circuit halts (a transition of a primary output does not occur), or there is a premature firing of a primary output. If neither case is guaranteed to occur, it is necessary to add an observation point. If the fault causes a premature firing $u\uparrow$ in circuit C, then u can be an observation point. If the premature firing of $u\uparrow$ causes a stable premature firing of another variable in C, then that variable can be an observation point in lieu of u.

5. Example: Microprocessor Control

The instruction fetch circuit for the asynchronous microprocessor [51, 50] consists of nine gates, and has one untestable fault. It occurs in a gate with output g, and production rules

$$\left\{\begin{array}{ccc} ao \wedge u \wedge bi \wedge \neg ci \wedge \neg gi & \rightarrow & g\uparrow \\ \neg u \wedge \neg eo & \rightarrow & g\downarrow . \end{array}\right.$$

Variables ao and eo are primary outputs, bi, ci, and gi are primary inputs, and u and g are internal variables.

Consider fault $bi[g]$ stuck-at-1. It is not inhibiting, and can only be tested with a premature firing. The fault will cause a premature firing of $g\uparrow$ in a state where

$$ao \wedge u \wedge \neg bi \wedge \neg ci \wedge \neg gi \wedge \neg g.$$

However, in any state where this condition holds, $u\downarrow$ may fire, as the production rule is

$$(ao \wedge \neg bi) \vee eo \rightarrow u\downarrow .$$

This means that any premature firing of $g\uparrow$ is unstable. By transforming u into a control point, the environment can delay transition $u\downarrow$, so that the firing of $g\uparrow$ is stable. It is not necessary to have control points for the other inputs of g: ao

remains **true** as long as u is **true**, and ci and gi are primary inputs, which the environment controls directly.

The stable firing of $g\uparrow$ results in a stable firing of a primary output, which the environment will detect. It is not necessary to have an observation point for this fault.

6. Design to Minimize Number of Test Points

I have described where to insert test points in a given circuit, to make the circuit fully testable. The next problem is how to design a circuit so as to minimize the number of test points needed. Ideally, one would like to design circuits that need no test points.

For many control-type programs, it is possible to derive a circuit that is fully testable without test points. For sequential synchronous circuits, it is not known how to design a circuit with a minimal number of test points [3]. The problem is not solved for delay-insensitive circuits either. The following are merely heuristics that minimize the number of test points and facilitate testing.

Maximize the number of inhibiting faults. It is more difficult to test for a prematurely firing production rule than to test for an inhibiting production rule. Faults that are both stimulating and inhibiting are usually tested by having an inhibited firing, so that the circuit under test halts.

To maximize the number of inhibiting faults, one has to change production rules, so that a fault on an input of a gate that is only stimulating also becomes inhibiting. Consider a gate with input s and output t. If literal s is included in the production rules of t by syntactic derivation or strengthening, and $\neg s$ is not, then fault s stuck-at-1 is only stimulating. If $\neg s$ can be included by strengthening one of the production rules, then the fault is also inhibiting.

Such a transformation does not necessarily mean that the fault is testable with an inhibited firing. If the state in the handshaking expansion with the inhibited firing can be reached before the state with the premature firing, then the fault is testable. Otherwise, the premature firing should be analyzed.

It is not always possible to make each stimulating fault also inhibiting. Consider the D-element, which has handshaking expansion

$$*[[li]; ro\uparrow; [ri]; u\uparrow; [u]; ro\downarrow; [\neg ri]; lo\uparrow; [\neg li]; u\downarrow; [\neg u]; lo\downarrow].$$

Since $[li]$ occurs immediately before $ro\uparrow$, literal li has to be included in the production rule for $ro\uparrow$ by syntactic derivation. Then fault $li[ro]$ stuck-at-1 is stimulating. It would also be inhibiting if a production rule for ro can be strengthened with $\neg li$. This is not possible, however, since all transitions of ro occur when li is **true**.

For a circuit that has a fault that is stimulating but not inhibiting, there necessarily is an isochronic fork. If $s[t]$ stuck-at-1 is stimulating, but not inhibiting, then $\neg s$ is not included in the production rules for t by either syntactic derivation or strengthening. Therefore no transition $s\downarrow$ is acknowledged with a transition of t, which implies that there is an isochronic fork. Specifically, the branch of s that is input to the gate with output t is an isochronic branch.

Maximizing the number of inhibiting faults generally means to reduce the number of isochronic forks in a circuit. The cost of adding terms to production rules is an increase in the number of transistors, and a degradation of performance.

There is one type of stimulating fault that is generally simple to test. Consider a handshaking expansion containing, in part, the sequence

$$\ldots; s\uparrow; [s]; t\uparrow; \ldots$$

Then s is included in the production rule for $t\uparrow$:

$$s \wedge B_0 \vee B_1 \rightarrow t\uparrow .$$

Fault $s[t]$ stuck-at-1 makes $t\uparrow$ fire prematurely in a state before the $[s]$-action. If s is a primary input, and t a primary output, and if this state is reachable from the initial state, then fault $s[t]$ stuck-at-1 is testable. In general, $t\uparrow$ fires prematurely in the last state before $s\uparrow$ where the environment waits for a primary input transition; the premature firing causes a sequence of transitions leading to the transition of the primary output immediately following $t\uparrow$.

This analysis leads to two rules for designing circuits with high fault coverage.

- From the handshaking expansion, produce production rules by syntactic derivation.
- If a production rule for variable t is strengthened with a literal, then a production rule for t also has to be strengthened with the negation of that literal.

Minimize concurrency. For a stimulating fault, a state has to be found where the fault causes a premature firing of a production rule. The more concurrency is introduced by reshuffling a handshaking expansion, the more states there are, and the more difficult it is to find a test vector.

Concurrency may also increase the length of a complete test. An example occurs in the instruction fetch process of the asynchronous microprocessor [51]. Part of the handshaking expansion is

$$\ldots\ ; go\uparrow; [gi \wedge \neg ei]; eo\uparrow; \ldots\ ,$$

resulting in production rule

$$go \wedge u \wedge gi \wedge \neg ei \rightarrow eo\uparrow\ .$$

Faults $ei[eo]$ stuck-at-0 and $gi[eo]$ stuck-at-1 are both only stimulating. The former causes a premature firing $eo\uparrow$ in a state where

$$\neg eo \wedge go \wedge u \wedge gi \wedge ei,$$

whereas the latter causes a premature firing of $eo\uparrow$ when

$$\neg eo \wedge go \wedge u \wedge \neg gi \wedge \neg ei.$$

The test sequences for these two faults are incompatible, that is, the faults cannot be tested in the same cycle.

The reason that these faults have different test sequences is that transition $eo\uparrow$ occurs only after two concurrent primary input transitions ($gi\uparrow$ and $ei\downarrow$) have been observed. If the transitions of the primary inputs were sequenced, then these faults could be tested with one test vector.

The requirement to minimize concurrency to facilitate testing is in obvious conflict with the aim to improve performance. Introducing concurrency is an elementary way to reduce the number of transitions in a critical path, and thereby to speed up the circuit. Performance, not testability, ought to be the prime measuring stick in circuit design.

Reduce interference in if-statements. The reason that a fault in the instruction fetch circuit is untestable is that it causes both guards of an if-statement to be evaluated to **true**. Consider part of the if-statement:

$$\begin{array}{lrcl} [& bi & \rightarrow & [\neg gi \wedge \neg ci];\, g\uparrow;\ldots \\ | & \neg bi & \rightarrow & u\downarrow \\ ]. & & & \end{array}$$

This results in a production rule

$$ao \wedge u \wedge bi \wedge \neg ci \wedge \neg gi \rightarrow g\uparrow .$$

Fault $bi[g]$ stuck-at-1 is only stimulating. To test the fault, bi should be **false**. When the circuit reaches the if-statement, the gate with output u evaluates bi to **false**, and, because of the fault, the gate with output g evaluates bi to **true**.

Therefore both $g\uparrow$ and $u\downarrow$ are enabled, but the firing of $g\uparrow$ is not stable, since u is a term in the guard of $g\uparrow$.

The firing would be stable if either u is not a term in the guard for $g\uparrow$, or if there is another wait-action between the guard evaluation of the if-statement and transition $u\downarrow$. Indeed, fault $bi[u]$ stuck-at-0 (which is also only stimulating) is testable since g is not in the production rules for u, and since there are two wait-actions between the guard evaluation for the if-statement and $g\uparrow$.

Keep interaction of state variables simple. There are several conditions for a stimulating fault to be testable. It has to cause a premature firing in a stable state, and the result has to be propagated to a primary output. If there are no internal variables, then a premature firing in a controllable state is always detected. With the introduction of state variables, the number of transient states increases. In addition, there are now possible premature firings (namely, of the state variables) that are not directly observed by the environment.

It is advantageous to keep the interconnection of state variables simple. For instance, each state variable should be in the guard of few other state variables. It is also advisable not to have circular dependencies among state variables (that is, if u is input to the gate with output v, then v should not be input to the gate with output u). Finally, a state variable should not have two or more transitions between consecutive controllable states in a handshaking expansion.

Consider, for example, a different implementation of the buffer (example 2.2 of chapter 2). See figure 5.4. It has four internal variables, u, v, x, and y, and

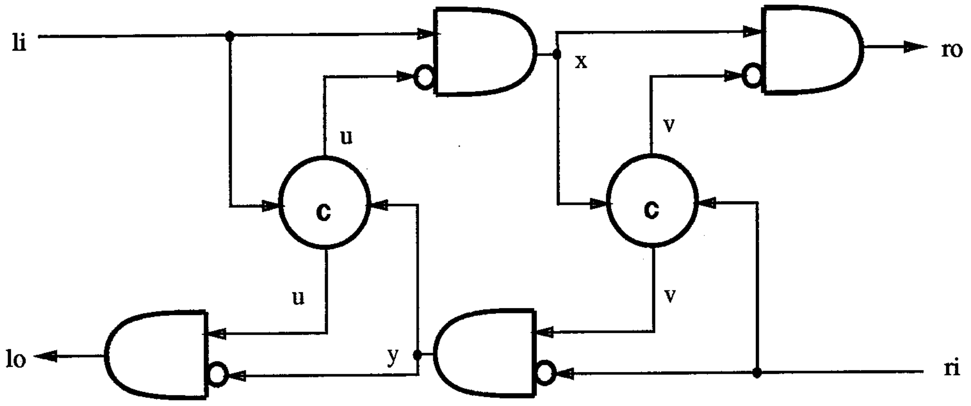

FIGURE 5.4. Implementation of buffer with two D-elements

handshaking expansion

```
$*[$  $[li]; x\uparrow; [x]; ro\uparrow;$
      $ri\uparrow; v\uparrow; [v]; ro\downarrow;$
      $[\neg ri]; y\uparrow; [y]; u\uparrow; [u]; x\downarrow; [\neg x]; v\downarrow; [\neg v]; y\downarrow; [\neg y]; lo\uparrow;$
      $[\neg li]; u\downarrow; [\neg u]; lo\downarrow$
$].$
```

This circuit is really two D-elements concatenated. It is redundant in the sense that there is an implementation of the handshaking expansion with just half the number of gates. Since there is no redundant gate, and no redundant input, the circuit itself is *not* redundant.

Between the $[\neg ri]$ action and the $[\neg li]$ action there are two transitions of state variable y. Hence y is only **true** in transient states. Now consider fault $y[lo]$ stuck-at-0. It is only stimulating, and causes a premature firing of $lo\uparrow$ only when

$$\neg lo \wedge u \wedge y$$

holds. This condition only holds in transient states, hence the fault is not testable.

In short, to design circuits with high fault coverage, there should be few transitions of internal variables between consecutive controllable states. A state variable should have at most one transition between controllable states. The number of inhibiting faults should be maximized, by having a variable and its negation included in a pair of production rules by syntactic derivation or by strengthening. And if, for a faulty circuit, two guards of an if-statement are evaluated to **true**, the resulting actions should not cause instability.

7. Test Circuitry

The addition of control and observation points to a circuit requires more connections between the circuit and its environment. For a circuit that is implemented as a chip, this means more pads. As most designs are pad-limited, it may be prudent to reduce the number of pads that are needed for test points. This is done with the addition of test circuitry.

The test circuitry itself should be fully testable for stuck-at faults. Moreover, the size of the test circuitry should be small compared to the remainder of the circuit, since it serves no purpose once the circuit has been tested, and since any additional active area on a chip increases the probability of a fault. Adding test circuitry degrades the performance of the original design. For an observation point the test circuitry merely adds a fork to a wire, increasing the capacitive load on that node. For a control point the value of the node set by the environment should replace the value set by the remainder of the circuit. This requires the addition of a stage of logic to the circuit, which degrades its performance.

The most common approach to the design of test circuitry is to string all control and observation points together into a queue or a stack, so that values can be read and written by the environment serially. The main advantages are, that the number of pads required for such a scheme is minimal, and independent of the number of test points, and that there are compact implementations for queue and stack elements.

The description of test points for delay-insensitive circuits in terms of control and observation points is no different from the description for synchronous circuits. Consequently, any design for test circuitry for synchronous circuits may be used also for delay-insensitive circuits. It is, perhaps, somewhat of a heresy to suggest the use of these clocked implementations to test delay-insensitive circuits. However, there are many compact and well-understood designs for clocked queue stages for use as test circuitry (most are smaller than their delay-insensitive counterparts) [25]. In addition, there is not a problem of clock skew. The clock that controls the queue can be relatively slow.

I now describe a delay-insensitive implementation of a test queue element, as well as some shortcuts to obtain a reasonably small design.

Consider a circuit C that is a delay-insensitive implementation of program P. A fault analysis yields a number of control and observation points. The problem is to change the program and the circuit to incorporate these test points into a queue. Circuit C should not change beyond the addition of the test points.

The new circuit now has four distinct modes of operation:

- Regular operation. Circuit C is used without using the test points;
- Shifting in values. The environment shifts values into the queue, to be used to set the control points;
- Test operation. The circuit operates using the values of the control points that have been set by the environment. The resulting values are then read from the observation points;
- Shifting out values. The environment shifts values out of the queue, that have been read from the observation points.

The second and fourth modes of operation may overlap, as one can shift values into and out of a queue simultaneously.

Let the queue have N stages, and assume that each test point is both a control

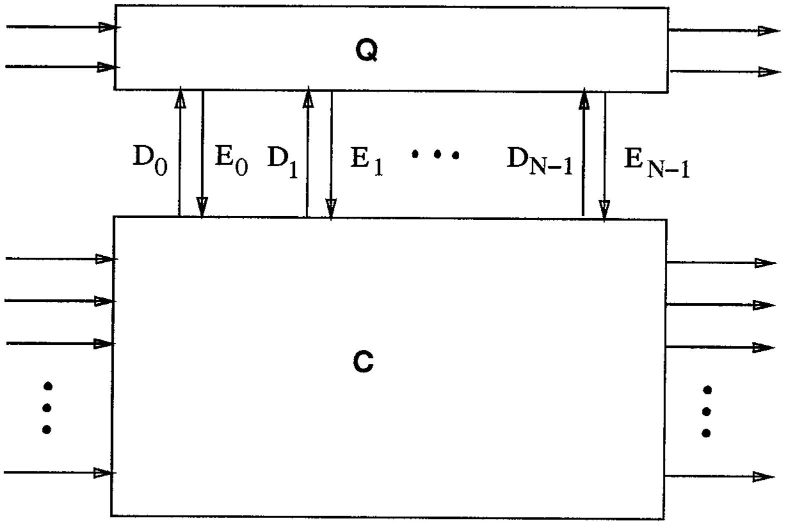

FIGURE 5.5. Schematic of circuit with test queue

point and an observation point. The queue is connected to the environment with channel L_0, with which values are entered into the queue, and with channel R_{N-1}, with which values are taken from the queue. For each queue element there are two connections to the original circuit, C. Channel E_i $(0 \leq i < N)$ is used to set a control point with the value of the ith queue element, and channel D_i $(0 \leq i < N)$ is used to take the value from an observation point. See figure 5.5.

A program for the complete circuit is

$$
\begin{array}{lll}
*[[& \neg test \rightarrow & P \| Q \\
\ \ | & test \rightarrow & \text{“}P_{test}\text{”} \\
]]. & &
\end{array}
$$

As long as the circuit is in regular operation, it executes program P. The queue may operate in parallel, as there is no connection between the two parts of the circuit when $test$ is **false**. In test operation, program P is executed, with the provision that values are taken from the queue to the control points, and sent to the queue from the observation points. Let $x[i]$ $(0 \leq i < N)$ be the value of the

ith element of the queue. The program for the queue itself, Q, is

```
*[ $i := N-1$;
   $*[i \geq 0 \rightarrow L_0?x[i]; i := i-1]$;
   $E_0!x[0], E_1!x[1], \ldots, E_{N-1}!x[N-1]$;
   $D_0?x[0], D_1?x[1], \ldots, D_{N-1}?x[N-1]$;
   $i := N-1$;
   $*[i \geq 0 \rightarrow R_{N-1}!x[i]; i := i-1]$
].
```

The environment (including circuit C) of the queue has program

```
*[ $i := 0$;
   $*[i < N \rightarrow L_0!x; i := i+1]$;
   $E_0?ctrl[0], E_1?ctrl[1], \ldots, E_{N-1}?ctrl[N-1]$;
   $(*)$
   $D_0!obs[0], D_1!obs[1], \ldots, D_{N-1}!obs[N-1]$;
   $i := 0$;
   $*[i < N \rightarrow R_{N-1}?x; i := i+1]$
],
```

where $ctrl[i]$ is the value of the ith control point, and $obs[i]$ the value of the ith observation point. The original circuit, C, operates in state $(*)$.

