Algebraic Techniques For Finding Tests For Logical Faults In Digital Circuits.

ALGEBRAIC TECHNIQUES FOR FINDING TESTS FOR LOGICAL FAULTS IN DIGITAL CIRCUITS

by

Mark Joel Flomenhoft

A Dissertation

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Doctor of Philosophy

in

Electrical Engineering

Lehigh University

1973

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Dec. 10, 1973

Alfred K. Susskind
Alfred K. Susskind
Professor in Charge

Accepted ______________

Special committee directing the doctoral work of Mr. Mark J. Flomenhoft

Alfred K Susskind
Alfred K. Susskind
Chairman

Thomas F. Arnold
Thomas F. Arnold

William A. Barrett
William A. Barrett

[illegible]
Kenneth K. Tzeng

ACKNOWLEDGMENT

I wish to express deep appreciation to Professor Alfred K. Susskind for his role as dissertation supervisor. His encouragement and the many discussions we held were an invaluable assist to me in completing my research.

I would also like to thank the other members of my thesis committee -- Dr. Thomas F. Arnold, Dr. William A. Barrett, and Dr. Kenneth K. Tzeng -- for their helpful comments and suggestions.

Financial support by Bell Telephone Laboratories is gratefully acknowledged.

TABLE OF CONTENTS

LIST OF THEOREMS

LIST OF PROCEDURES

LIST OF FIGURES

ABSTRACT

Algebraic techniques are used to find tests for permanent logical faults in combinational and sequential digital circuits. Circuits are assumed to be specified solely as an interconnection of gates; flip-flops, feedback loops, state variables, etc., need not be identified. Also, no assumptions are made about reconvergent fanout, redundancy, or the initial state of a sequential circuit.

Fundamental to the procedures for combinational circuits is the SPOOF, an algebraic notation that expresses the topology of a network as well as its logical function. Both fault-free and faulty behavior are readily characterized by the SPOOF, and it is shown how to obtain (1) all the test vectors for any given regional fault, which alters the function of some subnetwork in an arbitrary but specified manner; (2) all the test vectors for any given "logical" short circuit, i.e., an unintended connection between signal leads that behaves like a wired logic AND or OR; (3) all the test vectors for any given shorted gate-input diode, a fault characteristic of certain logic families; and (4) all the test vectors for any given combination of stuck-at faults. Also considered are faults that either cause unintended output pulses or inhibit intended output pulses. Such faults, which have traditionally been called "undetectable", are detectable by "dynamic tests". It is shown how to modify the SPOOF so that dynamic tests can be found.

Sequential circuits are analyzed by a new algebraic method that employs the unit-gate-delay timing model. First a "static"

analysis determines all the steady-state combinations of values for the primary inputs and the gates. Then a "transient" analysis determines the maximum number of gate delays before a circuit attains a stable state after the input vector is changed. At the same time, those input changes that cause circuit oscillation under the unit-gate-delay timing model are identified. Using these results, the response of a circuit to sequences of any given length can be characterized, and both synchronizing sequences and test sequences are found. The tests obtained are independent of initial state and, like the synchronizing sequences obtained, are guaranteed not to cause circuit oscillation under the unit-gate-delay model. Included in the test generation method is a procedure for identifying at least some undetectable faults from the expressions for the response to a single input vector.

CHAPTER 1

INTRODUCTION

A fault-detection test for a digital circuit is a sequence of input combinations that enables one to determine whether or not a given unit on hand satisfies its intended logical specification. In the case of combinational logic, an "exhaustive" test -- one consisting of all possible input vectors -- will detect any fault that does not make the circuit non-combinational (i.e., either sequential or non-digital). However, for a circuit with $\underline{m}$ binary inputs the length of such a test is 2^m, which is unacceptably large in many instances.

The alternative to exhaustive testing is to find tests specifically to check for the presence of particular faults. For example, due to a fault a circuit lead may be permanently fixed ("stuck") in a logical state or a pair of leads may be unintentionally connected ("shorted"). By limiting the converage of a test to specified classes of faults, tests can usually be obtained that are orders of magnitude shorter than an exhaustive test. Although a non-exhaustive test is unlikely to detect every possible fault, experience suggests that the probability of a fault occurring that is undetected by such a test is small, provided only that the test does detect some "reasonable" class of faults -- every single stuck-type fault, for example.

1.1 Fundamental Assumptions and Additional Constraints

This dissertation presents algorithms for finding tests for faults in combinational and sequential digital circuits. In developing these algorithms, we have adhered to six fundamental assumptions:

1. We consider only binary digital circuits, which means that the logical signals are two-valued.
2. We consider only gate-type digital circuits, where by a "gate" we mean a circuit block that implements one of the functions AND, OR, NOT, NAND, and NOR, or one of these functions with some number of complementing inputs.
3. The only combinational circuits we consider are those that are loop-free.
4. The only sequential circuits we consider are those that are level-input (the circuit senses input levels rather than input pulses) and fundamental-mode (given that the inputs are held constant for a sufficient time, the circuit will reach a stable state).
5. We consider only permanent logical faults, which means that the effect of a fault is to transform one gate-type binary digital circuit (the "intended" or "fault-free" circuit) into another (a "faulty" circuit).
6. Our goal is a unified theory applicable to a variety of fault types.

Assumptions 1 through 4 make explicit that we are considering the type of logic circuitry used in contemporary digital systems. Assumptions one and two exclude types of circuitry now only of theoretical interest, such as multi-level logic and threshold logic.

Also, we do not investigate extensions of our algorithms for circuits with "wired" logic (gate outputs tied together to realize wired-AND's and wired-OR's) or unconventional gate types (such as bilateral transmission gates and "tri-state" gates) or for contact-type (relay) circuits. (Bilateral transmission gates, a feature of MOS technology, are sometimes used as bidirectional gates and as dynamic-memory flip-flops. Tri-state gates have three output states -- logic 0, logic 1, and "high impedance.")

The assumption of circuits that are loop-free combinational or fundamental-mode sequential is compatible with most actual circuits. Looped combinational logic has only recently been recognized as a possibility [1], and its occurrences are rare. The restriction to level-input fundamental-mode sequential circuits includes most all sequential circuits, the two types of which are synchronous ("clocked") and asynchronous ("unclocked"). However, circuits of each type can be found that are pulse-input or non-fundamental-mode, and for these the applicability of our algorithms has not been investigated.

The fifth assumption -- that faults are permanent (or "solid") -- assures that results obtained throughout a test sequence are consistent. We do not consider intermittent faults, which greatly complicate the testing problem. (For a discussion on intermittent faults, see [2].)

As a consequence of the goal of obtaining a unified theory applicable to a variety of fault types, we avoid approaches tailored to characteristics of special fault types. But then we know of no alternative to finding tests for one fault at a time.

Besides the assumptions above we impose four additional constraints that are, we believe, typical for a manufacturing environment:

1. A circuit is specified solely as an interconnection of gates. Supplementary descriptive information -- a state table, the designation of "state variables", etc. -- is assumed unavailable.
2. Changing the circuit to enhance testability -- adding auxiliary inputs or outputs or opening feedback loops -- is disallowed.
3. The given circuit may be "redundant" in the sense that there may exist no tests for some faults.
4. We desire "completeness" in the sense that if a test to detect a fault does exist, our procedures should produce it.

1.2 <u>Properties of Algebraic Techniques for Finding Tests</u>

All of our techniques for finding tests are based on Boolean algebra. Specifically, tests for a fault are the solutions to a Boolean equation of the form $E=1$, where E is a Boolean expression. Test generation procedures formulated in this way have several advantageous properties:

1. Trial-and-error techniques are avoided. For a given fault we simply determine the appropriate Boolean expression E and solve the equation $E=1$.

2. An undetectable fault is signified when E=1 has no solutions, i.e., when E is identically zero.
3. A unified theory applicable to a variety of fault types can be obtained.
4. All tests for a given fault are found simultaneously. These are the multiple solutions to the equation E=1.
5. The tests are "minimally-constrained" in the sense that only the inputs that must be set to particular values are specified.

1.3 Labeling the Circuit Leads

Logical faults are characterized by the way they affect logic signals at various locations in a circuit. It has been found convenient to denote locations in a circuit in terms of leads rather than gates. The scheme we use for labeling the leads, appropriate for sequential as well as combinational circuits, is as follows:

Procedure 1.0

1. Assign a distinct integer label to each primary input lead and gate output lead. After this is done the only unlabeled leads are fanout branches.
2. To each of the _k_ fanout branches from a lead labeled _p_ assign a distinct label of the form _pa_, where _a_ is one of the first k letters in some alphabet. Repeat this step until all the leads are labeled.

We have two reasons for the special labeling of fanout branches. First, tests are commonly designed to specifically check for the presence of a single fault. However, when tests are generated using a circuit model that does not faithfully represent the actual wiring topology, some real single faults may remain untested. For example, an open circuit along lead 9b in Fig. 1.1(c) is a failure that can be modeled as a single stuck-type fault. However, this failure is not represented by any single fault in the circuit models of Fig. 1.1(a) and 1.1(b). Thus a single-fault test based on either of these models does not necessarily test all real single faults. The lead-labeling scheme used here allows us, if we choose, to model circuit wiring accurately.

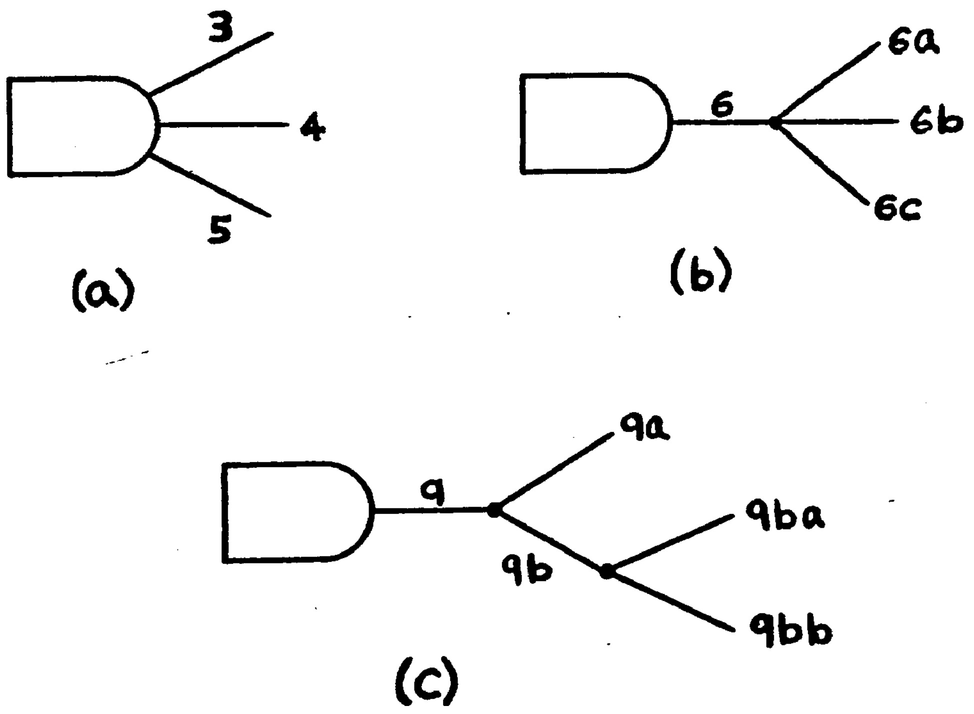


Figure 1.1. (a) Labeling that ignores fanout. (b) Labeling that defines fanout but ignores actual wiring topology. (c) Labeling that defines fanout and actual wiring topology.

Our second reason for explicitly labeling fanout branches is that finding tests for certain types of faults -- short circuits, for example -- requires the identification of fanout branches. Labeling the leads as we do thus simplifies test generation, which is described in Chapter 3.

1.4 Simply-Indistinguishable Single Stuck-at Faults

A single stuck-at fault is any malfunction that causes a single lead in a circuit to be permanently fixed in the 0-state ("stuck-at-0") or permanently fixed in the 1-state ("stuck-at-1"). It is popular to use the set of all single stuck-at faults as the class of faults for which tests are specifically found. We now show that, although a circuit with N leads has 2N possible single stuck-at faults, by inspecting the gate-level circuit diagram one can always reduce the number of faults for which tests need individually be found to about 1.2N.

As has been observed by Hayes [3] and Schertz and Metze [4], a set of indistinguishable single stuck-at faults is associated with every gate in a digital circuit. For example, if i is an input and p is the output of an AND-gate, then there is no test that can distinguish between i@0 and p@0. (We use the notation k@v to mean "lead k stuck-at-v".) Such faults will be called simply-indistinguishable, SI, and a test need be found for only one fault in each set of SI faults.

Consider a digital circuit with N leads. We partition the set of leads into three classes, as follows: S, the set of all leads that are inputs to single-input gates; M, the set of all leads that are inputs to multi-input gates; and Q, the set of all other leads -- the primary output leads and all leads that fan out. Let N_S, N_M, and N_Q denote the sizes of these sets. We show that, by inspecting the gate-level circuit diagram, the 2N possible single stuck-at faults can be merged into N_M+2N_Q SI sets.

Theorem 1.1

If G is a single-input gate, then associated with G are a pair of SI sets each consisting of a single stuck-at fault on the input lead and the output lead, and these sets can be specified.

Proof

Let G have input lead i and output lead p. If G is non-inverting [inverting], then i@0 is indistinguishable from p@0 [p@1] and i@1 is indistinguishable from p@1 [p@0]. Thus {i@0, p@0} and {i@1, p@1} [{i@0, p@1} and {i@1, p@0}] are SI sets.

Theorem 1.2

If G is a multi-input gate, then associated with G is an SI set consisting of a single stuck-at fault on each input lead and the output lead, and this set can be specified.

Proof

Let G have n inputs i,j,•••,m and output lead p. For the class of gates defined (AND, OR, NAND, or NOR, possibly with some inverting inputs), the logic-state of p is v for some particular input vector $(u_i,u_j,\cdots,u_m)$ and $\bar{v}$ for all the other 2^n-1 input vectors.

Then $k@\bar{u}_k$ is indistinguishable from $p@\bar{v}$ for every input k, and $\{i@\bar{u}_i, j@\bar{u}_j, \cdots, m@\bar{u}_m, p@\bar{v}\}$ is an SI set.

Definition

A digital circuit is fanout-free if it has a single primary output and every lead other than the primary output is a gate input.

Theorem 1.3

By inspecting the gate-level diagram of a fanout-free digital circuit, the 2N possible single stuck-at faults can be merged into N_M+2 SI sets, and these sets can be specified.

Proof

We account for the primary output faults by starting with two fault sets, each consisting of one of these faults. We account for all the other faults as follows:

Suppose G is a single input gate whose output faults have been accounted for. By Theorem 1.1, each input fault of G can be merged into an existing fault set. Thus we account for both of the input faults of G without adding any new fault sets.

Suppose G is a multi-input gate whose output faults have been accounted for. By Theorem 1.2, half the input faults of G can be merged into an existing fault set. Thus we account for all the input faults of G by adding one new fault set for each input.

Thus to account for all 2N faults, precisely N_M+2 fault sets are required. That these sets can be specified is evident, because the proof is constructive (it consists of a procedure).

Definition

A circuit C′ is a subcircuit of a digital circuit C if C′ can be obtained by deleting some number of leads and/or gates from C. Then C is said to contain C′; and if C′ is not identical to C, i.e., if the number of deletions is not zero, then the containment is proper. Also, a fanout-free subcircuit C′ of a digital circuit C is maximal if no other fanout-free subcircuit of C properly contains C′.

Theorem 1.4

The number of maximal fanout-free subcircuits of a digital circuit is precisely N_Q, the total number of primary output leads and leads that fan out.

Proof

Every primary output lead and every lead that fans out is the output of a maximal fanout-free subcircuit, and no other lead -- a gate input -- is the output of a maximal fanout-free subcircuit. Thus the number of maximal fanout-free subcircuits of a digital circuit is precisely N_Q.

Theorem 1.5

By inspecting the gate-level diagram of a digital circuit, the 2N possible single stuck-at faults can be merged into $N_M + 2N_Q$ SI sets, and these sets can be specified.

Proof

It is easily seen that the set of all maximal fanout-free subcircuits of a given digital circuit is a partition on that circuit.

Thus, by Theorems 1.3 and 1.4, the 2N possible single stuck-at faults can be merged into

$$\sum_{i=1}^{N_Q} (N_{Mi} + 2) = N_M + 2N_Q$$

SI sets. That these sets can be specified is evident, because the SI sets for each subcircuit can be specified.

Example 1.0

Figure 1.2 is the gate configuration for a simple network. The total number of leads N is 12, and N_M=8 and N_Q=3. Thus there are N_M+2N_Q=14 SI sets for this circuit. They are as follows (the lead-labeling is according to Procedure 1.0):

{1@0,2a@1,5a@0,6@0}

{2b@1,3@1,4@1,5@1}

{6@1,5b@0,7@1,8@1}

{5b@1,7@0}

All the other faults -- 1@1, 2@0, 2@1, 2a@0, 2b@0, 3@0, 4@0, 5@0, 5a@1, and 8@0 -- are in individual sets. The total number of sets is 14, as it must be for this circuit. Note that the "faults-to-leads" ratio is $\frac{14}{12}$, or about 1.2.

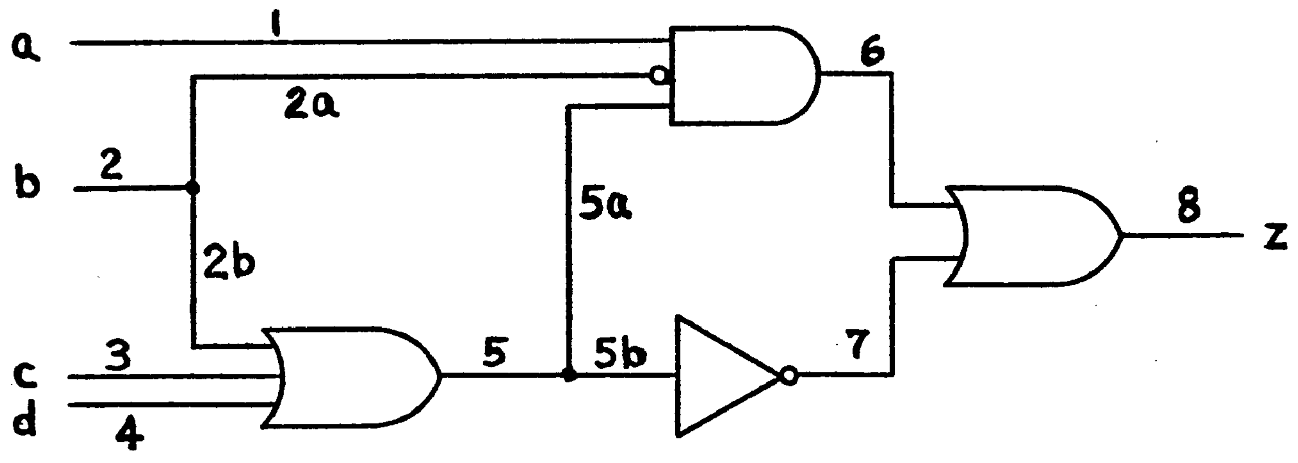


Figure 1.2. Circuit for examples.

