## General Disclaimer

## One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.

- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.

- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.

- This document is paginated as submitted by the original source.

- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

Produced by the NASA Center for Aerospace Information (CASI)

# A Reproduced Copy

## OF


N 71-16432

FACILITY FORM 602

| (ACCESSION NUMBER) | (THRU) |
| --- | --- |
| 174 (PAGES) | G3 (CODE) |
| CR-116199 (NASA CR OR TMX OR AD NUMBER) | 08 (CATEGORY) |

Reproduced for NASA

*by the*

**NASA Scientific and Technical Information Facility**




FFNo 672 Aug 65

FINAL TECHNICAL REPORT

CONTRACT NAS 12-2201

# "Fault Isolation and Diagnostic Test Techniques for Large Scale Arrays in Sequential Digital Networks"

TELPAR, INCORPORATED
4300 Sigma Road
Dallas, Texas 75240

DEC 1970
RECEIVED
NASA STI FACILITY
INPUT BRANCH

telpar

# TABLE OF CONTENTS


| | Page |
|---|---|
| Introduction | 3 |
| Summary of Effort & Results | 4 |
| Description of Combinational Test Generation Program | 6 |
| Heuristic for Circuits Containing Flip-Flops | 10 |

Appendices:

A Combinational Test Generation Survey

B Test Routines Based on Symbolic Logical Statements by Richard D. Eldred

C Derivation of Optimum Tests to Detect Faults in Combinational Circuits by J. F. Poage

D On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets by D. B. Armstrong

E Diagnosis of Automata Failures: A Calculus and a Method by J. Paul Roth

F Minimization of Fault Detection Tests--An Approximate Method by S. A. Szygenda

G Implementation and Extension of Multi-Dimensional Path Sensitizing in a Simulation and Diagnosis System by S. A. Szygenda and G. C. Goldbogen

H A Software System for Diagnostic Test Generation and Simulation of Large Digital Systems by S. A. Szygenda

I Program Listings

J Solution of Example Circuit


telpar

## INTRODUCTION

This document summarizes the work conducted under subject contract. The results are presented in the form of a description of the Combinational Test Generation Program (CTGP), together with a listing and program deck.

Also included is a detailed analysis of required extensions to the program for handling faults in RS and JK flip-flops.

Appendices present a survey of the principal test generation methods, and copies of the important papers in the field.

telpar

"Fault Isolation and Diagnostic Test Techniques for Large Scale Arrays in Sequential Digital Networks"

## SUMMARY OF EFFORT AND RESULTS

At the outset of the contract, it was agreed by the technical monitor and the contractor that the optimum way to begin development of suitable techniques for diagnosis and isolation of faults in large-scale sequential arrays would be to consider the problem of packaged flip-flops in combinational logic. It was further decided that it would be useful to have a fault simulation program available for use by cognizant government personnel during the interim period prior to delivery of the test generation program.

Accordingly, initial effort was expended in providing a suitable demonstration program. This program was not intended to perform pattern generation, but did provide fault modelling and simulation capability for combinational logic circuits containing flip-flops. The program was made available, and a preliminary copy of the User's Guide was delivered in December 1969. Several government-furnished logic diagrams were processed using a final version, the User's Guide for which was delivered in January 1970.

In the meantime, work was progressing on a test generation strategy. It soon became evident that programming and checkout of the complete flip-flop based approach could not be completed with the time and resources available, and the government then chose to have the combinational portion completed and made

operational, with the remainder of the effort flow-charted.

This was accomplished, and several large example circuits were processed through the generation system. A complete program deck, annotated listings, and example listings are included.

John L. Fike, P.E.
Principal Investigator

telpar

# COMBINATIONAL TEST GENERATION PROGRAM

This paper describes the capabilities and usage of the combination test generation program (CTGP). This information is also included on the program listing.

The program attempts to generate tests for any non-redundant fault in a combinational logic net. If the user does not specify a particular set of faults, the program will derive the collapsed set of all possible S-A-1 and S-A-0 faults. Tests will then be generated for these faults. The gate description of the logic net must be leveled before it is input to the program. This means that the logic must be organized such that no gate can have inputs from a gate of a higher number.

The required inputs to the program are as follows: (all values are right justified)

CARD 1: Col. 1- 80 = Circuit Identification

CARD 2: Col. 1- 4 = Number of Primary Inputs

5- 8 = Total number of gates (primary inputs and outputs are counted as gates).

9-12 = Number of faults in set being input (0 = prog. gen.)

13-16 = Number of tests desired for each fault (-1 = all)

17-20 = Trace of paths and tables. 0=No trace, 1=Complete trace, 2=Trace of paths.

CARD 3: Begins gate descriptions, each gate must be described as:

Col. 1- 4 = Gate number

5- 8 = Gate type (1=AND, 2=OR, 3=NAND, 4=NOR, 5=NOT, 6=INPUT, 7=OUTPUT)

9-44 = Inputs to the gate (in groups of 4)

45-80 = Outputs of the gates (in groups of 4)

(Primary output gates listed in the fan-out must be negative).

The gate numbers must start at 1 and run consecutively to the highest number gate. The primary inputs must all be numbered first.

If all faults are to be considered, the inputs are complete. Otherwise, the desired set of faults must be specified as follows:

CARD N: Col. 1- 4 = Machine (Fault) number (start with 2)

5- 8 = Gate number

9-12 = Signal number (faulty lead).

13-16 = Fault type (1 = S-A-1; 0 = S-A-0)

LAST CARD: Col. 1-4 = If the last net has been considered, enter 999. If additional logic nets are to be considered, enter any other 3 digit number.

The dimension statements are presently set to allow input of 400 gates and 500 faults. These limits can easily be expanded to utilize the capacity of the host machines.

OUTPUT

The desired output is specified by the user. The minimal output for a circuit is the following:

1. The circuit information as provided by the user.
2. The Fault table. This includes all the faults that were generated or the faults that were supplied.
3. For each fault that a test has been generated, the input specifications are given. The input specifications are given in terms 0, 1, or X where X stands for 'don't care.'
4. The output specification is also given for each test that is generated. The output specification will contain at least one D or -D (in the output specification, a "D" implies a value of '1' for the good machine and '0' for the faulty machine. A "-D" implies a value of '0' for the good machine, and '1' for the faulty machine), except in the case of the fault being on a primary output. In the latter case, only the good machine values are given.

In addition, the user can request a trace through the paths sensitized by the program (this includes unsuccessful paths) and the user can also request the status of the buffers. This information may be helpful for isolating circuit difficulties when tests do not exist, i.e., redundancy in the circuit.

Six example circuits have been provided which demonstrate all output configurations. Included in the examples is a 53 gate dual tree decoder. The decoder has five inputs and 32 outputs, therefore, the circuit description consists of 90 gates.

# DIAGNOSTIC TEST GENERATION HEURISTIC FOR COMBINATIONAL CIRCUITS CONTAINING FLIP FLOPS

Telpar, Incorporated
4300 Sigma Road
Dallas, Texas 75240

## INTRODUCTION

The goal of the work reported on in this paper was to develop heuristics capable of generating tests, if possible, for S-A-1 or S-A-0 faults that could occur in combinational logic nets containing flip flops.

The basic philosophy of this effort was to permit utilization, as much as possible, of techniques and programs developed for a multi-dimensional path sensitizing algorithm (CTGP).[1] The feasibility of extending these programs to handle flip flops has been established. While a considerable programming effort would be required, the logical structure of the system would remain the same. It was recognized that the direct utilization of the developed programs would probably not be likely for this present effort. However, the basic techniques and experience gained, in this original effort, could be time saving during the program development stages of the heuristics.

In line with these objectives, the basic philosophy adopted in the following heuristics can be summed up as follows. Utilize the multi-dimensional path sensitizing

techniques to propagate test conditions and to verify the consistency of the test. When flip flops are necessarily encountered, special subroutines would be called to establish homing sequences for the test application. After these conditions have been established, control will be returned to the CTGP. It is this ping-pong scheme that could efficiently utilize the CTGP techniques, and permit easy addition of new flip flop types as subroutines.

It should be pointed out that the basic philosophy will not be propagation of conditions through the flip flops. Instead, an attempt will be made to establish flip flop conditions necessary to detect the fault.

The following assumptions and constraints are used in the heuristics. Some of these constraints will be lifted during the program design phase of the project. Every attempt has been made not to adopt constraints which would complicate the developed procedures. In other words, worst case conditions are initially assumed; any additional information would probably enhance the heuristics potential and speed.

1. Only J-K and R-S flip flops will be considered.
2. Pre-set and pre-clear lines will not be available. (Having these available would enhance the heuristic. However, their availability is not always assured.)
3. A special heuristic for faults on clock lines will be developed. Hence, they are considered

here only to the extent that they might affect the consistency operation.

4. The present status of the flip flop may or may not be known. This will necessitate special handling in the flip flop subroutines.

5. Fault detection is the primary objective. Isolation of faults is not mandatory.

6. Single fault assumption, s-a-1 and s-a-0.

Before considering the heuristic, an explanation of how faults will affect R-S and J-K flip flops will be given. The heuristics are based on these conclusions. This will be followed by descriptions of the heuristics, and, finally, various aspects and examples of the heuristics will be considered.

## I. FLIP FLOP ANALYSIS

It can be shown that fault conditions can be propagated through combinational logic to the input leads to flip flops. If the fault would affect the flip flops, it would cause one of the flip flop inputs to carry a constant "1" or "0" value. This would be the same condition as if one of these inputs assumed a s-a-0 or s-a-1 value. Hence, one need only consider faults that could occur on the flip flop inputs or outputs. All possible faults on paths feeding these leads would automatically be detected.

## A. J-K Flip Flop Analysis

Consider a J-K flip flop with the following truth table.

| J | K | $Q_{t-1}$ | Q |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | d | 0 |
| 1 | 0 | d | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

d = don't-care




It is necessary to determine how faults on the flip flop can be detected. All possible s-a-1 and s-a-0 conditions will be considered. Furthermore, no knowledge of the flip flop's present state will be assumed.

1) s-a-0 on J

It will be necessary to exercise JKFF in such a manner as to show that the fault exists. We cannot attempt to set the flip flops (with the fault present, this may not be possible). Hence, the approach is to try exercising of the flip flops in a manner which

would be possible in the fault free case, and would be inhibited in the faulty case. This may require a sequence of tests (homing sequence) if the state of the flip flop is not known.

Assume input tests which take the form $t_i$ = (value of J, value of K). Consider the application of the two tests $t_1 = (1,1)$, $t_2 = (1,1)$. For the fault free case, this would produce an output change (from (0,1) to (1,0) or from (1,0) to (0,1), depending on the state of the flip flop). For the faulty case, J,K would have the values (0,1), (0,1) when tests $t_1$, $t_2$ are applied. Hence, the output would be (0,1), (0,1). Note that no output change has occurred. Therefore, if the test sequence $t_1 = (1,1)$, $t_2 = (1,1)$ could be produced, at JK, for the fault free case, it would be inhibited by any fault that could propagate to J as a s-a-0, and the fault would be detected by noting an absence of change in Q. For imbedded flip flops, it will be necessary to propagate these output conditions to primary outputs. This is considered in the heuristic description.

2) s-a-0 on K

For this condition, the analogous procedure to that used for a s-a-0 on J will be necessary. In fact, the same test sequence will detect the fault condition. Hence, we can conclude that the test sequence $t_1 = (1,1)$, $t_2 = (1,1)$ would be sufficient to detect any fault that

would propagate to the inputs of a J.K. flip flop in the form of a s-a-0 condition. (This may be either a s-a-0 or s-a-1 fault which is propagated.) This fault would be detected by observing no output change for the faulty case, as opposed to an output change for the fault free case.

3) s-a-1 on J

With J having the appearance of a s-a-1, we would attempt applying a 0 at J. There would be an added advantage if the K input would be specified as "1". (Then the same procedure can be used as in 1) and 2).) Hence, if we apply a test sequence $t_1$ = (0.1) and $t_2$ = (0,1), the fault free case would produce the output Q = (0,0) for the two tests. For the faulty case, the output sequence would be (0,1) or (1,0) since J would be s-a-1. Therefore, an output change would indicate a fault and the absence of a change would signal no s-a-1 fault of this type.

4) s-a-1 on K

This would be handled precisely as 3), except the test sequence would be $t_1$ = (1,0) and $t_2$ = (1,0). Therefore, we can conclude that the test sequences $t_1$ = (0,1), $t_2$ = (0,1), and $t_1'$ = (1,0), $t_2'$ = (1,0) would detect any fault that would manifest itself as a s-a-1 on either J or K. The faulty case would be detected by a change

on the output Q. No change at Q would indicate the absence of this type of fault.

5) s-a-0 on Q

This case could simply be detected by the application of a single test t = (1,0) which should force a change at Q. For the fault case it would not.

6) s-a-0 on $\bar{Q}$

The test, t = (0,1) would detect this fault, as above.

7) s-a-1 on Q

The application of the same test sequence as used for a s-a-0 on J or K ($t_1$ = (1,1), $t_2$ = (1,1)) would detect this condition by noting no change in output Q for the faulty case.

8) s-a-1 on $\bar{Q}$

This fault would be detected by the same test sequence as in 7), with no change occurring at $\bar{Q}$.

For the JK flip flop analysis to be complete, two additional points should be considered. First, what would happen if we always knew the present state of the flip flop, and secondly, what if we could not force the appropriate failure free flip flop input to the "1" state.

We can dispense with the first consideration by realizing that knowledge of the flip flop state may shorten the necessary test sequence. For example, if we know that $Q = 0$, it would be necessary to apply only one test, $t = (1,1)$, to detect the s-a-0 condition on J. Similar argruments can be given for the other conditions. However, it should be noted that this knowledge will not always shorten the test sequence. For example, if $Q = 1$, the application of the test $t = (1,1)$, with J s-a-0, would result in a change in Q. If we want to keep the same procedure as was used when we had no knowledge of the reset condition, it would not indicate a fault. The application of the second test $t_2 = (1,1)$ would, however, indicate the fault. The possible knowledge of the flip flop present state should be considered in the flip flop subroutine to economize test generation time and the length of test sequences.

For the second point, it would be necessary to generate an alternate test sequence, if the first one could not be achieved. For the s-a-0 on an input case, we could assume that the non-faulty input could be forced to "1" under the appropriate conditions. (If K or J could never be forced to "1", the $\bar{Q}$ or Q output would always be active, dependent on the clock, once it was first activated).

The appropriate conditions may be a function of time, requiring a longer test sequence to detect the fault (perhaps

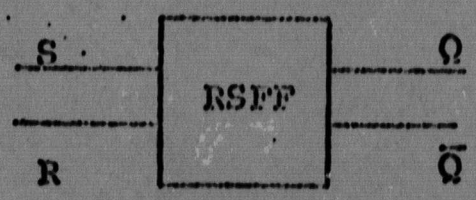
$t_1 = (1,1)$, $t_2 = (1,0)$, $t_3 = (1,1)$), or it may be a function of the other input. For example, if J = 1, then K = 0. If this is the case, an alternate test sequence, $t_1 = (0,1)$, $t_2 = (1,0)$ would produce a change for the fault free case and no change for the faulty case. This alternative should be provided in the flip flip sub-routine. Similar analysis has been performed for the other possible fault conditions. The analysis of the J-K flip flop is summarized in Table I.

| Fault Manifestation | Test Sequence | Test Conditions | |
|---|---|---|---|
| a) s-a-0 on J or K | $t_1 = (1,1)$, $t_2 = (1,1)$ | No Change on Q or $\bar{Q}$ | ⇒ Fault |
| b) s-a-1 on J | $t_1 = (0,1)$, $t_2 = (0,1)$ | Change on Q or $\bar{Q}$ | ⇒ Fault |
| c) s-a-1 on K | $t_1 = (1,0)$, $t_2 = (1,0)$ | Change on Q or $\bar{Q}$ | ⇒ Fault |
| d) s-a-0 on Q | $t_1 = (1,0)$ | No Change on Q | ⇒ Fault |
| e) s-a-0 on $\bar{Q}$ | $t_1 = (0,1)$ | No Change on $\bar{Q}$ | ⇒ Fault |
| f) s-a-1 on Q or $\bar{Q}$ | $t_1 = (1,1)$ | No Change on Q | ⇒ Fault on Q |
| | $t_2 = (1,1)$ | No Change on $\bar{Q}$ | ⇒ Fault on $\bar{Q}$ |

As can be seen, a maximum of six tests would be required to detect all faults which could manifest themselves as above. The ultimate utilization of this analysis will be described in the heuristic.

B. R-S Flip Flop Analysis

A similar analysis was performed for the R-S flip flop. The results of this analysis are provided in Table II for




| Fault Manifestation | Test Sequence (R S) | Test Conditions |
|---|---|---|
| a) s-a-1 on R | $t_1 = (0,1)$ | $\bar{Q} = (0,0) \Rightarrow$ Fault Free |
| | $t_2 = (0,0)$ | $\bar{Q} = (d,1) \Rightarrow$ Faulty |
| b) s-a-1 on S | $t_1 = (1,0)$ | $Q = (0,0) \Rightarrow$ Fault Free |
| | $t_2 = (0,0)$ | $Q = (d,1) \Rightarrow$ Faulty |
| c) s-a-0 on R | $t_1 = (0,1)$ | $\bar{Q} = (0,1) \Rightarrow$ Fault Free |
| | $t_2 = (1,0)$ | $\bar{Q} = (0,0) \Rightarrow$ Faulty |
| | | or Change on $Q \Rightarrow$ Fault Free |
| d) s-a-0 on S | $t_1 = (1,0)$ | $Q = (0,1) \Rightarrow$ Fault Free |
| | $t_2 = (0,1)$ | $Q = (0,0) \Rightarrow$ Faulty |
| | | or Change on $\bar{Q} \Rightarrow$ Fault Free |
| e) s-a-1 on Q | $t_1 = (0,1)$ | No Change on $Q \Rightarrow$ Fault |
| | $t_2 = (1,0)$ | |

| | | | |
|---|---|---|---|
| f) | s-a-1 on $\bar{Q}$ | $t_1 = (1,0)$<br>$t_2 = (0,1)$ | No Change on $\bar{Q}$ ⇒ Fault |
| g) | s-a-0 on $Q$ | $t_1 = (1,0)$<br>$t_2 = (0,1)$ | No Change on $\bar{Q}$ ⇒ Fault |
| h) | s-a-0 on $\bar{Q}$ | $t_1 = (0,1)$<br>$t_2 = (1,0)$ | No Change on $\bar{Q}$ ⇒ Fault |

If a similar analysis could be performed for other types of flip flops, they could be added to the heuristics as subroutines and then utilized in the heuristics.

It should be noted that the addition of complete test sequences are not necessary each time a flip flop is encountered. This will be demonstrated in the examples.

## II. TEST GENERATION HEURISTIC

We are now in the position to consider the overall heuristics. At least two approaches can be taken to diagnostic test generation.

1. Assuming a given fault some place in the circuit, generate a test sequence.
2. Generate a diagnostic test set, if possible, for all possible s-a-1, or s-a-0, faults in the circuit.

Techniques utilized for either of the above conditions would be identical. The difference would be in

how the techniques are combined to form the total heuristic. The user can be given the option as to specifying his basic objective, as indicated in the heuristic.

Since it will be necessary to backtrace through flip flops, we will consider that problem before providing the heuristic. Actually, this would not create any new problems. When a flip flop is encountered in the backtrace, a flip flop subroutine will be entered to determine the flip flop inputs necessary to produce the desired outputs. It is possible that a particular output is desired but cannot be achieved. For example, $Q,\bar{Q}$ = 1,1. If this occurs, that particular path will be abandoned. If all possible paths for a particular fault must be abandoned, no test exists, using this heuristic, for that fault. This condition, as well as the possibility, if it exists, that a particular fault condition cannot be propagated to a primary output, is what makes this procedure a heuristic and not an algorithm.

We are proceeding on the basic premise that these two conditions will rarely occur and need not cause undue consideration. The circuit can be organized in N levels (1, 2, 3, ···, N); in fact, if the CTG program is used, this will be necessary. Then, the flip flops will be located in various levels of the circuit.

The heuristic will now be given:

Test Generation Heuristic

1) Is diagnostic test generation required for all possible faults in the circuit?

↓ No (a particular fault set)

↓ Yes → (41)

(27)→ 2) Determine the location and type of fault to be detected. Assume the fault is in level i and consider all possible paths that could be propagated from the fault.

↓

(4)→ 3) Propagate all paths remaining under consideration
(6)→ to the next highest level.

↓

4) Are there any flip flops, encountered by the propagated paths, in this level?

↓ Yes

↓ No → 6

5) Are there any paths which do not encounter a flip flop?

↓ Yes

↓ No → 8

6) Was this the last level (N)?

↓ Yes → (7)

↓ No → (3)

(6) ↓

7) It has been possible to propagate the fault to primary outputs, without encountering any flip flops. Hence, the combinational test generation techniques were the only ones used.

↓ (17)

(5)→ 8) Go to the appropriate flip flop routine to establish propagation conditions.

↓ JK ↓ RS (36) ↓ Other (38)

9) Determine the type of fault manifestation at the flip flop inputs.

↓

10) Establish the necessary value, for the fault free lead, to propagate the fault condition for the primary or alternative case, respectively.

↓

(12)→11) Backtrace this value or values to the next lowest
(35)→ level. Are there any flip flops in this level?
(37)→ (If we propagate to primary outputs before this backtrace, we may be carrying paths which could not be consistent. Also, the backtrace would have to make special provisions for backtracing clock lines.)

↓ No (12) ↓ Yes (28)

(11)

(28)→12) Is this level 1?

(30)→

↓ Yes

↓ No → (11)

(15)→13) It has been possible to establish consistency for this condition. Propagate the fault conditions established at the outputs of the flip flops encountered to the next higher level.

↓

14) Are any flip flops encountered in this level?

↓ No

↓ Yes → (39)

15) Is this the last level (N)?

↓ Yes

↓ No → (13)

16) It has been possible to propagate the fault test conditions to the primary outputs.

↓

(7)→17) It will be necessary to backtrace and establish

(23)→ consistency for this test sequence.

(40)→

↓

(20)→18) Backtrace to the next lowest level.

↓

19) Are any flip flops encountered which must be considered at this level?

↓ No → (20)

↓ Yes (JK) → (28)

↓ Yes (RS) → (28)

↓ No (Other) → (28)

(19)

20) Is this the first level?

Yes ↓ / No → (18)

21) Consistency has been established for this test.

22) Does the detection of this fault require a test sequence greater than 1?

Yes ↓ / No → (24)

23) Has consistency been established for the entire sequence?

Yes ↓ / No (consider the next test in the seguence) → (17)

(22)→ 24) Simulate the generated test sequence with all faults being considered, inserted.

25) Eliminate the faults contained in the fault set being considered which are detected by this test sequence.

26) Is the fault set empty?

No → (27) / Yes → Test generation is complete

(26)

↓

27) Consider another fault contained in the fault set.

↓

(2)

(11)→ 28) Does any flip flop have to be considered? (Could

(19)→ we make arbitrary decisions to avoid flip flops.)

↓ Yes

↓ No

(12)

29) Attempt to determine conditions necessary to back-trace through the flip flops for the entire test sequence.

↓

30) Is it possible to backtrace through this level?

↓ No

↓ Yes

(12)

31) Have any possible alternate paths been eliminated previously in the procedure?

↓ No

↓ Yes

(40)

32) Have the alternate flip flop conditions been considered?

↓ No

↓ Yes

The heuristic cannot generate a test for this fault

33) Consider the leads that could not be backtraced through the flip flop.

(34)

(33)

↓

34) Determine which flip flop inputs have specified the inconsistent condition.

↓

35) Go to the appropriate flip flop routines and generate the alternate test sequences values.

↓

(11)

R-S Flip Flop

(8)→ 36) Determine the type of fault manifestation at the flip flop inputs.

↓

(38)→37) Establish the necessary value for the fault free lead to propagate the fault condition for the primary or alternative case, respectively.

↓

(11)

(8)→ 38) Take appropriate action for other types of flip flops which may be under consideration.

↓

(37)

JK, R-S or Other Flip Flops

(14)→39) If it is possible to use paths which do not encounter flip flops, go to (5). Determine the

conditions necessary, at the respective flip flop inputs, to propagate the entire test sequence through this level of flip flops.

↓

(11)

(31)→40) Alternative paths or conditions were eliminated by arbitrary decisions. It will be necessary to consider these alternate conditions and attempt the backtrace again.

↓

(17)

(1)→ 41) Determine the highest level of the circuit which contains flip flops.

↓

(45)→42) Arbitrarily choose one of these flip flops and

(48)→ attempt to generate a test sequence for each possible fault which could occur at the inputs.

↓

43) Simulate the generated test sequences with all faults inserted. Eliminate the detected faults from the initial fault list.

↓

44) Eliminate the considered flip flop from further consideration.

↓

(45)

(44)

45) Have all the flip flops in this level been considered?

Yes ↓

No → (42)

46) Consider the next lowest level containing flip flops.

↓

47) Are we at the primary inputs?

No ↓

Yes → (49)

48) Repeat the above procedure considering only those flip flop input faults which have not been previously detected.

↓

(42)

(47)→49) All faults that propagate through flip flops, and are detectable, should have been eliminated from the fault list. The remaining faults are either not detectable or indicate only combinational logic.

↓

50) Use the combinational logic test generation programs to generate tests for the remaining faults.

## III. DISCUSSION AND EXAMPLES

A number of points listed in the heuristic may need some clarification, which will be provided in this section.

In step (1), the user makes the basic decision as to whether he wants to have the program generate tests for all faults or just generate tests for a specific subset of faults. In the former case, the test generation philosophy would be somewhat different, since faults along a path to flip flop inputs would manifest themselves as faults on the inputs. Hence, if these faults are considered, the faults along the entire path would be detectable. Obviously, the longest paths will be those from the flip flops in the highest level; therefore, these are treated first, etc.

In step (3), all paths will be initially considered. As flip flops are encountered, that path is eliminated. If no path remains, at some point in the forward propagation, this indicates that a path with flip flops must be considered. By saving the paths which have been eliminated, it will be possible to choose the path containing the fewest flip flops.

The backtrace described in step (11) is performed each time the conditions have been propagated forward through one level of flip flops. In this manner, possible inconsistencies are detected with minimum effort. These

paths would then be terminated. Furthermore, necessary values will be established as needed, shortening the overall backtrace procedure.

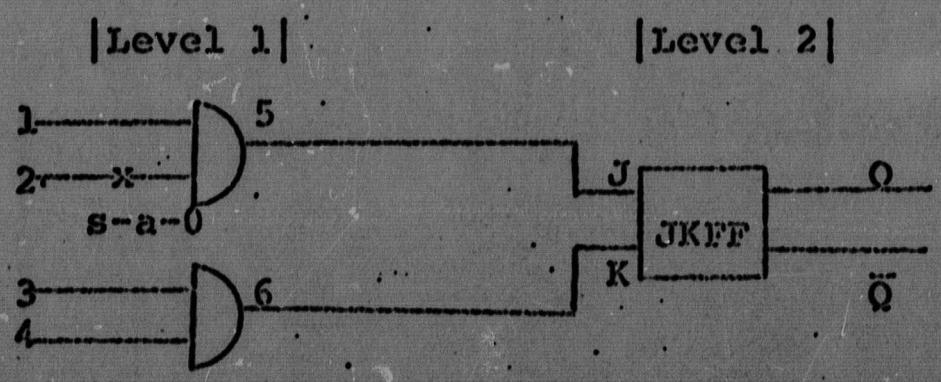
In this backtrace procedure, special consideration will have to be devoted to lines that originated as clocks. These may necessitate the carrying of multiple tests differing only in the values of the clock lines.

Step (22) considers the case where flip flops have been encountered and test sequences are necessary. For this condition each test will have to be propagated, and possibly increased if other flip flops are encountered, and each test must then be backtraced. This procedure is going to require extensive bookkeeping and data manipulation.

Step (26) represents the successful completion of the test generation and simulation procedure. It represents the desired ending point for the heuristic.

A number of examples will be given to demonstrate the heuristic.

Example 1.



Heuristic Steps

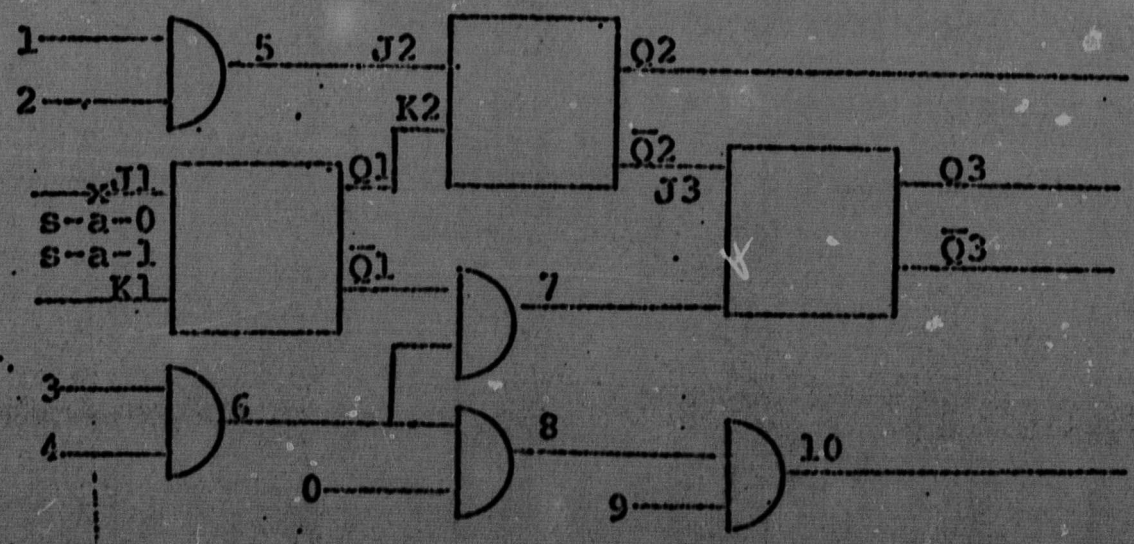
1. Generate a test to detect lead 2 s-a-0.
2. The only possible path is 2-5-J-$Q\bar{Q}$.
3. To propagate through level 1 we can require (1,2) = (1,1).
4. No flip flops in this level.

3 & 4. In level 2 we encounter a flip flop.

5. There are no other possible paths.
8. Go to the JKFF routine.
9. The fault manifestation is J, s-a-0
10. Set K = 1,1
11. Backtrace establishing (1,2,3,4) = (1,1,1,1); (1,1,1,1)
12. The backtrace terminates at level 1.
13. Propagate to next level.
14. No other flip flops.
15. Last level.
17. Backtrace is complete and test sequence is established.

test

Example 2.



Consider a s-a-0 on J1

Listing all steps of the heuristic would be very lengthy. Therefore, a general discussion of the procedure will be given.

Test J1, K1 = (1,1); (1,1) would propagate the fault to the output of the flip flop. These conditions must now be propagated to the primary outputs. Q1 would normally change, with the fault present it will hold constant, and if we can force J2 to a "1" value, we can propagate to Q2. This would give the following test sequence.

1,2,J1,K1 = (1,1,1,1); (1,1,1,1)

Fault Free Outputs Q2,Q3,$\bar{Q}$3,10 = Change in Q2

Faulty Case Q2,Q3,$\bar{Q}$3,10 = No Change in Q2

Consider a s-a-1 on J1

The situation is now somewhat changed. The test to J1,K1 would be (0,1); (0,1). This should not produce a change at Q1, but under fault conditions it will. If Q1 = 1, before the test is applied, then Q1 will go to "0" and Q2 will equal 1 if J2 = 1 and J1,K1 = 0,1 is applied. When these values are applied for the second time, Q1 will go to 1 and Q2 to 0. Hence, a change would occur at Q2; for the fault free circuit Q2 = 1,1.

Now, let's assume that Q1 = 0 before the first test is applied, then after the first test, Q1 = 1 and if Q2

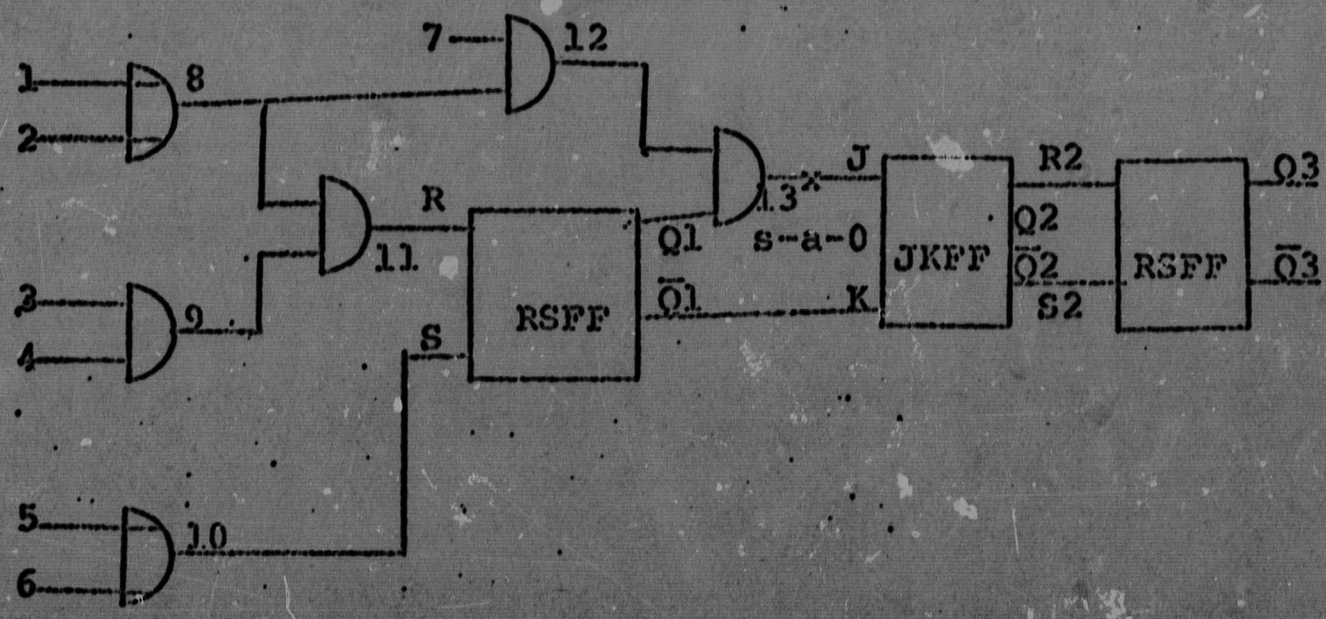
was 0, Q2 will go to 1. For test 2, Q1 will go to 0 and Q2 will go to 1. Hence, no difference would be noted for the fault case. This demonstrates the necessity of sensitizing all paths, or at least being capable of doing this should a situation arise such as this.

Two possibilities exist:

1. We could attempt to propagate the fault to Q3, or,
2. We could use more than two tests, since more than one flip flop is encountered.

First, let's consider (2). If we applied the test three times, it would become independent of any initial state and the fault would be detectable. It appears that, in the worst case, if N flip flops are encountered, a test sequence length N + 1 would be required. Analysis of case (1) showed that this fault would not be detectable using only two tests and this heuristic.

Example 3.



Consider a s-a-0 on J. To propagate through JKFF we would need the condition JK = (1,1); (1,1). We will attempt to backtrace these conditions. Notice that these conditions could not be produced since J,K are being fed by the outputs of an RS flip flop. Hence, we must try the alternate test conditions. The alternate test sequence is JK = (0,1); (1,0). Now we will attempt the backtrace of this sequence. These conditions can be produced for RS = (0,1); (1,0) and 12 = 1. These conditions can be generated at the primary inputs by the following sequence.

$T_1$ = 1,2,3,4,5,6,7 = d,d,d,0,1,d,1

= d,1,1,1,0,0,1

Hence, these conditions have been backtraced to primary inputs. It remains to propagate the conditions to the primary outputs.

Let's consider the outputs Q2, $\bar{Q}$2. For the faulty case the outputs would be Q2, $\bar{Q}$2 = (0,1); (0,1). For the fault free case, the outputs would be Q2, $\bar{Q}$2 = (0,1); (1,0). Therefore, the fault free case could produce primary outputs Q3, $\bar{Q}$3 = (0,1); (1,0) and the faulty case would produce Q3, $\bar{Q}$3 = (0,1); (0,1). Hence, the fault is detected by the test sequence above.

REFERENCE

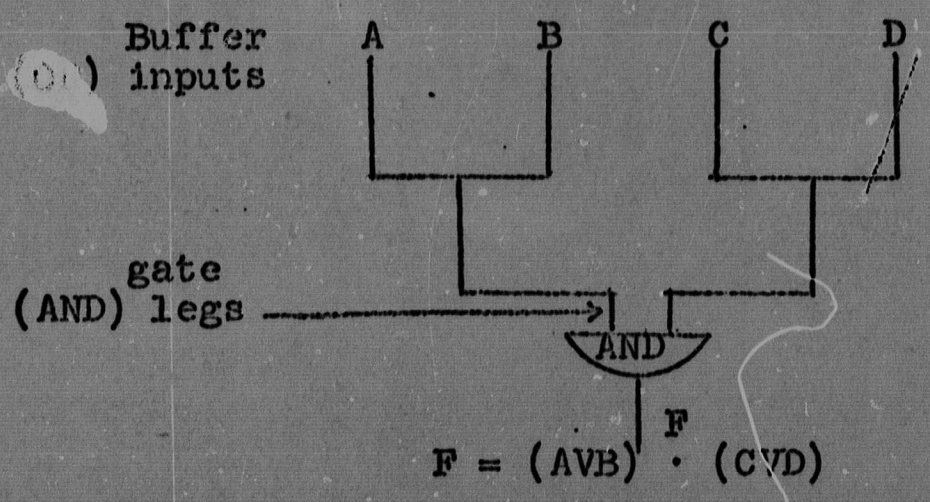
(1) S. A. Szygenda and G. Goldbogen, "Implementation and Extensions of Multi-Dimensional Path Sensitizing in a Simulation and Diagnosis System", Proceedings of the 7th Annual Allerton Conference on Circuit and System Theory, October 1969.

# COMBINATIONAL TEST GENERATION SURVEY

This report will attempt to present combinational test generation algorithms in a readable form. The primary approach to be considered is that of path-sensitizing. While multi-dimensional-path-sensitizing is of the most interest, we will consider one-dimensional-path-sensitizing and some earlier techniques. This approach, and some of the examples which have been taken from Chang, et al, is taken for its historical merit, as well as in an attempt to present a unified coherent discussion.

## I. Eldred

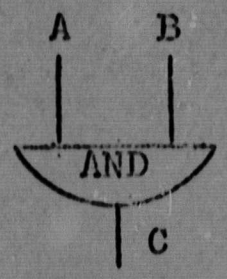
R. D. Eldred[1] wrote one of the first papers on combinational test generation in 1959. He was responsible for diagnostic program development for the Datamatic 1000. The logical function for this system is in the form of a buffer-gate-buffer type represented below.




Assume that the registers have been tested by cycling between its SET and RESET states. The problem then reduces to finding input vectors to a two level OR-AND circuit which allows detection of its faults by obser-

vation of its output.

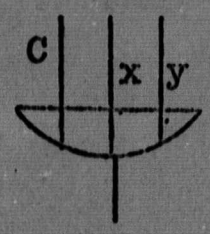
Lets consider a two input AND gate




The input vectors (0,1), (1,0) will detect an input line s-a-1. These tests will also detect the output s-a-1 fault. The test (1,1) would have to be added to detect the output s-a-0.

This approach can be easily generalized to deal with any logic element and any number of inputs. It should be noted that the test vectors for these logic blocks resemble the primitive d-cubes of a failure which we will discuss later.

The problem remaining is to be able to observe the outputs at some level beyond C. We want the AND to propagate its results to observable outputs. (In Eldred's case one additional level.) Lets consider the output C propagating through an OR gate.



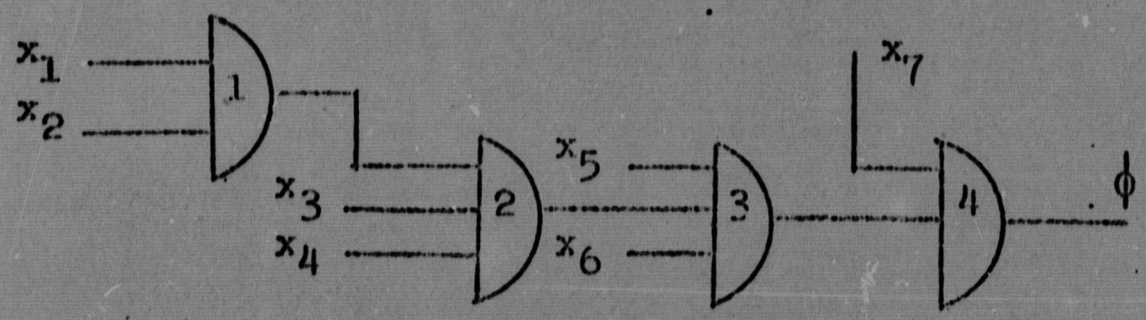

If we set x,y = 0, then C controls the OR's output. It should be recognized that this approach is similar to what we refer to as path-sensitizing and also resembles the primitive d-cube of a logical block, to be discussed later.

We can also give Eldred credit for recognizing that test input vectors often contain <u>don't cares</u> and these can be used for test minimization.

## II. One-Dimensional Path Sensitizing

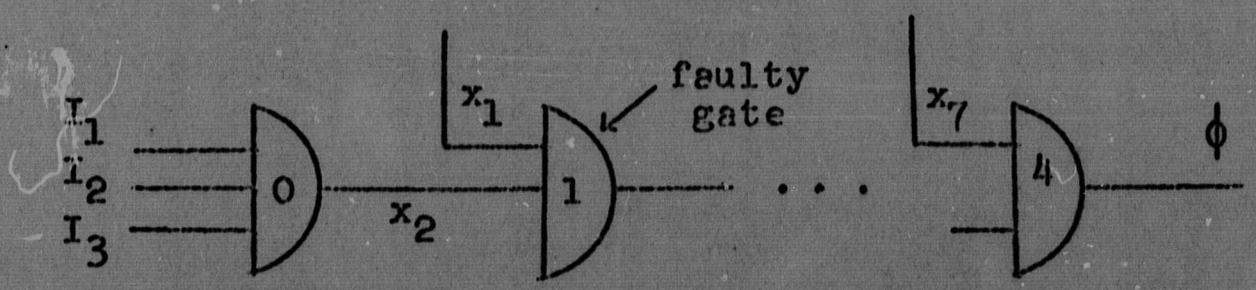
While Eldred's technique was applied to one or two level combinational circuits, one-dimensional-path-sensitizing extends this approach to any number of levels.

The basic approach is quite simple. From the failure, we choose a path to the observable outputs, such that the tests derived for the faulty gate dictate the output values. Consider the example below.



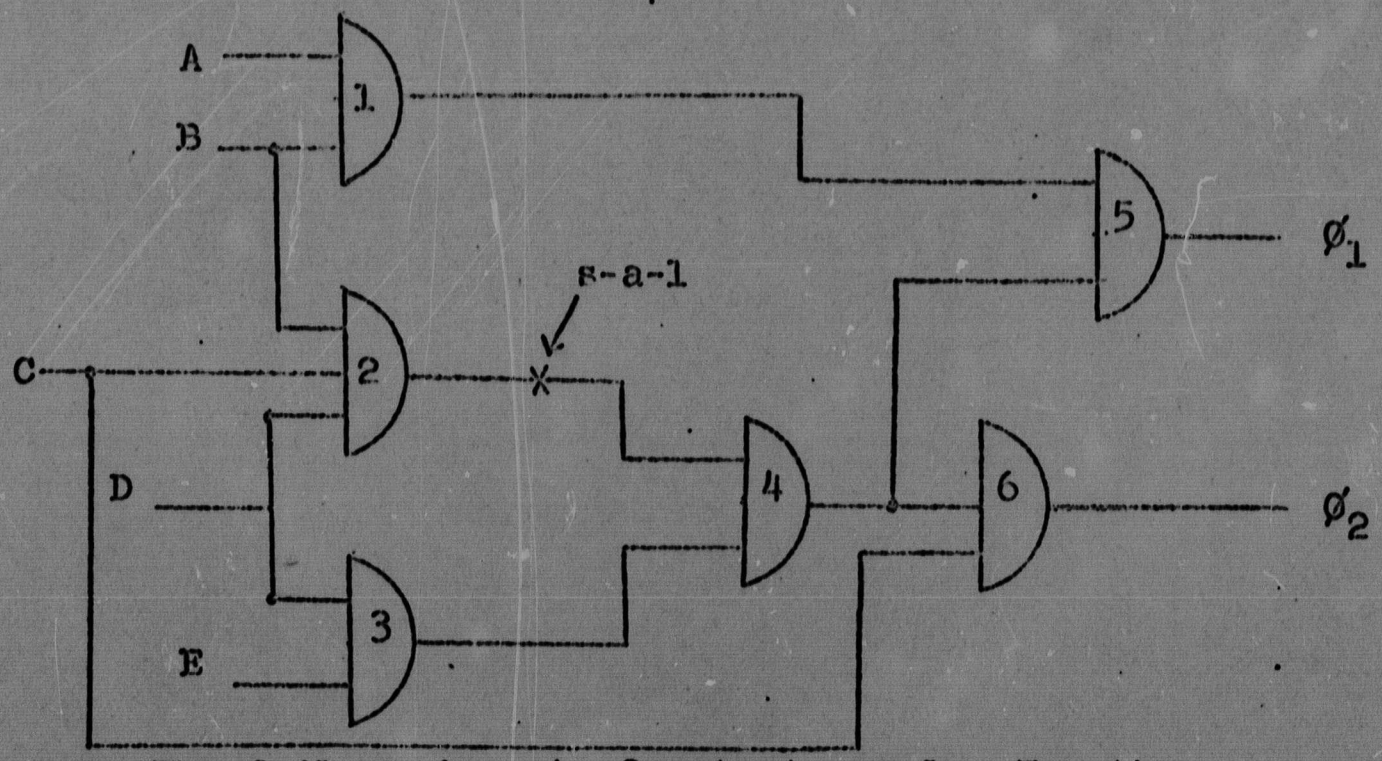

Say that a failure occurs at gate 1, then the inputs to gates 2, 3, and 4 are selected so that the output of gate 2 is determined by the input from gate 1, the output of gate 3 is determined by the input from gate 2, etc. D. B. Armstrong[2] labeled this path as a "sensitized path". The process discribed above is called the forward trace phase of the method.

Consider the case below.



We can see that it is not sufficient to know the inputs to the faulty gate 1. It is necessary to determine a primary input vector ($I_1$, $I_2$, $I_3$) which will realize all the necessary input values. This task is accomplished by tracing backwards from the inputs of gates 1, 2, 3, and 4 to primary inputs. This is referred to as the backward-trace phase of the method.

We are now in a position to work through an example using the techniques described. Consider the following circuit.




The failure is gate 2 output s-a-1. For the forward trace we must select a test which demonstrates the failure in terms of the inputs and outputs of gate 2.

A test could be (B, C, D) = (1, 0, 1). Then the output of gate 2 (g2) will be 0 if the fault is absent, and 1 if it is present.

Next we can arbitrarily choose the path g2, g4, g5 to the output $O_1$. We must now determine the necessary conditions to sensitize this path.

These are:

g3 = 1 to sensitize the path through g4
g1 = 1 to sensitize the path through g5

This is all that is needed for the forward trace of this example. It now becomes necessary to find the primary input vector needed to satisfy these conditions (the backward trace).

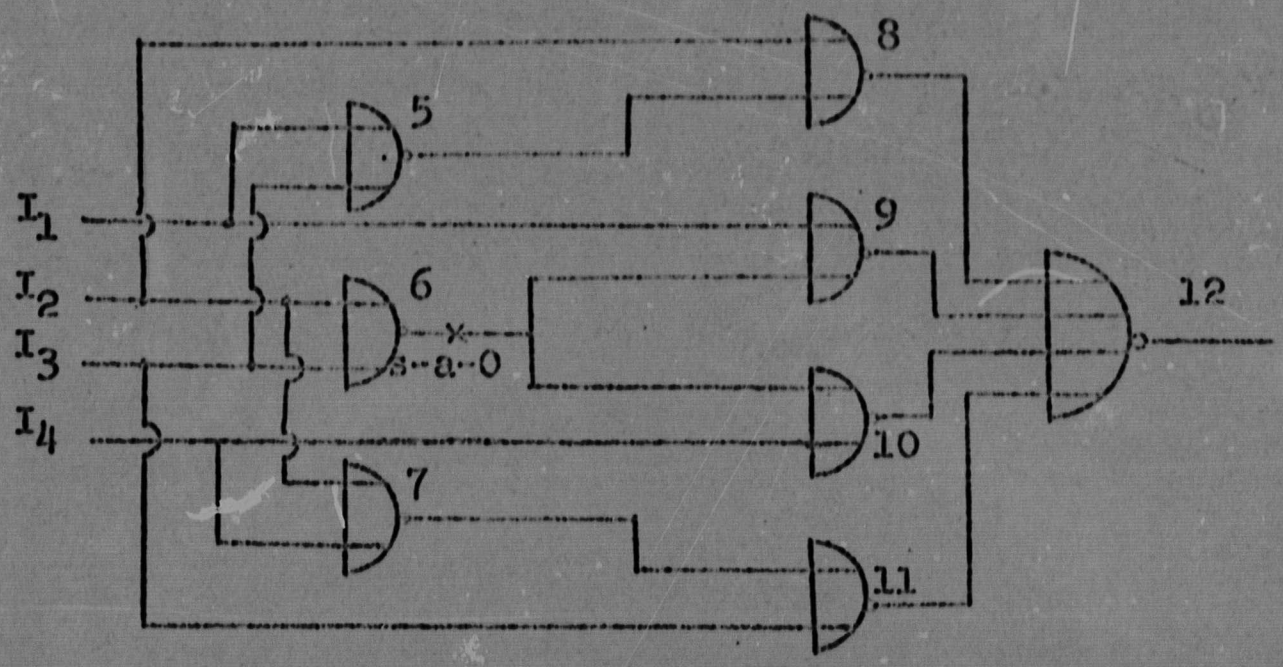
Since D = 1 we must set E = 1 for g3 to be equal to 1. Similarly for gate 1. Since B = 1 we must set A = 1 to have g1 = 1. This gives us a primary input vector (A, B, C, D, E) = (1, 1, 0, 1, 1) and the backward-trace phase is completed.

If this test is applied to the circuit the output $\phi_1$ is 1 for failure free operation and 0 for the case when the failure is present.

Note that path g2, g4, g6 has been sensitized by the test. However we could not observe the failure at output $\phi_2$. This is due to the fact that $\phi_2$ will be 0 whether or not the failure is present. This peculiarity is a result of the reconvergent fan-out.

In conclusion of this section, a number of points should be considered with respect to one-dimensional-path-sensitizing. First of all, it has been necessary to make a number of arbitrary decisions with respect to the gate input signals for the example considered. Conflicts in assigned signals can result from certain inappropriate initial choices. Under these conditions it would be necessary to back up and reconsider the arbitrary choices until the conflict is resolved. The difficulties with this back-up and search are self-evident.

The second point is that this technique is not an algorithm. That is, counterexamples exist for this method. A well known example of this condition is provided by Schneider[3]. The circuit considered by Schneider is given below.




In this example we have the failure condition g6 output s-a-0. A test cannot be generated, for this fault, using one-dimensional-path sensitizing. Any single path through g9 and g12 forces g8 = 0, g10 = 0 and g11 = 0, implying that g7 = 1 and $I_4$ = 1. However, if $I_4$ = 1, g7 cannot be 1. Hence a contradiction.

Considering the path through g10 and g12, we must have g8 = 0, g9 = 0, g11 = 0, which results in the same difficulty.

Therefore, we are not able to generate a test using one-dimensional-path-sensitizing. However, the test (0, 0, 0, 0) will detect this failure.

## III. Multi-Dimensional Path Sensitizing

It is easy to see, from Schneider's example, that the difficulty with one-dimensional-path-sensitizing is that it only sensitizes one path. The obvious remedy for this situation would be to simultaneously sensitize all possible paths from the location of the failure to the primary outputs of the circuit. One of the major attempts at formalizing this approach is due to J. P. Roth[4] who named it the d-algorithm.

Roth presented this work in a mathematical framework termed the calculus of cubical complexes. This approach, while being mathematically rigorous, seems somewhat vague to most engineers. The basic concepts of the d-algorithm are in some cases extensions of the work described in section I and II. As such, the basic concepts are not much more difficult to grasp once they have been translated to English. This report will attempt this translation.

We will proceed with this translation by first, defining, with examples, the basic concepts of the d-algorithm. Then we will tie the operations together with a somewhat more difficult example.

Definition: The singular cover of a gate is a relationship between the gates inputs and output. The relationship is such that one of the inputs to a gate defines the output of the gate wherever possible. The singular covers for AND and OR gates are given below.



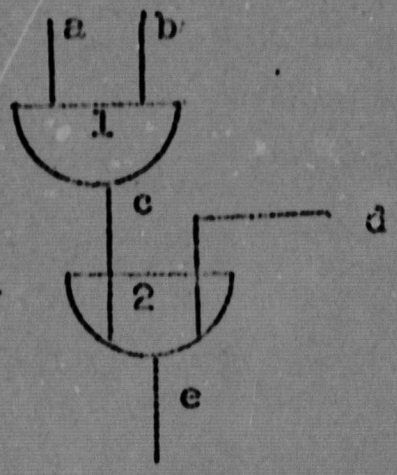

singular covers

| a | b | c |
|---|---|---|
| 1 | 1 | 1 |
| 0 | X | 0 |
| X | 0 | 0 |

| e | f | g |
|---|---|---|
| 0 | 0 | 0 |
| 1 | X | 1 |
| X | 1 | 1 |

The singular cover can be used to describe any gate, and the singular cover for a combinational circuit is built up from the singular covers of the individual

gates. For example:



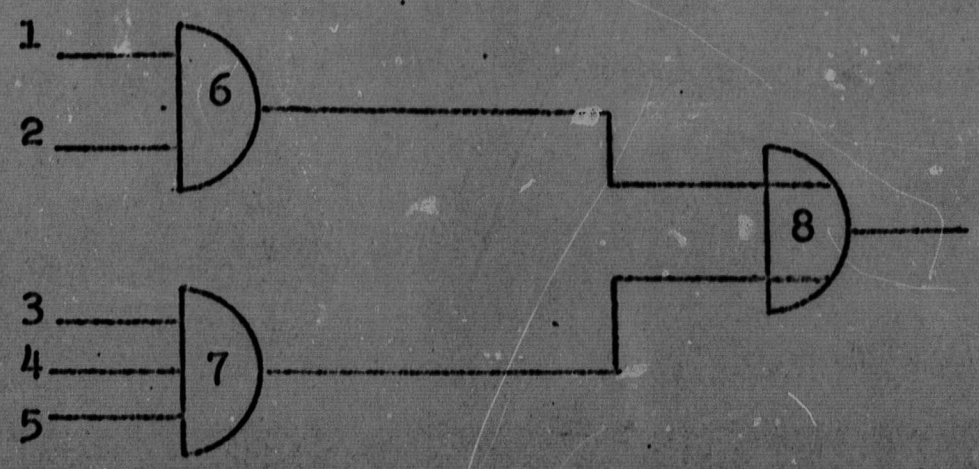

Singular cover of the circuit

| | a | b | c | d | e |
|---|---|---|---|---|---|
| gate 1 | 1 | 1 | 1 | | |
| | 0 | X | 0 | | |
| | X | 0 | 0 | | |
| gate 2 | 0 | 0 | 0 | | |
| | 1 | X | 1 | | |
| | X | 1 | 1 | | |

Definition: Each row of the singular cover is called a cube. For example, the singular cover for an AND gate has three cubes; (1,1,1),. (0,X,0), (X,0,0).

Definition: At this point, we will introduce the circuit labeling con e tion used in the d-algorithm. This scheme will simplify some of the analysis to be considered later.

First, identify the outputs of all gates with integers according to the following rule. No gates output may be an input to a gate with an associated lower value integer. Primary inputs are also labeled in this manner. This procedure can easily be understood with the aid of the following example.



In this example we could not interchange the integers 6 and 8. However, the ordering of the labels internal to a given level is of no particular importance. Also, it is not necessary to assign consecutive numbers as was done in the example.

This leveling rule is a function usually performed by logic organizing or pre-processing subroutines.

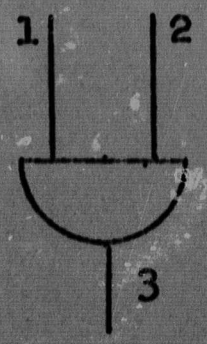
Definition: The primitive d-cube of a logical block is a technique for having one gate input singly determine the gate's output. This is a d-cube description of the circuit with no faults considered. We have previously seen this idea in Eldred's work.

Primitive d-cubes of a logical block can be developed by the following algorithm due to Roth. Intersect the cubes of the gates singular cover using the following rules. The two cubes to be intersected must have different output values. Then, the possible values obtained for the intersection are given in the table below.

| | 0 | 1 | X |
|---|---|---|---|
| 0 | 0 | $\bar{D}$ | 0 |
| 1 | D | 1 | 1 |
| X | 0 | 1 | X |

Table 1.

In this table D may assume the value 0 or 1. However, the value taken in a cube must be the same. If we look at an AND gate,



the singular cover is:

| | 1 | 2 | 3 |
|---|---|---|---|
| $c_1$ = | 1 | 1 | 1 |
| $c_2$ = | 0 | X | 0 |
| $c_3$ = | X | 0 | 0 |

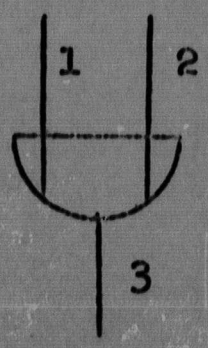
The primitive d-cubes are:

$c_1 \cap c_2$ = D 1 D

$c_1 \cap c_3$ = 1 D D

Note that the ordering can effect the intersections because, $1 \cap 0 = D$ and $0 \cap 1 = \bar{D}$. Hence, one must be consistant in the ordering of intersections.

The primitive d-cube of an OR gate is given below:




singular cover

| | 1 | 2 | 3 |
|---|---|---|---|
| $c_1$ = | 0 | 0 | 0 |
| $c_2$ = | 1 | X | 1 |
| $c_3$ = | X | 1 | 1 |

The primitive d-cubes are:

$c_1 \cap c_2$ = $\bar{D}$ 0 $\bar{D}$

$c_1 \cap c_3$ = 0 $\bar{D}$ D

Definition: The primitive d-cube of a failure (pdcf) is used to express tests for a failure in terms of the inputs and output values of a failed gate. .To construct

the pdef,

1) Develop a new singular cover for the failed gate with the fault effecting the inputs and/or outputs of the gate.

2) Intersect two cubes as specified above. The only difference is that one of the cubes come from the singular cover of the failure free gate, and the other cube is the corresponding cube from the singular cover of the failed gate.

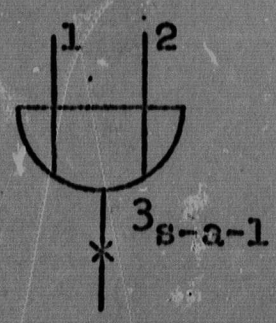
3) An intersection has the value $\bar{D}$ if the value of the output for the failed gate is 1 and the output of the failure free gate is 0. The intersection has a value D if the output of the failed gate is 0, the good gate output being 1.

4) With the exception of the cases given in 3) the rules given in table 1 are followed.

The following examples should clarify this point.

Example a.

First we will consider an OR gate with the output s-a-1.




Remember that the singular cover of the failure free gate was.

| | 1 | 2 | 3 |
|---|---|---|---|
| $c_1$ = | 0 | 0 | 0 |
| $c_2$ = | 1 | X | 1 |
| $c_3$ = | X | 1 | 1 |

The singular cover for the failed gate is,

| | 1 | 2 | 3 |
|---|---|---|---|
| $c_1^! =$ | 0 | 0 | 1 |
| $c_2^! =$ | 1 | X | 1 |
| $c_3^! =$ | X | 1 | 1 |

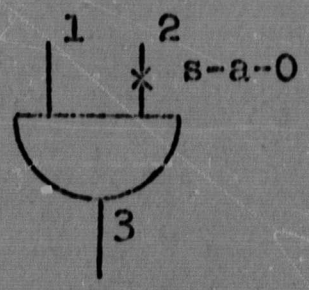
The only possible intersection is

$$c_1 \cap c_1^! = 0 \quad 0 \quad \bar{D}$$

which represents the pdcf for this fault.

Example b.

Now consider an AND gate with an input s-a-0




The singular cover for the failure free gate was,

| | 1 | 2 | 3 |
|---|---|---|---|
| $c_1 =$ | 1 | 1 | 1 |
| $c_2 =$ | 0 | X | 0 |
| $c_3 =$ | X | 0 | 0 |

The singular cover for the failed gate is,

| | 1 | 2 | 3 |
|---|---|---|---|
| $c_1^! =$ | 1 | 0 | 0 |
| $c_2^! =$ | 0 | 0 | 0 |
| $c_3^! =$ | X | 0 | 0 |

The possible intersection is,

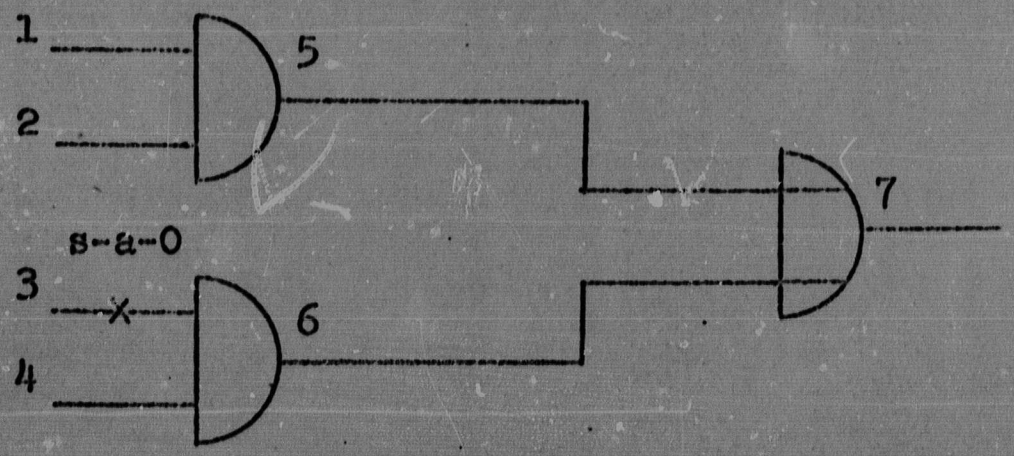
$$c_1 \cap c_1^! = 1 \quad D \quad D$$

which represents the pdcf for this fault. This says that the output would be 1 if the fault does not exist and 0 if it does.

At this point we are in a position to perform the d-drive or forward-trace portion of the algorithm.

Definition: The forward-trace technique is defined in two parts. First, we pick a test for the failed gate (pdcf). Then it is necessary to simultaneously sensitize all possible paths from the failure to all primary outputs. Continual checking is performed for reconvergent fan-out. Cancelled paths due to reconvergent fan-out are abandoned.

The information needed for the first part of the definition has been provided previously. Now it is necessary to develop techniques by which we can propagate the failure through a series of gates, the path sensitizing technique. Consider the following logic net.