Next the queue is distributed over N separate processes. Each queue element has a channel L to receive a value from its left neighbor, and a channel R to send a value to its right neighbor. The left channel of the first element and the right channel of the last are connected to the environment. Each element has channels E and D that connect to the remainder of the circuit, as before. The jth queue element has program

```
*[  $i := j;$
    $*[i < N - 1 \rightarrow L?x; R!x; i := i + 1$
    $|i = N - 1 \rightarrow L?x; E!x; D?x; R!x$
    $];$
    $i := 0;$
    $*[i < j \rightarrow L?x; R!x; i := i + 1]$
].
```

The implementation of the L and R channels is standard: two wires for the dual-rail encoded variable that is transmitted, and one acknowledgement wire. The implementation of channels D and E, which provide the interface between the original circuit and the queue, is more complicated.

Channel D takes the value of an observation point and sends it to the queue. The queue may be able to acknowledge receipt of such a value, but there is nothing in circuit C that can process such an acknowledgement. Therefore there is no acknowledgement for values transmitted on channel D. The value that is sent over channel D tests whether there is a premature firing in circuit C as a result of a fault. In case there is such a premature firing, the queue has to wait long enough before it receives the value on channel D. This is inherently not delay-insensitive; it is not possible to encapsulate such a transmission in a delay-insensitive communication protocol. Therefore the value is not dual-rail encoded, and the queue can observe directly the value of the observation point.

For channel E, which sends a value from the queue to a control point, there is also no acknowledgement, as circuit C does not generate such an acknowledgement. Since the control point is a single wire, the value sent on channel E is also not dual-rail encoded.

The handshaking expansion for the above program is now straightforward. The next problem is to find a reshuffling that will lead to a small queue element. The largest part of the test circuitry will be the queue itself. It is important to minimize this part of the design.

For the queue, I choose the implementation of chapter 3, which consists of two C-elements and two OR-gates. Given this circuit for the one-bit buffer process, I choose the following reshuffling of the handshaking expansion for the jth queue process:

```
$*[\ i := j$
    $*[i < N-1 \rightarrow$   $[l1 \rightarrow lo\uparrow; [\neg ri]; r1\uparrow; [\neg l1]; lo\downarrow; [ri]; r1\downarrow$
                              $|l2 \rightarrow lo\uparrow; [\neg ri]; r2\uparrow; [\neg l2]; lo\downarrow; [ri]; r2\downarrow$
                              $];$
                              $i := i + 1$
    $|i = N-1 \rightarrow$    $[l1 \vee l2]; [l1 \rightarrow e\uparrow\ |l2 \rightarrow e\downarrow];$
                              $[d \rightarrow lo\uparrow; [\neg ri]; r1\uparrow; [\neg l1 \wedge \neg l2]; lo\downarrow; [ri]; r1\downarrow$
                              $|\neg d \rightarrow lo\uparrow; [\neg ri]; r2\uparrow; [\neg l1 \wedge \neg l2]; lo\downarrow; [ri]; r2\downarrow$
                              $]$
    $];$
    $i := 0;$
    $*[i < j \rightarrow$     $[l1 \rightarrow lo\uparrow; [\neg ri]; r1\uparrow; [\neg l1]; lo\downarrow; [ri]; r1\downarrow$
                              $|l2 \rightarrow lo\uparrow; [\neg ri]; r2\uparrow; [\neg l2]; lo\downarrow; [ri]; r2\downarrow$
                              $];$
                              $i := i + 1$
    $]].$
```

In this handshaking expansion, signal e is the implementation of channel E from the queue to a control point, and signal d is the implementation of channel D from an observation point to the queue.

The main difficulty at this point is the implementation of i. It is impractical to have a counter for each queue element. The only use of variable i is to indicate whether $i = N - 1$ holds. Since this condition is true for each of the queue elements at the same moment, I introduce a global signal, s, that indicates whether $i \neq N - 1$. The program for the jth queue element is now simpler:

```
$*[[\ \ s \rightarrow L?x; R!x$
  $|\ \ \neg s \rightarrow L?x; E!x; D?x; R!x$
$]]$.
```

7.1. Queue Element for an Observation Point. For an observation point there is no communication on the E channel, only a communication on the D channel. A straightforward transformation of the above program into a handshaking expansion yields:

```
$*[[\ \ s \wedge l1 \rightarrow lo\uparrow; [\neg ri]; r1\uparrow; [\neg l1]; lo\downarrow; [ri]; r1\downarrow$
  $|\ \ s \wedge l2 \rightarrow lo\uparrow; [\neg ri]; r2\uparrow; [\neg l2]; lo\downarrow; [ri]; r2\downarrow$
  $|\ \ \neg s \wedge (l1 \vee l2) \rightarrow$
      $[\ \ \ d \rightarrow lo\uparrow; [\neg ri]; r1\uparrow; [\neg l1 \wedge \neg l2]; lo\downarrow; [ri]; r1\downarrow$
      $|\ \ \neg d \rightarrow lo\uparrow; [\neg ri]; r2\uparrow; [\neg l1 \wedge \neg l2]; lo\downarrow; [ri]; r2\downarrow$
      $]$
$]]$.
```

With state variables similar to the ones for the one-bit buffer, the resulting circuit is a one-bit buffer for which the production rules for $y1$ and $y2$ have been changed to include signals s and d. The disadvantage of this particular implementation is that it has an untestable stuck-at fault.

To obtain a fully testable queue element I restrict the specification of the queue. With a queue element for an observation point, a value is taken from the circuit, and sent – using the queue – to the environment. The original value of the queue element is overwritten with the value of the observation point. I now require that the original value of the queue element be *identical* to the value of the observation point for the correct circuit. If the value of the observation point is expected to be **true** (**false**), then a one (a zero) should be loaded by the environment into the corresponding queue element. The queue element acts as a comparator.

In the handshaking expansion this requirement translates into an additional wait-action. For the correct circuit this wait-action is superfluous, as the value of the queue element is the same as the value of the observation point. For a faulty circuit (where the observation point has the wrong value) I show that, because of this wait-action, the fault is detected by the environment.

The handshaking expansion is

$$\begin{array}{l} *[[\ \ l1 \rightarrow lo\uparrow; [\neg ri]; r1\uparrow; [\neg l1]; lo\downarrow; [ri]; [s \vee \neg s \wedge d]; r1\downarrow \\ \ \ |\ \ l2 \rightarrow lo\uparrow; [\neg ri]; r2\uparrow; [\neg l2]; lo\downarrow; [ri]; [s \vee \neg s \wedge \neg d]; r2\downarrow \\ ]]. \end{array}$$

I introduce, as before, state variables $y1$, $y2$, and yo, yielding:

$$\begin{array}{llll} *[[& l1 & \rightarrow & y1\uparrow; [y1]; lo\uparrow; [\neg ri]; r1\uparrow; yo\uparrow; [yo \wedge \neg l1]; \\ & & & y1\downarrow; [\neg y1]; lo\downarrow; [ri]; [s \vee \neg s \wedge d]; r1\downarrow; yo\downarrow; [\neg yo] \\ \ | & l2 & \rightarrow & y2\uparrow; [y2]; lo\uparrow; [\neg ri]; r2\uparrow; yo\uparrow; [yo \wedge \neg l2]; \\ & & & y2\downarrow; [\neg y2]; lo\downarrow; [ri]; [s \vee \neg s \wedge \neg d]; r2\downarrow; yo\downarrow; [\neg yo] \\ ]]. & & & \end{array}$$

Signal s is controlled by the environment. That means that now the environment has to count to $N-1$. Assuming that the environment fills the queue with a series of ones, the program for the environment is

```
$*[$  $i := 0; s\uparrow;$
    $*[i < N - 1 \rightarrow l_0 1\uparrow; [l_0 o]; l_0 1\downarrow; [\neg l_0 o]; i := i + 1]$
    $l_0 1\uparrow; [r_{N-1} 1]; test\uparrow;$
    $[ctrl[0] \wedge ctrl[1] \wedge \ldots \wedge ctrl[N-1]];$
    $(*)$
    $obs[0]\uparrow, obs[1]\uparrow, \ldots, obs[N-1]\uparrow;$
    $s\downarrow; s\uparrow; test\downarrow;$
    $r_{N-1} i\uparrow; [\neg r_{N-1} 1 \wedge l_0 o];$
    $l_0 1\downarrow; r_{N-1} i\downarrow; [r_{N-1} 1 \wedge \neg l_0 o];$
    $i := 0;$
    $*[i < N - 1 \rightarrow r_{N-1} i\uparrow; [\neg r_{N-1} 1]; r_{N-1} i\downarrow; [r_{N-1} 1]]$
$]$.
```

The remainder of the circuit, C, operates in state $(*)$, following which the values of the observation points are sent to the queue. In this case, all values are **true** ('$d_i\uparrow$').

There is no acknowledgement signal for signal s in the handshaking expansion of the queue element. It is possible to generate such an acknowledgement for each queue element, and then to merge all acknowledgement signals into one global acknowledgement signal, by means of a completion tree. This, however, is costly in area. The size of the queue is linear in the number of test points, whereas the size of a completion tree grows faster than linear.

In the following, I do not implement an acknowledgement for signal s. Instead, by judiciously changing the value of s in states where no transition occurs if s is either **true** or **false**, the size of the queue element circuit is significantly reduced. Note that the function of signal s in the queue is to determine whether the value of the previous queue element (encoded into $l1$ and $l2$) or the value of the observation point (d) has to be sent on to the next queue element. Since the circuit has to wait long enough for the value of d to change (in case of a premature firing), I assume that within this period of time any transition of s has also be received by the queue element.

Circuit C operates in test mode when the environment of the test queue is in state ($*$). At that point, each queue element is in the following state:

$$l1 \wedge \neg l2 \wedge \neg lo \wedge \neg y1 \wedge \neg y2 \wedge yo \wedge r1 \wedge \neg r2 \wedge \neg ri,$$

again assuming that the environment has sent a series of ones. This corresponds in the handshaking expansion for the queue element to the $[ri]$-action in the first (if a one was last received) or second (if a zero was last received) case of the if-statement. Therefore, if the environment is in state ($*$), there is no transition until ri holds, and the value of s can safely be changed ($s\downarrow$).

Likewise, when the environment changes s to **true**, there is no possible transition in the queue element. I assume, of course, that the value of d does not change between transitions $s\downarrow$ and $s\uparrow$.

It is now a rather straightforward task to derive a production rule set for the above handshaking expansion. Variables $y1$ and $y2$ are again C-elements.

$$\left\{\begin{array}{lcl} l1 \wedge \neg ri & \rightarrow & y1\uparrow \\ \neg l1 \wedge ri & \rightarrow & y1\downarrow \end{array}\right.$$

$$\left\{\begin{array}{lcl} l2 \wedge \neg ri & \rightarrow & y2\uparrow \\ \neg l2 \wedge ri & \rightarrow & y2\downarrow . \end{array}\right.$$

Variables lo and yo are, again, OR gates.

$$\left\{\begin{array}{lcl} r1 \vee r2 & \rightarrow & yo\uparrow \\ \neg r1 \wedge \neg r2 & \rightarrow & yo\downarrow \end{array}\right.$$

$$\begin{cases} y1 \vee y2 & \rightarrow \quad lo\uparrow \\ \neg y1 \wedge \neg y2 & \rightarrow \quad lo\downarrow . \end{cases}$$

For $r1$ and $r2$ the production rules are somewhat complicated. Because of the requirement that the value of the queue element be the same as the value of the observation point, the extra wait-action in the handshaking expansion is superfluous, and therefore no additional production rule is necessary. For the faulty circuit, however, the value of the queue element and the value of the observation point are different (due to a prematurely firing production rule), and this result has to be propagated. In a departure from standard design methods, I add a third production rule to the production rules for both $r1$ and $r2$ that fires only if the value of the queue element is different from the value of the observation point.

$$\begin{cases} s \wedge y1 \wedge \neg ri & \rightarrow \quad r1\uparrow \\ s \wedge \neg y1 \wedge ri & \rightarrow \quad r1\downarrow \\ \neg s \wedge \neg d & \rightarrow \quad r1\downarrow \end{cases}$$

$$\begin{cases} s \wedge y2 \wedge \neg ri & \rightarrow \quad r2\uparrow \\ s \wedge \neg y2 \wedge ri & \rightarrow \quad r2\downarrow \\ \neg s \wedge d & \rightarrow \quad r2\downarrow . \end{cases}$$

For a correct circuit, only the first two production rules of $r1$ and $r2$ will fire. If the observation point has the wrong value, due to a prematurely firing production rule in circuit C, then the third production rule fires. This third production rule is, in a way, an explicit encoding of a premature firing. If the environment operates the queue as before, then it will observe a premature firing of $l_0o\uparrow$ after transition $s\uparrow$ for the faulty circuit.

The full implementation of the queue element for an observation point is in figure 5.6.

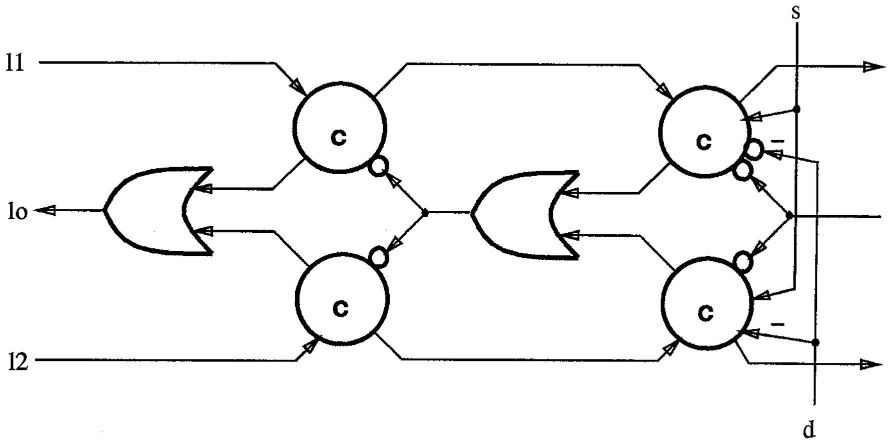

FIGURE 5.6. Circuit of a queue element for an observation point

7.2. Queue Element for a Control Point. What remains is the implementation of signal e, which sets a control point. When the environment is in state $(*)$, each queue element has to set the corresponding control point to the last received value. That is, if $l1$ holds, e should be **true**, and if $l2$ holds, e should be **false**.

In circuit C, consider a variable, u say, that is a control point. There is a duplicate of u, u' say, such that u' is identical to u when the circuit is in regular operation (that is, when $\neg test$ holds), and identical to e when the circuit is in test mode (when $test$ holds). A queue element for a control point can be made by taking the original one-bit queue element, and adding a gate that has output u'.