Let us restate the result of Theorem 1.5 in the form of a "faults-to-leads" ratio R -- $R = \frac{N_M+2N_Q}{N}$. Using $N_M+N_Q = N-N_S$, we obtain $R = 1+\frac{N_Q-N_S}{N}$. Typically, R turns out to be about 1.2, which means that about 40% of the 2N single stuck-at faults can be eliminated. Note that this result applies to sequential as well as combinational circuits.

1.5 Outline of Following Chapters and Summary of Results

In Chapter 2 we define the SPOOF, an algebraic notation by which both fault-free and faulty behavior of loop-free logic networks can be characterized. It is proved that the SPOOF expresses the topology as well as the function of a network. Procedures are given for determining the logical function at any point within a circuit in terms of the primary input variables and for determining the output function in terms of any given set of lead variables and the input variables.

The procedures of Chapter 2 are the basis of techniques for algebraically finding tests under a variety of fault models. In Chapter 3 we show how tests can be obtained for stuck-at faults by an existing method called the "Boolean difference" and a new method we call the "direct difference". The two approaches are compared, and the direct difference is shown to be more efficient than the Boolean difference. Procedures are also given to find tests for several kinds of faults not of the stuck-at type: regional faults, each of which alters the function of some subnetwork in an arbitrary but specified manner; logical short circuits, unintended connections between signal leads that behave like wired-logic AND's or OR's; and shorts in gate-input diodes, a failure mode exhibited by certain logic families.

In Chapter 4 we consider faults that either cause unintended output pulses or inhibit intended output pulses. Such faults, which have traditionally been called "undetectable", are detectable by "dynamic tests". We show how the direct difference can be used to find dynamic tests when the SPOOF is modified to account for propagation delays.

Chapter 5 introduces a new method for analyzing level-input fundamental-mode sequential circuits. First we do a "static" analysis that determines all possible steady-state combinations of values for the primary inputs and the gates. Then we do a "transient" analysis that determines the maximum number of gate delays before a circuit attains a stable state after the input vector is changed. At the same time, those input changes that cause circuit oscillation under the unit-gate-delay timing model are identified. Using these results, we show how to characterize the response of a circuit to sequences of any given length. The problem of finding synchronizing sequences (initializing sequences independent of starting state) is an important application of the "sequence-response" expressions obtained, and examples are used to illustrate the power of the technique.

In Chapter 6 the algebraic techniques presented in Chapter 5 are used as the basis of a method for finding tests for individual faults. The tests obtained are independent of initial state and guaranteed not to cause circuit oscillation under the unit-gate-delay timing model. The algorithm is based solely on the gate-level circuit description, and no heuristic "loop-cutting" is involved.

CHAPTER 2

THE SPOOF

The structure and parity-observing output function, SPOOF, was originated by F. W. Clegg as an algebraic notation to characterize the topology as well as the function of a loop-free logic network [5]. (A similar notation was developed independently by S. C. Si [6].) This characterization is achieved by associating with the algebraic input literals path lists that contain interconnection and inversion-parity data. This chapter describes the construction of SPOOF's and details properties important for fault analysis. Although single-output networks are implicitly assumed, all the results are applicable to multi-output networks by treating the outputs individually.

2.1 Construction of Network SPOOF's

Whereas the algebraic variables in a conventional Boolean expression for a logic network are literals -- primary inputs and their complements -- the variables in a SPOOF are literals together with path lists. Each path list is a sequence of lead labels i,j,···,p such that, for every subsequence m,n of adjacent labels, either m is an input to a gate whose output is n or m is a lead that fans out into n. For lack of a better name we call a literal together with a path list a term.

As will be shown (Theorem 2.5), when the path lists designate inversion parity as well as the lead interconnections, then a network can be reconstructed from its SPOOF alone. The representation of inversion parity is accomplished by using two types of path list entries -- lead labels and their "complements" -- which for the lead label <u>k</u> are denoted k and $\bar{k}$, respectively.

<u>Definition</u>

The <u>complement of a path list entry</u> e [$\bar{e}$] is $\bar{e}$ [e]. The <u>complement of a path list</u> is obtained by complementing each entry in the list. The <u>complement of a term</u> is obtained by complementing both the input literal and the path list.

Let S_p be the SPOOF for lead p expressed in terms of primary input literals and in sum-of-products form. (Other standard forms -- product-of-sums, AND/exclusive-OR, etc. -- could be used as well.) The procedure for obtaining SPOOF's for the leads in a circuit is as follows:

<u>Procedure 2.1</u>

1. If p is the primary input lead for input variable x [$\bar{x}$], then S_p is the term x_p [$\bar{x}_p$].
2. Suppose G is a gate with input leads i,j,$\cdots$,m and output lead p. If $y_i,y_j,\cdots,y_m$ are the Boolean input-lead functions and G implements the function $f_G(y_i,y_j,\cdots,y_m)$, then S_p is $f_G(S_i,S_j,\cdots,S_m)$ expressed in sum-of-products form and with p appended to every path list. In performing algebraic manipulations, SPOOF terms are treated like literals in conventional Boolean algebra. However, terms with path lists that are different (non-identical and non-complementary) are to be considered distinct algebraic variables.

3. If p is a fanout branch from lead m, then S_p is S_m with p appended to every path list.

Example 2.1

Figure 1.2 is a gate configuration that realizes the function $z = a\bar{b} + \bar{b}\bar{c}\bar{d}$. The network SPOOF, S_8, is obtained as follows:

$S_1 = a_1$ [Rule 1 of Procedure 2.1]

$S_2 = b_2$ [Rule 1]

$S_{2a} = (S_2)_{2a} = b_{2,2a}$ [Rule 3]

$S_{2b} = (S_2)_{2b} = b_{2,2b}$ [Rule 3]

$S_3 = c_3$ [Rule 1]

$S_4 = d_4$ [Rule 1]

$S_5 = (S_{2b}+S_3+S_4)_5 = b_{2,2b,5} + c_{3,5} + d_{4,5}$ [Rule 2]

$S_{5a} = (S_5)_{5a} = b_{2,2b,5,5a} + c_{3,5,5a} + d_{4,5,5a}$ [Rule 3]

$S_{5b} = (S_5)_{5b} = b_{2,2b,5,5b} + c_{3,5,5b} + d_{4,5,5b}$ [Rule 3]

$S_6 = (S_1 \cdot \bar{S}_{2a} \cdot S_{5a})_6 = a_{1,6}\bar{b}_{\bar{2},\overline{2a},6}b_{2,2b,5,5a,6}$
$+a_{1,6}\bar{b}_{\bar{2},\overline{2a},6}c_{3,5,5a,6}+a_{1,6}\bar{b}_{\bar{2},\overline{2a},6}d_{4,5,5a,6}$ [Rule 2]

$S_7 = (\bar{S}_{5b})_7 = \bar{b}_{\bar{2},\overline{2b},\bar{5},\overline{5b},7}\bar{c}_{\bar{3},\bar{5},\overline{5b},7}\bar{d}_{\bar{4},\bar{5},\overline{5b},7}$ [Rule 2]

$S_8 = (S_6+S_7)_8 = a_{1,6,8}\bar{b}_{\bar{2},\overline{2a},6,8}b_{2,2b,5,5a,6,8}$
$+a_{1,6,8}\bar{b}_{\bar{2},\overline{2a},6,8}c_{3,5,5a,6,8}$
$+a_{1,6,8}\bar{b}_{\bar{2},\overline{2a},6,8}d_{4,5,5a,6,8}$ [Rule 2]
$+\bar{b}_{\bar{2},\overline{2b},\bar{5},\overline{5b},7,8}\bar{c}_{\bar{3},\bar{5},\overline{5b},7,8}\bar{d}_{\bar{4},\bar{5},\overline{5b},7,8}$

Rule 2 of Procedure 2.1 states that terms whose path lists are different are to be considered distinct algebraic variables. Thus if J and K are different path lists, all of the following are irreducible: $x_J x_K$, $x_J + x_K$, $x_J \bar{x}_{\bar{K}}$, and $x_J + \bar{x}_{\bar{K}}$. The full consequence of this rule is that no algebraic simplifications are possible in forming SPOOF's. We prove this result using induction on the level of gates in a circuit. Gate levels are determined as follows:

Procedure 2.2

1. Primary input leads and their fanouts are level zero.
2. Let G be a gate all of whose inputs are level k or less and with at least one level-k input. Then G, its output lead, and all its fanouts are level k+1.

Theorem 2.1

In constructing SPOOF's according to Procedure 2.1, no algebraic simplifications are possible.

Proof

Let G be a gate with input leads $i, j, \cdots, m$ and output lead p. Let $\tilde{S}_p$ be S_p with p omitted from the path lists. For the class of gates defined either $\tilde{S}_p = S_i^* S_j^* \cdots S_m^*$ or $\tilde{S}_p = S_i^* + S_j^* + \cdots + S_m^*$, where S_k^* is S_k or $\bar{S}_k$.

Suppose G is level one. Each SPOOF for the input leads of G is itself irreducible, for each consists of a single term. Each of the input leads of G is distinctly labeled (by Procedure 1.0), so in constructing $\tilde{S}_p$, hence S_p, according to Procedure 2.1, no algebraic simplifications are possible.

Suppose G is level k, k>1. By inductive hypothesis, each SPOOF for the input leads of G is itself irreducible. Then because the input leads of G are distinctly labeled, in constructing S_p

according to Procedure 2.1, no algebraic simplifications are possible.

Theorem 2.1 is of computational significance, for it means that one can omit trying to reduce SPOOF expressions by the application of algebraic identities. Furthermore, as a corollary, we conclude that a network SPOOF will, in general, contain a large number of terms compared to that in a conventional Boolean expression for the network. However, that the SPOOF completely characterizes the topology as well as the function of a network (to be shown in Section 2.2) has consequences which, we believe, more than compensate for this drawback.

<u>Theorem 2.2</u>

The expression obtained when all the path lists are removed from a network SPOOF is the Boolean network function, although not necessarily in reduced form.

<u>Proof</u>

In constructing a SPOOF we follow all the rules of ordinary Boolean algebra, differing only in that we augment literals with path lists and omit normally-allowed simplifications. Hence removing the path lists gives an expression for the Boolean network function.

A second useful function obtainable directly from a SPOOF is the "E-expression", a Boolean expression in which "literal conflicts" of the type $x\bar{x}$ and $x+\bar{x}$ are preserved and no other algebraic simplification is performed. Klayton and Susskind have shown that complete multiple-fault-detection test-sets for stuck-at faults can be determined from the E-expression for a network without considering fault patterns enumeratively [7]. Although we do not utilize E-expressions because we aim at a theory applicable to a variety of fault types, we include the following theorem for completeness.

Theorem 2.3

The expression obtained when all the path lists are removed from a network SPOOF and no simplifications are performed is the E-expression for the network.

Proof

This theorem follows immediately from Theorem 2.1 and the definition of E-expressions [7].

Example 2.2

For the circuit of Fig. 1.2 the SPOOF was obtained in Example 2.1. Removing the path lists from the SPOOF, S_8, gives the network E-expression:

$$E_8 = a\bar{b}b + a\bar{b}c + a\bar{b}d + \bar{b}\bar{c}\bar{d}$$

Also, the network Boolean function is

$$\begin{aligned} z &= a\bar{b}b + a\bar{b}c + a\bar{b}d + \bar{b}\bar{c}\bar{d} \\ &= a\bar{b} + \bar{b}\bar{c}\bar{d}. \end{aligned}$$

It may be observed that, as a consequence of our lead-labeling convention, SPOOF terms can be compacted by eliminating path list entries of leads that fan out. For example, the path list 2,2b,5,5a,6,8 from Example 2.1 is unambiguously represented by

2b,5a,6,8. Henceforth we will write path lists in this compacted form, understanding, however, that the complete list is implied.

Readers familiar with the historical development of fault analysis will have recognized the similarity of the SPOOF to earlier algebraic notations by Poage and Armstrong. Specifically, a network function written in terms of Poage's "element propositions" [8] contains precisely the same information as the SPOOF, although Poage's notational form is cumbersome. On the other hand, Armstrong's "equivalent normal form" [9], which has the same form as the SPOOF, does not indicate inversion parity in its path lists, and this is a critical shortcoming.

2.2 Properties of SPOOF's

In this section we prove that the SPOOF defines the topology as well as the function of a logic network. Then we present theorems that are the basis of procedures for algebraically finding tests under a variety of fault models.

Theorem 2.4

Given the output SPOOF of a gate, the function and input SPOOF's of the gate can be determined.

Proof

Let G be the gate with output lead p for which the SPOOF S_p is given. The last path list entry of every term of S_p is p. Let $\tilde{S}_p$ be S_p with p omitted from the path lists. Construct the set E consisting of the last entry in each path list of $\tilde{S}_p$. Suppose $E=\{i^*,j^*,\cdots,m^*\}$, where k^* is k or $\bar{k}$. Then the inputs of G are $i,j,\cdots,m$.

When G is a single-input gate and $E=\{i\}$ $[E=\{\bar{i}\}]$, then G is non-inverting [inverting], and S_i is $\tilde{S}_p$ $[(\overline{\tilde{S}_p})]$.

When G is a multi-input gate, either every product of $\tilde{S}_p$ contains all the elements in E or every product contains a single distinct element in E. (This is because either $\tilde{S}_p=S_i^* S_j^* \cdots S_m^*$ or $\tilde{S}_p=S_i^*+S_j^*+\cdots+S_m^*$. In the former case, $\tilde{S}_p$ can be written as $P_i P_j \cdots P_m$, where each P_k is a sum-of-products expression in which every term has last path entry k*. In the latter case, $\tilde{S}_p$ can be written as $P_i+P_j+\cdots+P_m$ with each P_k as above. Then G can be represented by an AND or an OR, respectively, with some number of inverting inputs. In either case, if $k \varepsilon E$ $[\bar{k} \varepsilon E]$, then the corresponding input of G is non-inverting [inverting], and S_k is P_k $[(\overline{P_k})]$.

Theorem 2.5

Given a network SPOOF, the gate-level topology of the network and each of the internal lead SPOOF's can be reconstructed.

Proof

Whenever we have the SPOOF for the output of a gate -- the common last path entry of every term is not a fanout branch -- we use Theorem 2.4 to determine the function and input SPOOF's of the gate. Whenever we have the SPOOF for a fanout branch, say S_q, then the SPOOF for the stem lead of q is $\tilde{S}_q$. Using these rules we work from the network output to the inputs, thus reconstructing the gate-level topology and each of the internal lead SPOOF's.

An ordinary Boolean output expression characterizes network behavior as a function of the primary input variables. However, in order to determine network behavior in the presence of some types of faults not of the stuck-at type, it is necessary to write the output expression as a function of input variables and certain lead variables. More precisely, we need the output function z in terms of the input variables and the lead variables $y_i, y_j, \cdots, y_m$ corresponding to the leads i,j,···,m of interest, treating each of these lead variables as an independent variable. Such an expression is the decomposition of z with respect to $y_i, y_j, \cdots, y_m$, which we denote as $z[y_i, y_j, \cdots, y_m]$. The SPOOF enables such functional decompositions to be obtained easily.

Theorem 2.6

The decomposition $z[y_i, y_j, \cdots, y_m]$ corresponding to leads i,j,···,m can be determined from the network SPOOF S_z by the following procedure:

Procedure 2.3

Suppose a term in the network SPOOF S_z has path list a*,b*,···,z. If no lead in this list is a lead of decomposition, then simply remove the path list from the literal. If any leads in the list are leads of decomposition, let k be that decomposition lead closest to the network output (k* appears after every other decomposition lead in the list). Then replace the term by y_k $[\bar{y}_k]$ when k* is unbarred [barred]. Applying this procedure to every term in S_z yields $z[y_i, y_j, \cdots, y_m]$.

Proof

Consider the SPOOF S_z' obtained for a network Z′ exactly like the given one Z, except that in Z′ the leads i,j,···,m are made primary input leads driven by the independent variables $y_i, y_j, \cdots, y_m$. Clearly, the output function z′ of Z′ is precisely the desired decomposition $z[y_i, y_j, \cdots, y_m]$.

Any path J in Z that does not include a decomposition lead is duplicated in Z′. Consequently, each term x_J^* in S_z must be present in S_z'.

Any path K in Z that includes decomposition leads is broken in Z′ at each decomposition lead. So the effect of input variable x due to path K on S_z is taken over by variable y_k in S_z', where k is that decomposition lead in K closest to the output. Consequently, for each term x_K^* in S_z there is a corresponding term $(y_K^*)_L$ in S_z', where L is that part of K from k* to the output. Furthermore, since a path entry k* is unbarred [barred] when there is an even [odd] number of inversions from k to the output along path L, then y_k must be unbarred [barred] when k* is unbarred [barred].

The given procedure follows precisely the above prescription, except that the path lists are removed so that z′, instead of S_z', is obtained.

Theorem 2.7

The output function z_F of a network when some pattern of stuck-at faults is present can be determined from the network SPOOF S_z by the following procedure:

<u>Procedure 2.4</u>

Suppose a term in S_z has path list $a^*,b^*,\cdots,z$. If no lead in this list is faulted, then simply remove the path list from the literal. If any leads in the list are faulted, let k be that faulted lead closest to the network output (k^* appears after every other faulted lead in the list). Then if k is stuck-at-v, replace the term by v [$\bar{v}$] when k^* is unbarred [barred]. Applying this procedure to every term in S_z yields z_F.

<u>Proof</u>

Let F be the fault $i@v_i,j@v_j,\cdots,m@v_m$. Clearly, z_F is $z[y_i,y_j,\cdots,y_m]$ with $y_i=v_i,y_j=v_j,\cdots,y_m=v_m$. This is precisely the effect of Procedure 2.4.

