BOOLEAN APPROACHES IN DIGITAL DIAGNOSIS

THESIS

Reginald Harold Gilyard
Captain, USAF

AFIT/GCS/ENG/89D-4

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/89D-4

BOOLEAN APPROACHES IN DIGITAL DIAGNOSIS

THESIS

Reginald Harold Gilyard
Captain, USAF

AFIT/GCS/ENG/89D-4

Approved for public release; distribution unlimited

AFIT/GCS/ENG/89D-4

BOOLEAN APPROACHES IN DIGITAL DIAGNOSIS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Systems)

Reginald Haroid Gilyard, B.S.

Captain, USAF

December, 1989

Acknowledgments

I would like to thank Doctor Frank Brown for the help that he provided as my advisor. I would also like to thank Lieutenant Colonel Charles Bisbee and Captain Bruce George for reviewing this thesis. Lastly, I would like to thank Captain James Kainec for all of the help that he provided.

Reginald Harold Gilyard

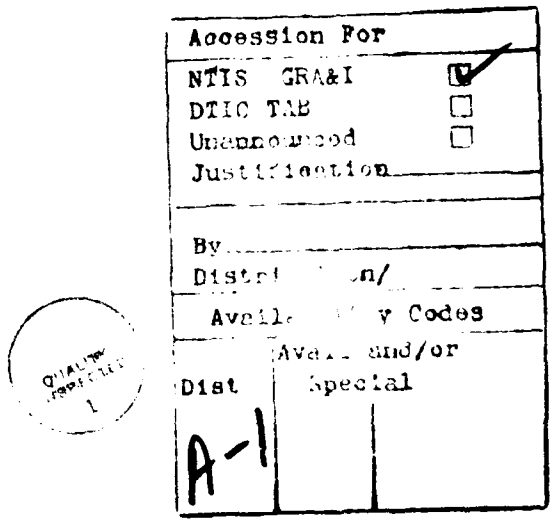


Table of Contents

List of Figures

Figure Page

List of Tables

Abstract

The goal of this thesis is to review and improve two existing methods that use Boolean reasoning as a basis for testing digital circuits. Extensions are made to research done by both Cerny and Kainec in this area.

The method developed by Cerny to generate test vectors capable of detecting single stuck-at, bridge and multiple stuck-at faults is reviewed and then extended in two ways. The first extension incorporates the capability to automatically analyze the results gained from applying a given vector. The second extension allows the diagnosis of sequential circuits. Since Cerny's original method was not automated the entire process is updated to include the extensions and then programmed.

Kainec developed an automated diagnostic system to test for multiple faults in combinational circuits. The original system is restricted to diagnosing faults in circuits with one output. An extension is designed and programmed to incorporate the capability to diagnose multiple output circuits. The extension shows that multiple output circuits offer the added advantage of being able to choose an optimal test vector from a set of genrated vectors, thereby shortening the required testing time for a given circuit.

The software routines are programmed in PC-Scheme (a dialect of LISP) on an IBM microcomputer. Due to a conversion program written by Kainec the software can also be run on a Sun-4 workstation in the T environment. T is derived from Scheme.

BOOLEAN APPROACHES IN DIGITAL DIAGNOSIS

I. Introduction

Background

Digital integrated circuit diagnosis involves three main activities: test vector generation, vector application and interpretation of the results obtained from application. The use of Boolean manipulation of circuit descriptions to accomplish the generation and interpretation steps enables the automation of these activities. The basic Boolean operations and formulas referenced and used throughout this paper can be found in Appendix A (18:187–204).

Test Vector Generation. Test vector generation, from a diagnostic viewpoint, is the process of generating a sequence of vectors capable of detecting a fault or faults in a circuit. Each vector comprises logic values which typically represent voltage values to be applied to the primary inputs of a circuit.

Reghbati decomposes the test vector generation process into three separate activities: selecting a model of the system to be tested, (the system is an integrated circuit in our case), developing a model for the type of fault being diagnosed, and finally generating tests to detect that particular fault (26:9).

In the first activity there are three different levels at which a circuit can be modelled. The first, the functional level, models the circuit as a collection of large functional parts: multiplexers, adders, counters, etc. (26:15). The second level is the logic-gate level, of which the functional-level components are composed. At this level a circuit is modelled as a network of inverters, NAND gates, NOR gates, flip-flops, etc. The last level is the transistor level. Transistors are the basic building blocks of logic gates. At this level a circuit is modelled as a network of switches, interacting to perform the circuit's logical

function. Very Large Scale Integration (VLSI) circuit designers typically lay out circuits at the transistor level to minimize area and maximize speed of operation.

As a test designer descends from the functional level to the transistor level of representation, the number of components rapidly increases. As the number of components increases, so does the complexity of representation and hence the computational complexity of vector generation.

Several authors recognize the need to attempt vector generation at the transistor level because of the widespread use of VLSI technology (27, 2, 10, 15, 9). Boolean approaches use the logic-gate level of representation because it is a form that can be directly manipulated by Boolean operations. In fact, most traditional diagnostic approaches use the gate level and find the transistor level a difficult level at which to work. Several present-day methods that test circuits at the transistor level first seek to convert transistors to gate-level components prior to attempting vector generation (1, 9, 15).

Within each level of representation a circuit can be described as combinational or sequential. Combinational circuit vector generation is commonplace. Sequential circuits include feedback paths (12), and have proven to be a difficult problem for diagnostic system developers from the standpoint of practical circuit representation, fault modelling and final algorithm development.