The production rules for u' are:

$$\left\{\begin{array}{rcl} u \wedge \neg test & \rightarrow & u'\uparrow \\ \neg u \wedge \neg test & \rightarrow & u'\downarrow \\ e \wedge test & \rightarrow & u'\uparrow \\ \neg e \wedge test & \rightarrow & u'\downarrow . \end{array}\right.$$

These production rules are akin to a double-throw switch in a synchronous circuit,

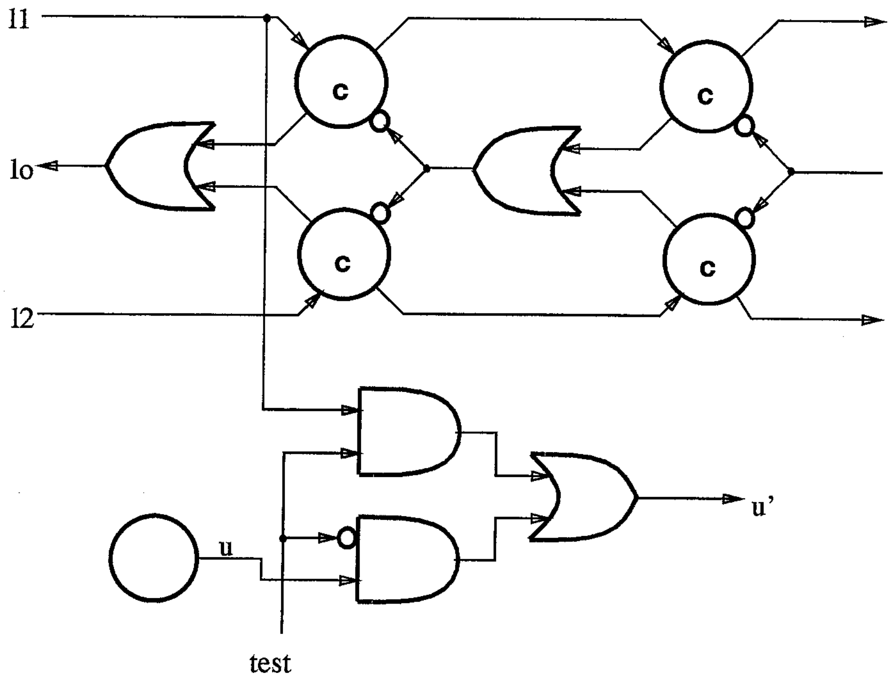

FIGURE 5.7. Circuit of a queue element for a control point for which the production rules would be:

$$\begin{cases} (u \wedge \neg test) \vee (e \wedge test) & \rightarrow \quad u'\uparrow \\ (\neg u \vee test) \wedge (\neg e \wedge \neg test) & \rightarrow \quad u'\downarrow . \end{cases}$$

Like signal s, signal *test* is global. Its main function is to determine, for each control point, when a value should be taken from the queue element. It is possible to have each queue element acknowledge the transitions of *test*, and then to merge this set of acknowledgements into a single acknowledgement by means of a completion tree. Again, such a completion tree grows faster than linear in the size of the queue. I implement *test* without acknowledgements.

The circuit for the control point is in figure 5.7.

8. Testability of the Test Circuitry

In order for the test circuitry to be useful at all, it has to be fully testable. In this section I analyze the testability of the queue element.

The original one-bit buffer, with four C-elements and two OR-gates, is fully testable for stuck-at faults by sending it a zero bit and a one bit. A queue of such buffers is fully testable by sending to it, and receiving from it, an alternation of zeros and ones. Consequently, most of the queue element for testing is testable in the same way. For the two C-elements (with outputs $y1$ and $y2$), and the two OR-gates (with outputs yo and lo) this is trivially true.

Next I show that a fault on signal d is testable. This also shows that a premature firing on an observation point is detectable. Suppose d is stuck-at-0. Fill the queue with values, such that $r1$ is **true**. During transition $s\downarrow$ the queue element for the observation point is in state

$$l1 \wedge \neg l2 \wedge \neg lo \wedge \neg y1 \wedge \neg y2 \wedge yo \wedge r1 \wedge \neg r2 \wedge \neg ri.$$

At that point, production rule $\neg s \wedge \neg d \rightarrow r1\downarrow$ fires [1]. This results in transitions $yo\downarrow, y1\uparrow$, and $lo\uparrow$. Then, after transition $s\uparrow$, this transition $lo\uparrow$ propagates to the left neighbor of this queue element, and so forth, until the environment observes a transition $l_0o\uparrow$. This is a premature transition, as for the correct circuit $l_0o\uparrow$ only takes place *after* the environment sets $r_{N-1}i$ to **true**.

By symmetry, fault d stuck-at-1 is also testable. Faults $d[r1]$ stuck-at-0 and $d[r2]$ stuck-at-1 are testable by the same reasoning.

For fault $d[r1]$ stuck-at-1, enter values into the queue such that $r1$ is **true**, and bring circuit C in a state where observation point d is **false** (for instance, the initial state). In the correct circuit there will be a transition $l_0o\uparrow$ directly after

[1] Note that the firing of $\neg s \wedge d \rightarrow r2\downarrow$ is vacuous.

$s\uparrow$, whereas in the faulty circuit this does not occur. Similarly $d[r2]$ stuck-at-0 is testable.

If s is stuck-at-1, then the queue never observes a premature firing. Again enter values into the queue such that $r1$ is **true**, and bring circuit C into a state where d is **false**. The transition $l_0o\uparrow$ that occurs in the correct circuit after $s\uparrow$ does not occur in the faulty circuit. If s is stuck-at-0, then the guards for $r1\uparrow$ and $r2\uparrow$ are both **false**, which is obviously detectable by using the queue.

For the queue element for the observation point, some faults on the inputs of $r1$ and $r2$ remain to be examined. The production rules for $r1$ are

$$\left\{\begin{array}{rcl} s \wedge y1 \wedge \neg ri & \rightarrow & r1\uparrow \\ s \wedge \neg y1 \wedge ri & \rightarrow & r1\downarrow \\ \neg s \wedge \neg d & \rightarrow & r1\downarrow . \end{array}\right.$$

I have shown that faults $d[r1]$ stuck-at-0 and stuck-at-1 are both testable, as are faults $s[ro]$ stuck-at-0 and stuck-at-1. If there is a stuck-at fault on $y1[r1]$ or $ri[r1]$, then either $r1\uparrow$ or $r1\downarrow$ never fires. This is detectable by entering and retrieving a one into and out of this queue element.

Any stuck-at fault on the inputs of $r2$ is testable in a similar manner.

Finally, I consider faults on the gate with output u', for the control point. The production rules are:

$$\left\{\begin{array}{rcl} u \wedge \neg test & \rightarrow & u'\uparrow \\ \neg u \wedge \neg test & \rightarrow & u'\downarrow \\ e \wedge test & \rightarrow & u'\uparrow \\ \neg e \wedge test & \rightarrow & u'\downarrow . \end{array}\right.$$

If u is stuck-at-0 or stuck-at-1, then that fault is already testable, possibly with another test point. This gate does not alter the testability of u, and any test for u can also be used for u'.

If *test* is stuck-at-0, then u' always has the same value as u, and it is not possible to set the value of u' with the queue element. One way to test this fault is to use the circuit in test mode, with e **false**. Then in the faulty circuit a transition $u'\uparrow$ in the handshaking expansion occurs, but in the correct circuit this transition is inhibited. Therefore the fault causes a premature firing of $u'\uparrow$. By continuing to execute the handshaking expansion, the fault is detected.

If *test* is stuck-at-1, then the converse holds. Use the circuit in regular operation, and let e be **false**. Then the faulty circuit has u' **false** invariantly. Therefore $u'\uparrow$ is inhibited for the faulty circuit, and this causes the circuit to halt.

Testing faults on e is somewhat more complicated. In circuit C, fault u stuck-at-0 is inhibiting. Note that if a string of ones, or an alternation of zeros and ones, is sent to and received from the queue, then signal e will alternate between **true** and **false**.

Fault u stuck-at-0 in circuit C is inhibiting, therefore there is a state (q, say) in the handshaking expansion where this fault causes an inhibited firing. For fault e stuck-at-0, fill the queue such that e is **false**, and operate circuit C (which has no fault) until it reaches the last controllable state before state q. Set *test* to **true**, and operate circuit C until it reaches state q. Then use the queue such that there is a transition $e\uparrow$. For circuit C this transition $e\uparrow$ causes a transition $u'\uparrow$, and the circuit operates as before. If e is stuck-at-0, however, there will not be transitions $e\uparrow$ and $u'\uparrow$, causing the circuit to halt. Fault e stuck-at-0 is then testable. Similarly for fault e stuck-at-1.

I have now shown that all faults in the test circuitry are testable. The test queue also allows the environment to test all previously untestable faults in the original circuit (C). The resulting circuit is therefore fully testable for stuck-at faults.

CHAPTER 6
Test Generation and Other Topics

There is some feeling that the prospects of LSI are limited. When thousands of circuits are packed onto a chip, there is a tendency for chip size to increase and for yield problems to increase accordingly.
— Ernest Braun and Stuart MacDonald, *Revolution in Miniature*

1. Introduction

If each stuck-at fault in a circuit is testable, then it is possible to test the circuit by concatenating the tests for each individual fault. This obviously results in a long test. To find the smallest test that will detect all faults is an NP-complete problem. I focus on ways to find a reasonably small test sequence that detects all faults. This process is guided by the handshaking expansion of which the circuit is an implementation. That this is possible is a major advantage of testing circuits that are synthesized from high-level programs.

I also investigate the fault location problem for the stuck-at fault model. Finally, I discuss the applicability of fault detection methods for stuck-at faults in delay-insensitive circuits to a more detailed fault model, one that considers stuck-open and stuck-on faults for transistors.

2. Heuristics for Test Generation

In chapter 3 I have explained how to derive a test vector for a particular stuck-at fault in a delay-insensitive circuit. The goal of testing is usually not to test for a particular fault, but to test for all possible faults in the circuit at once. Finding the shortest test that detects all stuck-at faults in a circuit is an NP-complete problem. In this section I give some heuristics that yield reasonably minimal test sequences.

The method in this section is suited for "control" type circuits. These are circuits for programs with only synchronization channels, and channels with few encoded bits. Of course, the distinction between "control" circuits and "datapath" circuits (with wide channels and large blocks of combinational logic) is somewhat arbitrary.

Most stuck-at faults in delay-insensitive circuits are inhibiting faults. Many inhibiting faults are testable simply by executing the program that the circuit implements. In the presence of such a fault, the circuit will halt. It is easy to derive a test from the handshaking expansion. For a program, P, that is an infinite repetition:

$$P \equiv *[S]$$

the test should be such that the circuit executes program part S at least once. If S contains an if-statement, then each alternative of the if-statement should be executed at least once. For instance, the one-bit queue element (chapter 3) has handshaking expansion

$$\begin{array}{lll} *[[& l1 \rightarrow & lo\uparrow; [\neg ri]; r1\uparrow; [\neg l1]; lo\downarrow; [ri]; r1\downarrow \\ \ \ | & l2 \rightarrow & lo\uparrow; [\neg ri]; r2\uparrow; [\neg l2]; lo\downarrow; [ri]; r2\downarrow \\ ]]. & & \end{array}$$

The if-statement has two branches. The first is taken when $l1$ holds, the test sequence generated from the handshaking expansion is:

$$l1\uparrow; [lo \wedge r1]; l1\downarrow; [\neg lo]; ri\uparrow; [\neg r1].$$

The other branch is taken if $l2$ holds. A test sequence, from the initial state, is:

$$l2\uparrow; [lo \wedge r2]; l2\downarrow; [\neg lo]; ri\uparrow; [\neg r2].$$

The circuit implementing this handshaking expansion has 42 different stuck-at faults. The first test sequence detects 31 faults (74 percent), and the second sequence the remaining eleven faults. In this example, the test derived from the handshaking expansion detects all faults. In general, there remain undetected faults after this step.

There is a test for each fault. The problem is to find a minimal number of tests that detect all remaining faults. If, for two different faults, the test that detects the one fault is a prefix of the test to detect the other, then the longer test will detect both faults. The two tests are *compatible.*

The test generation problem can then be transformed into a graph problem. The nodes of the graph are the faults in the circuit, and there is a vertex between nodes if their tests are compatible. Finding a minimal set of test vectors for the circuit is equivalent to finding a smallest clique cover in the graph. The clique cover problem is, of course, NP-complete [28].

Solving such a clique cover problem does not necessarily yield the smallest test sequence; there may be more than one test that detects a certain fault. A reshuffling of the actions in a test for a fault is often also a test for the same fault. Assume, for now, that there is no arbitration in the circuit. Then the order in which the environment changes the primary inputs to the circuit does not alter the final state of the circuit, as long as the handshaking protocol is obeyed. In the queue circuit above, for instance, there is no difference in the final state of the circuit whether $ri\uparrow$ takes place before or after $l1\downarrow$. In the case of a test for a fault, the actions of the test may be reshuffled as long as for the reshuffled sequence the fault does not cause a premature firing in a transient state.

Two faults are then compatible if a test for one fault can be reshuffled into a test for both faults. It is, alas, not true any more in general that, for a set of faults that are pairwise compatible, there is a single test sequence for all faults in the set.

The following heuristics seem to work well in finding a minimal test set.

Partition the tests according to the sequence of states in the handshaking expansion that each test causes. This is only useful for a handshaking expansion with one or more if-statements. If, for two tests, different branches of an if-statement are taken, then the two tests are not compatible.

Sort the tests in a set of the partition by length of the test sequence. Consider the longest test, and find the next longest compatible test. Continue until there are no more compatible tests; repeat the process with the remaining tests.

I demonstrate this with the faults in the queue circuit. In chapter 3 I derived a test for each individual stuck-at fault. This resulted in six different test sequences:

$$l1\uparrow; [lo \wedge r1]$$
$$l1\uparrow; [lo \wedge r1]; l1\downarrow; [\neg lo]$$
$$l1\uparrow; [lo \wedge r1]; l1\downarrow; [\neg lo]; ri\uparrow; [\neg ri]$$
$$l2\uparrow; [lo \wedge r2]$$
$$l2\uparrow; [lo \wedge r2]; l2\downarrow; [\neg lo]$$
$$l2\uparrow; [lo \wedge r2]; l2\downarrow; [\neg lo]; ri\uparrow; [\neg ri].$$

Since the handshaking expansion has one if-statement with two cases, the tests are partitioned into two sets; the first three test sequences correspond to the first case of the if-statement, the second three to the second. Next, take the longest sequence in the first set,

$$l1\uparrow; [lo \wedge r1]; l1\downarrow; [\neg lo]; ri\uparrow; [\neg ri],$$

and check if the actions can be reshuffled such that the next-longest sequence is a prefix of it. In this case, obviously, no reshuffling needs to be done, either for the second sequence or for the shortest one. All faults in the queue element are therefore testable with the following two test sequences:

$$l1\uparrow; [lo \wedge r1]; l1\downarrow; [\neg lo]; ri\uparrow; [\neg ri]$$
$$l2\uparrow; [lo \wedge r2]; l2\downarrow; [\neg lo]; ri\uparrow; [\neg ri].$$

Deriving a test sequence in this manner, the circuit has to be reinitialized between the different test sequences. In this case, however, the two sequences can be concatenated; faults that are detected in the second sequence are also detected if the first sequence is prepended.

The derivation above to find a test to test all faults in the queue process is typical for other circuits. The most naïve way to derive a test, namely executing the handshaking expansion as it is written, yields a high fault coverage. This is because the majority of faults is inhibiting. In addition, an inhibiting fault that is also stimulating will only cause a premature firing under specific conditions. For most such faults, these conditions do not occur during this simple test.

The difficult part of the test generation process is to find a small test sequence for the remaining faults, most of which are only stimulating. This sequence may be as long as, or longer than, the sequence that tests all other faults. It has, of course, been noted before, that to alter a test with high fault coverage to 100 percent fault coverage may require much computation, and may yield long tests [3].

I have analyzed the control part of the asynchronous microprocessor [51], using the methods described above. I derived a test from the handshaking expansion for each individual process, and combined these tests to form a test for the complete circuit. These results were verified with the COSMOS switch-level simulator with fault model. For each process I divide the faults into inhibiting faults, faults that are stimulating and not inhibiting, and untestable faults.

process	inhibiting faults	stimulating faults	not testable	total faults	instructions in test
FETCH	78	20	2	100	5
PC	87	13	0	100	6
EXEC	262	29	1	292	12
ALU	92	10	0	102	4
Total	519	72	3	594	27

To test the untestable faults, three test points are necessary. I have implemented a test queue for these test points. The size of the test queue is 146 transistors; the total number of transistors in the circuit is 1290. Since there are so few test points in this circuit, it would have been more economical to connect each test point directly to a pad, and not implement a test queue.

The total number of instructions to test the control circuit is the sum of the number of instructions for each process. It is possible that there is a smaller test, if the circuit is analyzed in its entirety. The number of actions per instruction (that is, changing the value of a primary input) is 13 for jump and branch, 17 for load and store, 23 for load and store offset and for load address and store program counter, and 27 for an ALU instruction.

3. Fault Location

Testing theories include fault detection and fault location. The tests in the previous section apply only to the fault detection problem. Because of the large number of equivalent faults, they only give a rough idea of the location of a particular fault.

To locate a fault, the tests have to be refined, and test points have to be added to the circuit. It is always possible to locate every stuck-at fault by adding enough test points to the circuit. The proof is the same as the proof that each fault is testable. But whereas for fault detection few test points are needed for most circuits, for fault location the large number of test points needed is prohibitively expensive. I explain this using a small circuit, the D-element.