<u>Example 2.3</u>

For the circuit of Fig. 1.2 the SPOOF was obtained in Example 2.1. Applying Procedure 2.3 to determine $z[y_{5a}]$ gives

$$\begin{aligned} z[y_{5a}] &= a\bar{b}y_{5a}+ a\bar{b}y_{5a}+ \bar{a}by_{5a}+ \bar{b}\bar{c}\bar{d} \\ &= a\bar{b}y_{5a}+ \bar{b}\bar{c}\bar{d} \end{aligned}$$

Applying Procedure 2.4 to determine z_F for the fault 2b@0 gives

$$\begin{aligned} z_F &= a\bar{b}0 + a\bar{b}c + a\bar{b}d + 1\bar{c}\bar{d} \\ &= a\bar{b} + \bar{c}\bar{d} \end{aligned}$$

We note that Procedure 2.4 has previously been proved as Theorem 6.12 in [10] (also published as Theorem 6.1 in [11]), but by a completely different approach.

By making use of decomposition, an algebraic equation solutions to which are tests can always be formulated for any given fault. However, this equation usually depends on lead variables as well as input variables. In order to solve such an equation for actual test vectors, one must substitute the appropriate lead-variable functions expressed in terms of the input variables. Obtaining the necessary lead functions is another task easily accomplished when the network SPOOF is known.

<u>Theorem 2.8</u>

The function y_i for lead i can be determined from the network SPOOF S_z by the following procedure:

<u>Procedure 2.5</u>

1. Find some term x_J^* in S_z that has i* as a path entry. Call L that part of J from i* to the output.
2. Suppose $S_z = \sum_{i=1}^{q} P_i$. If in a product P_i no term contains L, replace P_i by 0. If in a product P_i some term contains L, remove the path lists from all such terms, and replace by 1 the terms in P_i that do not contain L.
3. If the first entry of L is i [$\bar{i}$], the resulting expression is y_i [$\bar{y}_i$].

<u>Proof</u>

Every term in S_z with a path entry i [$\bar{i}$] originates from S_i [$\bar{S}_i$], hence y_i [$\bar{y}_i$]. Because S_z is in sum-of-products form, a product in S_z that is independent of i [$\bar{i}$] contributes true vertices to S_z that may not be present in S_i [$\bar{S}_i$]. Such vertices are eliminated by replacing these products by 0's. In a product dependent on i [$\bar{i}$], the terms not containing i [$\bar{i}$] may make S_z false even though S_i [$\bar{S}_i$] is true. Replacing these terms by 1's ensures that the expression obtained will be true whenever S_i [$\bar{S}_i$] is true.

Because in constructing SPOOF's no algebraic simplifications are possible, the entirety of S_i [$\bar{S}_i$] is present in S_z for every path from i to the output that has an even [odd] number of inversions. If there are k paths from i to the output with even [odd] inversion-parity, then simply considering the presence of i [$\bar{i}$] in terms produces the function y_i [$\bar{y}_i$] k times. Such a procedure is correct, for the sum of k y_i's [$\bar{y}_i$'s] is y_i [$\bar{y}_i$]. The given procedure, by extracting y_i [$\bar{y}_i$] from the terms due to a single path L, eliminates this algebraic redundancy.

Example 2.4

For the circuit of Fig. 1.2, the SPOOF was obtained in Example 2.1. Applying Procedure 2.5 to determine y_{5a}, we select L=5a,6,8. Then

$$\begin{aligned} y_{5a} &= 1\cdot 1\cdot b + 1\cdot 1\cdot c + 1\cdot 1\cdot d + 0 \\ &= b + c + d \end{aligned}$$

CHAPTER 3

USING SPOOF'S TO FIND TESTS FOR SEVERAL FAULT TYPES

In this chapter we show how one can use the SPOOF as the basis of algebraic techniques for finding tests for a variety of fault types. In particular, we give procedures that determine (1) all the test vectors for any given combination of stuck-at faults; (2) all the test vectors for any given regional fault, a fault that alters the function of some subnetwork in an arbitrary but specified manner; (3) all the test vectors for any given "logical" short circuit, an unintended connection between signal leads that behaves like an AND or OR; and (4) all the test vectors for any given shorted gate-input diode, a non-stuck-at fault exhibited by certain logic families.

3.1 The Boolean-Difference Method for Finding Tests for Stuck-at Faults

The most widely known algebraic method for finding tests is an application of the "Boolean difference". If z is a network output function and y is some lead variable, then the Boolean difference of z with respect of y is

$$\frac{dz}{dy} = z_0 \oplus z_1$$

where z_v is the residue of z at y=v (z evaluated with y=v) and $\oplus$ is the "exclusive-OR" operator. (By definition, $a\oplus b=a\bar{b}+\bar{a}b$.) Sellers, Hsiao, and Bearnsen have shown that solutions of the equation $y \frac{dz}{dy} = 1$ [$\bar{y} \frac{dz}{dy} = 1$] are all the fault-detection tests for the fault y@0 [y@1] [12].

From the proof of Theorem 2.7, the residue z_v is precisely the function z_F for the fault y@v. Thus given a network SPOOF, the Boolean difference $\frac{dz}{dy}$ is readily obtained.

In order to solve the equation $y^* \frac{dz}{dy} = 1$ for actual test vectors, one must know, besides $\frac{dz}{dy}$, the lead function y in terms of the primary input variables. This function is also easily obtained from the SPOOF by Procedure 2.5.

Example 3.1

For the circuit of Fig. 1.2 the SPOOF was obtained in Example 2.1. Let us determine tests for the faults 5a@0 and 5a@1. Applying Procedure 2.4, we determine the residues required for the Boolean difference:

$$z_0 = a\bar{b}0 + a\bar{b}0 + a\bar{b}0 + \bar{b}\bar{c}\bar{d} = \bar{b}\bar{c}\bar{d}$$

$$z_1 = a\bar{b}1 + a\bar{b}1 + a\bar{b}1 + \bar{b}\bar{c}\bar{d} = a\bar{b} + \bar{b}\bar{c}\bar{d}$$

The Boolean difference is

$$\frac{dz}{dy_{5a}} = z_0 \oplus z_1 = (\bar{b}\bar{c}\bar{d}) \oplus (a\bar{b} + \bar{b}\bar{c}\bar{d})$$

$$= a\bar{b}c + a\bar{b}d$$

From Example 2.4 the lead function is

$$y_{5a} = b + c + d$$

Thus the stuck-at-0 tests are solutions of

$$y \frac{dz}{dy} = (b + c + d)(a\bar{b}c + a\bar{b}d)$$

$$= a\bar{b}c + a\bar{b}d = 1$$

There are three tests, namely the input vectors abcd = 1010, 1011, and 1001.

The stuck-at-1 tests are solutions of

$$\bar{y}\,\frac{dz}{dy} = (\bar{b}\bar{c}\bar{d})(a\bar{b}c + a\bar{b}d) = 1$$

This equation has no solutions, and the fault is undetectable.

It should be noted that when Boolean differences and lead functions are found from the SPOOF, there is no need to pre-store tables of functions, nor is reconvergent fanout a matter requiring special attention (cf. [13]).

Susskind has shown that the Boolean difference can be used to find fault-location tests as well as fault-detection tests: Solutions of the equation $y_1^* \frac{dz}{dy_1} \oplus y_2^* \frac{dz}{dy_2} = 1$ are all the tests that distinguish the faults $y_1@v_1$ and $y_2@v_2$, where y_i is unbarred [barred] when $v_i=0$ [$v_i=1$]. Susskind also described Boolean difference test generation for multiple faults [14]. In all cases, the necessary residues and lead functions can be found from the SPOOF.

3.2 The Direct-Difference Method for Finding Tests for Stuck-at Faults

If z is the intended output function of a network and z_F is its function when some fault F is present, then the fault-detection tests for F are all the input vectors that distinguish z_F from z, i.e., solutions of the equation $z \neq z_F$. As Even and Meyer have shown, the solutions of $z \neq z_F$ are precisely the same as those of $z \oplus z_F = 1$ [15]. We call $z \oplus z_F$ the direct difference, since it is a "direct" expression of the necessary test conditions. When F is a stuck-at fault, both z and z_F are readily obtained from the network SPOOF (by Theorem 2.2 and Procedure 2.4, respectively).

<u>Example 3.2</u>

For the circuit in Fig. 1.2 the SPOOF was obtained in Example 2.1. Let F be the fault 5a@0. We find

$$z = a\bar{b} + \bar{b}\bar{c}\bar{d}$$
$$z_F = a\bar{b}0 + a\bar{b}0 + a\bar{b}0 + \bar{b}\bar{c}\bar{d} = \bar{b}\bar{c}\bar{d}$$

The tests are solutions of

$$\begin{aligned} z \oplus z_F &= (a\bar{b} + \bar{b}\bar{c}\bar{d}) \oplus (\bar{b}\bar{c}\bar{d}) \\ &= a\bar{b}c + a\bar{b}d = 1 \end{aligned}$$

This is the same equation as was found by the Boolean-difference method in Example 3.1. For the fault 5a@1,

$$\begin{aligned} z_F &= a\bar{b}1 + a\bar{b}1 + a\bar{b}1 + \bar{b}\bar{c}\bar{d} \\ &= a\bar{b} + \bar{b}\bar{c}\bar{d} \end{aligned}$$

The direct difference is

$$z \oplus z_F = (a\bar{b} + \bar{b}\bar{c}\bar{d}) \oplus (a\bar{b} + \bar{b}\bar{c}\bar{d}) = 0$$

Thus the equation $z \oplus z_F = 1$ has no solutions, and as was concluded by the Boolean-difference method in Example 3.1, the fault is undetectable.

Defining the direct difference for fault-location tests is just as simple as it was for fault-detection tests: The tests that distinguish a fault F1 from a fault F2 are those input vectors for which z_{F1} and z_{F2} differ, i.e., solutions of the equation $z_{F1} \oplus z_{F2} = 1$.

Again, when F1 and F2 are stuck-at faults, z_{F1} and z_{F2} are readily obtained from the network SPOOF.

3.3 Comparison of the Boolean-Difference and Direct-Difference Methods

In this section we compare the Boolean-difference and direct-difference methods for finding tests. First we show that the two methods yield exactly the same results.

Consider tests for the fault y@0. Since z can always be written in the form $z=\bar{y}z_o \oplus yz_1$ and $z_F=z_o$ for this fault, we have

$$\begin{aligned} z\oplus z_F &= (\bar{y}z_o \oplus yz_1)\oplus z_o \\ &= (1\oplus\bar{y})z_o \oplus yz_1 \\ &= yz_o \oplus yz_1 \\ &= y(z_o \oplus z_1) \\ &= y\,\frac{dz}{dy} \end{aligned}$$

When the fault is y@1, one similarly obtains $z\oplus z_F = \bar{y}\,\frac{dz}{dy}$. Thus for the case of fault detection, precisely the same set of tests is obtained by either method.

Now consider fault-location tests. Using the result just obtained, we have

$$\begin{aligned} y_1^* \frac{dz}{dy_1} \oplus y_2^* \frac{dz}{dy_2} &= (z\oplus z_{F1})\oplus(z\oplus z_{F2}) \\ &= z_{F1} \oplus z_{F2} \end{aligned}$$

Again, the Boolean-difference and direct-difference expressions are equivalent.

Now we compare the computational effort required by the two methods. We assume for both cases that the network SPOOF has been found and that all required expressions are derived from it. Finding a fault-detection test by the Boolean-difference method requires the following:

1. Find the residues z_0 and z_1.
2. Find the Boolean difference $\frac{dz}{dy} = z_0 \oplus z_1$.
3. Find the lead function y^*.
4. Form $y^* \frac{dz}{dy}$.

Finding a fault-detection test by the direct-difference method requires the following:

1. Find the fault function z_F.
2. Form $z \oplus z_F$.

As was shown in Section 1.4, when y is the lead variable for an input to a multi-input gate, a test need be found for only one of the two possible stuck-at faults. Then the direct-difference method is clearly more efficient: Only one residue (z_F) has to be determined, and finding the lead function is obviated.

When y is the lead variable for a lead that is not a gate input, tests need be found for both of the possible stuck-at faults. Then the required computational efforts are about the same. Thus using the direct-difference method can only result in computational savings, the advantage being dependent on the actual gate-level circuit configuration. Comparing the two methods for the case of fault-location tests yields the same conclusion.

It is appropriate to note here that the complexity of computing exclusive-OR's can be greatly reduced by employing the following identity:

$$(a+b)\oplus(a+c) = \bar{a}(b\oplus c)$$

In calculating a direct difference $z\oplus z_F$, for example, any product independent of the faulty leads will be present in both z and z_F, and the complement of the sum of all such products can be "factored out" of the exclusive-OR.

3.4 Finding Tests for Regional Faults

A fault that alters the logic function of some subnetwork in an arbitrary but specified manner will be called a regional fault. For example, the accidental insertion of a NOR gate in place of a NAND is a regional fault. When the network SPOOF is known, the tests that detect a given regional fault can be found by the following procedure:

Procedure 3.1

Let the region (subnetwork) of interest have input leads $i,j,\cdots,m$ and output lead p. Suppose fault F causes the function of y_p on lead p to be $y_{pF}= f(y_i,y_j,\cdots,y_m)$. From the network SPOOF determine $y_i,y_j,\cdots,y_m$ in terms of primary input variables and the decomposition of the network output function $z[y_p]$. Then z_F is $z[y_p]$ with y_{pF}, expressed in terms of primary input variables, substituted for y_p. Tests are solutions of the equation $z\oplus z_F= 1$.

Example 3.3

In the circuit of Fig. 1.2, suppose the gate with output lead 6 behaves as a majority gate. Then the region-function in the presence of the fault is

$$y_{6F} = \text{maj}(y_1, y_{2a}, y_{5a}) = y_1 y_{2a} + y_1 y_{5a} + y_{2a} y_{5a}.$$

From the network SPOOF (Example 2.1)

$$y_1 = a$$

$$y_{2a} = b$$

$$y_{5a} = b + c + d$$

and $$z[y_6] = y_6 + \bar{b}\bar{c}\bar{d}$$

Thus $$y_{6F} = ab + a(b+c+d) + b(b+c+d)$$

$$= b + ac + ad$$

and $$z_F = b + ac + ad + \bar{b}\bar{c}\bar{d}$$

$$= a + b + \bar{c}\bar{d}$$

Tests for F are solutions of

$$z \oplus z_F = (a\bar{b}+\bar{b}\bar{c}\bar{d}) \oplus (a+b+\bar{c}\bar{d}) = b = 1$$

Regional faults are very general in nature, including for example, stuck-at faults as a subclass. Procedure 3.1 can be used to find tests in all cases; however, procedures "tailored" to more specific fault types are generally more efficient.

Susskind has described test generation for arbitrary single-gate regional faults by a Boolean-difference method [14]. We expect that the direct-difference formulation has a small computational advantage over that technique.

3.5 Finding Tests for Short Circuits

A short circuit is an unintended connection between signal leads in a network. It is likely that short circuits comprise the largest and most commonly-occurring class of faults not detected by tests designed specifically for single stuck-at faults.

Whereas stuck-at faults have the effect of simplifying the logic topology (some connections and even entire gates are effectively eliminated), short-circuits complicate the topology (connections and "gates" are introduced). It is possible, for example, that these new connections produce a loop in a normally loop-free network, and these loops complicate test generation because they can make circuit operation sequential. Because the SPOOF contains at least one term for each complete network path, such occurrences are easily identified: Suppose the short circuit is between leads i and j. Then a cycle is indicated if any path list has both i* and j* as entries. Although we exclude cycle-causing faults from consideration at this time, that we easily identify these cases is of some importance, for at least it is then known which faults must be treated by more sophisticated techniques.

Because our basis for network analysis is a logical model, we can characterize faults only in terms of their logical effect. Thus, we assume that a short circuit can be modeled as a wired-logic AND or OR at the site of the connection. This assumption is usually valid for logic families with passive output pull-up or pull-down (RTL, DTL, ECL), but may be questionable for families with active pull-up and pull-down (TTL). (For families like TTL perhaps tests should be found to cover both assumptions.)

Tests for short-circuit faults that do not introduce cycles are obtained as follows:

Procedure 3.2

Suppose F is the fault that causes leads i and j to be *-shorted (* denotes either AND or OR). If i or j is a fan-out branch, represent it by the gate output or primary input that drives it. (This accounts for "backward" propagation of signals.) Use the network SPOOF to determine the lead functions y_i and y_j and the decomposition of the network function $z[y_i,y_j]$. Then $y_{iF} = y_{jF} = y_i * y_j$, and z_F is $z[y_i,y_j]$ with y_{iF} and y_{jF}, expressed in terms of the primary inputs, substituted for y_i and y_j. Again, solutions of $z \oplus z_F = 1$ are the tests for F.

Example 3.4

In the circuit of Fig. 1.2, suppose leads 1 and 7 are AND-shorted. From the network SPOOF (Example 2.1), this fault does not cause a cycle. Then

$$y_1 = a$$
$$y_7 = \bar{b}\bar{c}\bar{d}$$
$$z[y_1,y_7] = y_1\bar{b}b + y_1\bar{b}c + y_1\bar{b}d + y_7y_7y_7 = y_1\bar{b}c + y_1\bar{b}d + y_7$$
$$y_{1F} = y_{7F} = y_1 \cdot y_7 = a\bar{b}\bar{c}\bar{d}$$

$$z_F = a\bar{b}\bar{c}\bar{d}\bar{b}c + a\bar{b}\bar{c}\bar{d}\bar{b}d + a\bar{b}\bar{c}\bar{d} = a\bar{b}\bar{c}\bar{d}$$

$$z \oplus z_F = (a\bar{b}+\bar{b}\bar{c}\bar{d}) \oplus (a\bar{b}\bar{c}\bar{d}) = a\bar{b}c + a\bar{b}d + \bar{a}\bar{b}\bar{c}\bar{d}$$

Input combinations 1010, 1011, 1001, and 0000 are tests for this fault.

3.6 Finding Tests for Shorted Gate-Input Diodes

The last type of fault to be considered is shorts in gate-input diodes, a failure mode of certain logic families such as DTL. Shorted input-diode faults have been studied by Chang in connection with fault simulation [16]. We assume networks composed of all NAND or all NOR gates from the same logic family, although our method can be applied in other cases as well.