Reghbati's second test vector generation activity, fault modelling, looks to find a representation for the type of fault that one is seeking to detect. The two generic categories of logic faults are classical and non-classical faults. Classical faults come in two forms: a circuit line stuck-at-1 or a line stuck-at-0. A line stuck-at-1 (0) indicates that the line in question is permanently held at at a logic 1 (0) value, (corresponding to high and low voltages), as opposed to changing with normal circuit operation. Faults in integrated circuits have been traditionally modelled as stuck-at faults. More recently several authors have suggested that stuck-at fault models are inadequate in diagnosing faults in VLSI circuits (2, 9, 8, 15), which tend to be primarily non-classical in nature. Non-classical faults in VLSI circuits include transistor devices stuck-open and stuck-closed (shorted transistors) (2:17).

When modelling faults a test designer must also be concerned with the number of faults that the fault model will represent. Many tests use models that detect single faults; however a circuit may have several failures occur at once. If this is a concern, then the model should also be able to support multiple-fault diagnosis.

Reghbati's third vector generation activity is the actual generation of test vectors using some mathematical process as a basis. In this activity an algorithm is designed and implemented to generate test vectors. In general a given algorithm seeks to combine the circuit description and fault model (Reghbati's first and second activities) to arrive at the primary input values that will enable detection of a fault by observing the circuit's primary outputs.

Generation can be either adaptive or non-adaptive. Adaptive vector generation procedures typically generate a vector and then apply it before generating the next vector. This is done so that the results gained from application of the previous vector can be used to generate the next test vector. Non-adaptive systems generate and store all vectors prior to any test application.

Several known algorithms also allow specification of the particular line suspected to be at fault to support generation of a test designed to diagnose the suspected line (7, 19:28-47).

Interpretation of Results. After the vector is generated, by whatever means, the next step is to apply the vector by manual or automated means. Whether manual or automated, application requires no real algorithmic development and is not the primary focus of this thesis. The *results* of application, however, can be manipulated to yield information about the state of the circuit. As noted previously this information can also be used to generate successive test vectors.

Kainec has developed a means of combining the result of application with the vector applied to achieve a logical association between the two (18:80-82). This association is then used to gain information about internal circuit parameters which represent the checkpoints of the circuit. The original circuit description is used in conjunction with the information about these parameters to identify the presense or absence of a fault.

Problem

This research develops extensions to two existing digital circuit diagnostic approaches which use the Boolean manipulation of circuits described at the logic-gate level. The effort primarily focuses on the use of Boolean reasoning techniques to achieve the enhancements.

Cerny's Vector Generation Procedure. The first approach that is extended was originally developed by Cerny (7) to generate test vectors for digital circuits represented at the gate level. Cerny's algorithm generates vectors capable of detecting single stuck-at faults, multiple stuck-at faults, and bridge faults, (two lines shorted together), for a specified line or lines in the circuit. The procedure was developed for use on combinational circuits, but has been successful in producing vectors capable of diagnosing faults in combinational circuits with feedback loops. Cerny suggests that this result points to the possibility of supporting squential circuit diagnosis (7:26).

It should be noted that this procedure assumes that the fault, or combination of faults, specified is the only possible faulty condition in the circuit at that time. For example, if lines x_1 and x_2 of a circuit are suspected to be bridged, then they are tested under the assumption that all other lines are fault free.

The algorithm is not automated and has no capability to analyze the results of vector application. To make the procedure a true diagnostic system, it is extended in this thesis to incorporate an analysis capability. The second extension develops a system for diagnosing faults in sequential circuits using the product of the first extension as a starting point. Finally, the entire system is automated to enable practical use on circuits.

Kainec's Diagnostic System. One extension is made to the automated diagnostic system developed by Kainec (18) for diagnosing multiple stuck-at faults in combinational circuits. Kainec's system accepts gate-level circuit descriptions and analyzes the circuit following several iterations of vector generation and application. The number of iterations is directly related to the size and complexity of the circuit. The adaptive nature of the program minimizes the number of vectors required for testing. Kainec's test procedure is more comprehensive than Cerny's in that it tests a circuit as a whole with no specific

fault assumption that excludes all other possibilities. However, this makes Kainec's routine more computationally intensive than Cerny's.

Currently Kainec's system is limited to diagnosing circuits with one output. The extension described in this thesis enables the diagnosis of multiple output circuits.

Assumptions

Sequential testing with the methods described in this thesis requires physical access to the outputs of memory elements in the circuit. The capability to access these nodes is assumed. The method also assumes that each circuit under test can be reset at any time during the test process.

Scope

The scope of the extensions in this thesis is limited to diagnosing small and medium scale integrated circuits. In the case of both diagnostic processes the computational complexity increases exponentially with the number of primary input variables, making the diagnosis of large scale integrated circuits impractical. Only classical faults are addressed.

Methodology

The approach taken in this research follows the standard development steps in any software engineering endeavor: design, code and test (23:214).

The design of each extension begins with a review of the Boolean manipulation in the original works. A mathematical procedure for each of the extensions is then developed and incorporated into the original developments by Cerny and Kainec.

The coding process changes the manual boolean manipulation procedures into software. A functional decomposition of the steps in the manipulation process maps these steps into separate software subroutines. Two major software programs are developed, one to test digital circuits using the Cerny approach and one using the Kainec method. Coding is done in the Scheme programming language (a dialect of LISP) to take advantage of its symbol processing capabilities.