The D-element has handshaking expansion, including state variable u,

$$*[[li]; ro\uparrow; [ri]; u\uparrow; [u]; ro\downarrow; [\neg ri]; lo\uparrow; [\neg li]; u\downarrow; [\neg u]; lo\downarrow].$$

The production-rule set is

$$\left\{\begin{array}{rcrcl} li & \wedge & ri & \rightarrow & u\uparrow \\ \neg li & \wedge & \neg ri & \rightarrow & u\downarrow \end{array}\right.$$

$$\left\{\begin{array}{rcrcl} li & \wedge & \neg u & \rightarrow & ro\uparrow \\ \neg li & \vee & u & \rightarrow & ro\downarrow \end{array}\right.$$

$$\left\{\begin{array}{rcrcl} \neg ri & \wedge & u & \rightarrow & lo\uparrow \\ ri & \vee & \neg u & \rightarrow & lo\downarrow . \end{array}\right.$$

There are two primary inputs, two primary outputs, one state variable, and three forks in the circuit. Hence there are eleven fault locations, and twenty-two possible stuck-at faults. Each fault induces a faulty circuit, but these twenty-two circuits are not all distinct.

Consider, for example, the case where input li to the gate with output u is stuck-at-0. As a result of this fault, state variable u will be **false** invariantly. Therefore lo is always **false**, and ro has the same value as li. The production rules for this circuit are

$$\left\{\begin{array}{rcl} li & \rightarrow & ro\uparrow \\ \neg li & \rightarrow & ro\downarrow . \end{array}\right.$$

The fault is detected when testing the circuit, since lo does not have any up-transition, and since ro has a transition whenever li does.

Faults ri stuck-at-0, $ri[u]$ stuck-at-0, and u stuck-at-0 reduce the D-element to the same circuit. Without any test points, these four faults are equivalent for *any* test.

Even with test points, it is not possible to distinguish between a stuck-at-0 fault on an input of a C-element and a stuck-at-0 fault on the output of a C-element. The difference between ri stuck-at-0 and the other faults can be tested if u is added as a control point. In that case, the AND gate with output lo reduces to a wire with input u for fault ri stuck-at-0, but remains the same for the other three faults.

Another example of equivalent faults is ro stuck-at-0, $li[ro]$ stuck-at-0, and $u[ro]$ stuck-at-1. For these faults ro is **false**, and the D-element reduces to

$$\left\{\begin{array}{ccccc} li & \wedge & ri & \rightarrow & u\uparrow \\ \neg li & \wedge & \neg ri & \rightarrow & u\downarrow \end{array}\right.$$

$$\left\{\begin{array}{ccccc} \neg ri & \wedge & u & \rightarrow & lo\uparrow \\ ri & \vee & \neg u & \rightarrow & lo\downarrow . \end{array}\right.$$

In order to be able to distinguish between these three faults, u again has to be made into a control point. The result is that both inputs of this AND gate can be set independently, so that the gate can be fully tested.

In this small example, stuck-at-0 faults on inputs and outputs of the C-element are indistinguishable, even with test points, and some faults on the AND gates are distinguishable only if these gates can be set and observed separately from the remainder of the circuit. The only variable that the environment does not control is the state variable, u. For fault location it is necessary that u be a control point.

These observations generalize to larger circuits. Because of the sequential nature of delay-insensitive circuits, there are many equivalent faults. These faults can, in general, only be located if each gate in the circuit can be set and observed independently of the other gates. This requires a test point for each internal variable of the circuit.

The test points cannot be strung together into a test queue. Fault location in the test queue requires extra test points, which in turn requires test points, etc. The only alternative is to have a pad per test point in the circuit. Even for medium-sized circuits, this is prohibitively expensive in terms of the number of pads needed.

In case the extra expense for fault location is considered justified, there remains a fundamental impediment to fault location. These tests can be used to locate a stuck-at fault. Actual faults in a circuit are generally *not* stuck-at faults. Even if one can pinpoint the location of the stuck-at fault, that may not be where the actual fault has occurred. Moreover, to know the location of a fault is of limited importance. Most faults on a chip cannot be corrected.

In conclusion, to solve the fault location problem requires many additional test points, and these test points cannot be part of a queue design. The usefulness of stuck-at fault location is limited. Moreover, tests for fault *detection* are useful for a limited form of fault location. Depending on at what point in the test a fault is observed, it is possible to locate a region of the circuit where the fault has occurred.

4. Stuck-open and Stuck-on Faults

So far, I have focused on the stuck-at model as a fault model to derive tests for delay-insensitive circuits. The fault model is not accurate in describing actual faults in a circuit, however.

A refinement of the stuck-at fault model is one that has *stuck-open* and *stuck-on*

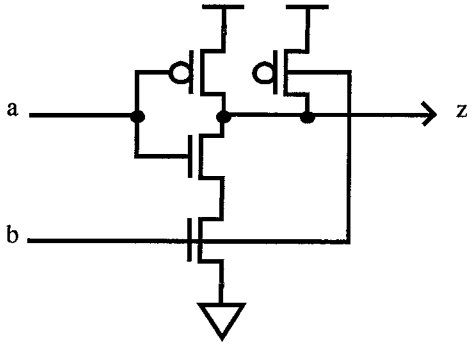

FIGURE 6.1. CMOS implementation of a NAND gate

faults. For a stuck-open fault, the source and drain of the transistor are permanently disconnected; for a stuck-on fault, the source and drain are permanently connected. This fault model is technology-dependent. It is possible, to some extent, to analyze these faults using the production rule set of the circuit. I discuss the application of this fault model for CMOS implementations, where there is a strong relation between a pair of production rules for a gate, and the transistor-implementation of that gate.

4.1. Stuck-open Faults. Consider a gate, with inputs a and b, and output z. The production rules for this gate are

$$\begin{cases} \neg a \;\vee\; \neg b \;\to\; z\uparrow \\ \;\;a \;\wedge\; \;\;b \;\to\; z\downarrow . \end{cases}$$

The CMOS implementation of this gate has two p-transistors in parallel, connected to two n-transistors in series (see figure 6.1).

If there is an stuck-open fault on the p-transistor with gate a, then the path to z from power through this transistor is cut. Such a fault is equivalent to changing the production rules to

$$\begin{cases} \qquad\quad \neg b \;\to\; z\uparrow \\ a \;\wedge\; \;\;b \;\to\; z\downarrow . \end{cases}$$

This is not a fault a stuck-at-1; the fault is only inhibiting, not stimulating. If $\neg a$ is not a redundant term, then the fault causes an inhibited firing, and the circuit halts.

Likewise, if the n-transistor with gate a is stuck open, then the guard for $z\downarrow$ is **false**. A test that detects fault z stuck-at-1 will also detect this stuck-open fault.

There is one complication that cannot be captured in production-rule form. A stuck-open fault transforms the NAND-gate into a state-holding element. In other words, there is not always a path to output z from either power or ground. If, during a test, the circuit is in a state where signal z is not driven, then the test should be executed fast enough that there is no significant decay of signal z.

For stuck-open faults on combinational gates with a like implementation the analysis is similar. A stuck-open fault is equivalent to transforming a term in a production rule guard to **false**. It is testable if that term is not redundant. The fault changes the gate into a dynamic state-holding gate, which may result in an intermediate voltage on the output.

There are several different CMOS implementations for state-holding elements. The production rules of a gate may not reflect the actual implementation of that gate. Consider a C-element with inputs a and b and output z:

$$\left\{\begin{array}{ccccc} a & \wedge & b & \rightarrow & z\uparrow \\ \neg a & \wedge & \neg b & \rightarrow & z\downarrow . \end{array}\right.$$

A direct implementation is a series of two p-transistors connected to a series of two n-transistors, followed by an inverter, as in figure 6.2. If any one of these six transistors has a stuck-open fault, then either all transitions $z\uparrow$ or all transitions $z\downarrow$ are inhibited. Such faults are testable as before.

This implementation of the C-element is dynamic. In general more transistors are added to the gate, to insure that the output is always driven. Since these added transistors are not necessary for the functionality of the circuit, any stuck-open

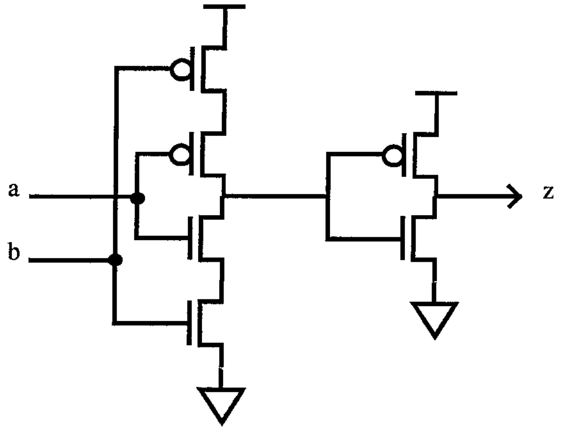

FIGURE 6.2. Dynamic CMOS implementation of a C-element

fault on these transistors is not testable. A common addition is a weak transistor to 'static-ize' the dynamic nodes [47], as in figure 6.3. The signal generated by the weak inverter is weaker than any other in the gate, so that the gate can switch, and the output is always driven.

The analysis of an stuck-open fault on one of the original six transistors is the same as before. Since the weak inverter is added, however, there is no danger that the output may have an intermediate value at any time. If there is a stuck-open fault on one of the two transistors of the weak inverter, then the output is either not always strongly driven to power, or not always strongly driven to ground. The only way to test for such a fault is to wait for the output to have an intermediate value. How to test for such an occurrence is beyond the scope of this thesis. The stuck-open fault is untestable.

In conclusion, most stuck-open faults in a CMOS implementation of a delay-insensitive circuit correspond to a literal in a guard of a production rule that evaluates to false. The fault is inhibiting, but not stimulating; it is testable if that literal is not redundant in the guard. Not testable are stuck-open faults on the part of a state-holding element that makes the output of the gate static.

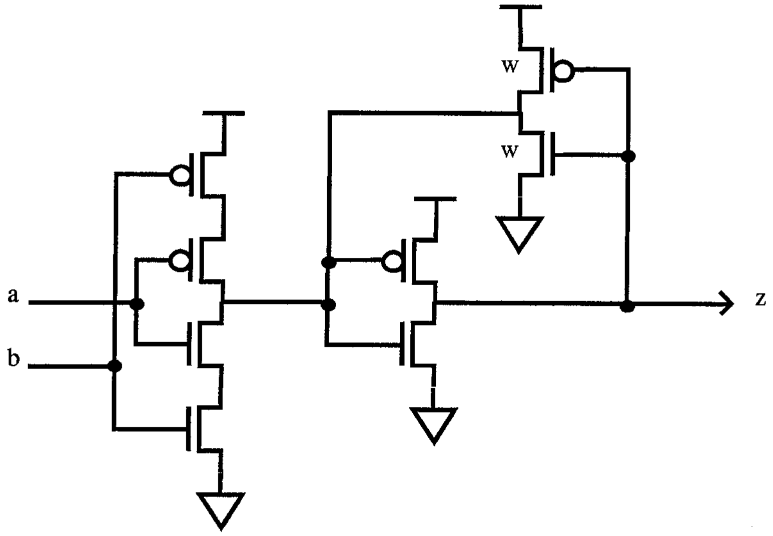

FIGURE 6.3. CMOS implementation of a C-element, with weak inverter

For synchronous combinational logic circuits, the analysis of a stuck-open fault is complicated, as such a fault transforms a combinational circuit into a sequential one. Delay-insensitive circuits, on the contrary, are already sequential, and these inhibiting faults are simple to analyze, as it is easy to detect that a circuit has halted.

4.2. Stuck-on Faults. If most stuck-open faults are inhibiting but not stimulating, it should not come as a surprise that most stuck-on faults are stimulating but not inhibiting. These stimulating faults may cause interference in a delay-insensitive circuit, which is difficult to analyze.

4.2.1. Combinational Gates

Consider again the NAND gate of figure 6.1. Let the p-transistor with gate a have a stuck-on fault. Then the output z is permanently connected to power. The production rules for this faulty gate are

$$\begin{cases} \mathbf{true} & \to & z\!\uparrow \\ a \wedge b & \to & z\!\downarrow . \end{cases}$$

To test the fault either a or b must be **true**. Then both guards of the production rules evaluate to **true**; there is interference. The fault is detected if one of the gates, to which z is connected, interprets the resulting signal z as **true** instead of **false**, and if this incorrect value is propagated to a primary output. With my definition of testability, the fault is not testable.

The same is true for all stuck-on faults in combinational gates. The stuck-on fault can only be tested in a state where the faulty transistor should not conduct, but does. If the correct circuit in that state has the output of this gate connected to ground, then for the faulty circuit the output is also connected to power, and vice versa. Hence a stuck-on fault in a combinational gate can only be tested by having interference.

4.2.2. State-holding Elements

For most state-holding elements, interference does not occur in any dynamic implementation. Consider again the dynamic implementation of the C-element, in figure 6.2. Let the p-transistor with gate a have a stuck-on fault. Then the production rules of the gate become

$$\begin{cases} b & \to & z\!\uparrow \\ \neg a \vee \neg b & \to & z\!\downarrow . \end{cases}$$

These are the production rules for an asymmetric C-element. Testing this fault is equivalent to testing fault a stuck-at-0 as a stimulating fault. The stuck-on fault is testable if $z\!\uparrow$ fires prematurely, that is, in a state where

$$\neg z \wedge \neg a \wedge b,$$

and if this premature firing can be propagated to a primary output.

The analysis for the n-transistor with gate *a* is similar. It can be tested the same way that fault *a* stuck-at-0 is tested as a stimulating fault.

The remaining two transistors of the dynamic implementation of the C-element form an inverter stage. Since an inverter is a combinational gate, any stuck-on fault on these transistors can only be tested in a state where there is interference. These faults are not testable with the current definitions of testability. The same is true, *a fortiori*, for stuck-on faults on the additional two transistors of the static implementation of the C-element.

In conclusion, most stuck-on faults are stimulating faults. With the current definition of testability, no stuck-on fault in a combinational gate is testable. Some stuck-on faults on state-holding elements are testable if the corresponding stuck-at fault is testable as a stimulating fault.

CHAPTER 7
Conclusions

I have developed a method to test faults in a delay-insensitive circuit, using the single stuck-at fault model. A fault can be inhibiting, when it strengthens the guard of a production rule, or stimulating, when it weakens the guard of a production rule. A fault on a primary input, or on the output of a gate is always both inhibiting and stimulating.

I have proven that for each inhibiting fault, there is a state in the high-level specification of the circuit where the fault causes an inhibited firing, and that for each stimulating fault, there is a state where the fault causes a premature firing. A fault that is only inhibiting is always testable, since the state with the inhibited firing is reachable from the initial state; the circuit subsequently halts. For a fault that is also stimulating, if a state with an inhibited firing is reachable before a state with a premature firing, then the fault is detectable, since the circuit halts during a test. Most faults in a circuit are testable this way. Otherwise, the fault causes a premature firing. If this firing is stable, and results in a premature firing of a primary output, or in the circuit halting, then the fault is testable.

There are untestable faults. Such faults can be made testable with the addition of test points. If a premature firing is unstable, then a control point is needed; if the premature firing is not propagated to a primary output, then an observation point is needed. These test points can be strung together in a test queue. I have shown a reasonably small test queue that is almost delay-insensitive. It is inherently impossible to have a test queue that is strictly delay-insensitive. Since the formulation in terms of control and observation points is the same as for synchronous circuits, any queue used in synchronous systems can also be used here.

For delay-insensitive circuits that implement combinational functions, I reduce the problem of finding test vectors to the problem of finding test vectors for a monotone synchronous combinational logic circuit. I have also shown that any algebraic method to find test vectors for synchronous combinational logic can be extended to cover these delay-insensitive circuits.

The problem of finding the smallest test sequence to test all stuck-at faults in a circuit is NP-complete. I have shown a few heuristics that yield reasonably small test sequences. In order to do this, I use the high-level specification of the circuit. This significantly reduces the search space. Many test methods use a bottom-up approach, where a gate in a circuit is set so that the fault causes a malfunctioning, which is then propagated forward to the primary outputs, and backward to the primary inputs to get a test vector. For any VLSI delay-insensitive circuit, this approach is prohibitively expensive. Unlike synchronous circuits, the interconnection of gates in a delay-insensitive circuit is highly complex, and a large number of gates is state-holding. I believe that any efficient method to test delay-insensitive circuits must rely heavily on the high-level specification and on the synthesis method to obtain good tests.

I have applied the testing theory to the control part of the asynchronous microprocessor. Of 594 single stuck-at faults in the circuit, 3 are not testable. The test queue that makes these faults testable takes up about 11 percent of the total number of transistors in the circuit. A test that detects all faults is 27 microprocessor instructions long. This test was derived from the high-level specification; it does not seem worthwhile to attempt to reduce the size of the test.

During the synthesis of a circuit, testability should be taken into account. In particular, the number of inhibiting faults should be maximized. If a variable, s say, is an input of a gate, then both s and $\neg s$ should be included in the production rules of the gate, by either syntactic derivation or by strengthening. Also, most untestable faults tend to occur for state variables that have transitions concurrently with transitions of primary outputs. One can either add a test point for such faults, or alter the specification, such that each state variable transition is directly acknowledged with a transition of a primary output. The latter solution is usually preferred, both in terms of transistor count and of performance.