To see how shorted gate-input diodes affect a network, consider the state of the circuit of Fig. 3.1 when a=1 and b=0. When the network is realized using DTL gates as illustrated in Fig. 3.2 and the diode for lead 1a (Fig. 3.1) is shorted, the input to the inverter (lead 1b) is a logic 0 instead of a 1. That is, the AND-ed level, usually blocked by the diode, "backward propagates" onto lead 1b. For a nominal b=0 voltage level of v_{CE}, the level at the input to the inverter is $v_{CE}+v_{BE}$ (the gate input voltage plus the voltage drop across the non-shorted input-diode), which is well below its threshold voltage of $2v_{BE}$. Observe that the output of the faulty gate is completely unaffected by the fault, and in Fig. 3.1 the fault can affect the network output only by virtue of the path through the inverter, i.e., the fanout on lead 1. In general, unless the input lead to the shorted diode is a fanout branch, the fault is of no consequence at all.

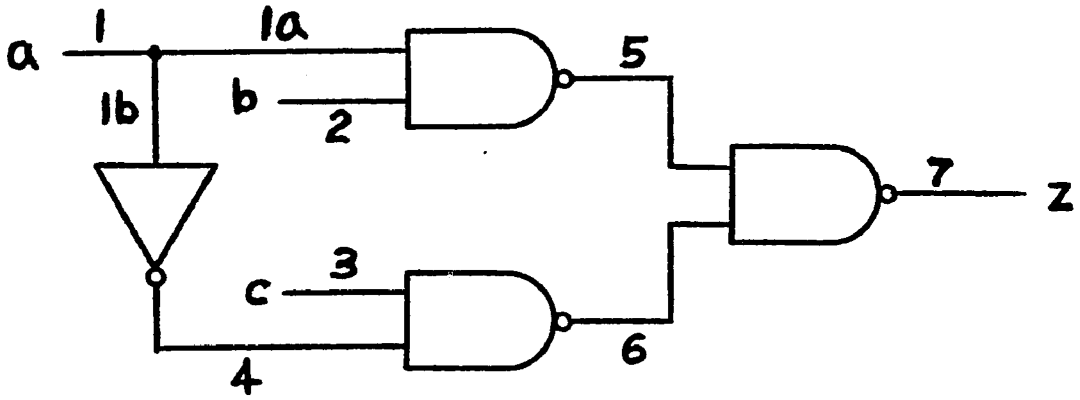


Figure 3.1.. Circuit to illustrate the effect of a shorted gate-input diode fault.

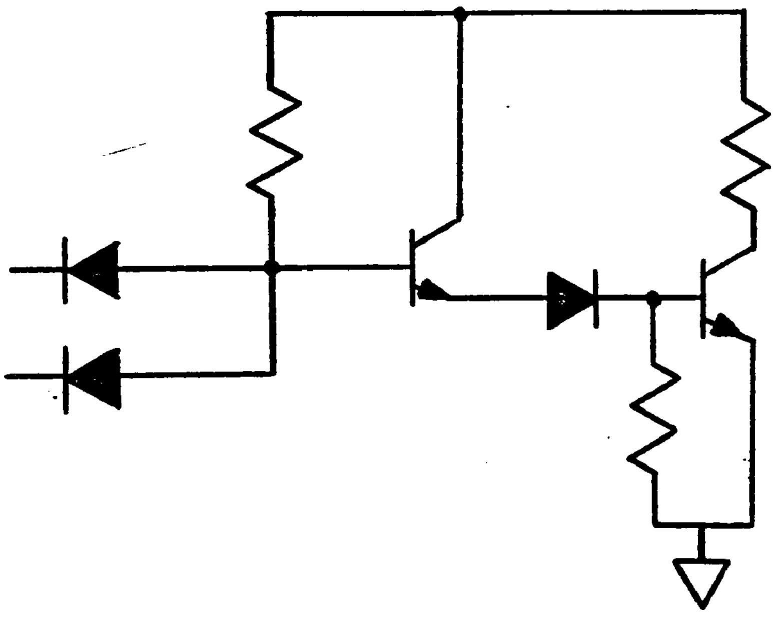


Figure 3.2 Typical DTL gate (shown is Motorola MC946).

In light of the preceding discussion, it follows that tests for shorted gate-input diodes are obtained as follows:

Procedure 3.3

Suppose i is some lead in a network whose SPOOF is S_z. First of all, we need to determine whether or not i is a gate input lead. (Obviously, if i is not a gate input, there is no shorted input-diode fault corresponding to i.) We do this as follows: Find some term in S_z that has a path entry i*. Let g* be the entry immediately to the right of i*. Then i is a gate input if and only if g is not a fanout branch, which is readily identified by our labelling convention.

Assuming i is a gate input, let F be the shorted input-diode fault corresponding to i. In order for F to be detectable, it is necessary that i is a fanout branch (recall that F is undetectable on gate output lead g). Let G be the gate to which i, a fanout branch, is an input. When the fault F occurs, signals on the other inputs of G will affect all the fanout branches connected to i. The inputs of G are determined as follows: In some term of S_z that contains path entry g*, let j* be the entry immediately to the left of g*. Then j is an input to G. If G is a single-input gate, F is undetectable. If G is a multi-input gate, call its inputs, i,j,•••,m. Also, let e be the gate output or primary input lead that branches into i. If the network contains a path from e to any input of G in addition to i, than F may cause oscillations or sequential operation (see [16]). The existence of such a path is indicated in S_z by the presence of a term with path entries p* and q* such that e=p and $q\epsilon\{j,\cdots,m\}$. When the acyclic assumption is valid, determine the lead functions $y_e, y_j, \cdots, y_m$ and the decomposition $z[y_e]$. Letting * denote either AND or OR (for NAND or NOR logic, respectively), we compute $y_{eF} = y_e * y_j * \cdots * y_m$, and z_F is $z[y_e]$ with y_{eF}, expressed in terms of primary input variables, substituted for y_e. Solutions of $z \oplus z_F = 1$ are the tests.

Example 3.5

For the circuit of Fig. 3.1, the only shorted gate-input diode fault that can affect network behavior corresponds to lead 1a. The network SPOOF is

$$S_7 = a_{1a,\bar{5},7} b_{2,\bar{5},7} + \bar{a}_{\overline{1b},4,\bar{6},7} c_{3,\bar{6},7}$$

Also
$$y_1 = a$$
$$y_2 = b$$
$$z[y_1] = y_1 b + \bar{y}_1 c$$
$$y_{1F} = y_1 * y_2$$
$$z_F = (a*b)b + \overline{(a*b)}c$$

For NAND gates, * = AND, so

$$z_F = (ab)b + \overline{(ab)}c = ab + c$$
$$z \oplus z_F = (ab+\bar{a}c) \oplus (ab+c) = a\bar{b}c$$

The fault is detected by the single input vector abc = 101.

3.7 Multi-Output Networks

Extending the direct-difference formulation to multi-output networks is easily done. From the network SPOOF's -- one for each output -- one obtains the fault-free functions $z_1, z_2, \cdots, z_q$ and the fault functions $z_{1F}, z_{2F}, \cdots, z_{qF}$. Tests for the fault are all the input vectors for which $z_i \neq z_{iF}$ for one or more i's, i.e., solutions of the equation $(z_1 \oplus z_{1F}) + (z_2 \oplus z_{2F}) + \cdots + (z_q \oplus z_{qF}) = 1$.

The direct-difference equation can be more succintly expressed in terms of the output-function vectors $\vec{z} = (z_1, z_2, \cdots, z_q)$ and $\vec{z}_F = (z_{1F}, z_{2F}, \cdots, z_{qF})$. Since two vectors are unequal if and only if at least one pair of corresponding components are unequal, $\sum_{i=1}^{q} (z_i \oplus z_{iF}) = \vec{z} \oplus \vec{z}_F$. Thus tests for the fault are the solutions of $\vec{z} \oplus \vec{z}_F = 1$.

CHAPTER 4

DYNAMICALLY-DETECTABLE FAULTS

In the literature on fault analysis, as in Chapter 3, a fault has been called undetectable when no input combination exists for which the output combinations of the faulty and fault-free networks differ. Also, networks with undetectable faults have been called redundant. For example, the networks N1 and N2 of Fig. 4.1 both realize the same truth-table. However,

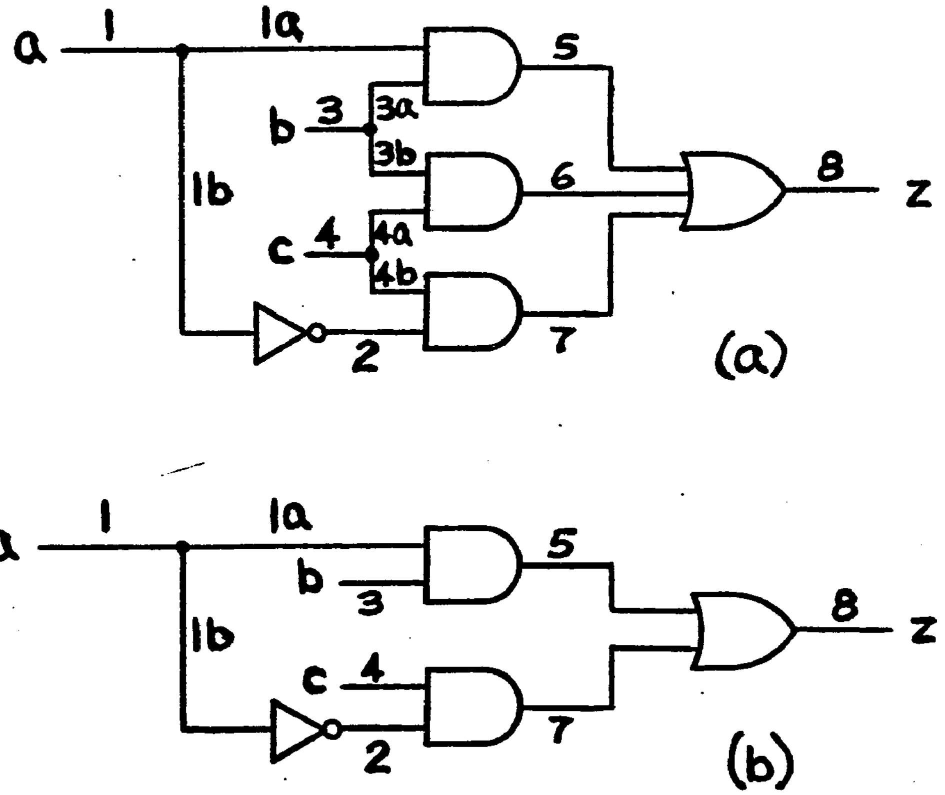


Figure 4.1, (a) Network N1. (b) Network N2.

N1 is redundant while N2 is not, for the stuck-at fault 6@0 in N1 is undetectable by conventional techniques, while all the stuck-at faults in N2 are detectable. On the other hand, when b=1 and c=1 and a goes from 1 to 0, N2 may, depending on the actual gate delays, output a 1-0-1 pulse. But N1 does not output a pulse regardless of its gate delays.

As is well known, network N2 possesses a hazard, while N1 is hazard-free. (A network possesses a hazard if there exists some input-state change and some combination of gate delays for which the network response is an unintended transient output sequence.) In general, hazard pulses are undesirable because, for example, they might set a flip-flop into the wrong state or cause an output indicator light to "blink" when it should not. We show how, when timing considerations are included, one can find "dynamic tests" that detect those faults (like 6@0) which cause a hazard-free network (N1) to behave like its equivalent-with-hazard (N2). Further, we show that the same technique can be used to find tests for one-shots and various pulse multipliers, circuits in which certain faults, also undetectable by conventional approaches, inhibit intended output pulses.

4.1 T-Expressions: Network Functions that Express Signal Delays

The dynamic gate model used here is that of an instantaneous (ideal) logic function with a pure time delay on each input. This delay can account for propagation delay along leads, as well as the delay of the gate itself. Delays are included in the SPOOF as follows:

If i is a gate input lead whose delay is d, then every appearance of i [$\bar{i}$] in the standard SPOOF is replaced by i(d) [$\bar{i}$(d)].

The term "SPOOF" used henceforth will always mean the form that includes delays.

Given the SPOOF for a network, we define the network <u>T-expression</u> as follows:

Replace each path list in the network SPOOF by the sum of the delays it contains. In the resulting expression, terms having different subscripts are to be considered distinct algebraic variables.

<u>Example 4.1</u>

In network N1 [Fig. 4.1(a)], let d=2 for the inverter input and d=3 for the other gate inputs. The network SPOOF is

$$S_8 = a_{1a(3)5(3)8}b_{3a(3)5(3)8} + b_{3b(3)6(3)8}c_{4a(3)6(3)8} + \bar{a}_{\overline{1b}(2)2(3)7(3)8}c_{4b(3)7(3)8}$$

Thus, the T-expression is

$$T_8 = a_6b_6 + b_6c_6 + \bar{a}_8c_6$$

The subscripts in a T-expression give the total propagation delay for each complete input-to-output path. That is, a term x_d [$\bar{x}_d$] follows input signal x [$\bar{x}$] but is delayed <u>d</u> time units. Dropping the subscripts, i.e., ignoring delay, gives the network function as an ordinary (static) Boolean expression.

4.2 Finding Tests for Dynamically-Detectable Faults

In Chapter 2 we showed how to determine the output function z_F of a network when some pattern of stuck-at faults is present. In the procedure for finding dynamic tests we make use of the output SPOOF S_{zF} for a network when some fault F is present. This variant of the SPOOF is obtained by applying the procedure for finding z_F (Procedure 2.4) except that path lists are retained instead of removed from terms independent of the faulty leads. Obtaining S_{zF} for other fault types is accomplished similarly.

When no static test exists for a given fault, we can look for dynamic tests as follows:

Procedure 4.0

1. From the network SPOOF determine S_{zF}.
2. Obtain the T-expressions T_z and T_{zF}, and form the direct difference $T_z \oplus T_{zF}$.
3. Let P be a product in the sum-of-products form of $T_z \oplus T_{zF}$ that contains a "conflict product" $x\bar{x}$. The following two cases exhaust all possibilities for a conflict product:
   a. There exist subscripts i and j, $i<j$, for which P contains $x_i\bar{x}_j$ but no x_g, $i<g<j$ and no $\bar{x}_h$, $h<j$. (The product $x_1x_2\bar{x}_3x_4\bar{x}_5$ is such a product, and here $i=2$ and $j=3$.) Then a 0-1 transition on x makes x_i go 0-1 after i time units, while $\bar{x}_j$ does not go 1-0 until after j time units. Thus

$x_i\bar{x}_j$ is 1 in the time interval (i,j) following the input change.

b. There exist subscripts i and j, $i<j$, for which P contains $\bar{x}_i x_j$ but no $\bar{x}_g$, $i<g<j$, and no x_h, $h<j$. Then a 1-0 transition on x makes $\bar{x}_i$ go 0-1 after i time units, while x_j does no go 1-0 until after j time units. Thus $\bar{x}_i x_j$ is 1 in the time interval (i,j) following the input change.

If P contains no conflict products due to primary inputs other than x, then all the other literals represent static inputs, and P can be made 1 during the interval (i,j). Since P is a product in a sum of products, the direct-difference can be made 1 for at least this interval, and a dynamic test has been found.

Note that the restriction to "simple" dynamic tests -- one input changing state once, all other inputs static -- is not essential to the procedure. However, more general dynamic tests are probably impractical because they require careful synchronization of the signal sources.

Example 4.2

In network N1 [Fig. 4.1(a)], suppose there is the fault 6@0. We find

$$z = ab + bc + \bar{a}c = ab + \bar{a}c$$

$$z_F = ab + \bar{a}c$$

Thus $z \oplus z_F = 0$, and no static test exists. However,

$$T_8 = a_6b_6 + b_6c_6 + \bar{a}_8c_6$$

$$T_{8F} = a_6b_6 + \bar{a}_8c_6$$

Then $$T_8 \oplus T_{8F} = (a_6b_6 + b_6c_6 + \bar{a}_8c_6) \oplus (a_6b_6 + \bar{a}_8c_6)$$

$$= \overline{(a_6b_6 + a_8c_6)}b_6c_6$$

$$= (\bar{a}_6a_8 + \bar{a}_6\bar{c}_6 + \bar{b}_6a_8 + \bar{b}_6\bar{c}_6)b_6c_6$$

$$= \bar{a}_6a_8b_6c_6$$

Thus a 1-0 transition on a with b=1 and c=1 is a simple dynamic test for this fault. The fault-indicating output pulse has a duration of two (8-6) time units and begins six time units after the input change.

Note that since z [z_F] can be obtained from T_z [T_{zF}], $z \oplus z_F$ is "contained" in $T_z \oplus T_{zF}$; so static and dynamic tests can be obtained from the same expression.

If delay variables, rather than actual numeric delays, are used in the SPOOF, then one can determine the range of delays for which the dynamic tests found are valid. This is best demonstrated by illustration.

Example 4.3

Network N1 [Fig. 4.1(a)] is shown in Fig. 4.2 with delay variables indicated. The SPOOF is

$$S_8 = a_{1a(A)5(H)8}\, b_{3a(B)5(H)8} + b_{3b(C)6(I)8}\, c_{4a(D)7(I)8} + \bar{a}_{\overline{1b}(G)2(F)7(J)8}\, c_{4b(E)7(J)8}$$

Thus the T-expression is

$$T_8 = a_{A+H}\, b_{B+H} + b_{C+I}\, c_{D+I} + \bar{a}_{G+F+J}\, c_{E+J}$$

For the fault 6@0 the T-expression is

$$T_{8F} = a_{A+H}\, b_{B+H} + \bar{a}_{G+F+J}\, c_{E+J}$$

The direct difference is

$$\begin{aligned} T_8 \oplus T_{8F} &= \overline{(a_{A+H}\, b_{B+H} + \bar{a}_{G+F+J}\, c_{E+J})}\, b_{C+I}\, c_{D+I} \\ &= \bar{a}_{A+H}\, a_{G+F+J}\, b_{C+I}\, c_{D+I} \\ &\quad + \bar{a}_{A+H} b_{C+I} \bar{c}_{E+J} c_{D+I} + a_{G+F+J} \bar{b}_{B+H} b_{C+I} c_{D+I} \\ &\quad + \bar{b}_{B+H} b_{C+I} \bar{c}_{E+J} c_{D+I} \end{aligned}$$

Thus a 1-0 transition on a̲ with b=1 and c=1 is a dynamic test provided (A+H)<(G+F+J). Alternatively, if (A+H)>(G+F+J), then a 0-1 transition on a̲ with b=1 and c=1 is a dynamic test.