The testing process is conducted on several simulated circuits to determine correct operation of the program on good and faulty circuits. Accuracy of diagnosis and time to completion serve as the overall metrics for success.

II. Summary of Current Knowledge

This chapter surveys two areas of research that support accomplishment of the thesis effort introduced in Chapter one. The first section discusses several of the works that pertain to the use of Boolean reasoning in circuit diagnosis. Review of these efforts provides an overall view of the primary research area that clearly must be well understood before attacking the problem of enhancing this area. Secondly, research in the area of sequential circuit diagnosis is reviewed to provide some insight into how this traditionally difficult problem might be approached.

Boolean Reasoning Approaches

Four Boolean-based methods relating to circuit diagnosis are introduced: Boolean difference, literal propositions, Cerny's method and Kainec's method. More detail is presented in this chapter on the first two methods than on the last two. Extensive detail on the Cerny and Kainec methods is provided in Chapters three and four for continuity of discussion when developing the mathematical basis for this thesis.

Boolean Difference Both Lala and Fujiwara discuss the method of calculating a Boolean difference in diagnosing digital circuits (12, 19:28-34). It is only a vector generation method and therefore only forms part of a complete diagnostic system.

In defining the Boolean difference a suspected faulty line is specified. A function is constructed using two versions of the circuit description: the circuit function in its original form, and also in a form that has the variable labeling the suspect line complemented. The term difference comes from the exclusive-or operator used to relate the two functions. The exclusive-or operator returns a zero if the logic values of two operands are the same and a one if they are different. The following equation defines the Boolean difference with respect to a specified suspected faulty line labeled x_i (19:28):

$$\frac{dF(\underline{x})}{dx_i} = F(x_1, ..., x_i, ..., x_n) \oplus F(x_1, ..., x_i', ..., x_n). \tag{2.1}$$

For the diagnosis of internal variables the functions on the right-hand-side of this equation are described in terms of the internal variable. In other words the terms or alterms that feed the internal variable are replaced by the internal variable in the above equation. For example, if y_1 is an internal variable equal to $x_2 + x_3$ then the Boolean difference used to diagnose stuck-at faults on this node is given by

$$\frac{dF(\underline{x})}{dy_1} = F(x_1, ..., y_1, ..., x_n) \oplus F(x_1, ..., y_1', ..., x_n) \tag{2.2}$$

where y_1 replaces the alterm $x_2 + x_3$ in the functions above.

Vectors capable of detecting a stuck-at fault are generated by first setting the calculated Boolean difference equal to one. To complete the input vector the suspected node (or logic supporting it if an internal node) is ANDed with the resultant Boolean difference. If testing for a stuck-at-one condition the complement of the suspected variable is used. If testing the stuck-at-zero case then the uncomplemented version is used.

Figure 2.1 shows an example circuit. To test a stuck-at-zero condition on line a we first calculate the Boolean difference which is

$$\frac{dF}{d(A)} = (AB + C) \oplus (A'B + C). \tag{2.3}$$

Reducing this equation yields

$$\frac{dF}{d(A)} = BC'. \tag{2.4}$$

The vector capable of detecting the stuck-at-zero fault is attained by solving the following equation:

$$ABC' = 1. \tag{2.5}$$

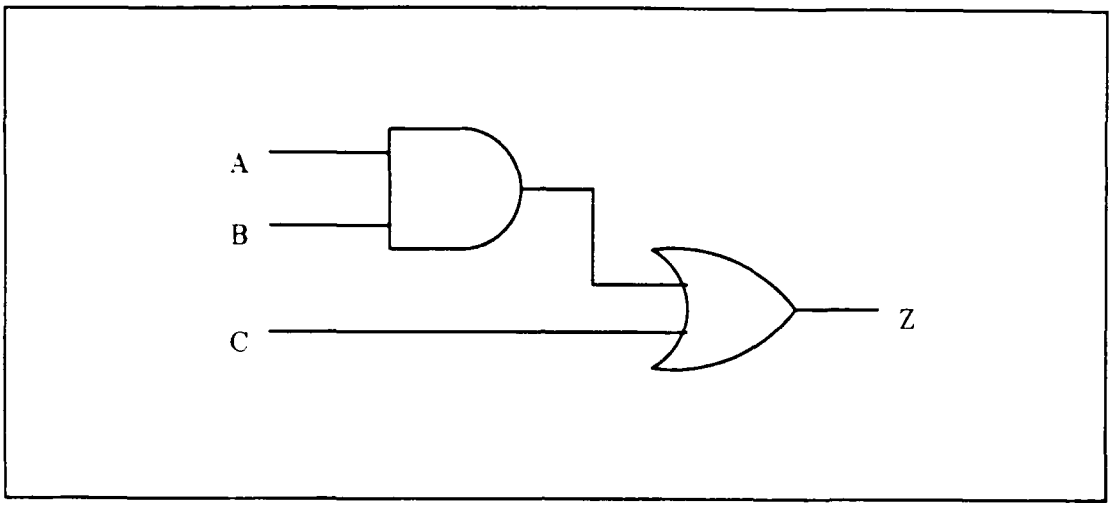


Figure 2.1. Example Circuit

Solution of this equation results in the test vector

$$\begin{aligned} A &= 1 \\ B &= 1 \\ C &= 0. \end{aligned}$$

Literal Propositions Johnson details an approach that generates test vectors by describing a given circuit in terms of propositions (16:489-491). A proposition is a declaration that must be true or false with no other alternatives. Therefore two propositions are generated for each line in the circuit. The propositions for each line are the complemented and uncomplemented forms of the variable that labels the line. The propositions incorporate fault modelling information by including the possible conditions for each line (normal, stuck-at zero or stuck-at one). As an example the following equations describe the two propositions for a line labeled a (16:490).

$$P_a = Aa_n + a_1 \tag{2.6}$$

and

$$P'_a = A'a_n + a_0, \tag{2.7}$$

where P_a is the proposition that a equals one and P'_a is the proposition that a is zero. A is the input to line a, a_n means a is normal, a_1 means a is stuck-at one and a_0 means a is stuck-at zero. Using these conventions equation (2.6) states that a will be equal to one if either A is one and the line is normal, or the line is stuck-at one. Equation (2.7) defines the proposition that a is not equal to one.