Assume that we want to sensitize a path between 3 and 7. For example, let 3 be s-a-0.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| The pdcf is | X | X | D | 1 | X | D | X |

This information is placed in a table as follows:

| | Intersection | | | | | | | Label | Activity | Fan-Out |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | | |
| pdcf | X | X | D | 1 | X | D | X | TC-0 | 6 | 7 |

Definition: The activity specifies the gates that have D's or $\bar{D}$'s as their outputs, and they are a primary output of the subcircuit defined by the coordinates of the test cubes whose values are assigned. That is, the outputs where the fault can be observed.

Definition: The fan-out is a list of those gates that can extend the paths sensitized to this point.

Definition: Now we are in a position to attempt the extension of the path through the specified fan-out. To accomplish this function every test cube (TC) generated must be intersected with every primitive d-cube for the gates specified in the fan-out list. Roth has termed this intersection the D-intersection.

The following table is used for forming the D-intersection of two cubes.

| INTERSECTION | 0 | 1 | X | D | $\bar{D}$ |
|---|---|---|---|---|---|
| 0 | 0 | $\emptyset$ | 0 | $\Psi$ | $\Psi$ |
| 1 | $\emptyset$ | 1 | 1 | $\Psi$ | $\Psi$ |
| X | 0 | 1 | x | D | $\bar{D}$ |
| D | $\Psi$ | $\Psi$ | D | $\mu$ | $\lambda$ |
| $\bar{D}$ | $\Psi$ | $\Psi$ | $\bar{D}$ | $\lambda$ | $\mu$ |

Table 2.

$\emptyset$ means that the intersection is empty.
$\Psi$ means that the intersection is undefined.
If only $\lambda$ (but not $\mu$) occurs in the intersection change all D's to $\bar{D}$'s (and vice-versa) in the second factor, do not do a reintersection.
For $\mu$ the analogous procedure is to be followed.
If both $\lambda$ and $\mu$ occur, the intersection is not defined.

The meaning of $\lambda$ and $\mu$ may be somewhat vague at this point. This difficulty will be cleared up later.

Now we can proceed with the example. We want to compute TCO (d) 7. This is presented in the following table.

| | d-intersection | | | | | | | Label | Activity | Fan-out |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | | |
| pdcf | X | X | D | 1 | X | D | X | TCO | 6 | 7 |
| TCO(7 | X | X | D | 1 | 0 | D | D | TCO-1 | 7 | Ø |

The occurence of Ø in the fan-out list means the final outputs have been reached and the forward-trace phase of the procedure is completed.

One remaining operation exists to complete the algorithm. Roth refers to this as the consistency operation. The consistency operation is identical to the concept of back-tracing encountered in the one-dimensional case. For this simple example we can specify the back-trace values by inspection. Position 5 in the table specifies the output of an AND gate as being 0. This indicates that one of the inputs must be 0. Hence, one of the x's in vertex 1 or 2 must be specified as 0. This provides X 0 D 1 0 D D as a possible test for the specified fault.

Now, let's formalize this back-trace operation. To perform the back-trace we need the rows with fan-out Ø of the d-intersection table (TCO-1) and the singular covers for the circuit,

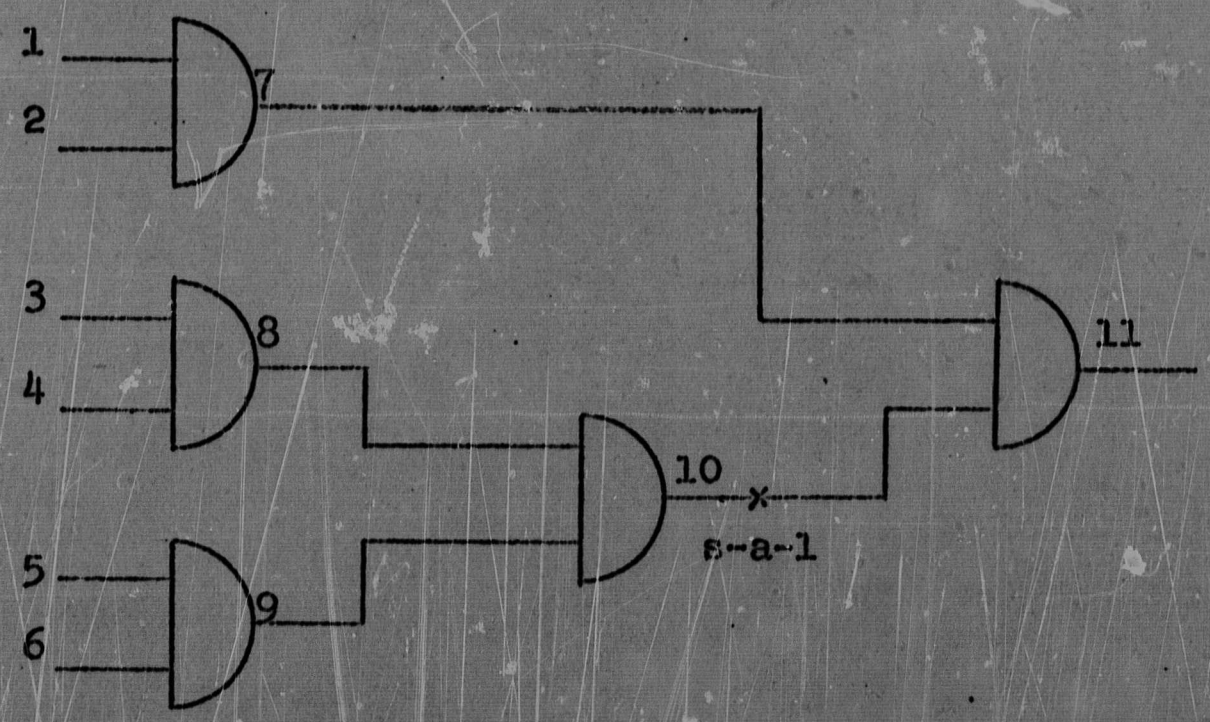
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| gate 5 | 1 | 1 | | | 1 | | |
| | 0 | X | | | 0 | | |
| | X | 0 | | | 0 | | |
| gate 6 | | | 1 | 1 | | 1 | |
| | | | 0 | X | | 0 | |
| | | | X | 0 | | 0 | |
| gate 7 | | | | | 0 | 0 | 0 |
| | | | | | 1 | X | 1 |
| | | | | | X | 1 | 1 |

We begin by examining the vertices, $v_i$, of TCO-1 for the largest i such that $v_i = 0$ or 1. (The circuit labeling procedure is being utilized in this phase of the algorithm.) For our example $v_5 = 0$. Now we look for a singular cover cube which has $v_i$ as an output vertex, with the same value as that specified in TCO-1. Then we take the d-intersection of that singular cover cube with TCO-1. This yields the following table.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| TCO-1 $\textcircled{d}$ $g5_b$ | 0 | X | D | 1 | 0 | D | D |
| TCO-1 $\textcircled{d}$ $g5_c$ | X | 0 | D | 1 | 0 | D | D |

Either of these tests represents a valid test for input 3, s-a-0.

We now have in our command the fundamentals needed to use the d-algorithm. It remains to connect the loose ends in a somewhat more uniform manner. This can best be accomplished by analyzing two somewhat more complicated examples, with the techniques developed.



Since the circuit is described with an acceptable labeling format, we can begin by determining the singular covers for the circuit.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| g7 | 1 | 1 | | | | | 1 | | | | |
| | 0 | X | | | | | 0 | | | | |
| | X | 0 | | | | | 0 | | | | |
| g8 | 1 | 1 | | | | | | 1 | | | |
| | 0 | X | | | | | | 0 | | | |
| | X | 0 | | | | | | 0 | | | |
| g9 | 1 | 1 | | | | | | | 1 | | |
| | 0 | X | | | | | | | 0 | | |
| | X | 0 | | | | | | | 0 | | |
| g10 | 1 | 1 | | | | | | | | 1 | |
| | 0 | X | | | | | | | | 0 | |
| | X | 0 | | | | | | | | 0 | |
| g11 | 1 | | | | | | | | | 1 | 1 |
| | 0 | | | | | | | | | X | 0 |
| | X | | | | | | | | | 0 | 0 |

Table 1. - Singular Covers

Next, we will need the primitive-d-cubes for the logical blocks.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| g7 | D | 1 | | | | | D | | | | |
| | 1 | D | | | | | D | | | | |
| g8 | D | 1 | | | | | | D | | | |
| | 1 | D | | | | | | D | | | |
| g9 | D | 1 | | | | | | | D | | |
| | 1 | D | | | | | | | D | | |
| g10 | D | 1 | | | | | | | | D | |
| | 1 | D | | | | | | | | D | |
| g11 | D | | | | | | | | | 1 | D |
| | 1 | | | | | | | | | D | D |

Table 2. - primitive d-cubes

The primitive d-cube of the failure is now computed. The singular covers for the failed gate is,

| | 8 | 9 | 10 |
|---|---|---|---|
| $c_1'$ = | 1 | 1 | 1 |
| $c_2'$ = | 0 | X | 1 |
| $c_3'$ = | X | 0 | 1 |

Taking the intersection of the singular covers for the failed and failure free gates we obtain,

| | 8 | 9 | 10 | |
|---|---|---|---|---|
| $c_2 \cap c_2'$ = | 0 | X | $\bar{D}$ | pdcf |
| $c_3 \cap c_3'$ = | X | 0 | $\bar{D}$ | |

The forward trace phase can now be performed. This results in Table 3.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | Label | Activity | Fan-out |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $TCO_1$ | | | | | 0 | X | $\bar{D}$ | | | | | $TCO_1$ | 10 | 11 |
| $TCO_1$ (d) 11 | | | | 1 | 0 | X | $\bar{D}$ | $\bar{D}$ | | | | $TCO_1$-1 | 11 | ø |
| $TCO_2$ | | | | | X | 0 | $\bar{D}$ | | | | | $TCO_1$ | 10 | 11 |
| $TCO_2$ (d) 11 | | | | 1 | X | 0 | $\bar{D}$ | $\bar{D}$ | | | | $TCO_2$-1 | 11 | ø |

Table 3. - Forward Trace

The back-trace for $TCO_1$-1 and $TCO_2$-1 is the only remaining task. This result is given in Table 4.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | Comments |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $TCO_1$-1 = B1 = | X | X | X | X | X | X | 1 | 0 | X | $\bar{D}$ | $\bar{D}$ | |
| B1 (d) $g8_a$ = $B2_a$ = | X | X | 0 | X | X | X | 1 | 0 | X | $\bar{D}$ | $\bar{D}$ | Either of these two rows can be chosen arbitrarily |
| B1 (d) $g8_b$ = $B2_b$ = | X | X | X | 0 | X | X | 1 | 0 | X | $\bar{D}$ | $\bar{D}$ | |
| $B2_a$ (d) g7 = $B3_a$ = | 1 | 1 | 0 | X | X | X | 1 | 0 | X | $\bar{D}$ | $\bar{D}$ | Same Choice as above. |
| $B2_b$ (d) g7 = $B3_b$ = | 1 | 1 | X | 0 | X | X | 1 | 0 | X | $\bar{D}$ | $\bar{D}$ | |
| $TCO_2$-1 = B1' = | X | X | X | X | X | X | 1 | X | 0 | D | D | |
| B1' (d) $g9_a$ = $B2_a'$ = | X | X | X | X | 0 | X | 1 | X | 0 | D | D | Either of these rows can be arbitrarily chose |
| B1' (d) $g9_b$ = $B2_b'$ = | X | X | X | X | X | 0 | 1 | X | 0 | D | D | |
| $B2_a'$ (d) g7 = $B3_a'$ = | 1 | 1 | X | X | 0 | X | 1 | X | 0 | D | D | Same choice as above. |
| $B2_b'$ (d) g7 = $B3_b'$ = | 1 | 1 | X | X | X | 0 | 1 | X | 0 | D | D | |

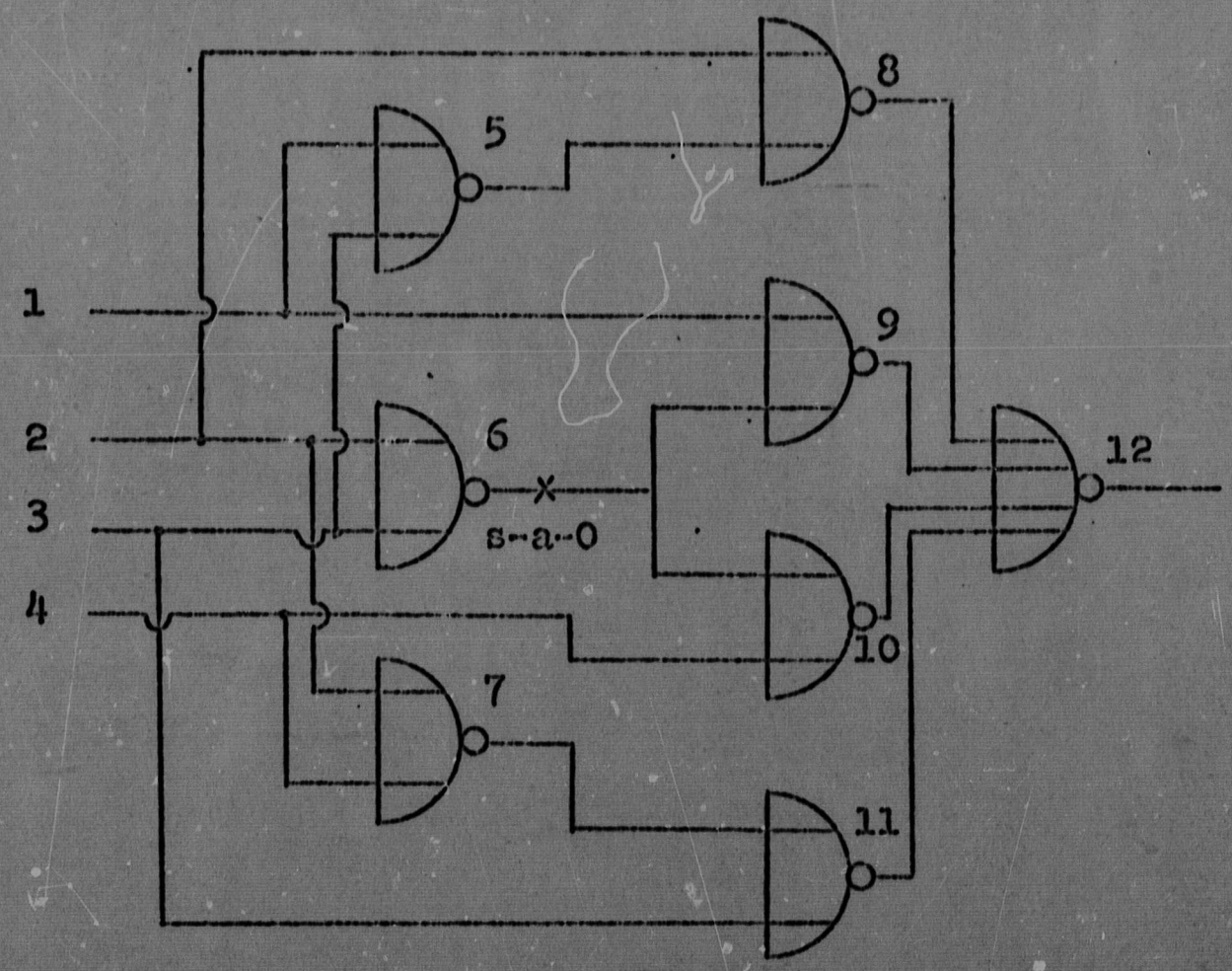
Table 4. - Back Trace

From the above table we can see that any of the tests;

$$1\ 1\ 0\ X\ X\ X = B3_a,$$
$$1\ 1\ X\ 0\ X\ X = B3_b,$$
$$1\ 1\ X\ X\ 0\ X = B3'_a,$$
$$1\ 1\ X\ X\ X\ 0 = B3'_b,$$

can be used to detect the fault. If any of these tests are applied to the good circuit the output (11) will be 0. Under the same test, with the fault present, the output (11) will be 1.

As a final example we will consider the circuit proposed by Schneider as a counter example to the one-dimensional path sensitizing technique. This circuit is given below.



First, we will compute the singular cover for the circuit.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| gate 5 | 0 | | 0 | | 1 | | | | | | | |
| | 1 | | X | | 0 | | | | | | | |
| | X | | 1 | | 0 | | | | | | | |
| gate 6 | 0 | 0 | | | | 1 | | | | | | |
| | 1 | X | | | | 0 | | | | | | |
| | X | 1 | | | | 0 | | | | | | |
| gate 7 | 0 | | 0 | | | | 1 | | | | | |
| | 1 | | X | | | | 0 | | | | | |
| | X | | 1 | | | | 0 | | | | | |
| gate 8 | 0 | | | | 0 | | | 1 | | | | |
| | 1 | | | | X | | | 0 | | | | |
| | X | | | | 1 | | | 0 | | | | |
| gate 9 | 0 | | | | | 0 | | | 1 | | | |
| | 1 | | | | | X | | | 0 | | | |
| | X | | | | | 1 | | | 0 | | | |
| gate 10 | 0 | | 0 | | | | | 1 | | | | |
| | 1 | | X | | | | | 0 | | | | |
| | X | | 1 | | | | | 0 | | | | |
| gate 11 | 0 | | | 0 | | | | 1 | | | | |
| | 1 | | | X | | | | 0 | | | | |
| | X | | | 1 | | | | 0 | | | | |
| gate 12 | 0 | 0 | 0 | 0 | 1 | | | | | | | |
| | 1 | X | X | X | 0 | | | | | | | |
| | X | 1 | X | X | 0 | | | | | | | |
| | X | X | 1 | X | 0 | | | | | | | |
| | X | X | X | 1 | 0 | | | | | | | |

Now, we need the primitive d-cubes of the logical block.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| gate 5 | $\bar{D}$ | | 0 | | L | | | | | | | |
| | 0 | | $\bar{D}$ | | D | | | | | | | |
| gate 6 | $\bar{D}$ | 0 | | | | D | | | | | | |
| | 0 | $\bar{D}$ | | | | D | | | | | | |
| gate 7 | $\bar{D}$ | | | 0 | | | D | | | | | |
| | 0 | | | $\bar{D}$ | | | D | | | | | |
| gate 8 | $\bar{D}$ | | | | 0 | | | D | | | | |
| | 0 | | | | $\bar{D}$ | | | D | | | | |
| gate 9 | $\bar{D}$ | | | | | 0 | | | D | | | |
| | 0 | | | | | $\bar{D}$ | | | D | | | |
| gate 10 | $\bar{D}$ | | | | | 0 | | | | D | | |
| | 0 | | | | | $\bar{D}$ | | | | D | | |
| gate 11 | $\bar{D}$ | | | | | 0 | | | | D | | |
| | 0 | | | | | $\bar{D}$ | | | | D | | |
| gate 12 | $\bar{D}$ | 0 | 0 | 0 | D | | | | | | | |
| | 0 | $\bar{D}$ | 0 | 0 | D | | | | | | | |
| | 0 | 0 | $\bar{D}$ | 0 | D | | | | | | | |
| | 0 | 0 | 0 | $\bar{D}$ | D | | | | | | | |

Assuming that output g is s-a-0 the primitive d-cube of the failure is computed from the singular cover for the failed gate.

| | 2 | 3 | 6 |
|---|---|---|---|
| $c_1^! =$ | 0 | 0 | 0 |
| $c_2^! =$ | 1 | X | 0 |
| $c_3^! =$ | X | 1 | 0 |

The primitive d-cube of the failure is given below.

| 2 | 3 | 6 | |
|---|---|---|---|
| $c_1 \frown c_1'$ | 0 | 0 | D (pdcf) |

We are now in position to begin the forward trace phase of the algorithm. See d-intersection table on following page.

The comment at TCO-1-2(d)12 implies that these inputs jointly control the output to 12. All the primitive d-cubes we generated assumed a single input controlling the output. Therefore, those cubes cannot be used and we have to generate a double d-cube with input 8 and 9 jointly controlling 12. To clarify this one remaining point we will consider the totality of primitive d-cubes for gate 12.

| | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|
| Single d-cubes | $\bar{D}$ | 0 | 0 | 0 | D |
| | 0 | $\bar{D}$ | 0 | 0 | D |
| | 0 | 0 | $\bar{D}$ | 0 | D |
| | 0 | 0 | 0 | $\bar{D}$ | D |
| Double d-cubes | $\bar{D}$ | $\bar{D}$ | 0 | 0 | D |
| | $\bar{D}$ | 0 | $\bar{D}$ | 0 | D |
| | $\bar{D}$ | 0 | 0 | $\bar{D}$ | D |
| | 0 | $\bar{D}$ | $\bar{D}$ | 0 | D |
| | 0 | $\bar{D}$ | 0 | $\bar{D}$ | D |
| | 0 | 0 | $\bar{D}$ | $\bar{D}$ | D |
| Triple d-cubes | $\bar{D}$ | $\bar{D}$ | $\bar{D}$ | 0 | D |
| | $\bar{D}$ | 0 | $\bar{D}$ | $\bar{D}$ | D |
| | 0 | $\bar{D}$ | $\bar{D}$ | $\bar{D}$ | D |
| | $\bar{D}$ | $\bar{D}$ | 0 | $\bar{D}$ | D |
| Quad. d-cube | $\bar{D}$ | $\bar{D}$ | $\bar{D}$ | $\bar{D}$ | D |

d-intersection

| Cycle 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | Activity | Fan-out | Label |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TC0 = | | 0 | 0 | | | D | | | | | | | 6 | 9,10 | TC0 |
| TC0@9 = | 0 | 0 | 0 | | | D | | | $\overline{D}$ | | | | 6,9 * | 9,10,12* | TC0-1-** |
| TC0@10 = | | 0 | 0 | 0 | | D | | | | $\overline{D}$ | | | 6,10 | 9,10,12 | TC0-2 |
| Cycle 2 | | | | | | | | | | | | | | | = |
| TC0-1@9 = | 0 | 0 | 0 | | | D | | | $\overline{D}$ | | | | 6,9 | 9,10,12 | TC0-1-1 |
| TC0-1@10 = | 0 | 0 | 0 | 0 | | D | | | $\overline{D}$ | $\overline{D}$ | | | 9,10 | 12 | TC0-1-2 |
| TC0-1@12 = | 0 | 0 | 0 | | | D | | 0 | $\overline{D}$ | 0 | 0 | D | 12 | ø | TC0-1-3 |
| Cycle 3 | | | | | | | | | | | | | | | = |
| TC0-2@9 = | 0 | 0 | 0 | 0 | | D | | | $\overline{D}$ | $\overline{D}$ | | | 6,10 | 9,10,12 | TC0-2-1 |
| TC0-2@10 = | | 0 | 0 | 0 | | D | | | | $\overline{D}$ | | | 6,10 | 9,10,12 | TC0-2-2 |
| TC0-2@12 = | | 0 | 0 | 0 | | D | | 0 | 0 | $\overline{D}$ | 0 | D | 12 | ø | TC0-2-3 |

Cycle 4

TC0-1-2@12 = ? It does not appear that this intersection is defined. However, if we look at TC0-1-2 we can see that vertex 9 and 10 are both inputs to 12 and they both have value $\overline{D}$.

---

** If we find a test cube that duplicates a cube specified earlier, it can be dropped.

* Note that the activity contains all vertices which are outputs having value D or $\overline{D}$. Also, the fan-out list contains all those gates which are potential extenders of all activity gates.

Roth provides an algorithm for computing multiple d-cubes. However, for standard gate types a table look-up seems faster.

Now we are in a position to look at TCO-1-2 ⓓ 12 where the intersection takes place with the double d-cube of 12. This gives,

| | d-intersection | | | | | | | | | | | | Acti- | Fan- | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | vity | out | Label |
| TCO-1-2 ⓓ 12 = | 0 | 0 | 0 | 0 | | D | | 0 | $\bar{D}$ | $\bar{D}$ | 0 | D | 12 | ∅ | TCO-1-2-1 |

The remaining portion of the procedure is to back trace for TCO-1-3, TCO-2-3, and TCO-1-2-1. We will individually perform the back-trace for each of these test cubes.

| | d-intersection | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| TCO-1-3 = | 0 | 0 | 0 | | | D | | 0 | D | 0 | 0 | D |
| TCO-1-3 ⓓ g11 = $B_1$ = | 0 | 0 | 0 | | | D | 1 | 0 | D | 0 | 0 | D |
| B1 ⓓ g10 = B2 = | 0 | 0 | 0 | 1 | | D | 1 | 0 | D | 0 | 0 | D |
| B2 ⓓ g8 = B3 = | 0 | 0 | 0 | 1 | 1 | D | 1 | 0 | D | 0 | 0 | D |
| B3 ⓓ g7 = B4 = | ∅, this implies that no test exists for TCO-1-3. | | | | | | | | | | | |

We will now proceed to TCO-2-3

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| TCO-2-3 = | 0 | 0 | 0 | | D | | 0 | 0 | $\bar{D}$ | 0 | D |
| TCO-2-3 ⓓ g11=B1 = | 0 | 0 | 0 | | D | 1 | 0 | 0 | $\bar{D}$ | 0 | D |
| B1 ⓓ g9 = B2 = | 1 | 0 | 0 | 0 | D | 1 | 0 | 0 | $\bar{D}$ | 0 | D |
| B2 ⓓ g8 = B3 = | 1 | 0 | 0 | 0 | 1 | D | 1 | 0 | 0 | $\bar{D}$ | 0 | D |
| B3 ⓓ g7 = B4 = | 1 | 0 | 0 | 0 | 1 | D | 1 | 0 | 0 | $\bar{D}$ | 0 | D |
| B4 ⓓ g5 = B5 = | ∅, same as above. | | | | | | | | | | | |

Finally we try TCO-1-2-1.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| TCO-1-2-1 = | 0 | 0 | 0 | 0 | | D | | 0 | $\bar{D}$ | $\bar{D}$ | 0 | D |
| TCO-1-2-1 ⓓ G11 =B1 = | 0 | 0 | 0 | 0 | | D | 1 | 0 | $\bar{D}$ | $\bar{D}$ | 0 | D |
| B1 ⓓ g8 = B2 = | 0 | 0 | 0 | 0 | 1 | D | 1 | 0 | $\bar{D}$ | $\bar{D}$ | 0 | D |
| B2 ⓓ g7 = B3 = | 0 | 0 | 0 | 0 | 1 | D | 1 | 0 | $\bar{D}$ | $\bar{D}$ | 0 | D |
| B3 ⓓ g5 = B4 = | 0 | 0 | 0 | 0 | 1 | D | 1 | 0 | $\bar{D}$ | $\bar{D}$ | 0 | D(Valid Test) |

Hence, we have been able to generate test (1,2,3,4) = (0,0,0,0) which we could not generate using the one-dimensional technique. This completes our discussion of the d-algorithm. However, a number of problems still exist with respect to the usefulness of multi-dimensional path sensitizing techniques. These will now be considered.

1.) If multi-dimensional path sensitizing is to be used practically it will most likely be used in conjunction with a digital logic simulator or even more practically as part of a total simulation and test generation system. The use in conjunction with a simulator (strictly combinational) has been described by Roth[2]. The use in conjunction with a total system has been described by Szygenda[5].

2.) After the discussion presented in this report it should be obvious that the technique described herein will need substantial modification before a practical program can be developed. During this modification, certain extensions can be made to enhance the total procedure. Szygenda and Goldbogen[6] (Appendix I) describe a working implementation of a multi-dimensional path sensitizing algorithm.

3.) The possibility exists that numerous tests could be generated by the test generator. This could lead to a considerable amount of redundant fault detection. Hence, some type of test minimization is likely to be required. A number of approaches have been suggested which would result in an absolute minimum number of required tests. Unfortunately, this technique would be extremely impractical.

to program for large logic nets. Szygenda[6] (Appendix 2) describes a procedure and program which implements an approximate method for test reduction. Experience with this technique has shown that near minimal sets of tests can be obtained.

4.) The techniques described in this report are only applicable to combinational logic nets. It seems likely that this technique could be extended to large sequential nets in a heuristic manner. This problem is presently under investigation.

FOR A BETTER COPY CONTACT DOCUMENT ORIGINATOR

## REFERENCES


1. Eldred, R. D., "Test Routines Based on Symbolic Logic Statements", Journal of ACM, Vol. 6, No. 1, 1959.

2. Armstrong, D. B., "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets", IEEE Trans. on Elec. Computers, Vol. EC-15, No. 1, 1966.

3. Schneider, P. R., "On the Necessity to Examine D-Chains in Diagnostic Test Generation - An Example", IBM Journal of Research and Development, Vol. II, No. 1, 1967.

4. Roth, J. P., "Diagnosis of Automata Failures: A Calculus and A Method", IBM Journal of Research and Development, Vol. 10, 1966.

5. S. A. Szygenda, "A Software Diagnostic System for Test Generation and Simulation of Large Digital Systems", Proceedings of the National Electronics Annual Conference, December, 1969.

6. S. A. Szygenda and G. Goldbogen, "Implementation and Extensions of Multi-Dimentional Path Sensitizing in a Simulation and Diagnosis System", Proceedings of the 7th Annual Allerton Conference on Circuit and System Theory, October 1969.

7. S. A. Szygenda, "Minimization of Fault Detection Tests - An Approximate Method", BTL Report MF 14-660712.1, July 1966.

8. Chang, Manning, & Metze, "Diagnosis of Digital Computers," Book in Preparation.

# Test Routines Based on Symbolic Logical Statements*

RICHARD D. ELDRED

*Datamatic, Newton Highlands, Massachusetts*

## 1. *Introduction*

In order for the successful operation of a test routine to guarantee that a computing system has no faulty components, the test conditions imposed by the routine should be devised at the level of the components themselves, rather than at the level of programmed orders. Therefore, it seems that the proper approach to writing test or maintenance routines is by way of the logical diagrams of the system, and not the list of machine orders. This is the only way in which all conditions of operation of each logical function can be uniquely and completely defined and all logical components within each logical function can be made to perform the task to which they are assigned. Orders can then be programmed to present all conditions, and further orders can be programmed to detect improper performance of the logical functions, thereby producing a minimum program which tests and detects failure in each logical component in the system.

## 2. *Background and Definitions*

A method of producing such a test routine was developed and applied to the Central Processor of the Datamatic 1000 system. The logical functions in the Datamatic 1000 were implemented by tubes and diodes. The functions are in the form of buffer-gate-buffer and were originally represented by logical statements. Since almost any system is originally represented by logical statements, the method described here could be applied equally well to another system.

## 3. *Description of Method*

A method for selecting the minimum number of test conditions necessary to determine whether all tubes and diodes within a gating structure are operating properly will now be described. External items, such as auxiliary amplifiers, delay circuits, transmission lines, and so forth, are also tested by the tube-diode tests for the gating structures with which they are associated.

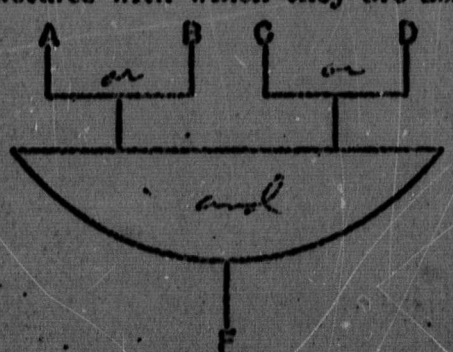



FIG. 1

* Presented at the meeting of the Association, Aug. 11–13, 1958.

Consider a simple buffer-gate-buffer structure as shown in figure 1. In logical notation this is represented by $F = (A \vee B)\cdot(C \vee D)$. It is desired to present a minimum set of conditions to this function to determine whether each component within this function is operating properly. Specifically, it is required to test:

(1) That each input buffer diode conducts when activated. Buffer inputs will hereafter be referred to as *or* inputs.

(2) That each gate leg diode inhibits when not activated. Gate legs will hereafter be referred to as *and* legs.

(3) That the output of the gating structure is not lost when traveling via output components.

(4) That the output components do not provide a signal when the input conditions are inappropriate.

Suppose we initially apply the input combination $A\cdot\bar{B}\cdot C\cdot\bar{D}$ which should activate the function and then check for an output at F. This checks requirement (1) for *or* inputs A and C. It also checks requirement (3). Now shift to a different combination of input variables, changing the active inputs at every *and* leg whenever it is possible to do so, but always keeping one, and only one, *or* input variable active to each *and* leg so that this input buffer diode bears the sole responsibility for activating the gate leg. This gives us the second input combination $\bar{A}\cdot B\cdot\bar{C}\cdot D$ which checks requirement (1) for *or* inputs B and D, and again condition (3).

Now apply the combination $\bar{A}\cdot\bar{B}\cdot(C \vee D)$, which should not activate the function, and check for no output at F. This checks condition (2) for *and* leg $(A \vee B)$ and also condition (4). Now shift to a different combination of input variables activating in turn all *and* legs except one so that each *and* leg has the sole responsibility for inhibiting the output of the function. This gives us the combination $(A \vee B)\cdot\bar{C}\cdot\bar{D}$, which tests the $(C \vee D)$ *and* leg and again condition (4).

In summary then, we have two sets of tests: the activation tests and the inhibition tests. The activation tests for the example given are:

$$\bar{A}\cdot B\cdot\bar{C}\cdot D \text{ and } A\cdot\bar{B}\cdot C\cdot\bar{D}.$$

The inhibition tests are:

$$\bar{A}\cdot\bar{B}\cdot(C \vee D) \text{ and } (A \vee B)\cdot\bar{C}\cdot\bar{D}.$$

To briefly consider a slightly more complex example, take the function represented in figure 2.

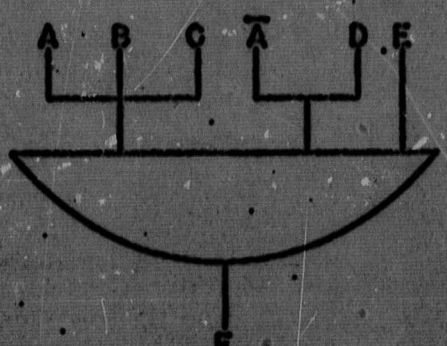



Fig. 2

The activation tests are as follows:

(1) $A \cdot \bar{B} \cdot \bar{C} \cdot D \cdot E$

(2) $\bar{A} \cdot B \cdot \bar{C} \cdot \bar{D} \cdot E$

(3) $\bar{A} \cdot \bar{B} \cdot C \cdot E$ (The state of D is arbitrary.)

For the inhibition tests we have:

(1) $\bar{A} \cdot \bar{B} \cdot \bar{C} \cdot E$ (The state of D is arbitrary.)

(2) $A \cdot \bar{D} \cdot E$ (The state of B and C is arbitrary.)

(3) $(A \vee B \vee C) \cdot (\bar{A} \vee D) \cdot \bar{E}$.

In activation test (1), we hold A active and B and C inactive, thereby testing *or* input A which now has the sole responsibility for activating the gate leg. We already have A active so we hold D active to test *or* input D. E, of course, must be active to activate the function. In test (2), we switch to holding B active and A and C inactive, and since we already have $\bar{A}$ we inactivate D to test *or* input $\bar{A}$. In test (3) we switch to C being active and since the second *and* leg has already been fully tested, the condition of D is arbitrary. In inhibition test (1), we hold the first *and* leg inactive by the combination $\bar{A} \cdot \bar{B} \cdot \bar{C}$. The second leg is now active by $\bar{A}$ so that the condition of D is arbitrary. E holds the third leg active. Gate leg 1 has now been tested for inhibition as it has the sole responsibility for inactivating the function. In test (2), $A \cdot \bar{D}$ holds the second *and* leg inactive. A and E hold the other two legs active so that the condition of B and C is arbitrary. In test (3), $\bar{E}$ holds the third leg inactive, and, as can be seen, any of several combinations will hold the other two legs active.

The illustrated examples were for one gate functions. If a function contains more than one gate, the tests are applied to each gate individually.

It can be seen that the maximum number of activation tests for a function with *or* inputs is equal to the total number of *or* inputs. The minimum number of tests for such a function is equal to the largest number of *or* inputs on one *and* leg. If there are no *or* inputs then the number of activation tests equals one. The number of inhibition tests for any function is always equal to the number of *and* legs. Whether the number of activation tests is at a maximum or minimum, of course, depends on any logical relationship which exists between the input variables.

### 4. *Application of Method*

This method was applied to develop a test routine for the high speed packages of the central processor of the Datamatic 1000 system. Charts were prepared and filled out by subsystem for each logical function in the system. The columns in the chart were defined to contain the following information:

(1) Symbolic name of logical function.

(2) The logical combinations necessary for all activation and inhibition tests.

(3) The necessary programmed order or orders to produce the logical combination. This included the name of the order and any requirements of addresses, operands, previous orders, and so forth, to produce the necessary logical combination.

(4) How the failure of the logical function under these conditions would affect the operation of the machine, assuming no built-in machine checking.

(5) Whether or not this failure would be detected by built-in machine checking.

(6) How the failure of the logical function under these conditions would affect the expected operation of the program, assuming no machine checking.

(7) Whether or not the effect on the program can be detected by a programmed check.

Every logical function in the system was listed in this manner, with the exception of some second and third level checking functions which assumed an error in their input conditions. Since a program is not capable of knowingly producing incorrect data, it was not possible to list these functions.

In order to write the test program then, it was necessary only to write a minimum set of orders which produced all the logical combinations in column (3). This guaranteed that all required test conditions were presented to all logical functions. To test the checking functions which could not be listed, a programmed check was placed opposite each test order. The programmed checks were defined by the effect of "the failure on the program" as listed in column (6).

### 5. *Results*

The result of this endeavor was a program containing approximately 110 test orders and an additional 200 machine words comprising constants, control, and programmed checks. This program checked approximately 700 high speed and associated packages containing 900 tubes and 21,000 diodes, for all conditions of activation and inhibition. The total time for one pass through this program is 0.08 seconds. From these specifications, it can be seen that this program is well suited for operation in conjunction with marginal checking.

This program has been used successfully during the debugging and production stages of the Datamatic 1000 and is now being used in the field as the principle maintenance routine for the Datamatic 1000 Central Processor.

It is recommended that the writing of a program such as this be done in conjunction with the logical design checking of the various machine units as it is a natural by-product of such a study.

# DERIVATION OF OPTIMUM TESTS TO DETECT FAULTS IN COMBINATIONAL CIRCUITS*

J. F. Poage
Department of Electrical Engineering
Princeton University, Princeton, New Jersey

Permanent component failures or faults within a combinational logic network may cause a network to realize an output function different from that for which it was designed. Boolean algebra, while sufficient to describe the normal operation of a network, gives no indication of the effect that a fault will have upon a network output function. To overcome this deficiency, an analysis procedure is developed for combinational logic networks which yields an output expression incorporating network structure in addition to input variables. From this one output expression it is possible to obtain the network output function when any permanent fault, single or multiple, exists within the network.

The output expression obtained from the analysis procedure is ideally suited to the generation of fault detection tests. Optimum tests to detect single faults may be derived by using the output expression to obtain the network output function for each single fault. A second method, developed in this paper, uses the output expression to list the set of faults, single and multiple, which a particular input combination may detect. Using the fault sets for all input combinations, an efficient procedure is developed for obtaining not only single fault detection tests but also tests which guarantee detection of all multiple faults as well.

## I. INTRODUCTION

The normal operation of a combinational logic network may be disturbed by a large class of faults, many defying the imagination of the most pessimistic trouble shooter. Figure 1(a) illustrates a simple gate network normally realizing the function $wx + yz$. If wire f has a permanent signal of zero on it (denoted $(f_0)$), the network of Fig. 1(a) is transformed to that of Fig. 1(b). Under these circumstances, the original network is said to have a single fault. Figure 1(c) illustrates a network equivalent to the original network in the presence of a single fault where wire a has a permanent signal of one (denoted $(a_1)$). The equivalent network for multiple fault $(b_0c_1)$ consisting of wire b with a permanent zero and wire c with a permanent one is shown in Fig. 1(d). These three faults are but a few of many possible for the network. They all have the property

* This work was performed while the author held a Bell Telephone Laboratories Employee Fellowship at Princeton University.

Presented at the *Symposium on Mathematical Theory of Automata*, Polytechnic Institute of Brooklyn, April 24, 25, 26, 1962

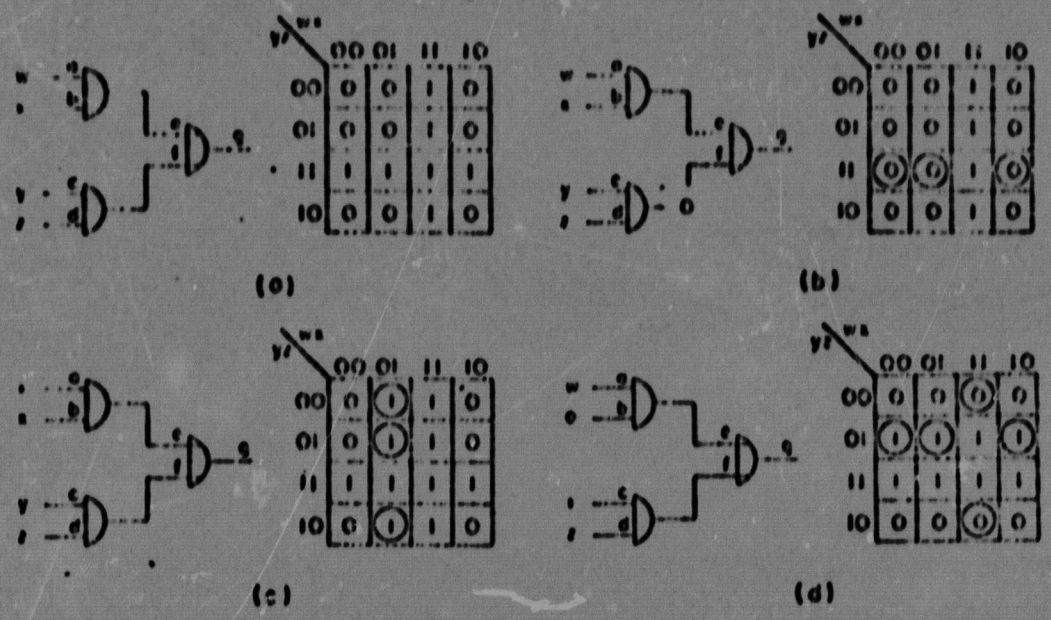



Fig. 1 The effect of faults in a gate network. (a) The normal network; (b) Wire f with a permanent signal of 0; (c) Wire a with a permanent signal of 1; (d) Wire b with a permanent signal of 0, and wire c with a permanent signal of 1.

that the function realized by the network is permanently changed because of the fault. The respective functions are seen to be wx, x+yz, and z. As seen from the Karnaugh maps of Fig. 1, input combinations w'xyz and w'xy'z will detect the faults. An output of zero resulting from application of, w'xyz detects fault ($f_0$). An output of one from input combination w'xy'z detects both fault ($a_1$) and fault ($b_0c_1$). However, w'xyz and w'xy'z are not sufficient to diagnose the faults. The response of the network with fault $a_1$ to w'xyz and w'xy'z is identical to the response of the network with fault $b_0c_1$. An additional input combination such as w'xy'z' is needed to complete the diagnostic test.

A test to detect and diagnose faults in a logic network consists of the application of a set of input combinations and a corresponding observation of the resulting outputs. Algorithms for generating such tests have been described.[1, 2] They generally require an analysis of the network for each possible fault to obtain the output function realized. A systematic comparison of these functions yields the test. Clearly, such algorithms are most useful when the set of reasonably probable faults is small. Typically, only single faults are considered.

The object of the following discussion is to present a method of network analysis which describes a network in a form particularly suitable for generating detection and diagnostic tests. The method's application to fault detection tests for gate networks and contact networks will be considered in detail. Using the network description provided by the analysis method, it is possible to construct a shortest or optimum test to detect single and multiple faults without computing

the output function caused by each possible fault. Methods used in generating detection tests are directly applicable to the generation of diagnostic tests. Some possibilities in this direction will be considered in subsequent reports.

## II. ANALYSIS OF LOGICAL NETWORKS

### Terminology and Definitions.

Figure 2 illustrates the block diagram of a logic network. There are n input lines and m output lines where n and m are not necessarily equal. All networks considered are combinational; i. e., output signals from a network are dependent only upon the present values of the input signals. All input signals to a network and output signals from a network are assumed to be binary. The symbols 0 and 1 will be used to represent values for these signals.

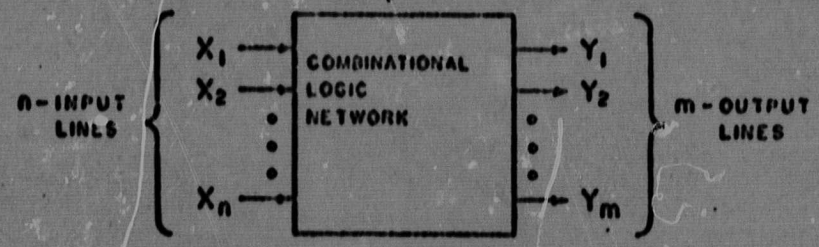



Fig. 2 Block diagram for combinational logic network

Generally, the analysis of a combinational logic network yields m Boolean functions describing the network outputs in terms of the network inputs. For the purposes of fault detection and diagnosis as well as the study of transient behavior in combinational networks, [3] an analysis of this sort is not sufficient. Output expressions dependent upon both network structure and network inputs are required. This section presents a method of network analysis which recognizes relevant network structure by including information regarding the elements of a network in the form of logical propositions.

DEFINITION: Associated with each input line $X_i$ are two input propositions, $x_i$ and $x_i'$, defined as follows:

$x_i$: the signal applied to input line $X_i$ has a value of 1;

$x_i'$: the signal applied to input line $X_i$ has a value of 0.

Each of the $2^n$ input states or input combinations will be denoted as $\underline{x}^k$ where $0 \leq k \leq 2^n-1$. The number k may be thought of as a binary number whose ith digit is the same as the signal value applied to input line $X_i$.

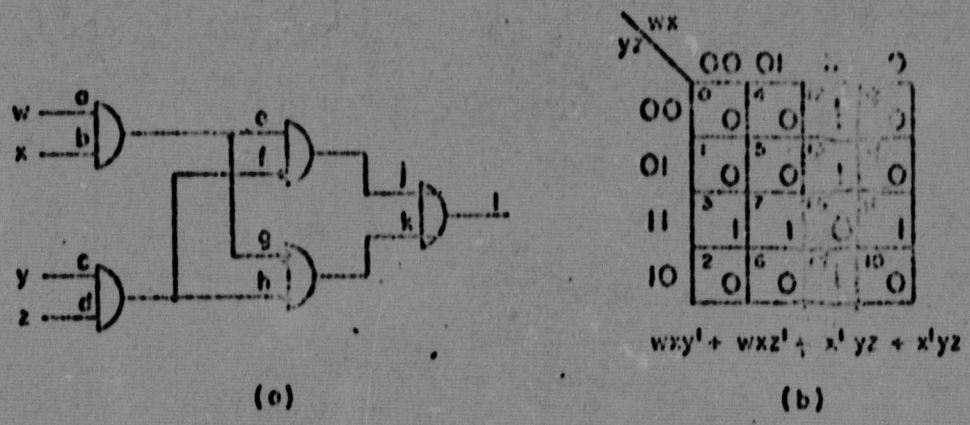



Fig. 3 Gate network. (a) Circuit diagram; (b) Karnaugh map.

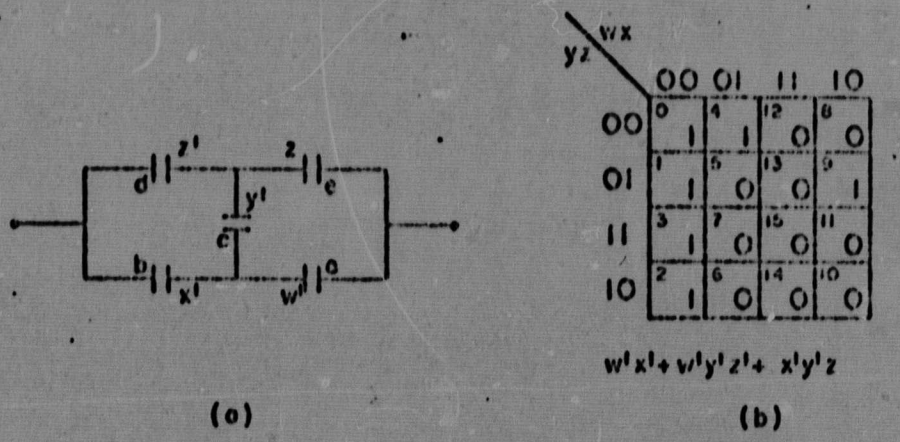



Fig. 4 Contact network. (a) Circuit diagram; (b) Karnaugh map.

A logical network may be thought of as a collection of physical elements $\{E\}$ whose members are denoted $e_1, e_2, \ldots, e_q$. Elements of $\{E\}$ have the property that they, too, are binary by nature. In gate networks the physical elements in $\{E\}$ are the wires connected to gate inputs as shown in Fig. 3. Signals on the wires may switch between two levels, the values of which are represented by the symbols 0 and 1. The set $\{E\}$ for contact networks consists of the contacts themselves, as illustrated in Fig. 4. A contact may switch between an open and shorted condition denoted by the letters o and s.

DEFINITION: A physical element (wire) of a gate network may exist in any one of three element states.

1. The wire is normal, in which case its signal may switch between the value 1 and the value 0.
2. The wire is permanently 1 if a signal of constant value 1 appears on the wire.

3. The wire is permanently 0 if a signal of constant value 0 appears on the wire.

DEFINITION: Associated with each element (wire) $e_i$ in a gate network are three element propositions:

$e_{in}$: wire $e_i$ is normal;

$e_{i1}$: wire $e_i$ is permanently 1;

$e_{i0}$: wire $e_i$ is permanently 0.

ASSUMPTION: It is assumed in the preceding definitions and throughout this analysis that permanent failures occurring in a gate network may be regarded as one or more permanent signals appearing on wires of the network. Gates are always assumed to operate perfectly even though one or more of their input wires may have permanent signals. [1]

Further, it is assumed that combinational gate networks are loop free. Each wire has one end connected to either a network input or a gate output and the other end connected to a network output or a gate input.

DEFINITION: A physical element (contact) of a contact network may exist in any one of three element states.

1. The contact is normal, in which case it may switch between the open and shorted condition.
2. The contact is permanently shorted.
3. The contact is permanently open.

DEFINITION: Associated with each element (contact) $e_j$ in a contact network are three element propositions defined as:

$e_{jn}$: contact $e_j$ is normal;

$e_{js}$: contact $e_j$ is permanently shorted;

$e_{jo}$: contact $e_j$ is permanently open.

Elements in logical networks other than gate networks and contact networks will have analogous element propositions.

The procedure presented in this section for analyzing a logical network consists of deriving an output proposition, $P'_{out}$, for each of the $m$ output lines in terms of input propositions and element propositions. Upon substitution of truth values for the input propositions and element propositions, the output proposition for output line $Y_i$ will take the value of T or F corresponding to an output signal of 1 or 0 respectively on line $Y_i$. If the truth values for only element propositions are inserted, the resulting output proposition

for line $Y_j$ is in terms of input propositions only. Then, if the input propositions are regarded as Boolean variables, the output proposition may be regarded as the Boolean function realized by the network on output line $Y_j$.

## Analysis of Gate Networks.

The network of Fig. 3 will be used to illustrate the procedure for analyzing gate networks. As with ordinary network analysis, each output line from an m output network may be analyzed separately.

TABLE 1: ANALYSIS IN TERMS OF INPUT PROPOSITIONS

| Wire | Proposition | Complementary Proposition |
|---|---|---|
| a | $w$ | $w'$ |
| b | $x$ | $x'$ |
| c | $y$ | $y'$ |
| d | $z$ | $z'$ |
| e | $wx$ | $w'+x'$ |
| f | $yz$ | $y'+x'$ |
| g | $wx$ | $w'+x'$ |
| h | $yz$ | $y'+z'$ |
| j | $wx(y'+z')$ | $w'+x'+yz$ |
| k | $yz(w'+x')$ | $y'+z'+wx$ |
| l | $f_{out} = wx(y'+z')+yz(w'+x')$ | $f'_{out} = (w'+x'+yz)(y'+z'+wx)$ |
| | $= wxy' + wxz' + w'yz + x'yz$ (Sum of Products Form) | $= w'y' + w'z' + x'y' + x'z' + wxyz$ (Sum of Products Form) |

Thus, only a single output network will be considered. A standard analysis in terms of input propositions only might proceed along the lines indicated in Table 1. A proposition describing the signal on each wire is found from the propositions for gate inputs and the logical functions realized by each gate. For example, the proposition for wire a is w, and the proposition for wire b is x. Wire e is the output of an AND gate driven by wires a and b. The proposition for wire e is then wx. Continuing in this fashion as indicated by Table 1, an output proposition for the network is finally derived. The output proposition has a value T for those input combinations

leading to a network output of 1. A complementary output proposition is also derived in Table 1. It has a value T for input combinations leading to a network output of 0. For convenience, both the output proposition and its complement are multiplied out* to obtain a sum-of-products form.

If a permanent signal of value 1 or value 0 appears on one or more input wires to a gate, the logical function realized by the gate will be changed. To account for these possible changes, a more complex proposition is required for each wire. Propositions whose truth value more accurately describes the signals on wire a are:

$(w\,a_n + a_1)$: a signal of value 1 appears on wire a;

$(w'a_n + a_0)$: a signal of value 0 appears on wire a.

Propositions w and w' are input propositions; $a_n$, $a_1$ and $a_0$ are element propositions. Using a similar form, propositions to describe the signals on wire b are:

$(x\,b_n + b_1)$: a signal of value 1 appears on wire b;

$(x'b_n + b_0)$: a signal of value 0 appears on wire b.

For convenience, define the following general propositions:

$P_{e_i}$: (A signal of value 1 appears on wire $e_i$).

The complementary proposition $P'_{e_i}$ has value T when a signal of value 0 appears on wire $e_i$. Hence, we may write:

$$P_a = (w\,a_n + a_1);$$

$$P'_a = (w'a_n + a_0);$$

$$P_b = (x\,b_n + b_1);$$

$$P'_b = (x'b_n + b_0).$$

Further, from Fig. 3 it is seen that propositions for wire e are

$$P_e = (P_a P_b e_n + e_1) = \left[(wa_n + a_1)(xb_n + b_1)e_n + e_1\right]$$

$$P'_e = \left[(P'_a + P'_b)e_n + e_0\right] = \left\{\left[(w'a_n + a_0) + (x'b_n + b_0)\right]e_n + e_0\right\}.$$

Utilizing theorems $X \cdot X = X$, $X + XY = X$, $(X+Y)Z = XZ + YZ$, and $X + X = X$, equivalent propositions with a more convenient structure may be obtained.

* A proposition is "multiplied out" to a sum of products form by first applying the theorem $X(Y+Z) = XY + YZ$ to obtain a sum of products. Theorems $X \cdot X' = F$, $X + X' = T$, $X \cdot X = X$, and $X + XY = X$ are used to eliminate redundant terms.

$$P_e = (P_a e_n + e_1)\ (P_b e_n + e_1) = \left[(w\,a_n + a_1)e_n + e_1\right]\left[(x\,b_n + b_1)e_n + e_1\right]$$

$$P'_e = (P'_a e_n + e_0) + (P'_b e_n + e_0) = \left[(w'a_n + a_0)e_n + e_0\right] + \left[(x'b_n + b_0)e_n + e_0\right] \tag{1}$$

Suppose wires a, b, and e are normal. A truth value T is assigned to element propositions $a_n$, $b_n$ and $e_n$ of Eq. (1), while F is assigned to $a_1$, $b_1$, $e_1$, $a_0$, $b_0$ and $e_0$. Using theorems $X \cdot T = X$ and $X + F = X$, proposition $P_e$ simplifies to wx and $P'_e$ simplifies to $w' + x'$. These are simply the propositions found for wire e in Table 1.

Suppose, now, that wires a and e are normal but a permanent signal of value 1 appears on wire b. Element propositions $a_n$, $b_1$ and $e_n$ have truth value T while F is assigned to $b_n$, $a_1$, $e_1$, $a_0$, $b_0$ and $e_0$. Proposition $P_e$ simplifies to w, and $P'_e$ simplifies to w'. From Fig. 3 it may be verified that input proposition w then describes the signals appearing on wire e when a permanent signal of value 1 appears on wire b.

Using similar procedures, propositions for output wires of gates other than AND gates may be found. Propositions $P_\ell$ and $P'_\ell$ for the output of the OR gate in Fig. 3 are given in terms of $P_j$ and $P_k$ as

$$P_\ell = \left[(P_j + P_k)\ell_n + \ell_1\right] = (P_j \ell_n + \ell_1) + (P_k \ell_n + \ell_1)$$

and

$$P'_\ell = \left[P'_j P'_k \ell_n + \ell_0\right] = (P'_j \ell_n + \ell_0)(P'_k \ell_n + \ell_0). \tag{2}$$

Propositions $P_j$ and $P'_j$ corresponding to the output of an inhibit gate may be written in terms of propositions $P_e$ and $P_f$:

$$P_j = \left[P_e P'_f j_n + j_1\right] = (P_e j_n + j_1)(P'_f j_n + j_1)$$

$$P'_j = \left[(P'_e + P_f)j_n + j_0\right] = (P'_e j_n + j_0) + (P_f j_n + j_0). \tag{3}$$

Equations (1), (2) and (3) illustrate the following theorem.

THEOREM 1: All propositions $P_{e_i}$ and $P'_{e_j}$ may be written in the forms:

$$P_{e_i} = \left[Q e_{in} + e_{i1}\right] \text{ and } P'_{e_j} = \left[R e_{jn} + e_{j0}\right].$$

where Q(R) is a proposition which has value T when the signal applied to wire $e_i(e_j)$ has the value 1(0).

PROOF: If $e_i(e_j)$ is a network input, then Q(R) is an input proposition. If $e_i(e_j)$ is driven by a gate output, then Q(R) is found from the propositions for the gate inputs because the gates themselves are always assumed to operate correctly. Clearly, if wire $e_i(e_j)$ is normal, then $P_{e_i}(P'_{e_j})$ must reduce to Q(R). If $e_i(e_j)$ is permanently 1(0), then $P_{e_i}(P'_{e_j})$ must reduce to T. Finally, if $e_i(e_j)$ is permanently 0(1), then $P_{e_i}(P'_{e_j})$ must reduce to F. The forms in the theorem satisfy these conditions.

THEOREM 2: The complements of propositions $P_{e_i}$ and $P'_{e_j}$ have the form:

$$\left(P_{e_i}\right)' = \left[Qe_{in} + e_{i1}\right]' = \left[Q'e_{in} + e_{i0}\right]$$

$$\left(P'_{e_j}\right)' = \left[Re_{jn} + e_{j0}\right]' = \left[R'e_{jn} + e_{j1}\right]$$

PROOF: Applying de Morgan's theorem,

$$\left(P_{e_i}\right)' = \left[Qe_{in} + e_{i1}\right]' = \left[Q' + (e_{in})'\right](e_{i1})';$$

$$\left(P'_{e_j}\right)' = \left[Re_{jn} + e_{j0}\right]' = \left[R' + (e_{jn})'\right](e_{j0})'.$$

Since elements may be in one of three states, complements of element propositions are:

$$(e_{in})' = e_{i1} + e_{i0} \ ;$$

$$(e_{i1})' = e_{in} + e_{i0} \ ;$$

Therefore,

$$(e_{i0})' = e_{in} + e_{i1} \ .$$

$$\left(P_{e_i}\right)' = \left[Q' + e_{i1} + e_{i0}\right](e_{in} + e_{i0}) = \left[Q'e_{in} + Q'e_{i0} + e_{i0}\right]$$

$$= Q'e_{in} + e_{i0}$$

$$\left(P'_{e_j}\right)' = \left[R' + e_{j1} + e_{j0}\right](e_{jn} + e_{j1}) = \left[R'e_{jn} + R'e_{j1} + e_{j1}\right]$$

$$= R'e_{jn} + e_{j1} \ .$$

Hence, we may conclude that

$$\left[Qe_{in}+e_{j1}\right]' = \left[Q'e_{in}+e_{j0}\right]$$

and

$$\left[Re_{jn}+e_{j0}\right]' = \left[R'e_{jn}+e_{j1}\right].$$

An analysis of the gate network of Fig. 3 in terms of input propositions and element propositions is shown in Table 2. The analysis proceeds in a manner similar to that of Table 1. Examination of the output proposition, $P_{out} = P_\ell$, reveals that it is composed of propositions having a nested structure typified by Eq. (4):

TABLE 2: ANALYSIS IN TERMS OF INPUT PROPOSITIONS AND ELEMENT PROPOSITIONS

| Wire | Proposition |
|---|---|
| a | $P_a = wa_n + a_1$ |
| b | $P_b = xb_n + b_1$ |
| c | $P_c = yc_n + c_1$ |
| d | $P_d = zd_n + d_1$ |
| e | $P_e = P_a P_b e_n + e_1 = [(wa_n + a_1)e_n + e_1] \cdot [(xb_n + b_1)e_n + e_1]$ |
| f | $P_f = P_c P_d f_n + f_1 = [(yc_n + c_1)f_n + f_1] \cdot [(zd_n + d_1)f_n + f_1]$ |
| g | $P_g = P_a P_b g_n + g_1 = [(wa_n + a_1)g_n + g_1] \cdot [(xb_n + b_1)g_n + g_1]$ |
| h | $P_h = P_c P_d h_n + h_1 = [(yc_n + c_1)h_n + h_1] \cdot [(zd_n + d_1)h_n + h_1]$ |
| j | $P_j = P_e P'_f j_n + j_1 = \{[(wa_n + a_1)e_n + e_1]j_n + j_1\} \cdot \{[(xb_n + b_1)e_n + e_1]j_n + j_1\}$ $\cdot (\{[(y'c_n + c_0)f_n + f_0]j_n + j_1\} + \{[(z'd_n + d_0)f_n + f_0]j_n + j_1\})$ |
| k | $P_k = P_h P'_g k_n + k_1 = \{[(yc_n + c_1)h_n + h_1]k_n + k_1\} \cdot \{[(zd_n + d_1)h_n + h_1]k_n + k_1\}$ $\cdot (\{[(w'a_n + a_0)g_n + g_0]k_n + k_1\} + \{[(x'b_n + b_0)g_n + g_0]k_n + k_1\})$ |
| ℓ | $P_\ell = (P_j + P_k)\ell_n + \ell_1 = (P_j \ell_n + \ell_1) + (P_k \ell_n + \ell_1) = P_{out}$ $= (([[(wa_n + a_1)e_n + e_1]j_n + j_1)\ell_n + \ell_1) \cdot (([[(xb_n + b_1)e_n + e_1]j_n + j_1)\ell_n + \ell_1)$ $\cdot \{(([[(y'c_n + c_0)f_n + f_0]j_n + j_1)\ell_n + \ell_1) + (([[(z'd_n + d_0)f_n + f_0]j_n + j_1)\ell_n + \ell_1)\}$ $+ (([[(yc_n + c_1)h_n + h_1]k_n + k_1)\ell_n + \ell_1) \cdot (([[(zd_n + d_1)h_n + h_1]k_n + k_1)\ell_n + \ell_1)$ $(([[(w'a_n + a_0)g_n + g_0]k_n + k_1)\ell_n + \ell_1) + (([[(x'b_n + b_0)g_n + g_0]k_n + k_1)\ell_n + \ell_1)\}$ |

$$\left(\left\{\left[(y'c_n + c_0)f_n + f_0\right]j_n + j_1\right\}\ell_n + \ell_1\right). \tag{4}$$

DEFINITION: A proposition containing a single input proposition (literal) and having the nested structure such as (4) is called a literal proposition.