Finally, the stuck-at fault model is a good fault model to analyze testability of these synthesized delay-insensitive circuits, since the analysis can be done at the gate level. Future work includes refining this fault model, to a technology-dependent fault model, perhaps tailored specifically to the analysis of delay-insensitive circuits. A transistor-based fault model with stuck-open and stuck-on faults is promising, since most faults in this fault model can still be described with production rules. Application of this fault model requires that the definition of testability be refined, too; some faults introduce temporary and permanent shorts, as well as nodes that are not driven for an arbitrary time. Application of a refined fault model will not significantly alter the tests I have derived in this thesis; rather, it will cause more defects to be detected.

APPENDIX A
Algorithms

This appendix is an overview of various algorithms to test delay-insensitive circuits.

1. Finding a Test Vector for a Fault

Given a handshaking expansion and a production rule set, find a test vector that will test a given fault in the circuit.

Assume, for now, that the fault is on the input of a gate. Call the input s, the gate G, and its output t. Assume the production rules for gate G are

$$\left\{\begin{array}{rcccl} s \wedge B_0 & \vee & B_1 & \rightarrow & t\uparrow \\ \neg s \wedge C_0 & \vee & C_1 & \rightarrow & t\downarrow, \end{array}\right.$$

where B_i and C_i are arbitrary boolean expressions not including s or $\neg s$. Consider fault s stuck-at-1. The other cases are similar.

(1) Find the conditions for s to cause an inhibited firing (*INH*) and a premature firing (*PRE*):

$$INH \equiv \neg s \wedge C_0 \wedge \neg C_1 \wedge t$$
$$PRE \equiv \neg s \wedge B_0 \wedge \neg B_1 \wedge \neg t.$$

For a fault that is only inhibiting, $PRE \equiv$ **false**, and for a fault that is only stimulating, $INH \equiv$ **false**.

(2) For each state of the handshaking expansion, determine the value of the inputs and the output of gate G. In each state, the value of the primary outputs and the internal variables is known; the value of some primary inputs may not be known (call this value "X" or don't-care). Since the environment controls the primary inputs, either **true** or **false** may be substituted for any "X" value.

(3) To test the fault as an inhibiting fault, find a state in the handshaking expansion where *INH* holds, reachable from the initial state without an intermediate state where *PRE* holds. The state where *INH* holds must be a state where $t\downarrow$ fires. A value "X" in such a state may be either **true** or **false**, whichever satisfies *INH*.

Then check if this state is reachable through a series of states for which $\neg PRE$ holds (from the handshaking expansion). If there exists such a series of states, then the fault is testable as an inhibiting fault; it causes the circuit to halt eventually. The test vector is implied by the sequence of states to reach the inhibiting state.

(4) To test the fault as a stimulating fault, find a state in the handshaking expansion where *PRE* holds, such that the premature firing is stable, and is propagated to a primary output. To find a state where *PRE* holds, we only have to consider states in the handshaking expansion where a variable in B_0 or B_1 fires; other states, including states where s or t fire, are irrelevant. If necessary, substitute either **true** or **false** for any value "X".

Next, similar to the previous case, check if there is a sequence of states from the initial state to the state where *PRE* holds, such that $\neg PRE$ holds in all intermediate states. This is the initial part of the test vector.

Check if the premature firing of $t\uparrow$ is stable. If not, find another state where *PRE* holds. If it is controllable, and t is not a primary output, then the premature firing has to be propagated. The problem, then, is to determine if fault t stuck-at-1 is testable, starting in this state (rather than the initial state), with the condition that there is no transition $t\downarrow$. This condition is a simple addition to conditions *PRE* and *INH* for fault t stuck-at-1. Apply the algorithm again for this fault with these new conditions.

(5) If there is no state where the fault is detected, then a test point is necessary.

If the fault is on the output of a gate, and this signal forks to different gates, then conditions *PRE* and *INH* are the disjunction of these conditions for the different branches. Let t fork to signals t' and t'', and let $PRE(q,i)$ and $INH(q,i)$ be the condition for a premature firing and an inhibited firing, respectively, due to fault q stuck-at-i. Then

$$\begin{aligned} PRE(t,i) &\equiv PRE(t',i) \vee PRE(t'',i) \\ INH(t,i) &\equiv INH(t',i) \vee INH(t'',i) \\ (i = 0,1). & \end{aligned}$$

The application of the algorithm is the same as before.

The number of cases to be analyzed can be reduced significantly by using the testability theorems for inhibiting faults. For instance, if t is a primary input or the output of a gate, then fault t stuck-at-0 inhibits the first transition $t\uparrow$ in the handshaking expansion.

2. Finding Test Points

Given a fault that is untestable, find appropriate test points

Consider a circuit with signal s, and let fault s stuck-at-i be untestable. Then there is a state in the handshaking expansion where the fault may cause a premature firing of another variable, say $t\uparrow$. There are two possible reasons that the fault is untestable.

(1) The firing of $t\uparrow$ is unstable. Then there are one or more other inputs of the gate with output t that may change before transition $t\uparrow$ is completed, and that falsify the guard for $t\uparrow$. These other inputs have to be control points, so that the environment can insure that the guard for $t\uparrow$ remains **true** until the transition is complete; the firing then is stable.

(2) The firing of $t\uparrow$ is stable, but it does not result in a halting circuit or in a series of transitions that result in a premature firing of a primary output. Then an observation point is needed. Variable t can be an observation point, or any variable that has a stable premature firing as a result of the premature firing of $t\uparrow$.

3. Finding a Test to Detect all Faults

Given a list of faults, and a test sequence for each individual fault, find a reasonably small test sequence to test all faults.

Most stuck-at faults in a circuit are inhibiting faults. Most inhibiting faults are testable by simply raising and lowering all signals in the circuit. This is done by executing all of the handshaking expansion. Such a test is derived as follows.

(1) In the handshaking expansion, replace each transition $t\uparrow$ with $[t]$ and $t\downarrow$ with $[\neg t]$, for primary outputs t, and replace each $[s]$ with $s\uparrow$ and $[\neg s]$ with $s\downarrow$ for primary inputs s.

(2) If the handshaking expansion is a straight-line program, the test consists of executing this sequence once; if there is an if-statement in the handshaking expansion, then the test is such that for each if-statement each case is executed at least once.

Now simulate all faults with this test, and list the faults that it does not detect. Then generate a test for the remaining faults as follows.

(1) For a given fault, there is a test sequence that detects it. This test sequence

implies a state in the handshaking expansion where the fault is detected. If there is an if-statement in the handshaking expansion, partition the faults according to the state where the fault is detected. Two faults that are detected in different parts of an if-statement are not compatible.

(2) For each set in the partition, order the test sequences by length, from longest to smallest.

(3) For a given set in the partition, pick the fault with the longest test sequence. Determine if there is a reshuffling of this test sequence, such that the next-longest test sequence is a prefix of it. If so, check if this reshuffled sequence still tests the first fault. If so, the reshuffled sequence tests both faults. Continue the process for the other faults.

(4) In this manner, each partition yields one or more testing sequences. Then concatenate the sequences (if necessary resetting the circuit between them), to yield a test that detects all faults.

An alternative method to derive a small test sequence is to first find a test sequence for all faults that are stimulating and not inhibiting. Such a test sequence will, in all likelihood, also detect most inhibiting faults. Then find a test sequence for remaining inhibiting faults, if any.

4. Calculating Fault Coverage for a Test

Given a test sequence and a production rule set, find the fault coverage.

(1) Characterize each state of the test as a boolean condition. There is no "X" or don't-care value, since the environment controls all transitions on primary inputs, as specified by the test.

(2) For each fault compute conditions *INH* and *PRE.*

(3) Simulate the test for the correct circuit

(4) For each action in the test, and for each undetected fault, check if condition

INH holds. If so, the fault causes the circuit to halt, and this is detected when a subsequent transition of a primary output does not take place. Let the action in the test be on signal t. Then condition *INH* only needs to be checked for faults on inputs to the gate with output t.

(5) For each action in the test, and for each undetected fault, check if condition *PRE* holds. After the initialization, condition *PRE* must be checked for each fault in the circuit; thereafter, a transition of variable t can only make condition *PRE* **true** for faults on gates that have t as an input.

(6) If there is a fault for which *PRE* holds, check if the premature firing is stable. Check if the premature firing is propagated to a primary output, or if it causes another firing to be inhibited. If not, check if the premature firing is detected as a result of a subsequent action of the test. There are three possibilities:

   (a) If the premature firing is stable, and causes a premature firing of a primary output, or an inhibited firing of another variable, then the test detects the fault, and it need not be considered again for this test.

   (b) If the premature firing is unstable, or if it results in an unstable firing, then the test does not detect the fault; the fault need not be considered again for this test.

   (c) If the premature firing is guaranteed to return the circuit to a legal state, then the fault is not now detected. It can be detected subsequently with the same test.

It is sufficient to check conditions *PRE* and *INH* only for faults on inputs of gates. Let signal t fork to t' and t''. If $\mathit{INH}(t', i)$ ($\mathit{INH}(t'', i)$) holds before either $\mathit{PRE}(t', i)$ or $\mathit{PRE}(t'', i)$, then fault t stuck-at-i is detected when t' stuck-at-i (t'' stuck-at-i) is detected. Otherwise, only if $\mathit{PRE}(t', i)$ or $\mathit{PRE}(t'', i)$ holds at some point during the test can there be a premature firing of t. The analysis for the premature firing of t is similar to the analysis for the premature firing of t' or t''. It is not necessarily true that the result of these premature firings is the same.

APPENDIX B
Principles of Circuit Testing

1. Introduction

This chapter is a short overview of concepts used in the theory of testing circuits. Traditional theories focus mainly on testing synchronous circuits, hence most circuits in this chapter are synchronous. The definitions are such that they can be applied both to synchronous and to delay-insensitive circuits. The definitions and methods are consistent with the ones described by Abramovici, Breuer, and Friedman [3], and by Fujiwara [25], in two excellent textbooks.

The structure of this chapter is as follows. I give an overview of different types of fault modeling, and I introduce the *single stuck-at fault model* as the fault model most suited to testing delay-insensitive circuits. Following is a discussion on the merits and shortcomings of that model.

A fault in a circuit is detected by means of applying a test to the circuit. I define what a test is, and when a fault is detected by a test, as well as what faults I consider testable. An important result for combinational circuits is that any single stuck-at fault is testable if and only if the circuit is *non-redundant.* I define redundancy, and I describe several methods to test combinational logic, such as the *Boolean Difference* method and the *D-Algorithm.* Finally, I give a short overview of how sequential synchronous circuits can be tested, and how such circuits are altered to facilitate testing (design for testability). This involves adding test circuitry to transform a sequential circuit into a combinational one, and to partition a combinational circuit into smaller circuits.

2. Levels of Fault Modeling

There are several levels at which faults in a circuit can be modeled. A fault is a physical defect in a component [25]. This definition excludes testing to debug a design. If a design error is equivalent to a fault in an otherwise correct circuit, then the error may be tested using the fault theory. In general, it is difficult to model an arbitrary design error [39].

Testing can be done at the circuit level, at the board level, or at the system level. Other than issues of testing complexity (hence testing time), there is no difference for the testing theory between these levels. For the sake of simplicity, I assume that the unit under test is a single chip that is connected to its environment via its pads (external testing).

A fault is either a parametric fault or a logic fault. A parametric fault occurs when a value of a circuit parameter (for instance, a capacitance) is outside of the specification. A logic fault is a fault that alters the functionality of the circuit. A parametric fault is typically detected, with a small test structure, by the manufacturer [1, 22]. Since a parametric fault typically does not alter the functionality of a circuit, a delay-insensitive circuit with only a parametric fault will typically still operate correctly. I therefore consider only logic faults.

Another distinction is between permanent and intermittent (or transient) faults. It is rather difficult to detect an intermittent fault. The only approach to testing intermittent faults seems to be to construct a test for permanent faults, and to repeat this test until the probability of detecting the intermittent fault is sufficiently high [10, 37]. In this thesis, all faults are permanent faults.

A chip consists of a number of layers arranged vertically. The most obvious approach to testing chips is to take the layout of the circuit, and to investigate the consequences of a fabrication error: two layers may have a spurious connection, two layers may lack a connection, a wire may be missing, or two wires in the same layer may have a spurious interconnection. This approach is known as *geometric fault modeling* [60]. Geometric fault modeling can only be done on small circuits, as the number of possible faults increases rapidly with circuit size. It yields statistics on actual occurrences of faults in fabricated chips [27], which can be used to measure the effectiveness of more abstract fault models.

A fault model for permanent logic faults depends on the level at which the circuit is modeled, for instance at the register-transfer level, at the gate level, or at the transistor level. It is also possible to generate tests without a fault model, in order to test the functionality of a component [3]. The most widely used fault model at the register-transfer level or at the gate level is the *Stuck-At Fault Model*, where one or more variables in the circuit are either permanently at a high voltage (stuck-at-1), or at a low voltage (stuck-at-0). A model for faults at the transistor level is to assume either that the source and drain of the transistor are directly connected (a *stuck-on* fault), or that the source and drain are never connected (an *stuck-open* fault).

The model with stuck-open and stuck-on faults is a refinement of the stuck-at fault model. I shall generate tests that will detect most, or all, faults in the stuck-at model. In chapter 6 I investigate the fault coverage for these tests for stuck-open and stuck-on faults.

3. The Single Stuck-At Fault Model

DEFINITION B.1 (STUCK-AT FAULT). *A stuck-at fault is located either on an input or on an output of a gate. It causes the input or output to be either perma-*

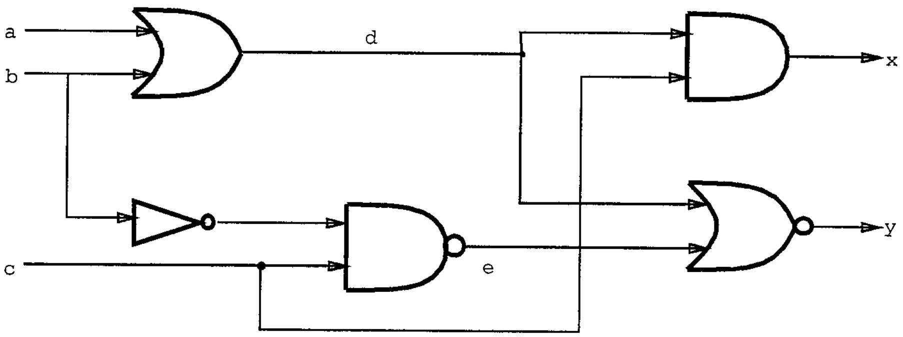

FIGURE B.1. Combinational circuit $C1$

nently at a high voltage (stuck-at-1), or at a low voltage (stuck-at-0).

If the output of a gate is stuck, then the corresponding input of any gate, to which it is connected, is also stuck. For an output of a gate that is connected to only one gate, there is no difference between a fault on the output and a fault on the corresponding input in the next gate. If an output, z, of a gate forks to N gates, where $N > 1$, then each of the N inputs of gates, to which z connects, can be stuck independently of the other inputs. In this case there are $N + 1$ different fault locations, and $2(N + 1)$ different stuck-at faults.

EXAMPLE B.1 (STUCK-AT FAULTS). *Combinational circuit $C1$ of figure* B.1 *has three inputs and two outputs, as well as three internal variables. There are three forks in the circuit, each with a fanout degree of two. Therefore there are fourteen different fault locations in the circuit, and twenty-eight possible stuck-at faults. For variable c the possible fault locations are (i) the primary input to the circuit, before the fork; (ii) the input to the NAND gate; and (iii) the input to the AND gate with output x.*

For a circuit with M fault locations, there are $3^M - 1$ different circuits with stuck-at faults. By contrast, there are just $2M$ different circuits with a single stuck-at fault. In the circuit of figure B.1 there are 28 different circuits with a single stuck-at fault, but almost 4.8 million circuits with multiple faults. For large circuits, it is impractical to consider all circuits with multiple faults. I therefore assume that any faulty circuit has a single stuck-at fault.

A *test* for a combinational circuit is an assignment of values to the primary inputs. A fault is *detected* if the value of at least one primary output in the faulty circuit is different from the correct circuit. In section 5 I give more precise definitions of a test and detectability, for arbitrary circuits.

The effect of a fault on the production rule set of a circuit is a substitution of some terms of the guards. If output t of a gate is stuck-at-0 (stuck-at-1), then all occurrences of t in the production rule set must be replaced by **false** (**true**). If input s to a gate with output t is stuck-at-0 (stuck-at-1), then all occurrences of s in the guards of the production rules for t must be replaced with **false** (**true**). After such a substitution, many guards can be further simplified, by reducing boolean expressions, until a guard is either **false** or **true**. This can be seen in the following example.