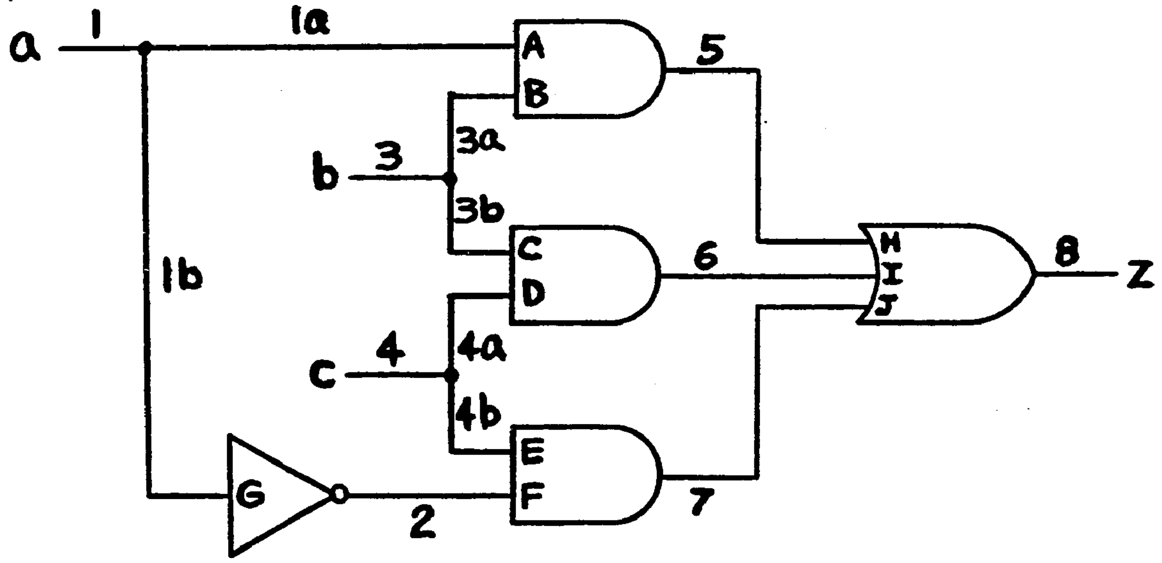
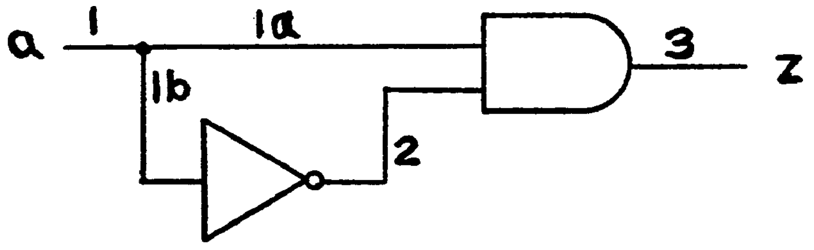
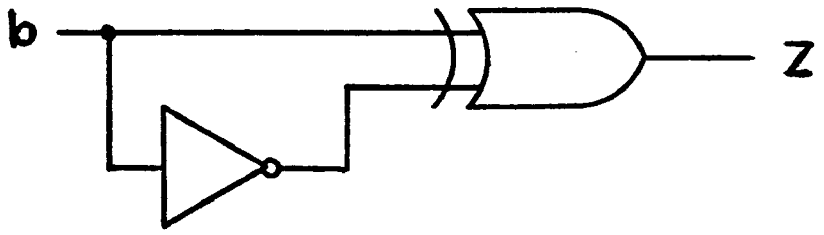


Figure 4.2. Network N1 with gate-input delay-variables indicated.

Dynamic analysis can also be used to characterize the fault-free and faulty behavior of one-shots and various pulse multipliers (see Fig. 4.3 for examples). These types of circuits fit the classification "loop-free logic networks", but they cannot be adequately described by ordinary Boolean algebra. We illustrate dynamic fault analysis for a one-shot with the following example:



(a)



(b)

Figure 4.3 (a) One-shot that outputs a 0-1-0 pulse when a makes a 0-1 transition.
(b) Pulse multiplier that outputs a 1-0-1 pulse each time b changes.

Example 4.4

In the one-shot of Fig. 4.3(a), let d=2 for the inverter input and d=3 for the AND-gate inputs. The SPOOF is

$$S_3 = a_{1a(3)3}\, \bar{a}_{\overline{1b}(2)2(3)3}$$

The fault 3@0 cannot be detected by any static test, for z=0 and z_F=0, so $z \oplus z_F = 1$ has no solutions. However,

$$T_3 = a_3\bar{a}_5$$
$$T_{3F} = 0$$
$$T_3 \oplus T_{3F} = a_3\bar{a}_5$$

Thus a going 0-1 is a simple dynamic test.

It might be expected that the dynamic-analysis approach described here can be used to evaluate fault-free networks for hazards. Indeed, this is precisely the technique fully described by McCluskey in [17].

CHAPTER 5

ALGEBRAIC ANALYSIS OF LEVEL-INPUT FUNDAMENTAL-MODE SEQUENTIAL CIRCUITS

Sequential circuits -- circuits whose output behavior depends on past as well as present inputs -- are of two principal types. In synchronous sequential circuits, SSC's, changes of the internal memory states can occur only when a special "clock" input is pulsed to sample the present input and internal states. In asynchronous sequential circuits, ASC's, the input state is sensed continuously, so changes of the internal states can occur whenever any of the inputs change.

Characterizing ASC's is more difficult than SSC's because time is a continuous rather than quantized parameter. Indeed, the only previous method for completely detailing the behavior of an ASC is the "state table," a matrix of 2^{m+n} n-tuples, where $\underline{m}$ is the number of binary input leads and $\underline{n}$ is the number of binary "state variables". For circuits of practical size, say $(m+n)>20$, constructing such a table from a gate-level network description is an immense undertaking. In this chapter we show how Boolean algebra alone can be used to obtain an alternative characterization. Besides requiring much less effort than state-table construction, to apply the algebraic procedure one does not have to identify state variables or feedback loops. Furthermore, the new procedure is based on the unit-gate-delay dynamic circuit model, the model most commonly employed in contemporary logic-circuit fault-simulators [18].

5.1 Level-Input Fundamental-Mode Sequential Circuits

The most common type of ASC is called "level-input fundamental-mode". Level-input means that the circuit senses input levels (static logic values continuous in time) rather than input pulses. (See [19] and [20] for descriptions of pulse-input ASC's.) Fundamental-mode means that, given that the inputs are held constant for a sufficient time, the circuit will reach a stable state, which means that every gate will remain in its present state until the input state of the circuit is changed.

A standard assumption for analyzing level-input fundamental-mode sequential circuits, LIFMSC's, is that input changes occur only when the circuit is in a stable state [21]. Our analysis procedure follows this assumption. Although in actual use circuit input levels may change before a stable state is reached, the assumption is usually valid when a circuit is tested for logical faults.

Another common assumption for LIFMSC's is that only one binary input changes state at a time. It turns out that for our algebraic-analysis procedures it is more convenient to make the more generous assumption that all arbitrary input-state changes are possible. However, the previous assumption that input changes occur only when a circuit is in a stable state implies that multiple changes must be simultaneous.

If an SSC is designed so that any clock-pulse width greater than a specified minimum is acceptable, then the SSC can be considered as an LIFMSC. Viewing an SSC in this way is especially useful for the purpose of test generation, for faults in an SSC may cause it to behave asynchronously, making tests based on the synchronous assumption of questionable validity.

In an LIFMSC it is common that not all 2^n combinations of the internal variables are possible stable internal states. For example, $y_1=y_2=0$ is not a stable internal state for the NAND latch shown in Fig. 5.1. We call A the set of all possible ("allowed") stable internal states of an LIFMSC. For the NAND latch A={01,10,11}. In Section 5.2 we show how to find A for any LIFMSC by a very simple algebraic formulation.

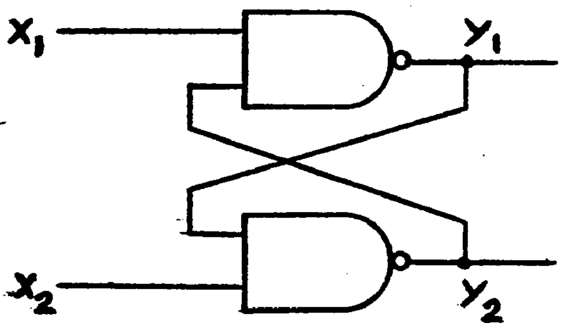


Figure 5.1 NAND latch.

When an LIFMSC is in a stable state and the input vector is changed, we call the combination of the new input state and the existing stable internal state before the input change a starting total state. The set of all possible starting total states consists of all 2^m input states, each combined with all of the internal states in A.

When an LIFMSC is put in a starting total state that is not a stable state, the circuit undergoes some sequence of internal-state transitions. If this state-transition sequence is finite and a deterministic stable state is eventually reached, we call the starting total state proper. On the other hand, due to "don't care" conditions in the design specification of the circuit or the presence of faults, certain starting total states may lead to an indeterminate stable state (due to a "critical race" [21]) or cause oscillation (loss of fundamental-mode operation). We call such starting total states improper. In Section 5.3 we give an algebraic procedure for finding the set P of proper starting total states and the set I of improper starting total states for any given LIFMSC under the assumption of equal gate delays. From the same procedure we also learn the transition time -- the length of the longest state-transition sequence over all proper starting total states. Using these results, we are able to characterize the response of a circuit to input sequences of any given length.

5.2 Stable-States (Steady-State) Analysis

Suppose some LIFMSC has m primary inputs and n gates. Let $\vec{x}=(x_1,x_2,\cdots,x_m)$ be the vector of the input variables and $\vec{y}=(y_1,y_2,\cdots,y_n)$ be the vector of the gate output variables. From a gate-level network description one can write the Boolean expression for a gate in terms of the input-lead variables of the gate. This expression is the next-state function for the gate under the unit-gate-delay dynamic circuit model. Let $\vec{Y}=(Y_1,Y_2,\cdots,Y_n)$ be the vector of these functions. The circuit is in a stable state if and only if $y_i=Y_i$ for every gate, i.e.,

$$\prod_{i=1}^{n} (y_i \odot Y_i) = 1$$

where $\odot$ is the "equivalence" operator. (By definition, $a\odot b = ab + \overline{a}\overline{b}$. Note that $a\odot b = \overline{a\oplus b}$.) Since two vectors are equal if and only if every pair of corresponding components are equal, this equation can be written as

$$\vec{y}\odot\vec{Y} = \prod_{i=1}^{n} (y_i\odot Y_i) = 1$$

We note that

$$\prod_{i=1}^{n} (y_i\odot Y_i) = 1$$

is the exact algebraic complement of the prescription

$$\sum_{i=1}^{n} (y_i\oplus Y_i) = 0$$

given by Even and Meyer for transforming a set of simultaneous equations into one equation [15].

Solutions of $\vec{y}\circledcirc\vec{Y}=1$ are all the stable total states of the circuit -- all possible steady-state combinations of values for the primary inputs and the internal leads. Of greater significance for our procedures, however, is an expression whose true vertices are the set A of all possible stable internal states independent of the primary inputs. (For notational simplicity, we denote both an expression and its set of true vertices by the same symbol. Whether we are referring to the expression or the set will always be clear from context.) Then, defining $\tilde{\lambda} = \vec{y}\circledcirc\vec{Y}$, the required expression A can be obtained by summing the residues of $\tilde{\lambda}$ taken over all possible input vectors. More simply, if $\tilde{\lambda}$ is in sum-of-products form, then A is obtained from $\tilde{\lambda}$ by replacing each input literal by 1.

<u>Example 5.1</u>

For the NAND latch of Fig. 5.1, the stable total states are all the solutions of

$$\begin{aligned}\tilde{\lambda} &= [y_1\circledcirc(\overline{x_1y_2})]\cdot[y_2\circledcirc(\overline{x_2y_1})]\\ &= (y_1\bar{x}_1+y_1\bar{y}_2+\bar{y}_1x_1y_2)(y_2\bar{x}_2+y_2\bar{y}_1+\bar{y}_2x_2y_1)\\ &= x_1\bar{y}_1y_2+x_2\bar{y}_2y_1+\bar{x}_1\bar{x}_2y_1y_2=1\end{aligned}$$

The possible stable internal states are the true vertices of

$$\begin{aligned}A &= 1\cdot\bar{y}_1y_2+1\cdot\bar{y}_2y_1+1\cdot1\cdot y_1y_2\\ &= y_1+y_2\end{aligned}$$

For this circuit $y_1=y_2=0$ ($\bar{A}=\bar{y}_1\bar{y}_2=1$) is not a stable internal state, as was pointed out previously.

5.3 Transition-Time (Transient) Analysis

We use the notation $y_i^{(k)}$ to denote y_i at k delay-times after an input change, and $\vec{y}^{(k)}=(y_1^{(k)},y_2^{(k)},\cdots,y_n^{(k)})$. An LIFMSC is stable with transition time t if t is the smallest integer such that $\vec{y}^{(t+1)}=\vec{y}^{(t)}$ over the set P of all proper starting total states. Before presenting a formal procedure for determining t and P, we describe how to find $\vec{y}^{(1)},\vec{y}^{(2)}$, etc.

When an LIFMSC is in stable internal state $\vec{y}$ and the input vector is changed to $\vec{x}$, then, under the assumption of equal gate delays,

$$y_i^{(1)} = Y_i(\vec{x},\vec{y})$$

Theorem 5.1

Under the assumption of equal gate delays, $y_i^{(k)}$ for k>1 is determined recursively as follows:

Procedure 5.1

$$y_i^{(k)}= y_i^{(j)}\Big|_{\vec{y}\leftarrow\vec{y}^{(k-j)}} \qquad \text{for any j such that } 1\leq j<k$$

(We use the notation $a\leftarrow b$ to mean "replace a by b" and the notation $c|_d$ to mean "c evaluated at d".)

Proof

The procedure is justified by considering an iterative-circuit realization for $\vec{y}^{(1)}, \vec{y}^{(2)}, \cdots, \vec{y}^{(k)}$. This circuit consists of k cells of n gates each. The i-th gate in every cell is the same in type as the gate with output y_i in the given LIFMSC, and the following interconnection rules make the output of the i-th gate in the j-th cell be $y_i^{(j)}$: If x_p is an input to the gate with output y_i in the LIFMSC, then x_p is an input to the i-th gate in every cell. (The input vector $\vec{x}$ is the same for all k because, by assumption, input changes do not occur during a state-transition sequence.) Also, if y_g is an input to the gate with output y_i in the LIFMSC, then y_g is an input to the i-th gate in ~~the first~~ cell, and the g-th gate in cell r drives the i-th gate in cell r+1, where $1 \leq r < k$. Figure 5.2 shows the cascade of three cells for the NAND latch of Fig. 5.1.

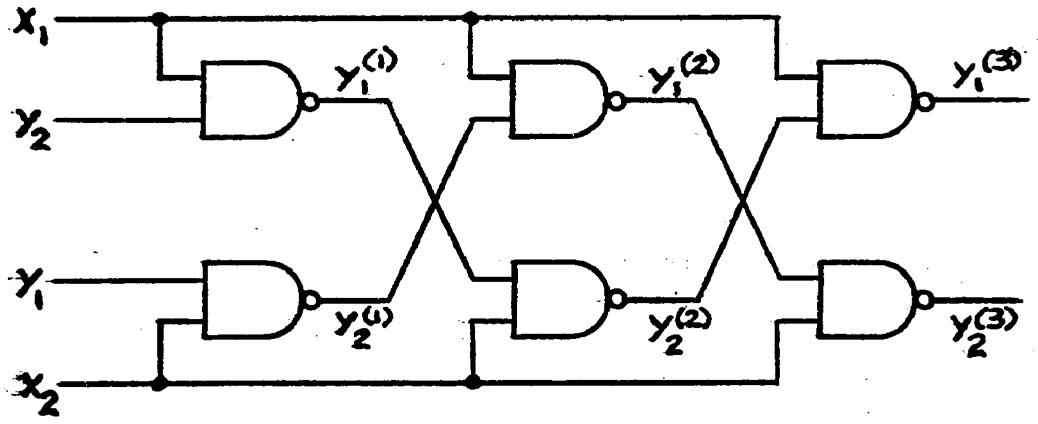


Figure 5.2. Cascade of three cells for the NAND latch.

Because the circuit is a uniform iterative cascade, the expressions for y_i taken over any _j_ cells are formally the same, differing only in the delay index (superscript). Thus we have

$$y_i^{(j)}(\vec{x},\vec{y})=y_i^{(k)}(\vec{x},\vec{y}^{(k-j)}) \qquad \text{for any } j \text{ such that } 1\leq j<k$$

Thus

$$y_i^{(k)}= y_i^{(j)}\Big|_{\vec{y}\leftarrow\vec{y}^{(k-j)}} \qquad \text{where } 1\leq j<k$$

It is instructive to compare the computational efforts for obtaining $y_i^{(k)}$ under different choices for _j_. At the extreme $j=1$,

$$y_i^{(k)}= y_i^{(1)}\Big|_{\vec{y}\leftarrow\vec{y}^{(k-1)}} ,$$

so it is necessary to know $y_i^{(1)}$ and the complete vector $\vec{y}^{(k-1)}$. On the other hand, for $j=k-1$,

$$y_i^{(k)}= y_i^{(k-1)}\Big|_{\vec{y}\leftarrow\vec{y}^{(1)}} ,$$

so it is necessary to know $y_i^{(k-1)}$ and the complete vector $\vec{y}^{(1)}$. When we need only a subset of the $y_i^{(k)}$'s, by choosing $j=k-1$ we obviate the finding of any unrequired $y_i^{(k)}$'s for $k>1$. Although not applicable at present, this observation can greatly reduce the work of finding tests, for example, where only the expressions for output gates are of consequence.

The functions $\vec{y}^{(1)}, \vec{y}^{(2)}$, etc., are used below in an algebraic "transient analysis" that determines the transition time and algebraic expressions for P and I, the sets of proper and improper starting total states. The procedure starts with P=A and I=0, thus assuming that all possible starting states are proper. (Note that P=A is an algebraic, not set, relation because the elements of set A are internal states, while the elements of set P are total states. Formally, set P initially contains all 2^m input states, each combined with all of the internal states in set A.)