Johnson goes on to describe how lines are combined using the gates of a circuit, resulting in a combination of propositions (16:489). The following system of equations describes an AND gate with input lines a and b and output line c.

$$
\begin{aligned}
P_a &= Aa_n + a_1 \\
P'_a &= A'a_n + a_1 \\
P_b &= Bb_n + b_1 \\
P'_b &= B'b_n + b_0 \\
P_c &= P_a P_b c_n + c_1 \\
P'_c &= (P'_a + P'_b)c_n + c_0.
\end{aligned} \tag{2.8}
$$

Continuing along these lines a collection of the gates in a circuit can be combined to form a system of equations that ultimately includes the circuit's output line. This group of propositions, which describes the entire circuit, is then used by Johnson to generate two other equation systems: one for a fault-free circuit, and one for a faulty circuit which is based on a user-specified fault. Figure 2.2 as an example circuit. Below are the systems of equations associated with the circuit, with node a suspected to be stuck-at one. The following are all of the line propositions:

$$
\begin{aligned}
P_a &= Aa_n + a_1 \\
P'_a &= A'a_n + a_1 \\
P_b &= Bb_n + b_1 \\
P'_b &= B'b_n + b_0 \\
P_c &= Cc_n + c_1 \\
P'_c &= C'c_n + c_0 \\
P_d &= P_a P_b d_n + d_1 \\
P'_d &= (P'_a + P'_b)d_n + d_0 \\
P_e &= (P_d + P_c)e_n + e_1 \\
P'_e &= (P'_d P'_c)e_n + e_0.
\end{aligned} \tag{2.9}
$$

The fault-free circuit propositions are found by first setting all variables with subscript n equal to one. Next all variables with subscripts 0 and 1 are set equal to zero. Finally, the equations describing gates are changed by substituting the propositions found one their right-hand-sides with the line propositions that make up the gate. The fault-free

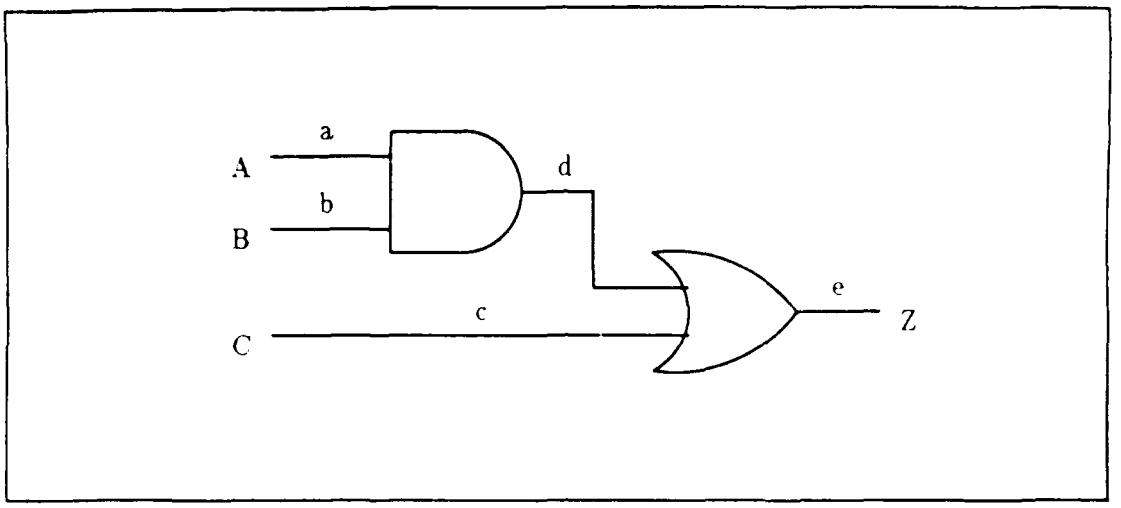


Figure 2.2. Example Circuit

propositions corresponding to the system of equations above are

$$\begin{aligned} P_a &= A \\ P'_a &= A' \\ P_b &= B \\ P'_b &= B' \\ P_c &= C \\ P'_c &= C' \\ P_d &= AB \\ P'_d &= A' + B' \\ P_e &= AB + C \\ P'_e &= (A' + B')C'. \end{aligned} \tag{2.10}$$

The a stuck-at-one propositions are found by setting A equal to one in the above equations resulting in

$$\begin{aligned} P_a &= 1 \\ P'_a &= 0 \\ P_b &= B \\ P'_b &= B' \\ P_c &= C \\ P'_c &= C' \\ P_d &= B \\ P'_d &= B' \\ P_e &= B + C \\ P'_e &= B'C'. \end{aligned} \tag{2.11}$$

Test vectors are generated based on a proposition concerning the output variable. Specifically, the output value in the fault-free circuit is the opposite of the output value in the faulty circuit. Mathematically Johnson states this proposition with the following equation (16:491).

$$P_e(n)P'_e(f) + P'_e(n)P_e(f) = 1. \tag{2.12}$$

where e is the output of the circuit, (n) associates z with a normal (fault-free) circuit and (f) relates to a faulty circuit.