A convenient compressed notation for the literal proposition of (4) is

$$y'(c_0 f_0 j_1 \ell_1).$$

Clearly, any literal proposition may be represented in a compressed notation of this form because of the regularity of the nested structure. Hence,

$$w(a_1 g_1 k_0 \ell_0) = \left(\left\{\left[(wa_n + a_1)g_n + g_1\right]k_n + k_0\right\}\ell_n + \ell_0\right)$$

THEOREM 3. Propositions for wires of a gate network may always be written in terms of literal propositions.

PROOF. From Theorems 1 and 2, the propositions for a wire may be written as

$$P_{e_i} = \left[Q\, e_{in} + e_{i1}\right]$$

and

$$P'_{e_i} = \left[Q' e_{in} + e_{i0}\right]$$

If $e_i$ is an input wire, then $Q(Q')$ must be an input proposition, in which case the theorem is satisfied.

If $e_i$ is driven by a gate output, then $Q$ and $Q'$ are found from the propositions for the gate input wires. Suppose the propositions for all of the gate input wires satisfy the theorem. Then $Q$ and $Q'$ can also be written in terms of literal proposition. Using theorem $(X+Y)Z = XZ+YZ$, $X \cdot X = X$, $XY+Z = (X+Z)(Y+Z)$ and $X+X = X$, $P_{e_i}$ and $P'_{e_i}$ may also be written in terms of literal propositions. Hence, if the propositions for gate input wires satisfy the theorem, propositions for wires driven by the gate also satisfy the theorem. Equations (1), (2) and (3) provide illustrations.

The assumption was made that each wire of a combinational network has one end connected to either a network input or a gate output and the other end connected to a network output or a gate input. From this assumption of structure and the fact that propositions for input wires have the form of literal propositions, we may conclude that the propositions for all wires of a network may be written in terms of literal propositions.

TABLE 3: DERIVATION OF $P_{out}$ IN COMPRESSED NOTATION

| Wire | Proposition |
|---|---|
| a | $P_a = w(a_1)$ |
| b | $P_b = x(b_1)$ |
| c | $P_c = y(c_1)$ |
| d | $P_d = z(d_1)$ |
| e | $P_e = P_a P_b e_n + e_1 = w(a_1 e_1) x(b_1 e_1)$ |
| f | $P_f = P_c P_d f_n + f_1 = y(c_1 f_1) z(d_1 f_1)$ |
| g | $P_g = P_a P_b g_n + g_1 = w(a_1 g_1) x(b_1 g_1)$ |
| h | $P_h = P_c P_d h_n + h_1 = y(c_1 h_1) z(d_1 h_1)$ |
| j | $P_j = P_e P'_f j_n + j_1 = w(a_1 e_1 j_1) x(b_1 e_1 j_1) \left[ y'(c_0 f_0 j_1) + z'(d_0 f_0 j_1) \right]$ |
| k | $P_k = P_h P'_g k_n + k_1 = y(c_1 h_1 k_1) z(d_1 h_1 k_1) \left[ w'(a_0 g_0 k_1) + x'(b_0 g_0 k_1) \right]$ |
| $\ell$ | $P_\ell = P_j \ell_n + \ell_1 + P_k \ell_n + \ell_1 = P_{out}$ |
| | $= w(a_1 e_1 j_1 \ell_1) x(b_1 e_1 j_1 \ell_1) \left[ y'(c_0 f_0 j_1 \ell_1) + z'(d_0 f_0 j_1 \ell_1) \right]$ |
| | $+ y(c_1 h_1 k_1 \ell_1) z(d_1 h_1 k_1 \ell_1) \left[ w'(a_0 g_0 k_1 \ell_1) + x'(b_0 g_0 k_1 \ell_1) \right]$ |
| | $P_{out} = w(a_1 e_1 j_1 \ell_1) x(b_1 e_1 j_1 \ell_1) y'(c_0 f_0 j_1 \ell_1)$ |
| | $+ w(a_1 e_1 j_1 \ell_1) x(b_1 e_1 j_1 \ell_1) z'(d_0 f_0 j_1 \ell_1)$ |
| | $+ y(c_1 h_1 k_1 \ell_1) z(d_1 h_1 k_1 \ell_1) w'(a_0 g_0 k_1 \ell_1)$ |
| | $+ y(c_1 h_1 k_1 \ell_1) z(d_1 h_1 k_1 \ell_1) x'(b_0 g_0 k_1 \ell_1)$ |

The analysis presented in Table 2 is retabulated in Table 3 using the compressed notation. Similarities between the output propositions of Tables 1 and 3 are quite apparent. Input propositions in Table 1 have been replaced by literal propositions in Table 3.

It is also possible to derive the complementary output function, $P'_{out} = P'_\ell$, in terms of the compressed notation. The derivation is

TABLE 4: DERIVATION OF $P''_{out}$ IN COMPRESSED NOTATION

| Wire | Proposition |
|---|---|
| a | $P''_a = w'(a_0)$ |
| b | $P''_b = x'(b_0)$ |
| c | $P''_c = y'(c_0)$ |
| d | $P''_d = z'(d_0)$ |
| e | $P''_e = (P''_a + P''_b)e_n + e_0 = w'(a_0e_0) + x'(b_0e_0)$ |
| f | $P''_f = (P''_c + P''_d)f_n + f_0 = y'(c_0f_0) + z'(d_0f_0)$ |
| g | $P''_g = (P''_a + P''_b)g_n + g_0 = w'(a_0g_0) + x'(b_0g_0)$ |
| h | $P''_h = (P''_c + P''_d)h_n + h_0 = y'(c_0h_0) + z'd_0h_0)$ |
| j | $P''_j = (P''_e + P''_f)j_n + j_0 = w'(a_0e_0j_0) + x'(b_0e_0j_0) + y(c_1f_1j_0)z(d_1f_1j_0)$ |
| k | $P''_k = (P_g + P''_h)k_n + k_0 = y'(c_0h_0k_0) + z'(d_0h_0k_0) + w(a_1g_1k_0)x(b_1g_1k_0)$ |
| ℓ | $P''_{out} = P''_\ell = P''_jP''_k\ell_n + \ell_0$ |
| | $= \left[w'(a_0e_0j_0\ell_0) + x'(b_0e_0j_0\ell_0) + y(c_1f_1j_0\ell_0)z(d_1f_1j_0\ell_0)\right]$ |
| | $\cdot \left[y'(c_0h_0k_0\ell_0) + z'(d_0h_0k_0\ell_0) + w(a_1g_1k_0\ell_0)x(b_1g_1k_0\ell_0)\right]$ |
| | $= w'(a_0e_0j_0\ell_0)y'(c_0h_0k_0\ell_0)$ |
| | $+ w'(a_0e_0j_0\ell_0)z'(d_0h_0k_0\ell_0)$ |
| | $+ w'(a_0e_0j_0\ell_0)w(a_1g_1k_0\ell_0)x(b_1g_1k_0\ell_0)$ |
| | $+ x'(b_0e_0j_0\ell_0)y'(c_0h_0k_0\ell_0)$ |
| | $+ x'(b_0e_0j_0\ell_0)z'(d_0h_0k_0\ell_0)$ |
| | $+ w(a_1g_1k_0\ell_0)x'(b_0e_0j_0\ell_0)x(b_1g_1k_0\ell_0)$ |
| | $+ y'(c_0h_0k_0\ell_0)y(c_1f_1j_0\ell_0)z(d_1f_1j_0\ell_0)$ |
| | $+ y(c_1f_1j_0\ell_0)z'(d_0h_0k_0\ell_0)z(d_1f_1j_0\ell_0)$ |
| | $+ w(a_1g_1j_0\ell_0)x(b_1g_1j_0\ell_0)y(c_1f_1j_0\ell_0)z(d_1f_1j_0\ell_0)$ |

Sum of Products Form

shown in Table 4. The compact form for the complementary output proposition of Table 4 is similar to that of Table 1. However, when expanded to the sum of products forms, the output proposition of Table 4 has several more terms than that of Table 1. One such term is

$$w'(a_0e_0j_0\ell_0)w(a_1g_1k_0\ell_0)\times(b_1g_1k_0\ell_0).$$

These extra terms are characterized by the presence of two literal propositions whose input propositions are complements of one another. Although the input propositions are complements, the literal propositions are not.

$$w'(a_0e_0j_0\ell_0) = \Big(\Big\{\big[(w'a_n+a_0)e_n+e_0\big]j_n+j_0\Big\}\ell_n+\ell_0\Big)$$
$$= (w'a_ne_nj_n\ell_n+a_0e_nj_n\ell_n+e_0j_n\ell_n+j_0\ell_n+\ell_0)$$

$$w(a_1g_1h_0\ell_0) = \Big(\Big\{\big[(w a_n+a_1)g_n+g_1\big]k_n+k_0\Big\}\ell_n+\ell_0\Big)$$
$$= (w a_ng_nk_n\ell_n+a_1g_nk_n\ell_n+g_1k_n\ell_n+k_0\ell_n+\ell_0)$$

Clearly, since both literal propositions have value T if wire $\ell$ is permanently 0, they are not complements and theorem $X\cdot X' = F$ does not apply. In Table 1, the situation is different. The process of multiplying out generates a term $w'wx$ which may be eliminated by the theorem $X\cdot X' = F$.

## Analysis of Contact Networks.

An output proposition written in terms of input propositions and element propositions may be obtained for contact networks as well as gate networks. The contact network of Fig. 4 will be used to illustrate the analysis procedure. Each contact of a network is an element of $\{E\}$ and is therefore assigned a different letter regardless of the relay upon which it is mounted. An input proposition is associated with each contact. When this input proposition is true, the contact will normally be closed.

Consider contact a of Fig. 4. Contact a is normally closed when input proposition w' is true. The proposition $(w'a_n+a_s)$ is true whenever transmission through contact a is possible. Proposition $a_s$ is true if a is shorted and $a_n$ is true if a is normal. Transmission is blocked by contact a if the proposition $(w'a_n+a_s)' = (wa_n+a_o)$ is true. Proposition $a_o$ is true if contact a is permanently open.

Transmission through the network (output of 1) results whenever all contacts in a tie path[3, 4, 5] are closed. Transmission is

TABLE 5: CUT AND TIE PATHS FOR CONTACT NETWORK OF FIG. 4

| Tie Paths | Cut Paths |
|---|---|
| ab | ae |
| acd | acd |
| bce | bce |
| de | bd |

blocked (output of 0) if all contacts of a cut path are open. A tie path is a path between the two external terminals of the network. A cut path is a path passing through contacts which separate the external terminals. Tie paths and cut paths for the network of Fig. 4 are listed in Table 5.

TABLE 6: $P_{out}$ and $P'_{out}$ FOR CONTACT NETWORK OF FIG. 4

$$P_{out} = (w'a_n + a_s)(x'b_n + b_s) = w'(a_s)x'(b_s)$$
$$+ (w'a_n + a_s)(y'c_n + c_s)(z'd_n + d_s) \quad + w'(a_s)y'(c_s)z'(d_s)$$
$$+ (x'b_n + b_s)(y'c_n + c_s)(z\,e_n + e_s) \quad + x'(b_s)y'(c_s)z\,(e_s)$$
$$+ (z'd_n + d_s)(z\,e_n + e_s) \quad + z'(d_s)z\,(e_s)$$

$$P'_{out} = (w\,a_n + a_o)(z'e_n + e_o) = w(a_o)z'(e_o)$$
$$+ (w\,a_n + a_o)(y\,c_n + c_o)(z\,d_n + d_o) \quad + w(a_o)y\,(c_o)z\,(d_o)$$
$$+ (x\,b_n + b_o)(y\,c_n + c_o)(z'e_n + e_o) \quad + x(b_o)y\,(c_o)z'(e_o)$$
$$+ (x\,b_n + b_o)(z\,d_n + d_o) \quad + x(b_o)z\,(d_o)$$

Substituting propositions of the form $(w'a_n + a_s)$ for the contacts in a tie path gives a set of conditions under which transmission through the network is possible. The disjunction of propositions obtained from all tie paths is the output proposition, $P_{out}$, in sum of products form. Substituting propositions of the form $(wa_n + a_o)$ for contacts of a cut path yields a set of conditions for which transmission may be blocked. The complementary output function, $P'_{out}$, is the disjunction of propositions corresponding to all cut paths. Table 6 lists $P_{out}$ and $P'_{out}$ for the network of Fig. 4. Observe that $P_{out}$

and $P'_{out}$ are composed of literal propositions such as $(wa_n + a_o)$. A compressed notation is possible for $P_{out}$ and $P'_{out}$, although it does not represent a significant saving. The important observation is that the output propositions for both contact networks and gate networks may be written in terms of literal propositions.

### $L_1$-Sets and $L_0$-Sets.

The preceding section has considered a method for obtaining output propositions and their complements for gate and contact networks. For the purposes of deriving fault detection and diagnostic tests, it is most convenient to use the output proposition and its complement in their sum of products form. The concepts of $L_1$-sets are introduced to facilitate use of the sum of products form.

DEFINITION: A set of literal propositions is an $L_1$-set if and only if: (1) a network output signal of value 1 is guaranteed to occur when all propositions of the set are true; and (2) removal of any literal proposition from the set destroys property (1).

An analogous definition is used for $L_0$-sets.

DEFINITION: A set of literal propositions is an $L_0$-set if and only if: (1) a network output signal of value 0 is guaranteed to occur when all propositions of the set are true; and (2) removal of any literal proposition from the set destroys property (1).

Literal propositions in the terms of the sum of products form for $P_{out}$ and $P'_{out}$ constitute the $L_1$-sets and $L_0$-sets respectively of a network. The $L_1$-sets and $L_0$-sets for the gate network of Fig. 3 may be obtained from the sum of products forms in Tables 3 and 4. The $L_1$-sets and $L_0$-sets for the contact network of Fig. 4 may be obtained from Table 6.

Two useful relations concerning literal propositions in $L_1$-sets and $L_0$-sets may be proved.

THEOREM 4: If a literal proposition, W, appears in an $L_1$-set ($L_0$-set), the literal proposition W' must appear in one or more $L_0$-sets ($L_1$-sets).

PROOF: $P_{out}$ may be written as the sum of products of literal propositions. The sum of products form for $P'_{out}$ may be derived by complementing $P_{out}$ to obtain a product of sums and multiplying out. By de Morgan's theorem, complementing a sum of terms results in complementing each term. Further, complementing a term results in complementing each factor (literal proposition) of the term. Multiplying out does not affect the fact that all literal propositions of $P_{out}$ were complemented to obtain $P'_{out}$. Since $L_1$-sets are obtained from $P_{out}$ and $L_0$-sets from $P'_{out}$ we conclude that if W ap-

pears in an $L_1$-set, then W' must appear in one or more $L_0$-sets. An analogous argument shows that if W appears in an $L_0$-set, then W' must appear in one or more $L_1$-sets.

THEOREM 5: If a literal proposition, W, appears in an $L_1$-set ($L_0$-set), literal proposition W' cannot appear in any $L_1$-set ($L_0$-set).

PROOF: Contact networks and gate networks must be regarded as separate cases. For gate networks, the rightmost element proposition in the compressed notation for every literal proposition in any $L_1$-set ($L_0$-set) is $e_{out\,1}$ ($e_{out\,0}$). From Theorems 2 and 3, complementation of a literal propositon changes $e_{out\,1}$ to $e_{out\,0}$ ($e_{out\,0}$ to $e_{out\,1}$). Hence, for gate networks, a literal proposition and its complement cannot both appear in $L_1$-sets ($L_0$-sets).

For contact networks, element propositions appearing in the compressed notation for literal propositions of $L_1$-sets ($L_0$-sets) have the form $e_{js}$ ($e_{jo}$). Complementation of a literal proposition changes $e_{js}$ to $e_{jo}$ ($e_{jo}$ to $e_{js}$). Hence, a literal proposition and its complement cannot both appear in $L_1$-sets ($L_0$-sets).

## Output Functions in the Presence of Faults.

To conclude this section, one of the more important properties of output proposition $P_{out}$ will be considered. Define a fault to be the existence of one or more elements of $\{E\}$ in a state other than normal. Truth values for all element propositions in the presence of a particular fault are easily obtained. Insertion into $P_{out}$ yields a proposition in terms of input propositions which may be regarded as a Boolean output function for the network in the presence of the fault.

A notation for describing particular faults has been found useful. Only those element propositions that are not normal are listed. Thus, $(a_1)$ means a fault where wire a of a gate network has a permanent signal of 1 and all other elements are normal; $(b_o c_s)$ denotes a multiple fault in a contact network where contact b is permanently open and contact c is permanently shorted.

Consider fault $(a_1)$ in the gate network of Fig. 3. Element propositions $a_1, b_n, c_n, \ldots, \ell_n$ are assigned truth value T while all others are assigned the value F. Insertion of these values into the expression for $P_{out}$ in Table 2 yields

$$P_{out}\Big|_{(a_1)} = x(y' + z') + yzx'$$

$$= xy' + xz' + x'yz$$

The same result may be obtained working with the compressed notation of Table 3:

$$w(a_1e_1j_1\ell_1)\Big|_{(a_1)} = T$$

$$w'(a_0g_0k_1\ell_1)\Big|_{(a_1)} = F$$

All other literal propositions reduce to their input proposition. Another example is provided by multiple fault $(a_0e_1)$:

$$w(a_1e_1j_1\ell_1)\Big|_{(a_0e_1)} = T$$

$$x(b_1e_1j_1\ell_1)\Big|_{(a_0e_1)} = T$$

$$w'(a_0g_0k_1\ell_1)\Big|_{(a_0e_1)} = T$$

All other literal propositions reduce to their input propositions. Hence

TABLE 7: COMPUTATION OF NETWORK FUNCTIONS IN THE PRESENCE OF FAULTS

| $L_1$-set \ Fault | Normal | $(a_1)$ | $(f_0)$ | $(a_0e_1)$ | $(a_0g_1)$ |
|---|---|---|---|---|---|
| $w(a_1e_1j_1\ell_1)x(b_1e_1j_1\ell_1)y'(c_0f_0j_1\ell_1)$ | $wxy'$ | $xy'$ | $wx$ | $y'$ | - |
| $w(a_1e_1j_1\ell_1)x(b_1e_1j_1\ell_1)z'(d_0f_0j_1\ell_1)$ | $wxz'$ | $xz'$ | $wx$ | $z'$ | - |
| $w'(a_0g_0k_1\ell_1)y(c_1h_1k_1\ell_1)z\ (d_1h_1k_1\ell_1)$ | $w'yz$ | | $w'yz$ | $yz$ | - |
| $x'(b_0g_0k_1\ell_1)y(c_1h_1k_1\ell_1)z\ (d_1h_1k_1\ell_1)$ | $x'yz$ | $x'yz$ | $x'yz$ | $x'yz$ | - |

$P_{out}|_{Normal} = wxy' + wxz' + w'yz + x'yz$

$P_{out}|_{(a_1)} = xy' + xz' + x'yz$

$P_{out}|_{(f_0)} = wx + w'yz + x'yz$

$P_{out}|_{(a_0e_1)} = y' + z' + yz = T$

$P_{out}|_{(a_0g_1)} = F$

from Table 3,

$$P_{out}\Big|_{(a_0 c_1)} = y' + z' + yz + x'yz.$$

Table 7 illustrates the computation of output functions for several faults which can exist in the gate network of Fig. 3.

## III. DERIVATION OF FAULT DETECTION TESTS

### Definitions and General Remarks.

A fault in a combinational logic network is defined as the existence of one or more elements which are not normal. A fault detection test for a combinational logic network consists or $r$ input combinations which are applied to the network. If the network has a single output line, one input combination can result in one of two output signals, 0 or 1. Application of $r$ input combinations results in one of $2^r$ possible output patterns. If all elements of the network are normal, an output pattern denoted $T_n$ is observed. A fault is detected if a pattern other than $T_n$ is observed. Detection of a fault requires that at least one of the $r$ input combinations yields an incorrect output signal.

Let $\{\mathcal{F}\}$ denote the set of all faults, both multiple and single, which can occur in a network. Usually it is regarded as impractical to devise a fault detection test which will detect any fault of $\{\mathcal{F}\}$. A practical test is usually one which guarantees detection of all faults in a subset denoted as $\{\mathcal{F}_p\}$. The choice of faults to be included in subset $\{\mathcal{F}_p\}$ influences, to an extent, the input combinations which will appear in the fault detection test. Often, $\{\mathcal{F}_p\}$ is chosen to be the set of single faults only, in which case it will be denoted as $\{\mathcal{F}_s\}$.

Another constraint imposed during the derivation of fault detection tests is a limitation on the input combinations which may be applied to a network. There are $2^n$ input combinations which could be applied to a network with $n$ input lines. However, in many cases it will be impossible to apply all $2^n$ input combinations because of limitations imposed by circuits external to the network. Let $\{X\}$ denote the set of all input combinations which may be applied to a network in order to test it. In general, $\{X\}$ will contain a number of input combinations not exceeding $2^n$.

Let the set of input combinations $\{X\}$ be partitioned into two disjoint subsets, $\{X(1)\}$ and $\{X(0)\}$. Input combinations belonging to $\{X(1)\}$ are those which normally result in a network output of 1. $\{X(0)\}$ contains those normally yielding an output of 0. Summarizing the notation:

$\{\mathcal{J}\}$ = the set of all possible faults;

$\{\mathcal{J}_p\}$ = a subset of $\{\mathcal{J}\}$ selected for practical reasons;

$\{\mathcal{J}_s\}$ = set of all single faults;

$\{X\}$ = set of all input combinations which may be applied to a network;

$\{X(1)\}$ = input combinations of $\{X\}$ which normally yield an output of 1;

$\{X(0)\}$ = input combinations of $\{X\}$ which normally yield an output of 0.

A fault detection test for a network is obtained as a solution of a fault table, which has one column for every fault in subset $\{\mathcal{J}_p\}$ and a row for every input combination of $\{X\}$. A cross is placed at the intersection of row $i$ and column $j$ if the fault of column $j$ is detected by the input combination of row $i$. That is, the network output, in response to the input combination of row $i$, is different for the normal network and the network with the fault of column $j$. Table 8 illustrates a fault table for the contact network of Fig. 4. Only single faults are considered in Table 8.

Fault tables as defined are equivalent to prime implicant tables considered in references 6, 7, 8. As stated in reference 8, two problems may be considered.

**First Problem:** For a given prime implicant table find a smallest set of rows such that every column of the table has a marked entry in at least one row of the set.

**Second Problem:** For a given table, find the set S of all sets of rows, such that for any element $S_i$ of S, there is at least one marked entry in every column in a row belonging to $S_i$, and also such that the deletion of any row from $S_i$ will destroy this property.

Finding an optimum or shortest fault detection test from a fault table is equivalent to a solution of the first problem. Finding the collection of all non-redundant fault detection tests is equivalent to a solution of the second problem. Thus, if a fault table can be derived for a given $\{\mathcal{J}_p\}$ and $\{X\}$, an optimum test or the collection of all non-redundant fault detection tests for a network can be found.

TABLE 8: FAULT TABLE FOR CONTACT NETWORK OF FIG. 4

| $\{X\}$ \ $\{f_p\}$ | $a_s$ | $b_s$ | $c_s$ | $d_s$ | $e_s$ | $a_o$ | $b_o$ | $c_o$ | $d_o$ | $e_o$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $X^0$ | | | | | | × | | | | |
| $X^1$ | | | | | | | × | | | |
| $X^2$ | | | | | | × | × | | | |
| $X^3$ | | | | | | × | × | | | |
| $X^4$ | | | | | | × | | × | × | |
| $X^9$ | | | | | | | × | × | | × |
| $X^5$ | | × | | × | | | | | | |
| $X^6$ | | × | × | | × | | | | | |
| $X^7$ | | × | | × | | | | | | |
| $X^8$ | × | | | | × | | | | | |
| $X^{10}$ | × | | | | × | | | | | |
| $X^{11}$ | × | | × | × | | | | | | |
| $X^{12}$ | × | | | | × | | | | | |
| $X^{13}$ | | × | | × | | | | | | |
| $X^{14}$ | | | | | × | | | | | |
| $X^{15}$ | | | | × | | | | | | |

## Construction of Fault Tables.

Two distinct approaches to the construction of fault tables are possible:

1. Consider each fault of subset $\{f_p\}$ individually. The fault table is constructed one column at a time.

2. Consider each input combination of $\{X\}$ individually. The fault table is then constructed one row at a time.

Construction of fault tables using the first approach has received the most attention in the literature.[1, 2] Constructing the table one col-

umn at a time will not be considered in this report except for the following few remarks. Finding a column of the fault table consists of finding the network output function in the presence of a given fault from $\{\mathcal{F}_p\}$. Comparison of the modified output function with the normal function yields the input combinations of $\{X\}$ which detect the given fault. Hence, one column of the fault table is constructed. As indicated in Table 7, it is convenient to use the output proposition of a network, $P_{out}$, to obtain the modified functions for the network in the presence of faults. Computation of the fault table using this approach presents no conceptual difficulties. However, for large $\{\mathcal{F}_p\}$ the practical aspects of computation become overwhelming.

Certainly, if the number of faults in $\{\mathcal{F}_p\}$ is less than the number of input combinations in $\{X\}$ the approach of considering faults of $\{\mathcal{F}_p\}$ individually is appropriate. However, if $\{\mathcal{F}_p\}$ contains a large number of multiple faults, the number of input combinations in $\{X\}$ may be small by comparison. The possibilities of considering each input combination individually becomes attractive.

For instance, consider a single output gate network with n input wires. A lower bound on the number of wires in the network is n+1, which arises if the network contains only one gate. Using n as an approximation to the number of wires, the number of faults in various fault sets $\{\mathcal{F}_p\}$ are:

| | |
|---|---|
| Single faults only | $2n$ |
| Single and double faults | $2n + \frac{2n(2n-2)}{2}$ |
| Up to triple faults | $2n + \frac{2n(2n-2)}{2} + \frac{2n(2n-2)(2n-4)}{6}$ |
| . . | . . |
| . . | . . |
| . . | . . |
| All possible faults | $3^n - 1$. |

The number of input combinations in $\{X\}$ must be less than or equal to $2^n$, so there is always a point at which considering each input combination individually becomes attractive. Derivation of tests to detect all possible faults must consider this approach.

Construction of a fault table one row at a time requires a method whereby all faults of $\{\mathcal{F}_p\}$ may be found which are detected by an input combination $\underline{x}^k$. Denote by $\{\mathcal{F}_p(\underline{x}^k)\}$ that subset of $\{\mathcal{F}_p\}$ which is detected by $\underline{x}^k$. Similarly, denote by $\{\mathcal{F}(\underline{x}^k)\}$ that subset of all faults $\{\mathcal{F}\}$ which is detected by $\underline{x}^k$.

Consider an input combination, $\underline{x}^q$, which normally yields an output of 1 ($\underline{x}^q \in \{X(1)\}$). Any fault, single or multiple, causing

$\underline{x}^9$ to yield an output of 0 is detected by $\underline{x}^9$. An output of 0 can occur only if all literal propositions of one or more $L_0$-sets are true. Hence, $\{\mathcal{F}(\underline{x}^9)\}$ consists of all faults which, together with $\underline{x}^9$, will make true all literal propositions of some $L_0$-set. As an example, consider $L_0$-set $w'(a_0c_0j_0\ell_0)y'(c_0h_0k_0\ell_0)$ from Table 4. From the Karnaugh map of Fig. 3, it is seen that input combination $\underline{x}^7 = w'xyz$ normally yields an output of 1. Under normal conditions, application of $\underline{x}^7$ to the network will make literal proposition $w'(a_0c_0j_0\ell_0)$ true and $y'(c_0h_0k_0\ell_0)$ false. If $\underline{x}^7$ is to make both literal propositions true, a fault must exist that will make $y'(c_0h_0k_0\ell_0)$ true and will not make $w'(a_0c_0j_0\ell_0)$ false. A multiple fault $(a_1c_0)$ will make $y'(c_0h_0k_0\ell_0)$ true but also makes $w'(a_0c_0j_0\ell_0)$ false. Hence, $\underline{x}^7$, together with fault $(a_1c_0)$, will not make both propositions of $L_0$-set $w'(a_0c_0j_0\ell_0)y'(c_0h_0k_0\ell_0)$ true. The nested form for $y'(c_0h_0k_0\ell_0)$ is

$$\left(\left\{\left[(y'c_n+c_0)h_n+h_0\right]k_n+k_0\right\}\ell_n+\ell_0\right).$$

which may be expanded to

$$(y'c_nh_nk_n\ell_n+c_0h_nh_nk_n\ell_n+h_0k_n\ell_n+k_0\ell_n+\ell_0).$$

Application of $\underline{x}^7$ amounts to inserting a truth value of F for $y'$, resulting in

$$(c_nh_nk_n\ell_n+h_0k_n\ell_n+k_0\ell_n+\ell_0). \qquad (5)$$

The terms of (5) may be regarded as conditions or faults which will make proposition $y'(c_0h_0k_0\ell_0)$ permanently true. Making a similar expansion of $w'(a_0c_0j_0\ell_0)$ yields

$$(w'a_nc_nj_n\ell_n+a_0c_nj_n\ell_n+c_0j_n\ell_n+j_0\ell_n+\ell_0).$$

Conditions which guarantee that $w'(a_0c_0j_0\ell_0)$ will not be false when $\underline{x}^7$ is applied are seen to be

$$(a_nc_nj_n\ell_n+a_0c_nj_n\ell_n+c_0j_n\ell_n+j_0\ell_n+\ell_0). \qquad (6)$$

The intersection of (5) and (6) contains those faults which, together with input combination $\underline{x}^7$, will make both literal propositions of $L_0$-set $w'(a_0c_0j_0\ell_0)y'(c_0h_0k_0\ell_0)$ true.

In forming $\{\mathcal{F}(\underline{x}^7)\}$, it is not really necessary to write out all conditions contained in the intersection of (5) and (6). A slight variation in the compressed notation is sufficient. Suppose the intersection of (5) and (6) is written as

$$(a_0e_0j_0\ell_0)_{\underline{F}}(c_0h_0k_0\ell_0)_T.$$

The subscript T means that the corresponding literal proposition will be true only if a fault makes it permanently true. $(c_0h_0k_0\ell_0)_T$ denotes the proposition (5), which must be true to guarantee that literal proposition $y'(c_0h_0k_0\ell_0)$ will have value T when $\underline{x}^7$ is applied. The subscript $\underline{F}$ means that the corresponding literal proposition must not be permanently false. It may be permanently true or all of its elements may be in the normal state, but it must not be false. $(a_0e_0j_0\ell_0)_{\underline{F}}$ denotes proposition (6), which must have truth value T in order to guarantee that $w'(a_0e_0j_0\ell_0)$ will have value T when $\underline{x}^7$ is applied.

As another example, $\underline{x}^7 = w'xyz$ will make true all literal propositions of $L_0$-set

$$w'(a_0e_0j_0\ell_0)w(a_1g_1k_0\ell_0)x(b_1g_1k_0\ell_0)$$

if conditions exist denoted by

$$(a_0e_0j_0\ell_0)_{\underline{F}}(a_1g_1k_0\ell_0)_T(b_1g_1k_0\ell_0)_{\underline{F}}.$$

If an input combination $\underline{x}^r$ normally yields an output of $0(\underline{x}^r \in \{X(0)\})$, then $\underline{x}^r$ will detect any fault which causes it to yield an output of 1. Since an output of 1 occurs only if an $L_1$-set is true, $\{\mathcal{J}(\underline{x}^r)\}$ will contain all faults which together with $\underline{x}^r$ make all literal propositions of some $L_1$-set true. For example, applying input combination $\underline{x}^6 = w'xyz'$, all literal propositions of $L_1$-set

$$w(a_1e_1j_1\ell_1)x(b_1e_1j_1\ell_1)y'(c_0f_0j_1\ell_1)$$

will be true for the conditions denoted by

$$(a_1e_1j_1\ell_1)_T(b_1e_1j_1\ell_1)_{\underline{F}}(c_0f_0j_1\ell_1)_T.$$

Generalizing, the faults which, together with an input combination $\underline{x}^q \in \{X(1)\}$ $(\underline{x}^r \in \{X(0)\})$, will make all literal propositions of an $L_0$-set, $L_{0_j}$, ($L_1$-set, $L_{1_i}$) true are found by the following rules:

1. Each literal proposition of $L_{0_j}$ ($L_{1_i}$) that is normally made true by $\underline{x}^q$ ($\underline{x}^r$) is marked as F;

2. Each literal proposition of $L_{0_j}$ ($L_{1_i}$) that is normally made false by $\underline{x}^q$ ($\underline{x}^r$) is marked as T;

3. Delete the input propositions.

The formation of $\{\mathcal{T}(\underline{x}^q)\}$ or $\{\mathcal{T}(\underline{x}^r)\}$ follows directly. To find $\{\mathcal{T}(\underline{x}^q)\}$ ($\{\mathcal{T}(\underline{x}^r)\}$), all $L_0$-sets ($L_1$-sets) of a given network must be considered. Having applied the preceding rules to each $L_0$-set ($L_1$-set) of the network, $\{\mathcal{T}(\underline{x}^q)\}$ ($\{\mathcal{T}(\underline{x}^r)\}$) is simply the union of the results. Table 9 illustrates the computation of fault set $\{\mathcal{T}(\underline{x}^k)\}$ for the gate network of Fig. 3. It is assumed for this example that $\{X\}$ is limited to the nine input combinations $\underline{x}^5$, $\underline{x}^6$, $\underline{x}^7$, $\underline{x}^9$, $\underline{x}^{10}$, $\underline{x}^{11}$, $\underline{x}^{13}$, $\underline{x}^{14}$ and $\underline{x}^{15}$. Entries of Table 9 are shown in an abbreviated form where only the subscripts T or F are shown. Since the $L_1$-sets

TABLE 9: $\{\mathcal{T}(\underline{x}^k)\}$ FOR THE GATE NETWORK OF FIG. 3

| $L_0$-sets \ $\{X(1)\}$ | $\underline{x}^7$ • w'xyz | $\underline{x}^{11}$ • wx'yz | $\underline{x}^{13}$ • wxy'z | $\underline{x}^{14}$ • wxyz' |
|---|---|---|---|---|
| $w'(a_0e_0j_0\ell_0)y'(c_0h_0k_0\ell_0)$ | FF | TT | TF | TT |
| $w'(a_0e_0j_0\ell_0)x'(d_0h_0k_0\ell_0)$ | FT | TT | TT | TF |
| $w'(a_0e_0j_0\ell_0)w\,a_1g_1k_0\ell_0)x\,(b_1g_1k_0\ell_0)$ | FTF | TFT | TFF | TFF |
| $x'(b_0e_0j_0\ell_0)y'(c_0h_0k_0\ell_0)$ | TT | FT | TF | TT |
| $x'(b_0e_0j_0\ell_0)z'(d_0h_0k_0\ell_0)$ | TT | FT | TT | TF |
| $w(a_1g_1k_0\ell_0)x'(b_0e_0j_0\ell_0)x\,(b_1g_1k_0\ell_0)$ | TTF | FFT | FTF | FTF |
| $y'(c_0h_0k_0\ell_0)y\,(c_1f_1j_0\ell_0)z\,(d_1f_1j_0\ell_0)$ | TFF | TFF | FTF | TFT |
| $y\,(c_1f_1j_0\ell_0)z'(d_0h_0k_0\ell_0)z\,(d_1f_1j_0\ell_0)$ | FTF | FTF | TTF | FFT |
| $w(a_1g_1k_0\ell_0)x\,(b_1g_1k_0\ell_0)y\,(c_1f_1j_0\ell_0)z\,(d_1f_1j_0\ell_0)$ | TFFF | FTFF | FFTF | FFFT |

| $L_1$-sets \ $\{X(0)\}$ | $\underline{x}^5$ • w'xy'z | $\underline{x}^6$ • w'xyz' | $\underline{x}^9$ • wx'y'z | $\underline{x}^{10}$ • wx'yz' | $\underline{x}^{15}$ • wxyz |
|---|---|---|---|---|---|
| $w(a_1e_1j_1\ell_1)x(b_1e_1j_1\ell_1)y'(c_0f_0j_1\ell_1)$ | TFF | TFT | FTF | FTT | FFT |
| $w(a_1e_1j_1\ell_1)x(b_1e_1j_1\ell_1)z'(d_0f_0j_1\ell_1)$ | TFT | TFF | FTT | FTF | FFT |
| $w'(a_0g_0k_1\ell_1)y(c_1h_1k_1\ell_1)z\,(d_1h_1k_1\ell_1)$ | FTF | FFT | TTF | TFT | TFF |
| $x'(b_0g_0k_1\ell_1)y(c_1h_1k_1\ell_1)z\,(d_1h_1k_1\ell_1)$ | TTF | TFT | FTF | FFT | TFF |

and $I_0$-sets are written out as row designations, the subscripts carry all the necessary information. $\{\mathcal{J}(\underline{x}^k)\}$ is found from the union of all entries in the column corresponding to $\underline{x}^k$. If written out in its entirety, the union lists all faults, single and multiple, which $\underline{x}^k$ can detect.

**Construction of Single Fault Tables from the $\{\mathcal{J}(\underline{x}^k)\}$.**

Unfortunately, a fault set $\{\mathcal{J}(\underline{x}^k)\}$ contains a large number of multiple faults, many of which may not be included in a subset $\{\mathcal{J}_p\}$ selected for a given network. It can then be a formidable data processing task to obtain the $\{\mathcal{J}_p(\underline{x}^k)\}$ from $\{\mathcal{J}(\underline{x}^k)\}$ by checking to see if faults in $\{\mathcal{J}_p\}$ are also contained in $\{\mathcal{J}(\underline{x}^k)\}$. For this reason, construction of fault tables by deriving the $\{\mathcal{J}_p(\underline{x}^k)\}$ will be limited to single fault tables. A comparatively simple method for constructing tests to detect all faults in $\{\mathcal{J}\}$ utilizing the $\{\mathcal{J}(\underline{x}^k)\}$ will be considered in the next section. Application to subsets $\{\mathcal{J}_p\}$ follows directly.

A single fault was defined as one network element stuck in a state other than normal with all other elements in the normal state. The property that all elements but one are normal permits rapid computation of $\{\mathcal{J}_s(\underline{x}^k)\}$, the set of single faults which can be detected by input combination $\underline{x}^k$. Consider once again $I_0$-set

$$w'(a_0e_0j_0\ell_0)y'(c_0h_0k_0\ell_0)$$

and input combination $\underline{x}^7 = w'xyz$. The contribution to $\{\mathcal{J}(\underline{x}^7)\}$ was seen to be $(a_0e_0j_0\ell_0)_F(c_0h_0k_0\ell_0)_T$. From (5), single fault $(c_0)$ makes proposition $y'(c_0h_0k_0\ell_0)$ permanently true since wires h, k, and $\ell$ are guaranteed to be normal under the single fault hypothesis. Similarly, single faults $(h_0)$, $(k_0)$, and $(\ell_0)$ will also make $y'(c_0h_0k_0\ell_0)$ true. Element propositions listed in the compressed notation $(c_0h_0k_0\ell_0)_T$ correspond directly to the set of single faults which make $y'(c_0h_0k_0\ell_0)$ true.

Faults $(a_1)$, $(e_1)$, $(j_1)$, and $(\ell_1)$ are the single faults which make proposition $w'(a_0e_0j_0\ell_0)$ false. The intersection of sets $\{(a_1), (e_1), (j_1), (\ell_1)\}$ and $\{(c_0), (h_0), (k_0), (\ell_0)\}$ is empty. Therefore, we may conclude that faults of the set $\{(c_0), (h_0), (k_0), (\ell_0)\}$ are all detectable by $\underline{x}^7$, since proposition $w'(a_0e_0j_0\ell_0)$ will not be false and $y'(c_0h_0k_0\ell_0)$ will be true.

Entry $(a_1g_1k_0\ell_0)_T(b_0e_0j_0\ell_0)_T(b_1g_1k_0\ell_0)_F$ from the first column of Table 9 illustrates a second facet of the problem. Single fault $(\ell_0)$ is the only single fault which can simultaneously satisfy the conditions $(a_1g_1k_0\ell_0)_T$ and $(b_0e_0j_0\ell_0)_T$. Fault $(\ell_0)$ is the intersection of sets $\{(a_1), (g_1), (k_0), (\ell_0)\}$, and $\{(b_0), (e_0), (j_0), (\ell_0)\}$. The intersection of $(\ell_0)$ and set $\{(b_0), (g_0), (k_1), (\ell_1)\}$, the faults

which violate condition $(b_1g_1k_0\ell_0)_F$, is empty. Hence $(\ell_0)$ may be detected by $\underline{x}^7$ since it satisfies both $(a_1g_1k_0\ell_0)_T$ and $(b_0e_0j_0\ell_0)_T$ and does not violate $(b_1g_1k_0\ell_0)_F$.

As a final example, consider entry

$$(c_1f_1j_0\ell_0)_{\underline{F}}(d_0h_0k_0\ell_0)_T(d_1f_1j_0\ell_0)_{\underline{F}}$$

from the first column of Table 9. Faults satisfying $(d_0h_0k_0\ell_0)_T$ are, of course, the set $\{(d_0), (h_0), (k_0), (\ell_0)\}$. Faults which would violate the conditions $(c_1f_1j_0\ell_0)_{\underline{F}}$ or $(d_1f_1j_0\ell_0)_{\underline{F}}$ are contained in the union

$$\{(c_0), (f_0), (j_1), (\ell_1)\} \cup \{(d_0), (f_0), (j_1), (\ell_1)\} =$$

$$\{(c_0), (d_0), (f_0), (j_1), (\ell_1)\}.$$

The intersection

$$\{(d_0), (h_0), (k_0), (\ell_0)\} \cap \{(c_0), (d_0), (f_0), (j_1), (\ell_1)\} = (d_0).$$

Single fault $(d_0)$, while satisfying $(d_0h_0k_0\ell_0)_T$, violates condition $(d_1f_1j_0\ell_0)_F$. Therefore, $\underline{x}^7$ will detect only the set $\{(h_0), (k_0), (\ell_0)\}$ due to $L_0$-set $y(c_1f_1j_0\ell_0)z'(d_0h_0k_0\ell_0)z(d_1f_1j_0\ell_0)$.

Table 10 illustrates the derivation of the $\{\mathcal{F}_s(\underline{x}^k)\}$ from Table 9. Entries of Table 10 are derived directly from the corresponding entries of Table 9 according to following general rules:

1. For each entry in $\{\mathcal{F}(\underline{x}^k)\}$ find:
   a) The set of single faults which satisfy all conditions subscripted with T (denoted as $\{T\}$);
   b) The set of all single faults which violate one or more of the conditions subscripted with $\underline{F}$ (denoted $\{\underline{F}\}$);
   c) The intersection $\{T\} \cap \{\underline{F}\}$.
2. $\underline{x}^k$ can detect the set of faults specified by $\{T\}$ less than those in $\{T\} \cap \{\underline{F}\}$.
3. Repeat the above process for every entry in $\{\mathcal{F}(\underline{x}^k)\}$.

The union of the results is $\{\mathcal{F}_s(\underline{x}^k)\}$. The set $\{\mathcal{F}_s(\underline{x}^k)\}$ is listed at the bottom of the column corresponding to input combination $\underline{x}^k$ in Table 10.

The single fault table for the gate network of Fig. 3 is presented as Table 11. There is a column for each possible single fault and a row for each of the nine input combinations of $\{X\}$. A cross in

TABLE 10-a $\left\{ \gamma_{n}(x^{k}) \right\}$ FOR THE GATE NETWORK OF FIG. 3

| $L_{0}$-sets \ $\{x(0)\}$ | $x^{7}$ · $w'xyz$ | $x^{11}$ · $wx'yz$ | $x^{13}$ · $wxy'z$ | $x^{14}$ · $wxyz'$ |
|---|---|---|---|---|
| $w'(a_0c_0j_0\ell_0)y'(c_0h_0k_0\ell_0)$ | $c_0h_0$ $k_0\ell_0$ | $\ell_0$ | $a_0c_0$ $j_0\ell_0$ | $\ell_0$ |
| $w'(a_0c_0j_0\ell_0)z'(d_0h_0k_0\ell_0)$ | $d_0h_0$ $k_0\ell_0$ | $\ell_0$ | $\ell_0$ | $b_0c_0$ $j_0\ell_0$ |
| $w'(a_0c_0j_0\ell_0)w\,(a_1g_1k_0\ell_0)x\,(b_1g_1k_0\ell_0)$ | $g_1k_0$ $\ell_0$ | $\ell_0$ | $c_0j_0$ $\ell_0$ | $c_0j_0$ $\ell_0$ |
| $x'(b_0c_0j_0\ell_0)y'(c_0h_0k_0\ell_0)$ | $\ell_0$ | $c_0h_0$ $k_0\ell_0$ | $b_0c_0$ $j_0\ell_0$ | $\ell_0$ |
| $x'(b_0c_0j_0\ell_0)z'(d_0h_0k_0\ell_0)$ | $\ell_0$ | $d_0h_0$ $k_0\ell_0$ | $\ell_0$ | $b_0c_0$ $j_0\ell_0$ |
| $w(a_1g_1k_0\ell_0)x'(b_0c_0j_0\ell_0)x\,(b_1g_1k_0\ell_0)$ | $\ell_0$ | $g_1k_0$ $\ell_0$ | $c_0j_0$ $\ell_0$ | $c_0j_0$ $\ell_0$ |
| $y'(c_0h_0k_0\ell_0)y\,(c_1f_1j_0\ell_0)z\,(d_1f_1j_0\ell_0)$ | $h_0k_0$ $\ell_0$ | $h_0k_0$ $\ell_0$ | $f_1j_0$ $\ell_0$ | $\ell_0$ |
| $y\,(c_1f_1j_0\ell_0)z'(d_0h_0k_0\ell_0)z\,(d_1f_1j_0\ell_0)$ | $h_0k_0$ $\ell_0$ | $h_0k_0$ $\ell_0$ | $\ell_0$ | $f_1j_0$ $\ell_0$ |
| $w(a_1g_1k_0\ell_0)x(b_1g_1k_0\ell_0)$ $y\,(c_1f_1j_0\ell_0)z\,(d_1f_1j_0\ell_0)$ | $a_1g_1$ $k_0\ell_0$ | $b_1g_1$ $k_0\ell_0$ | $c_1f_1$ $j_0\ell_0$ | $d_1f_1$ $j_0\ell_0$ |
| $\left\{ \gamma_{0}(x^{k}) \right\}$ | $a_1g_1$ $c_0d_0$ $h_0k_0$ $\ell_0$ | $b_1g_1$ $c_0d_0$ $h_0k_0$ $\ell_0$ | $c_1f_1$ $a_0b_0$ $c_0j_0$ $\ell_0$ | $d_1f_1$ $a_0b_0$ $c_0j_0$ $\ell_0$ |

TABLE 10-b

| $L_{1}$-sets \ $\{x(0)\}$ | $x^{5}$ · $w'xy'z$ | $x^{6}$ · $w'xyz'$ | $x^{9}$ · $wx'y'z$ | $x^{10}$ · $wx'yz'$ | $x^{15}$ · $wxyz$ |
|---|---|---|---|---|---|
| $w(a_1c_1j_1\ell_1)x(b_1c_1j_1\ell_1)y'(c_0f_0j_1\ell_1)$ | $a_1c_1$ $j_1\ell_1$ | $j_1\ell_1$ | $b_1c_1$ $j_1\ell_1$ | $j_1\ell_1$ | $c_0f_0$ $j_1\ell_1$ |
| $w(a_1c_1j_1\ell_1)x(b_1c_1j_1\ell_1)z'(d_0f_0j_1\ell_1)$ | $j_1\ell_1$ | $a_1c_1$ $j_1\ell_1$ | $j_1\ell_1$ | $b_1c_1$ $j_1\ell_1$ | $d_0f_0$ $j_1\ell_1$ |
| $w'(b_0g_0k_1\ell_1)y(c_1h_1k_1\ell_1)z\,(d_1h_1k_1\ell_1)$ | $c_1h_1$ $k_1\ell_1$ | $d_1h_1$ $k_1\ell_1$ | $k_1\ell_1$ | $k_1\ell_1$ | $a_0g_0$ $k_1\ell_1$ |
| $x'(b_0g_0k_1\ell_1)y(c_1h_1k_1\ell_1)z\,(d_1h_1k_1\ell_1)$ | $k_1\ell_1$ | $k_1\ell_1$ | $c_1h_1$ $k_1\ell_1$ | $d_1h_1$ $k_1\ell_1$ | $b_0g_0$ $k_1\ell_1$ |
| $\left\{ \gamma_{1}(x^{k}) \right\}$ | $a_1c_1$ $c_1h_1$ $j_1k_1$ $\ell_1$ | $a_1d_1$ $c_1h_1$ $j_1k_1$ $\ell_1$ | $b_1c_1$ $c_1h_1$ $j_1k_1$ $\ell_1$ | $b_1d_1$ $c_1h_1$ $j_1k_1$ $\ell_1$ | $j_1k_1\ell_1$ $a_0b_0$ $c_0d_0$ $f_0g_0$ |

TABLE 11: SINGLE FAULT TABLE FOR NETWORK OF FIG. 3

| $\{\underline{x}\}$ \ $\{\mathcal{F}\}$ | $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ | $g_1$ | $h_1$ | $j_1$ | $k_1$ | $\ell_1$ | $a_0$ | $b_0$ | $c_0$ | $d_0$ | $e_0$ | $f_0$ | $g_0$ | $h_0$ | $j_0$ | $k_0$ | $\ell_0$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\underline{x}^7$ | x | | | | | | x | | | | | | | x | x | | | | x | | x | x |
| $\underline{x}^{11}$ | | x | | | | | x | | | | | | | x | x | | | | x | | x | x |
| $\underline{x}^{13}$ | | | x | | | x | | | | | | x | x | | | x | | | | x | | x |
| $\underline{x}^{14}$ | | | | x | | x | | | | | | x | x | | | x | | | | x | | x |
| $\underline{x}^5$ | x | | x | | x | | | x | x | x | x | | | | | | | | | | | |
| $\underline{x}^6$ | x | | | x | x | | | x | x | x | x | | | | | | | | | | | |
| $\underline{x}^9$ | | x | x | | x | | | x | x | x | x | | | | | | | | | | | |
| $\underline{x}^{10}$ | | x | | x | x | | | x | x | x | x | | | | | | | | | | | |
| $\underline{x}^{15}$ | | | | | | | | | x | x | x | x | x | x | x | | | x | x | | | |

row $i$ at its intersection with column $j$ indicates that the input combination of row $i$ will detect the single fault of column $j$. Construction of Table 11 from the $\{\mathcal{F}_s(\underline{x}^k)\}$ of Table 10 is straightforward.

## Solution of Fault Tables.

Solution of a fault table consists of finding the smallest set of input combinations which will detect all single faults. This amounts to finding the smallest set of input combinations having a cross in each column. Reference 8 contains a complete discussion of prime implicant tables and their solution. Techniques devised there are directly applicable to the solution of fault tables. Only the simplification of tables prior to their solution will be considered here.

## Simplification of Fault Tables by Column Dominance.

Column dominance simplifies a fault table by the removal of unnecessary columns. A column headed by fault $\mathcal{F}^m$ is dominated by a column $\mathcal{F}^n$ if column $\mathcal{F}^n$ has an entry in every row where column $\mathcal{F}^m$ has an entry. The dominating column $\mathcal{F}^n$ may be removed since any input combination which detects $\mathcal{F}^m$ is guaranteed to detect $\mathcal{F}^n$.

Column $f_1$ of Table 11 is dominated by $a_0$, $b_0$, $c_0$, $j_0$, and $\ell_0$.

all of which may be removed. Table 12 illustrates the fault table after all dominating columns have been removed. A single fault table simplified by column dominance is important not only because the single fault detection test may be derived from it, but also because it forms the nucleus of the multiple fault table to be developed in the next section.

## Simplification by Row Dominance.

Table 12 is now in a form from which all possible non-redundant single fault detection tests may be found. Further simplifications in the form of row dominance may often be used if only the optimum single fault test is desired.

TABLE 12: SINGLE FAULT TABLE SIMPLIFIED BY COLUMN DOMINANCE

| $\{x\}$ \ $\{j\}$ | $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ | $g_1$ | $f_0$ |
|---|---|---|---|---|---|---|---|---|
| $\underline{x}^7$ | × | | | | | | × | |
| $\underline{x}^{11}$ | | × | | | | | × | |
| $\underline{x}^{13}$ | | | × | | | × | | |
| $\underline{x}^{14}$ | | | | × | | × | | |
| $\underline{x}^5$ | × | | × | | × | | | |
| $\underline{x}^6$ | × | | | × | × | | | |
| $\underline{x}^9$ | | × | × | | × | | | |
| $\underline{x}^{10}$ | | × | | × | × | | | |
| $\underline{x}^{15}$ | | | | | | | | × |

TABLE 13: A SOLUTION OF THE SINGLE FAULT TABLE

| $\{x\}$ \ $\{j\}$ | $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ | $g_1$ | $f_0$ |
|---|---|---|---|---|---|---|---|---|
| $\underline{x}^7$ | × | | | | | | × | |
| $\underline{x}^{11}$ | (×) | | | | | | ⊗ | ✓ |
| $\underline{x}^{13}$ | | | × | | | × | | |
| $\underline{x}^{14}$ | | | | ⊗ | | ⊗ | | ✓ |
| $\underline{x}^5$ | × | | × | | × | | | ✓ |
| $\underline{x}^6$ | × | | | | × | × | | |
| $\underline{x}^9$ | | × | (×) | | ⊗ | | | |
| $\underline{x}^{10}$ | | × | | | × | × | | |
| $\underline{x}^{15}$ | | | | | | | (×) | ✓ |

A row, $\underline{x}^m$, is dominated by row $\underline{x}^n$ if row $\underline{x}^n$ has a cross in every column where $\underline{x}^m$ has a cross. The dominated row, $\underline{x}^m$, is removed since input combination $\underline{x}^n$ is capable of detecting more faults than $\underline{x}^m$ and is therefore more desirable.

Removal of rows by row dominance does not aid in selecting an optimum test from Table 12. For completeness, Table 13 is included to show one of several solutions of the fault table requiring four input combinations. All single faults in the network of Fig. 3 may be detected by a test consisting of input combinations $\underline{x}^{11}$, $\underline{x}^{14}$, $\underline{x}^5$, and $\underline{x}^{15}$.

### Fault Table for a Contact Network.

To summarize the procedure for deriving a single fault table using the sets $\{\mathcal{J}(\underline{x}^k)\}$, an example will be worked for the contact network of Fig. 4. $L_1$-sets and $L_0$-sets for this network may be obtained from Table 6. It is assumed that $\{X\}$ contains all sixteen possible input combinations. Table 14 illustrates computation of $\{\mathcal{J}_s(\underline{x}^k)\}$. A fault table constructed from the $\{\mathcal{J}_s(\underline{x}^k)\}$ was illustrated in Table 8. Column dominance is used to eliminate columns $a_0$, $b_0$, and $c_0$. Row dominance accounts for rows $\underline{x}^0, \underline{x}^1, \underline{x}^2, \underline{x}^3, \underline{x}^7, \underline{x}^{10}, \underline{x}^{12}, \underline{x}^{13}, \underline{x}^{14}$, and $\underline{x}^{15}$. The fault table following from these simplifications is given in Table 15. An optimum single test consisting of $\underline{x}^4, \underline{x}^6, \underline{x}^9$, and $\underline{x}^{11}$ is found.

TABLE 14: $\{\mathcal{J}_0(\underline{x}^k)\}$ FOR CONTACT NETWORK OF FIG. 4

| $L_0$-sets \ $\{X(1)\}$ | $\underline{x}^0$ • w'x'y'z' | $\underline{x}^1$ • w'x'y'z | $\underline{x}^2$ • w'x'yz' | $\underline{x}^3$ • w'x'yz | $\underline{x}^4$ • w'xy'z' | $\underline{x}^9$ • wx'y'z |
|---|---|---|---|---|---|---|
| $w(a_0)x'(c_0)$ | $\underline{T}\underline{F}$ $a_0$ | TT | $T\underline{F}$ $a_0$ | TT | $T\underline{F}$ $a_0$ | $\underline{F}T$ $c_0$ |
| $w(a_0)y(c_0)z(d_0)$ | TTT | TT$\underline{F}$ | T$\underline{F}$T | T$\underline{F}\underline{F}$ $a_0$ | TTT | $\underline{F}$T$\underline{F}$ $c_0$ |
| $x(b_0)y(c_0)x'(c_0)$ | TT$\underline{F}$ | TTT | T$\underline{F}\underline{F}$ $b_0$ | T$\underline{F}$T | $\underline{F}T\underline{F}$ $c_0$ | TTT |
| $x(b_0)z(d_0)$ | TT | T$\underline{F}$ $b_0$ | TT | T$\underline{F}$ $b_0$ | $\underline{F}T$ $d_0$ | T$\underline{F}$ $b_0$ |
| $\{\mathcal{J}_s(\underline{x}^k)\}$ | $a_0$ | $b_0$ | $a_0b_0$ | $a_0b_0$ | $a_0c_0$ $d_0$ | $b_0c_0$ $e_0$ |

| $L_1$-sets \ $\{X(0)\}$ | $\underline{x}^5$ • w'xy'z | $\underline{x}^6$ • w'xyz' | $\underline{x}^7$ • w'xyz | $\underline{x}^8$ • wx'y'z' | $\underline{x}^{10}$ • wx'yz' | $\underline{x}^{11}$ • wx'yz | $\underline{x}^{12}$ • wxy'z' | $\underline{x}^{13}$ • wxy'z | $\underline{x}^{14}$ • wxyz' | $\underline{x}^{15}$ • wxyz |
|---|---|---|---|---|---|---|---|---|---|---|
| $w'(a_s)x'(b_s)$ | $\underline{F}T$ $b_s$ | $\underline{F}T$ $b_s$ | $\underline{F}T$ $b_s$ | T$\underline{F}$ $a_s$ | T$\underline{F}$ $a_s$ | T$\underline{F}$ $a_s$ | TT | TT | TT | TT |
| $w'(a_s)y'(c_s)z'(d_s)$ | $\underline{F}\underline{F}$T $d_s$ | $\underline{F}$T$\underline{F}$ $c_s$ | $\underline{F}$TT | T$\underline{F}\underline{F}$ $a_s$ | TT$\underline{F}$ | TTT | T$\underline{F}\underline{F}$ $a_s$ | T$\underline{F}$T | TT$\underline{F}$ | TTT |
| $x'(b_s)y'(c_s)z(e_s)$ | T$\underline{F}\underline{F}$ $b_s$ | TTT | TT$\underline{F}$ | $\underline{F}$T$\underline{F}$ $e_s$ | $\underline{F}$TT | $\underline{F}$T$\underline{F}$ $e_s$ | T$\underline{F}$T | T$\underline{F}\underline{F}$ $b_s$ | TTT | TT$\underline{F}$ |
| $z'(d_s)z(e_s)$ | T$\underline{F}$ $d_s$ | $\underline{F}T$ $e_s$ | T$\underline{F}$ $d_s$ | $\underline{F}T$ $e_s$ | $\underline{F}T$ $e_s$ | T$\underline{F}$ $d_s$ | $\underline{F}T$ $e_s$ | T$\underline{F}$ $d_s$ | $\underline{F}T$ $e_s$ | T$\underline{F}$ $d_s$ |
| $\{\mathcal{J}_s(\underline{x}^k)\}$ | $b_sd_s$ | $b_sc_s$ $e_s$ | $b_sd_s$ | $a_sc_s$ | $a_sc_s$ | $a_sc_s$ $d_s$ | $a_sc_s$ | $b_sd_s$ | $c_s$ | $d_s$ |

TABLE 15:
OPTIMUM SINGLE FAULT TEST
FOR CONTACT NETWORK

| $\{X\}$ \ $\{f_s\}$ | $a_s$ | $b_s$ | $c_s$ | $d_s$ | $e_s$ | $d_o$ | $e_o$ |
|---|---|---|---|---|---|---|---|
| $X^4$ | | | | | | × | ✓ |
| $X^9$ | | | | | | | × ✓ |
| $X^5$ | | × | | × | | | |
| $X^6$ | | × | × | | × | | ✓ |
| $X^8$ | × | | | | × | | |
| $X^{11}$ | × | | × | × | | | ✓ |

## IV. DETECTION OF MULTIPLE FAULTS

If the occurrences of faults within a network are independent, the probability of a multiple fault occurring becomes very small. This fact, in addition to the difficulties involved in manipulating the vast number of possible multiple faults, has led to an almost universal lack of interest in tests to detect and diagnose multiple faults. There are, of course, many ways in which multiple faults can occur with probabilities close to those of single faults. For instance, if two contacts labeled a and b in a relay network have input propositions x and x', they may be realized by a transfer contact as shown in Fig. 5.

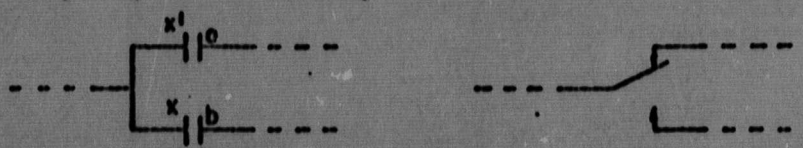



Fig. 5 Use of transfer contact

If contact a becomes permanently shorted it means that b becomes permanently open, leading to the multiple fault $(a_s b_o)$. In gate networks, a power supply lead may become disconnected causing several gates to behave improperly. A study of multiple fault detection is not without justification because, as just illustrated, the assumption that faults occur independently is not always valid.

In Section III, a procedure was outlined for constructing a fault table analogous to a prime implicant table. The fault table was used as a means of selecting the smallest set of input combinations which would detect all faults from a set $\{f_p\}$. Construction of the

fault table utilized the output proposition for the network obtained by the analysis procedure of Section II. The size of the fault table was reduced significantly through the use of column dominance. Upon completion of the column dominance simplification, the fault table no longer had a column for each possible fault. Nevertheless, an optimum test could still be obtained.

Generation of optimum tests to detect all faults, multiple as well as single, requires construction of a multiple fault table. The multiple fault table is also analogous to a prime implicant table and is solved to obtain an optimum multiple fault detection test. Clearly, a multiple fault table can be constructed by extending the fault set $\{\mathcal{F}_p\}$ to encompass all possible faults. However, for a network with $n$ elements, the number of faults considered in such an approach is $3^n-1$. Deriving a table with $3^n-1$ columns is a formidable undertaking and, as will be shown, is unnecessary.

## Critical Columns.

Suppose that a multiple fault table containing $3^n-1$ columns was actually constructed. This hypothetical multiple fault table would be simplified by column dominance prior to its solution. Performing the simplification generally results in a table with far fewer than $3^n-1$ columns. Since most of the $3^n-1$ columns are discarded, much of the labor involved in constructing a complete multiple fault table is useless.

A more practical approach is to recognize that the simplified single fault table as derived in the previous section is actually a first approximation to the multiple fault table. A test obtained from the single fault table is guaranteed to detect all single faults but, in addition, will detect many multiple faults. In particular, all multiple faults whose columns dominate a column of the single fault table will be detected. By this argument, it is clear that the optimum test to detect all faults can be found from a table which consists of the single fault table plus multiple fault columns which do not dominate any column of the single fault table. Multiple fault columns which do not dominate any column of the simplified single fault table are called critical columns.