EXAMPLE B.2. *The production rule set for circuit* $C1$ *of figure* B.1 *is:*

$$\left\{\begin{array}{rcrcl} a & \vee & b & \rightarrow & d\uparrow \\ \neg a & \wedge & \neg b & \rightarrow & d\downarrow \end{array}\right.$$

$$\left\{\begin{array}{rcl} b & \rightarrow & b'\uparrow \\ \neg b & \rightarrow & b'\downarrow \end{array}\right.$$

$$\left\{\begin{array}{rcrcl} \neg b' & \vee & \neg c & \rightarrow & e\uparrow \\ b' & \wedge & c & \rightarrow & e\downarrow \end{array}\right.$$

$$\left\{\begin{array}{rcrcl} d & \wedge & c & \rightarrow & x\uparrow \\ \neg d & \vee & \neg c & \rightarrow & x\downarrow \end{array}\right.$$

$$\begin{cases} \neg d & \wedge & \neg e & \rightarrow & y\uparrow \\ d & \vee & e & \rightarrow & y\downarrow . \end{cases}$$

If primary input c is stuck-at-0, then input c to the NAND gate and input c to the AND gate with output x are also stuck-at-0. Replacing each occurrence of c in the production rule set, and simplifying boolean expressions yields the following gates that are different from the gates in circuit $C1$*:*

$$\begin{cases} \textbf{true} & \rightarrow & e\uparrow \\ \textbf{false} & \rightarrow & e\downarrow \end{cases}$$

$$\begin{cases} \textbf{false} & \rightarrow & x\uparrow \\ \textbf{true} & \rightarrow & x\downarrow . \end{cases}$$

Therefore e is a constant, **true**, *and x is a constant,* **false**. *It is now possible to do a further substitution, namely for e as input to the gate with output y:*

$$\begin{cases} \textbf{false} & \rightarrow & y\uparrow \\ \textbf{true} & \rightarrow & y\downarrow . \end{cases}$$

Output y is permanently **false** *if there is a fault c stuck-at-0. The fault is detected if the primary inputs are set to values such that primary output x is 1 in the correct circuit, for instance when a, b, and c are all set to 1.*

If input c to the NAND gate is stuck-at-0, then input c to the AND gate is not necessarily stuck-at-0. The effect of such a fault is that each occurrence of c in the production rules for e are replaced with **false***:*

$$\begin{cases} \textbf{true} & \rightarrow & e\uparrow \\ \textbf{false} & \rightarrow & e\downarrow . \end{cases}$$

Again, since e is now permanently **true**, *y as a consequence is permanently* **false**. *The production rules for x, however, do not change as a result of this fault.*

It is *not* true that a test detecting all single stuck-at faults will also detect any multiple stuck-at fault. The reason is a phenomenon known as *masking* [20]. Again consider circuit $C1$ of figure B.1. If the circuit has fault a stuck-at-0, then this fault is detected only with a test where input a is set to 1, and input b is set to 0. If in this circuit not only a is stuck-at-0, but also input b to the OR gate is stuck-at-1, then the same test does not detect this multiple fault: fault a stuck-at-0 *masks* the multiple fault. The multiple fault is detected for a test where a, b, and c are all 0. This is also a test for the single fault input b to the OR gate stuck-at-1. Therefore, this single fault does *not* mask the multiple fault.

For some circuits, a fault f_0 masks the multiple fault $\{f_0, f_1\}$, and fault f_1 also masks $\{f_0, f_1\}$. Therefore, a test that detects either f_0 or f_1 does not detect $\{f_0, f_1\}$ [69]. This is known as *circular masking*, a necessary but not sufficient condition for the multiple fault to be undetectable.

There are several algorithms to find multiple stuck-at faults that cannot be detected with any test for single faults [2, 15]. The fraction of such multiple faults is usually small. For instance, Jacob and Biswas [35] have shown that it is less than 0.4 percent for circuits with at least three primary outputs. Agarwal and Fung [4] show that for some combinational circuits less than 2 percent of multiple faults, consisting of six or fewer single faults, are undetectable with a test for single stuck-at faults.

4. Merits and Shortcomings of the Stuck-At Fault Model

The stuck-at fault model is an accurate and concise model for analysis of faults in relay circuits. Few faults on components currently produced, however, can be accurately described as stuck-at faults. This is a discussion on why the stuck-at fault model is still useful.

The stuck-at fault model describes faults at the gate level. Therefore it is independent of the technology with which the gates are implemented. This is both an advantage and a shortcoming of the model.

For a given test we can calculate the number of faults that it can detect. The fault coverage of the test, then, is independent of the way the gates are implemented. This is, a priori, unrealistic.

The description of delay-insensitive circuits is at the gate level. Such a circuit operates correctly regardless of the technology with which it is implemented (see chapter 2). Since the stuck-at fault model is also at the gate level, it is ideally suited for fault analysis of delay-insensitive circuits.

A parasitic transistor, or a short of internal nodes inside of a gate may result in a different gate. Such a fault may not be a stuck-at fault [40]. Even shorts between gates may lead to faults that are not stuck-at faults: A typical fabrication fault in MOS circuits is a spurious connection between crossing metal wires in different layers [27]. Unless one of the metal wires is either power or ground, such a fault is not equivalent to a stuck-at fault. For example, in a 80486 processor, there are about 1700 metal wires in one direction crossing about 1700 metal wires in the other direction. Thus there are almost three million possible faults that cannot be modeled as stuck-at faults [36]. Many faults that are not stuck-at faults can be modeled as a multiple stuck-at fault, however [40].

What makes the single stuck-at fault model attractive, is that the number of faults considered is relatively small, and that it allows fault theories to be formulated using boolean logic. On the practical side, it has been shown that there is a strong correlation between fault coverage under the single stuck-at model (that is, the percentage of stuck-at faults that can be detected by a test) and the percentage of faulty parts that fail such a test [5, 18, 53, 68].

5. Tests

A circuit, C, is tested by a sequence of actions of its environment. The environment repeatedly sets the value of the primary inputs of C, then it observes the primary outputs of C. If, at some point during execution of the test, the value of at least one primary output of C differs from the output of a fault-free circuit, then a fault is detected in C.

DEFINITION B.2 (TEST). *A* test *for circuit C is a finite sequence of actions by the environment of C. An action is either setting the value of a primary input of C, or observing the value of a primary output of C.*

A primary input, a, of a circuit is set either to **true** (denoted $a\uparrow$), or **false** (denoted $a\downarrow$). As a result of setting the primary inputs, a primary output, z, may change, either to **true** (denoted $[z]$, "wait for z"), or **false** (denoted $[\neg z]$, "wait for not z"). For a test, there are two composition operators: sequential composition (";") and parallel composition (","). For delay-insensitive circuits, the sequential composition operator is commutative for wait-operations. Therefore $[x]; [y]$, and $[y]; [x]$ are equivalent. It is also written as $[x \wedge y]$. Parallel composition, of course, is also commutative.

In the execution of a test, there is a difference in interpretation of the wait-actions between synchronous and delay-insensitive circuits. For a synchronous circuit, the primary outputs are allowed to change arbitrarily between clock ticks [54]. After the primary inputs of the circuit are set, the environment has to wait for a predetermined amount of time before the primary outputs can be observed. This time is at least the clock period, but could be longer. For a faulty circuit, some output may not change before a full clock period. This is the case when the critical path is longer than the clock period as a result of the fault.

For a delay-insensitive circuit, each primary output changes value at most once while the primary inputs remain constant, because of stability. This is a consequence of the stability requirement for delay-insensitive circuits. For a correct circuit, once the primary outputs have the correct values, the environment may set the primary inputs to the next set of values in the test sequence. In case the circuit is faulty, however, primary output changes may occur *after* the environment has observed the correct primary output values. In order to be able to observe these incorrect output changes, the environment has to wait for some time between observing the correct primary outputs, and setting the values of the primary inputs. This leads to problems in the model for delay-insensitive circuits, where delays are assumed to be arbitrary and unbounded. For a discussion, see chapter 3. In an actual circuit, we can assume that there is an upper bound on the time that the environment has to wait for such incorrect output changes.

EXAMPLE B.3. *In the combinational circuit of figure* B.1, *assume that in the initial state all primary inputs are* **false**. *Then the primary outputs are also* **false**, *and internal variable e is* **true**. *If c is set to* **true**, *then primary output y becomes* **true**. *If, after y has been observed, b is set to* **true**, *then x becomes* **true**, *and y becomes* **false**. *Hence a test for this circuit is*

$$c\uparrow; [y]; b\uparrow; [x \wedge \neg y].$$

Suppose this test is executed, and no transition $y\uparrow$ occurs after c is set to **true**. Then there must be a fault in the circuit: the fault is *detected.*

DEFINITION B.3 (DETECTED FAULT). *Let T be a test for circuit C. Let circuit C' be the same circuit, but with a single stuck-at fault. If, during execution of test T, there is a point at which a primary output of C' has a value that cannot occur at the same point in the test for circuit C, then the fault is* detected.

There are two ways that a primary output in the faulty circuit can differ from the same primary output in the correct circuit. Either there is a transition on the primary output when no transition is expected, or there is no transition when one or more is expected. If circuit $C1$ has primary input c stuck-at-0, then no transition

$y\uparrow$ will occur after execution of $c\uparrow$ as the first action of the test. Therefore fault c stuck-at-0 is detected by the test. Conversely, if circuit $C1$ has input d to the AND gate stuck-at-1, then transition $c\uparrow$ will cause a transition $x\uparrow$ that does not occur in the correct circuit. The test also detects this fault.

If, in circuit $C1$, both b and c are set to **true** simultaneously, then eventually output x will become **true**. The test sequence is

$$b\uparrow, c\uparrow; [x].$$

Depending on the relative propagation delays in the circuit, it is possible that there are transitions $e\downarrow$ and $e\uparrow$. This is the case if the propagation delay of the inverter (with input b) is longer than the propagation delay from input c to the NAND gate plus the propagation delay of that gate. As a result of these transitions of e, transitions $y\uparrow$ and $y\downarrow$ are possible. During execution of the test a voltage spike may be observed on primary output y. This condition is known as a *hazard*.

For synchronous circuits, a hazards are allowed. After waiting "long enough", the primary outputs have stable values, and are not going to change unless the primary inputs change. For delay-insensitive circuits, a hazard condition may wreak havoc on the circuit. If the circuit of figure B.1 is (part of) a delay-insensitive circuit, then setting both b and c to **true** simultaneously in the initial state must be disallowed.

Even if a circuit is hazard-free, a fault may cause a hazard condition. In other words, the fault results in a circuit that violates the stability requirement. When a test for the circuit is executed, there may be a voltage spike on a primary output, so that the fault is detected. But any hazard can only occur under specific propagation delays in the circuit that may not occur during testing. I shall only consider a fault detectable if it is detected when no assumption is made on the propagation delays in the circuit (except, of course, that the propagation delays are finite). A fault that causes only a possible hazard during a test is therefore *not* detectable with that test.

DEFINITION B.4 (DETECTABLE FAULT). *A fault in circuit C is* detectable *with test T if the fault is guaranteed to be detected, regardless of the propagation delays in the circuit.*

DEFINITION B.5 (TESTABLE FAULT). *A fault in circuit C is* testable *if there is a test that detects the fault.*

EXAMPLE B.4. *For circuit $C1$, the test*

$$c\uparrow; [y]; b\uparrow; [x \wedge \neg y]$$

will detect faults c stuck-at-0, and input d to the AND gate stuck-at-1, regardless of the propagation delays. Both faults are detectable.

DEFINITION B.6 (FAULT COVERAGE). *Let T be a test for a circuit. The fault coverage for T is the percentage of testable faults that are detectable by applying test T to the circuit.*

The goal of test generation is to find a test that maximizes the fault coverage, while minimizing the test length. A test that detects all testable faults in a circuit is known as a *complete test*.

There are more sophisticated ways to detect a fault than by observing the primary outputs some time after the primary inputs have been set. As mentioned, a fault can be detected if a hazard on a primary output occurs. Some faults can cause a short circuit. By measuring the power consumption of the circuit, such faults are detected. Also, given the physical parameters of a chip, the propagation delays can be computed within some range. If, during a test, a primary output changes much faster or much slower than expected, then, again, a fault is detected. This can also be due to a parametric fault [1].

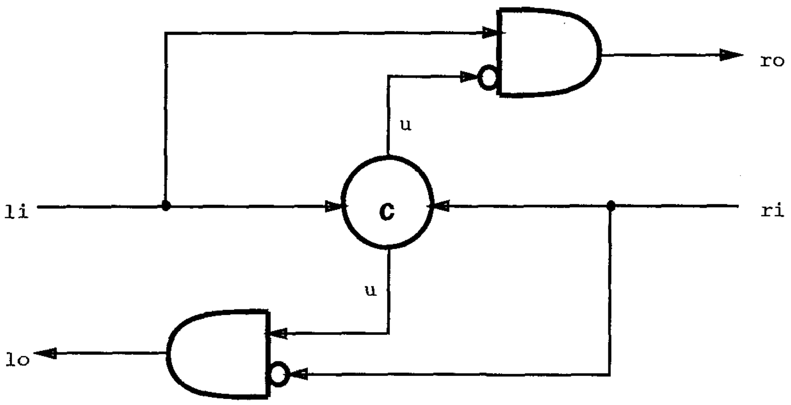

FIGURE B.2. Delay-insensitive circuit $C2$, a D-element

6. Redundancy

Consider circuit $C2$ of figure B.2. This delay-insensitive circuit is known as a D-element. Its production rule set is:

$$\begin{cases} li \wedge ri \rightarrow u\!\uparrow \\ \neg li \wedge \neg ri \rightarrow u\!\downarrow \end{cases}$$

$$\begin{cases} li \wedge \neg u \rightarrow ro\!\uparrow \\ \neg li \vee u \rightarrow ro\!\downarrow \end{cases}$$

$$\begin{cases} \neg ri \wedge u \rightarrow lo\!\uparrow \\ ri \vee \neg u \rightarrow lo\!\downarrow . \end{cases}$$

In the initial state of the circuit, all variables are **false**. Define test T_D to be the following sequence:

$$li\!\uparrow; [ro]; ri\!\uparrow; [\neg ro]; ri\!\downarrow; [lo]; li\!\downarrow; [\neg lo].$$

With this test the circuit is hazard-free. Let's assume that the only way the circuit is operated is by executing test T_D, or repeating test T_D an arbitrary number of times.

Consider the gate with output ro. If the production rule $\neg li \vee u \rightarrow ro\downarrow$ is replaced with $u \rightarrow ro\downarrow$, then the circuit still has no hazard, and it will still execute test T_D correctly. The term $\neg li$ is *redundant* in the production rule for $ro\downarrow$.

Definition B.7 (Redundant Term). *Let C be a circuit, and let s be an input to a gate with output t, such that s or $\neg s$ occurs as a term in the guard of production rule $t\uparrow$. Replace the term with either* **true** *or* **false**. *If the resulting production rule set is still a correct implementation of the specification of C, then this term is* redundant. *Similarly for $t\downarrow$.*

For the above production rule set of the D-element, the other redundant terms are ri in the guard for $lo\downarrow$, li in the guard for $u\uparrow$, and $\neg ri$ in the guard for $u\downarrow$.

Eliminating redundant terms in a circuit does not always result in a smaller circuit. For instance, if $\neg li$ is removed as a term in the production rule for $ro\downarrow$, then the gate with output ro changes from a combinational gate into a state-holding element, which may require more transistors to implement.

If an input to a gate is replaced with either **true** or **false**, then the resulting gate can be implemented with fewer transistors. If the resulting circuit is still a correct implementation of the specification, then the input is said to be *redundant.*

Definition B.8 (Redundant Input). *Let C be a circuit, and let s be an input to a gate with output t. Replace each occurrence of s and $\neg s$ in the production rules for t with either* **true** *or* **false**. *If the resulting production rule set is still a correct implementation of the specification of C, then s is a* redundant input.

Definition B.9 (Redundant Gate). *Let C be a circuit, and let t be the output of gate G. Replace each occurrence of t and $\neg t$ in the production rule set with either* **true** *or* **false**. *If the resulting production rule set is still a correct implementation of the specification of C, then G is a* redundant gate.

DEFINITION B.10 (REDUNDANT CIRCUIT). *A circuit is redundant if it has a redundant input or a redundant gate.*

EXAMPLE B.5. *Consider circuit $C1$, and assume that it is used as a synchronous circuit. The boolean expressions that the circuit implements are*

$$x \equiv (a \vee b) \wedge c$$
$$y \equiv \neg a \wedge \neg b \wedge c$$

Call the output of the inverter b'. If input b' to the gate with output e *is replaced with* **true**, *then the circuit still implements the same boolean expressions: b' is a redundant input. Since b' is input to only one gate, the inverter must therefore be a redundant gate. Circuit $C1$ is a redundant circuit.*

Replacing b' with **true** *reduces the gate with output* e *to a simple inverter.*

For the D-element, there are no redundant inputs or redundant gates, hence the circuit is not redundant.