Using the fault-free and faulty propositions for the example circuit of Figure 2.1 equation (2.12) becomes

$$(AB + C)(B'C') + (A' + B')(C')(B + C) = 1. \tag{2.13}$$

After simplification the resulting equation is

$$A'BC' = 1. \tag{2.14}$$

This equation yields the input vector

$$\begin{aligned} A &= 0 \\ B &= 1 \\ C &= 0. \end{aligned} \tag{2.15}$$

Inspection of this vector shows that setting A equal to zero properly excites the suspected line while the other variable settings sensitize a path from the suspected fault to the output.

Cerny's Method. Cerny's method for test vector generation begins with two structural descriptions of a given circuit (7:13). The first is a system of equations that describes the interconnection of gates in the circuit, where each gate has a Boolean equation that relates the inputs and outputs of the gate. The second description is an overall single Boolean equation that relates the primary inputs to the output of the circuit. If the circuit contains several outputs then there is an overall equation for each output.

The first structural description is modified by interjecting information about the suspected location of a fault (or faults) in the circuit (7:14). The user specifies a variable that identifies the suspected faulty line in the circuit. The variable is isolated in the circuit by replacing it (or the logic that feeds it if an internal variable) with a "test" variable.

The modified first structural description is collapsed into one equation and then combined with the second structural description using Boolean reasoning techniques. After this combination of circuit structure and fault model information is manipulated test variable remains. The test variable is then substituted by one or zero to generated a stuck-at one or stuck-at zero vector, respectively.

Kainec's Method. Kainec's method (18) begins with a system of equations that represents the structure of a circuit, much like the literal proposition and Cerny methods. The approach differs from the others in several ways. The method is a complete diagnostic system that uses the results of vector application to detect and isolate stuck-at faults. It is also an adaptive system, using the results of each succesive vector application to generate the next vector. In addition to these differences the Kainec method does not require specification of fault variables. It tests for all possible stuck-at faults in a circuit automatically.

The system of equations that Kainec starts with is modified by inserting stuck-at fault model information. The stuck-at fault model (discussed in Chapter four) specifies normal and faulty conditions for each critical line in the system by substituting a fault model equation for each of these lines. Certain lines, formally known as checkpoints, are critical in the sense that they must be considered for fault modelling to insure maximum fault coverage for the circuit. A checkpoint is defined as a fanout branch of a node that fans out or a primary input node that does not fan out. The fault model also specifies constraints on the checkpoints that are included when altering the structural description.

Once modified, the system of equations is collapsed into one equation that Kainec calls the "characteristic equation" (18:69-73). The characteristic equation includes all structural and fault model information for the circuit. It is this equation that is manipulated to produce a test vector. When all possible information has been gained from

application of test vectors the states of the checkpoints can be deduced to detect and isolate faults that may be present. Kainec's method also uses the information gained from acompleted testing session to derive the actual function of the circuit.

Sequential Circuit Diagnosis

Very few research efforts have been accomplished in the area of sequential circuit diagnosis. Even less has been done using Boolean reasoning to diagnose sequential circuit faults. Two common techniques for general diagnosis exist. The first involves the conversion of a sequential circuit to a combinational one (19:49-50)(12:67-74). The second seeks to verify the state table that describes a given sequential circuit (19:50-66).

Conversion Method. The conversion of a sequential circuit into a combinational one is detailed in Chapter three to maintain continuity in the description of the approach taken in this thesis to diagnose faults in sequential circuits.

In general the process begins by visualizing a sequential circuit as a combination of combinational logic and sequential elements (typically flip-flops) (19:50). The two parts are recognizably distinct and are often referred to as a combinational block and a sequential block. Some or all of the outputs of the combinational block feed the inputs of the sequential block. The outputs of the sequential block then feed back as inputs to the combinational block.

To do the conversion the sequential elements are "flattened". Flattening treats the outputs of the sequential block as primary circuit outputs, and the feedback inputs are replaced on the combinational block by the intial state of the sequential circuit. The elimination of the feedback path creates a iterative combinational circuit. The flattened sequential elements are referred to by Fujiwara as "pseudo" flip-flops (12:70).

Once the conversion is made the typical approach is to use a slightly modified version of the D-algorithm to diagnose faults in the modified circuit. However, it would seem that any algorithm capable of testing a combinational circuit could be used.