A procedure to find the optimum multiple fault detection test is:

1. Construct a single fault table and simplify by column dominance.

2. Utilizing the network description in terms of $L_1$-sets and $L_0$-sets, derive the set of multiple faults corresponding to critical columns.

3. Construct the multiple fault table by adding the critical columns to the simplified single fault table.

4. Solve for the optimum detection test.

This procedure does not require examination of all $3^n-1$ faults, yet produces a multiple fault table from which the optimum test may be obtained.

## Derivation of Multiple Faults Resulting in Critical Columns.

Critical columns are those which do not dominate any column of the simplified single fault table. Let columns of the simplified single fault table be denoted by $C_j$ where $1 \leq j \leq J$ Let $\{\mathcal{F}(C_j)\}$ denote the set of all faults whose columns do not dominate column $C_j$. If the sets $\{\mathcal{F}(C_j)\}$ are available for each column $C_j (1 \leq j \leq J)$, the set of faults whose columns do not dominate any one of the $C_j$ must be contained in the intersection

$$\bigcap_{1 \leq j \leq J} \{\mathcal{F}(C_j)\} .$$

The intersection

$$\bigcap_{1 \leq j \leq J} \{\mathcal{F}(C_j)\}$$

is simply the set of faults whose columns are the desired critical columns.

Two problems arise in deriving the multiple faults which correspond to critical columns:

1. Computation of the sets $\{\mathcal{F}(C_j)\}$ .
2. Computation of the intersection $\bigcap_{1 \leq j \leq J} \{\mathcal{F}(C_j)\}$ .

Details of these computations are given in the Appendix. The computations rely heavily upon the compressed notation devised for literal propositions. The compressed notation permits large sets of faults to be represented in a compact form. Union, intersection and complementation of these sets may be performed while the sets are represented in the compact form.

Computation of the $\{\mathcal{F}(C_j)\}$ and their intersection to obtain the multiple faults corresponding to critical columns is a straightforward task, but still involves a good deal of manipulation. However, for networks of even modest size, it is far more convenient than examining the totality of multiple faults.

## Examples.

The contact network of Fig. 4 has five elements. A total of

$$3^5-1 = 242$$

faults are possible. Table 8 illustrates the single fault table for the network. Columns $a_o$, $b_o$ and $c_o$ are eliminated by column dominance to obtain the simplified single fault table. The optimum single fault test consists of four input combinations, $\underline{x}^4, \underline{x}^0, \underline{x}^6$, and $\underline{x}^{11}$. The procedure for finding multiple faults with critical columns reveals only two: $a_o b_s c_n d_o e_s$ and $a_s b_o c_n d_s e_o$. Both of these faults are detected by the same set of input combinations, $\underline{x}^2, \underline{x}^3, \underline{x}^5, \underline{x}^8, \underline{x}^{12}$, and $\underline{x}^{13}$. Table 16 shows the addition of this one critical column to the simplified single fault table. Optimum tests are of length five, one of which is the single fault test with one additional input combination: $\underline{x}^4, \underline{x}^0, \underline{x}^5, \underline{x}^6$, and $\underline{x}^{11}$.

TABLE 16: MULTIPLE FAULT TABLE FOR CONTACT NETWORK OF FIG. 4

| {x} \ {f} | $a_s$ | $b_s$ | $c_s$ | $d_s$ | $e_s$ | $d_o$ | $e_o$ | $c_o$ | $a_s$ $b_o$ $c_n$ $d_s$ |
|---|---|---|---|---|---|---|---|---|---|
| $\underline{x}^0$ | | | | | | | | | |
| $\underline{x}^1$ | | | | | | | | | |
| $\underline{x}^2$ | | | | | | | | × | |
| $\underline{x}^3$ | | | | | | | | × | |
| $\underline{x}^4$ | | | | | | × | | | ✓ |
| $\underline{x}^0$ | | | | | | | × | | ✓ |
| $\underline{x}^5$ | | × | | × | | | | × | ✓ |
| $\underline{x}^6$ | | × | × | | × | | | | ✓ |
| $\underline{x}^7$ | | × | | × | | | | | |
| $\underline{x}^8$ | × | | | | × | | | × | |
| $\underline{x}^{10}$ | × | | | | × | | × | | |
| $\underline{x}^{11}$ | × | | × | × | | | | | ✓ |
| $\underline{x}^{12}$ | × | | | | × | | | × | |
| $\underline{x}^{13}$ | | × | | × | | | | × | |
| $\underline{x}^{14}$ | | | | | × | | | | |
| $\underline{x}^{15}$ | | | | × | | | | | |

An optimum multiple fault detection test for the gate network of Fig. 3 may be found starting from the simplified single fault table illustrated as Table 12. There are 16 multiple faults whose columns do not dominate columns of Table 12. These 16 faults and the input combinations which detect them are:

$a_1 b_n c_n d_1 (e_n + e_1) f_n g_n (h_n + h_1) j_n k_n \ell_n$

detected by $\underline{x}^5, \underline{x}^7, \underline{x}^{10}, \underline{x}^{14}$;

$a_1 b_n c_1 d_n (e_n + e_1) f_n g_n (h_n + h_1) j_n k_n \ell_n$

detected by $\underline{x}^6, \underline{x}^7, \underline{x}^9, \underline{x}^{13}$;

$a_n b_1 c_1 d_n (e_n + e_1) f_n g_n (h_n + h_1) j_n k_n \ell_n$

detected by $\underline{x}^5, \underline{x}^{10}, \underline{x}^{11}, \underline{x}^{13}$;

$a_n b_1 c_n d_1 (e_n + e_1) f_n g_n (h_n + h_1) j_n k_n \ell_n$

detected by $\underline{x}^6, \underline{x}^9, \underline{x}^{11}, \underline{x}^{14}$.

Only four critical columns arise from the 16 faults. Addition of the critical columns to the simplified single fault table results in Table 17 from which optimum tests of

TABLE 17: MULTIPLE FAULT TABLE FOR GATE NETWORK OF FIG. 3

| $\{x\}$ \ $\{p\}$ | $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ | $f_1$ | $g_1$ | $f_0$ | $a_1$ $d_1$ | $a_1$ $c_1$ | $b_1$ $c_1$ | $b_1$ $d_1$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x^7$ | × | | | | | | × | | × | × | | | |
| $x^{11}$ | | × | | | | | × | | | | × | × | ✓ |
| $x^{13}$ | | | × | | | × | | | | × | × | | |
| $x^{14}$ | | | | × | | × | | | × | | | × | ✓ |
| $x^5$ | × | | × | | × | | | | × | | × | | ✓ |
| $x^6$ | × | | | × | × | | | | | × | | × | |
| $x^9$ | | × | × | | × | | | | | × | | × | ✓ |
| $x^{10}$ | | × | | × | × | | | | × | | × | | |
| $x^{15}$ | | | | | | | | × | | | | | ✓ |

length five may be obtained. The necessity of adding only four columns to the single fault table is striking indeed when it is realized that a total of

$$3^{11}-1 = 177,146$$

faults could occur.

## CONCLUSION

An analysis procedure for combinational logic circuits has been presented which yields an output proposition in terms of input propositions and element propositions. The network output function in the presence of any fault, single or multiple, may be found by simply inserting the appropriate truth values for the element propositions into the output propositions. If truth values for the input propositions are inserted, two sets of faults or conditions may be obtained; those which an input combination can detect and those which an input combination cannot detect.

Utilizing the properties of output propositions, the problems of detecting single and multiple faults were considered. A fault table, analogous to a prime implicant table, was constructed to obtain an optimum single fault test. A method was developed for expanding the single fault table into a multiple fault table by the addition of critical columns. Solution of the multiple fault table yields an optimum multiple

fault detection test.

Usefulness of output propositions is not limited only to the detection of faults; it also provides a useful tool in the diagnosis of network faults. A subsequent paper will deal with the problems of fault diagnosis.

## ACKNOWLEDGMENT

The author wishes to thank Professors E. J. McCluskey and W. L. Kilmer of Princeton University for their many helpful comments and suggestions, and Messrs. S. O. Chagnon, R. S. Gaines, B. A. Lindsey and W. W. Patterson for writing a CDC1604 program to perform the network analysis.

## REFERENCES

1. J.M. Galey, R.E. Norby, and J.P. Roth, "Techniques for the Diagnosis of Switching Circuit Failures," *Proc. Second Annual Symposium on Switching Circuit Theory and Logical Design* (Detroit: October 17-20, 1961), pp. 152-160.

2. W.H. Kautz, "Automatic Fault Detection in Combinational Switching Networks," *Proc. Second Annual Symposium on Switching Circuit Theory and Logical Design* (Detroit: October 17-20, 1961), pp. 195-214.

3. E.J. McCluskey, Jr., "Transient Behavior of Combinational Logic Networks," Tech. Rept. No. 13, Princeton University, Department of Electrical Engineering, Digital Systems Laboratory (January 1962); and *Proc. Symposium on Redundancy Techniques for Computing Systems* (Washington, D.C.: February 6-7, 1962).

4. S.H. Caldwell, *Switching Circuits and Logical Design* (New York: John Wiley and Sons, Inc., 1959).

5. D.A. Huffman, "The Design and Use of Hazard-Free Switching Networks," *J Assoc. for Computing Machinery*, Vol. 4, pp. 47-62 (January 1957).

6. E.J. McCluskey, Jr., "Minimization of Boolean Functions," *Bell Syst. Tech. J.*, Vol. XXXV, No. 6, pp. 1417-1444 (November 1956).

7. I.B. Fyne, and E.J. McCluskey, Jr., "An Essay on Prime Implicant Tables," Tech. Rept. No. 1, Princeton University, Department of Electrical Engineering, Digital Systems Laboratory (October 1960); and *J. Soc. Indust. Appl. Math.*, Vol. 9, No. 4, pp. 604-631 (December 1962).

8. I.B. Fyne and E.J. McCluskey, Jr., "The Reduction of Redundancy in Solving Prime Implicant Tables," Tech. Rept. No. 9, Princeton University, Department of Electrical Engineering, Digital Systems Laboratory (May 1961); also to appear in *IRE Trans. on Electronic Computers.*

9. N.A. Mastellatto and R.H. Mayne, "On Test Conditions for Combinational Logic Circuits," *submitted for publication.*

## APPENDIX

**COMPUTATION OF $\{\mathcal{J}(C_j)\}$ :** The set of faults, $\{\mathcal{J}(C_j)\}$, are those faults whose columns do not dominate column $C_j$ of a simplified single fault table. Recall that a column $\mathcal{J}^n$ dominates a column $\mathcal{J}^m$ if $\mathcal{J}^n$ has a cross in every row where $\mathcal{J}^m$ has a cross. Conversely, a column $\mathcal{J}^p$ does not dominate a column $\mathcal{J}^n$ if there is at least one row where $\mathcal{J}^n$ has a cross and $\mathcal{J}^p$ does not. There must be at least one input combination which detects the fault of column $\mathcal{J}^n$ but does not detect the fault of column $\mathcal{J}^p$. Based on this observation, the sets $\{\mathcal{J}(C_j)\}$ may be found quite easily.

Let $\{X(C_j)\}$ denote the input combinations which detect the fault of column $C_j$, that is, have a cross in $C_j$. Let $\{\mathcal{J}(\underline{x}^k)\}$ denote the set of faults which input combination $\underline{x}^k$ cannot detect. The faults of $\{\mathcal{J}(C_j)\}$ are all faults contained in the union of the sets $\{\mathcal{J}(\underline{x}^k)\}$ where $\underline{x}^k \in \{X(C_j)\}$.

$$\{\mathcal{J}(C_j)\} = \bigcup_{\underline{x}^k \in \{X(C_j)\}} \overline{\{\mathcal{J}(\underline{x}^k)\}}$$

As an example, consider the column, $C_{b_8}$, corresponding to faults $(b_8)$ in Table 5. Input combinations $\underline{x}^5, \underline{x}^6, \underline{x}^7$, and $\underline{x}^{13}$ detect fault $(b_8)$ because they yield an output of 1 in the presence of $(b_8)$ whereas they normally give an output of 0. $\{\mathcal{J}(\underline{x}^5)\}$, the faults which $\underline{x}^5$ will not detect, are those faults which, together with $\underline{x}^5$, make all literal propositions of at least one $L_0$-set true, that is, yield an output of 0. Sets $\{\mathcal{J}(\underline{x}^6)\}$, $\{\mathcal{J}(\underline{x}^7)\}$, and $\{\mathcal{J}(\underline{x}^{13})\}$ have similar properties.

**TABLE 18: TABLE OF UNDETECTED FAULTS**

| $L_0$-sets \ $\underline{x}^k$ | $\underline{x}^5$ w'xy'z | $\underline{x}^6$ w'xyz' | $\underline{x}^7$ w'xyz | $\underline{x}^{13}$ wxy'z | $\{\mathcal{J}(C_{b_8})\}$ |
|---|---|---|---|---|---|
| $w(a_o)z'(e_o)$ | $\underline{T}\underline{T}$ | $\underline{T}\underline{F}$ | $\underline{T}\underline{T}$ | $\underline{F}\underline{T}$ | $\underline{T}\underline{F}$<br>$\underline{F}\underline{T}$ |
| $w(a_o)y(c_o)z\ (d_o)$ | $\underline{T}\underline{T}\underline{F}$ | $\underline{T}\underline{F}\underline{T}$ | $\underline{T}\underline{F}\underline{F}$ | $\underline{F}\underline{T}\underline{F}$ | $\underline{T}\underline{F}\underline{F}$<br>$\underline{F}\underline{T}\underline{F}$ |
| $x(b_o)y(c_o)z'(e_o)$ | $\underline{F}\underline{T}\underline{T}$ | $\underline{F}\underline{F}\underline{F}$ | $\underline{F}\underline{F}\underline{T}$ | $\underline{F}\underline{T}\underline{T}$ | $\underline{F}\underline{F}\underline{F}$ |
| $x(b_o)z\ (d_o)$ | $\underline{F}\underline{F}$ | $\underline{F}\underline{T}$ | $\underline{F}\underline{F}$ | $\underline{F}\underline{F}$ | $\underline{F}\underline{F}$ |

A table of undetected faults is constructed to list the sets $\{\mathcal{J}(\underline{x}^k)\}$ and to facilitate formation of $\bigcup \{\mathcal{J}(\underline{x}^k)\} = \{\mathcal{J}(C_{b_8})\}$. Table 18 is such a table. It has a column for each of the input combinations $\underline{x}^5, \underline{x}^6, \underline{x}^7$, and $\underline{x}^{13}$. Entries of each column comprise the set of faults, $\{\mathcal{J}(\underline{x}^k)\}$, which input combination $\underline{x}^k$ cannot detect. Each row corresponds to an $L_0$-set of the contact network in Fig. 4. An entry in the p-th row and q-th column lists the conditions under which the input combination of the q-th column will yield an output of 0 because all literal propositions in the $L_0$-set of row p are true. The entry of row $w(a_o)y(c_o)z(d_o)$ and column $\underline{x}^5 = w'xy'z$ is $(a_o)_T(c_o)_T(d_o)_F$ since these conditions will cause a network output of 0 when $\underline{x}^5$ is applied. Rules for assigning subscripts T and F are identical to those used previously in the derivation of $\{\mathcal{J}(\underline{x}^k)\}$, those faults which $\underline{x}^k$ can detect.

The rightmost column of Table 18 is the union of the $\{\mathcal{J}(\underline{x}^k)\}$ and thus specifies $\{\mathcal{J}(C_{b_8})\}$. Simplifications of the form

$$(a_o)_T(c_o)_T(d_o)_F \cup (a_o)_T(c_o)_F(d_o)_F = (a_o)_T(c_o)_F(d_o)_F$$

are used in forming the union.

Generalizing, let $C_j$ denote the j-th column of a single fault table. Let $\{\mathcal{J}(C_j)\}$ denote the set of faults (actually includes the normal case) whose columns of the multiple fault table will not dominate column $C_j$. $\{\mathcal{J}(C_j)\}$ is obtained by constructing a table of undetected faults according to the following process:

1. A row is listed for each $1_1$-set and $1_0$-set of the network.

2. A column is included for every input combination, $\underline{x}^k$, marked with a cross in column $C_j[\underline{x}^k \in \{X(C_j)\}]$

3. The entry in column q and row p is constructed according to one of two rules:

   a. If $\underline{x}^k \in \{X(1)\}$ ($\underline{x}^k \in \{X(0)\}$) and row p corresponds to an $1_1$-set ($1_0$-set), the entry lists those conditions under which $\underline{x}^k$ can make all literal propositions of the $1_1$-set ($1_0$-set) true.

   b. If $\underline{x}^k \in \{X(1)\}$ ($\underline{x}^k \in \{X(0)\}$) and row p corresponds to an $1_0$-set ($1_1$-set), the entry is left blank.

4. The union of all $\{\overline{\mathcal{J}(\underline{x}^k)}\}$ derived in the table of undetected faults is formed to obtain $\{\mathcal{J}(C_j)\}$.

COMPUTATION OF $\bigcap\{\mathcal{J}(C_j)\}$: Suppose a table of undetected faults is constructed for each column $C_j$ of a simplified single fault table. The intersection of the sets $\{\mathcal{J}(C_j)\}$ obtained from these tables contains the faults corresponding to critical columns. The most straightforward approach to constructing $\bigcap\{\mathcal{J}(C_j)\}$ is to list each set $\{\mathcal{J}(C_j)\}$ as a disjunction of subsets. The conjunction of these

TABLE 19: $\bigcap\{\mathcal{J}(C_j)\}$ AS THE CONJUNCTION OF DISJUNCTS

$$\bigcap\{\mathcal{J}(C_j)\} =$$

$$\{(a_o)_{\underline{F}}(e_o)_{\underline{F}} + (a_o)_{\underline{F}}(e_o)_{\underline{F}}(d_o)_{\underline{F}} + (b_o)_{\underline{F}}(c_o)_T(e_o)_{\underline{F}} + (b_o)_T(c_o)_{\underline{F}}(e_o)_{\underline{F}} + (b_o)_T(d_o)_{\underline{F}} + (b_o)_{\underline{F}}(d_o)_T\}^{a_s}$$

$$\cdot\{(a_o)_T(e_o)_{\underline{F}} + (a_o)_{\underline{F}}(e_o)_T + (a_o)_{\underline{F}}(c_o)_T(d_o)_{\underline{F}} + (a_o)_T(e_o)_{\underline{F}}(d_o)_{\underline{F}} + (b_o)_{\underline{F}}(c_o)_{\underline{F}}(e_o)_{\underline{F}} + (b_o)_{\underline{F}}(d_o)_{\underline{F}}\}^{b_s}$$

$$\cdot\{(a_o)_T(e_o)_{\underline{F}} + (a_o)_{\underline{F}}(e_o)_T + (a_o)_{\underline{F}}(c_o)_{\underline{F}}(d_o)_{\underline{F}} + (b_o)_{\underline{F}}(c_o)_{\underline{F}}(e_o)_{\underline{F}} + (b_o)_T(d_o)_{\underline{F}} + (b_o)_{\underline{F}}(d_o)_T\}^{c_s}$$

$$\cdot\{(a_o)_{\underline{F}}(e_o)_T + (a_o)_{\underline{F}}(e_o)_{\underline{F}}(d_o)_{\underline{F}} + (b_o)_{\underline{F}}(c_o)_{\underline{F}}(e_o)_T + (b_o)_{\underline{F}}(d_o)_{\underline{F}}\}^{d_s}$$

$$\cdot\{(a_o)_{\underline{F}}(e_o)_{\underline{F}} + (a_o)_{\underline{F}}(c_o)_{\underline{F}}(d_o)_T + (b_o)_{\underline{F}}(c_o)_{\underline{F}}(e_o)_{\underline{F}} + (b_o)_{\underline{F}}(d_o)_T\}^{e_s}$$

$$\cdot\{(a_s)_{\underline{F}}(b_s)_T + (b_s)_T(c_s)_{\underline{F}}(e_s)_T + (a_s)_{\underline{F}}(c_s)_{\underline{F}}(d_s)_{\underline{F}} + (d_s)_{\underline{F}}(e_s)_T\}^{d_o}$$

$$\cdot\{(a_s)_T(b_s)_{\underline{F}} + (b_s)_{\underline{F}}(c_s)_{\underline{F}}(e_s)_{\underline{F}} + (a_s)_T(c_s)_{\underline{F}}(d_s)_T + (d_s)_T(e_s)_{\underline{F}}\}^{e_o}$$

disjuncts represents the desired intersection. Table 19 lists the disjunctions of subsets forming the $\{\mathcal{J}(C_j)\}$ for the contact network of Fig. 4. Each disjunct corresponds to one column of the simplified single fault table. For large networks, expanding such a conjunction to obtain the intersection is a formidable task. A procedure utilizing the structure of an intersection table and properties of literal propositions is proposed which significantly reduces the labor involved. However, use of the intersection table amounts to nothing more than a systematic method of expanding or multiplying out the conjunction of disjuncts.

An intersection table consists of the rightmost columns from the tables of undetected faults. There is a row for each $L_1$-set and $L_0$-set. Each column corresponds to one of the $\{\mathcal{J}(C_j)\}$. An intersection table for the contact network of Fig. 4 is shown in Table 20.

TABLE 20: INTERSECTION TABLE FOR CONTACT NETWORK

| $L_0$-sets \ $\{C_j\}$ | $C_{a_s}$<br>x 8 x 10<br>x 11 x 12 | $C_{b_s}$<br>x 5 6<br>x x<br>x 7 13<br>x x | $C_{c_s}$<br>x 6 11<br>x x | $C_{d_s}$<br>x 5 x 7<br>x 11 13<br>x x<br>x 15 | $C_{e_s}$<br>x 6 x 8<br>x 10 x 12<br>x x<br>x 14 | $C_{d_o}$<br>x 4 | $C_{e_o}$<br>x 9 |
|---|---|---|---|---|---|---|---|
| $w(a_o)z'(e_o)$ | FF | 0, 2, 4<br>TF<br>9<br>FT | 0, 2, 4<br>TF<br>9<br>FT | 9<br>FT | FF | | |
| $w(a_o)y(c_o)z(d_o)$ | FFF | 9<br>FTF<br>3<br>TFF | FFF | FFF | FFT | | |
| $x(b_o)y(c_o)z'(e_o)$ | 4<br>FTF<br>2<br>TFF | FFF | FFF | FFT | FFF | | |
| $x(b_o)z(d_o)$ | 1, 3, 9<br>TF<br>4<br>FT | FF | 1, 3, 9<br>TF<br>4<br>FT | FF | 4<br>FT | | |
| $L_1$-sets | | | | | | | |
| $w'(a_s)x'(b_s)$ | | | | | | 5, 6, 7<br>FT | 8, 10, 11<br>TF |
| $x'(b_s)y'(c_s)z(e_o)$ | | | | | | 5, 8<br>12, 13<br>TFT | FFF |
| $w'(a_s)y'(c_s)z'(d_s)$ | | | | | | FFF | 5, 8<br>12, 13<br>TFT |
| $z'(d_s)z(e_s)$ | | | | | | 6, 8<br>10, 12<br>14<br>FT | 5, 7<br>11, 13<br>15<br>TF |

Basically, extraction of $\bigcap \{\mathcal{T}(C_j)\}$ from an intersection table requires two phases:

1. Initial simplification of the table by removal of entries which obviously cannot contribute to $\bigcap \{\mathcal{T}(C_j)\}$.

2. Evaluation of $\bigcap \{\mathcal{T}(C_j)\}$ from the remaining entries.

INITIAL SIMPLIFICATION OF INTERSECTION TABLES. A substantial number of entries in an intersection table contribute nothing to $\bigcap \{\mathcal{T}(C_j)\}$ and may be removed. An entry in column $C_j$ of the intersection table specifies a set of faults whose columns do not dominate column $C_j$ of the single fault table. However, it is possible that all faults of the entry will dominate some other column, $C_\ell$, of the single fault table. In this event the entry may be removed.

A simple test for proving that an entry of row $i$ may be removed is:

1. Write the input proposition associated with each set of conditions which is marked F;

2. If row $i$ corresponds to an $L_1$-set ($L_0$-set), input combinations which detect faults specified by the entry are those that belong to $\{X(0)\}$ ($\{X(1)\}$) and contain the input propositions listed in step 1.

3. If the input combinations found in step 2 form a column which dominates one or more columns of the single fault table, the entry may be removed.

Suppose the test is applied to entry $(d_s)_F(e_s)_T$ in column $(C_{d_o})$ of Table 20. Entry $(d_s)_F(e_s)_T$ is derived from $L_1$-set $z'(d_s)z(e_s)$. Input combinations which detect faults of $(d_s)_F(e_s)_T$ are those which normally give an output of 0 and contain the input proposition $z'$. Examination of the Karnaugh map in Fig. 4 shows the input combinations to be $\underline{x}^6, \underline{x}^8, \underline{x}^{10}, \underline{x}^{12}$, and $\underline{x}^{14}$. These five input combinations dominate the column $C_{e_s}$ of the single fault table given as Table 8. Hence, any fault specified by $(d_s)_F(e_s)_T$ dominates column $C_{e_s}$ even though it does not dominate column $C_{d_o}$.

If an entry is derived from an $L_1$-set ($L_0$-set), the proposed test simply finds all $\underline{x}^k \in \{X(0)\}$ $\underline{x}^k \in (\{X(1)\})$, which, together with faults specified by the entry,

TABLE 21: INTERSECTION TABLE FOLLOWING INITIAL SIMPLIFICATIONS

| $L_0$-sets | $C_{a_s}$ | $C_{b_s}$ | $C_{c_s}$ | $C_{d_s}$ | $C_{e_s}$ | $C_{d_o}$ | $C_{e_o}$ |
|---|---|---|---|---|---|---|---|
| $w(a_o)z'(e_o)$ | FF | | | | FF | | |
| $w(a_o)y(c_o)z(d_o)$ | FFF | TFF | FFF | FFF | FFT | | |
| $x(b_o)y(c_o)z'(e_o)$ | TFF | FFF | FFF | FFT | FFF | | |
| $x(b_o)z(d_o)$ | | FF | | FF | | | |
| $L_1$-sets | | | | | | | |
| $w'(a_s)x'(b_s)$ | | | | | | FT | TF |
| $x'(b_s)y'(c_s)z(e_s)$ | | | | | | TFT | FFF |
| $w'(a_s)y'(c_s)z'(d_s)$ | | | | | | FFF | TFT |
| $z'(d_s)z(e_s)$ | | | | | | | |

will make all literal propositions of the $L_1$-set ($L_0$-set) true. These input combinations then detect the faults specified by the entry. The entry may be eliminated if the input combinations dominate a column of the single fault table.

Another example from Table 20 is entry $(b_0)_T(c_0)_F(c_0)_F$ from column $C_{a_0}$. Having originated from an $L_0$-set, input combinations which detect faults of $(b_0)_T(c_0)_F(c_0)_F$ must normally give an output of 1 (belong to $\{X(1)\}$) and must contain input propositions y and z'. Satisfying these requirements is $x^2$. Entry $(b_0)_T(c_0)_F(c_0)_F$ cannot be eliminated since $x^2$ dominates no column of the single fault table.

Numbers included with each entry of Table 20 indicate input combinations which detect the faults of the entry. Elimination of all possible entries from Table 20 using the test just given leaves the intersection table as shown in Table 21. Faults of $\bigcap \{\vartheta(C_j)\}$ may still be found as the intersection of columns from Table 21.

FORMATION OF $\bigcap \{\vartheta(C_j)\}$ FROM THE REMAINING ENTRIES. The $\bigcap \{\vartheta(C_j)\}$ contains faults obtained from the intersection of table entries. Evaluating the intersection of entries may be simplified by a few basic rules. For convenience, let A, B, C, and D denote ordered sets of element propositions which may comprise all or part of the element propositions appearing in the compressed notation for literal propositions. If subscripts on the element propositions contained in A, B, C, and D are complemented, the resulting ordered sets are denoted by A', B', C', and D'.

**Rule 1:**

If an intersection of entries contains two sets of conditions with forms $(A)_F$ and $(A')_F$, then all elements specified by A must be normal. $(A)_F(A')_F = (A)_N$.

**Example:**

$$(b_1g_1k_0\ell_0)_F(b_0g_0k_1\ell_1)_F = \{b_ng_nk_n\ell_n, b_1g_nk_n\ell_n, g_1k_n\ell_n, k_0\ell_n, \ell_0\} \bigcap \{b_ng_nk_n\ell_n, b_0g_nk_n\ell_n, g_0k_n\ell_n, k_1\ell_n, \ell_1\} = \{b_ng_nk_n\ell_n\}.$$

**Rule 2:**

If an intersection of entries contains two sets of conditions with forms $(A)_T$ and $(A)_F$ the pair may be simplified to $(A)_T$. $(A)_T(A)_F = (A)_T$.

**Example:**

$$(b_1g_1k_0\ell_0)_T(b_1g_1k_0\ell_0)_F = \{b_ng_nk_n\ell_n, b_1g_nk_n\ell_n, g_1k_n\ell_n, k_0\ell_n, \ell_0\} \bigcap \{b_1g_nk_n\ell_n, g_1k_n\ell_n, k_0\ell_n, \ell_n\} = \{b_1g_nk_n\ell_n, g_1k_n\ell_n, k_0\ell_n, \ell_n\}.$$

**Rule 3:**

If an intersection of entries contains two sets of conditions with the forms $(A)_T$ and $(A')_T$ or $(A)_T$ and $(A')_F$, the intersection of entries is empty. $(A)_T(A')_T = \phi$, $(A)_T(A')_F = \phi$.

**Example:**

$$(b_1g_1k_0\ell_0)_T(b_0g_0k_1\ell_1)_F = \{b_ng_nk_n\ell_n, b_1g_nk_n\ell_n, g_1k_n\ell_n, k_0\ell_n, \ell_0\} \bigcap \{b_ng_nk_n\ell_n, b_0g_nk_n\ell_n, g_0k_n\ell_n, k_1\ell_n, \ell_1\} = \phi.$$

Once it is known that certain elements of a network must be normal, there are other simplifications that may be made.

**Rule 4:**

If two sets of conditions having the forms $(BC)_{\underline{F}}$ and $(B'D)_{\underline{F}}$ appear in an intersection of entries where it is known that elements specified by C and D are normal, all elements of B must be normal as well. $(BC)_{\underline{F}}(B'D)_{\underline{F}} = (BC)_N(B'D)_N$.

**Example:**

$(a_0e_0j_0\ell_0)_{\underline{F}}(a_1g_1k_0\ell_0)_{\underline{F}}$ with e, g, j, k, and $\ell$ normal

$= \{a_ne_nj_n\ell_n, a_0e_nj_n\ell_n\} \cap \{a_ng_nk_n\ell_n, a_1g_nk_n\ell_n\}$

$= \{a_ne_ng_nj_nk_n\ell_n\}$

**Rule 5:**

If two sets of conditions having forms $(BC)_T$ and $(B'D)_T$ or $(BC)_T$ and $(B'D)_{\underline{F}}$ appear in an intersection of entries where it is known that elements specified by C and D are normal, the intersection of entries is empty. $(BC)_T(B'D)_T = \phi$, $(BC)_T(B'D)_{\underline{F}} = \phi$.

**Example:**

$(a_0e_0j_0\ell_0)_T(a_1g_1k_0\ell_0)_{\underline{F}}$ with e, g, k, and $\ell$ normal

$= \{a_0e_nj_n\ell_n\} \cap \{a_ng_nk_n\ell_n, a_1g_nk_n\ell_n\}$

$= \phi$

**Rule 6:**

A set of conditions of the form $(BC)_{\underline{F}}$ in an intersection of entries may be written as $(BC)_T$ if it is known that an element specified by B is not normal.

**Example:**

$(a_0e_0j_0\ell_0)_{\underline{F}} = \{e_nj_n\ell_n, e_0j_n\ell_n, j_0\ell_n, \ell_n\}$ if $a_0$ is true;

$= \{e_0j_n\ell_n, j_0\ell_n, \ell_n\}$ if $a_1$ is true.

Both conditions are subsets of $(a_0e_0j_0\ell_0)_T$.

**Summarizing:**

Rule 1: $(A)_{\underline{F}}(A')_{\underline{F}} = (A)_N$.

Rule 2: $(A)_T(A)_{\underline{F}} = (A)_T$.

Rule 3: $(A)_T(A')_T = \phi$, $(A)_T(A')_{\underline{F}} = \phi$.

Rule 4: $(BC)_{\underline{F}}(B'D)_{\underline{F}} = (BC)_N(B'D)_N$ if elements of C and D are normal.

Rule 5: $(BC)_T(B'D)_{\underline{F}} = \phi$, $(BC)_T(B'D)_{\underline{F}} = \phi$ if elements of C and D are normal.

Rule 6: $(BC)_{\underline{F}} = (BC)_T$ if an element of B is not normal.

In general, the procedure for finding $\mathfrak{P}\{\mathfrak{I}(C_j)\}$ from the intersection table requires the following steps:

1. Select a column from the intersection table having the fewest entries. Denote it as $C_m$. An entry $c_i$ is selected from $C_m$.

2. All non-empty intersections of entries involving $c_j$ are found.

3. The process is repeated for each entry of $C_m$ to obtain all conditions in $\oint \{\mathfrak{I}(C_j)\}$.

Finding all non-empty intersections involving a selected entry utilizes the six rules just developed. Suppose the column of $C_{a_s}$ from Table 21 is selected as $C_m$. Column of $C_{a_s}$ does not contain the fewest entries but provides a more illuminating example. Consider as $c_j$ the entry $(b_o)_T(c_o)_{\underline{F}}(e_o)_{\underline{F}}$ from row $x(b_o)y(c_o)z'(e_o)$. Every fault specified by this entry must satisfy $(b_o)_T$ or, more simply, must contain fault $(b_o)$. Since the entry is derived from an $L_0$-set, we know from Theorem 4 that among the $L_1$-sets are some which must contain literal proposition $x'(b_s)$. Every entry in rows corresponding to these $L_1$-sets must contain $(b_s)_T = b_s$ or $(b_s)_F = (b_o + b_s)$. By Rule 3, $(b_o)_T(b_s)_T = (b_o)_T(b_s)_F = \phi$. As far as entry $(b_o)_T(c_o)_{\underline{F}}(e_o)_{\underline{F}}$ is concerned, Rule 3 eliminates all rows designated by $L_1$-sets containing literal proposition $x'(b_s)$. Rule 3 may also be used to eliminate all entries containing $(c_s)_T$ and $(c_s)_F$.

Within row $x(b_o)y(c_o)z'(e_o)$, columns of $b_s$, $c_s$, and $e_s$ have entries $(b_o)_{\underline{F}}(c_o)_{\underline{F}}(e_o)_{\underline{F}}$ which clearly contain all conditions of $(b_o)_T(c_o)_{\underline{F}}(e_o)_{\underline{F}}$. Hence, when entry $(b_o)_T(c_o)_{\underline{F}}(e_o)_{\underline{F}}$ is under investigation, the columns $C_{b_s}$, $C_{c_s}$ and $C_{e_s}$ may be eliminated by Rule 2.

TABLE 22: INTERMEDIATE TABLE IN ELIMINATION OF ENTRY $(b_o)_T(c_o)_{\underline{F}}(e_o)_{\underline{F}}$

| $L_0$-sets \ $C_j$ | $(a_s)$ | $(d_s)$ | $(d_o)$ | $(e_s)$ |
|---|---|---|---|---|
| $w(a_o)y(c_o)z\ (d_o)$ | | FFF | | |
| $x(b_o)y(c_o)z'(e_o)$ | TFF | FFT | | |
| $x(b_o)z\ (d_o)$ | | FF | | |
| $L_1$-sets | | | | |
| $w'(a_s)y'(c_s)z\ (e_s)$ | | FFF | TFT | |

Performing the simplifications indicated for $(b_o)_T(c_o)_{\underline{F}}(e_o)_{\underline{F}}$ results in a new intersection table outlined by heavy lines in Table 22. This new table must be solved in exactly the same way as the original in order to complete the investigation of entry $(b_o)_T(c_o)_{\underline{F}}(e_o)_{\underline{F}}$. The intersection of $(b_o)_T(c_o)_{\underline{F}}(e_o)_{\underline{F}}$ and the intersection defined by the new table denotes all conditions specified by $(b_o)_T(c_o)_{\underline{F}}(e_o)_{\underline{F}}$ which appear in $\oint \{\mathfrak{I}(C_j)\}$.

To complete the example, suppose entry $(a_s)_T(c_s)_F(d_s)_T$ is selected as $c_j$ in Table 22. Simplifications using Rules 2 and 3 result in the final table illustrated in Table 23. All columns of the final table contain but one entry. Intersection of these entries comprises one subset of faults appearing in $\oint \{\mathfrak{I}(C_j)\}$. Only one fault is specified by Table 23. It is

$$(a_s)_T(b_o)_T(b_o)_{\underline{F}}(c_o)_{\underline{F}}(c_s)_{\underline{F}}(d_s)_T(e_o)_T(e_o)_{\underline{F}} = a_s b_o c_n d_s e_o.$$

TABLE 23: FINAL TABLE IN ELIMINATION OF ENTRY $(b_0)_F(c_0)_{\underline{F}}(c_0)_{\underline{F}}$

| $L_0$-sets \ $C_j$ | $(a_s)$ | $(e_s)$ | $(d_s)$ |
|---|---|---|---|
| $x(b_0)y(c_0)z'(c_0)$ | TFF | | FFT |
| $L_1$-sets | | | |
| $w'(a_s)y'(c_s)z(c_s)$ | | TFT | |

Having exhausted the possibilities for $(b_0)_F(c_0)_{\underline{F}}(c_0)_{\underline{F}}$, it may be removed from Table 21. Entries in row $x'(b_0)y'(c_0)z(c_0)$ may be simplified to include only those conditions that are not specified by $(b_0)_F(c_0)_{\underline{F}}(c_0)_{\underline{F}}$. Thus, entries $(b_0)_{\underline{F}}(c_0)_{\underline{F}}(c_0)_{\underline{F}}$ become $(b_0)_N(c_0)_{\underline{F}}(c_0)_{\underline{F}}$.

Investigating the remaining entries of Table 21 reveals only one other essential fault. It appears as a condition of entry $(a_0)_F(c_0)_{\underline{F}}(d_0)_{\underline{F}}$ and is $a_0b_sc_nd_0e_s$. Thus, of all the possible multiple faults which can exist in the network of Fig. 4, there are only two whose columns in the multiple fault table do not dominate a column of the single fault table.

RECURSIVE PROCEDURE FOR OBTAINING $\prod \{\mathcal{J}(C_j)\}$ FROM AN INTERSECTION TABLE: The intersection defined by an intersection table has been found when the following process terminates:

1. Select column $C_m$ having the fewest entries. Separate it from the table.
2. If column $C_m$ is empty, the process terminates.
3. Select an entry, $c_i$, from row $i$ of $C_m$.
4. Simplify the remainder of the intersection table by comparing each entry with $c_i$ and applying Rules 1 to 6 to form a new intersection table.
5. Find the intersection of $c_i$ and the intersection defined by the new intersection table. These conditions constitute a subset of $\prod \{\mathcal{J}(C_j)\}$.
6. Remove $c_i$ from column $C_m$ and simplify all entries on row $i$.
7. Return to step 2.

DETERMINATION OF CRITICAL COLUMNS FROM $\prod \{\mathcal{J}(C_j)\}$: Faults of $\prod \{\mathcal{J}(C_j)\}$ can be written out such as $a_sb_0c_nd_se_0$ or they can be specified by a collection of final tables such as Table 23. If written out, the element propositions may be inserted into the network output proposition, $P_{out}$, to find the output function in the presence of the fault. Comparison with the normal output function yields the set of input combinations in the critical column.

Input combinations of the critical columns may also be found from the final tables:

1. Form the intersection of all entries on a row of the final table.
2. Determine the input combinations which detect the conditions specified in each row using the same rules applied earlier in initially simplifying the intersection table.

Table 23 was generated in the derivation of fault $a_8b_0c_0d_1e_0$. The entry in row $w'(a_8)x'(c_8)z(c_8)$ of the final table may be detected by input combinations belonging to $\{X(0)\}$ and containing $y'$. These are seen to be $\underline{x}^5, \underline{x}^8, \underline{x}^{12}$ and $\underline{x}^{13}$. The intersection of entries in row $x(b_0)y(c_0)z'(c_0)$ is $(b_0)_T(c_0)_P(c_0)_T$. This set of conditions is detected by input combinations of $\{X(1)\}$ containing $y$. Here, $\underline{x}^2$ and $\underline{x}^3$ are suitable. Hence, the critical column for fault $a_8b_0c_0d_8c_0$ has crosses for input combinations $\underline{x}^2, \underline{x}^3, \underline{x}^5, \underline{x}^8, \underline{x}^{12}$, and $\underline{x}^{13}$.

Computing the critical column for fault $a_0b_8c_0d_0e_8$ reveals it to be identical with the column for $a_8b_0c_0d_8e_0$. Adding the one column to Table 8 following its simplification by column dominance yields the multiple fault table illustrated in Table 16. Solution of the table for an optimum multiple fault detection test yields, among others, $\underline{x}^4, \underline{x}^5, \underline{x}^6, \underline{x}^9, \underline{x}^{11}$.

# On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets

D. B. ARMSTRONG

*Abstract*—A procedure is described for finding, by shortcut methods, a near-minimal set of tests for detecting all single faults in a combinational logic net. The procedure should prove useful for nets which are too large to be treated by more exact methods [2]. The set of tests so found also appears useful for diagnosing (i.e., locating) faults. The class of faults considered is that which causes connections to be stuck at logical one or logical zero. The nets considered may include AND, OR, NAND, NOR, and NOT gates.

The bases of the procedure are the "path sensitizing" concept, and reduction of a net to its "equivalent normal form," abbreviated enf. It is shown that if a set of tests can be found which detects an appropriate subset of faults in the enf, this set will detect all faults in the original net. The enf also provides a vehicle for systematically finding the most desirable tests, namely those which each detect many faults in the net. The procedure is illustrated in detail by an example.

## I. Introduction

IN PREVIOUS papers [1]–[3] on fault detection in combinational logic, the following general strategy for finding a minimal set of fault detection tests has been employed. A particular class of faults is hypothesized, and the net in question is analyzed to find which faults of the class are detected by each possible test. For combinational logic a test is just an input combination. A test detects a fault if the net output differs from the correct output when the test is applied and the fault is present. The analysis permits a "fault table" to be constructed. A fault table is a table in which there is a row for each possible test and a column for each possible fault. A 1 is entered at the intersection of a row and column if the corresponding test detects the corresponding fault; otherwise a 0 is entered. The problem of finding a minimum set of tests then reduces to the problem of finding a minimum set of rows in the table such that they include at least one 1 in every column. This is identical to the well-known prime implicant covering problem [4].

For large nets the previously described procedure becomes extremely unwieldy because of the size of the fault table. For example, for a net with 20 input variables the table could contain more than a million rows. Galey et al. [3] presented a method for deriving the complete set of tests for each fault in a compact form. They then derived an approximately minimal set of tests for all faults by performing a prescribed sequence of intersections on the test sets for the individual faults.

Manuscript received July 12, 1965; revised October 18, 1965.
The author is with Bell Telephone Labs, Inc., Murray Hill, N. J.

In this paper the fault table approach is again invoked; however, only a very small fraction of all possible tests are entered in the table. The tests entered are primarily those which detect large numbers of faults, and the main problem attacked here is the derivation of a shortcut procedure for finding these desirable tests. This procedure also appears to ensure finding a sufficient set of tests to detect all detectable faults. This point is discussed further in the last paragraph of Section II. It is not guaranteed that the set of tests thus derived will contain a truly minimal subset, but it is likely to contain one that is near minimal. The procedure may become cumbersome when the number of paths through the net is large. This tends to be the case when many gates have a high-fanout index.

The method is developed for nets composed of any of the following gate types: AND, OR, NOT, NAND, and NOR. Only single faults are considered. The fault type is that which causes any wire to be (or appear logically to be) "stuck at one" or "stuck at zero." These names are abbreviated to *s-a*-1 and *s-a*-0, respectively, in the sequel. This is the same type as treated in previous papers [2], [3].

Section II of this paper develops some concepts underlying the procedure and Section III describes the procedure. It appears that the set of tests so generated is also useful for fault diagnosis (i.e., fault location). This aspect is briefly considered in Section IV.

Before proceeding, it is worthwhile pointing out that if a network contains logical redundancy, not all faults are detectable. Specifically we have:

*Theorem 1*

All *s-a*-1 and *s-a*-0 faults in a net are detectable if and only if the net is irredundant.

A formal proof of the theorem is omitted, since it follows directly from the definition of irredundance. A net is defined to be irredundant if it contains no redundant connections. A connection is redundant if it can be cut without altering the output functions. A cut input connection to a NAND or AND gate is logically equivalent to having that input stuck at one. A cut input connection to a NOR or OR gate is logically equivalent to having that input stuck at zero. It appears that Martellotto and Mayne [5] were first to state the aforementioned theorem. In the remainder of this paper, when detecting "all faults" is spoken of, it will mean "all detectable faults."

FOR A BETTER COPY CONTACT DOCUMENT ORIGINATOR

## II. Path Sensitizing and Reduction to Equivalent Normal Form

This procedure is based on the "path sensitizing" technique[1] for detecting faults. Figure 1 depicts a portion of a net, and it is desired to find a test which detects a *s-a-1* fault at *B*. Figure 1 shows a path from *B* to an output *J*. Suppose this is the only path from *B* to the outputs. Then a test which detects a *s-a-1* fault at *B* must be such that, when the circuit is normal, a "0" appears on *B* and "1"s appear on all remaining inputs to NAND and AND gates in the path, while "0"s appear on all remaining inputs to NOR and OR gates in the path. This is necessary to insure that when *B* changes from 0 to 1 upon occurrence of the fault, the change propagates to output *J* where it is detected. The path is now said to be "sensitized."

The previously stated conditions on a test may not hold on gates which are points of reconvergence for two or more fanout paths from some preceding gate. This complication does not invalidate the procedure used here, as will be illustrated by a later example. More details on reconverging fanout are given in Appendix II-B.

The actual changes in values along the path in Fig. 1, upon occurrence of the fault at *B*, are also indicated in Fig. 1. Connections *C*, *D*, and *E* undergo the change 1→0, while connections *B* and *J* undergo the change 0→1. It is now apparent that a test which detects a *s-a-1* fault on *B* also detects a *s-a-1* fault on *J* and *s-a-0* faults on *C*, *D*, or *E*. If it is desired to detect a *s-a-0* fault on *B*, a test would be applied such that a 1 appears on *B*, while the remaining inputs to the gates on the path remain as before. This new test would also detect the complementary set of faults to those detected by the first test on this path. Thus, the two tests together would detect all *s-a-1* and *s-a-0* faults on this path.

This suggests that if tests are chosen which detect all faults on the net inputs, all faults in the network will be detected, provided only that the set of tests chosen sensitizes a set of paths containing all connections in the net. This, in fact, is guaranteed to happen if there is no fanout in the net, because in this case there is only one path from each net input to the outputs; therefore, the set of paths originating at net inputs will contain all connections. When fanout exists there occurs the additional problem of sensitizing a set of paths which, in fact, contain all connections.

It has been observed, by trying numerous examples, that tests usually exist having the property that each sensitizes several paths through a net simultaneously. Obviously, these tests must be discovered and used if a near-minimal set of detection tests is to be obtained. It would be desirable if such tests could be found by direct inspection of the net. Unfortunately, no such direct technique has been discovered. However, if the net is first reduced to its "equivalent normal form" (enf), systematic procedures can then be applied to find these "good" tests.

The enf of a net is obtained by expanding the (generally factored) Boolean expressions representing each output function in the net, using well-known procedures, but with the following restrictions:

1) The identity of every path from each net input to the outputs must be retained.

2) In general, no redundant terms or literals resulting from the expansion may be discarded. The one exception to this rule is discussed in Appendix II-B.

The reduction of a single-output net to its enf is illustrated for the net of Fig. 2. The input variables are *a*, *b*, *c*, and *d*. The internal connections are identified by the letters *f*, *j*, *k*, *m*, *u*, and *v*; the output by *y*. The gates are identified by numbers 1 through 7. The gates are assigned to levels (1 through 5) in an obvious manner, starting with level 1 at the output and working backward. The derivation of the enf proceeds in steps. In terms of the inputs to gates in level 1, we have

$$y = (u + v)_7 \qquad (1)$$

where the subscript 7 identifies the involved gate in level 1. In terms of the inputs to level 2 we have

$$y = ((bj)_4 + (km)_6)_7 \qquad (2)$$

where subscripts 4 and 6 identify the involved gates in level 2.

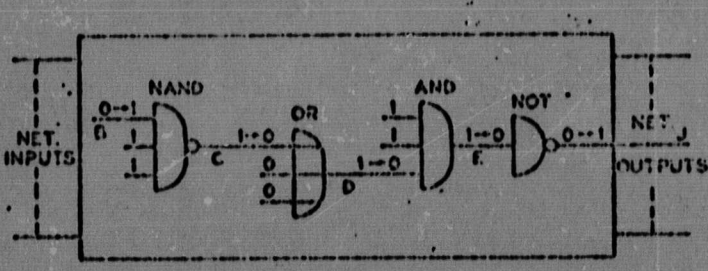



Fig. 1. Example of a sensitized path in a net.

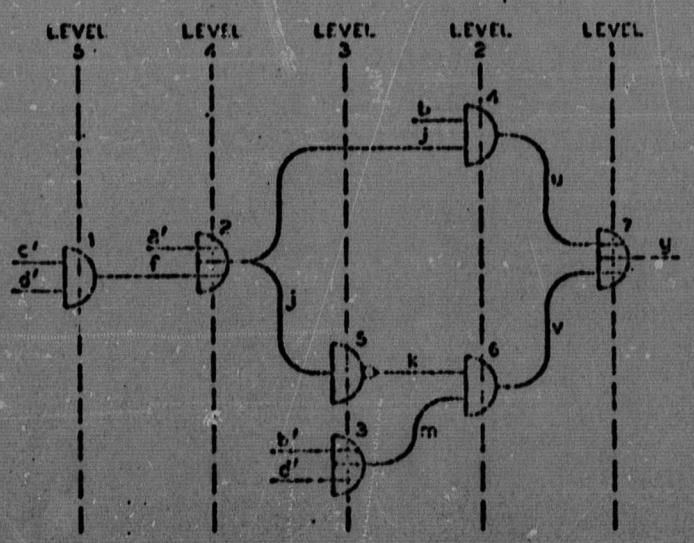



Fig. 2. An illustrative network.

[1] To the author's knowledge this approach was first suggested at the Conference on Diagnosis of Failures in Switching Circuits, sponsored by the AIEE at the University of Michigan, May, 1961. Recently, J. P. Roth [9] has also exploited this technique.

The analogous relations for the remaining levels are

$$y = ((bj)_4 + ((j')_5(b' + d')_5)_6)_7 \quad (3)$$

$$y = ((b(a' + f)_2)_4 + (((af')_2)_5(b' + d')_5)_6)_7 \quad (4)$$

$$y = ((b(a' + (c'd)_1)_2)_4 + (((a(c + d')_1)_2)_5(b' + d')_5)_6)_7. \quad (5)$$

The right side of (5) is now expanded, eliminating parentheses but retaining subscripts. The result is the enf for the net of Fig. 2.

$$y = a'_{247}b_{17} + b_{17}c'_{1247}d_{1247} + a_{2567}c_{12567}b'_{367} + a_{2567}c_{12567}d'_{367} + a_{2567}d'_{12567}b'_{367} + a_{2567}d'_{12567}d'_{367}. \quad (6)$$

Certain characteristics of the enf of a single output net, useful in the sequel, are now noted.

1) The enf is a "sum of products" expression; hence, the name normal form.

2) Each literal consists of an input variable, possibly primed, subscripted by a sequence of gate numbers. The variable and its subscript sequence identifies a path from the corresponding net input to the output. For example, the literal $a_{2567}$ in the third term of (6) identifies the path from input $a'$ on gate 2, through gates 5, 6, and 7 to the output.

3) The enf may contain redundant terms and literals even though the original net is irredundant. For example, one of the literals involving $d'$ in the last term is logically redundant. So also is the next-to-last term.

4) From the method constructing the enf and observation 3), it follows that there is at least one literal appearing in the enf for every possible path from each input to the outputs.

5) If the path associated with a literal in the enf contains an odd number of inversions (an inversion is produced by each NAND, NOR, and NOT element in the path), the priming on the literal will be opposite to that on the corresponding input in the original net. If the number of inversions on the path is even, the priming on the literal will be the same as on the corresponding input in the original net. For example, the literal $a_{2567}$ in the third term of (6) is unprimed. It is associated with the path 2567 which contains an odd number (= 1) of inversions. This literal corresponds to the input $a'$ on gate 2.

Performing fault detection on the enf of a single-output net is now considered. Since a enf is a sum-of-products expression, it corresponds to a hypothetical net consisting of a set of AND gates feeding a single OR gate. The path sensitizing rules therefore require, to test a particular literal for *s-a-1*, that literal must be assigned value 0, while the remaining literals in the term are assigned value 1. Also, at least one literal in each remaining term must be 0 in order that all remaining inputs to the OR gate be 0. By similar reasoning, a test which assigns value 1 to all literals of a particular term and assigns at least one 0 to literals in each remaining term tests all literals in that particular term for *s-a-0*.

The enf of a multioutput net is defined to be the set of enfs for the individual outputs of the net. For clarity we will refer to the individual enfs in the multioutput case as sub-enfs. The rules for detecting faults in the enf of a multioutput net are the same as those previously mentioned, except that a single test now can have the property of simultaneously testing literals in some of the sub-enfs for a *s-a-1* and literals in other sub-enfs for *s-a-0*. This is possible because the paths in the two-level AND-OR net corresponding to any sub-enf are disjoint from the paths in the two-level net corresponding to any other sub-enf.

In order to state precisely how fault detection in the enf is related to fault detection in the original net, it is necessary to distinguish between literals and appearances of literals in the enf. A particular literal may make several appearances. For example, in (6) the literal $a_{2567}$ makes appearances in each of the last four terms. A theorem and a corollary may now be stated.

*Theorem 2*

A test devised for a literal appearance in the enf sensitizes the path in the original net associated with that literal appearance.

*Corollary 2*

If a set of literals $\{L\}$ is selected whose paths contain every connection in the net, and if a set of tests $\{T\}$ can be found which tests at least one appearance of each of these literals for *s-a-1* and *s-a-0*; then the set of tests detects every *s-a-1* and *s-a-0* fault in the net.

Theorem 2 is proved in Appendix I. The corollary follows sufficiently directly from the theorem and previous discussion so that a proof is omitted. Observe that Corollary 2 does not guarantee finding a set $\{T\}$ which tests every literal in $\{L\}$ for *s-a-1* and *s-a-0*. However, this paper conjectures that such a set of tests can be found. More precisely, it is believed that the following result holds: there exist at least one set of literals $\{L\}$ and an associated set of tests $\{T\}$ that tests an appearance of every literal in $\{L\}$ for *s-a-1* and *s-a-0* and also detects every fault in the net. As yet this result has not been proven; however, efforts to find a counter example have been unsuccessful.

## III. Algorithm for Selecting Tests

Trials on several sample nets indicate that the algorithm which follows has high probability of finding the most desirable tests; namely those which each detect many faults. The algorithm will be described rather loosely, primarily by applying it to find tests for the net of Fig. 2. The first version applies to single-output nets and it is later extended to multioutput nets.

As suggested by Corollary 2, the first step of the algorithm might be to select a set of literals in the enf whose

aths cover the net, and then determine tests for these erals. However, for nets with much fanout there may many different sets of covering paths. Consequently, ·ed" choice of covering set may be made, in the sense at the best set of tests for the selected set of paths is from optimal and, indeed, may not even detect all lts.

Therefore it is chosen to proceed as follows. Tests will selected sequentially without regard initially to the icular paths they sensitize. The first test will be one hich tests as many distinct literals as possible for ther *s-a-*1 or *s-a-*0. The next test will test as many as et untested literals as possible, and so on. At various tages during the test-forming process a check may be ade to determine whether the set of literals tested to ate corresponds to a set of paths covering the net. rnatively, one could select tests until at least one ppearance of every literal is tested for *s-a-*1 and *s-a-*0. his should ensure testing every connection in the net, t may include many more tests than necessary if the et contains much fanout. The particular strategy for rminating the test-forming process is left open.

Not only are tests selected sequentially, but the construction of each test is a sequential process. That is, alues are first assigned to the smallest possible subset of the net input variables, which results in sensitizing at ast one path through the net. A further small subset the variables is then assigned so as to sensitize at ast one additional path. This is continued until all ariables have been assigned, thus completing a test.

More specifically, a *s-a-*1 test is constructed as follows. First, a term in the enf exhibiting the fewest number of variables is chosen. Call this "property A." Then, hat literal in the term such that the prime of the variable appearing in this literal has the most number of appearances in the remaining terms of the enf is chosen. all this "property B." Value 0 is then assigned to this iteral and value 1 to the remaining literals in this term. roperty A assures that when the s[illegible] term has hese values assigned, as many va[illegible] as possible in unassigned, thus giving maxi[illegible] lexibility in ng values in the remaining ter[illegible] Property B esults in many 1s being assigned in all terms, a desirble goal since the ideal condition is to have just a ingle 0 assigned in each term and the remaining literals ssigned 1s. Properties A and B are the two main heuristics employed in constructing *s-a-*1 tests. Together hey result in constructing a test which simultaneously ests a literal in each of many terms for *s-a-*1.

In practice, the order of testing literals is determined assigning a numerical score to each literal appearance. At any stage of formation of a test, variable values assigned so as to test the as yet untested literal earance of highest score, provided no other appearance of this literal has already been tested. Otherwise his literal appearance is passed over. The scoring function assigns appropriate weights to properties A and B previously given. The scoring function currently in use is defined in Appendix II-A.

The construction of *s-a-*0 tests requires somewhat different heuristics from the two used for *s-a-*1 tests, thus requiring a new scoring function. Recalling observation (5) following (6) in Section II, and the analysis of Fig. 1, it is seen that in nets containing inversions, a mixture of *s-a-*0 and *s-a-*1 faults are detected by a single *s-a-*0 or *s-a-*1 test on the enf. Therefore, it is undesirable to introduce two different scoring functions if this can be avoided. It is, in fact, avoidable by making use of theorem 3.

*Theorem 3*

A *s-a-*0 (1) test for a particular literal in the enf of a single-output net (or sub-enf of a multioutput net) is a *s-a-*1 (0) test for the corresponding literal in the complemented enf (or sub-enf).

The proof of Theorem 3 is quite direct and so is not included.

The complemented enf is defined to be the enf that is derived from the complements of the Boolean expressions representing the original net. For example, the Boolean expression representing the net of Fig. 2 is given by relation (5). The corresponding complemented enf is obtained by complementing the right-hand side of relation (5) and expanding it. The result is

$$\begin{aligned} y' = {} & a'_{2567}b'_{47} + a_{247}a'_{2567}c_{1247} \\ & + a_{247}a'_{2567}d'_{1247} + b'_{47}c'_{12567}d_{12567} \\ & + a_{247}c_{1247}c'_{12567}d_{12567} + a_{247}c'_{12567}d'_{1247}d_{12567} \\ & + b'_{47}b_{367}d_{367} + a_{247}b_{2567}c_{1247}d_{257} \\ & + a_{247}b_{367}d'_{1247}d_{367}. \end{aligned} \tag{7}$$

Theorem 3 states that *s-a-*0 tests can be obtained for literals in an enf by generating the complemented enf and finding *s-a-*1 tests for the corresponding literals in the latter. For example, a test which is a *s-a-*1 test for $a'_{2567}$ in (7) is a *s-a-*0 test for $a_{2567}$ in (6).

Therefore, both the enf and the complemented enf will be employed and only *s-a-*1 tests will be constructed, using the *s-a-*1 scoring function for both enfs. During construction of tests, the next test to be constructed will start with the as yet untested literal appearance of highest score in either the enf or the complemented enf. The remainder of the construction of this test will be confined to the particular enf in which it started.

To illustrate, the algorithm is applied to the net of Fig. 2. For brevity, replace the subscript sequences in the enf (6) and in the complemented enf (7) as follows: 247→1, 47→2, 1247→3, 2567→4, 367→5, 12567→6. Two tables are constructed, Table I for the enf and Table II for the complemented enf. Row 1 in each Table exhibits the relevant enf. Only five of the nine terms comprising the complemented enf appear in Table II. The remaining four terms were eliminated

TABLE I
TESTS FOR ENF

| enf→ | 1 $a_1'$ | 4 $b_2$ | ✓ $b_2$ .. | 1 $c_3'$ | x $d_3$ | ✓ $a_4$ | x $b_5'$ | 8 $d_5'$ | ✓ $a_4$ | 1 $b_5'$ | 8 $c_6$ | ✓ $a_4$ | ✓ $c_6$ | 1 $d_5'$ | 4 $a_4$ | x $d_6'$ | x $d_6'$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $sc_1$→ | 200 | 168 | 100 | 100 | 132 | 84 | 100 | 84 | 84 | 100 | 84 | 84 | 84 | 84 | 152 | — | — |
| Ordering of Literals | 1 | 2 | 5 | 6 | 4 | 9 | 7 | 10 | 11 | 8 | 12 | 13 | 14 | 15 | 3 | — | — |
| Test 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| Test 4 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| Test 8 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |

TABLE II
TESTS FOR COMPLEMENTED ENF

| Complemented enf | 3 $a_1'$ | 2 $b_2'$ | ✓ $b_2'$ | 3 $c_4'$ | 7 $d_6$ | ✓ $b_2'$ | 3 $b_5$ | x $d_5$ | 2 $a_1$ | ✓ $b_5$ | 5 $c_3$ | ✓ $d_5$ | x $a_1$ | x $b_5$ | 5 $d_3'$ | 6 $d_5$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $sc_1$→ | 170 | 187 | 119 | 85 | 85 | 187 | 187 | — | 17 | 51 | 17 | 17 | — | — | 136 | 85 |
| Ordering of Literals | 4 | 1 | 6 | 7 | 8 | 2 | 3 | — | 11 | 16 | 12 | 13 | — | — | 5 | 9 |
| Test 2 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| Test 3 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| Test 5 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| Test 6 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| Test 7 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

because none of their literals are testable for *s-a-1*. See Appendix II for further explanation.

The second row in the tables, labeled $Sc_1$, contains numbers proportional to the *s-a-1* score for each literal appearance, derived as shown in Appendix II-A. The third row gives an ordering of the literal appearances within the particular table according to their scores. The ordering is arbitrary in case of tie scores. Henceforth, any literal appearance is referred to by the number assigned in this row. The remaining rows contain the binary values assigned to each literal by *s-a-1* tests generated by the algorithm. Derivation of the first two tests is now described.

First, the literal appearance of highest score in either table is selected. This is literal 1 of Table I; therefore, the first test will be constructed entirely within Table I. Literal 1 is assigned value 0 and the remaining literal in this term is assigned value 1. This assigns value 1 to variables *a* and *b*. Next, assign all other appearances of these variables their appropriate values in Table I. Now check to see if the values of additional variables are forced in order to prevent the test from aborting. A test aborts if the variable assignment produces one or more terms of all 1s. Variable *d* falls in this category. It must be assigned 1 because if it is assigned 0, all literals in the last term will be 1. It is now apparent that variable *c* must be assigned 1, otherwise all literals in the second term will be 1, again aborting the test. The test is now complete, because all variables have been assigned.