There are other forms of redundancy. For instance, two inverters in series can always be replaced by a single wire. For some circuits it is possible to remove one or more gates, and to reconnect the remaining gates, resulting in a smaller circuit. In chapter 5, I show that the specification of two D-elements connected in series is the same as the specification of a single D-element. As a result, half of the gates in the former circuit are "redundant". It is rather more difficult to detect this general type of redundancy in a circuit than the simple input- or gate redundancy defined above. I shall not consider general redundancy.

7. Testing Combinational Logic

The following theorem relates the testability of faults in combinational logic to redundancy.

THEOREM B.1. *Let C be a circuit. Then C is non-redundant if and only if every single stuck-at fault is testable.*

Proof: If C is redundant, then there is a redundant input or a redundant gate. Let s be a redundant input, or the output of a redundant gate. Then replacing s with **true** or **false** does not alter the functionality of C. If s can be replaced with **true**, then fault s stuck-at-1 is not testable, and if s can be replaced with **false**, then fault s stuck-at-0 is not testable.

Let s be a variable in C such that fault s stuck-at-0 is not testable. Then for all values of the primary inputs the correct circuit and the circuit with fault s stuck-at-0 produce the same values for the primary outputs. Therefore replacing s with **false** does not alter the functionality of C: the circuit is redundant. □

Whereas any non-redundant combinational circuit is fully testable for single stuck-at faults, determining whether the circuit is non-redundant is a hard problem. Whether a given single stuck-at fault in a combinational circuit is testable is an NP complete problem [34]. The problem remains NP complete if the set of circuits is restricted to n-level monotone circuits, where $n \geq 3$. For 2-level monotone circuits, the problem is polynomially solvable [24, 26].

Given that a combinational circuit is non-redundant, the test generation problem is also hard. For a non-redundant monotone circuit, finding a test that detects a given single stuck-at fault is an NP complete problem [25]. Efficient test generation for combinational circuits is therefore an important area of research. I shall describe two simple approaches to the test generation problem, the boolean differences method and the D-algorithm.

Boolean Differences. The boolean difference method is an algebraic method to find all the test vectors that will detect a given stuck-at fault [67]. Let C be a combinational circuit, with primary inputs x_i $(0 \leq i < m)$ and primary outputs

y_j $(0 \leq j < n)$. Then there are boolean functions f_j $(0 \leq j < n)$ such that for each j

$$y_j = f_j(x_0, x_1, \ldots, x_{m-1}).$$

If input x_i is stuck-at-0, then, for any test vector with which the fault is detected, input x_i is **true**. The fault is detected if there is an output y_j that has different values in the correct and in the faulty circuit. In a formula, fault x_i stuck-at-0 is testable if

$$\begin{aligned} \exists_j \exists_{X_0, X_1, \ldots, X_{m-1}} : \quad & f_j(X_0, \ldots, X_{i-1}, \textbf{false}, X_{i+1}, \ldots, X_{m-1}) \\ \neq \; & f_j(X_0, \ldots, X_{i-1}, \textbf{true}, X_{i+1}, \ldots, X_{m-1}). \end{aligned}$$

DEFINITION B.11 (BOOLEAN DIFFERENCE). *For a boolean function of m variables $f(x_0, x_1, \ldots, x_{m-1})$ the* boolean difference with respect to input x_i *is*

$$f(x_0, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{m-1}) \otimes f(x_0, \ldots, x_{i-1}, \neg x_i, x_{i+1}, \ldots, x_{m-1}).$$

The boolean difference is denoted

$$\frac{df(x_0, \ldots, x_{m-1})}{dx_i.}$$

THEOREM B.2. *For a combinational circuit with output function f, the fault x_i stuck-at-0 is detectable if the primary inputs are set such that*

$$x_i \wedge \frac{df}{dx_i}$$

holds. Fault x_i stuck-at-1 is detected if there is an assignment to the primary inputs such that

$$\neg x_i \wedge \frac{df}{dx_i}$$

holds.

Proof: Trivial □

EXAMPLE B.6. *For circuit $C1$ the function that primary output x implements is $(a \vee b) \wedge c$. Therefore*

$$\begin{aligned}\tfrac{dx}{da} &\equiv ((\mathbf{true} \vee b) \wedge c) \otimes ((\mathbf{false} \vee b) \wedge c) \\ &\equiv c \otimes (b \wedge c) \\ &\equiv \neg b \wedge c.\end{aligned}$$

Fault a stuck-at-0 is detectable with any test for which $a \wedge \neg b \wedge c$, and fault a stuck-at-1 is detectable with any test for which $\neg a \wedge \neg b \wedge c$. In that case, the value of primary output x is different in the correct circuit from the faulty one.

The result can be extended to faults of internal variables. Let s be an internal variable. Re-write the output function f as a function not only of the primary inputs, but also of s (as if s were an additional primary input). Then fault s stuck-at-0 is testable if there is an assignment to the primary inputs such that $s \wedge \frac{df}{ds}$ holds, and s stuck-at-1 is testable if $\neg s \wedge \frac{df}{ds}$ holds for some assignment to the primary inputs.

EXAMPLE B.7. *Consider internal variable e in circuit $C1$. Primary output y, as a function of e, is $\neg(e \vee a \vee b)$, and $e \equiv \neg(\neg b \wedge c)$. Fault e stuck-at-0 is testable in any state where*

$$\begin{aligned}e \wedge \tfrac{dy}{de} &\equiv e \wedge (\neg(\mathbf{true} \vee a \vee b) \otimes (\neg(\mathbf{false} \vee a \vee b)) \\ &\equiv e \wedge (\mathbf{false} \otimes (\neg a \wedge \neg b)) \\ &\equiv e \wedge \neg a \wedge \neg b) \\ &\equiv \neg(\neg b \wedge c) \wedge \neg a \wedge \neg b) \\ &\equiv \neg a \wedge \neg b \wedge \neg c.\end{aligned}$$

Fault e stuck-at-1 is testable in any state where

$$\begin{aligned}\neg e \wedge \tfrac{dy}{de} &\equiv \neg e \wedge (\neg(\mathbf{true} \vee a \vee b) \otimes (\neg(\mathbf{false} \vee a \vee b))) \\ &\equiv \neg e \wedge \neg a \wedge \neg b \\ &\equiv \neg a \wedge \neg b \wedge c.\end{aligned}$$

For variable b' we have $y = \neg a \wedge \neg b \wedge b' \wedge c$ and $b' = \neg b$. Fault b' stuck-at-1 is testable if

$$\begin{aligned} \neg b' \wedge \tfrac{dy}{db'} &\equiv b \wedge (\neg a \wedge \neg b \wedge c \otimes \mathbf{false}) \\ &\equiv \mathbf{false}. \end{aligned}$$

Therefore there is no test that detects fault b' stuck-at-1. Indeed, variable b' is redundant.

If there is a test vector for a given stuck-at fault, then the boolean difference method will find it. However, the boolean difference method will generate all tests for which the fault is detectable, which is more than strictly necessary. The D-Algorithm, on the contrary, tries to find a single setting of the primary inputs with which the fault is detectable. This involves backtracking.

8. The D-Algorithm

Let circuit C have a single stuck-at fault, on gate G. To detect the fault during a test, there are two conditions:

(1) The circuit has to be in a state where the fault causes the output of gate G to have the wrong value.

(2) Once the output of G has the wrong value, the result has to be propagated to a primary output, so that the environment can detect the fault.

Since a combinational circuit does not contain any state-holding gates, the primary inputs have to be set only once to make a given testable fault detectable. A method to derive a test vector for such a fault is first to derive a condition on the inputs of the gate G, so that the output has the wrong value. For a combinational gate with production rules

$$\left\{ \begin{array}{ccccccc} x & \wedge & B & \vee & C & \rightarrow & z\uparrow \\ (\neg x & \vee & \neg B) & \wedge & \neg C & \rightarrow & z\downarrow, \end{array} \right.$$

where B and C are boolean conditions, let input x be stuck-at-1. If the circuit is in a state where $\neg x \wedge B \wedge \neg C$, then the correct circuit will have output z **false**, whereas the faulty circuit has output z **true**. This occurs in no other state. Then the value of the primary inputs are derived (by backward propagation), as well as the primary output that will have the wrong value for the faulty circuit (by forward propagation).

If forward propagation is successful, then there is a path through a sequence of gates, where the output of the last gate in the sequence is a primary output, such that each gate in the the sequence has the wrong value for the faulty circuit. If there is exactly one such path for a given setting of the primary inputs, then it is known as *single-path sensitization.* Most faults are detectable with single-path sensitization, but there are faults that are not detectable with single-path sensitization [65].

The D-Algorithm [63] tries to find any single or multiple path to the primary outputs that will make the fault testable. It employs a five-valued logic. Apart from the standard values 1 (or **true**), 0 (or **false**), and X (or don't care), there are D, for a variable that is 1 in the correct circuit, and 0 in the faulty one, and $\overline{\mathrm{D}}$, for a variable that is 0 in the correct circuit, and 1 in the faulty one.

With the values D and $\overline{\mathrm{D}}$ the logic tables are extended as follows:

$\wedge$	0	1	X	D	$\overline{\mathrm{D}}$
0	0	0	0	0	0
1	0	1		D	$\overline{\mathrm{D}}$
X	0				
D	0	D		D	0
$\overline{\mathrm{D}}$	0	$\overline{\mathrm{D}}$		0	$\overline{\mathrm{D}}$

$\vee$	0	1	X	D	$\overline{\mathrm{D}}$
0	0	1		D	$\overline{\mathrm{D}}$
1	1	1	1	1	1
X		1			
D	D	1		D	1
$\overline{\mathrm{D}}$	$\overline{\mathrm{D}}$	1		1	$\overline{\mathrm{D}}$

$\neg$	0	1	X	D	$\overline{\mathrm{D}}$
	1	0		$\overline{\mathrm{D}}$	D

The blank entries are undefined.

These tables specify how a fault is propagated forward towards the primary outputs. For instance, if one input of an AND gate is 1, and the other D, then, by implication, the output of the gate must be D. To propagate backward towards the primary inputs, there are similar implications. For instance, if the output of an AND gate is D, then both inputs must be set to 1, and if the output is $\overline{\mathrm{D}}$, then at least one input has to be 0. For an OR gate, if the output is D, then at least one input must be 1, and if the output is $\overline{\mathrm{D}}$, then both inputs must be 0.

The D-Algorithm to detect a single stuck-at fault now works as follows:

(1) Select a variable with a stuck-at fault in the circuit. For a stuck-at-1 fault, set the variable to $\overline{\mathrm{D}}$, for a stuck-at-0 fault set the variable to D. Set all other variables to X (don't care).

(2) Forward propagation. List all gates with at least one input D or $\overline{\mathrm{D}}$, and output X. These gates comprise the so-called *D-frontier.* Select a gate in the D-frontier, and set the other inputs of that gate in such a way that the output of the gate is D or $\overline{\mathrm{D}}$.

(3) Compute the values of any variable that is implied by the known values in the circuit (any value that is not X). If the result is an inconsistent

assignment (where a variable has to be both 0 and 1), then the selected gate in the D-frontier cannot be used for forward propagation. Return to step 2, selecting another gate. If the result is still consistent, then return to step 2, computing a new D-frontier, until the output of the gate selected is a primary output.

(4) Backward propagation. Once a sensitized path has been found from the faulty gate to a primary output, the values of the primary inputs are determined. Compute the value of any inputs that are implied by the value of the output of a gate. For instance, if the output of an AND gate is 1, then both inputs are 1.

(5) If there remain variables with undetermined values, then select one, and set it to 0 or 1. Compute the value of any variable that is implied by this assignment. Again, if the assignment is inconsistent, select another value for the variable, and if it is consistent, then select another variable with an unknown value, etc.

(6) If the algorithm terminates with an assignment for all variables in the circuit, and the assignment is consistent, then a test vector is found that makes the fault testable, since at least one primary output will have value either D or $\overline{\mathrm{D}}$. Otherwise, there is no assignment of the primary inputs that makes the fault detectable.

The algorithm involves backtracking, and is not efficient. This is to be expected, as the problem it solves is NP-complete. If there is a test for a fault, then the D-algorithm will find it. The algorithm can easily be extended to cover multiple faults.

EXAMPLE B.8. *Consider circuit* C3 *of figure* B.3. *Let output e be stuck-at-0. Applying the D-algorithm, all variables are set to X, except for e, which is D. The*

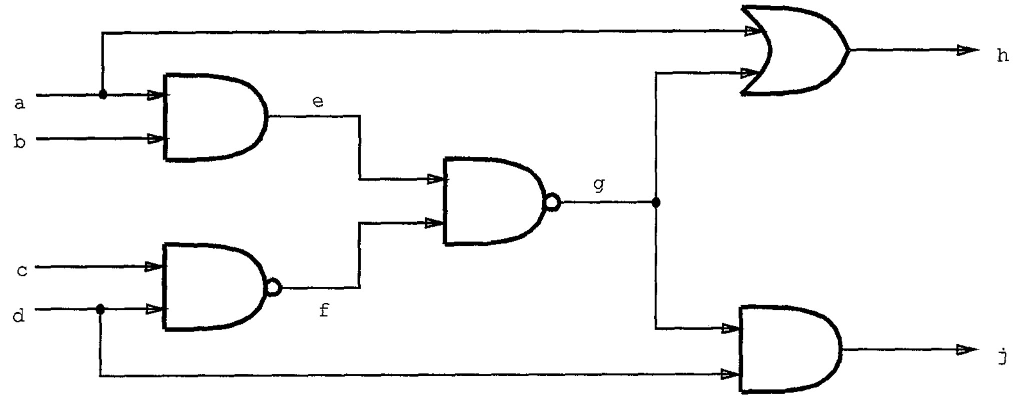

FIGURE B.3. Combinational circuit $C3$. A test for fault e stuck-at-0 is derived using the D-algorithm

D-frontier is the gate with output g. This gate has to be used for forward propagation. Therefore f has to be 1, and output g is $\overline{D}$*. The D-frontier then becomes the gates with outputs h and j. The implication step does not yield any further values at this point. Choose the OR gate with output h for forward propagation. Then output h is* $\overline{D}$ *if input a is 0. In the backward propagation step, this assignment will turn out to be inconsistent, since if a is 0, then e is 0 for the correct circuit.*

Backtracking, choose the gate with output j for forward propagation. Then j is $\overline{D}$ *while d is 1. Since j is a primary output, the forward propagation is done. The values of primary inputs a, b, and c are still unknown. In order for e to be D, a and b both have to be 1, and f with value 1 while d is 1 implies that c is 0. This is a consistent assignment to the primary inputs.*

From this example it is clear that a combining of forward and backward propagation steps can be more efficient than doing the backward propagation after the forward propagation [64]. There are many other algorithms to detect a fault in a combinational circuit. Most are based on the D-algorithm. Optimizations include using a calculus with more values [6, 14], and algorithms to reduce the search space for cases in which the D-algorithm is known to perform badly, such

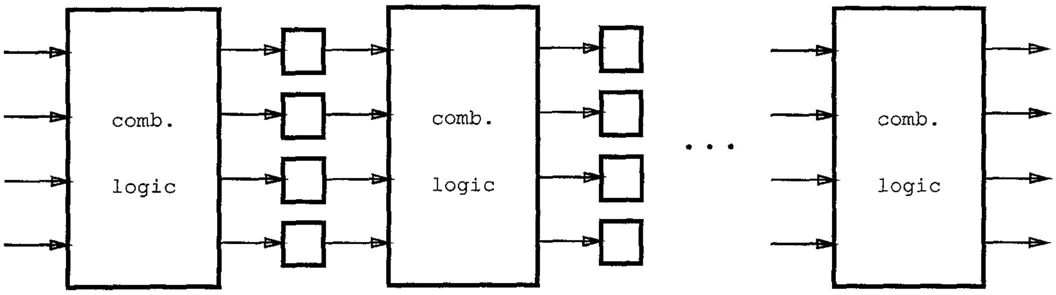

FIGURE B.4. Test generation for a sequential circuit by replication as PODEM [29].

9. Testing Sequential Synchronous Circuits

The problem of testing sequential synchronous circuits, without the addition of test circuitry, is much more complicated than testing combinational circuits. Since a sequential circuit has state-holding elements, a circuit with a fault does not necessarily reset to the same initial state as the correct circuit. It is even possible that, because of a fault, a circuit does not have a well-defined initial state [3]. For faults that cause the circuit to be in a well-defined but unknown initial state, it may be possible to apply an *initialization sequence* to the circuit, so that both the faulty and the correct circuit are in a known (but not necessarily the same) state [3].