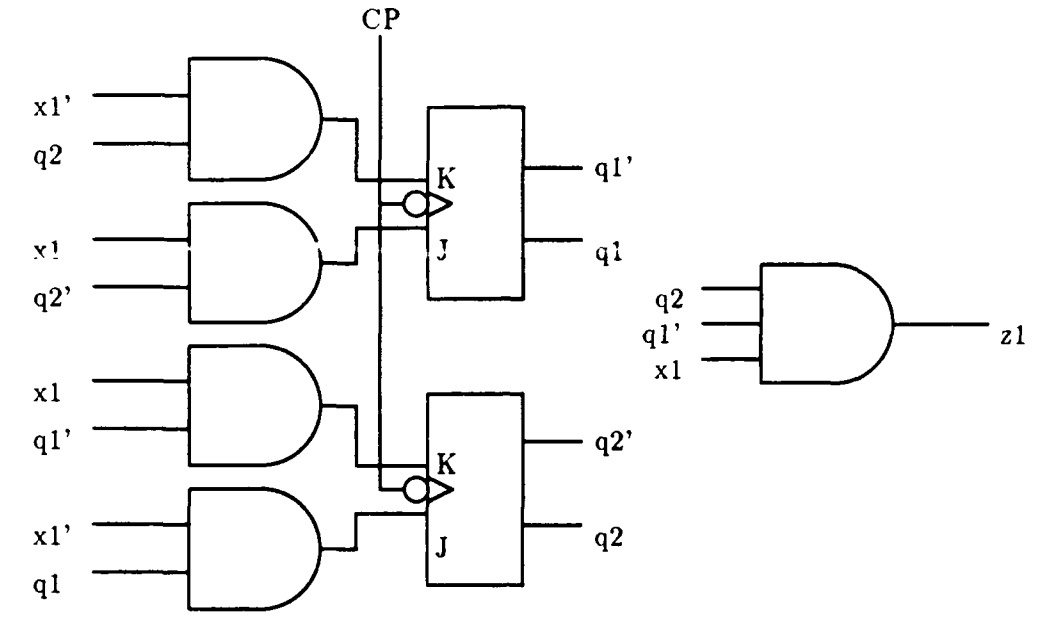
State Table Verification. Lala outlines a method of testing sequential circuits by verifying the state table associated with a given circuit (19:50-66). A state table identifies the possible transitions (and associate outputs) from state to state in the circuit based on the present state and input applied. Figure 2.3 shows an example circuit with its state diagram and state table (21:218-220).

By applying an input and examining the state and outputs of the circuit a comparison can be made with the state table. To completely test the circuit all rows of the state table must be verified, assuming that the machine can be placed in all possible states of the table.

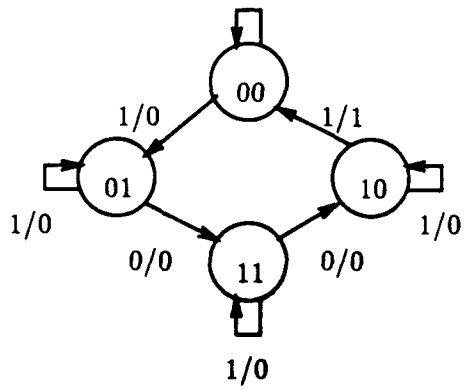
Summary

Boolean reasoning methods for circuit diagnosis offer a logical means of incorporating a representation of a suspected fault, or class of faults, within the structural description of a given circuit. Most methods are designed to process circuits described at the gate level. Typically only stuck-at faults are diagnosed.

Sequential circuit diagnosis has always proven to be a difficult area in fault diagnosis. The most common approaches to solving the problem seek to transform the sequential circuit into a combinational circuit.



Sequential Circuit



STATE DIAGRAM

Present State	Next State/ Output	
	x = 0	x = 1
q1q2	q1q2/z1	q1q2/z1
00	00/0	01/0
01	11/0	01/0
10	10/0	00/1
11	10/0	11/0

STATE TABLE

Figure 2.3. Example Sequential Circuit

III. Mathematical Development of Cerny-based Diagnostic Algorithm

This chapter details the mathematical development that is used to diagnose stuck-at faults in combinational and sequential circuits using the work of E. Cerny as a basis (7). This development is primarily extracted from Cerny's examples (7:13-22) which end with the generation of test vectors capable of detecting single stuck-at faults, bridging faults and multiple stuck-at faults.

Two extensions are made to the original Cerny approach. The first incorporates the capability to analyze the results of an input/output experiment thus creating a true diagnostic system. An input/output experiment involves applying a generated vector and reading the circuit outputs. The second extension enables the diagnosis of sequential circuits.

The derivations are divided into four sections. The first section discusses single stuck-at fault diagnosis by reviewing Cerny's method for vector generation and then developing the extension for analyzing test results. The next two sections discuss Cerny's methods for generating vectors to detect bridge faults and multiple stuck-at faults. The extension for analyzing test results is not repeated in these two sections because it is the same regardless of the type of fault being diagnosed. In the fourth section sequential circuit diagnosis is addressed.

The last section in this chapter describes a another approach to test vector generation that yields the same test vectors as the Cerny method when there is only one circuit output. It is easier to process on paper but appears to require the same number of computational steps required with Cerny's approach. For some multiple output circuits it generates less possibilities for test vectors than the Cerny method.

The procedures described in this chapter (except for the last section) have been automated, and the implementation of the software system is described in Chapter five.

Single Fault Diagnosis

This section discusses the mathematical development of the single-fault diagnostic routine for combinational circuits. A running example will be used based on the circuit