Above those literal appearances in Table I which were tested by this test, the number of the test is written. Four distinct literals are seen to have been tested. A check mark is also placed above all other appearances in the enf of these literals, since only one appearance of each need be tested.

To start the second test the highest scoring literal appearance in Table I or II, which does not represent an already-tested literal, is chosen. In this case it is literal 1 of Table II, so the second test is constructed entirely in Table II. Literal 1 is assigned value 0, and the other literal in this term is assigned 1. This assigns values 0 and 1 to variables *a* and *b*, respectively. Next, assign all other appearances of these variables their appropriate values in Table II. It is now observed that no further variables need be assigned at this time solely for the purpose of avoiding aborting the test. Therefore, the as yet untested literal appearance of highest score, which is literal 2 appearing in the third term, is sought. Since another appearance of this literal has already been

tested by this test, it is passed over. Further search indicates that only one other literal can be tested by the current test, namely literal 11 in the fourth term. To test it, variables $c$ and $d$ must both be assigned value 1, completing the construction of this test. The literals tested are $a_1$ and $b_2'$. If this test is applied to the enf, it will be seen that it tests literals $a_1'$ and $b_2$ in the first term of the latter for $s$-$a$-0, as expected.

Proceeding as previously stated, eight tests which test every distinct, testable literal in both the enf and complemented enf for $s$-$a$-1 were constructed. (Literals in Tables I and II having an $x$ above them are untestable.) It is found, by constructing and reducing the complete fault table for this net, that six of these eight tests form a minimal fault detection set. Two such minimal test sets are (1, 3, 4, 5, 6, 7) and (1, 2, 3, 6, 7, 8). The underlined tests are essential; they must be included since they each detect faults not detected by any other test.

Extension of the algorithm to multioutput nets is quite straightforward. The same $s$-$a$-1 scoring function is used and is applied over the entire enf rather than to sub-enfs separately. The complemented enf is assumed available and is scored in the same manner as is the enf. The complemented enf is defined to be the set of complemented sub-enfs.

The only novel condition which can arise in forming tests in the multioutput case is that a valid test may produce terms of all 1s in some sub-enfs. The situation can be clarified by an example. Consider a two-output net, whose sub-enfs are designated $X$ and $Y$, with corresponding complemented sub-enfs designated $CX$ and $CY$. In initiating construction of a test, suppose the as yet untested literal with highest score occurs in $X$. Suppose also that the partial variable assignment which tests this literal for $s$-$a$-1 also produces a term of all 1s in $Y$. This does not abort the test, but merely means that this test will detect $s$-$a$-1 faults in the sub-enfs $X$ and $CY$.

Note that a test cannot produce terms of all 1s in both a sub-enf and its complement simultaneously, because this would imply that the corresponding net output takes on the values 0 and 1 simultaneously, which is impossible. In general, therefore, a $s$-$a$-1 test will develop over one of the members of each pair consisting of a sub-enf and its complement. Nevertheless, a partially formed test may abort, for the same reason as in the single-output case: namely, when a literal in some sub-enf has already been tested for $s$-$a$-1 by this partial test and further variable assignments then produce a term of all 1s in the same sub-enf.

## IV. Applicability to Diagnosis

It is believed that the set of tests generated by the previous algorithm for fault detection will also be reasonably adequate for fault diagnosis. An attempt to justify this conclusion by qualitative considerations is now given. A set of tests is considered to perform complete fault diagnosis if it permits every distinguishable pair of faults to be distinguished. There is a close analogy between the construction of efficient single-error-correcting codes and the construction of efficient diagnostic tests sets. The corresponding entities are shown in the accompanying table.

| Error Correction | Diagnosis |
|---|---|
| Set of bits to be corrected | Set of faults to be diagnosed |
| Set of parity check bits | Set of diagnostic tests |
| Group of bits entering a parity check | Group of faults detected by a diagnostic test |

The two important characteristics of a minimal single-error correcting code for our purposes are:

1) The number of bits in each parity check group is approximately half of the number of bits being corrected.

2) Each parity check group roughly half overlaps every other group.

These two properties result in a minimal number of check bits, approximately equal to $\log_2$ of the number of bits to be corrected.

The analogy suggests that a minimal set of diagnostic tests should have the aforementioned two properties with respect to the group of faults that each test detects. However, we do not have similar freedom in selecting diagnostic tests. The network topology fixes the group of faults detected by each test and, in general, the two properties are far from being satisfied. Analysis of realistic nets indicates that, in general, few, if any, tests detect as many as half the faults in the net, and that the amount of overlap of these fault sets is considerably less than half. It is therefore expected that in most cases the minimal number of diagnostic tests will be appreciably greater than $\log_2$ of the number of faults.

The algorithm of this paper has high probability of choosing the tests which each detect the most faults, therefore satisfying property 1) as far as possible. By making three small modifications, the algorithm will choose tests which also satisfy property 2) to an appreciable extent. These are:

1) Generate tests for every literal in the enf, not just enough literals to form a covering of the net.

2) Test every appearance of each literal rather than just one appearance of each literal.

3) If, in forming a test, the situation arises that some variables remain unassigned but no further as yet untested literals can be tested, then the remaining variables are to be assigned so as to test as many as possible already tested literals.

With the addition of the previous three rules, it is believed that the tests obtained by the algorithm will form the core of an efficient set of diagnostic tests. However, they may not form a complete diagnostic set. That is, additional tests may be needed to distinguish pairs of faults not distinguished by the previous tests. Methods for finding the additional tests are not treated here.

## Appendix I

*Proof of Theorem 2*

The theorem needs to be proven for single-output nets only, because the enf of a multioutput net is just a set of sub-enfs which are constructed independently for the individual outputs. Let the gates of a single-output net be organized into $n+1$ levels, placing the output gate on level 1 and working backward. This can be accomplished in such a way that every gate is placed on a higher level than that of any gate which it feeds.

Theorem 2 is now proved by induction. It is obviously true for $n=1$. Suppose it is true for $n$. Now modify the circuit by cutting the wires connecting the gates in level $n+1$ to the gates in lower levels. Suppose one of the gates in level $n+1$ is an AND gate, labeled $k$, having two inputs labeled $u$, $v$ and two outputs (fanout = 2) labeled $w$, $x$. The situation is as shown in Fig. 3. The enf for the modified net may be written in the following form. (The symbols 0, 1, and $X$ appearing above literals in (8) and (9) will be explained.)

$$Y_{\text{mod}} = \begin{pmatrix} 0 & 1 \\ w_{i_1}\cdots i_n & A \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ w'_{i_1}\cdots i_n & B \end{pmatrix} + \begin{pmatrix} 0 & X \\ x_{j_1}\cdots j_n & C \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ x'_{j_1}\cdots j_n & D \end{pmatrix} \quad \begin{matrix} 0 \\ E \end{matrix} \tag{8}$$

where $A$, $B$, $C$, $D$, and $E$ are Boolean variables that are independent of $w$ and $x$. The enf for the net with the AND gate $k$ included may be written in the form

$$Y = \begin{pmatrix} 0 & 1 & 1 \\ u_{ki_1}\cdots i_n & v_{ki_1}\cdots i_n & A \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ u'_{ki_1}\cdots i_n & B \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ v'_{ki_1}\cdots i_n & B \end{pmatrix} + \begin{pmatrix} 0 & 1 & X \\ u_{kj_1}\cdots j_n & v_{kj_1}\cdots j_n & C \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ u'_{kj_1}\cdots j_n & D \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ v'_{kj_1}\cdots j_n & D \end{pmatrix} + E. \tag{9}$$

Now, for example, a test of the first literal in the first term of (9) for $s$-$a$-1 requires the particular variable assignment written above the literals in (9). That is

$$u = B = D = E = 0,$$

$$v = A = 1,$$

$$C = X \text{ (i.e., don't care).}$$

This test obviously sensitizes the fragmentary path from input $u$ to output $w$ of gate $k$. This variable assignment results in $w = x = 0$; therefore, the test in question assigns the literal values written above the literals in (8). From these it is evident that this test tests the first literal in the first term of (8).

Since the theorem is assumed true for $n$ levels, the test sensitizes the path $(i_1 \cdots, i_n)$ from $w$ to the output. Therefore, the previous analysis shows that this test sensitizes the path $ki_1 \cdots, i_n$ from $u$ to the output Q.E.D. The preceding analysis may be performed with each remaining gate type (OR, NAND, and NOR) replacing AND gate $k$, and for $s$-$a$-1 tests as well as $s$-$a$-0 tests. The remaining cases will not be treated in this paper. It is also evident that the results generalize to any number of inputs and outputs from gate $k$.

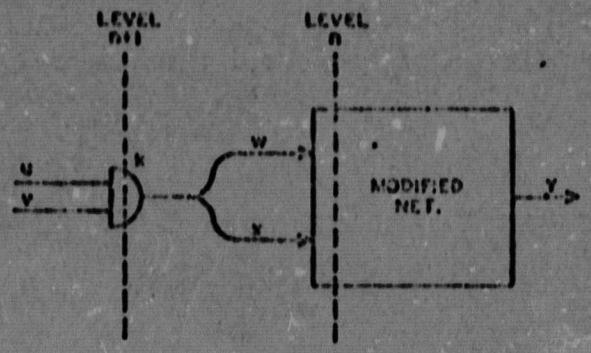



Fig. 3. Net decomposition used in proving Theorem 2.

## Appendix II

*A. A scoring function for s-a-1 faults, which appears to assign appropriate relative weights to properties A and B (which were discussed in Section III), is*

$$(Sc_1)_k = \left(1 - \frac{V_j}{V}\right) + \frac{\lambda_k}{L}$$

(provides for property A) (provides for property B)

where

$(Sc_1)_k$ = the $s$-$a$-1 score for the $k$th literal in the enf,

$V_j$ = the number of variables exhibited in the $j$th term of the enf, where this term contains the $k$th literal,

$V$ = total number of variables,

$L$ = total number of literal appearances in the enf,

$\lambda_k = \begin{cases} Ai', \text{ if the } k\text{th literal is unprimed} \\ Ai, \text{ if the } k\text{th literal is primed} \end{cases}$,[2]

$Ai$ = number of unprimed appearances of the $i$th variable in the enf, and

$Ai'$ = number of primed appearances of the $i$th variable in the enf.

To illustrate the use of the scoring function, the score for a particular literal in the enf of Table I is computed in the following table. First the values of $Ai$ and $Ai'$ for each variable in the enf are tabulated.

| Variable | $Ai$ | $Ai'$ |
|---|---|---|
| $a$ | 4 | 1 |
| $b$ | 2 | 2 |
| $c$ | 2 | 1 |
| $d$ | 1 | 4 |

[2] It is assumed that the $k$th literal represents an appearance of the $i$th variable.

For example, the first entry in column $Af$ indicates that there are four appearances of variable $a$ in the enf. These appearances are in the literal $a_1$, which occurs in each of the last four terms. In this example the number of variables $l'$ is 4. The total number of literal appearances $L$ in the enf is 17.

To compute the score of the literal $a_1'$ in the first term of the enf in Table I, observe that the number of variables exhibited in this term is 2, and $\lambda_2$ for this literal is 4. Therefore, the score for this literal is

$$Sc_1 = \left[\left(1 - \frac{2}{4}\right) + \frac{4}{17}\right] = \frac{50}{68} = \frac{200}{272}.$$

All literal scores in Tables I and II have been brought to the common denominator 272, and this denominator is omitted throughout.

*Fanout paths that reconverge again are referred to as "reconverging fanout" paths. Two cases of interest can occur.*

*Case 1:* All reconverging fanout paths between a specified fanout node and a specified node of reconvergence have the same Inversion Parity.

*Case 2:* Not all these paths have the same Inversion Parity. The Inversion Parity of a reconverging fanout path is defined to be the number of inversions, modulo 2, along the path between the specified fanout node and the specified node of reconvergence.

In Case 1, suppose a test is applied which sensitizes $n$ ($>1$) of the reconverging fanout paths. In this case the $n$-input wires to the node of reconvergence, which belong to these paths, will have the same logical value. Upon occurrence of a fault at or preceding the fanout node, the $n$ values in question will change simultaneously to the same new value. For Case 1, therefore, the sensitizing rules for a node of reconvergence are a direct extension of those given in Section II for gates which are not nodes of reconvergence. Specifically, if the node is an OR or NOR gate, all inputs other than those contained in the sensitized paths must have value 0; if the node is an AND or NAND gate, all inputs other than those contained in the sensitized paths must have value 1.

In Case 2, suppose a test is applied which simultaneously sensitizes two reconverging fanout paths with unequal Inversion Parities. The associated input wires at the node of reconvergence will have complementary values. Upon occurrence of a fault at, or preceding the fanout node, these values will change simultaneously and thus remain complementary. This implies that the output of the node of reconvergence does not change its value; hence, the effect of the fault cannot propagate beyond it. Such tests are, therefore, inadmissible.

In the worked example, two reconverging fanout paths with unequal Inversion Parities occur between gates 2 and 7 (See Fig. 2). In the complemented enf associated with this example, four of the nine terms were eliminated because no $s$-$a$-1 tests could be found for the literals appearing in them. The reason is that each of the four terms contains literals associated with both reconverging fanout paths. For example, in term $a_{247}a'_{2567}c_{1247}$ of relation (7), literal appearances $a_{247}$ and $a'_{2567}$ involve the paths in question. The values of these literals are always complementary, regardless of whether the corresponding net input $a'$, to gate 2, is faulty or normal. Therefore, $a_{247}$ and $a'_{2567}$ cannot be tested in this particular term. Furthermore, since one or the other of them has value 0 at all times, no other literal appearance in this term can be tested. Therefore, such terms may be disregarded when deriving tests.

## Acknowledgment

Suggestions by H. Y. Chang, M. C. Paull, and T. H. Crowley in deriving a proof for Theorem 2 are gratefully acknowledged.

## References

[1] R. D. Eldred, "Test routines based on symbolic logical statements," *J. ACM*, vol. 6, pp. 33–36, January 1959.
[2] J. F. Poage, "Derivation of optimum tests to detect faults in combinational circuits," *Proc. 1963 Symp. on Math. Theory of Automata*, Brooklyn, N. Y.: Polytechnic Press, pp. 483–528.
[3] J. M. Galey, R. E. Norby, and J. P. Roth, "Techniques for the diagnosis of switching circuit failures," *IEEE Trans. on Communications and Electronics*, vol. 83, pp. 509–514, September 1961.
[4] E. J. McCluskey, Jr., "Minimization of Boolean functions," *Bell Sys. Tech. J.*, vol. 35, pp. 1417–1444, November 1956.
[5] N. A. Marteilotto and R. H. Mayne, "On Test Conditions for Combinational Logic Circuits," Bell Telephone Labs., Murray Hill, N. J., unpublished.
[6] J. P. Roth "Algorithms for diagnosis for automaton failures," *IBM J. Res. and Develop.*, vol. 9, April 1966, to be published.

[63]

J. Paul Roth

24

# Diagnosis of Automata Failures: A Calculus and a Method

**Abstract:** The problem considered is the diagnosis of failures of automata, specifically, failures that manifest themselves as logical malfunctions. A review of previous methods and results is first given. A method termed the "calculus of D-cubes" is then introduced, which allows one to describe and compute the behavior of failing acyclic automata, both internally and externally. An algorithm, called the D-algorithm, is then developed which utilizes this calculus to compute tests to detect failures. First a manual method is presented, by means of an example. Thence, the D-algorithm is precisely described by means of a program written in Iverson notation. Finally, it is shown for the acyclic case in which the automaton is constructed from AND's, NAND's, OR's and NOR's that if a test exists, the D-algorithm will compute such a test.

## Introduction

This paper describes a notation and calculus for representing the behavior of failing acyclic automata,* and gives an algorithm—the D-algorithm—for the computation of tests for failures. It is established for acyclic automata (i.e., without feedback) constructed from AND's, OR's, NAND's and NOR's that if a test for any given logical failure exists, the D-algorithm will compute such a test. It is estimated that the algorithm will be about as efficient as the best of previously known techniques, which could not, however, guarantee that a test would be computed even if one existed.

Section 1 begins the treatment by providing a resume of the various methods, including that of the D-algorithm. Section 2 advances the notion and calculus of D-cubes, which are used to describe failure phenomena in circuits. Section 3 describes, via example, a manual procedure for executing the D-algorithm, while Section 4 presents a program, written in the Iverson notation, for the principal procedure of the algorithm and, as well, a verbal elucidation of that program. Section 5 gives a proof of the principal contention that if a test for failure exists, the D-algorithm will find such a test.

We approach the discussion by defining four terms that will be frequently used:

*A failure* — In a logical circuit (strictly, a Boolean graph, Refs. 2, 3) any transformation of hardware that changes the logical character of the function realized by the hardware.

*A primary input* — In a logical circuit, a line that is not fed by any other line in that circuit.

*A primary output* — In a logical circuit, a line whose signal output is accessible to the exterior of the circuit.

*A test (for a failure)* — A pattern of signals on primary inputs such that the value of the signal on some primary output will differ according to the presence or absence of that failure. Also, *test* will often be extended to mean the total pattern of signals on all lines of the circuit.

## 1. Review of methods

• *The truth-table method.*

In this method and others, let $G$ denote the function described by the circuit. If it is a multiple output circuit, $G$ might be thought of as several Boolean functions, one for each output. Let $F$ denote the function described by the circuit with a given failure. To find a test to determine whether or not this particular failure has occurred, it is "merely" necessary to compare the truth-tables for these two functions. This, of course, requires that the truth tables for each function actually be constructed and then compared to ascertain those inputs for which the output differed. Presumably, this process would be necessary for every failure of interest. Then, to determine a subset of these tests to ascertain whether or not any failure occurs, one might go through some sort of covering procedure

* References 1a & 1b show techniques such as SCAN for adapting such methods to circuits with feedback.

(*e.g.*, the extraction algorithm, Ref. 4). Clearly, this method would be effective only for small problems, certainly not for problems in 100 variables.

• *The method by complements*

Suppose that we compute the function $G$, which is defined by the good circuit and $\bar{G}$, the complement of $G$ (the set of complements, if there are many outputs). This computation may be done, *e.g.*, by the $\pi^*$-algorithm (Ref. 5). This computation gives, for each output, a normal form expression for the function and its inverse. Then to determine the set of all tests to detect the given failure, where $F$ again denotes the function of the failing circuit, it would be necessary only to form the intersections $\bar{G} \cap F$ and $G \cap \bar{F}$. This method is superior to the truth-table method in that it works with normal form expressions rather than with canonical terms. However, the formation of the intersection can be a formidable problem; for instance, if $F$ and $G$ each were expressed by a thousand terms (a thousand cubes), then a million intersections would be required. Notwithstanding, the method is in many cases a substantial improvement over the truth-table method. A procedure similar to that for the truth-table method might be used to find a set of tests to detect whether a failure has occurred and a second set of tests to ascertain, knowing that a failure has occurred, just which it is.

• *Pruning*

This method is described in Reference 5. Its basic innovation is that in one step it accomplishes the process of computing $F$, $\bar{F}$, $G$ and $\bar{G}$ and forming their intersections, as described above, in roughly the same amount of computation as to form $F$ alone. The method amounts to the following: At the point in the circuit where the failure would occur, it is "cut" and the $\pi^*$ procedure is used to compute the output in terms of the standard inputs and the "pseudo-input" at the point of cut. That is to say, the circuit is replaced by another which has the point of cut as an input (the branch above the point of cut is discarded). The branch remaining after "pruning" is then substituted, in the arrays produced, for the pseudo-input, to obtain a set of all tests to detect the given failure. Pruning is considerably more efficient than the method by complements, yet it frequently shares with the latter a need for a large memory and, possibly, a very large amount of computing time, for the so-called ON-arrays and OFF-arrays generated can be very large. The program of a considerably refined version of this algorithm, written by P. N. Sholtz, J. L. Sanborn, and J. M. Galey, has seen interesting use in IBM.

The effective computability of the pruning algorithm is limited chiefly by the size of its storage requirements for problems with a large number of inputs. For example, it was calculated that one problem in 115 variables would require $10^6$ reels of IBM magnetic tape to record a minimum normal form expression for the complement of the function, yet the circuit itself comprised only 65 logical blocks.

• *Tracing*

This method has been developed and brought to a high point of efficiency by C. B. Stieglitz.* A failure at a given point in the circuit is assumed. Then one computes the signals that are necessary at that point for the failure to be detected on the output of the block to which the point is immediately connected. The signals are "traced" to an output assigning "as-you-go," on lines feeding each block encountered, signals that are required for transmitting information concerning the alleged failure through the block. This tracing proceeds systematically, with backing up taking place whenever "conflicts" occur between signals required to appear on any given line, and continues until an output is found that is sensitive to changes in the signal of the failing line. When one such output is found, the tracing proceeds backwards from the point of failure to the inputs, in hope of finding a set of inputs which will bring up a set of signals on the failing line and on other lines interior to the circuit in order to effect this test. This procedure is continued until such a set of inputs is obtained or a specified running time is exceeded. Thus, this process computes a hypothetical test for each failure, after which it is necessary to employ "simulation" to determine whether or not the hypothetical test is a true test for that failure. If not, then computation and simulation must continue. The advantage of the tracing method is that more often than not it is fast. The disadvantage, as simple examples show, is that there is no guarantee that in fact a test will be computed, even though a test may exist.

The tracing method was improved and adapted for FLT (Fault Location Technology) by K. Maling, F. N. Evans, M. W. Evans, and W. C. Carter (Ref. 1a) and R. J. Preiss and embodied in an IBM 7094 program. This program, joined with the SCAN concept (Ref. 1b) has been massively used for the computation of diagnostic tests for IBM System/360.

• *The* D*-algorithm*

The D-algorithm we are to describe proposes a new calculus of complexes in which the internal line structure of a circuit is utilized to describe the function and its failures. (The notation it employs was alluded to in Reference 7.) The method has the advantage that the number of "terms" (more precisely, cubes) that are required to describe the function of a circuit and its failures is directly and solely proportional to the number of logical blocks in the circuit. It will be seen that if a test exists for a given failure, the algorithm will find such a test. In a particularly simple manner, the theory is extended to include the detection of

* A paper describing this method is in preparation. See Ref. 6.

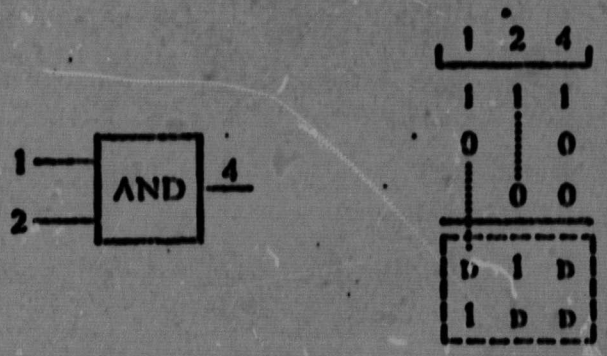



Figure 1 An AND circuit, its singular cover and, within dashed lines, two D-cubes.

short circuits between logically distant lines as well as *any* logical failures associated with the logical block.

Although explicit comparisons of methods must await results from programs soon to be run (on an experimental Iverson time-sharing interpreter at the IBM Research Center), it can be supposed that the running time of the D-algorithm will not be greater than that of the tracing method. Indeed, it should usually be considerably less, for the reason that computation of a true test with the D-algorithm requires about the same running time as does the computation of a single hypothetical test via the tracing method. Memory requirements for the D-algorithm are expected to be comparable to those for the tracing method.

## 2. Calculus of D-cubes

To describe D-cubes and their properties can best be done by presenting a few examples: Figure 1 consists of a single AND block with inputs 1 and 2, and with output 4. At the right is a kind of truth table, the first row specifying that the output is 1 when both inputs are 1, the second and third, that the output is 0 when either input is 0. This is termed the *singular cover* of the function (Ref. 2). Now, 1 and 2 are the *input coordinates* and 4 is the *output coordinate* of the logical block. The cubes that make up this singular cover may also be written with x's in lieu of blanks, which would be a form closer to the author's earlier notation. Employing x's we have

```
(1 2 4)
1 1 1
0 x 0
x 0 0.
```

The cube 0 x 0 stands for the two vertices

```
(1 2 4)        (1 2 4)
0 0 0   and    0 1 0
```

or, in a notation imposed by typography, the two vertices $^{1}0\ ^{2}0\ ^{4}0$ and $^{1}0\ ^{2}1\ ^{4}0$. In any program of the method, the blanks or, equivalently, the x's, will of necessity be represented in the coding of symbols.

Now, Figure 1 also gives two so-called D-cubes,

```
(1 2 4)        (1 2 4)
D 1 D   and    1 D D
```

or, in the alternative notation, $^{1}D\ ^{2}1\ ^{4}D$ and $^{1}1\ ^{2}D\ ^{4}D$. These have the following interpretation: The letter D may assume just two values, 0 and 1, so that D 1 D stands for the two states $^{1}1\ ^{2}1\ ^{4}1$ and $^{1}0\ ^{2}1\ ^{4}0$; it specifies that, when the logic block is working properly, the value of the signal on the output line 4 must be the same as that of the input on line 1, provided that line 2 is kept at 1.

Recalling the definition for a test, we see that, for example, the cube $^{1}D\ ^{2}1\ ^{4}D$ contains two embryonic* tests: The input signal 1 1 1 (obtained by setting D = 1) constitutes a test for line 1 stuck-at-0 and line 4 stuck-at-0; the input signal 0 1 0 (for D = 0) is a test for each of these lines stuck-at-1. A similar interpretation holds for D-cube 1 D D. In order to be quite clear to the reader interested in making comparisons with the author's previous notation, we consider another example. First, however, we remark that a vertex in the 6-dimensional "space" of coordinates 1, 2, 3, 4, 5, 6 is simply a vector of six binary digits, for example,

```
(1 2 3 4 5 6)
1 0 0 1 0 1.
```

In the three-dimensional cube $^{1}1\ ^{3}x\ ^{5}x\ ^{4}1\ ^{6}0\ ^{6}x$, each x is able to take on, independently of any other x, two values; thus this cube "covers" $2^3$ vertices formed by allowing each x to assume values 1 or 0. Likewise, the cube $^{1}D\ ^{2}1\ ^{3}D\ ^{4}0\ ^{5}D\ ^{6}D$ represents only *two* vertices, since all D's are to be thought of as *all* having the same values together. We shall be dealing with "mixed" cubes, consisting of cubes containing *both* D's and x's (or, equivalently, blank spaces rather than x's).

For example, if we hook up three logic blocks as in Figure 2, the behavior of the block with output 4 of Figure 1 is given by the first three cubes of the cover. (The cube $^{1}1\ ^{2}1\ ^{3}\ ^{4}1\ ^{5}\ ^{6}$ may also be written $^{1}1\ ^{2}1\ ^{3}x\ ^{4}1\ ^{5}x\ ^{6}x$, and similarly for the other cubes.) The next three cubes specify the input-output relationship of block 5, and the next three specify that of block 6. These nine cubes are termed the *singular cover* of the Boolean function of line 6, the output signal, in terms of its primary input signals. Internal lines are utilized in making explicit this relationship and in general, particularly for large circuits, this mode of expression is phenomenally more concise than an ordinary cubical cover (normal form). Furthermore, as we shall see, it is particularly well suited for computing tests for failures of the circuit.

* We use the term "embryonic" to emphasize that the test pattern refers only to the inputs and outputs of the block itself.

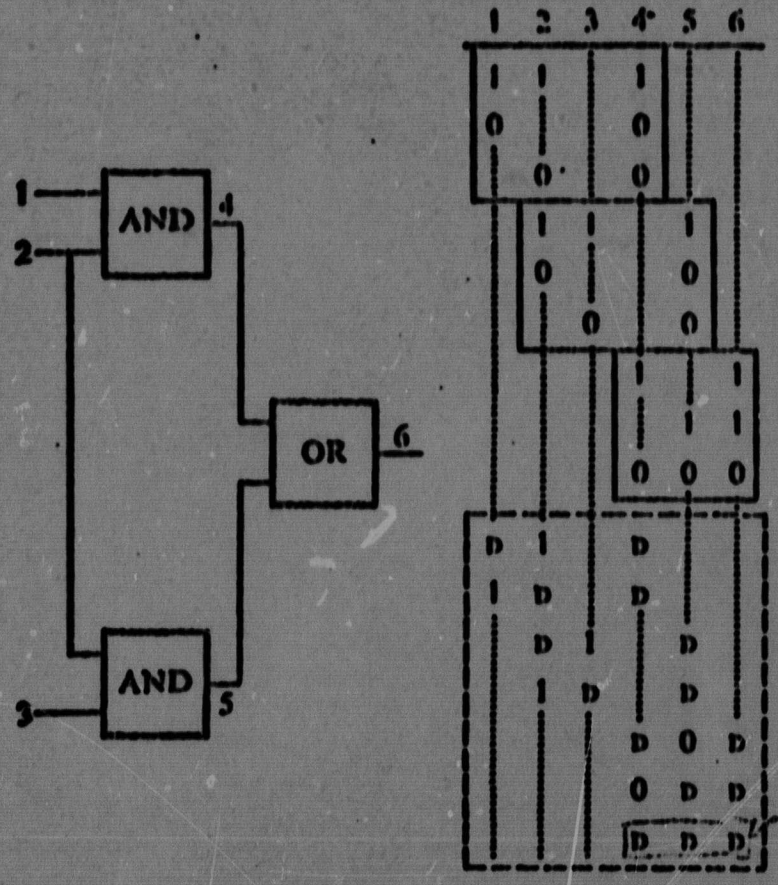



**Figure 2 A simple circuit with its singular cover and, within dashed lines, D-cubes.**

Below this singular cover are listed the "primitive" D-cubes for the circuit. Each of these contain two embryonic tests for specific failures on each block. These embryonic tests are in terms of signals on the immediate outputs of each block. For example, the cube $^4D\ ^5\bar{0}\ ^6D$ contains two tests, $^41\ ^50\ ^61$ and $^40\ ^50\ ^60$; these are tests for failures of line 4 in terms of the output on line 6. The seven D-cubes listed constitute all the D-cubes necessary to compute tests for the entire circuit. (This reflects the general case in that such a body of primitive D-cubes of logical blocks, together with the primitive D-cubes of the failures (see page 282), and together with the "multiple D-cubes" which are computed on a demand basis, are sufficient to compute tests for all failures.)

Now let us see how we can *combine* these D-cubes to compute tests for lines in terms of signals imposed on primary inputs. For example, the D-cube $^1D\,^21\,^4D$ contains tests for line 1 in terms of line 4; the D-cube $^4D\ ^5\bar{0}\ ^6D$ contains tests for 4 in terms of 6. When these are combined or "intersected" they yield

| 1 | 2 | 4 | 5 | 6 |
|---|---|---|---|---|
| D | 1 | D | 0 | D |

and we now have tests for 1 *and* 4 in terms of 6. Now, the D-cube above has this interpretation: When line 2 has signal 1 and line 5 has signal 0, lines 1, 4, and 6 will have the same signal D, with D equal to 0 or 1. Thus, for example, to test for whether line 4 is stuck-at-1 (Cf. Reference 5) it would be sufficient to have signals 1 and 0 on lines 2 and 5, respectively, and to have 0 on line 1; then if 4 is stuck-at-1, 6 will be 1, but otherwise 6 will be 0.

If we let some circuit have $n$ lines, then a D-cube of that circuit is an $n$-tuple of symbols consisting of 1's, 0's, D's, $\bar{D}$'s, x's, or blanks. The D-cube is termed *complete* if no coordinate is x or blank. The interpretation of a complete D-cube is fairly simple: each 1 or 0 represents a signal on the corresponding line; each D indicates a possible pair of signals, either 1 or 0 but both being the same for all lines corresponding to coordinates having the value D; conversely, all lines having coordinates $\bar{D}$ have the same value but one that is opposite to that of the lines having coordinates D. If a D-cube is not complete, then it has coordinates which are not specified; i.e., are blank. Let there be $r$ of these. Then this D-cube represents $2^{r+1}$ possible configurations of signals on the circuit. As indicated above, these blanks may be changed to x's as for the standard cubes that have been treated by the author. Thus in Figure 2, the incomplete D-cube $^1D\ ^21\ ^3\ ^4D\ ^5\bar{0}\ ^6D$ may be written $^1D\,^21\,^3x\,^4D\,^5\bar{0}\,^6D$.

The *vertices* of a D-cube having $r$ x's are obtained by setting each x to a 0 or 1, each D to a 1, and each $\bar{D}$ to a 0 (or vice versa, each D to a 0, and each $\bar{D}$ to a 1) to define $2^{r+1}$ such vertices. We say D-cube $A$ D-*contains* D-cube $B$ if the set of vertices of D-cube $A$ contains all those of D-cube $B$. For example, $A = 1\ x\ 0\ x\ D\ \bar{D}$ must D-contain $1\,1\,0\,x\,0\,1$ and $1\,x\,0\,1\,0\,1$, as well as six others.

### • *Construction of* $c(T, F)$

Let $T$ be a test and $F$ be a logical failure in a circuit. Then each test $T$ and failure $F$ define a D-cube, $c(T, F)$, in the following manner: If, for any coordinate, the signal as induced by the test input in line $i$ is the same, say 1, whether or not a failure exists, then the D-cube has this signal, say 1, as its $i^{th}$ coordinate. If the perfect circuit has a 1 on a given line in response to the given test and the circuit with the given failure has a 0, then assign to the corresponding coordinate a D. If the reverse, assign a $\bar{D}$.

Observe that this construction is possible only if the logic circuit is a Boolean graph, i.e., has no feedback (for if not, the signals on lines in the failing circuit are not unique). Note also that $c(T, F)$ conversely defines $T$, being the values of $c(T, F)$ on the primary inputs $pi$.

*Lemma 1*: If a given test $T$ detects a given logical failure $F$, then the coordinates expressed by the corresponding D-cube, $c = c(T, F)$, defined by $T$ and $F$, contain a connected chain of coordinates having values D or $\bar{D}$, linking the output line of the failing block to a primary output.

*Proof*: By definition, some primary output is a function

of the failure in the sense that its value changes depending upon the presence or absence of the failure. This dependence must be through some chain connecting the failing block to the primary output. For if there were no connected chain of coordinates having the value D or $\bar{D}$ linking these two, then, along every path between the failing block and the sensitive primary output, the chain of D's would terminate; the "next" block in the chain would have a fixed output value whether or not the failure had occurred. Thus, along each path from failing block to output, eventually there would be no change due to the failure. Consequently, the primary output would also not change. *Q.E.D.*

The general objective is now to combine or "intersect" the "primitive" D-cubes constructed from each block to achieve a test for a corresponding set of failures.

• *Intersection of cubes*

Let the coordinates $(a_1, \cdots, a_n)$ and the coordinates $(b_1, \cdots, b_n)$ specify cubes $a$ and $b$, where $a_i$ and $b_i$ are 0 or 1 or x. Cubes $a$ and $b$ will intersect *in an empty cube* $\phi$, if, for some $i$, $a_i = 1$ and $b_i = 0$, or $a_i = 0$ and $b_i = 1$. If this condition does not occur their intersection is specified by coordinates $(c_1, \cdots, c_n)$, where each $c_i = a_i \cap b_i$ is formed according to the rules below (Cf. Ref. 8):

$0 \cap 0 = 0 \cap x = x \cap 0 = 0;$

$1 \cap 1 = 1 \cap x = x \cap 1 = 1;$

$x \cap x = x.$

For example, $0\,x\,x \cap 1\,x\,x$ intersects in the empty cube $\phi$, but

$0\,1\,x\,x\,1 \cap 0\,x\,1\,x\,1 = 0\,1\,1\,x\,1$

and

$x\,x\,x\,1 \cap 1\,x\,0\,x = 1\,x\,0\,1.$

• *Primitive D-cubes of a failure.*

Consider a logical block with a given singular cover specifying $\alpha 1$, the totality of conditions under which the output of the block is 1 (these are specified by all those cubes whose output coordinate is 1) and $\alpha 0$, the totality of conditions under which the output is 0 (these are specified by all those cubes for which the output is 0). Now let $F$ be a failure of the logical block that changes the logical function which it performs. Let the singular cubes for which the output of the failing block is 1 be denoted by $\beta 1$ and those for which the output is 0, by $\beta 0$. For example, suppose the logical block is an AND which under a given failure is transformed into an OR, as shown below, with 1, 2 as input coordinates and 4 as output coordinate.

| 1 | 2 | 4 | |
|---|---|---|---|
| 1 | 1 | 1 | $\alpha 1$ |
| 0 | x | 0 | $\alpha 0$ |
| x | 0 | 0 | $\alpha 0$ |
| 1 | x | 1 | $\beta 1$ |
| x | 1 | 1 | $\beta 1$ |
| 0 | 0 | 0 | $\beta 0$ |

Then the totality of input conditions for which the outputs disagree, depending on whether or not there is a failure, is obtained by: (1) changing the output coordinate of all cubes of $\alpha 1$ and intersecting them with $\beta 0$ and (2) changing the output coordinate of all cubes of $\alpha 0$ and intersecting them with $\beta 1$. Those cubes in the first class we shall denote by assigning a D to their output coordinate and those of the second, by a $\bar{D}$.

In the example, we thus generate the D-cubes

| 1 | 2 | 4 |
|---|---|---|
| 0 | 1 | $\bar{D}$ |
| 1 | 0 | $\bar{D}$ |

These cubes now have the following interpretation: When the inputs 0, 1 are applied respectively to lines 1 and 2, the output will be 0 if the circuit is perfect, or 1 if it has changed to an OR output. Similarly for the input 1, 0.

In the case of a multiple output block such as we shall meet for the diagnosis of shorts between lines, the procedure for generating the D-cubes is exactly similar. For example, let 1, 2, 3, 4 be input coordinates and 5, 6 output coordinates. Suppose that the singular cover for the perfect circuit contains the cube $^1 1\ ^2 x\ ^3 1\ ^4 x\ ^5 1\ ^6 0$, and the failing circuit contains $^1 x\ ^2 1\ ^3 x\ ^4 1\ ^5 0\ ^6 1$. Then the construction yields the D-cube $^1 1\ ^2 1\ ^3 1\ ^4 1\ ^5 D\ ^6 \bar{D}$.

• *Primitive D-cubes of a logical block.*

For purposes of computing tests for failures which can be detected by primary outputs of the circuit, we introduce another type of D-cube, one which essentially specifies the signals on all inputs to a block but one (or more) under which a change of signal in this input (inputs) induces a change on the output of the block. The example of Figure 1 depicts these D-cubes.

We shall first give the construction for change on a single input coordinate $i$. For each cube in the singular cover for which the $i^{th}$ coordinate is not x, change its value and change the value of the output coordinate. Intersect this cube with each other cube in the singular cover; for each such cube in the intersection, assign the value D to the $i^{th}$ coordinate; if the value of the output in the original cube is the same as that of the $i^{th}$ coordinate, assign to it the value D; otherwise, $\bar{D}$.

Basically, we need two kinds of primitive $D$-cubes: $D$-cubes with only one input coordinate equal to $D$ or $\bar{D}$, and multiple input $D$-cubes, which have more. In constructing the multiple input $D$-cubes, it is expedient to deal with canonical covers, i.e., covers of cubes containing no x's. Otherwise, the procedure for changes in $r$ input coordinates is exactly analogous to that for one.

For example, if the singular cover contained the cubes

```
1 2 3 4 5 6,
1 0 1 0 x 1
0 1 x 0 1 0,
```

with 1, 2, 3, 4, and 5 the input coordinates and 6 the output coordinate, and with 1 and 2 the particular coordinates, the $D$-cube

```
1 2 3 4 5 6,
D D̄ 1 0 1 D
```

would be constructed. This has the interpretation that, when input lines 3, 4, and 5 have values 1, 0, and 1, then when 1 and 2 change respectively from 10 to 01 the output on line 6 changes from 1 to 0 and vice versa.

*Lemma 2.* The totality of all input configurations, such that an inversion in the value of each $r$ particular input lines effects a change in the output, is obtained by the above construction and each such input configuration is contained in one of these primitive $D$-cubes.

*Proof*: This lemma follows directly from the construction of these primitive $D$-cubes: For each cube in the cover, examine the particular $r$ coordinates in question. In canonical terms, every possible input configuration is represented. Suppose there was a multiple input $D$-cube for which there are $r$ input $D$-coordinates. Then, setting $D = 1$, we are yielded one term which must appear as a canonical term in the cover; similarly the cube obtained by setting $D = 0$ also must be in the cover and must by construction be obtained from the first by inversion of the particular $r$ input coordinates. Hence, this $D$-cube would be obtained by construction from the canonical cover. *Q.E.D.*

### $D$-*intersection.*

First we define the coordinate $D$-intersection of the symbols 0, 1, x, $D$, and $\bar{D}$. This requires the introduction of four new symbols: $\phi$, meaning empty $D$-intersection; $\psi$, meaning that $D$-intersection is not defined; and $\lambda$ and $\mu$, signifying a more complicated subroutine for the definition shown in the table immediately below:

| $\cap$ | 0 | 1 | x | $D$ | $\bar{D}$ |
|---|---|---|---|---|---|
| 0 | 0 | $\phi$ | 0 | $\psi$ | $\psi$ |
| 1 | $\phi$ | 1 | 1 | $\psi$ | $\psi$ |
| x | 0 | 1 | x | $D$ | $\bar{D}$ |
| $D$ | $\psi$ | $\psi$ | $D$ | $\mu$ | $\lambda$ |
| $\bar{D}$ | $\psi$ | $\psi$ | $\bar{D}$ | $\lambda$ | $\mu$ |

Thus, if for any coordinate, the coordinate $D$-intersection is $\phi$, then the $D$-intersection is said to be the empty cube. If any coordinate intersection is $\psi$ then the $D$-intersection is undefined.

Assume now in what follows that no coordinate $D$-intersection $\phi$ or $\psi$ occurs. Then if *both* $\lambda$ and $\mu$ occur, the coordinate $D$-intersection is not defined. If only $\mu$ occurs, then for these coordinates let $D \cap D = D$, $\bar{D} \cap \bar{D} = \bar{D}$ and let the $D$-intersection be the set of coordinate intersections. If only $\lambda$ occurs, then in the second factor change all coordinates which are $D$ to $\bar{D}$, and which are $\bar{D}$ to $D$. Then using the rules for coordinate $D$-intersection of $D \cap D = D$, $\bar{D} \cap \bar{D} = \bar{D}$, the $D$-intersection is the set of these coordinate $D$-intersections. It is seen, by comparing the intersection table in Reference 8 with that above, that when cubes $a$ and $b$ are ordinary (nonsingular) cubes, i.e., with no coordinates equal to $D$ or $\bar{D}$, that intersection and $D$-intersection coincide.

*Lemma 3*: Where products are defined, $D$-intersection is commutative and associative.

In the algorithm to follow, we will use the *restricted* $D$-intersection which insists that there be at least one coordinate for which both cubes have the value $D$ or $\bar{D}$; that is, their $D$'s must "interact." We shall say that $D$-cube $a$ *$D$-contains* $D$-cube $c$ if for each coordinate $i$, $a_i = c_i$ or x. It follows from this definition that if $a$ $D$-contains $c$, then the set of all vertices of $a$ contains the set of all vertices of $c$, so that $a$ also contains $c$. If $a$ $D$-contains $c$ we shall write $a \supset c$.

*Lemma 4.* Let $T$ be a test for a failure $F$ of logic block $\lambda$ and again let $c(T, F)$ be the $D$-cube defined by this test and failure. Then $c(T, F)$ is $D$-contained in a primitive $D$-cube of failure $F$.

*Proof*: The coordinate(s) of the output of $\lambda$ in $c(T, F)$ must clearly be $D$ or $\bar{D}$ since otherwise no primary output would be capable of distinguishing the existence or nonexistence of failure $F$, by the above lemma. Thus consider the $D$-cube defined by having the same values as $c(T, F)$ for the immediate inputs to $\lambda$, the same value $D$ (or $\bar{D}$) which $c(T, F)$ has for the output of $\lambda$ and, for all other coordinates, its value is x: this is clearly a primitive $D$-

cube of the failure $F$ and it clearly D-contains $c(T, F)$. *Q.E.D.*

*Lemma 5*: If D-cubes $a$ and $b$ each D-contain D-cube $c \neq \phi$, then their D-intersection is defined and it D-contains $c$.

*Proof*: By hypothesis, for each coordinate, $a_i = c_i$ or x and $b_i = c_i$ or x, so that $a_i \cap b_i = c_i$ or x or $\mu$. Now if the coordinate D-intersection is $\mu$, then $c_i$ must be D or $\bar{D}$ and, according to the rule for D-intersection, this coordinate is changed to D or $\bar{D}$. It follows for this case that $a_i \cap b_i = c_i$. Hence in general, the coordinate D-intersection $a_i \cap b_i = c_i$ or x so that $a \cap b$ by definition D-contains $c$. *Q.E.D.*

*Lemma 6*: Let $T$ be a test for a failure $F$ of a given Boolean graph (a logical circuit without feedback). Let $c(T, F)$ denote the D-cube defined, as described above, by $T$ and $F$. Let $\lambda$ be a logic block, not the site of the failure $F$, for which the output coordinate of $\lambda$ in $c(T, F)$ is D (or $\bar{D}$). Then $c(T, F)$ is D-contained in a primitive D-cube of $\lambda$.

*Proof*: If an output coordinate of block $\lambda$ of $c(T, F)$ consists of a D or $\bar{D}$, then at least one input coordinate to the block of $c(T, F)$ must also be D or $\bar{D}$, for if all the input coordinates had values 1 or 0, i.e., did not change whether or not failure $F$ existed, then the output also obviously could not change.

Let $a_{i(1)}, \cdots, a_{i(r)}, b_{j(1)}, \cdots, b_{j(s)}$ be the coordinates of $c(T, F)$ for logic block $\lambda$, with the $a$'s input coordinates and the $b$'s the output coordinates. Assigning to D the value 1 and to $\bar{D}$ the value 0, we define a cube which is "covered" by the singular cover of $\lambda$; i.e., each of its vertices is contained in some cube of the singular cover. Similarly a cube is defined for the opposite assignment: D = 0 and $\bar{D}$ = 1. On the other hand, these two cubes (they are vertices of the cover of $\lambda$), when combined by the method above, go together to form a primitive D-cube *pdc* of the block $\lambda$. Now, to be explicit, if *pdc* is extended over all coordinates of the Boolean graph by assigning the value x to those as yet unassigned, then it is clear that *pdc* D-contains $c(T, F)$. *Q.E.D.*

• *Detection of short circuits.*

We next consider shorts between lines of the circuit. Consider the typical case as shown in Fig. 3, consisting of logical blocks A and B whose output lines a and b cross. If the lines short at the point of crossing we shall assume that the short behaves as a so-called dot-OR function. The treatment is exactly analogous in case the short behaves as an AND. We assume, therefore, the existence of a pseudo-block C with inputs $a$ and $b$ and outputs $a^*$ and $b^*$. When no short occurs $a^* = a$ and $b^* = b$. Under failure, however, $a^*$ and $b^*$ become the OR of $a$ and $b$: $a^* = b^* = a \vee b$. It is easy to show therefore that the D-cubes which

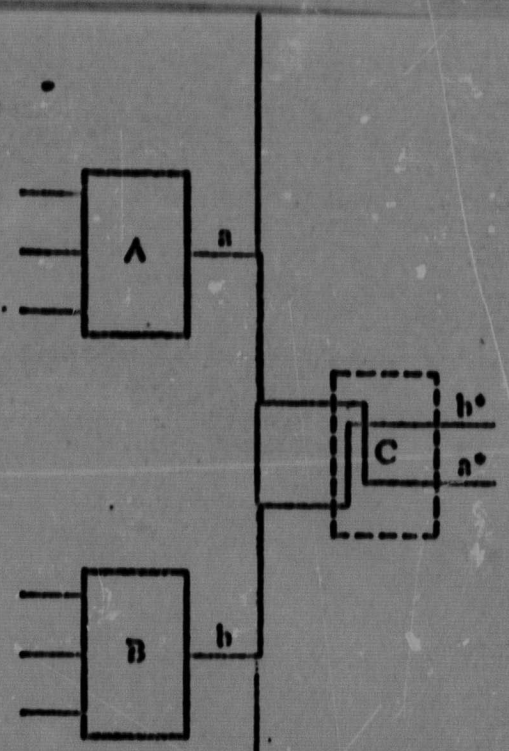



Figure 3 Shorts between lines: an illustrative circuit.

contain all tests to detect these failures are

| $a$ | $b$ | $a^*$ | $b^*$ |
|---|---|---|---|
| 1 | 0 | 1 | $\bar{D}$ |
| 0 | 1 | $\bar{D}$ | 1. |

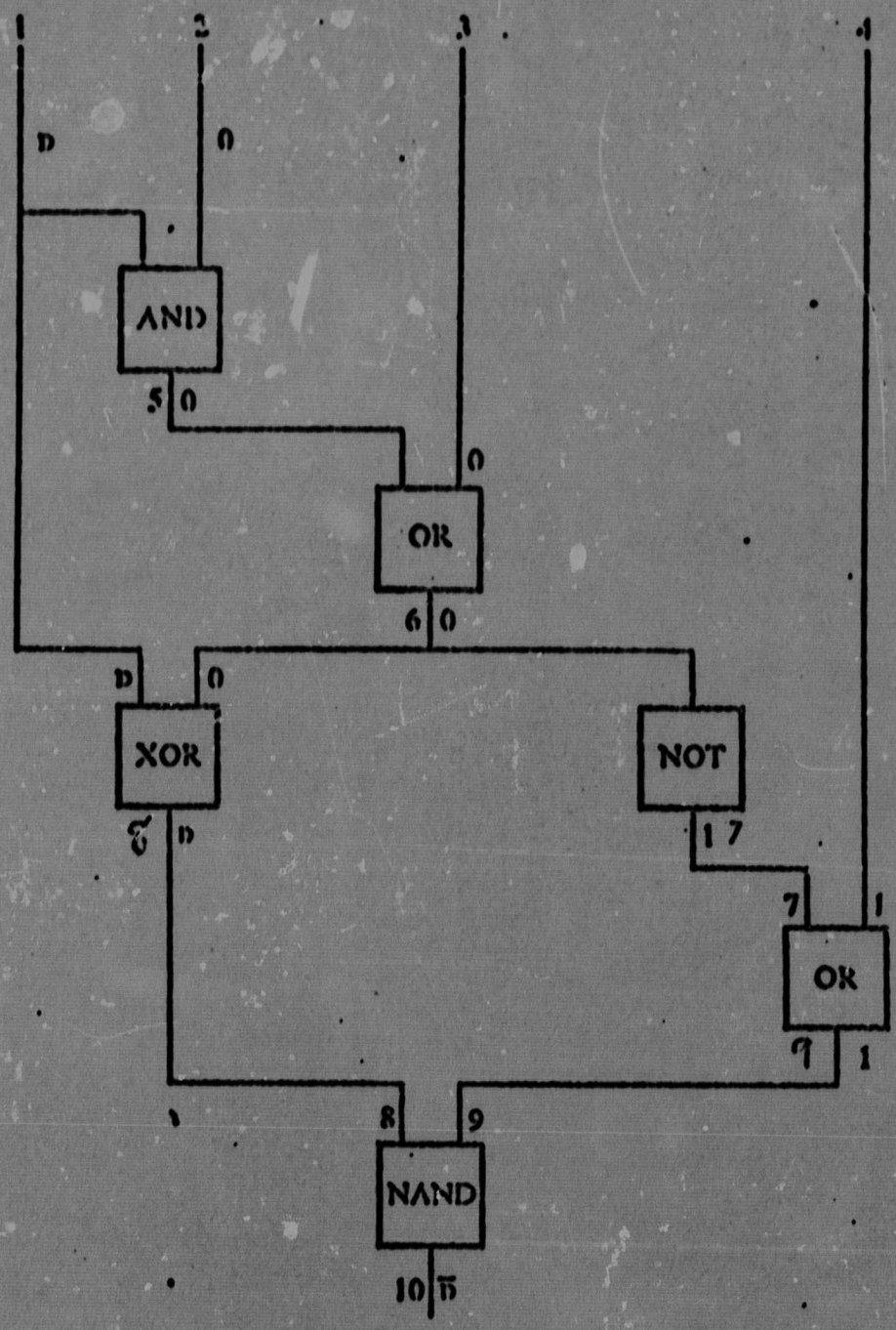
These cubes may therefore be inserted into the algorithm for the generation of tests of Section 3 or 4, to obtain tests in terms of primary inputs for this failure.]

## 3. Manual procedure for D-algorithm

The D-algorithm will be described in two forms: First by means of a manual procedure in this section, in which the set of primitive D-cubes will be priorly computed and stored, and second, by means of a program written in Iverson notation in Section 4. In both cases the underlying steps are quite simple: *First*, having selected a primitive D-cube of a given failure, this cube is successively D-intersected with primitive D-cubes in an attempt to form a connected chain of D-coordinates to a primary output; *second*, having generated such a D-cube, we attempt to "complete" it by means of D-intersecting it with the singular cover $S$. This will be called the CONSISTENCY operation.

We shall describe the manual procedure by means of the example shown in Figure 4. The figure on the left is described analytically by the singular cover $S$ on the right, consisting of rows a through r. The primitive D-cubes of the logic blocks (the single-output variety) are represented by the rows A through M.

The first step in the algorithm is to select at the site of the failure a primitive D-cube of the failure. Consider first the case of a failure in line 1 (we treat the failures stuck-at-1 and stuck-at-0 simultaneously): Thus the initial test cube $tc^0 = {}^1\mathrm{D}$. We form the following chain of D-inter-



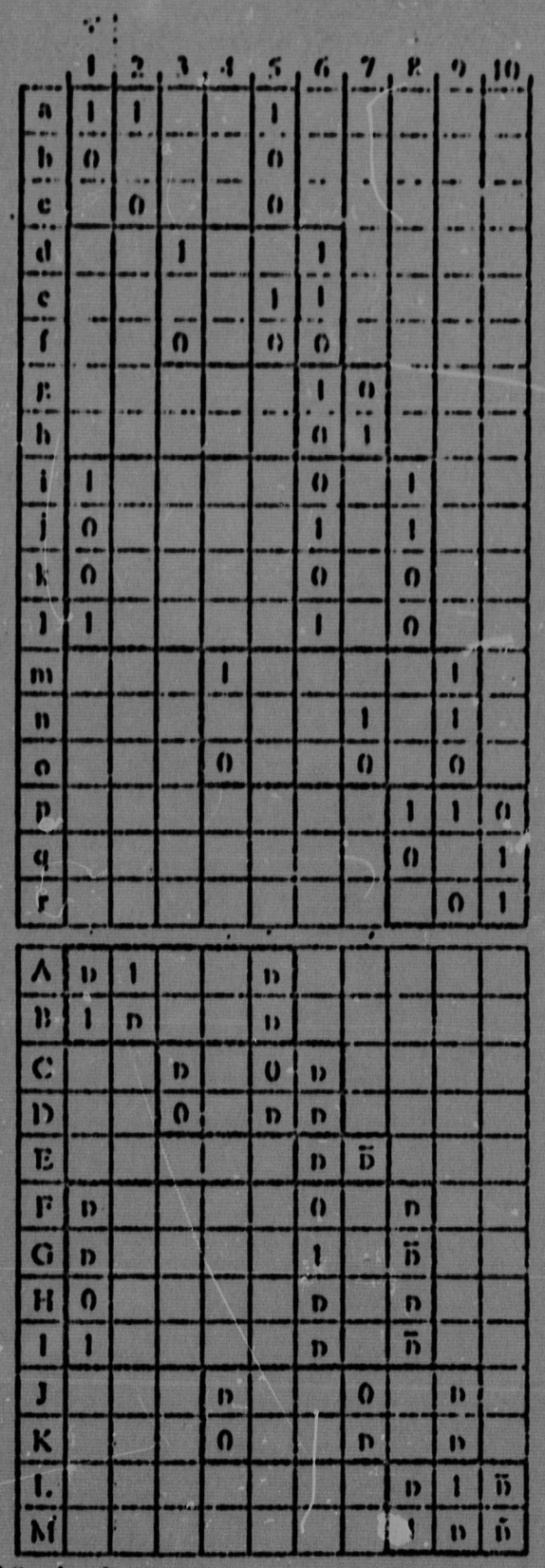



Figure 4 The circuit, singular cover, and primitive D-cubes for the D-algorithm of Section 3.

sections: (The order for D-intersection should be lexicographical; it turns out, in this example, that the first such intersection that "works" is the one shown.)

$$\underline{1\ 6\ 8}$$

$$tc^1 = tc^0 \cap F = \mathrm{D}\ 0\ \mathrm{D}$$

$$tc^2 = tc^1 \cap L$$

$$\underline{1\ 6\ 8\quad 8\ 9\ 10}\ ,\quad \underline{1\ 6\ 8\ 9\ 10}$$

$$= \mathrm{D}\ 0\ \mathrm{D} \cap \mathrm{D}\ 1\ \bar{\mathrm{D}} = \mathrm{D}\ 0\ \mathrm{D}\ 1\ \bar{\mathrm{D}}.$$

Now for each D-cube $tc$ we define the *activity vector* $a$ consisting of the set of all coordinate numbers $j$ of $tc$ for which: (1) the coordinate $tc_j = \mathrm{D}$ or $\bar{\mathrm{D}}$; and (2) this coordinate $j$ is a primary output of the subgraph of the circuit defined by those coordinates of $tc$ not equal to x. In the example above $a^1$ of $tc^1$ is $\{1, 8\}$; for $tc^2$, $a^2 = \{10\}$.

In Section 4, the notion of the D-fanout of $c$, $fo$, figures prominently. It keeps track of the number of successors of each element of $a$ which are potential extenders of the D-chain to primary outputs, i.e., are on the "frontiers" of $tc$; if a line is a primary output, then 1 is added to this

number (see step 3 of Fig. 5.). In this example $fa^1 = ({}^1 1, {}^8 1)$.

Thus, returning to the description of the algorithm, when the D's have been driven forward as far as possible and primary outputs $po$ have been encountered, the CONSISTENCY operation, moving back to the primary inputs, via D-intersecting with $S$, is begun. Thus, in the example, since the activity vector $a^2$ of $tc^2$ consists only of the primary output 10, the first part of the algorithm is finished.

The next step is the CONSISTENCY operation. The purpose of this operation is to "fill in" the remaining coordinates and to ensure that they are "consistent" with the singular cover. In general, it is necessary to have "prime" cubes (prime implicants) as a singular cover for this operation. This time we shall, for the sake of convenience, write down all the singular cubes in tabular fashion and then intersect them all simultaneously with $tc^2$.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $tc^2 =$ | D | | | | | 0 | | D | 1 | D |
| $n$ | | | | | | | 1 | | 1 | |
| $h$ | | | | | | 0 | 1 | | | |
| $f$ | | | 0 | | 0 | 0 | | | | |
| $\cap c$ | | 0 | | | 0 | | | | | |
| $tc^3$ | D | 0 | 0 | | 0 | 0 | 1 | D | 1 | D |

The fourth coordinate is blank (or x); this means that it may be given an arbitrary value. Thus $tc^3$ describes four tests, the primary inputs for which are:

| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |

The first two of the four tests are for line 1 being stuck-at-0, the last two, for line 1 stuck-at-1.

Another path in the algorithm also applied to the D-cube ${}^1D$ of the failure line 1 stuck-at-1 (or stuck-at-0) is the following, shown up to $tc^3$:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $tc^0 =$ | D | | | | | | | | | |
| $tc^1 = tc^0 \cap A =$ | D | 1 | | | D | | | | | |
| $tc^2 = tc^1 \cap D =$ | D | 1 | 0 | | D | D | | | | |
| $tc^3 = tc^2 \cap E =$ | D | 1 | 0 | | D | D | D | | | |

At this point, the vector $a$ of active coordinates consists of 1, 6, and 7. Now the block with output 8 has both 1 and 6 as inputs and, therefore, the need for a "double" D-cube is indicated. Let us go through this computation.

The first cube in the singular cover for block with output 8 is ${}^1 1\ {}^6 0\ {}^8 1$. In accordance with the method for generating multiple D-cubes (Section 2), change the input coordinates affected, 1 and 6, and its output, 8, to obtain ${}^1 0\ {}^6 1\ {}^8 0$ and D-intersect it with the remaining cubes ${}^1 1\ {}^6 1\ {}^8 0$ and ${}^1 0\ {}^6 0\ {}^8 0$. They do not D-intersect so that a double D-cube does not exist and the generation of the test cube $tc^4$ is abandoned.

It is instructive to compute a test for line 2. The table below shows the result $tc^3$ of the first five effective D-intersections, up to but not including D-intersection with the primitive D-cubes of the block with output 10.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $B$ | 1 | D | | | D | | | | | |
| $D$ | | | 0 | | D | D | | | | |
| $E$ | | | | | | D | D | | | |
| $I$ | 1 | | | | | D | | D | | |
| $\cap K$ | | | | 0 | | | D | | D | |
| $tc^3$ | 1 | D | 0 | 0 | D | D | D | D | D | |

The vector $a$ of active coordinates consists of lines 8 and 9, both of which are inputs to block 10. This fact indicates a need for a "double" D-cube, and this is generated by the following procedure:

Cube ${}^8 1\ {}^9 1\ {}^{10} 0$ has its input coordinates 8 and 9 changed and also its output coordinate to yield the cube ${}^8 0\ {}^9 0\ {}^{10} 1$. This cube is tested to ascertain whether or not it lies in the singular cover: it does, because cube q is ${}^8 0\ {}^9$ ${}^{10} 1$ or

$^{9}0\,^{8}x\,^{10}1$, whose intersection with $^{8}0\,^{9}0\,^{10}1$ is indeed $^{8}0\,^{9}0\,^{10}1$. This then defines the double D-cube $N = {}^{8}D\,{}^{9}D\,{}^{10}\bar{D}$.

We now return to the subroutine for generating the test cube $tc^{6} = tc^{5} \cap N$,

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $tc^{5}$ | 1 | D | 0 | 0 | D | D | $\bar{D}$ | $\bar{D}$ | $\bar{D}$ | |
| $N$ | | | | | | | | D | D | $\bar{D}$ |
| $tc^{5} \cap N$ | 1 | D | 0 | 0 | D | D | $\bar{D}$ | $\bar{D}$ | $\bar{D}$ | D |

Now the vector $a$ of active coordinates consists solely of line 10, which is a primary output $po$, $a \subset po$.

The next step is the CONSISTENCY operation. In this case, the set $g$ of coordinates of $tc^{5}$ which have the values 1 or 0 consist solely of primary inputs $pi$, $g \subset pi$, so that the CONSISTENCY operation need not be performed. This completes the computation for the test cube $tc^{6}$. This cube determines the tests $^{1}1\,^{2}1\,^{3}0\,^{4}0$ for line 2 stuck-at-0, and $^{1}1\,^{2}0\,^{3}0\,^{4}0$ for line 2 stuck-at-1.

A method has been developed using D-notation for identifying all failures that a test detects; that method will be described in a subsequent paper.

## 4. Programming the D-algorithm

A program is given here for the central part of the D-algorithm. It is written, as Fig. 5 [the foldout] reveals, using a quite restricted subset of the Iverson notation. Sections 1.2 through 1.5 of Reference 9 will be found to cover all that is used here. An alternative reference is Falkoff, Iverson, and Sussenguth, Ref. 10, pp. 198–202. Further, the present section also gives a step-by-step description of the program itself.

In contradistinction to the manual procedure of Section 3, the program here is restricted to logic circuits composed of AND's, OR's, NAND's and NOR's (the transistor-type of logic used almost universally in today's technology). This restriction would be easy to remove, however, involving as it does only a few instructions. It is also assumed that the logic blocks or lines of the circuits are labelled with integers in such a way that the number assigned to any block exceeds that of all the lines that feed it; a very simple method is sufficient for such an assignment.