Once the circuit is in a known state, a test is applied to the circuit. The test typically consists of a sequence of events to bring the circuit in a state where the fault causes the output of a gate to have the wrong value, followed by a sequence to propagate this wrong value to a primary output.

A common approach to test generation for sequential circuits is to duplicate the circuit several times, to form a sequence of combinational circuits. For a test sequence of length n the circuit is duplicated n times. See figure B.4. A single stuck-at fault in the original circuit now corresponds to a multiple stuck-at fault in the duplicated circuit. An algorithm, such as the D-algorithm, can now be used to generate a test sequence for this multiple fault, similar to the algorithms for combinational circuits [25]. A complication for the D-algorithm is that multiple faults may cancel each other. Also, because of the backtracking involved, application of the algorithm for large circuits with long test sequences may take very long.

For that reason, extra circuitry is usually added to the circuit to reduce the complexity of test generation, as well as to reduce the actual test length, at a cost of circuit area and a degradation of the speed of the circuit. A test structure may also be added to a combinational circuit, so as to partition the circuit, in order to simplify the test generation.

Suppose that the state of each state-holding element in the circuit can be set and observed by the environment. Then the resulting circuit is equivalent to a combinational circuit. Because of pin-count limitations, it is typically not possible to connect the state-holding element directly to the environment for the purpose of testing the circuit. Therefore a queue or stack is added to the circuit, so that the state-holding elements can be set sequentially, and so that the values of the state-holding elements can be observed by the environment sequentially. Such a structure is known as a *scan design* [25].

Testing the circuit then is done as follows. Using the queue, the values of all the state-holding elements are set. Then the circuit proper operates for one cycle, after which the values of the state-holding elements are read. Since the circuit being tested is equivalent to a combinational circuit, the D-algorithm for single stuck-at faults can be used. In addition, test vectors have to be added to test for any faults in the test structure, or in any of the state-holding elements.

A simple scan design is the so-called *shift-register modification* [72]. The state-holding elements are clocked D flip-flops. The flip-flops are connected together

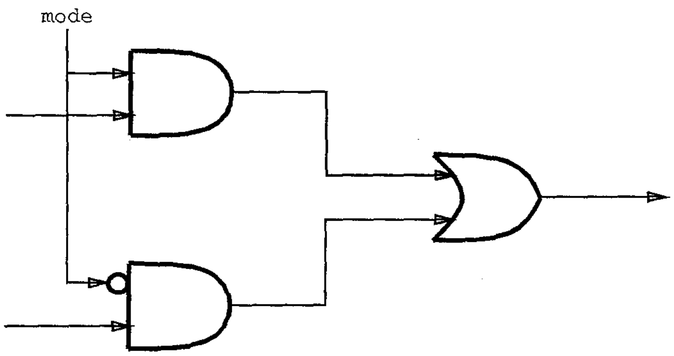

FIGURE B.5. A double-throw switch

to form a shift register, and to each flip-flop a double-throw switch is added (see figure B.5). Each double-throw switch is controlled with a “mode” bit. If it is 0, then the circuit operates as before the addition of the shift register: the state of the flip-flops is sent to the combinational logic, and the result from the combinational logic is stored in the flip-flops for the next clock cycle. If the mode bit is 1, then the results from the combinational logic are no longer sent to the flip-flops. Instead, the flip-flops form a shift register, and values can be shifted in and out of the circuit by setting the value of the first flip-flop in the chain, and by observing the last one (see figure B.6).

Shift-register modification is but one of many scan designs. Similar to shift-register modification, but using more hardware, is a *scan path* [25], which allows for easy partitioning of combinational logic. Another approach is *level-sensitive scan design*, or LSSD, which is primarily used in IBM systems [21, 56]. It also uses more hardware than shift-register modification. The main advantage is that it results in circuits that are hazard-free, both during testing and during normal operation.

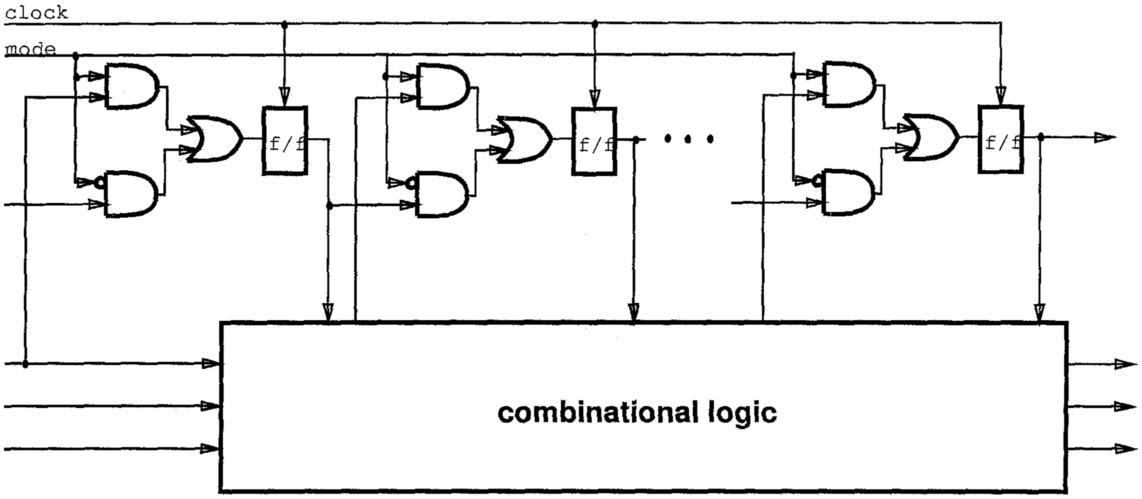

FIGURE B.6. A sequential synchronous circuit with shift-register modification

Bibliography

[1] Jacob A. Abraham and Vinod K. Agarwal. Test generation for digital systems. In Dhiraj K. Pradhan, editor, *Fault-Tolerant Computing*, chapter 1. Prentice-Hall, Englewood Cliffs, N.J., 1986.

[2] Miron Abramovici and Melvin A. Breuer. Fault diagnosis in synchronous sequential circuits based on an effect-cause analysis. *IEEE Transactions on Computers*, C-31(12):1165 – 1172, December 1982.

[3] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design.* Computer Science Press, 1990.

[4] Vinod K. Agarwal and Andy S.F. Fung. Multiple fault testing of large circuits by single fault test sets. *IEEE Transactions on Computers*, C-30(11):855 – 865, November 1981.

[5] Vishwani D. Agarwal, Sharad C. Seth, and Prathima Agarwal. LSI product quality and fault coverage. In *Proceedings of the 18th Design Automation Conference*, pages 196 – 203, 1981.

[6] Sheldon B. Akers. A logic system for fault test generation. *IEEE Transactions on Computers*, C-25(6):620 – 629, June 1976.

[7] Peter A. Beerel and Teresa H.-Y. Meng. Semi-modularity and self-diagnostic asynchronous control circuits. In Carlo H. Séquin, editor, *Advanced Research*

in VLSI: Proceedings of the 1991 University of California/Santa Cruz Conference, pages 103 – 117, Cambridge, Mass./London, 1991. The MIT Press.

[8] J.M. Berger. A note on error detection codes for asymmetric channels. *Information and Control*, 4:68 – 73, 1961.

[9] Rodolfo Betancourt. Derivation of minimum test sets for unate logical circuits. *IEEE Transactions on Computers*, C-20(11):1264 – 1269, November 1971.

[10] Melvin A. Breuer. Testing for intermittent faults in digital circuits. *IEEE Transactions on Computers*, C-22(3):241 – 246, March 1973.

[11] Steven M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.

[12] Steven M. Burns and Alain J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In Jonathan Allen and F. Thomas Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*, pages 35 – 50, Cambridge, Mass./London, 1988. The MIT Press.

[13] Steven M. Burns and Alain J. Martin. Performance analysis and optimization of asynchronous circuits. In Carlo H. Séquin, editor, *Advanced Research in VLSI: Proceedings of the 1991 University of California/Santa Cruz Conference*, pages 71 – 86, Cambridge, Mass./London, 1991. The MIT Press.

[14] Charles W. Cha, William Donath, and Füsun Özgüner. 9-V algorithm for test pattern generation of combinational digital circuits. *IEEE Transactions on Computers*, C-27(3):193 – 200, March 1978.

[15] C.W. Cha. Multiple fault diagnosis in combinational networks. In *Proceedings of the 16th Design Automation Conference*, pages 149 – 155, June 1979.

[16] Thomas J. Chaney and Charles E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421 – 422, April 1973.

[17] T.A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications.* PhD thesis, Massachusetts Institute of Technology, 1987.

[18] R.G. Daniels and W.C. Bruce. Built-in self-test trends in Motorola microprocessors. *IEEE Design and Test of Computers*, 2(2):64 – 71, 1985.

[19] Ilana David, Ran Ginosaur, and Michael Yoeli. An efficient implementation of boolean functions as self-timed circuits. *IEEE Transactions on Computers*, C-41(1):2 – 11, January 1992.

[20] Francisco J.O. Dias. Fault masking in combinational logic circuits. *IEEE Transactions on Computers*, C-24(5):476 – 482, May 1975.

[21] E.B. Eichelberger and T.W. Williams. A logic design structure for LSI testability. In *Proceedings of the 14th Design Automation Conference*, pages 462 – 468, 1977.

[22] Robert J. Feugate, Jr. and Steven M. McIntyre. *Introduction to VLSI Testing.* Prentice-Hall, Englewood Cliffs, N.J., 1988.

[23] C.V. Freiman. Optimal error detection codes for completely asymmetric binary channels. *Information and Control*, 5:64 – 71, 1962.

[24] H. Fujiwara and S. Toida. The complexity of fault detection: An approach to design for testability. In *Proceedings of the 12th International Symposium on Fault-Tolerant Computing*, pages 101 – 108, 1982.

[25] Hideo Fujiwara. *Logic Testing and Design for Testability.* Computer System Series. The MIT Press, Cambridge, Mass., 1985.

[26] Hideo Fujiwara and Shunichi Toida. The complexity of fault detection problems for combinational logic circuits. *IEEE Transactions on Computers*, C-31(6):555 – 560, June 1982.

[27] J. Galiay, Y. Crouzet, and M. Vergniault. Physical versus logical fault models MOS LSI circuits: Impact on their testability. *IEEE Transactions on Computers*, C-29(6):527 – 531, June 1980.

[28] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W.H. Freeman, New York, 1983.

[29] Prabhakar Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Transactions on Computers*, C-30(3):215 – 220, March 1981.

[30] J. Hartmanis and R.E. Stearns. *Algebraic Structure Theory of Sequential Machines.* Prentice-Hall International Series in Applied Mathematics. Prentice-Hall, Englewood Cliffs, N.J., 1966.

[31] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666 – 677, August 1978.

[32] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, N.J., 1985.

[33] M.Y. Hsiao and Dennis K. Cha. Boolean difference for fault detection in asynchronous sequential machines. *IEEE Transactions on Computers*, C-20(11):1356 – 1361, November 1971.

[34] Oscar H. Ibarra and Sartaj Salmi. Polynomially complete fault detection problems. *IEEE Transactions on Computers*, C-24(3):242 – 249, March 1975.

[35] J. Jacob and N.N. Biswas. GTBD faults and lower bounds on multiple fault coverage of single fault test sets. In *Proceedings of the International Test Conference*, pages 849 – 855, 1987.

[36] Mark G. Johnson. Private communication.

[37] S. Kamal. An approach to the diagnosis of intermittent faults. *IEEE Transactions on Computers*, C-24(5):461 – 467, May 1975.

[38] Zvi Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill Computer Science Series. McGraw-Hill, New York, 2nd edition, 1978.

[39] Y. Levendel and P.R. Menon. Fault simulation. In Dhiraj K. Pradhan, editor, *Fault-Tolerant Computing*, chapter 3. Prentice-Hall, Englewood Cliffs, N.J.,

1986.

[40] Wojciech Maly. Realistic fault modeling for VLSI testing. In *24th ACM/IEEE Design Automation Conference*, pages 173 – 180, 1987.

[41] Alain J. Martin. A delay-insensitive fair arbiter. Technical Report 5193:TR:85, California Institute of Technology, 1985.

[42] Alain J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 247 – 260, Rockville, Md., 1985. Computer Science Press.

[43] Alain J. Martin. The probe: An addition to communication primitives. *Information Processing Letters*, 20:125 – 130, 1985.

[44] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1:226 – 234, 1986.

[45] Alain J. Martin. Self-timed FIFO: An exercise in compiling programs into VLSI circuits. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, 1986.

[46] Alain J. Martin. A synthesis method for self-timed VLSI circuits. In *ICCD 87: 1987 IEEE International Conference on Computer Design*, pages 224 – 229. IEEE Computer Society Press, 1987.

[47] Alain 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*, Reading, Mass., 1989. Addison-Wesley.

[48] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, Cambridge, Mass./London, 1990. The MIT Press.

[49] Alain J. Martin. Asynchronous datapaths and the design of an asynchronous adder. In *Formal Methods in System Design.* Kluwer, 1992.

[50] Alain J. Martin, Steven M. Burns, T.K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The first asynchronous microprocessor: The test results. Technical Report Caltech-CS-TR-86-6, California Institute of Technology, 1989.

[51] Alain J. Martin, Steven M. Burns, T.K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Carlo H. Sèquin, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 350 – 373, Cambridge, Mass./London, 1990. The MIT Press.

[52] Alain J. Martin and Pieter J. Hazewindus. Testing delay-insensitive circuits. In Carlo H. Sèquin, editor, *Advanced Research in VLSI: Proceedings of the 1991 University of California/Santa Cruz Conference*, pages 118 – 132, Cambridge, Mass./London, 1991. The MIT Press.

[53] E.J. McCluskey and F. Buelow. IC quality and test transparency. In *Proceedings of the International Test Conference*, pages 295 – 301, September 1988.

[54] Carver Mead and Lynn Conway. *Introduction to VLSI Systems.* Addison-Wesley, Reading, Mass., 1980.

[55] Teresa H.-Y. Meng, Robert W. Brodersen, and David G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, CAD-8(11):1185 – 1205, November 1989.

[56] Melvin Ray Mercer, Vishwani D. Agarwal, and Carlos M. Roman. Test generation for highly sequential scan-testable circuits through logic transformation. In *Proceedings of the 1981 International Test Conference*, pages 561 – 565, 1981.

[57] Raymond E. Miller. *Switching Theory.* Wiley, New York, 1965.

[58] David E. Muller and W.S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium of the Theory of Switching*, volume 1, pages 204 – 243, Cambridge, Mass., 1959. Harvard University Press.

[59] W.M.C.J. van Overveld. On arithmetic operations with M-out-of-N-Codes. Technical Report 85/02, Department of Mathematics and Computing Science, Eindhoven Institute of Technology, 1985.

[60] Dhiraj K. Pradhan, editor. *Fault-Tolerant Computing.* Prentice-Hall, Englewood Cliffs, N.J., 1986.

[61] Gianfranco R. Putzulo and J. Paul Roth. A heuristic algorithm for the testing of asynchronous circuits. *IEEE Transactions on Computers*, C-20(6):639 – 647, June 1971.

[62] Sudhakar M. Reddy. Complete test sets for logic functions. *IEEE Transactions on Computers*, C-22(11):1016 – 1020, November 1973.

[63] J. Paul Roth. Diagnosis of automata failures: A calculus and a method. *IBM Journal of Research and Development*, 10(4):278 – 291, July 1966.

[64] J.P. Roth. *Computer Logic, Testing and Verification.* Computer Science Press, Woodland Hills, Calif., 1980.

[65] R.R. Schneider. On the necessity to examine d-chains in diagnostic test generation. *IBM Journal of Research and Development*, 11(1):114, January 1967.

[66] Charles L. Seitz. System timing. In *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, Reading, Mass., 1980.

[67] Frederick F. Sellers, M.Y. Hsiao, and L.W. Bearnson. Analyzing errors with the boolean difference. *IEEE Transactions on Computers*, C-17(7):676 – 683, July 1968.

[68] Sharad C. Seth and Vishwani D. Agarwal. Characterizing the LSI yield equation from wafer test data. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, CAD-3(2):123 – 126, April 1984.

[69] James E. Smith. On the existence of combinational logic circuits exhibiting multiple redundancy. *IEEE Transactions on Computers*, C-27(12):1221–1225, December 1978.

[70] J. Staunstrup and M.R. Greenstreet. Synchronized transitions. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design*, Amsterdam, 1990. Elsevier Science Publishers B.V.

[71] Tom Verhoeff. Delay-insensitive codes: An overview. Technical Report 87/04, Department of Mathematics and Computing Science, Eindhoven Institute of Technology, 1987.

[72] Michael J.Y. Williams and James B. Angell. Enhancing testability of large scale integrated circuits via test points and additional logic. *IEEE Transactions on Computers*, C-22(1):46 – 60, January 1973.