The transition time is the smallest integer t such that $\vec{y}^{(t+1)}=\vec{y}^{(t)}$ over the subspace P. (Starting total states that are improper [contained in I] or impossible [contained in $\bar{A}$] are inconsequential.) The true vertices of $U_k = (\vec{y}^{(k+1)} \oplus \vec{y}^{(k)}) \cdot P$ are precisely the proper starting total states -- according to the current knowledge of P -- for which $\vec{y}^{(k+1)} \neq \vec{y}^{(k)}$.

Assume $U_1 \neq 0$ and $U_2 \neq 0$. The vertices for which $\vec{y}^{(3)}=\vec{y}^{(1)}$ and $\vec{y}^{(3)} \neq \vec{y}^{(2)}$ cause state-repeating cycles, which, with the input vector held constant, will repeat indefinitely. Starting total states that cause oscillations are improper and must be eliminated from P. The true vertices of $I_2 = (\vec{y}^{(3)} \odot \vec{y}^{(1)}) \cdot U_2$ are improper starting total states presently in P. (Recall that U_k is a subset of P.) If $I_2 \neq 0$, then we update P and I -- $P \leftarrow P \cdot \bar{I}_2$ and $I \leftarrow I + I_2$. If both $U_1 \leftarrow U_1 \cdot \bar{I}_2$ and $U_2 \leftarrow U_2 \cdot \bar{I}_2$ are still nonzero, we form U_3 and I_3. This process is continued until t is determined.

Theorem 5.2

Under the assumption of equal gate delays, the transition time and the sets P and I of proper and improper starting total states for an LIFMSC can be determined by the following procedure:

Procedure 5.2

```
1.  $k \leftarrow 1$.  $P \leftarrow A$.  $I \leftarrow 0$.
2.  $U_k = (\vec{y}^{(k+1)} \oplus \vec{y}^{(k)}) \cdot P = \sum_{i=1}^{n} (y_i^{(k+1)} \oplus y_i^{(k)}) P$
    If $U_k = 0$, the transition time of the circuit is $t = k$.
    If $U_k \neq 0$ and $k = 1$, then $k \leftarrow 2$ and repeat step 2.
3.  $I_k = \sum_{j=1}^{k-1} (\vec{y}^{(k+1)} \odot \vec{y}^{(j)}) \cdot U_k$
        $= \sum_{j=1}^{k-1} [\prod_{i=1}^{n} (y_i^{(k+1)} \odot y_i^{(j)})] U_k$
    If $I_k = 0$, then $k \leftarrow k+1$ and repeat step 2.
4.  $P \leftarrow P \cdot \bar{I}_k$.  $I \leftarrow I + I_k$.  $j \leftarrow 1$.
5.  $U_j \leftarrow U_j \cdot \bar{I}_k$.  If $U_j = 0$, the transition time of the circuit
    is $t = j$.
    If $U_j \neq 0$ and $j < k$, then $j \leftarrow j+1$ and repeat step 5.  If
    $j = k$, then $k \leftarrow k+1$ and repeat step 2.
```

Proof

The procedure is an exhaustive search. Termination is guaranteed to occur because there are only a finite number of distinct vectors $\vec{y}^{(k)}$.

Example 5.2

We apply Procedure 5.2 to the NAND latch of Fig. 5.1. From the circuit diagram,

$$y_1^{(1)} = \bar{x}_1 + \bar{y}_2$$
$$y_2^{(1)} = \bar{x}_2 + \bar{y}_1$$

From Example 5.1,

$$A = y_1 + y_2$$

Step 1 of Procedure 5.2: $k=1$, $P=y_1+y_2$, and $I=0$.

Step 2: $y_1^{(2)} = \bar{x}_1 + \bar{y}_2^{(1)} = \bar{x}_1 + x_2y_1$

$y_2^{(2)} = \bar{x}_2 + \bar{y}_1^{(1)} = \bar{x}_2 + x_1y_2$

$U_1 = [(y_1^{(2)} \oplus y_1^{(1)}) + (y_2^{(2)} \oplus y_2^{(1)})]P$

$= x_1\bar{x}_2y_1\bar{y}_2 + \bar{x}_1x_2\bar{y}_1y_2 + x_1x_2y_1y_2$

$U_1 \neq 0$, so $k=2$. Repeat step 2.

Step 2: $y_1^{(3)} = \bar{x}_1 + \bar{y}_2^{(2)} = \bar{x}_1 + x_2\bar{y}_2$

$y_2^{(3)} = \bar{x}_2 + \bar{y}_1^{(2)} = \bar{x}_2 + x_1\bar{y}_1$

$U_2 = [(y_1^{(3)} \oplus y_1^{(2)}) + (y_2^{(3)} \oplus y_2^{(2)})]P$

$= x_1x_2y_1y_2$

$U_2 \neq 0$.

Step 3: $I_2 = (y_1^{(3)} \odot y_1^{(1)})(y_2^{(3)} \odot y_2^{(1)})U_2$

$= x_1x_2y_1y_2$

$I_2 \neq 0$.

Step 4: $P \leftarrow P \cdot \bar{I}_2$

$P = \bar{x}_1 y_1 + \bar{x}_2 y_2 + y_1 \bar{y}_2 + \bar{y}_1 y_2$

$I \leftarrow I + I_2$

$I = x_1 x_2 y_1 y_2$

$j=1$.

Step 5: $U_1 \leftarrow U_1 \cdot \bar{I}_2$

$U_1 = x_1 \bar{x}_2 y_1 \bar{y}_2 + \bar{x}_1 x_2 \bar{y}_1 y_2$

$U_1 \neq 0$ and $j<k$, so $j=2$. Repeat step 5.

Step 5: $U_2 \leftarrow U_2 \cdot \bar{I}_2$

$U_2 = 0$, so the transition time is two.

This algebraic transient analysis should be vastly more efficient than constructing a state table starting from a gate-level network description. Three characteristics of the method are especially noteworthy:

1. The analysis is totally based on the gate-level network description, and no state variables or feedback loops need be identified.
2. All possible arbitrary input changes are assumed initially, and those that are improper under the unit-gate-delay model are identified.
3. The procedure is exhaustive but non-enumerative. All possible starting total states are considered simultaneously because the technique is algebraic.

5.4 Synchronous-Equivalent Circuits

If an LIFMSC has transition time t, then $y^{(t)}$ defines the next stable state in terms of the new input state and the existing stable internal state before the input change. We can interpret $y^{(t)}$ as defining an SSC whose next-state behavior is precisely the next-stable-state behavior of the given LIFMSC for all proper starting total states.

We have need to obtain algebraic expressions for the circuit response to sequences of input vectors. Let $\vec{y}$ be the initial stable internal state, $\vec{x}^j$ be the j-th input vector, and $\vec{y}^k$ be the stable internal state in response to $\vec{x}^1\vec{x}^2\cdots\vec{x}^k$. Then for an SSC that is the "synchronous-equivalent" of an LIFMSC, $\vec{y}^1$ is precisely $\vec{y}^{(t)}$, where t is the transition time, with $\vec{x}$, the "new" input vector in $\vec{y}^{(t)}$, renamed as $\vec{x}^1$.

For an SSC, expressions for the response to an input sequence of length $k>1$ can be found by a recursive procedure:

$$y_i^k = y_i^j \Bigg|_{\substack{\vec{x}^{r-k+j} \leftarrow \vec{x}^r \quad \text{for all } r \text{ such that } (k-j)<r\leq k \\ \vec{y} \leftarrow \vec{y}^{k-j}}}$$

This procedure for finding y_i^k is easily justified from an iterative-circuit realization for $\vec{y}^1, \vec{y}^2, \cdots, \vec{y}^k$. The argument is analogous to that given for justifying the prescription for $y_i^{(k)}$ (Procedure 5.1).

We note that the concept of synchronous-equivalent circuits was originated independently by Askhinazy [22] and Thayse [23]. However, instead of working with algebraic expressions, they each transformed the state table of an LIFMSC to that of a synchronous-equivalent SSC. Then a checking experiment -- a fault-detection test for an SSC based on its state table [23]-[25] -- can be found for the given LIFMSC. However, we believe such test sequences are excessively long and do not consider such techniques further.

5.5 Sequence-Response Expressions for LIFMSC's; Finding Synchronizing Sequences

Because an LIFMSC may be subject to critical races and oscillations whereas a synchronous-equivalent circuit is well-behaved under all possible input sequences, the sequence-response expressions defined in Section 5.4 may not properly characterize an LIFMSC. To ensure that only proper input sequences are admitted into our analyses, we must restrict the set of possible starting total states at each input-vector change to that allowed by P, the set of all proper starting total states. Algebraically, we accomplish this by "AND-ing" the expression P to the synchronous-equivalent sequence-response expressions at each input-vector step, where $\vec{x}$ is renamed as $\vec{x}^1$ in P (the input variables present in P as determined by Procedure 5.2 do not have any time-identifying superscript).

One must be careful to interpret correctly the sequence-response expressions restricted to P. In particular, the true vertices of $y_i^1 \cdot P$ $[\bar{y}_i^1 \cdot P]$ are proper combinations of input vectors and initial internal states that make $y_i^1 = 1$ $[y_i^1 = 0]$. But $y_i^1 \cdot P$ and

$\bar{y}_i^1 \cdot P$ are not complementary expressions. Indeed, consider the expression

$$(y_i^1 \cdot P) \circledcirc (\bar{y}_i^1 \cdot P) = \bar{P} = \bar{A}+I$$

where we use the identify $P=A \cdot \bar{I}$. We see that $y_i^1 \cdot P = \bar{y}_i^1 \cdot P$ for both impossible events ($\bar{A}$) and improper events (I).

This result has an appealing interpretation from the point of view of three-valued logic simulation. As Breuer has reported, many contemporary logic-circuit simulators allow three logic states -- 0, 1, and "don't know" [18]. One of the principal uses for don't-know's is to represent the outcomes of oscillations, which are assumed to terminate due to stray (non-ideal) delays but leave the circuit in an unknown state. Evidently, equivalence between $y_i^1 \cdot P$ and $\bar{y}_i^1 \cdot P$ (and, more generally, $y_i^k \cdot P$ and $\bar{y}_i^k \cdot P$) is the mechanism by which don't-know's are expressed in two-valued Boolean algebra.

As an application of sequence-response expressions, consider the problem of finding <u>synchronizing sequences</u>, SS's, input sequences that drive a circuit output into a unique final state regardless of the starting state of the circuit. When the sequence-response expression of an output is written as a sum-of-products, an SS is specified by any product that is independent of the internal (gate) variables. But SS's will never be apparent from sequence-response expressions restricted to P because every product in P necessarily depends on internal variables. (Recall that P is covered by A, i.e., $P \cdot A = P$; and every product in A consists of internal variables only.) We conclude that the present prescription for finding sequence-response expressions is incomplete.

In order to make SS's evident, the products in a sequence-response expression must be written in as few literals as possible. But as yet we have not utilized applicable "don't care" conditions to simplify these expressions: Recall that the true vertices of A are all the possible stable internal states of a circuit. Consequently, the event $\bar{A}=1$ is impossible, and the true vertices of $\bar{A}$ are don't-care's. Thus we can include $\bar{A}$ in a sequence-response expression without affecting the function where we "care." Algebraically, this is accomplished by "OR-ing" $\bar{A}$ after AND-ing P to the synchronous-equivalent sequence-response expressions. We then obtain algebraic simplifications of the type $\bar{a}b+b = a+b$.

We use the symbology $\tilde{y}_i^k$ $[\overset{=}{y}_i^k]$ to denote the sequence-response expressions for y_i in response to $\vec{x}^1\vec{x}^2\cdots\vec{x}^k$. The expressions $\tilde{y}_i^k$ and $\overset{=}{y}_i^k$ are determined as follows:

<u>Procedure 5.3</u>

1. $P \leftarrow P\big|_{\vec{x}\leftarrow\vec{x}^1}$.

2. $\tilde{y}_i^1 = y_i^{(t)}(\vec{x}^1,\vec{y})\cdot P+\bar{A}$

   $\overset{=}{y}_i^1 = \bar{y}_i^{(t)}(\vec{x}^1,\vec{y})\cdot P+\bar{A}$

   where t is the transition time.

3. $\tilde{y}_i^k = \tilde{y}_i^{k-1}\cdot P+\bar{A}$

   $\overset{=}{y}_i^k = \overset{=}{y}_i^{k-1}\cdot P+\bar{A}$

where $k>1$ and each literal in $\tilde{y}_i^{k-1}$ and $\bar{\bar{y}}_i^{k-1}$ is replaced as follows:

(1) $x_r^{j-1} \leftarrow x_r^{j}$

$\bar{x}_r^{j-1} \leftarrow \bar{x}_r^{j}$

for all r ($1 \leqslant r \leqslant m$, where m is the number of primary inputs) and all j such that $1<j \leqslant k$.

(2) $y_s \leftarrow \tilde{y}_s^{1}$

$\bar{y}_s \leftarrow \bar{\bar{y}}_s^{1}$

for all s ($1 \leqslant s \leqslant n$, where n is the number of gates).

Unless a sequence-response expression is written as the sum of <u>prime</u> implicants, SS's may be obscured by the apparent dependence of products on gate variables that are inessential. Furthermore, so that no SS's are overlooked, it is necessary that <u>all</u> prime implicants be included.

<u>Example 5.3</u>

Figure 5.3 is the gate configuration for a positive-edge-triggered type-D flip-flop. The static analysis gives

$$\begin{aligned}\lambda = {} & \bar{x}_1 x_2 y_1 y_2 y_3 \bar{y}_4 y_5 \bar{y}_6 \\ & + \bar{x}_1 x_2 y_1 y_2 y_3 \bar{y}_4 \bar{y}_5 y_6 \\ & + x_1 \bar{x}_2 y_1 \bar{y}_2 y_3 y_4 y_5 \bar{y}_6 \\ & + x_1 x_2 y_1 \bar{y}_2 y_3 \bar{y}_4 y_5 \bar{y}_6\end{aligned}$$

$$+ \bar{x}_1\bar{x}_2\bar{y}_1 y_2 y_3 y_4 y_5 \bar{y}_6$$

$$+ \bar{x}_1\bar{x}_2\bar{y}_1 y_2 y_3 y_4 \bar{y}_5 y_6$$

$$+ x_1\bar{y}_1 y_2 \bar{y}_3 y_4 \bar{y}_5 y_6$$

Then

$$A = \bar{y}_1 y_2 y_3 y_4 y_5 \bar{y}_6 + y_1 y_3 \bar{y}_4 y_5 \bar{y}_6$$

$$+ y_1 y_2 y_3 \bar{y}_4 \bar{y}_5 y_6 + y_1 \bar{y}_2 y_3 y_5 \bar{y}_6$$

$$+ \bar{y}_1 y_2 y_4 \bar{y}_5 y_6$$

From the transient analysis the transition time is three, and the next-stable-state expressions are

$$y_1^1 = \bar{x}_1 x_2 + x_1\bar{y}_2 + x_1\bar{y}_4$$

$$y_2^1 = \bar{x}_1 + \bar{x}_2\bar{y}_1 + \bar{y}_1\bar{y}_3$$

$$y_3^1 = \bar{x}_1 + \bar{y}_2 + \bar{y}_4$$

$$y_4^1 = \bar{x}_2 + x_1\bar{y}_1\bar{y}_3$$

$$y_5^1 = \bar{y}_2 + x_1\bar{y}_4 + \bar{x}_1\bar{y}_6$$

$$y_6^1 = \bar{y}_1\bar{y}_3 + \bar{x}_1\bar{y}_3 + \bar{y}_3\bar{y}_5$$

$$+ \bar{x}_1\bar{y}_5 + x_1\bar{x}_2\bar{y}_1$$

Also,

$$P = \bar{x}_1\bar{y}_1 y_2 y_3 y_4 y_5 \bar{y}_6 + \bar{x}_2\bar{y}_1 y_2 y_4 \bar{y}_5 y_6$$

$$+ y_1\bar{y}_2 y_3 y_5 \bar{y}_6 + y_1 y_3 \bar{y}_4 y_5 \bar{y}_6$$

$$+ \bar{x}_1\bar{y}_1y_2y_4\bar{y}_5y_6 + \bar{x}_2\bar{y}_1y_2y_3y_4y_5\bar{y}_6$$

$$+ y_1y_2y_3\bar{y}_4\bar{y}_5y_6 + \bar{y}_1y_2\bar{y}_3y_4\bar{y}_5y_6$$

Following Procedure 5.3 we obtain the sequence-response expressions for sequences of length one and determine all the prime implicants of $\tilde{y}_5^1$ (17 products) and $\bar{\bar{y}}_5^1$ (18 products). Neither of these expressions has any implicant independent of the gate variables, so there are no length-one SS's for the output.

From $\tilde{y}_5^2$ and $\bar{\bar{y}}_5^2$ one finds that $\bar{x}_1^1x_1^2x_2^1$ and $\bar{x}_1^1x_1^2\bar{x}_2^1\bar{x}_2^2$ are length-two SS's for $y_5=1$ and $y_5=0$, respectively. Note that $x_2^2=0$ in the SS for $y_5=0$, whereas x_2^2 is unspecified in the SS for $y_5=1$. In fact, $\bar{x}_1^1x_1^2\bar{x}_2^1x_2^2$ causes a circuit oscillation under the unit-gate-delay model and thus is not a valid SS. Only the proper SS $\bar{x}_1^1x_1^2\bar{x}_2^1\bar{x}_2^2$ is admitted by $\bar{\bar{y}}_5^2$.