In Table 1 we provide a list of the symbols used in the program, together with their meaning, given in the order of their appearance in the discussion below.

There are two parts to the algorithm. In the first, a primitive D-cube $tc$ of the failure is recursively intersected with primitive D-cubes of logic blocks seeking to form a D-cube $tc$ which provides a "connected chain" of D-coordinates to some primary output $po$ (Steps 1 through 38). The second part, the CONSISTENCY operation (Steps 39 to 58), consists of intersecting this D-cube $tc$ with the singular cover $S$. This amounts to "driving backward"; i.e., amounts to successively assigning values to the uncommitted coordinates (those equal to x) in a way consistent with the singular cover, if this is possible.

Step 1 is to set initial parameters $m$ and $h_m$ to 0. Step 2 is to load $tc$, the test cube, with a primitive D-cube of a given failure. Step 3 defines the "activity" vector $a$, consisting of all coordinates $j$ whose value is equal to D or $\bar{D}$; the values of $a$ are integers, belonging to the set $\iota$ of all integers. Referring to Step 4, $d^{k}$ consists of the set of logic blocks driven by block $k$, $\nu d^{k}$ means the *number* or cardinality of this set, and $\nu d^{a}$ is to denote the vector whose $k^{th}$ component is this number. To complete the initialization of $fa$, the quantity $\epsilon_{a}^{po}$ is added to $\nu d^{a}$ in order to record the fact that any particular $a_j$ is a $po$.

Step 4 unconditionally branches to Step 9 (we shall return to Steps 5 through 8). Now $j$ is the index on the coordinate number of the activity vector $a$; it runs, therefore, from 0 to $\nu a$, the dimension of $a$. Thus Step 9 sets $j$ to 0. Step 10 compares $j$ with $\nu a$; if $j \neq \nu a$, then Step 11 follows, which increments $j$ by 1, $j \leftarrow j + 1$. (On the first time in reaching Step 10, $j$ would be 0 and thus in general less than $\nu a$. Thus we proceed first with the chain of instructions following the branch $j \neq \nu a$.)

In Step 12, $k$ is the index on the set (or vector) $d^{a_j}$ of logic blocks driven by $a_j$. Thus $k$ ranges from 0 to the number $\nu d^{a_j}$ of elements in $d^{a_j}$. Step 12 sets $k$ to the initial value of 0 and Step 12.1 defines the final value of $kf$.

Step 13 compares the magnitude of $k$ with $\nu d^{a_j}$. If in Step 13 $k = kf$, then we return to Step 10 where we try the next coordinate of $a$ if possible.

If $k \neq kf$, $k$ is incremented by 1 in Step 14. (Thus, for the first time through, $k = 1$.)

Now, to simplify notation, in Step 15 and subsequent steps we let $d_{k}^{a_j}$, the $k^{th}$ successor of $a_j$, be denoted by $s$ ($s$ for successor). For the same reason in Step 16 and subsequent ones, $\tau^{s}$, the set of predecessors of line $s$, is denoted by $p$.

Now the algorithm as described in this program is restricted for convenience of expression to the logic blocks AND, OR, NAND, and NOR. In Step 17, the expression within oblique brackets /;;/ defines a vector $t$. The expression $\epsilon_{p}^{a}$ is a vector obtained from $p$ by replacing each component by a 1 or 0 according to whether it is or is not in $a$. When it is, the value /;;/ is D, the right term, so that the corresponding coordinate of $t$ would also be D. When it is not, the value of the corresponding coordinate of $t$ is the *value* of the proposition ($\wedge$, $\underline{\vee}$ AND, NAND): 1 if logic block $s$ is an AND or NAND, and 0 otherwise, in which case it is either an OR or NOR. Thus Step 17 determines all the values of the input coordinates of $s$.

Step 18 determines its output coordinate by the expression in the oblique brackets and catenates this with $t$, to form $pdc$, the appropriate primitive D-cube of $s$. Within

/;;/ now, ↓, ∨ NOR, NAND is a proposition equal to 1 or 0 depending upon whether or not the logic block of $s$ is or is not a NOR or NAND. If it is, then the output coordinate shall be Ð, the right member of the expression. If not, then D. Thus is *pdc* the primitive D-cube of $s$ defined.

Step 19 forms the D-intersection of *tc* with this primitive D-cube *pdc*. However, only the subset of the coordinates of *tc* corresponding to the block $s$ are needed and these are specified by the subscript $p, s$. The D-intersection is called $w$.

Now in contrast to the definition in Section 2, this $w$ is meant to be the coordinate D-intersection and in Step 20 this is tested. Step 20 evaluates the proposition: Are any coordinates of $w$ equal to $\phi$, $\psi$ or do both $\lambda$ and $\mu$ appear? If this proposition is true (equals 1) then no D-intersection is formed and we branch to Step 28 to update $a$, $fa$, and $j$.

Step 21 tests to ascertain whether or not a $\mu$ occurs in the coordinate D-intersection $w$. If $\mu$ does *not* occur, then, according to the rules for D-intersection, the D's and Ð's of *pdc* must be interchanged. This is accomplished in steps 22 and 23.

Step 22 defines the intermediate primitive D-cube *ipdc*: where *pdc* has coordinate D, *ipdc* has coordinate Ð; where *pdc* does *not* have coordinate D, *ipdc* coincides with *pdc*. In step 23, where *pdc* had coordinate Ð, respecified *pdc* has D; otherwise the "new" *pdc* coincides with *ipdc*. If $\mu$ *does* occur (in step 21) then we skip steps 22 and 23. In either case we arrive at step 24.

Steps 24 through 27 are concerned with what quantities, if any, should be stored in case the branching process returns to this stage in the execution of the algorithm.

Steps 28 through 31 are bookkeeping operations associated with $a$. Now $a$ consists of those coordinates of *tc* which are "active", that is, coordinates which lie on the frontiers of *tc*. The respecification of $a$ (and $fa$) is accomplished by use of the survivor vector *sur*. Imagine that for each coordinate of $a$ that is a predecessor $p$ of $s$, the fanout vector $fa$ is decremented by 1, i.e., $nfa \leftarrow (fa - \epsilon^{p}_{o})$. Only those coordinates of "new" $fa$ which are not zero should survive since only they have a chance of propagating. This *sur* is defined in step 28 as a logical vector whose $i^{th}$ coordinate is 1 if and only if $nfa_i$ is 1; otherwise 0. In step 29 all coordinates of $nfa$ that are 0 are deleted, i.e., *sur* compresses $nfa$, this respecifies $a$. In step 30 $a$ is similarly compressed by *sur*, to eliminate all coordinates of $a$ which are not on the frontiers of *tc*. Step 31 updates the index $j$ on $a$ to compensate for the compression. For each coordinate $r \leq j$ whose corresponding entry in *sur* is 0, $j$ is decremented by 1.

Step 32 decides (again) whether or not a valid D-intersection was formed. If it was, then we branch to step 35 to form the new *tc*. If it was not, step 33 tests to see whether or not the D-intersection failed because of a D and Ð both being present in $tc_p$. If a D-, Ð-combination does not occur, we return to step 13 to resume the examination of successors. When both a D and Ð are present, step 33.1 tests to see if the block output has previously been set to the incorrect value: if so, it is necessary to back up, thus a branch to step 6 is executed. Step 33.2 tests to see if the output is x. If it is, step 34 is used to set the output. If the output is not an x, it must already have been set correctly; the program branches to step 13.

Now what we want of new *tc* is that its old coordinates shall be those of old *tc* while its new shall be those associated with block $s$, namely either its input coordinate $p$ or its output $s$. Thus the expression within the left oblique bracket \;;\ (which Iverson calls *mesh*) contains a vector $\epsilon^{p,s}$. If the $r^{th}$ coordinate of this vector (or proposition) is 1, then the corresponding coordinate of *tc* is that of $w$. If it is 0, then the corresponding coordinate of *tc* is that of old *tc*, which is obtained by restricting *tc* to the subset $\bar{\epsilon}^{p,s}$, *not* in the inputs or outputs of the block $s$. This is what $\bar{\epsilon}^{p,s}/tc$ specifies.

Step 36 adjoins $s$ to $a$. In step 37, necessary modifications for $fa$ are made (Cf. step 3).

Step 38 determines whether or not the "D-drive" to the output has been completed, i.e., whether all entries in $a$ are primary outputs which have been driven forward as far as possible. If not, the action returns to step 13, to resume the examination of successors. If it has, then we have pursued the D's to primary outputs and we have finished with the first part of the algorithm.

The only instructions in the first part of the algorithm yet to be explained are steps 5 through 8. These are reached through the backup procedure in branching. Step 5 is accessed from step 10 when all elements in $a$ have been examined. Step 5 ascertains whether or not a *po* has been reached. If it has, we branch to step 38. If a *po* has not been reached, we ask in step 6 whether any branching levels remain to be investigated: i.e., is $m = 0$? If not, then no test exists. If so, steps 7 and 8 back up in the branching structure.

The last part of the algorithm, the CONSISTENCY operation, begins with step 39. This step defines a vector $g$ consisting of all lines whose coordinates of *tc* have values 1 or 0. Step 40 asks whether or not $g$ consists solely of primary inputs *pi*. If it does, the algorithm STOPS, for the purpose of the CONSISTENCY operation is consistently to drive back the values on the lines $g$ to primary inputs, to determine the actual test, in terms of primary inputs, to the circuit. Step 41 deletes those coordinates of $g$ which are already primary inputs. Since we will always be working with the last element of $g$, step 42 sets the $g$ index, $n$, to $\nu g$. Step 43 tests the inputs (predecessors) of logic block $g_n$ to see if both a D and Ð occurs in *tc*. If both do occur, this block can be skipped (its output was determined strictly by the D, Ð; cf. step 34) so that step 44 deletes $g_n$ from $g$ and branches back to step 40 where the completion test is

made. If they do not both occur, step 45 is entered. Step 45 sets to 0 the index $l$ on the number of rows of the singular cover ${}^{r}S$ of the $r^{th}$ logical block. As was indicated in Section 3, it is necessary that $S$ consist of prime cubes, that is, of prime implicants. In Step 46 $\mu({}^{r}S)$ is the number of rows of the matrix ${}^{r}S$ and Step 46 tests whether or not all rows of ${}^{r}S$ have been tried (unsuccessfully). If they have, then this test cube $tc$ is finished and control shifts to Step 55 where $m$ is tested to ascertain whether all branches have been exhausted. If $m = 0$, the process is ended and there is no test. If $m \neq 0$, then in Step 56 the parameter $h_m$ is tested. If $h_m = 1$ then the next branch is in the CONSISTENCY mode and Steps 57 and 58 prepare for a return to Step 46. But if $h_m = 0$ in Step 56 control shifts to Step 7 for commencing the first part of the algorithm. But we return to Step 46. If $l$ is less than $\mu({}^{r}S)$ then more D-intersections can be formed: Step 47 increments $l$ and Step 48 forms the coordinate D-intersection. Step 49 is a test to ascertain whether or not a nontrivial D-intersection is formable. If not, control returns to Step 46. If so, then Steps 50, 51, 52 ascertain whether to save this stage of the computation: if $l < \mu({}^{r}S)$ then it is saved. In any event in step 53, $g_n$ is deleted from $g$ and those coordinates of ${}^{r}S$, whose value is 0 or 1 and for which $tc = x$, are added to $g$. Step 54 respecifies $tc$ by $y$. Finally control is shifted to Step 40 where the test for completion is made: is $g$ contained in the set $pi$ of primary inputs? With this, then, the description of the program is complete.

## 5. Proof of validity of algorithm

• *Definition*: Let $\eta = \eta^1, \eta^2, \cdots, \eta^r$ be a set of D-cubes. Let $\vartheta(\eta)$ be defined as their D-intersection

$$\vartheta(\eta) = \eta^1 \cap \eta^2 \cap \cdots \cap \eta^r.$$

If $\eta$ is the empty set $\phi$, let $\vartheta(\eta) = xx \cdots x$, of dimension $vG$.

For notational convenience let $c(T, F)$ be denoted $c$. As in Section 4 we restrict ourselves to the functions AND, OR, NAND, and NOR for the logic blocks $\lambda$.

• *A-construction*: Let $I$ be a logic block not the site of the failure $F$, for which the output coordinate $c_i$ of $c$ is D or $\bar{D}$. We define a D-cube $\alpha^I$ in the following manner: for the immediate inputs $p$ of $I$ and output $I$, let the coordinates of $\alpha^I$ coincide with those of $c$; let the coordinates of $\alpha^I$ elsewhere be x. The proof of Lemma 6 establishes that $\alpha^I$ is a primitive D-cube of block $I$.

Let $\alpha$ denote the set $\alpha^{I_1}, \cdots, \alpha^{I_m}$ of all such primitive D-cubes, one for each block $I$, which has a D or $\bar{D}$ in its output coordinate and is not the site of failure $F$.

• *B-construction*: Suppose that for logic block $J$ the output coordinate of $c$ is 1 (or 0) and both D and $\bar{D}$ do not occur on the inputs. Then $c$ can be used to form a singular cube $\beta^{J*}$ according to the following construction. Let $r$ be an output or input coordinate of block $J$: if $c(r) =$ 1 (or 0), let $\beta^{J*}(r) = 1$ (or 0); if $c(r) =$ D, $\bar{D}$ or x, let $\beta^{J*}(r) =$ x; let all other coordinates of $\beta^{J*}$ be x.

*Lemma 7*: (1) The cube $\beta^{J*}$ is contained in a singular cube $\beta^J$ of ${}^{J}S$. (2) Furthermore, $\beta^J$ D-contains $c(T, F)$.

*Proof*: We shall prove part (1) of the lemma for $\lambda$, being an AND. Similar proofs establish the lemma for the other three possibilities.

*Case I*: The output of block $J$ is a 1. This implies that all inputs are 1 and the lemma immediately follows.

*Case II*: The output of block $J$ is a 0. Now the only ways in which it is possible in the D-algorithm for this output to be 0 is that either (a) a D-, $\bar{D}$-combination occurs, which is ruled out by hypothesis or (b) some input $z$ of $J$ is 0. Clearly $\beta^{J*}$ is contained in the prime cube $\beta^J$ of ${}^{J}S$, consisting of a 0 in lines $z$ and $J$, and an x in all other coordinats.

Part (2) of the lemma follows immediately from the construction. *Q.E.D.*

Let $\beta$ denote the set $\beta^{J_1}, \cdots, \beta^{J_n}$ of such constructed singular prime cubes, one for each block $J$ which has in $c$ a 1 or 0 in its output and whose input does not contain both D and $\bar{D}$.

• *T-construction*: Suppose that for logic block $K$ its output coordinate in $c(T, F)$ is 1 (or 0) and that a D and $\bar{D}$ occur as input coordinates. In this case we define cube $\gamma^K$ consisting of a 1 (or 0) in coordinate $K$ and x's elsewhere. Let $\gamma = \gamma^{K_1}, \cdots, \gamma^{K_t}$ be the set of all such $\gamma^K$'s.

First we state

*Hypothesis II*: Given a Boolean graph $G$, a failure $F$ in $G$, and the existence of a test $T$ to detect $F$ in $G$.

*Theorem 1*: Under Hypothesis *II*, the following conclusions hold:

(1) The D-cube of test-and-failure $c(T, F)$ (as defined in Section 2) can be constructed.

(2) This D-cube $c(T, F)$ conversely defines $T$ as its primary input coordinates $pi$.

(3) There exists a primitive D-cube-of-failure $\alpha^0$ which D-contains $c(T, F)$

$$\alpha^0 \supset c(T, F).$$

(4.0) There exists a set $\alpha = \alpha^{I_1}, \alpha^{I_2}, \cdots, \alpha^{I_m}$ of primitive D-cubes of a set of logic blocks in $G$ such that their D-intersection $\vartheta(\alpha)$ D-contains $c(T, F)$,

$$\vartheta(\alpha) \supset c(T, F).$$

(4.1) There exists a set $\beta$ of prime singular cubes $\beta^{J_1}, \beta^{J_2}, \cdots, \beta^{J_n}$ of $G$ such that $\vartheta(\beta)$ D-contains $c(T, F)$,

$$\vartheta(\beta) \supset c(T, F).$$

(5) Let $c^* = \alpha^0 \cap \partial(\alpha) \cap \partial(\beta) \cap \partial(\gamma)$. Then $c^*$ D-contains $c(T, F)$.

$c^* \supset c(T, F)$.

and $c^*$ defines a set $T^*$ of tests containing $T$.

*Proof of Theorem 1*

(1) The construction of $c(T, F)$ requires that, for each line of $G$ in both the good and failing circuit, a unique signal value occurs. This is ensured by the hypothesis that $G$ is a Boolean graph.

(2) By definition, $T$ is completely specified by the values of the coordinate positions of the primary inputs of $c(T, F)$.

(3) This is a restatement of Lemma 4, with $\alpha^0$ being the primitive-D-cube-of-failure.

(4.0) By Lemma 6 for each logic block I, for which an output coordinate of $c(T, F)$ is D or $\bar{D}$, there is a primitive D-cube of failure $\alpha' \supset c(T, F)$. That $\partial(\alpha) \supset c(T, F)$ follows directly from Lemmas 3 and 5.

(4.1) By Lemma 7 for each logic block $J$ for which the output coordinate of $c(T, F)$ is a 1 or 0 and for which a D–$\bar{D}$-combination does not appear on its inputs, a singular cube $\beta^J$ is constructed, each D-containing $c(T, F)$. Let there be $n$ of these blocks $J$, and thus $n$ $\beta$'s: $\beta^{J_1}, \cdots, \beta^{J_n}$. By Lemmas 3 and 5, $\partial(\beta) = \beta^1 \cap \beta^2 \cap \cdots \cap \beta^n \supset c(T, F)$.

(5) That $c^* \supset c(T, F)$ follows directly from the A-, B- and I'-constructions and from Lemma 5. Now since $c^* \supset c(T, F) \equiv c$, the only way in which $c^*$ differs, if at all, from $c$ in *any* coordinate $r$ is for $c^*_r$ to be x while $c_r$ is 1 or 0. Thus, denoting $c^*_{pi}$ as $T^*$ and $c_{pi}$ as $T$, we have that $T^* \supset T$. *Q.E.D.*

*Theorem 2*: If for a given failure $F$ of the circuit there exists a test $T$ to detect that failure, then the D-algorithm will compute a test cube $c(T', F)$ for some test $T'$.

*Proof of Theorem 2.*

Assume that, in the execution of the D-algorithm, as described in Fig. 5, no test cube $tc \equiv c(T', F)$ has been constructed before the test cube $c(T^*, F)$ has been encountered, for if one such has been, then the theorem is already proven. We shall demonstrate, under this assumption, that $c(T^*, F)$ is in fact generated by the D-algorithm.

Assume that the set $\eta = \alpha \cup \gamma$ has been arranged in ascending order with respect to superscripts, logic block numbers, i.e., $\eta = \eta^{L_1}, \cdots, \eta^{L_{m+q}}$ where $L_i < L_{i+1}$. Let $\beta$ be similarly ordered.

The following terms will be convenient in the exposition. Block $w$ is said to be a D-*successor* of block $v$ if and only if block $w$ has a D or $\bar{D}$ in the coordinate position corresponding to block $v$. We shall dually describe this relationship by saying that block $v$ is a D-*predecessor* of block $w$.

A sequence of sets $H^v$ (composed of elements from $\alpha^0$, $\eta$, and $\beta$) and corresponding test cubes $tc^v$, $v = 0, 1, \cdots, n + m + q$, will now be constructed, recursively, in such a manner that the $tc^v$ will correspond directly to a sequence of test cubes generated by the algorithm in Fig. 5.

Let $H^0 = \alpha^0$ and $tc^0 = \partial(H^0) = \alpha^0$. Having defined the set $H^v$ and test cube $tc^v$ we proceed to define $H^{v+1}$ and $tc^{v+1}$ from $\eta^v$, the $v^{th}$ entry in $H^v$, as follows.

*Case 1*: Suppose $\eta^v$ is a $\gamma$. Then

$H^{v+1} = H^v$

$tc^{v+1} = tc^v$;

i.e., in the algorithm the successors of blocks arising from $\gamma$'s are never examined.

*Case 2*: Suppose $\eta^v$ is an $\alpha$. Then

$H^{v+1} = H^v \cup \Sigma(\eta^v)$

$tc^{v+1} = tc^v \cap \partial[\Sigma(\eta^v)]$,

where $\eta^w \in \Sigma(\eta^v)$ if and only if: (a) $\eta^w$ is not an element of $H^v$; (b) $\eta^w$ is a D-successor of $\eta^v$; and (c) one of the following two conditions holds:

Condition 1. Suppose $\eta^w$ is an $\alpha$. Then all D-predecessors of $w$ must have their corresponding primitive D-cubes of failure, $\alpha$, in $H^v$.

Condition 2. Suppose $\eta^w$ is a $\gamma$. Then there must be at least one D-predecessor of block $w$ whose primitive D-cube of failure is in $H^v$ and has value *opposite* to that of $\eta^v$ in its output coordinate.

After $v^* = m + q$ iterations, $H^v$ contains all the elements of $\alpha_0$, $\alpha$, and $\gamma$. This corresponds to the completion of the D-drive: the $\eta^v$ do not have any new D-successors, and the primary outputs $po$ must therefore have been reached.

The final $n$ iterations in the formation of $H^v$ and $tc^v$, where $v^* < v \leq v^* + n$, are carried out as follows. Pick the highest coordinate position, hence a logic block, in $tc^v$ which is a 1 or 0, which is not a primary input $pi$, and which has not already been examined (Cf. Steps 41 and 42 of Fig. 5). If there is both a D and $\bar{D}$ on the inputs of this block skip the coordinate. If not, then the B-construction generated a $\beta^w$ for this block. Now form

$H^{v+1} = H^v \cup \beta^w$

$tc^{v+1} = tc^v \cap \beta^w$.

After $n$ iterations, this final intersection process must terminate.

We can now make the following assertions:

1. Theorem 1 proves that $c^* \equiv \alpha^0 \cap \partial(\alpha) \cap \partial(\beta) \cap \partial(\gamma)$ defines a set of tests $T^*$ containing $T$.
2. The above construction follows precisely the steps of (algorithm) i.e. the construction forms

$\alpha^0 \cap \partial(\alpha) \cap \partial(\beta) \cap \partial(\gamma) = c^*$ Q.E.D.

the D-algorithm of Fig. 5 in generating the test cube $tc^{\eta^1\eta^2\eta^3}$: the order of examination of $\eta^i$ and formation of $\sum(\eta^i)$ corresponds precisely to that specified by Steps 5 through 38 and the order of intersection of the $\beta$'s is specified by Steps 39 through 58.

Since intersection is commutative (Lemma 3) we have

$$tc^{\eta^1\eta^2\eta^3} = c^*.$$

Thus the algorithm generates a test $T^*$ which contains $T$.

*Q.E.D.*


## Acknowledgement

The author is most grateful for the manifold contributions of P. R. Schneider to the final formulation of this paper; specifically, the structure and proof of Theorems 1 and 2 are the joint work of JPR and PRS. Further, the final version of the programmed algorithm (Figure 5 and Section 4) was a joint undertaking of JPR, PRS, and W. G. Bouricius. The author is much indebted to E. G. Wagner who read critically an earlier version of the paper and who continued to make penetrating observations. The author is also endebted to K. E. Iverson and Adin Falkoff who, at a still earlier stage, helped him in writing his first version of the Iverson-notation program. Finally, thanks are due to B. O. Evans, then Vice President of the IBM Data Systems Division, who first urged the preparation of such a paper for the IBM Journal.


## References


1a. W. C. Carter, H. C. Montgomery, R. J. Preiss, and H. J. Reinheimer, "Design of Serviceability Features for the IBM System/360," *IBM Journal* 8, 2 (1964).

1b. K. Maling and F. L. Allen, Jr., "A Computer Organization and Programming System for Automated Maintenance," *IEEE Trans. Elect. Comp.* EC-12, No. 6, 887-895 (December, 1963).

2. J. P. Roth and R. M. Karp, *Minimization over Boolean Graphs*, *IBM Journal* 6, 2 (1962).

3. R. E. Miller, *Switching Theory*, *Vol. I*, John Wiley and Sons, New York, 1965, p. 233.

4. J. P. Roth, "Algebraic Topological Methods in Synthesis," in *Proceedings of an International Symposium on the Theory of Switching, 2-5 April* 1957, *Part I:* The Annals of the Computation Laboratory of Harvard University, Vol. XXIX, pp. 57-73, Harvard University Press, Cambridge, Mass., 1959.

5. J. M. Galey, R. E. Norby, and J. P. Roth, "Techniques for the Diagnosis of Switching Circuit Failures," *Transactions of the IEEE Communications and Electronics* 83, 74, Sept. 1964, pp. 509-514.

6. C. B. Stieglitz, Paper in preparation.

7. J. P. Roth, "A Pragmatic Theory of Automata," Lecture given at *A Symposium on Switching Theory and Automata*, International Federation of Automatic Control, Moscow, 1962. Cf. also IBM Data Systems Division Report TR00.918, Sept. 21, 1962, Poughkeepsie, N. Y.

8. J. P. Roth, "Algebraic Topological Methods for the Synthesis of Switching Systems, I.", *Transactions of the American Mathematical Society* 88, 2, 301-326, July 1958. Cf. also Institute for Advanced Study, Princeton, N. J., ECP 56-02, April 1956 and General Electric Company Report No. R55GL345, Schenectady, N. Y., Sept. 1955.

9. K. E. Iverson, *A Programming Language*, John Wiley & Sons, New York, 1963.

10. A. D. Falkoff, K. E. Iverson and E. H. Sussenguth, "A Formal Description of System 360," *IBM Systems Journal* 3, 3, 1964, pp. 198-263.



*Received October 15, 1964*
*Revised Manuscript received March 14, 1966.*

BELL TELEPHONE LABORATORIES
INCORPORATED

SUBJECT: No. 1 ESS - Minimization of Fault Detection Tests - An Approximate Method - Case 36279-113,120

DATE: July 12, 1966

FROM: S. A. Szygenda
MF14-660712.1

## Introduction

A diagnostic test for a logic circuit can be designed by assuming the existence of some single fault, then determining a suitable test to detect the fault. If this process is repeated for the various faults, a set of tests are generated which constitutes a diagnosis of the circuit. If a given circuit may contain x faults and a separate test is generated to detect each of these faults, then a total of x diagnostic tests are involved in the total diagnosis of the circuit. Actually, most of the x tests can detect more than one fault, and this property can be enhanced by proper selection of the states of "don't care" conditions. Thus, several tests can be combined to form one test which detects every fault detected by the constituent tests. The problem of properly selecting and combining tests from a given (large) set to produce a new (much reduced) set of tests, capable of detecting all the faults detected by the original tests, is an important one. This paper describes an automated method for performing this test reduction.

A technique presently exists for automatically generating tests for faults in circuits containing flip-flops, NAND, NOT and AND gates, but no feedback loops.[1] This technique is capable of generating all possible tests for a set of faults. That is, for fault i, q different tests may be generated ($t_{i1}, t_{i2}, \ldots, t_{iq}$) each of which is capable of detecting fault i. Thus, this test generation procedure generates many more tests than faults. It is advantageous to consider all of these generated tests when attempting to minimize, since the additional flexibility theoretically allows a more extensive degree of reduction. However, computer time and memory limitations severely handicaps our ability to operate on an arbitrarily large number of input tests.

The problem of selecting a minimum subset of fault detection tests has repeatedly plagued computer diagnosticians. Although there have been several proposed solutions to this problem, they become extremely difficult to implement when large numbers of tests are involved. The difficulty with these approaches is that they attempt to obtain an absolute minimal subset of tests and, in most cases, this becomes a problem of attempting to solve a very large prime implicant table. As far as we have been able to determine, perfect minimization is not a practical objective for large numbers of tests. Instead, we assume that it is acceptable to obtain less than perfect minimization, as long as significant reduction in the number of

tests is realized, and if the method for doing this lends itself easily to programming and has the facility for handling large numbers of tests.

The automated approach to be described here is quite simple to program, and consequently, is acceptable from that point of view. A discussion of a computer program to implement this approach will be forthcoming.

This minimization technique has been utilized in some recent attempts to automatically generate diagnostic tests for the central control of the No. 1 Electronic Switching System.[2] Preliminary results of these attempts indicate that the minimization technique is efficient and capable of generating "near" minimal sets of tests.

Appendix A includes a method which can achieve an even greater minimization if applied in conjunction with the existing procedure. This method has not been implemented, as yet, because the amount of further minimization that may be accomplished has not been determined and may not justify the amount of effort required.

Description of Algorithm I

If a fault is introduced into a circuit, we may find different combinations of inputs which can detect this fault. These inputs may assume the values 1, 0, or "d" where d = don't care. A particular combination of variables which can detect a given fault, $F_1$, is a test, $t^1$, for the fault. If we assume

that $q_i$ distinct tests are available to detect fault $F_i$, then we may distinguish one from another by subscripting the symbol $t^i$ appropriately (i.e., $t^i_z$ = zth test for fault $F_i$). If we assume a circuit having m input variables, then an individual test, $t^i_z$, is an ordered set of m input variable values:

$$t^i_z = \left[v^i_{z,1}, \ldots, v^i_{z,m}\right],$$

where $v^i_{z,j}$ can be "0", "1", or "d". The various tests $\left[t^i_1 \ldots t^i_{qi}\right]$ that detect fault $F_i$ comprise a subset of tests, $T_i$, for that fault. These test subsets make up a set, S which contains all of the tests to be considered. Using this symbology, the entire set S may be represented as follows (assuming a total of N faults).

$$T_1 = \begin{cases} t_1^1 = \{v_{1,1}^1, v_{1,2}^1, \ldots, v_{1,m}^1\} \\ t_2^1 = \{v_{2,1}^1, v_{2,2}^1, \ldots, v_{2,m}^1\} \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ t_{q1}^1 = \{v_{q1,1}^1, v_{q1,2}^1, \ldots, v_{q1,m}^1\} \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \quad \cdot \end{cases}$$

$$S = T_j = \begin{cases} t_1^j = \{v_{1,1}^j, v_{1,2}^j, \ldots, v_{1,m}^j\} \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ t_k^j = \{v_{k,1}^j, v_{k,2}^j, \ldots, v_{k,m}^j\} \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ t_{qj}^j = \{v_{qj,1}^j, v_{qj,2}^j, \ldots, v_{qj,m}^j\} \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \quad \cdot \end{cases}$$

$$T_N = \begin{cases} t_1^N = \{v_{1,1}^N, v_{1,2}^N, \ldots, v_{1,m}^N\} \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \quad \cdot \\ t_{qN}^N = \{v_{qN,1}^N, v_{qN,2}^N, \ldots, v_{qN,m}^N\} \end{cases}$$

Therefore,

$$v^j_{k,x} \varepsilon t^j_k \varepsilon T_j \varepsilon S \quad \text{where} \quad \text{"}\varepsilon\text{"} \approx \text{"subset of."}$$

Throughout this description we will use the symbol "$\frown$" to represent the operation of intersecting sets. We will make use of the following axioms.

(1) $\{0\} \frown \{0\} = 0$

(2) $\{1\} \frown \{0\} = \Phi$ ($\Phi$ = the null set)

(3) $\{0\} \frown \{1\} = \Phi$

(4) $\{1\} \frown \{1\} = 1$

(5) $\{d\} \frown \{0\} = 0$ (assuming d is assigned the value 0)

(6) $\{d\} \frown \{1\} = 1$ (assuming d is assigned the value 1)

(7) $\{d\} \frown \{d\} = d$ (assuming all d's have the same value)

(8) $\{\Phi\} \frown \{d\} = \Phi$

For purposes of minimization, we conceive of a test as a matrix, or <u>ordered set</u> of elements, each element representing the <u>value</u> of a particular input variable. Thus, our concept of test intersection is based on the intersections of corresponding elements of the test. Since such an ordered intersection is an operation different from the set intersection operator "$\frown$", we will employ a special symbol: <u>"$\frown$"</u> for test intersection. This operator functions as follows:

$$\left\{t^A_x \frown t^B_y\right\} = \left\{\left[v^A_{x,1} \frown v^B_{y,1}\right], \left[v^A_{x,2} \frown v^B_{y,2}\right], \ldots, \left[v^A_{x,m} \frown v^B_{y,m}\right]\right\}.$$

Therefore, if

$$t_A = \{1,0,d\}$$

and

$$t_B = \{1,d,0\},$$

then

$$\{t_A \dot{\cap} t_B\} = \{1,0,0\},$$

whereas

$$\{t_A \dot{\cap} t_B\} = \{1,0,d\}.$$

One further definition: any test containing one or more $\Phi$'s as elements is said to be an empty test, $t_\Phi$.

The minimization procedure begins by taking the intersection:

$$t_1^1 \dot{\cap} t_1^2 \quad \text{where}$$

$$t_1^1 \in T_1 \quad \text{and} \quad t_1^2 \in T_2.$$

If <u>any</u> element of $t_1^1 \dot{\cap} t_1^2 = \Phi$, then the tests $t_1$ and $t_2$ are disjoint, and the intersection is $t_\Phi$, the empty test. In this case we continue by attempting to find a nondisjoint pair of tests, that is, we take $t_1^1 \dot{\cap} t_2^2$. Generalizing, we take intersections of the form $t_1^1 \dot{\cap} t_h^2$ for (h = 1,2,...,q2), looking

for a nonempty intersection. Temporarily assume that all intersections produced in this way are empty. That is:

(9) $$t_1^1 \cap t_h^2 = t_\Phi \quad \text{for} \quad h = 1,2,\ldots,q2.$$

For convenience, we symbolically express this situation as:

$$t_1^1 \cap T_2 = \Phi.$$

If condition (9) holds, we proceed with our search for a nondisjoint intersection, by forming $t_2^1 \cap t_h^2$ for various h. If we assume that this operation leads to

$$t_2^1 \cap T_2 = \Phi,$$

and we continue in this manner, we may eventually find that

$$t_g^1 \cap t_h^2 = \Phi$$

for all allowable values of g, h.

We would symbolically represent such a situation as:

(10) $$T_1 \cap T_2 = \Phi.$$

If it is the case that $T_1 \cap T_2 = \Phi$, then there is no nondisjoint intersection between pairs of tests taken from the test subsets $T_1$ and $T_2$. In this situation we continue by looking for possible nondisjoint intersections between tests of $T_1$ and the other test subsets contained in S.

If we find that $T_1 \cap T_f = \Phi$ for all $f$ $(2 \leq f \leq N)$, we can conclude that there is no nondisjoint intersection between any test of the subset $T_1$ and any test of any other subset contained in S. Therefore, we must keep one of the tests from the subset $T_1$ if we wish to have the ability to detect fault $F_1$. The selection of a $T_1$ test to be kept may be based on some criterion such as ease of generating the variables required by the test.

Now consider the case where

$$t_g^1 \cap t_h^2 = t_{gh} \neq t_\Phi,$$

for some values of g, h. In this situation a new test subset, $T_{1,2}$, is formed, each test of which is a $t_{gh} \neq t_\Phi$ for particular values of g and h. Note that $T_{1,2}$ may contain as many as $(q1)\cdot(q2)$ tests. Since $T_{1,2}$ represents the nondisjoint intersections of tests in $T_1$ with tests in $T_2$, we will symbolically write

$$T_{1,2} = T_1 \cap T_2.$$

After $T_{1,2} \neq \Phi$ has been created, we ignore the original test subsets $T_1$ and $T_2$, since any test contained in $T_{1,2}$ can detect both faults $F_1$ and $F_2$. Now we search for other nondisjoint intersections by examining $T_{1,2} \cap T_3$. If $T_{1,2} \cap T_3 = \Phi$, we retain $T_3$ and proceed to examine $T_{1,2} \cap T_4$. However, if

$T_{1,2} \cap T_3 = T_{1,2,3} \neq \Phi$ then $T_{1,2}$ and $T_3$ are replaced by $T_{1,2,3}$. Continuing in this manner we eventually obtain a "combination test subset" of the form $T_{a1,a2,\ldots,ax} = T_{c1}$, and a number of original (primary) test subsets $(T_{p1}, T_{p2}, \ldots, T_{py})$ each of which have the property that $T_{pj} \cap T_{c1} = \Phi$.

We now divide the test subsets $(T_{p1}, T_{p2}, \ldots T_{py})$ into two parts, such that those test subsets $T_{pi}$ where $pi < a1$ are taken out of the set $(T_{p1}, T_{p2}, \ldots T_{py})$. This gives us

$$T_{p1}, T_{p2}, \ldots T_{pi}, T_{c1}, (T_{pi+1}, T_{pi+2}, \ldots T_{py}) .$$

This division is made to avoid repeated searching for nondisjoint intersections between $T_{p1}, T_{p2}, \ldots T_{pi} = D^1$ and the test subsets $T_{pi+1}, T_{pi+2}, \ldots, T_{py}$; we have already determined that there are no disjoint intersections between these subsets of tests.

We now define a new "universal" set, $S^1$, consisting of test subsets $(T_{pi+1}, T_{pi+2}, \ldots, T_{py})$ and apply the same technique as was applied to set S. This results in a combination test subset, $T_{c2}$, a new set of disjoint subsets $D^2$ and a new set $S^2$, consisting of the remaining subsets of tests. We continue in this manner until all intersections have been taken. Essentially, what is happening is that the sets of tests $S, S^1, S^2, \ldots, S^{ri}$ get increasingly smaller, since the test subset for a particular fault is either put into set D (if there are no nondisjoint intersections between that test subset and any of the others), or into $T_{ci}$ (if nondisjoint intersections exist).

The final (minimized) set of tests required for detecting all faults of the original tests may now be created by selecting one test from each of the subsets of tests contained in $D_1, D_2, \ldots, D_u$ and one from each of the subsets $T_{c1}, T_{c2}, \ldots, T_{cw}$.

The following example can serve as a demonstration of the minimization algorithm.

$$\begin{array}{rrl} & & \begin{matrix} v_1 & v_2 & v_3 & v_4 & v_5 \end{matrix} \\ & t_1^1 = & ( \; 1 \quad 1 \quad 0 \quad 1 \quad d \; ) \\ T_1 = & t_2^1 = & ( \; 1 \quad 1 \quad 0 \quad d \quad 0 \; ) \\ & t_3^1 = & ( \; 1 \quad 0 \quad d \quad d \quad 1 \; ) \\ & & \\ & t_1^2 = & ( \; d \quad 0 \quad 1 \quad 1 \quad d \; ) \\ \text{Let } S = \quad T_2 = & t_2^2 = & ( \; 0 \quad 0 \quad 0 \quad d \quad 1 \; ) \\ & t_3^2 = & ( \; d \quad 1 \quad d \quad 1 \quad 1 \; ) \\ & & \\ & t_1^3 = & ( \; d \quad 1 \quad d \quad 0 \quad 0 \; ) \\ T_3 = & t_2^3 = & ( \; 0 \quad d \quad d \quad 0 \quad d \; ) \\ & t_3^3 = & ( \; 1 \quad 0 \quad d \quad 1 \quad 0 \; ) \\ & & \\ T_4 = & t_1^4 = & ( \; 1 \quad d \quad 0 \quad 1 \quad d \; ) \end{array}$$

$$T_1 \cap T_2 = \begin{bmatrix} t_1^1 \cap t_1^2 = t_\Phi \\ t_1^1 \cap t_2^2 = t_\Phi \\ t_1^1 \cap t_3^2 = (1\ 1\ 0\ 1\ 1) \\ t_2^1 \cap t_1^2 = t_\Phi \\ t_2^1 \cap t_2^2 = t_\Phi \\ t_2^1 \cap t_3^2 = t_\Phi \\ t_3^1 \cap t_1^2 = (1\ 0\ 1\ 1\ 1) \\ t_3^1 \cap t_2^2 = t_\Phi \\ t_3^1 \cap t_3^2 = t_\Phi \end{bmatrix}$$

We see that $T_1 \cap T_2 \neq \Phi$ and we proceed to form $T_{12}$:

$$T_{12} = \begin{matrix} t_1^{12} = (\ 1\ 1\ 0\ 1\ 1\ ) = \lambda_1 \\ t_1^{12} = (\ 1\ 0\ 1\ 1\ 1\ ) = \lambda_2 \end{matrix}$$

These tests are common to faults $F_1$ and $F_2$, so we no longer consider the original sets $T_1$ and $T_2$. We continue by taking

$$T_{12} \cap T_3 = \begin{bmatrix} \lambda_1 \cap t_1^3 = \Phi \\ \lambda_1 \cap t_2^3 = \Phi \\ \lambda_1 \cap t_3^3 = \Phi \\ \lambda_2 \cap t_1^3 = \Phi \\ \lambda_2 \cap t_2^3 = \Phi \\ \lambda_2 \cap t_3^3 = \Phi \end{bmatrix}$$

Therefore, $T_{12} \cap T_3 = \Phi$, and $T_3$ is retained for later reduction. We now examine $T_{12} \cap T_4$, and find

$$T_{12} \cap T_4 = \begin{bmatrix} \lambda_1 \cap t_1^4 = 1\ 1\ 0\ 1\ 1 \\ \lambda_2 \cap t_1^4 = \Phi \end{bmatrix}$$

Thus, we eliminate $T_4$, and form $T_{124} = \{\lambda_1 = (11011)\}$.

Since we have no further test subsets to examine, we have:

$$T_{124} = T_{c1} = \{11011\}$$

and

$$T_3 = T_{p1} = \begin{Bmatrix} d1d00 \\ 0dd0d \\ 10d10 \end{Bmatrix}$$

There are no further test subsets with which $T_3$ can be intersected. Thus, we have reduced to two the number of tests required to detect the four faults:

Test 1: {11011}

Test 2: any of the tests from $T_3$

Conclusion

The set of tests which are finally created is obviously dependent upon the order in which tests are selected for

examination,* and is not necessarily minimal. In practice this procedure has given reductions of several factors in numbers of tests required for logic circuit diagnosis. This success, in addition to the ease with which the process may be programmed, leads us to conclude that this procedure is satisfactory for many practical applications.

S. A. SZYGENDA

---

* In the above description, we started out by using test subset $T_1$, this was mainly for ease in describing the procedure. We could have started with any subset $T_i$. Although the order in which the subsets are chosen for intersection may vary the amount of reduction that can be achieved, it appears that this variance is often small for large numbers of tests. A discussion of a method which could be used to eliminate this variance is given in Appendix A.

## REFERENCES


1. D. B. Armstrong; "On Fault Detection in Large Combinational Logic Circuits"; MM-64-1271-11.
2. S. A. Szygenda; "Programs to Edit and Minimize Fault Detection Tests"; Report in Preparation.

## APPENDIX A

Some variability in the size of the "near minimal" test set has been observed. This is due to the possibility of arranging the original set of tests in different orderings before minimization. The following description provides a procedure which can be used to judiciously order the tests before application of Algorithm I.

For this method the axioms given on page 6 must be modified as follows:

Axiom (5) $\{d\} \cap \{0\} = 0$ becomes

(5a) $\{d\} \cap \{0\} = \Phi$

Axiom (6) $\{d\} \cap \{1\} = 1$ becomes

(6a) $\{d\} \cap \{1\} = \Phi$

all other axioms remain the same. We have stated that our subset of tests $T_j$ contains the tests $t_k^j$, that is

$$T_j = t_1^j, t_2^j, \ldots, t_{qj}^j$$

We will set up a table T and label each column with the couple $(j,k)$, $j = 1,2,\ldots,N$, $k = 1,2,\ldots,qj$, which locates each test in the set S.

Consider the intersection, as defined by equations $(1),\ldots,(4),(5a),(6a),(7),(8)$, between the tests $t_1^1$ and tests in subsets $T_2, T_3, \ldots, T_N$. That is, form the intersections:

$t_1^1 \cap t_1^2$, $t_1^1 \cap t_2^2$, ..., $t_1^1 \cap t_{q2}^2$; $t_1^1 \cap t_1^3$, ..., $t_1^1 \cap t_{q3}^3$; ...;

$t_1^1 \cap t_1^N, \ldots, t_1^1 \cap t_{qN}^N$. For each such intersection which is nondisjoint, enter the intersecting test in the column of the table headed 1,1. Similarly, take $t_2^1 \cap T_j$ for $j = 2,\ldots,N$, and enter nondisjoint test intersections in column 1,2. Continue in this fashion until all the tests specified by the headings of each column of table T have been intersected with all other tests in other test subsets of S.

The changes in the axioms (5) and (6) have permitted us to find intersections without forcing any of the don't cares in those tests which head each column. We now proceed to reorder the tests for subsequent processing by Algorithm I. We desire that Algorithm I use first a test which creates many nondisjoint intersections with tests in other test subsets. The strategy used to form a desirable ordering of tests is based upon the contents of various columns of table T. We proceed as follows. Take the last entry in the column of table T containing the least number of entries. If there is more than one column containing equal numbers of entries, this number being less than any of the other columns, arbitrarily select one of these columns. Take this test identification entry and search the table to determine if this entry exists in any of the other columns, if so remove the entry from the selected (shortest) column. Continue the above process, selecting "shortest columns," and removing entries that appear elsewhere from these columns, until no further deletions can be made. The

column heading is considered as a column entry after all other entries in that column have been eliminated. That is, we may have a condition where all entries in a column have been checked and it has not been possible to remove some of these entries. Consequently, we cannot remove the column heading, even if it is contained in another column of greater length. In this case we will remove this entry from all columns except the one it heads.

After this process has been completed the table T will contain each test identification exactly once and they will be contained in those columns having the longest path of intersections. We now select those tests which are identified by the entries heading each column and attempt to find nondisjoint intersections between them, as defined in Algorithm I. If a nondisjoint intersection exists we will form a new column containing all entries which were contained in the original two columns and these will be under the heading of the intersected test. The two original columns will be eliminated. The repeated application of Algorithm I, until all column headings have been intersected, leads to the final reduced set of detection tests.

The following example shows the use of this method as well as its applicability.

$$
\text{Let } S = \begin{array}{l} \\ T_1 = t_1^1 \\ T_2 = t_1^2 \\ T_3 = t_1^3 \\ T_4 = t_1^4 \\ T_5 = t_1^5 \\ T_6 = t_1^6 \\ T_7 = t_1^7 \\ T_8 = t_1^8 \\ T_9 = t_1^9 \end{array} \begin{array}{c} \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{array}{ccc} = ( d & 0 & 0 ) \\ = ( 0 & d & 0 ) \\ = ( d & 1 & 0 ) \\ = ( 1 & 1 & d ) \\ = ( d & 1 & d ) \\ = ( 0 & 1 & d ) \\ = ( 0 & 1 & 0 ) \\ = ( 1 & 1 & 1 ) \\ = ( d & d & 1 ) \end{array} \end{array}
$$

By applying Algorithm I we get the following nondisjoint intersections.

$$
\begin{aligned} T_1 \cap T_2 \neq \Phi &= ( 0\ 0\ 0 ) \\ T_3 \cap T_4 \cap T_5 \neq \Phi &= ( 1\ 1\ 0 ) \\ T_6 \cap T_7 \neq \Phi &= ( 0\ 1\ 0 ) \\ T_8 \cap T_9 \neq \Phi &= ( 1\ 1\ 1 ) \end{aligned}
$$

We see that these four tests are sufficient to cover all tests in S.

Now we consider the table T with headings $t_1^1, t_1^2, \ldots, t_1^9$. Taking intersections, as defined above, we get

$$
\begin{array}{lccccccccc}
T = & t_1^1 & t_1^2 & t_1^3 & t_1^4 & t_1^5 & t_1^6 & t_1^7 & t_1^8 & t_1^9 \\
 & & & t_1^5 & t_1^5 & & t_1^5 & t_1^2 & t_1^4 & \\
 & & & & & & & t_1^3 & t_1^5 & \\
 & & & & & & & t_1^5 & t_1^9 & \\
 & & & & & & & t_1^6 & &
\end{array}
$$

We see that $t_1^1$ is not listed in any of the other columns, consequently, it cannot be removed. $t_1^2$ is included under $t_1^7$, therefore, it can be eliminated as can $t_1^5$ from column 5 and $t_1^9$ from column 9. $t_1^5$ is included under $t_1^7$, therefore, it can be eliminated as can $t_1^3$ in column 3. $t_1^5$ and $t_1^4$ can be eliminated from column 4 because they are covered in columns 7 and 8. $t_1^5$ and $t_1^6$ can be eliminated from column 6 because they are covered in column 7. Finally $t_1^5$ can be eliminated from column 8. The table then takes on the following appearance.

$$
\begin{array}{lccccccccc}
T = & t_1^1 & t_1^2 & t_1^3 & t_1^4 & t_1^5 & t_1^6 & t_1^7 & t_1^8 & t_1^9 \\
 & & & t_1^5 & t_1^5 & & t_1^5 & t_1^2 & t_1^4 & \\
 & & & & & & & t_1^3 & t_1^5 & \\
 & & & & & & & t_1^5 & t_1^9 & \\
 & & & & & & & t_1^6 & &
\end{array}
$$

This gives three remaining columns.

$$
\begin{array}{cc}
t_1^1 & t_1^7 \\
t_1^2 & t_1^4 \\
t_1^3 & t_1^9 \\
t_1^5 & \\
t_1^6 &
\end{array}
$$

We see that $t_1^1 = d\ 0\ 0$

$t_1^7 = 0\ 1\ 0$

$t_1^8 = 1\ 1\ 1$

and these cannot be reduced any further by Algorithm I. Therefore, these three tests can cover all nine of the original tests. This means that if we had chosen a different ordering it would have been possible to reduce the nine tests to three by the use of Algorithm I alone.

Proceedings of the Seventh Annual Allerton Conference on Circuit and Systems Theory

# IMPLEMENTATION AND EXTENSION OF MULTI-DIMENSIONAL PATH SENSITIZING IN A SIMULATION AND DIAGNOSIS SYSTEM

S.A. SZYGENDA and G.C. GOLDBOGEN
Computer Science Department
University of Missouri-Rolla, Rolla, Missouri

## ABSTRACT

This paper considers the problem of diagnostic test generation for large combinational logic nets. The foundations of multi-dimentional path sensitizing, as well as some new implementations and optimization techniques, are considered. Results are given for various test cases which were used for system evaluation.

## I. PRELIMINARY

The system considered in this paper has as its primary objective the generation of diagnostic tests for large combinational logic nets. Combinational test generation has been considered extensively elsewhere[1,2,5]. However, many of the techniques proposed do not possess the capability of generating tests for all non-redundant faults, or are not easily implemented for large nets.

The implementation to be discussed is part of a total simulation and diagnostic test generation system[4,5]. As such, it will be required to perform the following functions:

1) Generate a test for any specified non-redundant fault occuring in a large combinational logic net. Since the combinational test generator will comprise only a portion of the total system, efficiency and speed are considered to be of the utmost importance.
2) Generate tests that can distinguish between two or more given faults.
3) Possess the capability of verifying arbitrary input or output specifications. This may involve considering a specified subset of inputs and/or outputs and then generating complete input-output vectors that are consistent with the original specifications.
4) The combinational test generation algorithm should be usable as a heuristic for sequential test generation.

The present version of the program is restricted to considering single logical faults of the stuck-at-one (s-a-1) or stuck-at-zero (s-a-0) type. This limitation can be easily modified to consider other classes of faults, if necessary.

To accomplish the first function, a multi-dimensional path sensitizing technique has beem implemented[3]. This technique utilizes the d-algorithm, proposed by Roth[3], with modifications to achieve program efficiency and flexibility. The algorithm has the ability of generating a test, if one exists, for any fault in the combinational logic. A brief discussion of the general character of this algorithm will ensue a more detailed discussion of the implementation will be given in Section II.

Assume that a fault, F, exists in the circuit. It is desired to generate a test, T, capable of detecting, at least, the fault under consideration. Then it is necessary to

propagate the fault conditions to primary outputs. This is usually referred to as the forward-trace phase of the procedure.

On the other hand, it is usually necessary to have the test specified in terms of the primary inputs to the circuit. To satisfy this condition it will be necessary to "back-up" the specified output values to determine the critical primary input values. This process is called the back-trace phase of the algorithm.

After a test has been generated, using the test generation algorithm, it is simulated to obtain whatever additional information is inherent in the test. The results of this "ping-pong" strategy between test generation and simulation is monitored by a decision module of the system. This module commands the two sub-systems as needed and keeps track of tasks completed and remaining. The operation of the system, to satisfy the conditions considered above, can best be illustrated by a simple example.

Consider a circuit, C, containing faults $M_1$, $M_2$, $M_3$,..., $M_n$. The failure free circuit will be labeled $M_0$. The objective is to generate a sequence of tests capable of distinguishing the circuits containing faults from $M_0$. The process consists of the following steps:

a) The decision module assumes control and monitors the stack of faults. When the fault list is empty this function is complete.
b) If the fault list is not empty, the top entry in the stack ($M_i$) is considered.
c) The combinational test generator is called upon to generate a test for the specified fault.
d) The generated test ($T_j$) is simulated to determine which other faults may be detectable by the test. These faults are then removed from the fault list.
e) The generated test is added to the test sequence with a pointer to the list of faults detectable by the test. This information is used for partitioning and fault catalog production.

This process is demonstrated by the following diagram:

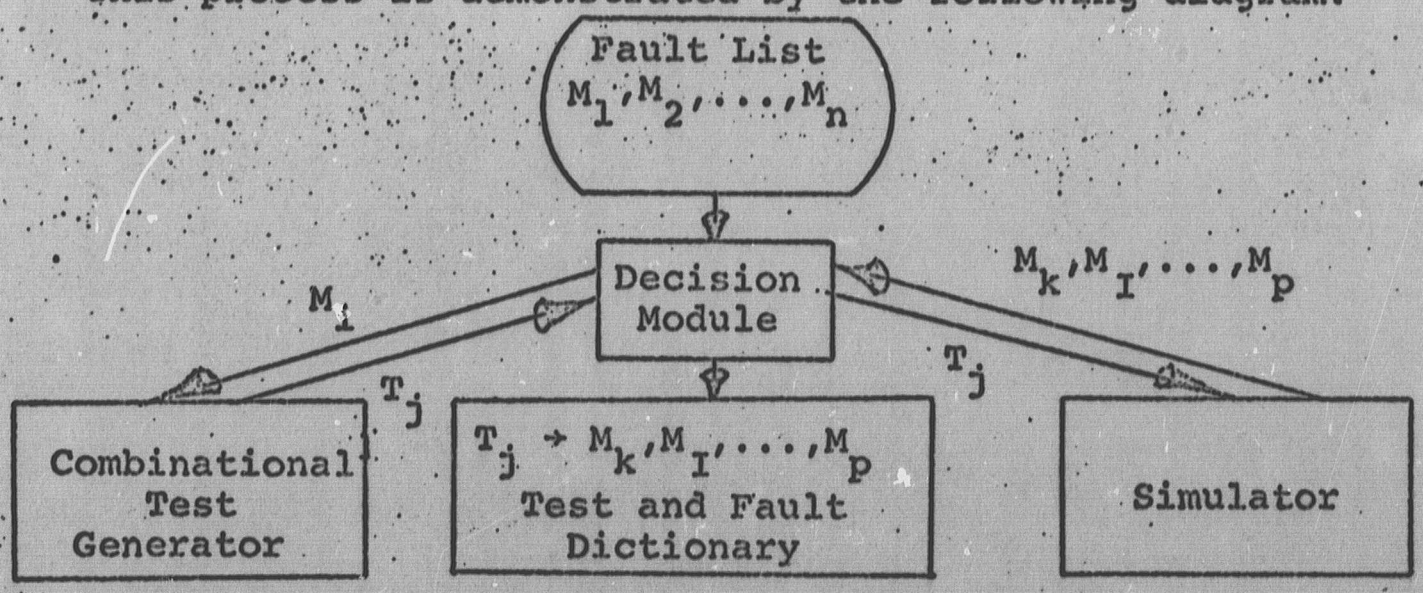



Figure 1. Combinational Test Generation

Before considering the second function, it is noteworthy that the system is implemented in such a manner that the performance of the various functions differs only in the initialization step of the process. The second function extends the

path sensitizing technique. This extension provides the ability of generating a test (if a test exists) capable of distinguishing between two or more faults. This algorithm utilizes the physical relation of the faults and the various connecting paths between the faults, and/or the propagation of the effects of the faults. When a path is chosen, a starting point is determined and the multi-dimensional path sensitizing technique is used to complete the test.

The next function involves compatability rather than test generation. Consider the possibility that a partial list of primary inputs or outputs are specified for the combinational circuit. This can occur if certain inputs are constrained to a logical zero or one value to conform to overall system operational requirements. The algorithm will take these subsets of values and generate a complete input vector that is compatible with the specified inputs and outputs.

The following illustration will demonstrate the usefulness of this function.

Consider the system described in Figure 2, with faults in module B being considered.

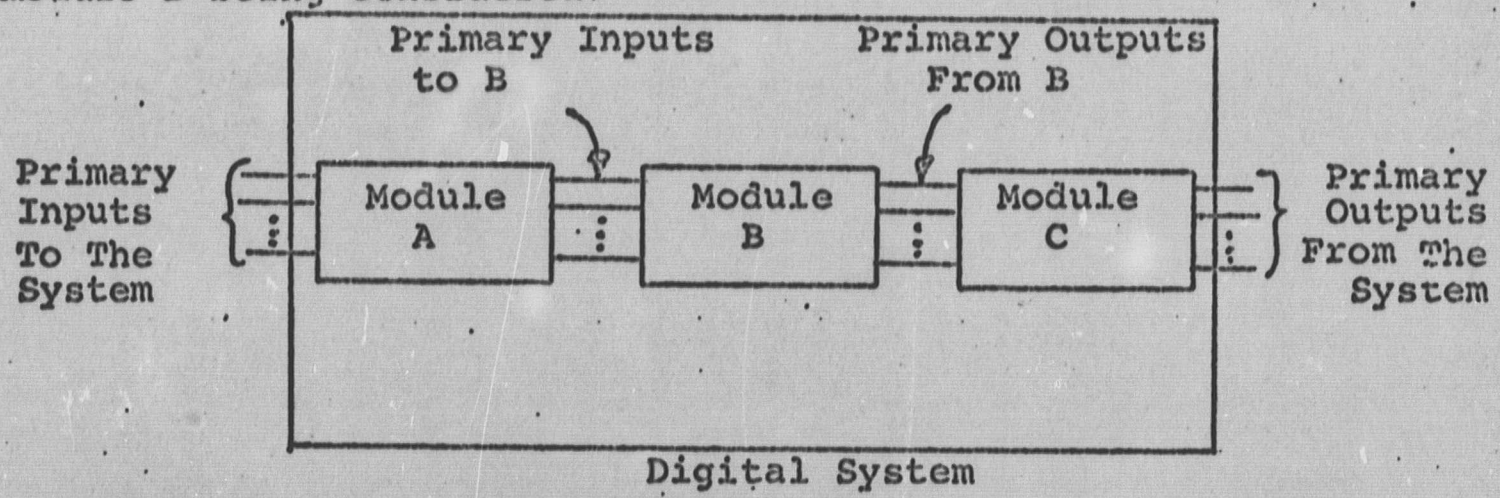



Figure 2. Compatability of Primary Inputs and Outputs

The test generated for a given fault will be in terms of primary inputs and outputs of B. The first question to be considered is, can the test specified be realized through the primary inputs to the system? The second question involves a determination of whether or not the primary outputs of B, which distinguish the fault, are observable as primary outputs to the system? These questions become extremely practical when one considers large digital systems. Nothing can be more useless than a set of diagnostic tests specified as inputs to module B and not having these inputs available, or at least possessing the capability of generating the B inputs through the primary inputs.

This algorithm can answer the questions posed above as a special case of the situation where a subset of inputs or outputs are provided.

Section II of this paper will consider details of the system implementation to perform the functions considered thus far. This will be followed by a discussion of the results and evaluations of the algorithms.

One required function of the test generation system remains to be considered. It is not grouped with the other functions because it is controlled by the overall simulation and diagnosis system and not the combinational test generation segment. This

function utilizes the combinational test generator as an aid for sequential logic test generation. The combinational algorithm can be utilized for this function in at least two ways. First, we can utilize the following heuristic: break the feed-back loops in a sequential circuit, consider the broken loops to represent primary inputs and outputs to the circuit, and then generate a test for a fault in the corresponding combinational net. After the test is generated, the feedback loops are reconnected and the test is applied by the simulator. For many cases the simulation will show that this particular test does not detect the fault. However, for sequential logic with limited feedback, the process is more successful. In lieu of anygeneral algorithm for sequential test generation, which can be implemented for large nets, this approach appears to be acceptable.

The second possibility, presently under investigation, is the extension of the combinational testing procedures to specific classes of sequential circuits; for example, circuits containing only set-reset flip-flops. Preliminary results in this area appear to offer some encouragement.

## II. IMPLEMENTATION

Consider a combinational circuit, C, consisting of N gates and L levels. The gates in the first level are input gates, that is, primary inputs to C, and the gates in level L are output gates, that is, primary outputs of C. Seven gate types are presently considered: AND, OR, NAND, NOR, NOT, INPUT, or OUTPUT, but the method presented is easily extended to other gate types. First, the gates are numbered sequentially by levels. The line (or lines) forming the output of gate I is called line I. A line will have one of the following line values: X, undetermined line value; 0, constant value 0; 1, constant value 1; D, variable value, 1 for the fault free circuit, 0 for the faulty free circuit, $\bar{D}$, variable value, 0 for the fault free circuit, 1 for the faulty circuit.

The forward trace is accomplished by the TRACF subroutine which accepts a gate number, gate type, fan in, and fan out. TRACF examines the line values of the fan in lines and either performs a successful forward trace or signals that it can not. A successful forward trace occurs (1) when one or more input lines have a value D or $\bar{D}$ (note that a mixture of one line being D and one $\bar{D}$, prevents a forward trace), and (2) when the remaining input lines have value X or IVA (Input Value Allowed) where the IVA depends on the gate type. The IVA for an AND gate, for example, is 1. Under all other conditions a successful forward trace cannot occur and, in fact, a value OVF (output value) is forced on the output lines. Finally, adopting the notation that D and $\bar{D}$ are of opposite parity, and noting that the successful forward trace involves setting all input lines with X values to IVA and setting the output to proper parity, the functioning of TRACF for four gate types is summarized in Table I.