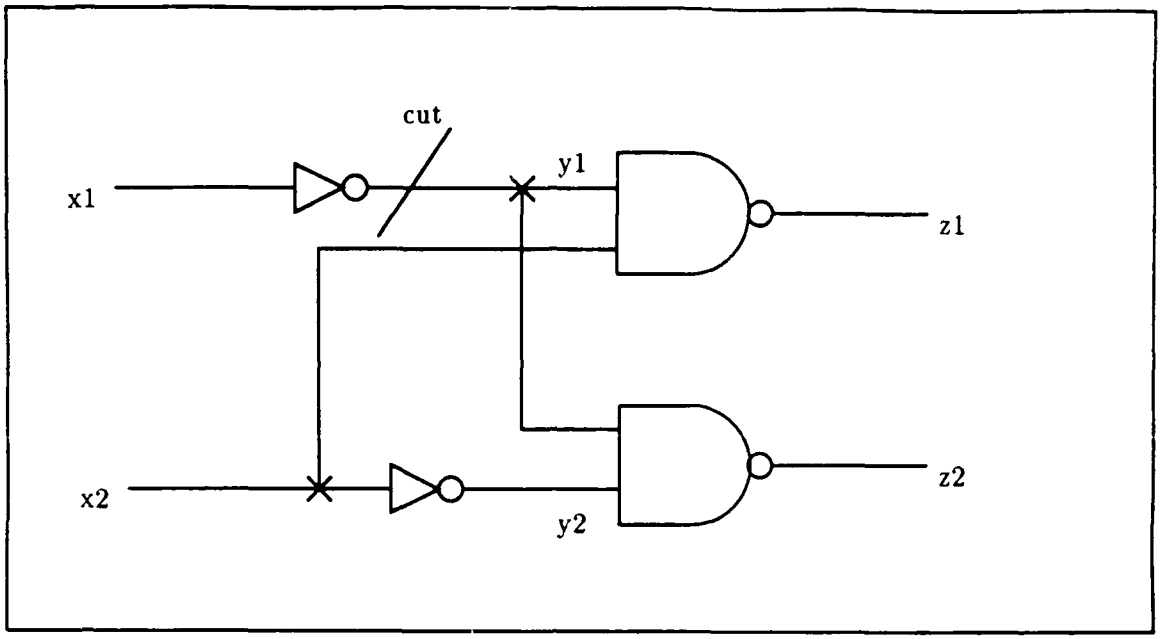


Figure 3.1. Example Circuit for Combinational Single SA Fault Diagnosis

in Figure 3.1 to give a specific look at how the general equations (general in that they describe the procedure for any abitrary combinational circuit) are derived.

Test Vector Generation. The derivation of Cerny's procedure for generating test vectors begins by forming an overall circuit characteristic function (CCF) (7:13-15) from a system of equations describing the circuit. However, what he actually ends up with is an equation that is the CCF set equal to one. We will call this equation an overall circuit characteristic equation (CCE) and talk primarily in terms of equations throughout the discussion. The overall CCE is a structural description of the circuit at the gate level.

The overall CCE is formed by combining the individual CCEs in the circuit. An individual CCE is an equation that identifies the inputs and output of each logic gate in the circuit. Referencing Figure 3.1, the running example for combinational single fault diagnosis, the individual CCEs are

$$\begin{aligned} y_1 &= x_1' \\ y_2 &= x_2' \\ z_1 &= y_1' + x_2' \\ z_2 &= y_1' + y_2'. \end{aligned} \tag{3.1}$$

Node y_1 is the suspected fault node. Cutting line y_1 yields

$$\begin{aligned} y_2 &= x_2' \\ z_1 &= TEST' + x_2' \\ z_2 &= TEST' + y_2'. \end{aligned} \tag{3.2}$$

By equation (A.31) these equations can be represented in the form that sets a function of the inputs and outputs equal to one, using the exclusive-nor operation, producing

$$\begin{aligned} y_2 \odot x_2' &= 1 \\ z_1 \odot (TEST' + x_2') &= 1 \\ z_2 \odot (TEST' + y_2') &= 1. \end{aligned} \tag{3.3}$$

These equations are combined using equations (A.52) and (A.53) to form the single equation

$$[y_2 \odot x_2'] \cdot [z_1 \odot (TEST' + x_2')] \cdot [z_2 \odot (TEST' + y_2')] = 1. \tag{3.4}$$

Simplifying this equation results in

$$x_2 y_2' z_1' z_2 TEST + x_2 y_2' z_1 z_2 TEST' + x_2' y_2 z_1 z_2 TEST' + x_2' y_2 z_1 z_2' TEST = 1. \tag{3.5}$$

The result is an equation that includes the primary inputs, internal variables, test variable and outputs of the circuit. To get the overall CCE we need to determine a relationship between the inputs, test variable and output. Therefore the internal variables ($\underline{y}$) can be eliminated, using the disjunctive eliminant defined in equation (A.60), yielding the following overall CCE for the example circuit:

$$x_2 z_1' z_2 TEST + x_2 z_1 z_2 TEST' + x_2' z_1 z_2 TEST' + x_2' z_1 z_2' TEST = 1. \tag{3.6}$$

In general the overall CCE is

$$\phi_{ccf}(\underline{x}, TEST, \underline{z}) = 1. \tag{3.7}$$

where $\underline{x}$ is the vector of primary inputs (minus those that are deleted when the fault node is cut), $TEST$ is the test variable and $\underline{z}$ are the outputs.

Having developed the overall CCE for a given circuit, the next step is to form a description of the circuit that relates the inputs to the outputs with no manipulation to account for a suspected faulty node (as opposed to the overall CCE which is altered to isolate the node). This description is called the fault-free circuit equation (FCE). The FCE is formed by combining the individual output equations. The individual output equations are the equations that individually relate each output to the primary inputs that feed it.