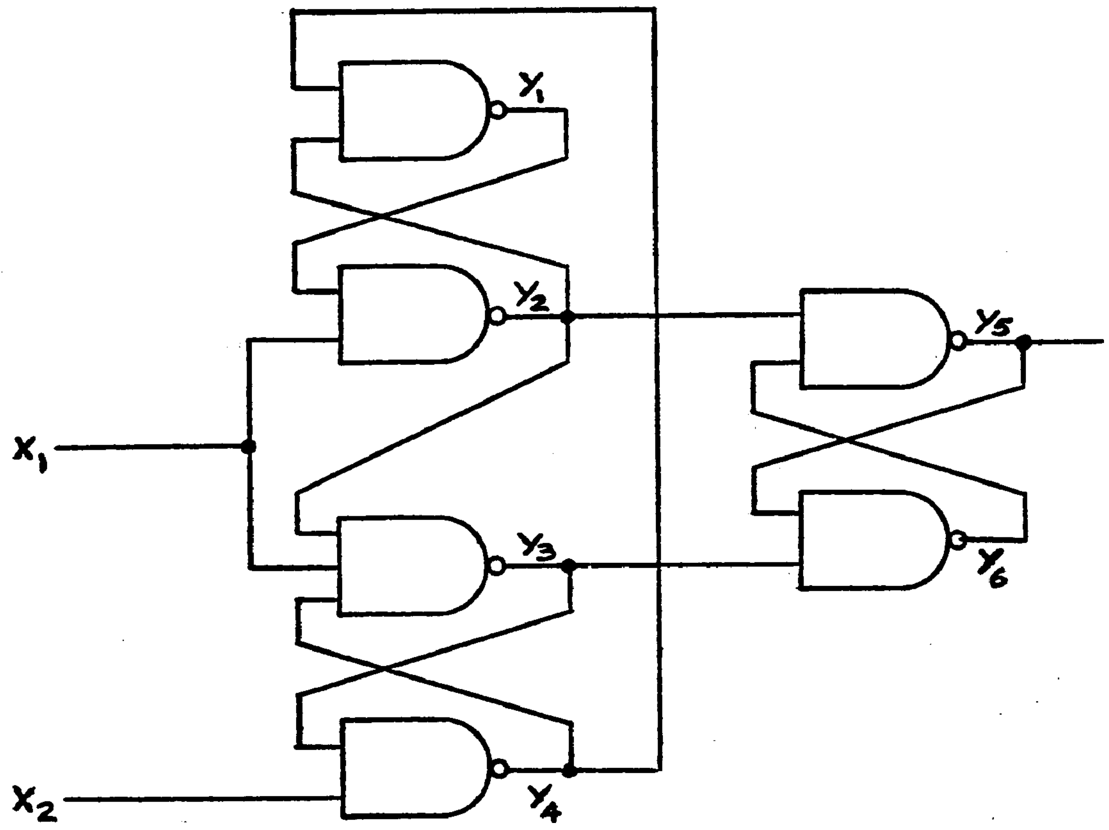


Figure 5.3. Positive-edge-triggered type-D flip-flop.

As an alternative to modifying synchronous-equivalent sequence-response expressions in two steps (AND-ing P, then OR-ing $\bar{A}$), let us consider the one-step operation of AND-ing $\bar{I}$ so that the expressions are restricted to vertices that are "not improper". But then, for example, $y_i^k \cdot \bar{I} = y_i^k \cdot P + y_i^k \cdot \bar{A}$ because of the identity $I = A\bar{P}$. Here, only certain of the don't-care's, namely those that coincide with y_i^k, are being allowed to contribute "one's" to the function. The resulting expressions are thus unnecessarily dependent on internal variables, and some of the SS's may not be apparent. Since $y_i^k \cdot \bar{I} + \bar{A} = y_i^k \cdot P + \bar{A}$, as long as one OR's $\bar{A}$, the choice of $\bar{I}$ or P is arbitrary, which is a satisfying result.

Another point of view from which the necessity of using both P and $\bar{A}$ (or $\bar{I}$ and $\bar{A}$) is sensible is as follows: The expressions A and P are "independent," one characterizing a static property of the circuit, the other characterizing a dynamic property. It is appealing that both are needed to properly characterize sequence-responses.

It was observed previously that equivalence between $y_i^k \cdot P$ and $\bar{y}_i^k \cdot P$ is the mechanism by which don't-know's are expressed in two-valued Boolean algebra. This conclusion is unaffected by the OR-ing of $\bar{A}$, for

$$(y_i^k \cdot P + \bar{A}) \odot (\bar{y}_i^k \cdot P + \bar{A}) = \bar{P} + \bar{A} = \bar{A} + I$$

where we use the identity $P = A \cdot \bar{I}$.

Although Procedure 5.3 is surely correct (necessary), we do not know how to prove it is complete (sufficient). However, the method has given complete results for the small number of circuits on which it has been tried. The following two examples detail the most interesting data obtained.

Example 5.4

Figure 5.4 is the gate configuration for a negative-edge-clocked master-slave JK flip-flop. The minimum-length SS for $y_3=1$ is found to be $\bar{c}^1 c^2 \bar{c}^3 j^2 \bar{k}^2$. (Intuition might lead one to expect that $c^1 \bar{c}^2 j^1 \bar{k}^1$ is an SS of length two. However, the input sequence

$$\bar{c}^1 c^2 \bar{c}^3 c^4 c^5 c^6 \bar{c}^7 j^2 j^3 j^4 j^5 j^6 j^7 \bar{k}^2 \bar{k}^3 \bar{k}^4 k^5 \bar{k}^6 \bar{k}^7,$$

which ends with the supposed SS displaced in time, leaves y_3 in the 0-state.)

That our method finds SS's for this circuit is significant, for no alternative technique other than exhaustive search with each possible starting state assumed is known to this author that also finds SS's. In particular, under three-valued logic simulation, in which the enumeration of possible starting states is avoided by assigning don't-know values initially, there apparently is <u>no</u> input sequence that takes this circuit out of the don't-know state. The reason for this failure is that the <u>j</u> and <u>k</u> inputs either enable or inhibit toggling (state complementation) of the flip-flop. But a don't-know, complemented or not, remains a don't-know.

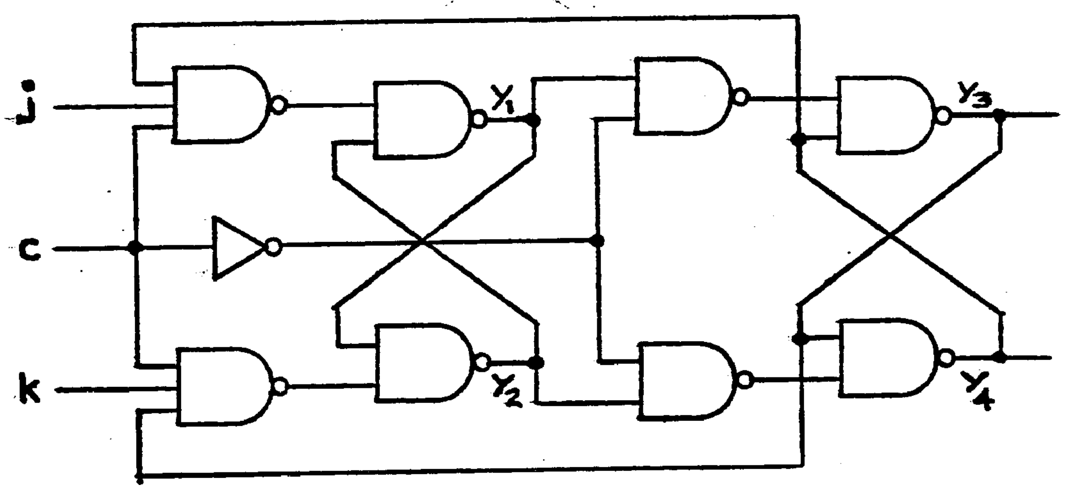


Figure 5.4. Negative-edge-clocked master-slave JK flip-flop.

Example 5.5

For the edge-triggered type-D flip-flop of Fig. 5.3, let us use $\underline{c}$ to designate x_1 and $\underline{d}$ to designate x_2. From Example 5.3 recall that $\bar{c}^1 c^2 d^1$ is a length-two SS for y_5=1. Let us investigate SS's of length three.

Since $\bar{c}^1 c^2 d^1$ is a length-two SS, the definition of SS's leads one to expect that any length-k sequence ending with $\bar{c}^{k-1} c^k d^{k-1}$ is also an SS. However, when one investigates $\tilde{y}_5^3$, one finds only the following four SS's, none of which allows an arbitrary input vector x^1:

$$\bar{c}^1 c^2 d^1$$
$$\bar{c}^1 \bar{c}^2 c^3 d^2$$
$$\bar{c}^1 c^3 d^1 d^2$$
$$\bar{c}^1 c^3 d^1 d^2$$

The reason the expected sequence $\bar{c}^2 c^3 d^2$ is not included is that this sequence permits the prefix $\bar{c}^0 c^1 \bar{d}^0 d^1$, where $\bar{c}^0 \bar{d}^0$ is a possible input state before $c^1 d^1$ is applied, and this prefix causes an oscillation under the unit-delay model. But Procedure 5.3 admits only input sequences that are proper throughout. Although it is likely that $\bar{c}^2 c^3 d^2$ will suffice in practice as an SS because stray delays in the actual circuit will cause the oscillation to terminate, stray delays violate the assumed dynamic model. Incidentally, $\bar{c}^1 c^2 d^1$ is a valid length-three as well as length-two SS because with c^2=1 there is no way the output can change at input-vector step three.

So far we have described SS's that drive a single output into a designated state. To obtain SS's that drive a set of gates into some combined state, one needs only to form the logical product of the appropriate sequence-response expressions. For example, to find an SS for the output state 110 of a given circuit, one investigates $z_1^{\sim k} z_2^{\sim k} z_3^{=k}$ for SS's. That is, we look for SS's while demanding that $z_1=1$, $z_2=1$, and $z_3=0$ simultaneously.

Kohavi and Winograd have shown that if a finite-state machine (SSC) has an SS, its length is at most $\frac{w(w+1)(w-1)}{6}$, where $\underline{w}$ is the number of states [27]. This bound similarly applies to LIFMSC's where $\underline{w}$ is now the number of elements in A, i.e., the number of stable internal states. If Procedure 5.3 can be proved to be sufficient as well as necessary, then we have an algorithm for determining minimum-length SS's: One looks for SS's of length one, then length two, etc., until either an SS is found or the bound is reached, in which case no SS exists.

5.6 Reducing the Number of Variables

The preceding discussions are all based on the unit-gate-delay timing model, and the analysis procedures include one algebraic variable for each gate in the circuit. For many circuits even a less precise timing model is sufficient for accomplishing our goals, and the consequent reduction in the number of variables is computationally important. We now suggest an alternative model that appears to be adequate for a large class of circuits.

From the gate-level logic diagram for a circuit, one can identify what we call "memory variables": the outputs of all NAND and NOR latches and other gates that are sources of feedback. Using only the output variables of these elements as the set of internal variables is equivalent to assuming that all elements but these have zero delay. Such a model preserves the principal timing relationships in a circuit while eliminating much of the detail. Although the selection of a minimum set of memory variables is a subtle issue, note that it is preferable to select too many variables rather than too few: The analyses may then be unnecessarily complicated, but at least they will be correct.

Example 5.6

When the above guidelines are applied to the JK flip-flop of Fig. 5.4, five of the nine gate variables are eliminated. The gate expressions remaining are the following:

$$y_1^{(1)} = jcy_4 + \bar{y}_2$$
$$y_2^{(1)} = kcy_3 + \bar{y}_1$$
$$y_3^{(1)} = \bar{c}y_1 + \bar{y}_4$$
$$y_4^{(1)} = \bar{c}y_2 + \bar{y}_3$$

It turns out for this circuit that the transient analyses based on this model and the complete one both predict that no input-state change causes indeterminate behavior. Furthermore, the SS's found from the simpler model are precisely those found from the complete model.

Note that since eliminating a gate variable is equivalent to assuming the gate has zero propagation delay, all of the procedures of this chapter actually apply to a circuit model of zero-delay and unit-delay gates.

CHAPTER 6

FINDING TESTS FOR LEVEL-INPUT FUNDAMENTAL-MODE SEQUENTIAL CIRCUITS

The algebraic techniques presented in Chapter 5 for analyzing LIFMSC's are used as the basis of a method for finding tests for individual logical faults. The approach is a simple extension of the direct difference, defined previously for combinational logic only. Included in the method is a procedure for identifying undetectable faults from the expressions for the response to a single input vector. However, it is not clear at this time whether this characteristic is merely a sieve or in fact all undetectable faults are recognized.

Some previous test generation strategies have required that a circuit can always be initialized to a specified "reset" state even in the presence of faults [28], [29]. However, such a presumption is often invalid, and we avoid making any assumption whatsoever about the initial state of the circuit.

6.1 The Direct Difference for LIFMSC's

If $\tilde{z}^k$ and $\approx{z}^k$ [$\tilde{z}_F^k$ and $\approx{z}_F^k$] are sequence-response expressions for the output of a fault-free [faulty] LIFMSC, then the solutions of

$$\sum_{j=1}^{k} (\tilde{z}^j \, \overset{\approx}{z}_F^j + \overset{\approx}{z}^j \, \tilde{z}_F^j) = 1$$

are fault-detection test conditions of length $\underline{k}$ or less for the fault F. Note that this "direct difference" cannot be expressed as the sum of exclusive-OR's because $\tilde{z}^1$ and $\approx z^1$, for example, are not complementary.

By a test sequence we shall mean an input sequence that, for all possible starting states, produces different output sequences depending on whether the circuit is faulty or not. Algebraically, this means that we accept as test sequences for a fault only those implicants of the direct difference that are independent of the initial values of the internal variables.

It is not surprising that the sets A and P of all possible stable internal states and proper starting total states, respectively, are fault-dependent. Indeed, this compels us to consider faults enumeratively, and we do not see how our techniques can be extended to handle all faults of a given type simultaneously, as the E-algorithm does for stuck-at faults in combinational logic [7].

The true vertices of $X^k = \tilde{z}^k \overset{\approx}{z}^k_F + \overset{\approx}{z}^k \tilde{z}^k_F$, are input sequences of length k for which the k-th output signals of the fault-free and faulty circuits differ. It might seem that examining this expression for k=1,2, etc., would be a sufficient procedure for finding a minimum-length test sequence, if one exists. In fact, for certain faults that are detectable, this procedure fails to find tests of any length. To illustrate this point, consider the fault 1@1 in the edge-triggered type-D flip-flop, which is shown in Fig. 6.0 with the leads labeled. (Note that the lead-labeling scheme used for combinational circuits -- Procedure 1.0 -- applies to sequential circuits as well.) Let us hypothesize that a test sequence for this fault is $\bar{c}^1 c^2 \bar{c}^3 c^4 d^1 d^2 \bar{d}^3 \bar{d}^4$, which for the fault-free circuit has the effect "load a 1, then load a 0". In the faulty circuit, the clock input is stuck-at-1, and the output does not change from its initial value. Thus, if the circuit starts in the 0-state, the fault is detected at input-vector step two, when the output should be logic 1; and if the circuit starts in the 1-state, the fault is detected at input-vector step four, when the output should be logic 0. We conclude that the sequence is a test, but we cannot say a priori at what step the fault is detected. It should be clear that for this circuit no implicant of X^k is independent of the initial-state variables for any k.

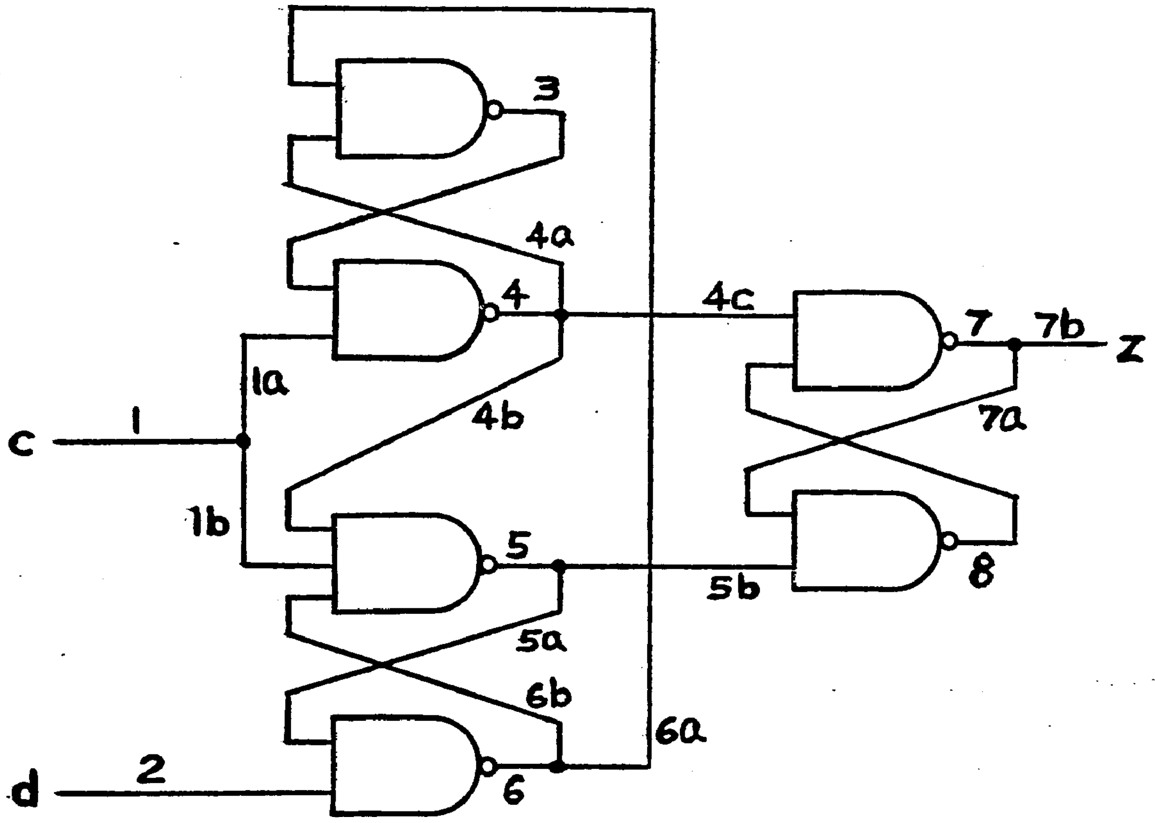


Figure 6.0. Positive-edge-triggered type-D flip-flop.

This failure to find tests for faults that are in fact detectable is remedied by formulating the direct difference so that it displays sequences that detect a fault <u>anywhere</u> in the test, rather than demanding that a fault be detected precisely at the <u>last</u> step. That is, by making the direct difference $\sum_{j=1}^{k} X^j$ we allow for detection at any step in the sequence.

<u>Example 6.1</u>

For the fault 1@1 in the edge-triggered type-D flip-flop of Fig. 6.0, there are no test sequences of length one, two, or three. From the prime implicants of the direct difference for sequences of length four, we find the two tests

$$\bar{c}^1 c^2 \bar{c}^3 c^4 d^1 \bar{d}^3 \bar{d}^4$$

and

$$\bar{c}^1 c^2 \bar{c}^3 c^4 \bar{d}^1 \bar{d}^2 d^3$$

Note that in both cases the improper input subsequence $\bar{c}^i c^{i+1} \bar{d}^i d^{i+1}$ is avoided.