The back trace function is accomplished by the TRACB subroutine, which accepts a gate number, gate type, fan in, and fan out. An attempted back trace thru a gate has one of four outcomes: (1) successful back trace, only one possible way to accomplish back trace, (2) successful back trace, K ways to accomplish back trace, where K is the number of input lines with value x, (3) back trace already completed, (4) back trace

impossible because of a conflict. The functioning of TRACB can be summarized in the following table II showing the outcomes as numbered above under various input line conditions:

| Gate Type | IVA | OVF | Successful Output |
|---|---|---|---|
| AND | 1 | 0 | Same parity as input |
| OR | 0 | 1 | Same parity as input |
| NAND | 1 | 1 | Opposite parity from input |
| NOR | 0 | 0 | Opposite parity from input |

Table I. TRACF Function

| Gate Type | AND | | OR | | NAND | | NOR | |
|---|---|---|---|---|---|---|---|---|
| Value to be Traced Back | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| All K Lines = X | (1) | (2) | (2) | (1) | (2) | (1) | (1) | (2) |
| Some=IVA, Remaining K=X | (1) | (2) | (2) | (1) | (2) | (1) | (1) | (2) |
| Some=$\overline{\text{IVA}}$, Remaining K=X | (4) | (3) | (3) | (4) | (3) | (4) | (4) | (3) |
| Some=IVA, Some=$\overline{\text{IVA}}$ | (4) | (3) | (3) | (4) | (3) | (4) | (4) | (3) |
| Any 2 = D and $\overline{\text{D}}$ | (4) | (4) | (4) | (4) | (4) | (4) | (4) | (4) |

Table II. TRACB Function

Note: for (1) Set K lines to IVA ...
(2) Set 1 of K lines to $\overline{\text{IVA}}$

Consider the following circuit with fault 1 ($F_1$) present (but not $F_2$):

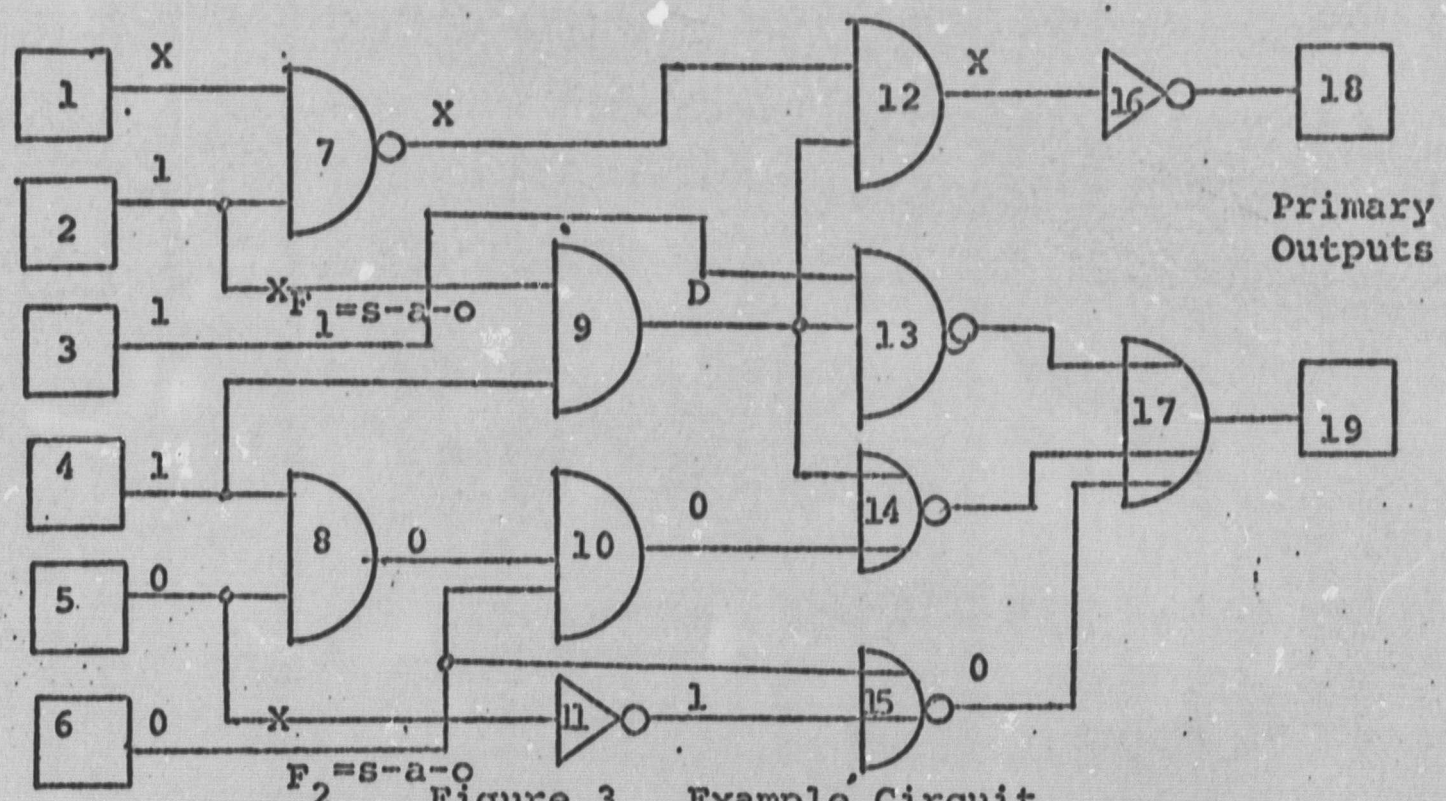



Figure 3 Example Circuit

It is desired to generate a test for fault $F_1$. In this circuit, there are many possible combinations of forward and back traces, some of which lead to a solution. In order to consider all of these combinations systematically and efficiently, the method used is to dynamically build and search a three dimensional tree or digraph described below.

The values of line 9 may be traced forward thru one of 3 single paths {9,12,16,18}, {9,13,17,19} or {9,14,17,19}, or one of 4 multiple paths which are combinations of the single paths 2 at a time, and 3 at a time. The forward trace portion of the tree corresponding to these paths is in the X Y plane. Thus, initially the value D is thru gate 9 which has fan out 12,13,14. This corresponds to the tree.

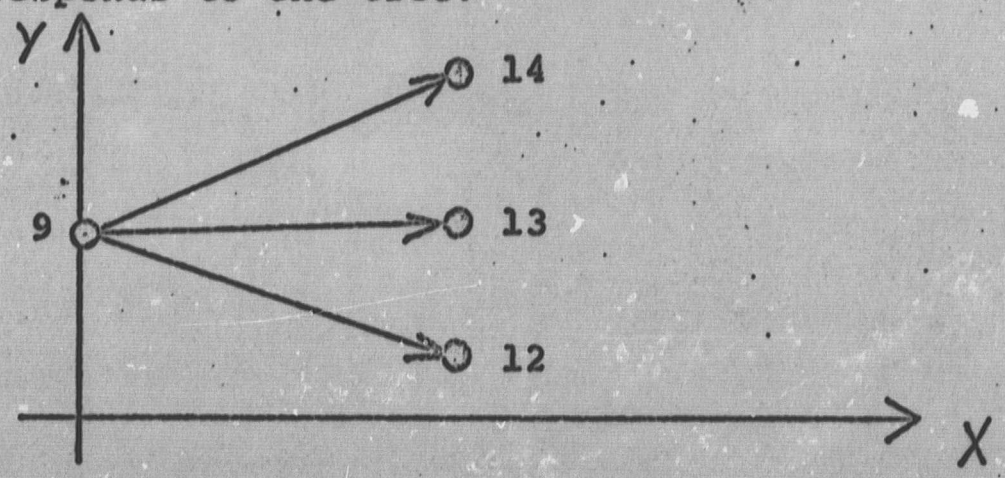



Figure 4 Digraph Partial Base

This digraph may grow and modify itself by adding and deleting branches until a solution is reached. Thus, the portion of the digraph for the solution in the X Y plane will be a subset of:

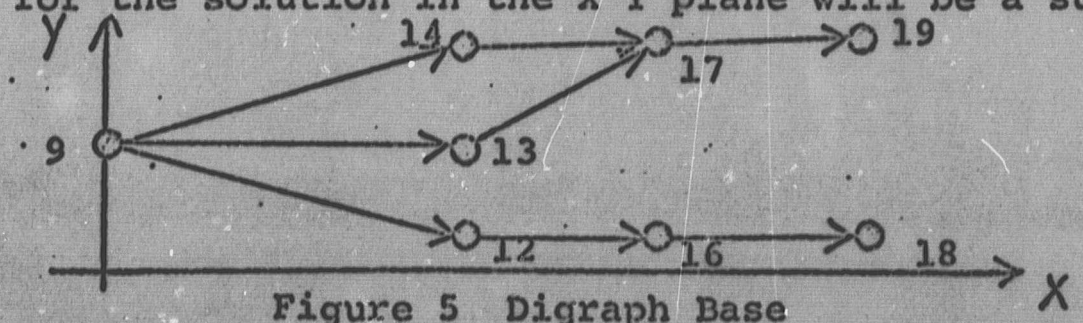



Figure 5 Digraph Base

Each node in the digraph is the beginning of a sub-digraph in the z dimension. Each of these subgraphs represents the particular trace back used to justify the line configuration. For example, in order to trace a D thru gate 14, line 10 must be set to 0. The trace back must be one of the following:

| Line | 4 | 5 | 6 | 8 | 10 |
|---|---|---|---|---|---|
| Value | X | X | 0 | X | 0 |
| | 0 | X | X | 0 | 0 |
| | X | 0 | X | 0 | 0 |

which means

that the subgraph above 14 would be one of these

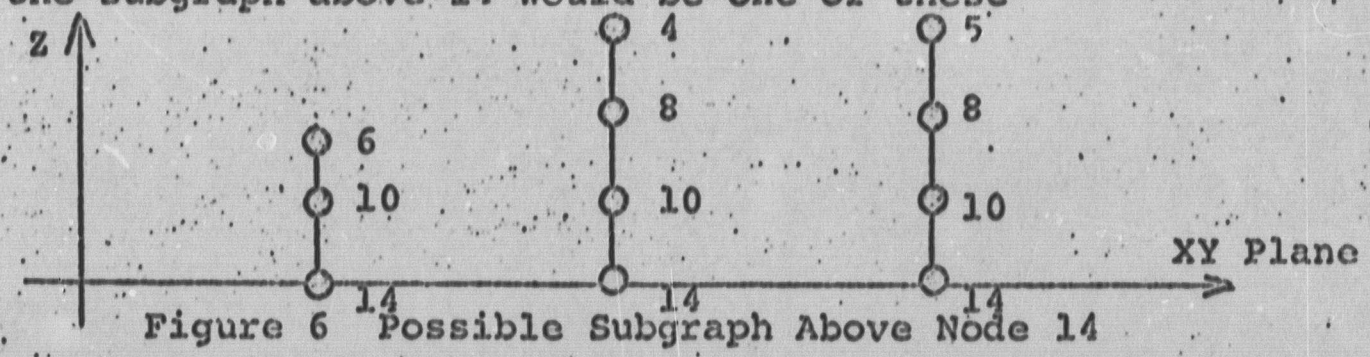



Figure 6 Possible Subgraph Above Node 14

The complete digraph is constructed node by node, back tracing at each node. The leading points of the digraph are entries in the ELIST of the program. The leading points of the subgraph being constructed in the z dimension are entries in the BLIST. The process of the mainline program is flow charted below:

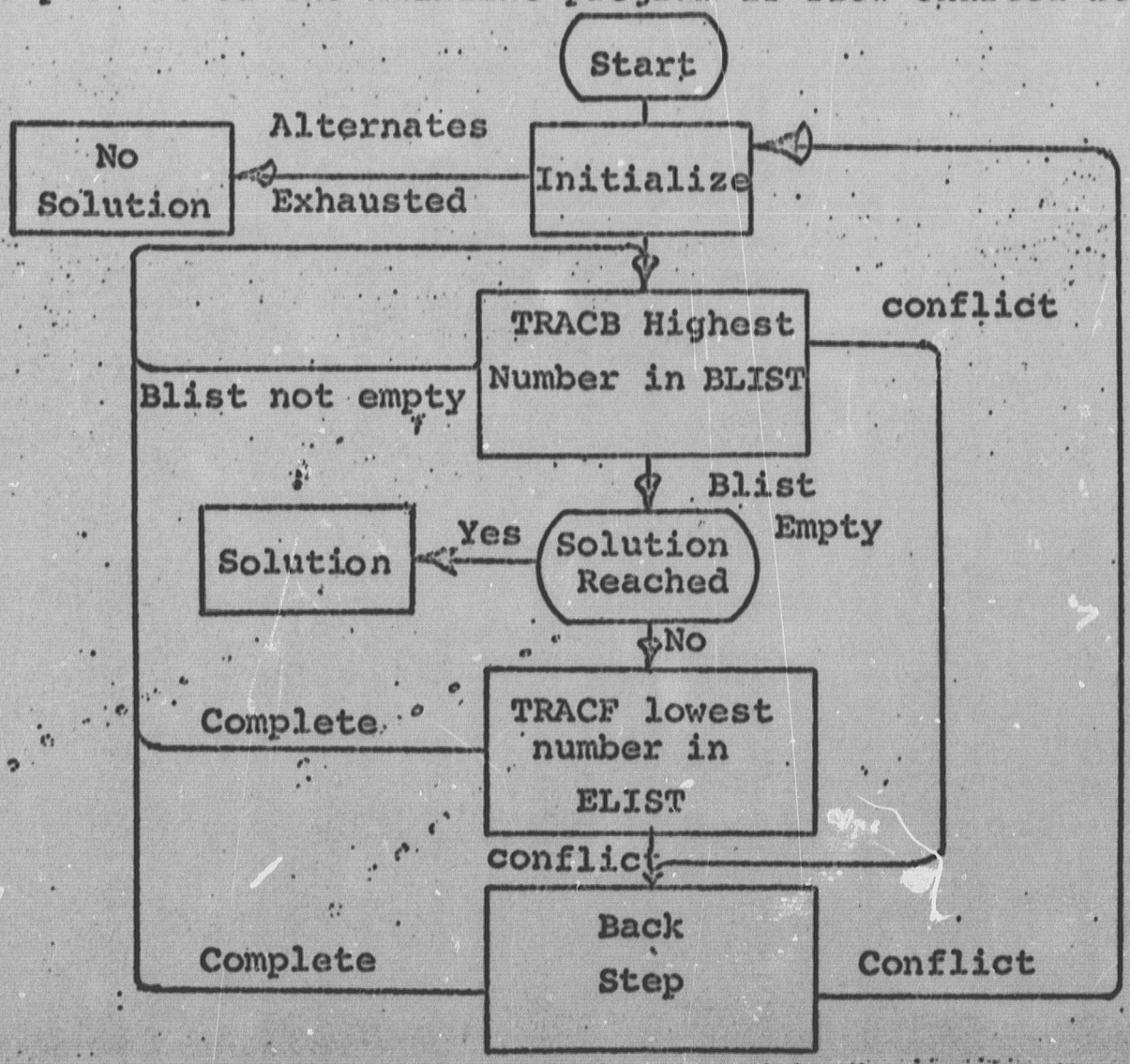
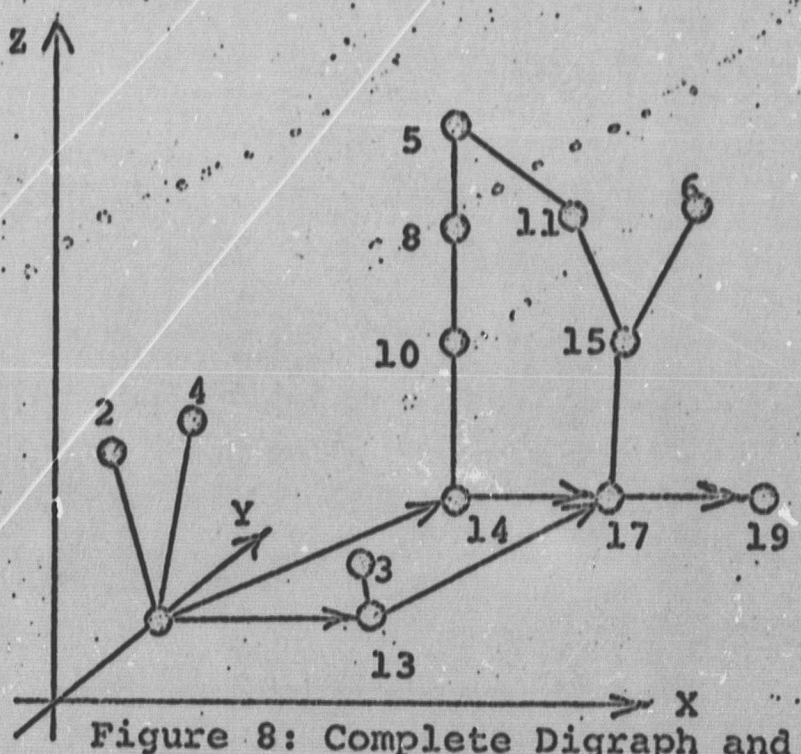



Figure 7 Mainline Flow Chart

The back step procedure collapses the portions of the graph that are in conflict until a branch point is reached that will resolve the conflict by taking an alternate branch. It does so, efficiently, by bypassing branches where exploring alternate plans would accomplish nothing. This is a significant feature of the implementation, saving a considerable amount of time in the test generation. It accomplishes this simply by noticing what line values cause the conflict, then erasing the graph until a node number is encountered that effects one of these line values.

The bookkeeping of the mainline is done in a MFILE. This file has a record for each forward trace (F record) and one for each back trace, and reflects the current structure of the digraph and how it was built. Each record has m+2 entries where m is the number of fan in lines; the entries are NG = gate number, NP = preference, and LN (M entries) = a list of lines altered by that forward or back trace. NP is a preference number that prevents the back step procedure from taking a branch more than once, and taking each significant branch at least once. As an illustration, consider a solution of the problem in our example circuit, the corresponding digraph, and the MFILE given below:

| SOLUTION: line | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| values | X | 1 | 1 | 1 | 0 | 0 | X | 0 | D | 0 | 1 | X | $\bar{D}$ | $\bar{D}$ | 0 | X | $\bar{D}$ | X | $\bar{D}$ |




MFILE

| NG | NP | LN |
|---|---|---|
| 9 | -1 | 2,4 |
| 12 | 0 | 0,0 |
| 13 | -2 | 3 |
| 14 | -2 | 10 |
| 10 | 1 | 8 |
| 8 | 0 | 5 |
| 17 | -1 | 15 |
| 15 | 0 | 6,11 |
| 11 | 0 | 0 |
| 19 | -1 | 0 |

Figure 8: Complete Digraph and Final MFILE for Solution

Finally, note that the digraph is meant to be an aid in visualizing the process and that the mainline works with the MFILE, CFILE (which describes the circuit), BLIST and ELIST only. The "back step" process is done by back stepping up the MFILE to the appropriate record.

As noted in section I, the functions (2) and (3) of the system are performed exactly as function (1) except that the initialization step is different. To generate a test to distinguish between two given faults, $F_1$ and $F_2$ (the extension to many faults is straight forward although it involves some combinatorics) the initializer must (a) connect $F_1$ and $F_2$ by a path which brings the faults into opposite parity, or (b) cut off one from the path of propagation of the other, either by terminating its path or by directing it to a different primary output.

Thus, considering $F_1$ and $F_2$, in the example, the initializer may choose to cut off $F_2$ by setting line 6 to zero or it may propagate both to line 17 where they meet in opposite parity.

The final function, the setting of the lines to the specified subsets of primary inputs and/or primary outputs is an initialization step; the mainline very simply will verify the specified values if the configuration is valid. Space consideration prevent a more extensive explanation of these functions.

## III. CONCLUSION AND RESULTS

Fortran IV was utilized for the programs described above. The storage requirements are as follows. The mainline program initializer, TRACF, and TRACB each require approximately 1K words of core. The array requirements vary with the circuit size. If N is the number of gates in the circuit, and M is the maximum fan in or fan out, then the CFILE is less than Nx(2M) words while the MFILE never grows larger than M records M+2 words per record = N x (M+2) words

Minimum core requirements (overlaying the subroutines and the initializer) are slightly in excess of 2K words (this includes small buffers).

Of course, if enough storage is available, it is desirable to have both files in core. At the other extreme, a minimal buffer size would be enough for one record for each file (containing the last record read or written). For available storage of intermediate size the following strategy is used: When a circuit has little or no reconvergent fan out, it is rarely necessary to read from the MLIST; hence for this type of circuit, divide the available storage into a 10% MLIST buffer and a 90% CFILE buffer. On the other hand, for a circuit with extensive reconvergent fan out, it is necessary to increase the MLIST buffer to 70% of available storage, leaving 30% for the CFILE buffer.

The time necessary to generate a test for a circuit of approximately 100 gates varies between 1 to 10 seconds, depending on the structure of the circuit, the particular fault, and the availability of files in memory. Larger circuits (containing more than 1000 gates) are presently being prepared for analysis).

## IV. References


1. Poage, J.F., "Derivation of Optimal Tests to Detect Faults in Combinational Circuits", Mathematical Theory of Autcmata, Polytechnic Press, Brooklyn, N.Y., 1963.

2. Armstrong, D.B., "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets", IEEE Trans. on Elec. Computers, Vol. EC-15, No. 1, 1966.

3. Roth, J.P., "Diagnosis of Automata Failures: A Calculus and A Method", IBM Journal of Research and Development, Vol. 10, 1966.

4. S.A. Szygenda, "A Software Diagnostic System for Test Generation and Simulation of Large Digital Systems", Proc. of the N.E.C. Annual Conference, December 1969.

5. S.A. Szygenda, and D. Rouse, "Functional Models for Simulation of Large Digital Systems", Proc. of the Joint Conf. on Math. and Computer Aids to Design, October, 1969.

Proceedings of the 1969 National Electronics Conference

# A SOFTWARE SYSTEM FOR DIAGNOSTIC TEST GENERATION AND SIMULATION OF LARGE DIGITAL SYSTEMS

Stephen A. Szygenda
University of Missouri-Rolla
Rolla, Missouri 65401

## I. INTRODUCTION

The ever increasing reliability requirements being placed on digital computing systems, have stimulated a growing interest in digital simulation and fault diagnosis. Early approaches to computer diagnosis relied heavily on the maintenance man's experience and intuition. This approach proved to be wholly inadequate, even for those early systems. To relieve this problem, the burden of diagnosis was shifted to software diagnostic programs. The technique of testing computer hardware by the use of software diagnostic tests has proved to be fruitful.

The need for digital simulation and diagnosis has been well established. However, major problems remain with respect to the manner in which diagnostic tests are generated and validated. Due to the magnitude of the problem, for large digital networks, it is generally agreed that digital simulation and automatic diagnostic test generation are essential to meet existing and future reliability demands. It is this portion of the problem with which we will be concerned.

Many people are actively working on problems in this area. Unfortunately, most approaches to date have been limited by numerous constraints. Peculiarities of hardware often dictate approaches which are applicable only to the particular hardware being considered. Demands to produce diagnostic tests, within stringent time constraints, have also dictated limited approaches to the overall problem. Other limitations include: an unfamiliarity of the diagnostician with the hardware to be diagnosed; absence of knowledge of fruitless approaches previously taken; and a lack of familiarity with the tools available for diagnosis. In many cases, these limitations are a result of poor communication between diagnosticians. As a consequence of these conditions, it often happens that a vague problem definition is provided and the resulting solution, if one is achieved, is not a solution to the intended problem.

Due to the magnitude of the problems associated with developing a test generator for large systems, (assuming this task is possible), it is not surprising that most present efforts have been limited to meet particular needs.

This paper will attempt to provide a summary of a software system design for a general purpose digital simulator and diagnostic test generator. Furthermore, we will attempt to define major problem areas and specify the interfaces necessary for the numerous components of the software system.

## II. SYSTEM OVERVIEW

The proposed system will be governed by the following objectives:

1. The primary objective will be generality of application and ease of use.
2. The system should possess the capability of simulating large digital computing systems, perhaps an entire processor, within reasonable time limits.
3. The simulator should be designed to serve the dual role of hardware design verification and diagnostic test evaluation.
4. The system should be able to automatically generate diagnostic tests for various failure modes.
5. Optimization of diagnostic tests should be considered
6. Variations in hardware descriptions should not severely limit the use of the system.
7. The ability should exist to produce various output configurations, as specified by the user.
8. The system should not be restricted to operations on a particular series of computers, i.e. should not be machine dependent.

The system is comprised of five major areas (figure 1) including master control, pre-processing, simulation, diagnosis , and post-processing.

The master control will read and interpret all commands issued by the user. It will also supply default conditions for options not specified by the user. Under normal operation, the user will only have contact with the master control portion of the system. These control programs will coordinate all program interfaces, as well as establishing queues for the various tasks.

The complexity of the problems associated with the other areas of the overall system warrents a more detailed discussion. This is provided in the following sections.

## III. PRE-PROCESSING OF LOGIC DESCRIPTIONS

The importance of the pre-processing phase of any simulation and diagnosis system is often underestimated. The problem is somewhat less critical if the circuit to be analyzed is contained on a design file. However, the same class of problems exist with respect to the initial generation of the design file. Since an integration of design and diagnosis is likely in the future, the problem is of importance in either case.

Figure 2 provides a diagram of the pre-processing system. There would be two major tasks for these programs. First, we have the validation and bookkeeping associated with the input logic specifications. Secondly, the logic must be organized in a form suitable for simulation. The latter task could be performed in the simulation portion of the program, but it will be considered as a pre-processing function because of memory size restrictions.

### LANGUAGES

The first question to be answered, with respect to pre-processing, is the language to be used for the circuit description and programs. The desired system generality dictates the use of a high level language for the programs. Also, the language used should be capable of handling different logic nets irrespective of the gate level design language used by the designer. For example, names of gates used on the logic drawings, fan-in or fan-out specifications, etc., should not require any special action to be taken by the user.

Because of time and storage consideration, it is likely that a functional description of logic modules would be desirable. With respect to the logic description, this creates a number of problems. The possible kinds of functional modules are

extremely large, therefore, it would be impractical to consider all of the possibilities. This system would assume certain functional descriptions, such as: Boolean equations for combinational logic, register and transfer descriptions, and flow tables. It is realized that this imposes some limitations on the user. To minimize this difficulty, the facility to easily add additional macro-type functional descriptions should be provided.

## VALIDATION OF THE INPUT LOGIC DESCRIPTION

Checking the logic description which is being input is the one task most grossly underestimated. It is not uncommon to spend weeks in the preparation of a large net for simulation, with the simulation and analysis taking only a few hours.

A number of possible problems and solutions in this area will be considered. Due to space limitations, this is by no means intended to represent a complete set.

### Input Format

First, we will consider the more common input format checks. Numerous errors can occur during this phase of preperation. These include: card format errors, incorrect labeling of gates, primary input and output specification, omission or duplication of portions of the logic description, assignment of the wrong logic function, etc. Existing systems usually perform some type of format check. However, most of these are not adequate. This system would provide a complete check and cross check of the input format description.

It should be pointed out that at least one study, by the author, has shown that technicians, with little formal training in logic design, have proved to be the individuals most reliable for coding gate networks. Their reliability is even further enhanced if they are required to describe a gate and its corresponding inputs and outputs independently of the other gates. The interconnections of the gates is performed by the program. This approach is taken in the system being described.

### Relabeling of Gate Names

Often, a logic designer introduces a naming scheme for the logic gates which provides him with information about the functioning of the logic. These gate names are usually not in a form suitable for efficient digital simulation. To obtain a suitable simulation form, we could either require the user to relabel the gates, a procedure obviously prone to numerous errors, or require the programs to automatically relabel the gates. The latter approach is the one to be taken.

Unfortunately, the relabeled gate names would carry little significance for the designer. Hence, before any information is output from the system, the names would be translated back to the original names.

### Primary Inputs, Outputs and Test Points

Digital simulation techniques often require the user to manually provide a complete specification of primary inputs, outputs, and test points. For large logic nets, not restricted to register inputs and outputs, this could prove to be an extremely time consuming and error producing procedure. For these reasons, primary input and outputs would be generated automatically and cross-checked with those provided. Any discrepancy will be reported to the user. Test points are handled as primary outputs, for observability, when in actuality they may not be primary outputs.

### Interconnection of Logic Descriptions

Considering one sheet of a logic net description, $\ell_i$, as representing a subset of the logic net, L, then a complete specification of the logic exists if we have a description of all the $\ell_i$ and their interconnections. That is,

$$L = \ell_1, \ell_2, \ell_3, \ldots, \ell_n$$

The reason for this approach is twofold; first, it proves to be much simpler to independently describe sheets of a logic net, and secondly, it provides a built-in modularity of the description. The obvious problem is the interconnection of the $\ell_i$. With the possibility of numerous sheets and interconnections, this procedure is obviously not reliably performed by humans. Hence, the software system would perform this final task of the pre-processor logic validation phase.

## LOGIC ORGANIZATION

Organization of digital logic for simulation is dependent upon the simulation model or models adopted. For strictly combinational logic, an organization with respect to logic levels is likely. This scheme has also been used for sequential asynchronous logic[1]. Regardless of the model to be chosen, a number of common functions are likely. These are summarized below.

### Fan-In, Fan-Out Description

For checking purposes, and generality of the system, the logic is organized in both a fan-in and fan-out description. Either, or both, could be provided by the user. The program would generate the missing information and cross-check the description. Beside providing an option for the input description, this procedure would provide a final over-all check of the logic description to be used for simulation.

### Leveling of the Logic

Regardless of the simulation model adopted, it is likely that logic leveling will be necessary, at least for portions of the logic. Therefore, programs to efficiently perform the logic leveling will be necessary. This phase of the system also seems to be a likely place to define the logic in terms of simulation modules. All of the gate level logic would be organized in modules with corresponding functional modules, where possible. During simulation, these functional and gate level modules would be used interchangeably to satisfy space and time specifications.

Depending on the simulation model being used, other organization algorithms may be necessary. These should also be provided along with the facility for easy program modification, if necessary.

### Location of Internal Feed-Back Loops

For various reasons, most simulation and diagnosis systems require the location, and in some cases, the opening of internal feedback loops. Unfortunately, these systems require this function to be performed manually. For large logic nets with a heavy density of internal feedbacks, the performance of this function, in a reasonable time, is practically impossible. For these reasons, this system would automatically locate, and if necessary, attempt to open the feedback loops. This is by no

means a trivial task. The opening of the feedbacks must be done in such a manner as to insure that the logic net will still act analogously to the physical system. Some basic theoretical work [2] has been done in this area and will be included in the system.

The discussion above summarizes the specifications of the pre-processor phase of the system. The master control will coordinate the functioning of the pre-processor. When the pre-processing has been completed, the logic description in a form suitable for simulation, will be stored for use by the simulator. Hence, the pre-processor will not reside in memory during the simulation and diagnosis phase.

## IV. SIMULATION OF LARGE SCALE DIGITAL LOGIC NETS

Digital simulation, that is, the asking of programs to predict a systems behavior under both failure and fault free conditions, would serve two major functions in this configuration. First, the circuit design will be verified through the use of simulation, and secondly, the simulation will be used as a tool in the generation and evaluation of diagnostic tests. Figure 3 illustrates such a system.

To maintain the machine independence desired, the programs would be written in a high level language. It would not be mandatory that the program be in the same language as the pre-processor, since they are relatively independent software systems. The possibility does exist that a list processing language would prove to be beneficial for portions of the simulator.

### FUNCTIONAL SIMULATION

To possess the capability of simulating very large digital systems using a reasonable amount of computer time and memory, it will be necessary to perform functional simulation, as well as some lower level simulation. To achieve this end, it is suggested that the logic be divided wherever possible, into either register and transfer logic, combinational logic, or sequential logic. This division would be made in an attempt to achieve modularity at a level permitting easy interface with the lower level simulation. The functional divisions would be largely dependent on the subject hardware. Hence, the divisional specification for functional simulation would have to be provided by the user. A special design language, closely representing the hardware functions, may be useful in decreasing the burden placed on the user.

### NODAL SIMULATION

Functional simulation would be beneficial in decreasing the simulation time and storage requirements, as well as aiding in the validation of the logic. However, functional simulation could not provide critical information needed for logic diagnosis. For this task, we need some form of nodal simulation, that is , simulation at a level specified by the user, for diagnostic resolution. Many existing systems simulate at a gate level. This would be necessary if faults were to be resolved to each individual gate; however, this is usually not the case. Most users of diagnostic tests would be satisfied with fault resolution to some replaceable unit or card. This would be especially true if that approach would produce a considerable savings of computer time spent for simulation. The level of fault resolution could also be approached from a diagnostic test generation viewpoint, as will be done later.

This system would provide the user with the capability of providing a description for lower level simulation at either a gate level, a commonly used combination of gates level, or at some replaceable pack level.

Another important consideration is the type of elements which can be simulated. This should be kept flexible enough to permit easy insertion of new elemental simulation types.

### SIMULATION MODEL

The selection of a simulation model, or models, is probably the most important problem in any simulation system. If the simulation model does not closely represent the physical functioning of the logic, the simulation is useless. Because of the wide variety of techniques used for digital logic design, it is felt that a general purpose digital simulator should possess the capability of simulating types of logic from combination to sequential and asynchronous to synchronous, as well as systems using combinations of these. Whether or not this goal can be achieved is still under investigation. The system under discussion would provide both synchronous and asynchronous simulation of combinational and sequential logic. Even with this limited attack at the problem, a number of modeling questions need to be answered. Some of these are: whether propogation delays should be considered; if so, should the propogation delay be considered to be uniform; can feedback loops be set [3] even under failure conditions; is a Huffman model sufficient for race analysis; what type of hazard analysis is to be performed; will the logic function in a fundamental or pulse mode; etc.

It is felt that questions, with respect to modeling, have been the primary impasse to a general logic simulator, and hence, should be the area receiving maximum attention from diagnosticians and circuit designers.

### SIMULATION TECHNIQUES

The following techniques have been used for digital simulation. Their relative advantages and disadvantages are greatly dependent on the simulation model being used.

Compiled simulation uses the machine operations, available on the host machine, to replace each node description with a sequence of code capable of simulating the node. Simulation is then performed by simply executing the compiled logic.

Table driven simulators store the simulation data in tables (preferably dynamically relocatable tables) to be used in the simulation procedure. Simulation is performed by the manipulation of the stored data with the aid of the logical operations of the program language.

Another approach being taken to simulation is the use of some form of a path sensitizing [4] technique for simultaneous simulation and diagnostic test generation. This approach will be more throughly discussed in the diagnosis section.

Another possibility not to be ruled out is the use of some mixture of the techniques described, to achieve maximum efficiency. This appears to be especially true for sequential logic.

Regardless of the simulation technique adopted, it is likely that the use of selective trace would result in a savings of simulation time. Simulation

performed with the aid of selective trace takes into consideration the fact that the repeated application of input stimuli does not always result in all internal nodes changing state. If simulation is performed only along those paths which are changing state, a considerable savings in simulation time can be achieved.

The next section of this paper will deal with fault diagnosis. It should be understood that simulation and diagnostic test generation are closely interdependent. For simplicity of discussion, we have considered them separately. However, it is likely that any efficient software system will possess many programs used by both these areas.

## V. DIAGNOSIS OF LARGE SCALE DIGITAL LOGIC

The problems of diagnostic test generation are by no means new. A number of significant steps have been taken in this area. However, many problems remain; in fact, most techniques presently being used are seriously restricted by assumptions which may not be valid.

The question of diagnosis can be divided into four major areas; fault specification, combinational logic test generation, sequential logic test generation, and diagnostic test optimization. These four areas, as shown in Figure 4 will now be considered.

### FAULT SPECIFICATIONS

Presently, the logical stuck at zero (s-a-o) and stuck at one (s-a-1) faults are accepted as representing the major failure modes. However, it is not obvious that this condition will hold for different hardware technologies. For this reason, a diagnosis system should posses an open ended program where different failure modes could be easily added by the user.

Fault Insertion: effect on selection of simulation techniques

The topic of fault insertion could be discussed as part of either simulation or diagnosis. The treatment of the faults is a function of diagnosis, but fault insertion techniques are greatly dependent on the simulation methods adopted. Because of the complications which can be caused by considering fault insertion for a general class of failure modes, a certain amount of simulation efficiency is lost in an attempt to maintain fault specification flexibility. The extent of this loss may very likely dictate the degree of system generality to be achieved in this area.

### COMBINATIONAL LOGIC TEST GENERATION

The problem of combinational test generation has been the one most vigorously attacked. With the formalization of the one dimensional path sensitizing algorithm[4], tests could be generated for most logical s-a-1, and s-a-o type faults. An extension of this one dimensional method to a multi-dimensional method[5] appears to provide the added capability necessary to generate tests capable of detecting all these logical faults. A number of problems still exist, with respect to optimization of tests. These will be discussed later.

The system being described will contain a multi-dimensional path sensitizing technique. Implementation problems include: a more efficient technique for test generation, optimization of tests, both during and after generation, other methods for combinational test generation, and the use of combinational test generation techniques for sequential circuits (to be discussed in the next section).

### SEQUENTIAL LOGIC TEST GENERATION

Unfortunately, the outlook for sequential logic test generation does not look as bright as it does for the combinational case. No practical algorithm for sequential test generation of large logic nets is likely. This is due to the fact that each possible test input must usually be evaluated for each possible circuit state. Therefore, the amount of computation required doubles for each memory state in the circuit. Furthermore, this approach may require extensive homing sequences for initialization.

For these reasons, this system will implement a number of heuristic techniques, which will hopefully be capable of generating tests for almost all the logical type faults.

Four basic strategies have been implemented by Seshu[1]. Extensive use of these strategies has discovered a number of weaknesses. Some of these are being corrected by modifying the existing approaches. However, new heuristics will also be necessary. One of the problem areas is concerned with the local nature of many existing heuristics, that is, very little attention is devoted to the implications of future diagnosis when tests are selected. For example, if the test generation criterion is fault detection only, it is likely that we would choose that test capable of detecting the largest number of faults. If test $t_1$ detected 10 faults and test $t_2$ detected 5 faults, $t_1$ would be chosen. However, it is possible that on the next cycle, a test $t_i$ would be found capable of detecting 20 faults if $t_2$ had been chosen, when in actuality, it was necessary to settle for detecting 5 faults because $t_1$ was chosen. Hence, this system will include heuristics concerned with the more global implications of the test selection procedure.

Another heuristic would make use of the combinational test generation algorithm. If the feedback loops of a sequential logic net are broken, tests could be generated for the resulting combinational logic. The feedback loops could then be reconnected and the test evaluated under realistic conditions. If major changes in the internal state space have not resulted from the changes in the input space, the test may be useful. Other heuristics should also be investigated to produce an efficient overall system.

The use of a particular heuristic for diagnostic test generation is usually dependent on the particular hardware being analyzed. It has been found that certain heuristics are usually successful on certain types of logic. Hence, the heuristics will be kept in a queue where the most successful approach, as indicated by a user dictated criterion is always tried first. This procedure could result in a significant savings of computer run time.

### DIAGNOSTIC TEST OPTIMIZATION

The previous discussion has been directed toward generation of diagnostic tests. An equally important, and not at all disjoint portion of the problem is test optimization. A large logic net could possibly contain tens of thousands of faults. To attempt to generate a test for each of these faults would be impractical. Furthermore, an attempt to apply this number of tests would be so

FOR A BETTER COPY CONTACT DOCUMENT ORIGINATOR.

time consuming as to render the entire procedure useless. Hence, test optimization is a prime consideration.

For a system to be practical, it is likely that test optimization should be performed simultaneously with test generation whenever possible. Performing test minimization as a separate function would require the generation of a large number of tests, followed by the analysis of the tests to achieve a reduction in size.

To perform optimization simultaneously with test generation, the overall degree of diagnosis desired must be known. If only fault detection is required, the objective would be the least number of tests possible. If diagnostic resolution to a particular fault is required, very little test minimization would be possible. In actuality, the degree of diagnosis is usually somewhere between these two extremes. Fault isolation to either one or a few replaceable cards is, in most cases, acceptable. Since the degree of diagnosis is a variable, depending on the particular users requirements, the optimization programs should possess enough generality to permit the user to specify the fault resolution desired.

Another important consideration is the number of tests required to detect a fault. Although, through simulation, we may have determined that a particular test will detect a particular fault, we must keep in mind that we are dealing with a simplified model of the actual system. Hence, a non-zero probability exists that this test would be masked in such a manner as to prevent fault detection. Some investigators contend that every fault should be covered by at least two, and preferably three, tests. This requirement represents another variable which is highly dependent on the maintainability requirements. Therefore, test coverage should be specified by the user. The test optimization techniques would be greatly influenced by this objective.

## VI. POST-PROCESSING

The major function of the post-processor will be the production of a fault catalogue or dictionary. This dictionary can list the diagnostic tests, the faults detected by the tests, the faulty output patterns and the location of the replaceable pack containing the fault. The production of a fault dictionary can be an extremely laborious task. However, if simulation and diagnosis are implemented as described in this system, the post-processing task will be minimal. All the data needed for dictionary production will be available at some stage of the analysis procedure. Hence, bookkeeping and output format problems will be the post-processor's assigned function.

It should be kept in mind that rapid replacement of faulty logic and resumption of service is the supreme goal of any diagnosis system. The speed with which a maintenance man can locate information in the fault dictionary is an important consideration. For these reasons, different dictionary formats, such as diagnostic trees, tabular form, graphic form, etc., should be available to the user upon request.

## VII. CONCLUSION

As was stated previously, this paper does not attempt to solve the problems associated with large scale logic simulation and test generation. However, it does attempt to define a system necessary to accomplish this goal. The magnitude of the problem itself creates complications, with respect to understanding, that may, in many instances, delay, if not totally prevent, the solutions required. It is to this problem of understanding that this paper has been directed.

The implementation of a general simulation and diagnostic system, similar to that described, is the desire of most diagnosticians. With the advent of new hardware technologies, this desire may very likely become a necessity. The implementation of this type of system could greatly facilitate the integration of circuit design and diagnosis, which in turn could increase the computers' reliability and maintainability.

The described system is presently under development at the University of Missouri. Preliminary results have been quite promising, and it is hoped that this system will provide a solution to some of the problems described in this paper.

## REFERENCES


1. Seshu, Sundaram, "The Logic Organizer and Diagnosis Programs," Report R-226, Coordinated Science Lab., University of Illinois, Urbana, July 1964, pp. 1-88.

2. Ramamoorthy, C.V., "A Structural Theory of Machine Diagnosis," Proc of the Spring Joint Computer Conference, 1967, pp. 743-756.

3. Huffman, D.A., "The Synthesis of Sequential Switching Circuits," Sequential Machines, edited by E.F. Moore, Addison Wesley, 1964.

4. Armstrong, D.B., "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets," I.E.E.E. Transactions on Electronic Computers, Vol. EC-15, pp. 66- 73, Feb. 1966

5. Roth, J.P., "Diagnosis of Automata Failures; A Calculus and a Method," IBM Journal, July 1966, pp. 278-291.

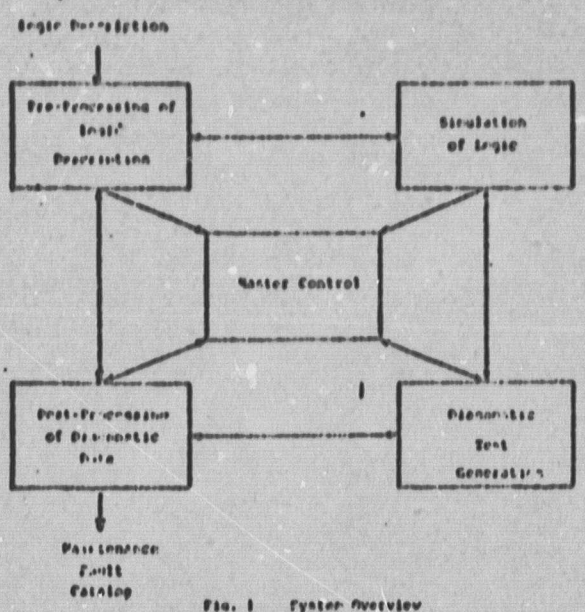



Fig. 1 System Overview

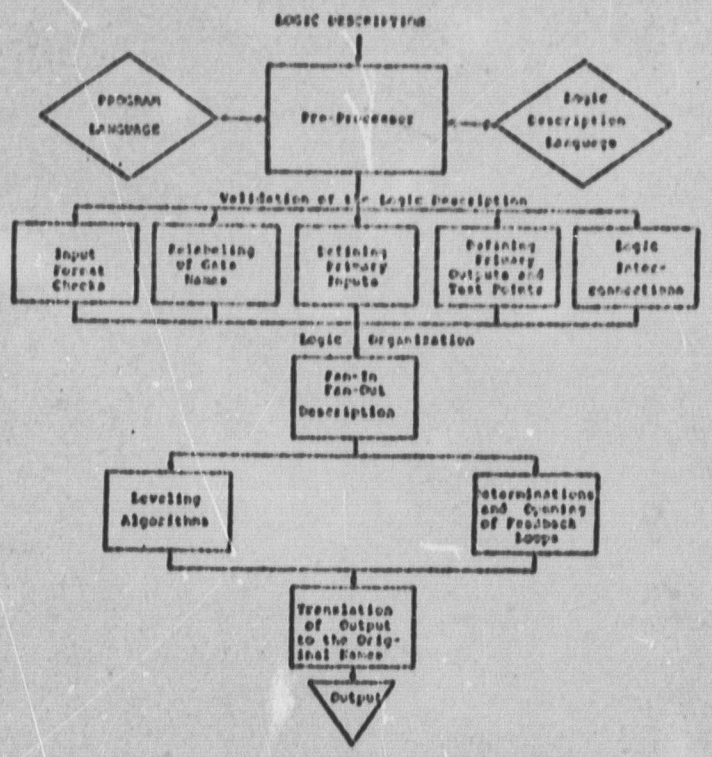



Fig. 2 Pre-Processor for Logic Simulation

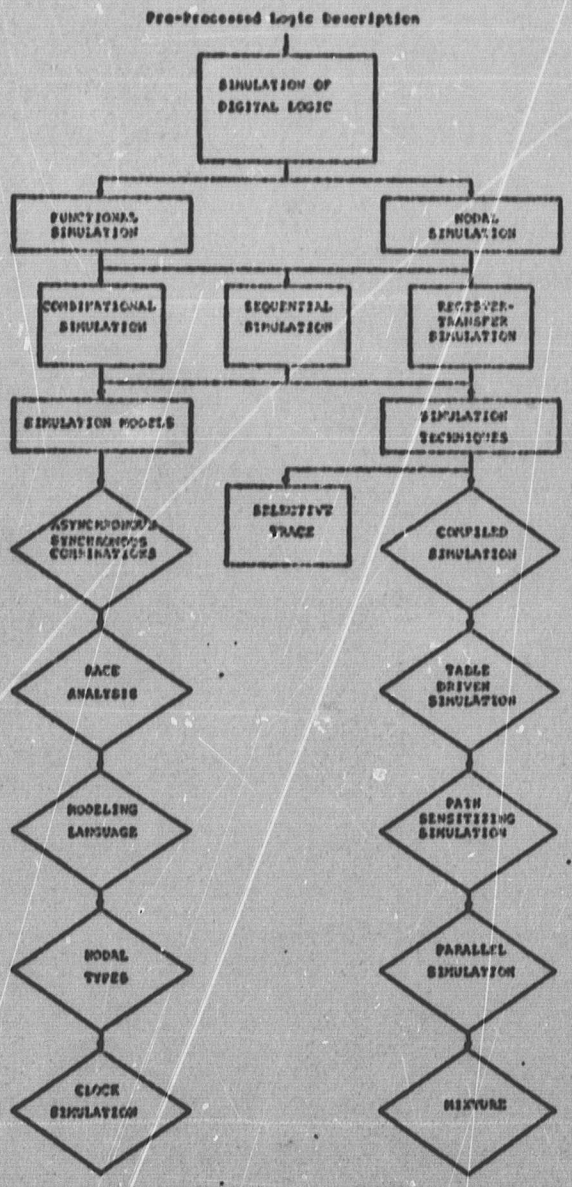



Fig. 3 SIMULATION OF LARGE DIGITAL NETS

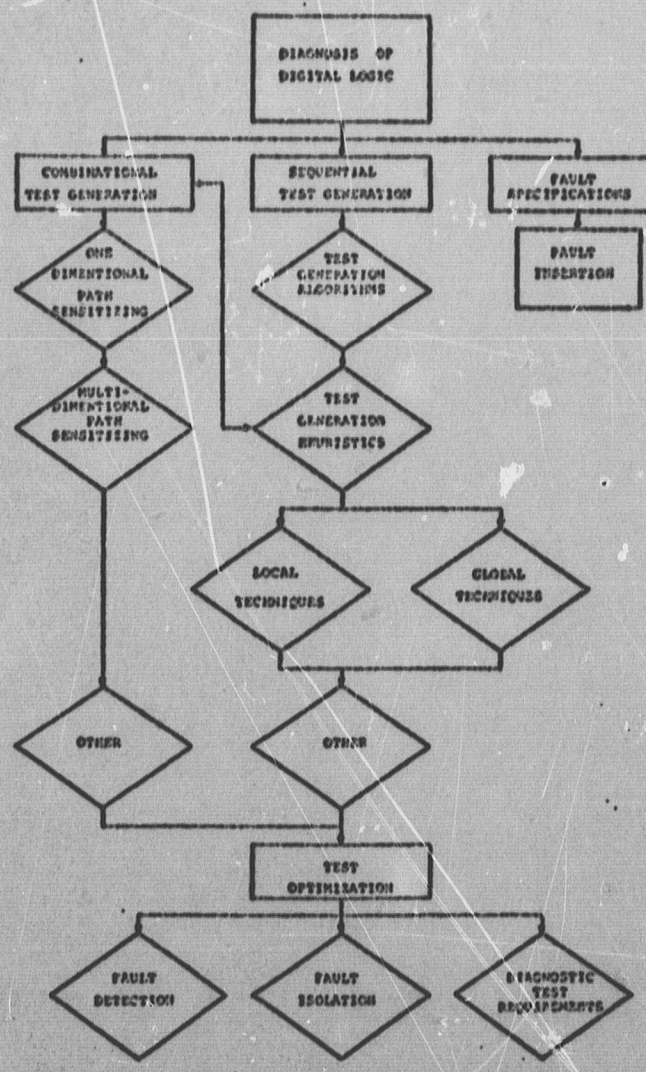



Fig. 4 DIAGNOSIS OF LARGE DIGITAL NETS

```
C
C
C  MULTI-DIMENSIONAL PATH SENSITIZING PROGRAM FOR COMBINATIONAL TEST
C  GENERATION
C
C
C
C       THIS PROGRAM IS AN IMPLEMENTATION OF A MULTI-DIMENSIONAL PATH
C     SENSITIZING ALGORITHM PROPOSED BY S.A.SZYGENDA AND G. GOLDBOGEN
C     AT THE 7|TH ANNUAL ALLERTON CONFERENCE ON CIRCUITS AND SYSTEMS
C     THEORY,OCT. 1969.
C
C       THE PROGRAM ATTEMPTS TO GENERATE TESTS FOR ANY NON REDUNDANT
C     FAULT IN A COMBINATIONAL LOGIC NET.  IF THE USER DOES NOT SPECIFY
C     A PARTICULAR SET OF FAULTS, THE PROGRAM WILL DERIVE THE COLAPSED
C     SET OF ALL POSSIBLE S-A-1 AND S-A-0 FAULTS. TESTS WILL THEN BE
C     GENERATED FOR THESE FAULTS.  THE GATE DESCRIPTION OF THE LOGIC
C     NET MUST BE LEVELED BEFORE IT IS INPUT TO THE PROGRAM.
C
C       THE REQUIRED INPUTS TO THE PROGRAM ARE AS FOLLOWS:(ALL VALUES
C     ARE RIGHT JUSTIFIED)
C     CARD 1; COL. 1-4=VNUMBER OF NETS TO BE RUN.
C     CARD2; COL. 1-4 = NUMBER OF LOGIC LEVELS.
C            COL. 5-8 = NUMBER OF PRIMARY INPUTS.
C            COL. 9-12= TOTAL NUMBER OF GATES(PRIMARY INPUTS AND OUTPUTS
C                ARE COUNTED AS GATES.)
C            COL. 13-16=NUMBER OF FAULTS IN SET BEING INPUT (0=PROG.GEN.)
C            COL. 17-20=NUMBER OF TESTS DESIRED FOR EACH FAULT (-1=ALL)
C            COL. 21-24=TRACE OF PATHS AND TABLES. 0=NO TRACE,1=COMPLETE
C                TRACE,2=TRACE OF PATHS.
C     CARD 3 BEGINS GATE DESCRIPTIONS, EACH GATE MUST BE DESCRIBED AS:
C            COL. 1-4 = GATE NUMBER.
C            COL. 5-8 = GATE TYPE (1=AND,2=OR,3=NAND,4=NOR,5=NOT,
C                6=INPUT,7=OUTPUT)
C            COL. 9-44= INPUTS TO THE GATE(IN GROUPS OF 4).
C            COL. 45-80=OUTPUTS OF THE GATES(IN GROUPS OF4).
C     IF ALL FAULTS ARE TO BE CONSIDERED, THE INPUTS ARE COMPLETE.
C     OTHERWISE,THE DESIRED SET OF FAULTS MUST BE SPECIFIED AS FOLLOWS:
C     CARD N;COL. 1-4 = MACHINE(FAULT) NUMBER (START WITH 2).
C            COL. 5-8 = GATE NUMBER.
C            COL. 9-12= SIGNAL NUMBER (FAULTY LEAD)
C            COL. 13-16=FAULT TYPE (1=S-A-1,0=S-A-0).
C        FUNCTION: GIVEN A CIRCUIT DESCRIPTION
C                  1) USING  GFT, GENERATE A FAULT TABLE  FAULT VERSION
C                  2) USING  INIT,GENERATE AN INITIAL CONDITION FOR FAULT I
C                  3) USING  CTG, GENERATE A TEST FOR FAULT I,INITIAL CONDITION
C           LAST CARD; COL. 1-4 = IF TH LAST NET HAS BEEN CONSIDERED
C     ENTER 001, IF ADDITIONAL LOGIC NETS ARE TO BE CONSICERED ENTER 999.
C     THIS GIVES THE USER THE CAPABILITY TO INTERMIX JOBS HAVING THE
C     FAULT TABLE GENERATED WITH JOBS SPECIFYING THE FAULT SET.
C
C
C
1       IMPLICIT INTEGER*2 (I-N)
2       COMMON/LINES/L(2000)
3       COMMON/MEM/MBUF(100,11),LB(400),LE(100),NB,NE,KPO,IR
4       COMMON ME(100,20),NET(300,4)
```

```
0005            COMMON/RETURN/NRET(20,2),MRET(20,2)
              C         READ  MF
0006            PRINT 740
0007          740 FORMAT(10X,'POSSIBLE TYPES OF GATES ARE; 1=AND,2=OR,3=NAND,4=NOR,5
                 1=NOT,6=INPUT,7=OUTPUT.')
0008            PRINT 741
0009          741 FORMAT(10X,'CODES FOR LINE VALUES IN SOLUTION ARE;0=X,1=1,2=0,3=D,
                14=-D.')
0010            PRINT 742
0011          742 FORMAT(10X,'BLIST IS THE LIST OF GATES WHOSE OUTPUT VALUES HAVE NO
                 1T BEEN BACKTRACED')
0012            PRINT 743
0013          743 FORMAT(10X,'ELIST IS THE LIST OF LINES HAVING VALUES D OR -D WHOSE
                 1 VALUES MAY BE TRACED FORWARD')
0014            PRINT 852
0015          852 FORMAT(10X,'D=1 FOR GOOD MACHINE, 0 FOR FAULTY MACHINE')
0016            PRINT 853
0017          853 FORMAT(10X,'-D=0 FOR THE GOOD MACHINE, 1 FOR THE FAULTY MACHINE')
0018            READ 100,NUMB
0019           10 DO 99 INUMB=1,NUMB
0020          100 FORMAT(20I4)
0021            READ  100,ML,NI,NG,NLAST,MULTY,KTRACE
0022            PRINT 720,ML,NI,NG
0023          720 FORMAT('1','NUM. OF LEVELS = ',I4,' NUM. OF PRIM INPUTS = ',I4,' N
                1UM. OF GATES = ',I4,/)
0024            IF(MULTY.GT.0)GO TO 501
0025            PRINT 502
0026          502 FORMAT(' ALL SOLUTIONS DESIRED',/)
0027            GO TO 503
0028          501 PRINT 504,MULTY
0029          504 FORMAT(' NUMBER OF SOLUTIONS DESIRED=',I4,/)
0030          503 IF(KTRACE.EQ.1)GO TO 507
0031          505 IF(KTRACE.EQ.0)GO TO 508
0032            PRINT 506
0033          506 FORMAT(' PATH TRACE REQUESTED',/)
0034            GO TO 525
0035          507 PRINT 509
0036          509 FORMAT(' COMPLETE TRACE REQUESTED',/)
0037            GO TO 525
0038          508 PRINT 510
0039          510 FORMAT(' NO TRACE REQUESTED',/)
0040          525 CONTINUE
              C         NL= NO. OF LEVELS
              C         NI= NO. OF INPUTS  (HIGHEST NUMBERER INPUT)
              C         NG= NO. OF GATES
              C         READ  CIRCUIT DESCRIPTION = MF
0041            PRINT 721
0042          721 FORMAT(' GATE TYPE',' -',13X,'9 INPUTS',12X,'--',10X,'9 OUTPUTS',1
                12X,'-')
0043            DO 1 I=1,NG
0044            READ 100,(MF(I,J),J=1,20)
0045            1 PRINT 100,(MF(I,J),J=1,20)
              C         GENERATE FAULT TABLE
0046            IF(NLAST.NE.0)GO TO 785
0047            CALL GFT2(NG,KLAST)
0048            GO TO 780
              C         PRINT  FAULT TABLE
0049          785 IJK=NLAST+1
```

```
0050                DO 790 I=1,IJK
0051                DO 790 J=1,4
0052          790 NFT(I,J)=0
0053                DO 787 I=1,NLAST
0054          787 READ 786,(NFT(I,J),J=1,4)
0055          786 FORMAT(4I4)
0056                KLAST=NLAST+1
0057          780 PRINT 120
0058          120 FORMAT(6X,/,' FAULT TABLE',//,' MACHINE NO.    GATE NUMBER   SIGNA
                 1L NUMBER   FAULT TYPE',/)
0059                DO 2 I=1,KLAST
0060            2 PRINT 101,(NFT(I,J),J=1,4)
0061          101 FORMAT(4(6X,I7))
0062                KF=0
              C          STEP THROUGH FAULT TABLE
0063          888 K=0
0064            5 K=K+1
0065              IF(NFT(K,1).EQ.0)GO TO 99
0066              NTYPE= 0
0067              KI=NI
0068              MULT=MULTY
0069              CALL CTG(NTYPE,KI,NG,MULT,NFT(K,1),KTRACE,K,NG)
0070              GO TO 5
0071           99 CONTINUE
0072              READ 100,ISP
0073              IF(ISP.NE.999)GO TO 999
0074              NUMB=1
0075              GO TO 10
0076          999 STOP
0077              END
```

```
0003          COMMON MF(100,20),NFT(300,4)
      C           GENERATE  FAULT  TABLE
      C           N= NUMBER OF GATES
      C           K= LAST ENTRY IN FAULT TABLE
      C           MF=MAIN FILE (CIRCUIT DESCRIPTION)
      C           NFT=FAULT TABLE
0004          KK=1
0005          K=0
0006          DO 99 I=1,N
      C           COUNT FI AND FO
0007          NFI = 0
0008          DO 19 J=1,9
0009          IF(MF(I,2+J).EQ.0)GO TO 20
0010       19 NFI = NFI+1
0011       20 NFO = 0
0012          DO 21 J=1,9
0013          IF(MF(I,11+J).EQ.0)GO TO 22
0014       21 NFO=NFO+1
0015       22 MMM=MF(I,2)
0016          GO TO(1,2,1,2,3,4,5),MMM
0017        1 KTT =1
0018          GO TO 6
0019        2 KTT = 0
0020        6 DO 29 J=1,NFI
0021          K=K+1
0022          KK=KK+1
0023          NFT(K,1)=KK
0024          NFT(K,2)= I
0025          NFT(K,3)=MF(I,J+2)
0026       29 NFT(K,4)=KTT
0027          GO TO 4
0028        3 IF(NFO.GT.1)GO TO 31
0029          MMM=MF(I,2)
0030          GO TO (7,8,7,8),MMM
0031        7 KT=1
0032          GO TO 33
0033        8 KT=0
0034          GO TO 33
0035        4 IF(NFO.EQ.1)GO TO 99
0036       31 K=K+1
0037          KK=KK+1
0038          NFT(K,1)=KK
0039          NFT(K,2)=I
0040          NFT(K,3)=I
0041          NFT(K,4)=0
0042          KT = 1
0043       33 K=K+1
0044          KK=KK+1
0045          NFT(K,1)=KK
0046          NFT(K,2)=I
0047          NFT(K,3)=I
0048          NFT(K,4)=KT
0049          GO TO 99
0050        5 MM = MF(I,3)
0051          K= K+1
0052          KK=KK+1
```

```
      NFT(K,1)=KK
      NFT(K,2)= I
      NFT(K,3)= MM
      NFT(K,4)= 0
      K = K+1
      KK=KK+1
      NFT(K,1)=KK
      NFT(K,2)=I
      NFT(K,3)=MM
      NFT(K,4)= 1
   99 CONTINUE
      K=K+1
      DO 4          4
   42 NFT(          )
      RETUR
      END
```

```
0001            SUBROUTINE INIT(NCH,N1,N2,N3,N4,KF)
0002            IMPLICIT INTEGER*2 (I-N)
0003            COMMON/LINES/L(2000)
0004            COMMON/MEM/MBUF(100,11),LB(400),LE(100),NB,NE,KPO,IR
0005            COMMON MF(100,20),NFT(300,4)
0006            COMMON/RETURN/NRET(20,2),MRET(20,2)
0007            DIMENSION NFI(9),NFO(9)
0008            NE=0
0009            NB=0
0010            KG=N2
0011            KS=N3
0012            DO 5 I=1,2000
0013          5 L(I)=0
0014            KPO=1
0015            IP= 1
0016            KFT=N4
0017            NGT=MF(KG,2)
0018            IF(NCH)2,89,50
0019          2 IF(NGT.EQ.6)GO TO 31
0020            IF(NGT.EQ.7)GO TO 9
0021            IF(NGT.EQ.5)GO TO 151
0022          1 GO TO (6,7,6,7,51),NGT
0023        151 NS=2-KFT
0024            GO TO 31
0025          6 NS = 1
0026            GO TO 9
0027          7 NS  = 2
0028          9 IF(NCH.GT.0)GO TO 50
0029            IF(KS.EQ.KG)GO TO 30
0030            NCH = 0
0031            DO 11 J=1,9
0032            MM = MF(KG,2+J)
0033            IF(MM.EQ.0)GO TO 13
0034            IF(MM.EQ.KS)GO TO 11
0035            L(MM)= NS
0036         11 CONTINUE
0037         13 L(KS)=3+KFT
0038            KMIN=0
0039            DO 14 J=1,9
0040            NFI(J)= MF(KG,J+2)
0041         14 NFO(J)= MF(KG,J+11)
0042            CALL TRACE(KG,NGT,KTT,NFI,NFO,KMIN)
0043            IF(NGT.EQ.7)GO TO 99
0044            IF(KTT.EQ.1)GO TO 15
0045         51 PRINT 201,KG,KS,KFT
0046        201 FORMAT(' ERROR TRAP',3I5)
0047            STOP
0048         15 L(KS)= L(KS)-2
0049            NB=0
0050            DO 16 J=1,9
0051            MM = MF(KG,J+2)
0052            IF(MM.EQ.0)GO TO 17
0053            NB= NB+1
0054         16 LB(NB)=MM
0055         17 NE = 0
0056            DO 18 J=1,9
0057            MM = MF(KG,J+11)
0058            IF(MM.EQ.0)GO TO 19
```

```
0059                IF(MM.LT.0)KPO = 2
0060                NE= NE+1
0061             13 LE(NE)= MM
0062             19 GO TO 70
0063             30 IF(KFT.EQ.1)GO TO 33
0064                GO TO (31,32,32,31),NGT
0065             33 GO TO (32,31,31,32),NGT
0066             31 NCH=0
0067                NB  = 0
0068                DO 35 J=1,9
0069                MM = MF(KG,J+2)
0070                IF(MM.EQ.0)GO TO 37
0071                L(MM)=NS
0072                NB=NB+1
0073             35 LB(NB)=MM
0074             37 NE = 0
0075                L(KG)=3+KFT
0076                DO 38 J=1,9
0077                MM = MF(KG,J+11)
0078                IF(MM.EQ.0)GO TO 39
0079                IF(MM.LT.0)KPO =2
0080                NE=NE+1
0081             38 LE(NE)=MM
0082             39 GO TO 70
0083             70 RETURN
                 C        COUNT NEI
0084             32 NEIN= 0
0085                L(KG)=KFT+3
0086                DO 48 J=1,9
0087                MM= MF(KG,J+2)
0088                IF(MM.EQ.0)GO TO 49
0089             48 NEIN =NEIN+1
0090             49 NCH = NEIN +1
0091             50 NCH = NCH-1
0092                L(KG)= 3+KFT
0093                IF(NCH.LE.0)GO TO 89
0094                K = NEIN+1 - NCH
0095                NE= 0
0096                MM = MF(KG,2+K)
0097                NB = 1
0098                LB(NB)= MM
0099                GO TO (54,53,54,53),NGT
0100             53 L(MM)= 1
0101                GO TO 55
0102             54 L(MM)= 2
0103             55 DO 56 J=1,9
0104                MM = MF(KG,J+11)
0105                IF(MM.EQ.0)GO TO 57
0106                NE = NE+1
0107                LE(NE)= MM
0108                IF(MM.LT.0)KPO =2
0109             56 CONTINUE
0110             57 GO TO 70
0111             99 RETURN
0112             89 NCH = -1
0113                RETURN
0114                END
```

```
0001                SUBROUTINE  CTG(NTP,KY,NGATES,MULT,MACHN,KTRACE,NV,NGQ2)
0002                IMPLICIT INTEGER*2 (I-N)
0003                COMMON/LINES/L(2000)
0004                COMMON/MEM/MBUF(100,11),LB(400),LE(100),NB,NE,KPO,IR
0005                COMMON MF(100,20),NFT(300,4)
0006                DIMENSION NFI(9),NFO(9)
0007              100 FORMAT(2014)
                C           CIRCUIT HAS BEEN INITIALIZED
                C           ALSO INITIALIZED BLIST,NB,ELIST,NE,LINES
                C           FIRST MBUF RECORD IS ALL ZEROS
0008                ITI=NTP
0009                IT2=KY
0010                IT3=NGATES
0011                IT5=MACHN
0012                IT6=KTRACE
0013                IT7=NV
0014                IT8=NGQ2
0015                KF=0
0016                MULTK=MULT
0017                KKSOL=0
0018                MACHN=NFT(NV,1)
0019                NCH=-1
0020                IPDCF=0
0021              620 IPDCF=IPDCF+1
0022                CALL INIT(NCH,NFT(NV,1),NFT(NV,2),NFT(NV,3),NFT(NV,4),KF)
                C           PRINT OUT  INITIAL CONDITION FOR MACHINE NFT(K,1)
0023                IF(NCH.LT.0)GO TO 982
0024                PRINT 121,NFT(NV,1)
0025              121 FORMAT(/////' +++ INITIAL CONDITIONS FOR MACHINE +++',2I4,///)
0026                PRINT 723,NB,NE,NCH
0027              723 FORMAT(3X,'NUM. OF GATES IN BLIST = ',I4,' NUM. OF GATES IN ELIST
                   1= ',I4,' NUM. OF REMAIN. PDCF = ',I4)
0028                PRINT 724
0029              724 FORMAT(3X,'BLIST')
0030                IF(NB.NE.0)PRINT 100,(LB(I),I=1,NB)
0031                PRINT 725
0032              725 FORMAT(3X,'ELIST')
0033                IF(NE.NE.0)PRINT 100,(LE(I),I=1,NE)
0034                PRINT 722
0035              722 FORMAT(3X,'INITIAL STATUS OF ALL LINES')
0036                PRINT 100,(L(I),I=1,NGQ2)
0037              303 CONTINUE
0038                IFL=0
                C           PICK LARGEST MEMBER FROM BLIST
                C           IF BLIST EMPTY GO TO CHECK IF SOLUTION WAS REACHED
0039               20 IF(NB.EQ.0)GO TO 30
0040                IP=0
0041               55 IN=0
0042                MAX=0
0043                DO 21 I=1,NB
0044                IF(LB(I).LE.MAX)GO TO 21
0045                IN=I
0046                MAX=LB(I)
0047               21 CONTINUE
                C           ERROR TRAP 1
0048                IF(IN.GT.0)GO TO 23
0049                IERR=1
0050               22 PRINT 102,IERR
```

```
      102 FORMAT(' MAINLINE ERROR TRAP',I4)
          GO TO 999
       23 NG= LB(IN)
C            IF  ALL BLIST ARE PI'S,BLIST IS SET TO ZERO,GO TO CHECK FOR  SOLUTION
          IF(NG.LE.KY)GO TO 29
C            TAKE NG OFF BLIST CLOSE UP LIST
          IF( IN.EQ.NB)GO TO 25
          IN1= IN+1
          DO 24 I=IN1,N5
       24 LB(I-1)=LB(I)
       25 NB =NB-1
C            ATTEMPT  TRACE BACK THRU NG
          NT=MF(NG,2)
          DO 26 I=1,9
       26 NFI(I)=MF(NG,I+2)
          CALL TRACB(NG,NT,NFI,IP,IKON)
C            IKON=1  NO CONFLICT     IKON=2 CONFLICT
          IF(KTRACE.EQ.0)GO TO 703
          IF( IKON.EQ.1)GO TO 701
          PRINT 702,NG
      702 FORMAT(' TRACE BACK THROUGH GATE',I5,' PATH IS NOT SUCCES.')
          GO TO 703
      701 PRINT 160,NG
      160 FORMAT(' TRACE BACK THRU GATE',I5,' PATH IS SUCCESSFUL')
      703 IF(IKON.EQ.2)GO TO 51
          IEL=0
          GO TO 20
       29 NB = 0
C            ONLY PI'S  LEFT ON BLIST
C            CHECK IS SOLUTION REACHED (IF SO KPD =2)
       30 IF(NIP.EQ.0)GO TO 330
          IF(NE.EQ.0)GO TO 32
          GO TO 31
      330 GO TO (31,32),KPD
C            SOLUTION IS REACHED
       32 KKSOL=KKSOL+1
          PRINT 201,KKSOL,MACHN
      201 FORMAT('  SOLUTION NUMBER ',I3,'   FOR MACHINE NUMBER ',I5)
          PRINT 210
      210 FORMAT(//' --SOLUTION (SEQUENTIAL VALUE OF LEADS)--')
          PRINT 100,(L(I),I=1,NGATES)
          PRINT 202
      202 FORMAT(/' INPUT SPECIFICATIONS')
          PRINT 745
      745 FORMAT('    GATE VALUE')
          DO 795 I=1,KY
          IKK=L(I)
          GO TO (710,711,712,713),IKK
      714 PRINT 715,I
      715 FORMAT(3X,I4,'   =X')
          GO TO 795
      710 PRINT 716,I
      716 FORMAT(3X,I4,'   =1')
          GO TO 795
      711 PRINT 717,I
      717 FORMAT(3X,I4,'   =0')
          GO TO 795
      712 PRINT 718,I
```

```
102          718 FORMAT(3X,I4,'    =D')
103              GO TO 795
104          713 PRINT 719,I
105          719 FORMAT(3X,I4,'    =-D')
106          795 CONTINUE
107              PRINT 203
108          203 FORMAT(/' OUTPUT SPECIFICATIONS')
109              PRINT 745
110              KI1=KY+1
111              DO 895 I=KI1,NGATES
112              IF (MF(I,2).NE.7)  GO TO 895
113              LLL=MF(I,3)
114              IKJ=L(LLL)
115              GO TO (810,811,812,813),IKJ
116          814 PRINT 715,I
117              GO TO 895
118          810 PRINT 716,I
119              GO TO 895
120          811 PRINT 717,I.
121              GO TO 895
122          812 PRINT 718,I
123              GO TO 895
124          813 PRINT 719,I
125          895 CONTINUE
126              IF(KKSOL.EQ.MULTK)GO TO 898
127              GO TO 993
128          898 KMULT=KKSOL
129              GO TO 989
             C            IF  NE IS 0 BEGIN BACK STEP
130           31 IF(NE.EQ.0)GO TO 22
             C            SELECT ELIST MIN  COUNT NUMBER OF TIME MIN APPEARS ON LIST
131          312 MIN=2001
132              IN = 0
133              KMIN=0
134              DO 34 I=1,NF
135              IF(LE(I)-MIN)35,36,34
136           35 KMIN=0
137              IN=I
138              MIN= LE(I)
139           36 KMIN=KMIN+1
140           34 CONTINUE
             C            REMOVE MIN FROM ELIST KMIN TIMES
141              IF(KMIN.EQ.NF)GO TO 776
142              IBACK=0
143              DO 37 I=1,NE
144              IF(LE(I).NE.MIN)GO TO 38
145              IBACK=IBACK+1
146              GO TO 37
147           38 LE(I-IBACK) = LE(I)
148           37 CONTINUE
149          776 NE=NF-KMIN
             C            TRACE  FORWARD
150              NG=MIN
151              NT= MF(NG,2)
152              DO 39 I=1,9
153              NFI(I)=MF(NG,I+2)
154           39 NFO(I)=MF(NG,I+11)
155              CALL TRACF(NG,NT,KT,NFI,NFO,KMIN)
```

```
C          KT=1  SUCCESSFUL    KT=2  NOT SUCCESSFUL
          IF(KT.EQ.1)GO TO 705
          IF(KTRACE.EQ.0)GO TO 707
          PRINT 706,NG
      706 FORMAT(30X,' TRACE FORWARD THRU GATE',I5,', PATH IS NOT SUCCES.')
          GO TO 707
      705 IF(KTRACE.EQ.0)GO TO 7039
          PRINT 161,NG
      161 FORMAT(30X,' TRACE FORWARD THRU GATE',I5,', PATH IS SUCCESSFUL')
     7039 CONTINUE
          GO TO(20,311),KT
      707 CONTINUE
      311 IF(NE.GT.0)GO TO 312
C             BACK  STEP  PROCEEDURE
C             PRINT PRESENT MFILE
       51 IF(KTRACE.NE.1)GO TO 61
          PRINT 205
      205 FORMAT(' MBUF  BEFORE BACK STEP')
          DO 53 I=1,IR
       53 PRINT 100,(MBUF(I,J),J=1,11)
C             IR POINT TO THE MBUF RECORD AVAILABLE TO BE WRITTEN INTO
       61 IR=IR-1
      661 IF(IR.EQ.0)GO TO 997
          IF(MBUF(IR,2).LE.0)GO TO 71
C             B RECORD
C             IN'S COME OFF BLIST  AND X  OUT  CORRESPONDING LINES
          IF(KTRACE.EQ.1)PRINT 405,MBUF(IR,1),MBUF(IR,2)
      405 FORMAT(' BRECORD FOR',I4,' WITH PREFERENCE',I4)
          DO 62 I=1,9
          NN = MBUF(IR,I+2)
          IF(NN.LE.0)GO TO 64
          L(NN)=0
          IF(NB.EQ.0)GO TO 62
          DO 63 J=1,NB
          IF(NN.EQ.LB(J))GO TO 65
       63 CONTINUE
          GO TO 64
       65 J1=J+1
          IF(J1.GT.NB)GO TO 67
          DO 66 J=J1,NB
       66 LB(J-1)=LB(J)
       67 NB=NB-1
       62 CONTINUE
C             PUT NG ON BLIST
       64 NB = NB+1
          LB(NB)= MBUF(IR,1)
          IF(MBUF(IR,2).EQ.10)GO TO 61
          IP  = MBUF(IR,2)
          GO TO 55
C             E  RECORD
C             TAKE NG'S FAN OUT OFF ELIST (ONCE) IF NG LINE = D OR DBAR
       71 NG = MBUF(IR,1)
          IF(KTRACE.EQ.1)PRINT 406,MBUF(IR,1),MBUF(IR,2)
      406 FORMAT(' ERECORD FOR',I4,' WITH PREFERENCE',I4)
          IF(MBUF(IR,2).EQ.0)GO TO 455
          DO 76 I=1,9
          NN = MF(NG,I+11)
          IF(NN.LT.0)KPO=1
```

```
0205                IF(NN.EQ.0)GO TO 775
0206                DO 72 J=1,NE
0207                IF(NN.EQ.LE(J))GO TO 73
0208             72 CONTINUE
0209             73 IF(J.EQ.NE)GO TO 76
0210                J1= J+1
0211                DO 77 J=J1,NE
0212             77 LE(J-1)=LE(J)
0213             76 NE= NE-1
           C              TAKE LN'S OFF BLIST AND X OUT LINES
0214            775 DO 82 I=1,9
0215                NN= MBUF(IR,I+2)
0216                IF(NN.EQ.0)GO TO 84
0217                L(NN)=0
0218                DO 83 J=1,NB
0219                IF(NN.EQ.LB(J))GO TO 85
0220             83 CONTINUE
0221                GO TO 82
0222             85 IF(J.EQ.NB)GO TO 87
0223                J1 =J+1
0224                DO 86 J=J1,NB
0225             86 LB(J-1)=LB(J)
0226             87 NB=NB-1
0227             82 CONTINUE
           C              X OUT L(NG)
0228             84 L(NG)=0
0229                IF(NE.EQ.0)GO TO 75
           C              WRITE  O PREF MBUF RECORD
0230                MBUF(IR,1)=NG
0231                MBUF(IR,3)=MBUF (IR,2)
0232                MBUF(IR,2)=0
0233                DO 444 I=4,11
0234            444 MBUF(IR,I)=0
0235                IR=IR+1
0236                GO TO 31
0237             75 IQA=-MBUF(IR,2)
0238                DO 476 IQB=1,IQA
0239                NE= NE+1
0240            476 LE(NE)=NG
0241                GO TO 61
0242            455 KKM =-NBUF(IR,3)
0243                DO 456 I=1,KKM
0244                NE=NE+1
0245            456 LE(NE)=NG
0246                L(NG)=0
0247                GO TO 61
0248            993 IF(MULT.EQ.0)GO TO 999
0249                MULT=MULT-1
0250                GO TO 51
0251            997 IF(MULT.NE.MULTK)GO TO 981
0252                PRINT 996,IPDCF
0253            996 FORMAT(' NO SOLUTION FOR PDCF NUMBER ',I3)
0254                GO TO 6200
0255            981 IF(MULTK.GT.0)GO TO 620
0256                PRINT 793,IPDCF
0257            793 FORMAT(' ALL SOLUTIONS FOUND FOR PDCF NUMBER ',I3)
0258           6200 KY=II2
0259                NTP=III
```

```
      NGATES=II3
      MACHN=II5
      KTRACE=II6
      NV=II7
      NGO2=II8
      GO TO 620
  982 IF(MULTK.LT.0)GO TO 999
      KMULT = MULTK-MULT
  989 PRINT 994,MULTK,KMULT
  994 FORMAT(' SOLUTIONS REQUESTED',I3,'          SOLUTIONS FOUND',I4)
  999 CONTINUE
      RETURN
      END
```

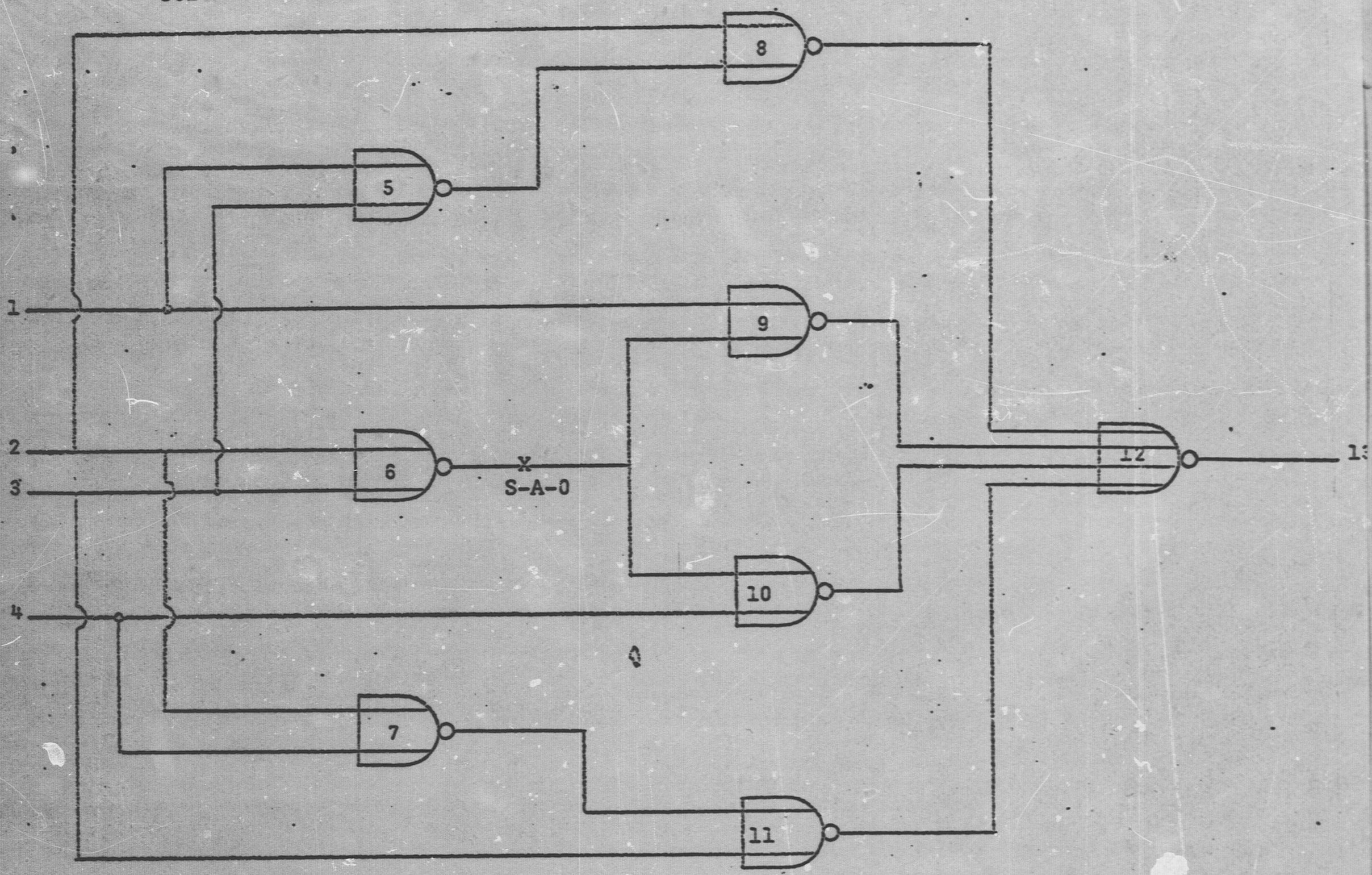
```
0001            SUBROUTINE TRACB(NG,NT,NFI,IP,IKON)
0002            IMPLICIT INTEGER*2 (I-N)
                C     BACK TRACE
                CC    IP= PRECIDENCE      NFI = FI ARRAY      NG = GATE NUMBER
                C     NT= GATE TYPE       IR = MBUF ROW       IKON= CONFLICT FLAG
                C     ICL=CLIST ARRAY     NCL=ICL INDEX
0003            COMMON/LINES/L(2000)
0004            COMMON/MEM/MBUF(100,11),LB(400),LE(100),NB,NE,KPO,IR
0005            DIMENSION NFI(9),IVNCH(4),IVCH(4),NS(9),MC(9)
0006            DATA IVNCH/1,2,1,2/,IVCH/2,1,2,1/
0007            KC=0
0008            NKC=0
0009            NCL=0
0010            LN = L(NG)
0011            LL  = LN+1
0012            MBUF(IR,1)= NG
0013            MBUF(IR,2)=10
0014            DO 9 I=1,9
0015          9 MBUF(IR,I+2)=0
0016            IKON=1
0017            GO TO (3,4,4,3,3),LL
0018          4 GO TO (1,1,1,1,2,3,40),NT
0019          1 IVA=IVCH(NT)
0020            IVB=IVNCH(NT)
0021            GO TO (5,6),LN
0022          5 GO TO (30,20,20,30),NT
0023          6 GO TO (20,30,30,20),NT
0024          2 NN= NFI(1)
0025            NNL  = L(NN)
0026            IF(NNL) 7,8,7
0027          7 IF(NNL.EQ.(3-LN))RETURN
0028            GO TO 41
0029         40 NN=NFI(1)
0030            IF(L(NN).NE.L(NG))GO TO 441
0031            RETURN
0032        441 IF(L(NN).NE.0)GO TO 41
0033            L(NN)=L(NG)
0034            MBUF(IR,3)=NN
0035            IR=IR+1
0036            NB=NB+1
0037            LB(NB)=NN
0038            RETURN
0039          8 MBUF(IR,3)= NN
0040            NB=NB+1
0041            LB(NB)=NN
0042            L(NN)=3-LN
0043            IR=IR+1
0044            RETURN
0045          3 CONTINUE
0046            GO TO 41
                C     PREFENCE  CHOICES
0047         20 CONTINUE
0048            DO 21 I=1,9
0049            NN = NFI(I)
0050            IF(NN.EQ.0)GO TO 22
0051            IF(L(NN).EQ.0)GO TO 23
0052         24 IF(L(NN).EQ.IVA)RETURN
0053            NKC=NKC+1
```

```
0054                GO TO 21
0055             23 KC=KC+1
0056                NS(KC)=NN
0057             21 CONTINUE
0058             22 IF(KC)3,27,26
                C                CONFLICT ALL VALUES IVB
0059             27 IF(IP)3,3,41
0060             41 IKON=2
0061                IR=IR+1
0062                RETURN
0063             26 IP=IP+1
0064                IF(KC.LT.IP)GO TO 41
0065                MBUF(IR,2)=IP
0066                NSP=NS(IP)
0067                MBUF(IR,3)=NSP
0068                NB= NB+1
0069                LB(NB)=NSP
0070                L(NSP)=IVA
0071                IR= IR+1
0072                RETURN
0073             30 DO 31 I=1,9
0074                NN = NFI(I)
0075                IF(NN)3,32,33
0076             33 IN= L(NN)+1
0077                GO TO (34,35,35,3,3),IN
0078             35 IF(IN-1.EQ.IVA)GO TO 41
0079                GO TO 31
0080             34 KC=KC+1
0081                NS(KC)=NN
0082             31 CONTINUE
0083             32 IF(KC.EQ.0)RETURN
0084                DO 37 I=1,KC
0085                NN=NS(I)
0086                MBUF(IR,2+I)=NN
0087                NB=NB+1
0088                LB(NB)=NN
0089             37 L(NN)= IVB
0090                IR=IR+1
0091                RETURN
0092                END
```

```
0001            SUBROUTINE TRACF(NG,NT,KT,NFI,NFO,KMIN)
0002            IMPLICIT INTEGER*2 (I-N)
                C       FORWARD  TRACE      KPO= PO REACHED FLAG
                C       NG = GATE NUMBER    NT = GATE TYPE       KT = RETURN KEY  1 SUCCESSFUL
                C       NFI= FAN-IN ARRAY   IR = MBUF ROW INDEX NFO= FANOUT       2 TERMINATE
                C       IV = ALLOWABLE IN   JV = FORCE OUT      NS = SAVE ARRAY X-VALUED GATES
                C       LB = BLIST          LE = ELIST          NB,NE =BLIST + ELIST INDICES
0003            COMMON/LINES/L(2000)                                                  TF000020
0004            COMMON/MEM/MBUF(100,11),LB(400),LE(100),NB,NE,KPO,IR
0005            DIMENSION NFI(9),NS(9),IV(4),JV(4),NFO(9)                             TF000040
0006            DATA IV/1,2,1,2/,JV/2,1,1,2/
0007            KPO=1                                                                 TF000055
0008            KT = 1                                                                TF000060
0009            MBUF(IR,1)=NG                                                         TF000070
0010            MBUF(IR,2)=-KMIN
0011            DO 1 J=3,11                                                           TF000090
0012          1 MBUF(IR,J)=0                                                          TF000100
0013            IF(NT.GT.4)GO TO 21                                                   TF000110
0014            IVA = IV(NT)                                                          TF000120
0015            KL  = -1                                                              TF000130
                C       KL IS SET TO D OR DBAR WHEN INPUT LINE WITH D OR DBAR IS FOUND
0016            J=0                                                                   TF000140
                C       J COUNTS INPUT LINE WITH VALUE X
0017            I=0                                                                   TF000150
                C       I COUNTS INPUT LINES
0018          2 I=I+1                                                                 TF000160
0019            KI = NFI(I)                                                           TF000170
0020            IF(KI.EQ.0)GO TO 7                                                    TF000180
0021            LK = L(KI)                                                            TF000190
0022            IF(LK.GT.2)GO TO 3                                                    TF000200
0023            IF(LK.EQ.IVA)GO TO 2                                                  TF000210
0024            IF(LK.NE.0)GO TO 6                                                    TF000220
0025            J= J+1                                                                TF000230
0026            NS(J)=KI                                                              TF000240
0027            GO TO 2                                                               TF000250
0028          3 IF(KL)4,5,5                                                           TF000260
0029          4 KL = LK                                                               TF000270
0030            GO TO 2                                                               TF000280
0031          5 IF(LK.EQ.KL)GO TO 2                                                   TF000290
0032          6 L(NG)= JV(NT)                                                         TF000300
0033            KT = 2                                                                TF000310
0034            GO TO 38
0035          7 IF(I.EQ.1)GO TO 11                                                    TF000330
0036            IF(J.EQ.0)GO TO 9                                                     TF000340
0037            DO 8 JJ=1,J                                                           TF000350
0038            NJ = NS(JJ)                                                           TF000360
0039            L(NJ)= IVA                                                            TF000370
0040          8 MBUF(IR,JJ+2)=NJ                                                      TF000380
0041          9 IF(KL.LT.1)GO TO 10                                                   TF000400
                C       GATE HAS SOME INPUTS = IVA AND SOME = KL
0042            IF(NT.LT.3)L(NG)= KL                                                  TF000420
0043            IF(NT.GT.2)L(NG)= 7-KL                                                TF000430
0044            GO TO 31                                                              TF000445
                C       GATE HAS ALL INPUTS = IVA
0045         10 IF(NT.LT.3)L(NG)= IVA                                                 TF000450
0046            IF(NT.GT.2)L(NG)= 3-IVA                                               TF000460
0047            KT= 2                                                                 TF000470
0048            GO TO 38
```

POSSIBLE TYPES OF GATES ARE: 1=AND, 2=OR, 3=NAND, 4=NOR, 5=NOT, 6=INPUT, 7=OUTPUT.
CODES FOR LINE VALUES IN SOLUTION ARE: 0=X, 1=1, 2=0, 3=D, 4=-D.
BLIST IS THE LIST OF GATES WHOSE OUTPUT VALUES HAVE NOT BEEN BACKTRACED
FLIST IS THE LIST OF LINES HAVING VALUES D OR -D WHOSE VALUES MAY BE TRACED FORWARD
D=1 FOR GOOD MACHINE, 0 FOR FAULTY MACHINE
-D=0 FOR THE GOOD MACHINE, 1 FOR THE FAULTY MACHINE

SOLUTION OF EXAMPLE CIRCUIT



NUM. OF LEVELS = 5 NUM. OF PRIM INPUTS = 4 NUM. OF GATES = 13

ALL SOLUTIONS DESIRED

NO TRACE REQUESTED

| GATE TYPE | 9 INPUTS | | | | | | | | | 9 OUTPUTS | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 | 7 | 8 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 6 | 11 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 4 | 1 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 4 | 2 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 9 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 4 | 2 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 4 | 2 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 4 | 1 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 4 | 4 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 11 | 4 | 3 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12 | 4 | 8 | 9 | 10 | 11 | 0 | 0 | 0 | 0 | -13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 13 | 7 | 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

FAULT TABLE

| MACHINE NO. | GATE NUMBER | SIGNAL NUMBER | FAULT TYPE |
|---|---|---|---|
| 2 | 1 | 1 | 0 |
| 3 | 1 | 1 | 1 |
| 4 | 2 | 2 | 0 |
| 5 | 2 | 2 | 1 |
| 6 | 3 | 3 | 0 |
| 7 | 3 | 3 | 1 |
| 8 | 4 | 4 | 0 |
| 9 | 4 | 4 | 1 |
| 10 | 5 | 1 | 0 |
| 11 | 5 | 3 | 0 |
| 12 | 6 | 2 | 0 |
| 13 | 6 | 3 | 0 |
| 14 | 6 | 6 | 0 |
| 15 | 6 | 6 | 1 |
| 16 | 7 | 2 | 0 |
| 17 | 7 | 4 | 0 |
| 18 | 8 | 2 | 0 |
| 19 | 8 | 5 | 0 |
| 20 | 9 | 1 | 0 |
| 21 | 9 | 6 | 0 |
| 22 | 10 | 4 | 0 |
| 23 | 10 | 6 | 0 |
| 24 | 11 | 3 | 0 |
| 25 | 11 | 7 | 0 |
| 26 | 12 | 8 | 0 |
| 27 | 12 | 9 | 0 |
| 28 | 12 | 10 | 0 |
| 29 | 12 | 11 | 0 |
| 30 | 13 | 12 | 0 |
| 31 | 13 | 12 | 1 |
| 0 | 0 | 0 | 0 |

```
+++ INITIAL CONDITIONS F  MACHINE +++    2
  NUM. OF GATES IN BLIST =     0 NUM. OF GATES IN ELIST =     2 NUM. OF REMAIN. PDCF =     0
  BLIST
  ELIST
  5  9
  INITIAL STATUS OF ALL LINES
  3  0  C  C  0  0  0  0  0  0  0  0  0
 SOLUTION NUMBER   1   FOR MACHINE NUMBER     2

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  3  2  2  2  4  1  1  3  2  2  2  4  0

INPUT SPECIFICATIONS
   GATE VALUE
     1   =D
     2   =0
     3   =0
     4   =0

OUTPUT SPECIFICATIONS
   GATE VALUE
    13   =-C
 SOLUTION NUMBER   2   FOR MACHINE NUMBER     2

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  3  1  1  1  0  2  0  2  4  2  2  3  0

INPUT SPECIFICATIONS
   GATE VALUE
     1   =D
     2   =1
     3   =1
     4   =1

OUTPUT SPECIFICATIONS
   GATE VALUE
    13   =D
 SOLUTION NUMBER   3   FOR MACHINE NUMBER     2

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  3  1  1  1  0  2  0  2  4  2  2  3  0

INPUT SPECIFICATIONS
   GATE VALUE
     1   =D
     2   =1
     3   =1
     4   =1

OUTPUT SPECIFICATIONS
   GATE VALUE
    13   =D
ALL SOLUTIONS FOUND FOR PDCF NUMBER   1
```

+++ INITIAL CONDITIONS FO MACHINE +++ 3
NUM. OF GATES IN BLIST = 0 NUM. OF GATES IN ELIST = 2 NUM. OF REMAIN. PDCF = 0
BLIST
ELIST
5 9
INITIAL STATUS OF ALL LINES
4 C 0 C C 0 C C 0 0 C 0 0
SOLUTION NUMBER 1 FOR MACHINE NUMBER 3

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
4 2 2 2 3 1 1 4 2 2 2 3 0

INPUT SPECIFICATIONS
GATE VALUE
1 =-D
2 =0
3 =0
4 =0

OUTPUT SPECIFICATIONS
GATE VALUE
13 =D
SOLUTION NUMBER 2 FOR MACHINE NUMBER 3

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
4 1 1 1 0 2 0 2 3 2 2 4 0

INPUT SPECIFICATIONS
GATE VALUE
1 =-D
2 =1
3 =1
4 =1

OUTPUT SPECIFICATIONS
GATE VALUE
13 =-D
SOLUTION NUMBER 3 FOR MACHINE NUMBER 3

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
4 1 1 1 0 2 C 2 3 2 2 4 0

INPUT SPECIFICATIONS
GATE VALUE
1 =-D
2 =1
3 =1
4 =1

OUTPUT SPECIFICATIONS
GATE VALUE
13 =-D
ALL SOLUTIONS FOUNC FOR PDCF NUMBER 1

+++ INITIAL CONDITIONS F[illegible] MACHINE +++ 4
NUM. OF GATES IN BLIST = 0 NUM. OF GATES IN ELIST = 3 NUM. OF REMAIN. PDCF = 0
BLIST
ELIST
6 7 8
INITIAL STATUS OF ALL LINES
0 3 0 C 0 0 0 C 0 0 0 0 0
SOLUTION NUMBER 1 FOR MACHINE NUMBER 4

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
2 3 2 2 1 4 4 2 3 3 3 4 0

INPUT SPECIFICATIONS
GATE VALUE
1 =0
2 =0
3 =0
4 =0

OUTPUT SPECIFICATIONS
GATE VALUE
13 =-0
SOLUTION NUMBER 2 FOR MACHINE NUMBER 4

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
1 3 1 1 2 0 0 4 2 2 2 3 0

INPUT SPECIFICATIONS
GATE VALUE
1 =1
2 =D
3 =1
4 =1

OUTPUT SPECIFICATIONS
GATE VALUE
13 =D
SOLUTION NUMBER 3 FOR MACHINE NUMBER 4

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
1 3 1 1 2 C 0 4 2 2 2 3 0

INPUT SPECIFICATIONS
GATE VALUE
1 =1
2 =D
3 =1
4 =1

OUTPUT SPECIFICATIONS
GATE VALUE
13 =D
ALL SOLUTIONS FOUND FOR PDCF NUMBER 1

+++ INITIAL CONDITIONS FOR MACHINE +++ 5
NUM. OF GATES IN BLIST = 0 NUM. OF GATES IN ELIST = 3 NUM. OF REMAIN. PDCF = 0
BLIST
ELIST
6 7 8
INITIAL STATUS OF ALL LINES
0 4 0 0 0 0 0 0 0 0 0 0 0
SOLUTION NUMBER 1 FOR MACHINE NUMBER 5

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
2 4 2 2 1 3 3 2 4 4 4 3 0

INPUT SPECIFICATIONS
GATE VALUE
1 =0
2 =-D
3 =0
4 =0

OUTPUT SPECIFICATIONS
GATE VALUE
13 =D
SOLUTION NUMBER 2 FOR MACHINE NUMBER 5

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
1 4 1 1 2 0 0 3 2 2 2 4 0

INPUT SPECIFICATIONS
GATE VALUE
1 =1
2 =-D
3 =1
4 =1

OUTPUT SPECIFICATIONS
GATE VALUE
13 =-D
SOLUTION NUMBER 3 FOR MACHINE NUMBER 5

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
1 4 1 1 2 0 0 3 2 2 2 4 0

INPUT SPECIFICATIONS
GATE VALUE
1 =1
2 =-D
3 =1
4 =1

OUTPUT SPECIFICATIONS
GATE VALUE
13 =-D
ALL SOLUTIONS FOUND FOR PDCF NUMBER 1

```
   5    6   11
  INITIAL STATUS OF ALL LINES
  0    0    3    0    0    0    0    0    0    0    C    C    0
 SOLUTION NUMBER    1    FOR MACHINE NUMBER      6

 --SOLUTION (SEQUENTIAL VALUE OF LEADS)--
   2    2    3    2    4    4    1    3    3    3    2    4    0

 INPUT SPECIFICATIONS
    GATE VALUE
       1    =0
       2    =0
       3    =D
       4    =0

 OUTPUT SPECIFICATIONS
    GATE VALUE
      13    =-D
 SOLUTION NUMBER    2    FOR MACHINE NUMBER      6

 --SOLUTION (SEQUENTIAL VALUE OF LEADS)--
   1    1    3    1    0    0    2    2    2    2    4    3    0

 INPUT SPECIFICATIONS
    GATE VALUE
       1    =1
       2    =1
       3    =D
       4    =1

 OUTPUT SPECIFICATIONS
    GATE VALUE
      13    =D
 SOLUTION NUMBER    3    FOR MACHINE NUMBER      6

 --SOLUTION (SEQUENTIAL VALUE OF LEADS)--
   1    1    3    1    0    0    2    2    2    2    4    3    0

 INPUT SPECIFICATIONS
    GATE VALUE
       1    =1
       2    =1
       3    =D
       4    =1

 OUTPUT SPECIFICATIONS
    GATE VALUE
      13    =D
 ALL SOLUTIONS FOUND FOR PROF NUMBER    1
```

```
+++ INITIAL CONDITIONS FOR MACHINE +++    7
  NUM. OF GATES IN BLIST =     0 NUM. OF GATES IN ELIST =     3 NUM. OF REMAIN. PDCF =     0
  BLIST
  ELIST
  5   6  11
  INITIAL STATUS OF ALL LINES
  0   0   4   0   0   0   0   0   0   0   0   0   0
 SOLUTION NUMBER    1    FOR MACHINE NUMBER      7

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  2   2   4   2   3   3   1   4   4   4   2   3   0

 INPUT SPECIFICATIONS
    GATE VALUE
      1    =0
      2    =0
      3    =-D
      4    =0

 OUTPUT SPECIFICATIONS
    GATE VALUE
     13    =D
 SOLUTION NUMBER    2    FOR MACHINE NUMBER      7

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  1   1   4   1   0   0   2   2   2   2   3   4   0

 INPUT SPECIFICATIONS
    GATE VALUE
      1    =1
      2    =1
      3    =-D
      4    =1

 OUTPUT SPECIFICATIONS
    GATE VALUE
     13    =-D
 SOLUTION NUMBER    3    FOR MACHINE NUMBER      7

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  1   1   4   1   0   0   2   2   2   2   3   4   0

 INPUT SPECIFICATIONS
    GATE VALUE
      1    =1
      2    =1
      3    =-D
      4    =1

 OUTPUT SPECIFICATIONS
    GATE VALUE
     13    =-D
 ALL SOLUTIONS FOUND FOR PDCF NUMBER    1
```

```
+++ INITIAL CONDITIONS F  MACHINE +++    8
  NUM. OF GATES IN BLIST =     0 NUM. OF GATES IN ELIST =     2 NUM. OF REMAIN. PDCF =     0
  BLIST
  ELIST
   7  10
  INITIAL STATUS OF ALL LINES
  0   0   0   3   0   0   0   C   0   0   0   0   0
 SOLUTION NUMBER   1   FOR MACHINE NUMBER     8

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  2   2   2   3   1   1   4   2   2   2   3   4   0

INPUT SPECIFICATIONS
   GATE VALUE
     1   =0
     2   =0
     3   =0
     4   =0

OUTPUT SPECIFICATIONS
   GATE VALUE
    13   =-0
 SOLUTION NUMBER   2   FOR MACHINE NUMBER     8

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  1   1   1   3   0   2   0   2   2   4   2   3   0

INPUT SPECIFICATIONS
   GATE VALUE
     1   =1
     2   =1
     3   =1
     4   =0

OUTPUT SPECIFICATIONS
   GATE VALUE
    13   =0
 SOLUTION NUMBER   3   FOR MACHINE NUMBER     8

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  1   1   1   3   0   2   0   2   2   4   2   3   0

INPUT SPECIFICATIONS
   GATE VALUE
     1   =1
     2   =1
     3   =1
     4   =0

OUTPUT SPECIFICATIONS
   GATE VALUE
    13   =0
 ALL SOLUTIONS FOUND FOR PDCF NUMBER   1
```

```
+++ INITIAL CONDITIONS FOR MACHINE +++   9
  NUM. OF GATES IN BLIST =     0 NUM. OF GATES IN ELIST =     2 NUM. OF REMAIN. PDCF =     0
  BLIST
  ELIST
  7  10
  INITIAL STATUS OF ALL LINES
  0   0   0   4   C   0   C   C   0   0   0   0   0.
 SOLUTION NUMBER    1    FOR MACHINE NUMBER      9

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  2   2   2   4   1   1   3   2   2   2   4   3   0

INPUT SPECIFICATIONS
   GATE VALUE
     1    =0
     2    =0
     3    =0
     4    =-D

OUTPUT SPECIFICATIONS
   GATE VALUE
    13    =D
 SOLUTION NUMBER    2    FOR MACHINE NUMBER      9

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  1   1   1   4   C   2   0   2   2   3   2   4   0

INPUT SPECIFICATIONS
   GATE VALUE
     1    =1
     2    =1
     3    =1
     4    =-D

OUTPUT SPECIFICATIONS
   GATE VALUE
    13    =-D
 SOLUTION NUMBER    3    FOR MACHINE NUMBER      9

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  1   1   1   4   0   2   0   2   2   3   2   4   0

INPUT SPECIFICATIONS
   GATE VALUE
     1    =1
     2    =1
     3    =1
     4    =-D

OUTPUT SPECIFICATIONS
   GATE VALUE
    13    =-D
ALL SOLUTIONS FOUND FOR PDCF NUMBER    1
```

```
+++ INITIAL CONDITIONS FOR MACHINE +++  10
  NUM. OF GATES IN BLIST =     2 NUM. OF GATES IN ELIST =     1 NUM. OF REMAIN. PDCF =     0
  BLIST
  1   3
  ELIST
  8
  INITIAL STATUS OF ALL LINES
  1   0   2   0   4   C   0   C   0   0   C   0   0
 SOLUTION NUMBER    1   FOR MACHINE NUMBER     10

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  1   2   2   2   4   1   1   3   2   2   2   4   0

INPUT SPECIFICATIONS
   GATE VALUE
     1    =1
     2    =0
     3    =0
     4    =0

OUTPUT SPECIFICATIONS
   GATE VALUE
    13    =-0
ALL SOLUTIONS FOUND FOR PDCF NUMBER    1


+++ INITIAL CONDITIONS FOR MACHINE +++  11
  NUM. OF GATES IN BLIST =     2 NUM. OF GATES IN ELIST =     1 NUM. OF REMAIN. PDCF =     0
  BLIST
  1   2
  ELIST
  8
  INITIAL STATUS OF ALL LINES
  2   0   1   C   4   0   0   0   0   0   0   0   0
NO SOLUTION FOR PDCF NUMBER    1


+++ INITIAL CONDITIONS FOR MACHINE +++  12
  NUM. OF GATES IN BLIST =     2 NUM. OF GATES IN ELIST =     2 NUM. OF REMAIN. PDCF =     0
  BLIST
  2   3
  ELIST
  9  10
  INITIAL STATUS OF ALL LINES
  0   1   2   C   0   4   0   0   0   0   0   0   0
NO SOLUTION FOR PDCF NUMBER    1


+++ INITIAL CONDITIONS FOR MACHINE +++  13
  NUM. OF GATES IN BLIST =     2 NUM. OF GATES IN ELIST =     2 NUM. OF REMAIN. PDCF =     0
  BLIST
```

```
 2
 ELIST
 9  10
 INITIAL STATUS OF ALL LINES
 0  2  1  C  0  4  C  0  C  0  0  C  0
NO SOLUTION FOR PDCF NUMBER   1


+++ INITIAL CONDITIONS FOR MACHINE +++  14
 NUM. OF GATES IN BLIST =    2 NUM. OF GATES IN ELIST =    2 NUM. OF REMAIN. PDCF =    0
 BLIST
 2  3
 ELIST
 9  10
 INITIAL STATUS OF ALL LINES
 0  2  2  C  0  3  0  C  0  0  0  0  0
SOLUTION NUMBER   1   FOR MACHINE NUMBER    14

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
 2  2  2  2  1  3  1  2  4  4  2  3  0

INPUT SPECIFICATIONS
  GATE VALUE
    1   =0
    2   =0
    3   =0
    4   =0

OUTPUT SPECIFICATIONS
  GATE VALUE
   13   =0
ALL SOLUTIONS FOUND FOR PDCF NUMBER   1


+++ INITIAL CONDITIONS FOR MACHINE +++  15
 NUM. OF GATES IN BLIST =    1 NUM. OF GATES IN ELIST =    2 NUM. OF REMAIN. PDCF =    2
 BLIST
 2
 ELIST
 9  10
 INITIAL STATUS OF ALL LINES
 0  1  0  0  0  4  C  C  0  0  0  0  0
SOLUTION NUMBER   1   FOR MACHINE NUMBER    15

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
 2  1  1  2  0  4  0  2  3  3  2  4  0

INPUT SPECIFICATIONS
  GATE VALUE
    1   =0
    2   =1
    3   =1
    4   =0
```

```
OUTPUT SPECIFICATIONS
    GATE VALUE
     13    =-D
 SOLUTION NUMBER    2    FOR MACHINE NUMBER     15

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  2   1   1   1   0   4   0   2   3   2   2   4   0

INPUT SPECIFICATIONS
    GATE VALUE
      1    =0
      2    =1
      3    =1
      4    =1

OUTPUT SPECIFICATIONS
    GATE VALUE
     13    =-D
 SOLUTION NUMBER    3    FOR MACHINE NUMBER     15

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  1   1   1   2   0   4   0   2   2   3   2   4   0

INPUT SPECIFICATIONS
    GATE VALUE
      1    =1
      2    =1
      3    =1
      4    =0

OUTPUT SPECIFICATIONS
    GATE VALUE
     13    =-D
ALL SOLUTIONS FOUND FOR PDCF NUMBER    1


+++ INITIAL CONDITIONS FOR MACHINE +++   15
  NUM. OF GATES IN BLIST =     1 NUM. OF GATES IN FLIST =     2 NUM. OF REMAIN. PDCF =     1
   BLIST
   3
   FLIST
   9  10
   INITIAL STATUS OF ALL LINES
   0   0   1   0   0   4   0   0   0   0   0   0   0
 SOLUTION NUMBER    4    FOR MACHINE NUMBER     15

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  2   1   1   2   0   4   0   2   3   3   2   4   0

INPUT SPECIFICATIONS
    GATE VALUE
      1    =0
      2    =1
      3    =1
```

```
     4    =0'

OUTPUT SPECIFICATIONS
    GATE VALUE
      13    =-0
 SOLUTION NUMBER    5    FOR MACHINE NUMBER     15


--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
   2    1    1    1    0    4    0    2    3    2    2    4    0

INPUT SPECIFICATIONS
    GATE VALUE
      1    =0
      2    =1
      3    =1
      4    =1

OUTPUT SPECIFICATIONS
    GATE VALUE
      13    =-0
 SOLUTION NUMBER    6    FOR MACHINE NUMBER     15


--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
   1    1    1    2    0    4    0    2    2    3    2    4    0

INPUT SPECIFICATIONS
    GATE VALUE
      1    =1
      2    =1
      3    =1
      4    =0

OUTPUT SPECIFICATIONS
    GATE VALUE
      13    =-0
ALL SOLUTIONS FOUND FOR PDCF NUMBER    2


+++ INITIAL CONDITIONS FOR MACHINE +++   16
  NUM. OF GATES IN BLIST =     2 NUM. OF GATES IN ELIST =     1 NUM. OF REMAIN. PDCF =     0
  BLIST
   2    4
  ELIST
  11
  INITIAL STATUS OF ALL LINES
   0    1    0    2    0    0    4    0    0    0    0    0    0    0
NO SOLUTION FOR PDCF NUMBER    1


+++ INITIAL CONDITIONS FOR MACHINE +++   17
  NUM. OF GATES IN BLIST =     2 NUM. OF GATES IN ELIST =     1 NUM. OF REMAIN. PDCF =     0
  BLIST
   2    4
```

```
 FLIST
11
 INITIAL STATUS OF ALL LINES
 0  2  0  1  0  0  4  0  0  0  0  0  0
SOLUTION NUMBER   1   FOR MACHINE NUMBER    17

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
 2  2  2  1  1  1  4  2  2  2  3  4  0  .  .

INPUT SPECIFICATIONS
   GATE VALUE
     1   =0
     2   =0
     3   =0
     4   =1

OUTPUT SPECIFICATIONS
   GATE VALUE
    13   =-D
ALL SOLUTIONS FOUNC FOR PDCF NUMBER   1


+++ INITIAL CONDITIONS FOR MACHINE +++  18
 NUM. OF GATES IN BLIST =    2 NUM. OF GATES IN FLIST =    1 NUM. OF REMAIN. PDCF =    0
 BLIST
 2  5
 FLIST
12
 INITIAL STATUS OF ALL LINES
 0  1  0  0  2  0  0  4  0  0  0  0  0
SOLUTION NUMBER   1   FOR MACHINE NUMBER    18

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
 1  1  1  1  2  0  0  4  2  2  2  3  0

INPUT SPECIFICATIONS
   GATE VALUE
     1   =1
     2   =1
     3   =1
     4   =1

OUTPUT SPECIFICATIONS
   GATE VALUE
    13   =0
SOLUTION NUMBER   2   FOR MACHINE NUMBER    18

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
 1  1  1  1  2  0  0  4  2  2  2  3  0

INPUT SPECIFICATIONS
   GATE VALUE
     1   =1
     2   =1
     3   =1
```

```
     4    =1

OUTPUT SPECIFICATIONS
    GATE VALUE
      13    =0
ALL SOLUTIONS FOUND FOR PDCF NUMBER    1


+++ INITIAL CONDITIONS FOR MACHINE +++   19
  NUM. OF GATES IN BLIST =     2 NUM. OF GATES IN ELIST =     1 NUM. OF REMAIN. PDCF =     0
  BLIST
   2    5
  ELIST
  12
  INITIAL STATUS OF ALL LINES
   0    2    0    0    1    0    0    4    0    0    0    0    0    0
 SOLUTION NUMBER    1    FOR MACHINE NUMBER     19

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
   2    2    2    2    1    1    1    4    2    2    2    3    0

INPUT SPECIFICATIONS
    GATE VALUE
      1    =0
      2    =0
      3    =0
      4    =0

OUTPUT SPECIFICATIONS
    GATE VALUE
      13    =0
ALL SOLUTIONS FOUND FOR PDCF NUMBER    1


+++ INITIAL CONDITIONS FOR MACHINE +++   20
  NUM. OF GATES IN BLIST =     2 NUM. OF GATES IN ELIST =     1 NUM. OF REMAIN. PDCF =     0
  BLIST
   1    6
  ELIST
  12
  INITIAL STATUS OF ALL LINES
   1    0    0    0    0    2    0    0    4    0    0    0    0    0
 SOLUTION NUMBER    1    FOR MACHINE NUMBER     20

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
   1    1    1    1    0    2    0    2    4    2    2    3    0

INPUT SPECIFICATIONS
    GATE VALUE
      1    =1
      2    =1
      3    =1
      4    =1
```

OUTPUT SPECIFICATIONS
GATE VALUE
13 =0
SOLUTION NUMBER 2 FOR MACHINE NUMBER 20

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
1 1 1 1 0 2 0 2 4 2 2 3 0

INPUT SPECIFICATIONS
GATE VALUE
1 =1
2 =1
3 =1
4 =1

OUTPUT SPECIFICATIONS
GATE VALUE
13 =0
ALL SOLUTIONS FOUND FOR PDCF NUMBER 1

+++ INITIAL CONDITIONS FOR MACHINE +++ 21
NUM. OF GATES IN BLIST = 2 NUM. OF GATES IN FLIST = 1 NUM. OF REMAIN. PDCF = 0
BLIST
1 6
FLIST
12
INITIAL STATUS OF ALL LINES
2 0 0 0 0 1 0 0 4 0 0 0 0
SOLUTION NUMBER 1 FOR MACHINE NUMBER 21

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
2 2 2 2 1 1 1 2 4 2 2 3 0

INPUT SPECIFICATIONS
GATE VALUE
1 =0
2 =0
3 =0
4 =0

OUTPUT SPECIFICATIONS
GATE VALUE
13 =0
ALL SOLUTIONS FOUND FOR PDCF NUMBER 1

+++ INITIAL CONDITIONS FOR MACHINE +++ 22
NUM. OF GATES IN BLIST = 2 NUM. OF GATES IN FLIST = 1 NUM. OF REMAIN. PDCF = 0
BLIST
4 6
FLIST

```
 12
  INITIAL STATUS OF ALL LINES
  0   0   0   1   0   2   0   0   0   4   0   0   0
 SOLUTION NUMBER    1    FOR MACHINE NUMBER     22

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  1   1   1   1   0   2   0   2   2   4   2   3   0

INPUT SPECIFICATIONS
   GATE VALUE
     1    =1
     2    =1
     3    =1
     4    =1

OUTPUT SPECIFICATIONS
   GATE VALUE
    13    =0
 SOLUTION NUMBER    2    FOR MACHINE NUMBER     22

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  1   1   1   1   0   2   0   2   2   4   2   3   0

INPUT SPECIFICATIONS
   GATE VALUE
     1    =1
     2    =1
     3    =1
     4    =1

OUTPUT SPECIFICATIONS
   GATE VALUE
    13    =0
 ALL SOLUTIONS FOUND FOR PDCF NUMBER    1


+++ INITIAL CONDITIONS FOR MACHINE +++  23
  NUM. OF GATES IN BLIST =     2 NUM. OF GATES IN ELIST =     1 NUM. OF REMAIN. PDCF =     0
  BLIST
  4   6
  ELIST
 12
  INITIAL STATUS OF ALL LINES
  0   0   0   2   0   1   0   0   0   4   0   0   0
 SOLUTION NUMBER    1    FOR MACHINE NUMBER     23

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  2   2   2   2   1   1   1   2   2   4   2   3   0

INPUT SPECIFICATIONS
   GATE VALUE
     1    =0
     2    =0
     3    =0
     4    =0
```

```
OUTPUT SPECIFICATIONS
   GATE VALUE
     13   =0
ALL SOLUTIONS FOUND FOR PDCF NUMBER    1


+++ INITIAL CONDITIONS FOR MACHINE +++   24
  NUM. OF GATES IN BLIST =     2 NUM. OF GATES IN ELIST =     1 NUM. OF REMAIN. PDCF =     0
  BLIST
  3   7
  ELIST
 12
  INITIAL STATUS OF ALL LINES
  0   0   1   0   0   0   2   0   0   0   4   0   0
 SOLUTION NUMBER    1    FOR MACHINE NUMBER     24

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  1   1   1   1   0   0   2   2   2   2   4   3   0

INPUT SPECIFICATIONS
   GATE VALUE
     1    =1
     2    =1
     3    =1
     4    =1

OUTPUT SPECIFICATIONS
   GATE VALUE
     13   =0
 SOLUTION NUMBER    2    FOR MACHINE NUMBER     24

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  1   1   1   1   0   0   2   2   2   2   4   3   0

INPUT SPECIFICATIONS
   GATE VALUE
     1    =1
     2    =1
     3    =1
     4    =1

OUTPUT SPECIFICATIONS
   GATE VALUE
     13   =0
ALL SOLUTIONS FOUND FOR PDCF NUMBER    1


+++ INITIAL CONDITIONS FOR MACHINE +++   25
  NUM. OF GATES IN BLIST =     2 NUM. OF GATES IN ELIST =     1 NUM. OF REMAIN. PDCF =     0
  BLIST
  3   7
  ELIST
```

12
INITIAL STATUS OF ALL LINES
0 0 2 0 0 0 1 0 0 0 4 0 0
SOLUTION NUMBER 1 FOR MACHINE NUMBER 25

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
2 2 2 2 1 1 1 2 2 2 4 3 0

INPUT SPECIFICATIONS
GATE VALUE
1 =0
2 =0
3 =0
4 =0

OUTPUT SPECIFICATIONS
GATE VALUE
13 =0
ALL SOLUTIONS FOUND FOR PDCF NUMBER 1

+++ INITIAL CONDITIONS FOR MACHINE +++ 26
NUM. OF GATES IN BLIST = 4 NUM. OF GATES IN ELIST = 1 NUM. OF REMAIN. PDCF = 0
BLIST
8 9 10 11
ELIST
-13
INITIAL STATUS OF ALL LINES
0 0 0 0 0 0 0 1 2 2 2 4 0
SOLUTION NUMBER 1 FOR MACHINE NUMBER 26

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
1 2 1 1 2 0 0 1 2 2 2 4 0

INPUT SPECIFICATIONS
GATE VALUE
1 =1
2 =0
3 =1
4 =1

OUTPUT SPECIFICATIONS
GATE VALUE
13 =0
SOLUTION NUMBER 2 FOR MACHINE NUMBER 26

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
1 2 2 2 2 1 1 1 2 2 2 4 0

INPUT SPECIFICATIONS
GATE VALUE
1 =1
2 =0
3 =0
4 =0

```
OUTPUT SPECIFICATIONS
   GATE VALUE
     13    =-0
ALL SOLUTIONS FOUND FOR PDCF NUMBER    1


+++ INITIAL CONDITIONS FOR MACHINE +++  27
  NUM. OF GATES IN BLIST =     4 NUM. OF GATES IN ELIST =     1 NUM. OF REMAIN. PDCF =     0
  BLIST
  8   9  10  11
  ELIST
-13
  INITIAL STATUS OF ALL LINES
  0   0   0   0   0   0   0   2   1   2   2   4   0
 SOLUTION NUMBER    1    FOR MACHINE NUMBER     27

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  2   1   1   1   0   2   0   2   1   2   2   4   0

INPUT SPECIFICATIONS
   GATE VALUE
     1    =0
     2    =1
     3    =1
     4    =1

OUTPUT SPECIFICATIONS
   GATE VALUE
     13    =-D
ALL SOLUTIONS FOUND FOR PDCF NUMBER    1


+++ INITIAL CONDITIONS FOR MACHINE +++  28
  NUM. OF GATES IN BLIST =     4 NUM. OF GATES IN ELIST =     1 NUM. OF REMAIN. PDCF =     0
  BLIST
  8   9  10  11
  ELIST
-13
  INITIAL STATUS OF ALL LINES
  0   0   0   0   0   0   0   2   2   1   2   4   0
 SOLUTION NUMBER    1    FOR MACHINE NUMBER     28

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  1   1   1   2   0   2   0   2   2   1   2   4   0

INPUT SPECIFICATIONS
   GATE VALUE
     1    =1
     2    =1
     3    =1
     4    =0
```

```
OUTPUT SPECIFICATIONS
   GATE VALUE
    13   =-0
ALL SOLUTIONS FOUND FOR PDCF NUMBER    1


+++ INITIAL CONDITIONS FOR MACHINE +++  29
  NUM. OF GATES IN BLIST =     4 NUM. OF GATES IN ELIST =     1 NUM. OF REMAIN. PDCF =     0
  BLIST
   8   9  10  11
  ELIST
-13
  INITIAL STATUS OF ALL LINES
   0   0   0   0   0   0   0   2   2   2   1   4   0
 SOLUTION NUMBER    1   FOR MACHINE NUMBER     29

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
   1   1   2   1   0   0   2   2   2   2   1   4   0

INPUT SPECIFICATIONS
   GATE VALUE
    1   =1
    2   =1
    3   =0
    4   =1
OUTPUT SPECIFICATIONS
   GATE VALUE
    13   =-0
 SOLUTION NUMBER    2   FOR MACHINE NUMBER     29

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
   2   2   2   1   1   1   2   2   2   2   1   4   0

INPUT SPECIFICATIONS
   GATE VALUE
    1   =0
    2   =0
    3   =0
    4   =1
OUTPUT SPECIFICATIONS
   GATE VALUE
    13   =-0
 SOLUTION NUMBER    3   FOR MACHINE NUMBER     29

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
   2   2   2   0   1   1   2   2   2   2   1   4   0

INPUT SPECIFICATIONS
   GATE VALUE
    1   =0
    2   =0
    3   =0
    4   =X
```

```
OUTPUT SPECIFICATIONS
   GATE VALUE
    13   =-0
ALL SOLUTIONS FOUND FOR PDCF NUMBER    1


+++ INITIAL CONDITIONS FOR MACHINE +++   30
  NUM. OF GATES IN BLIST =     1 NUM. OF GATES IN FLIST =     1 NUM. OF REMAIN. PDCF =     0
  BLIST
 12
  FLIST
-13
  INITIAL STATUS OF ALL LINES
  0   0   0   0   0   0   0   0   0   0   0   1   3
 SOLUTION NUMBER    1   FOR MACHINE NUMBER     30

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  1   1   1   1   0   0   0   2   2   2   2   1   3

INPUT SPECIFICATIONS
   GATE VALUE
     1   =1
     2   =1
     3   =1
     4   =1
OUTPUT SPECIFICATIONS
   GATE VALUE
    13   =1
 SOLUTION NUMBER    2   FOR MACHINE NUMBER     30

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
  2   2   2   2   1   1   1   2   2   2   2   1   3

INPUT SPECIFICATIONS
   GATE VALUE
     1   =0
     2   =0
     3   =0
     4   =0
OUTPUT SPECIFICATIONS
   GATE VALUE
    13   =1
ALL SOLUTIONS FOUND FOR PDCF NUMBER    1


+++ INITIAL CONDITIONS FOR MACHINE +++   31
  NUM. OF GATES IN BLIST =     1 NUM. OF GATES IN FLIST =     1 NUM. OF REMAIN. PDCF =     0
  BLIST
 12
  FLIST
```

-13
INITIAL STATUS OF ALL LINES
0 0 0 C 0 0 0 0 0 0 C 2 4
SOLUTION NUMBER 1 FOR MACHINE NUMBER 31

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
1 2 0 0 2 0 C 1 0 0 0 2 4

INPUT SPECIFICATIONS
GATE VALUE
1 =1
2 =0
3 =X
4 =X

OUTPUT SPECIFICATIONS
GATE VALUE
13 =0
SOLUTION NUMBER 2 FOR MACHINE NUMBER 31

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
0 2 1 C 2 C 0 1 0 0 0 2 4

INPUT SPECIFICATIONS
GATE VALUE
1 =X
2 =0
3 =1
4 =X

OUTPUT SPECIFICATIONS
GATE VALUE
13 =0
SOLUTION NUMBER 3 FOR MACHINE NUMBER 31

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
2 1 1 0 2 2 C C 1 0 0 2 4

INPUT SPECIFICATIONS
GATE VALUE
1 =0
2 =1
3 =1
4 =X

OUTPUT SPECIFICATIONS
GATE VALUE
13 =0
SOLUTION NUMBER 4 FOR MACHINE NUMBER 31

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
2 0 1 C 2 2 0 0 1 0 0 2 4

INPUT SPECIFICATIONS
GATE VALUE
1 =0
2 =X

3 =1
4 =X

OUTPUT SPECIFICATIONS
GATE VALUE
13 =0
SOLUTION NUMBER 5 FOR MACHINE NUMBER 31

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
0 1 0 2 2 2 0 0 0 1 0 2 4

INPUT SPECIFICATIONS
GATE VALUE
1 =X
2 =1
3 =X
4 =0

OUTPUT SPECIFICATIONS
GATE VALUE
13 =0
SOLUTION NUMBER 6 FOR MACHINE NUMBER 31

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
0 0 1 2 2 2 0 0 0 1 0 2 4

INPUT SPECIFICATIONS
GATE VALUE
1 =X
2 =X
3 =1
4 =0

OUTPUT SPECIFICATIONS
GATE VALUE
13 =0
SOLUTION NUMBER 7 FOR MACHINE NUMBER 31

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
0 1 2 0 2 2 2 0 0 0 1 2 4

INPUT SPECIFICATIONS
GATE VALUE
1 =X
2 =1
3 =0
4 =X

OUTPUT SPECIFICATIONS
GATE VALUE
13 =0
SOLUTION NUMBER 8 FOR MACHINE NUMBER 31

--SOLUTION (SEQUENTIAL VALUE OF LEADS)--
0 0 2 1 2 2 2 0 0 0 1 2 4

INPUT SPECIFICATIONS

GATE VALUE
1 =^
2 =X
3 =0
4 =1

OUTPUT SPECIFICATIONS
GATE VALUE
13 =0
ALL SOLUTIONS FOUND FOR POCF NUMBER 1