Not only have all the minimum-length test sequences been found (as can be determined by exhaustive search), but no alternative technique other than exhaustive search with each possible starting state assumed is known to this author for obtaining this result. In particular, neither test sequence would be recognized as a test by unit-gate-delay three-valued-logic fault simulators because the faulty circuit never leaves the initial don't-know state. In such cases, the results of simulation are misleading, and thus simulation to "verify" tests can very well produce confusion.

6.2 Computational Considerations

Formulating the direct difference as $\sum_{j=1}^{k} X^j$ has unexpected computational benefits. First, it has been observed from examples tried that $\sum_{j=1}^{k} X^j$ may be a "simpler" expression (expressable as the sum of fewer products) than X^k itself. For example, for the fault 1@1 in the edge-triggered type-D flip-flop, after all possible applications of the Boolean identity $ab\bar{c} + ac = ab + ac$ have been performed, X^4 consists of 158 products, while $\sum_{j=1}^{4} X^j$ consists of just 96 products. This simplification occurs because each X^j is independent of X^r for $r>j$, and so it is likely that many products in X^2 are covered by those in X^1, that many products in X^3 are covered by those in $X^1+ X^2$, etc. (A product P "covers" a product Q if P+Q=P or, equivalently, Q·P=Q.) It is reasonable to expect that the effort required to find all the prime implicants, from which test sequences are evident, is related directly to the complexity of the direct-difference expression.

Perhaps a fact of even greater computational significance is that one need not find prime implicants at every step to be sure of finding all the tests. This result follows from the algebraic property that if P is a prime implicant of $\sum_{j=1}^{k} X^j$, then some cover of P (perhaps P itself) is a prime implicant of $\sum_{j=1}^{r} X^j$ for $r>k$. Thus in going to longer input sequences we cannot "lose" tests, and it makes sense to determine prime implicants only at selected input-sequence lengths.

6.3 An Upper Bound on the Length of Test Sequences

We derive an upper bound on the length of a test sequence to detect a given fault. The argument is based on concepts from the theory of finite-state machines [30], [31]. The bound obtained in this way is easily adapted to arrive at a bound for LIFMSC's.

A finite-state machine is a mathematical model for an SSC. The model consists of a finite set Q of internal states $q_1,q_2,\cdots,q_n$; a finite set X of input symbols $x_1,x_2,\cdots,x_m$; a finite set R of output symbols $r_1,r_2,\cdots,r_p$; and two mappings -- the next-state function $N:X\otimes Q\rightarrow Q$ and the next-output function $Z:X\otimes Q\rightarrow R$. [We use the notation $A\otimes B$ to mean the "Cartesian product of A by B", i.e., the set of all ordered pairs (a,b), where a is some element of a set A and b is some element of a set B.] If q is the present state and x is the present input symbol, then N(x,q) is the next state, and Z(x,q) is the next output symbol.

If T is an input sequence, the terminal state when T is applied and the starting state is q is called the T-successor of q. It is convenient to extend the definition of N so that T-successors can be denoted. Thus if $T=x^1x^2\cdots x^L$ and $T^k=x^1x^2\cdots x^k$, $1\leqslant k\leqslant L$, then

$$N(T^1,q) = N(x^1,q)$$

and

$$N(T^k,q) = N[x^k,N(T^{k-1},q)] \qquad \text{for } 1<k\leqslant L$$

Similarly, the definition of Z is extended to denote the output sequence produced when T is applied and the starting state is q:

$$Z(T^1,q) = Z(x^1,q)$$

and

$$Z(T^k,q) = Z(T^{k-1},q)Z[x^k,N(T^{k-1},q)] \text{ for } 1<k\leqslant L$$

The problem of finding a test sequence for a fault is precisely that of determining an input sequence T that distinguishes a pair of finite-state machines. That is, given a pair of finite-state machines M_G and M_F having a common input-symbol set X and with state sets G and F, respectively, an input sequence T is a test sequence to distinguish M_F from M_G if $Z_F(T,f) \neq Z_G(T,g)$ for every state-pair (g,f) contained in $G \otimes F$.

This problem can be restated in a simpler form in terms of a single "Cartesian product machine" $M_C = M_G \otimes M_F$, defined as follows: The state set C of M_C consists of all the state-pairs (g,f) defined by $G \otimes F$. The input-symbol set is X, and the output-symbol set is {0,1}. The next-state function is

$$N_C(x,c) = [N_G(x,g), N_F(x,f)]$$

and the next-output function is

$$Z_C(x,c)=1 \quad \text{if} \quad Z_F(x,f) \neq Z_G(x,g)$$
$$Z_C(x,c)=0 \quad \text{if} \quad Z_F(x,f) = Z_G(x,g)$$

where c=(g,f). Clearly, an input sequence T is a test sequence to distinguish M_F from M_G if and only if, for every state c of M_C, $Z_C(T,c)$ contains at least one $\underline{1}$.

Theorem 6.1

Given a pair of finite-state machines M_G and M_F that can be distinguished, there exists a test sequence of length at most $\frac{p(p+1)}{2}$, where p is the number of states in $M_G \otimes M_F$.

Proof

A state c in $M_C = M_G \circledast M_F$ will be called "detected" by an input sequence T' if $Z_C(T',c)$ contains at least a single $\underline{1}$. We prove by induction on $\underline{k}$ that, when the set of undetected states contains p-k+1 members, there exists an input sequence of length at most k that detects an additional state.

Unless M_G and M_F can both be reduced to combinational circuits, there exists some input symbol x and some pair of states for at least one of the machines, say M_G, such that the next-output function is state-dependent, i.e., $N_G(x,g_i) \neq N_G(x,g_j)$. Then whatever the form of N_F, there exists some state c in M_C such that $N_C(x,c)=1$. Thus the set of undetected states can always be reduced from size p to size p-1 (or smaller) with an input sequence of length one, and the basis of the induction (k=1) is proved.

Now let T' be an input sequence that detects k-1 states, thus leaving p-k+1 states undetected. Let d and u be members of sets D and U of detected and undetected states, respectively. There are three possibilities to consider:

Case 1. There exists an input symbol x such that $Z_C[x,N_C(T',u)]=1$ for some u. In this case the size of u can be reduced by adding x to T'.

Case 2. There exists a state u such that $N_C(T',u)=d$ for some d. By inductive hypothesis there exists an input sequence T'' of length at most k-1 that detects d. In this case the size of U can be reduced by adding T'' to T'.

Case 3. If neither case 1 nor case 2 applies, then there must exist an input symbol x such that $N_C[x,N_C(T',u)]=d$ for some d. For otherwise, $N_C[x,N_C(T',u)]$ is in U and $Z_C[x,N_C(T',u)]=0$ for every x and every u, and then the states in U are undetectable by input sequences of any length, which is contrary to assumption. By inductive hypothesis there exists an input sequence T" of length at most k-1 that detects d. In this case the size of U can be reduced by adding xT" to T'.

Thus the set of undetected states can always be reduced from size p-k+1 to size p-k (or smaller) with an input sequence of length k, and the induction is proved.

In the worst case the complete test sequence consists of p subsequences of lengths 1,2,$\cdots$,p, and the total length is
$$\sum_{k=1}^{p} k = \frac{p(p+1)}{2} .$$

Note that in the above argument we made no assumptions whatsoever about whether M_G or M_F possesses "equivalent" states or not and whether the machines are "strongly connected" or not. [A pair of states (q_i,q_j) is equivalent if $Z(T,q_i)=Z(T,q_j)$ for every input sequence T. A machine is strongly connected if, for every pair of states (q_i,q_j), there exists an input sequence T such that $N(T,q_i)=q_j$.] We avoided such assumptions because a faulty circuit may very well not be strongly connected, and even a fault-free circuit may have equivalent states.

Using the concept of synchronous-equivalent circuits, we can determine an upper bound on the length of a test sequence to detect a given fault in an LIFMSC.

Theorem 6.2

Let A $[A_F]$ be the set of all stable internal states in a fault-free [faulty] LIFMSC, and let p be the product of the number of states in A and the number of states in A_F. Then either a test sequence of length at most $\frac{p(p+1)}{2}$ exists or the fault is undetectable.

Proof

The theorem follows immediately as a corollary to Theorem 6.1.

This result is not very exciting because, for example, if A and A_F contain about w states each, then the bound is about $w^4/2$. On the other hand, we can now state that our test generation procedure is an algorithm, since we formally know when to terminate an unsuccessful search for a test sequence.

6.4 Identifying Undetectable Faults

The upper bound on the length of a test sequence for a fault -- on the order of $w^4/2$ where w is the number of internal states in the given circuit -- is extremely large. Thus it is impractical to rely on this bound as the criterion for deciding that a fault is undetectable.

An alternative method for recognizing undetectable faults is as follows: If $\tilde{y}^1_{iF} = \tilde{y}^1_i$ and $\bar{\bar{y}}^1_{iF} = \bar{\bar{y}}^1_i$ for every gate and A_F=A and P_F=P, then F is undetectable. Note that we only claim that this is a sufficient condition. (Indeed, intuition leads us to expect that the condition is <u>not</u> necessary.) We do not know of any criterion both necessary and sufficient other than test-length bound.

<u>Example 6.2</u>

For the fault 6b@1 in the edge-triggered type-D flip-flop of Fig. 6.0, $\tilde{y}^{1}_{iF} = \tilde{y}^{1}_{i}$ and $\overset{=}{y}^{1}_{iF} = \overset{=}{y}^{1}_{i}$ for each of the gates. Also $A_F=A$ and $P_F=P$. Thus this fault is undetectable. Incidentally, the unmodified synchronous-equivalent sequence-response expressions for the faulty and fault-free circuits for sequences of length one are <u>not</u> equal for every gate.

6.5 Tests for Complete Sets of Faults

A complete test sequence for a circuit is simply a concatenation, in any order, of the test sequences for each fault. (Recall that sequence-response expressions, hence test sequences for individual faults, are proper for all possible previous total states.) If test-sequence length is important, then judicious assignment of the unspecified inputs may allow one to overlap these subsequences, thus reducing the total length.

Note that the assumption that the state of a circuit is unknown at the beginning of each subsequence is necessary even if the order of concatenation is predetermined, as the following argument shows: Let T be a test sequence for fault F, and let F′ be some other fault. When T is applied and F′ is present, it is possible that the final state is incorrect yet the fault is undetected. Then a test sequence for F′ that assumes a known initial state, namely that of the fault-free circuit at the end of T, may in fact not detect F′.

Although constructing a test sequence for a large set of faults by finding a test for one fault at a time presents no conceptual problem, such an approach is probably not economically feasible. A more practical approach is to use heuristic test generation and simulation until that becomes ineffective and then employ algorithmic algebraic techniques for the remaining undetected faults.

6.6 Summary

The test-generation algorithm presented in this chapter has the following important attributes:

1. The algorithm is applicable to any level-input fundamental-mode circuit and can deal with both asynchronous and synchronous sequential circuits.
2. The tests obtained are independent of initial state and guaranteed not to cause circuit oscillation under the unit-gate-delay timing model. Employing a unit-gate-delay three-valued-logic fault simulator to "verify" the tests is not only unnecessary but can be misleading.
3. The algorithm is not tied to any particular fault model. Short circuits that introduce loops are handled in the same manner as stuck-at faults.
4. The algorithm is based solely on the gate-level circuit description. Flip-flops, feedback loops, state varia-

bles, etc., need not be identified, and no heuristic "loop-cutting" is involved.

5. At least some undetectable faults can be so-recognized.

In addition, if the prescription for finding sequence-response expressions (Procedure 5.3) is in fact complete, as we believe, then the following are also true:

1. If a test sequence exists for a fault, the algorithm will find it.
2. The algorithm determines all test sequences of a given length or less and thus those of minimum length.

REFERENCES

1. W. H. Kautz, "The Necessity of Closed Circuit Loops in Minimal Combinational Circuits," IEEE Trans. Computers, vol. C-19, no. 2, pp. 162-164, Feb. 1970.

2. M. A. Breuer, "Testing for Intermittent Faults in Digital Circuits," IEEE Trans. Computers, vol. C-22, no. 3, pp. 241-246, Mar. 1973.

3. J. P. Hayes, "A NAND Model for Fault Diagnosis in Combinational Logic Networks," IEEE Trans. Computers, vol. C-20, no. 12, pp. 1496-1506, Dec. 1971.

4. D. R. Schertz and G. Metze, "A New Representation for Faults in Combinational Digital Circuits," IEEE Trans. Computers, vol. C-21, no. 8, pp. 858-866, Aug. 1972.

5. F. W. Clegg, "Use of SPOOF's in the Analysis of Faulty Logic Networks," IEEE Trans. Computers, vol. C-22, no. 3, pp. 229-234, Mar. 1973.

6. S. C. Si, Extensions to the E-Algorithm for Fault Detection in Combinational Logic Circuits, Master's Thesis, Electrical Engineering, Lehigh Univ., May 1972.

7. A. R. Klayton and A. K. Susskind, "Multiple-Fault-Detection Tests for Loop-Free Logic Networks," 1971 IEEE Internat. Computer Society Conf., pp. 77-78, Sept. 1971.

8. J. F. Poage, "Derivation of Optimum Tests to Detect Faults in Combinational Circuits," Proc. Symp. Mathematical Theory of Automata, Polytechnic Institute of Brooklyn, pp. 483-528, Apr. 1962.

9. D. B. Armstrong, "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets," IEEE Trans. Electronic Computers, vol. EC-15, no. 1, pp. 66-73, Feb. 1966.

10. F. W. Clegg, Algebraic Properties of Faults in Logic Networks, Ph.D. Dissertation, Electrical Engineering, Stanford Univ., May 1970.

11. F. W. Clegg, The SPOOF: A New Technique for Analyzing the Effects of Faults on Logic Networks, Report SEL-70-073, Stanford Electronics Laboratories, Stanford Univ., Aug. 1970, and Repository Paper R-71-105, IEEE Computer Society, Northridge, Calif.

12. F. F. Sellers, M. Y. Hsiao, and L. W. Bearnson, "Analyzing Errors with the Boolean Difference," IEEE Trans. Computers, vol C-17, no. 7, pp. 676-683, Jul. 1968.

13. S. S. Yau and Y.-S. Tang, "An Efficient Algorithm for Generating Complete Test Sets for Combinational Logic Circuits," IEEE Trans. Computers, vol. C-20, no. 11, pp. 1245-1251, Nov. 1971.

14. A. K. Susskind, "Additional Applications of the Boolean Difference to Fault Detection and Diagnosis," 1972 Internat. Symp. Fault-Tolerant Computing, pp. 58-61, June 1972.

15. S. Even and A. R. Meyer, "Sequential Boolean Equations," IEEE Trans. Computers, vol. C-18, no. 3, pp. 230-240, Mar. 1969.

16. H. Y. Chang, "A Method for Digitally Simulating Shorted Input Diode Failures," BSTJ, vol. 48, pp. 1957-1966, Jul.-Aug. 1969.

17. E. J. McCluskey, Jr., "Transients in Combinational Logic Circuits," in Redundancy Techniques for Computing Systems, ed. R. H. Wilcox and W. C. Mann, pp. 9-46, Washington, D.C.: Spartan Books, 1962.

18. M. A. Breuer, "A Note on Three-Valued Logic Simulation," IEEE Trans. Computers, vol. C-21, no. 4, pp. 399-402, Apr. 1972.

19. G. Frosini and G. B. Gerace, "Pulse Input Asynchronous Sequential Circuits," IEEE Trans. Computers, vol. C-20, no. 4, pp. 436-442, Apr. 1971.

20. M. P. Marcus, Switching Circuits for Engineers, 2nd ed., Chapts. 18 and 19, Englewood Cliffs, N. J.: Prentice-Hall, 1967.

21. S. H. Unger, Asynchronous Sequential Switching Circuits, New York: Wiley-Interscience, 1969.

22. A. Ashkinazy, Fault Detection in Asynchronous Sequential Machines, Ph.D. Dissertation, Electrical Engineering, Columbia Univ., 1970.

23. A. Thayse, "Testing of Asynchronous Sequential Switching Circuits," Philips Res. Repts., vol. 27, no. 1, pp. 99-106, Feb. 1972.

24. F. C. Hennie, "Fault Detecting Experiments for Sequential Circuits," Proc. Fifth Annual Symp. Switching Circuit Theory and Logical Design, pp. 95-110, Princeton Univ., Nov. 1964.

25. E. P. Hsieh, "Checking Experiments for Sequential Machines," IEEE Trans. Computers, vol. C-20, no. 10, pp. 1152-1166, Oct. 1971.

26. D. E. Farmer, "Algorithms for Designing Fault-Detection Experiments for Sequential Machines," IEEE Trans. Computers, vol. C-22, no. 2, pp. 159-167, Feb. 1973.

27. Z. Kohavi and J. Winograd, "Bounds on the Length of Synchronizing Sequences and the Order of Information Losslessness," in International Symposium on the Theory of Machines and Computations, ed. Z. Kohavi and A. Paz, pp. 197-206, New York: Academic Press, 1971.

28. J. F. Poage and E. J. McCluskey, Jr., "Derivation of Optimum Test Sequences for Sequential Machines," Proc. Fifth Annual Symp. Switching Theory and Logical Design, Princeton Univ., pp. 121-132, Nov. 1964.

29. S. Seshu, "On an Improved Diagnosis Program," IEEE Trans. Electronic Computers, vol. EC-14, no. 1, pp. 76-79, Feb. 1965.

30. A. Gill, Introduction to the Theory of Finite-State Machines, New York: McGraw-Hill, 1962.

31. F. C. Hennie, Finite-State Models for Logical Machines, New York: Wiley, 1968.

VITA

Mark Joel Flomenhoft was born to Rosalie and Jay Flomenhoft in Philadelphia, Pennsylvania, on July 19, 1945. He received a B.S.E.E., cum laude, from the University of Pennsylvania in 1967 and an M.S.E. from Princeton University in 1968.

Since 1967 Mr. Flomenhoft has been with Bell Telephone Laboratories. His principal responsibilities have been the development of computer aids for test generation and the circuit design and layout of TTL M.S.I. chips. He has presented papers at the Design Automation Workshop -- "A System of Computer Aids for Designing Logic Circuit Tests" (1970) -- and the International Symposium on Fault-Tolerant Computing -- "Algebraic Techniques for Finding Tests for a Variety of Fault Types" (1973).

Mr. Flomenhoft is a member of Tau Beta Pi and Eta Kappa Nu, and he is an associate member of Sigma Xi.