Using the present example the individual output equations are as follows:

$$z_1 = x_1 + x_2', \tag{3.8}$$

$$z_2 = x_1 + x_2. \tag{3.9}$$

Using equations (A.31), (A.52) and (A.53) equations (3.8) and (3.9 can be combined to get

$$[z_1 \odot x_1 + x_2'] \cdot [z_2 \odot x_1 + x_2] = 1. \tag{3.10}$$

Simplification yields the following FCE for the example circuit:

$$x_1' x_2 z_1' z_2 + x_1' x_2' z_1 z_2' + x_1 z_1 z_2 = 1. \tag{3.11}$$

The general form of the FCE is

$$\psi_{fcf}(\underline{x}, \underline{z}) = 1, \tag{3.12}$$

where ψ_{fcf} is the fault-free circuit function. The fault-free circuit function is a function that relates the primary inputs of the circuit to the circuit's outputs.

Cerny's next step in the process of generating test vectors is to create an output characteristic equation (OCE) (7:15). The OCE is an equation that describes the relationship between the test variable and the primary inputs of a normally operating ciruit. It is developed by combining the overall CCE ($\phi_{ccf}(\underline{x}, TEST, \underline{z}) = 1$) and the FCE ($\psi_{fcf}(\underline{x}, \underline{z}) = 1$), and then disjunctively eliminating the output variables.

Using the example the result of combining CCE and FCE is

$$\begin{aligned}[x_2 z_1' z_2 TEST + x_2 z_1 z_2 TEST' + x_2' z_1 z_2 TEST' + x_2' z_1 z_2' TEST] \cdot \\ [x_1' x_2 z_1' z_2 + x_1' x_2' z_1 z_2' + x_1 z_1 z_2] = 1.\end{aligned} \tag{3.13}$$

Simplifying and disjunctively eliminating the output variables yields the following OCE for the example circuit:

$$x_1' TEST + x_1 TEST' = 1. \tag{3.14}$$

The general form of the OCE is

$$\Theta_{ocf}(\underline{x}, TEST) = 1. \tag{3.15}$$

As previously noted the OCE describes the relationship between the test variable (and hence the fault node) and the primary inputs under normal circuit operation. Therefore

Cerny's next step is to derive a realtionship that is contrary to normal circuit operation. By complementing the OCF and setting it equal to one a relationship contrary to normal circuit operation results. Using the example the resulting equation is

$$x_1' TEST' + x_1 TEST = 1. \tag{3.16}$$

The general form of this result is

$$\Theta'_{ocf}(\underline{x}, TEST) = 1. \tag{3.17}$$

The process of vector generation continues by substituting the test variable with the appropriate value (zero for a stuck-at-zero test, one for stuck-at-one) in equation (3.17), and solving the resulting equation for $\underline{x}$ (7:16–17). Continuing the example the following equations yield test vectors capable of detecting stuck-at zero and stuck-at one faults, respectively:

$$x_1' = 1, \ (stuck - at - zero \ \ test) \tag{3.18}$$

$$x_1 = 1, \ (stuck - at - one \ \ test). \tag{3.19}$$

The general equations for generating the test vectors are

$$\Theta'_{ocf}(\underline{x}, 0) = 1, \ (stuck - at - zero \ \ test) \tag{3.20}$$

$$\Theta'_{ocf}(\underline{x}, 1) = 1, \ (stuck - at - one \ \ test). \tag{3.21}$$

Typically, several solutions exist for a given single stuck-at fault test. As an example, given a stuck-at-zero test, each solution corresponds to a particular minterm of the function $\Theta'_{ocf}(\underline{x}, 0)$ (see Appendix A, Solutions of Boolean Equations). Any of these solutions will yield a test vector capable of detecting a stuck-at-zero fault on the fault node, so one is arbitrarily chosen, set equal to one, and solved for $\underline{x}$ to provide one of the solutions to (3.20). As is the case with the example, to get all possible solutions the function $\Theta'_{ocf}(\underline{x})$ must be expanded with respect to the variables not explicity present in each term.

Given the example circuit, expanding the left-hand-side of equation (3.18) identifies the minterms of the function on the left-hand-side. The following equations, which set each of the minterms equal to one, will yield the two solutions available for testing a stuck-at-zero fault on node y_1:

$$x_1' x_2' = 1, \tag{3.22}$$

$$x_1' x_2 = 1. \tag{3.23}$$

As an aside it should be noted here that the example chosen is relatively simple to promote clarity of the explanation. It may be misleading because the derivation results in very trivial test vector equations (3.18, 3.19), and hence only two minterms for the chosen stuck-at fault. A more complex test vector equation might be

$$x_1 + x_2' = 1, \tag{3.24}$$

which when expanded leads to

$$x_1 x_2' + x_1 x_2 + x_1' x_2' = 1, \tag{3.25}$$

resulting in three minterms, and therefore three possible vectors.

To continue the example later will arbitrarily choose the solution obtained from equation (3.23) as the test vector to apply.

In the general case the following equation represents a particular choice of several minterms that could be used to yield vectors:

$$m_i(\underline{x}) = 1, \tag{3.26}$$

where i is a number from $0..2^m - 1$. The variable m is the number of primary input variables.

Continuing the example, the vector $[x_1 = 0, x_2 = 1]$ is the solution of equation (3.23) for the primary input variables $\underline{x}$.

Running an Input/Output Experiment. Having generated a test vector the process continues with the application of that vector to the circuit's primary inputs. Assuming a digital circuit conforming to a two-valued Boolean algebra, each of the elements of $\underline{z}$ will take on one of two logic values, $r_{ij} \in \{0, 1\}$, following vector application. Subscript i denotes the minterm $m_i(\underline{x})$ used to determine the applied test vector, and j specifies the output z_j in the circuit that is read by the user to get the result. Mathematically this point is represented as follows (18:80):

$$m_i(\underline{x}) = 1 \quad \Rightarrow \quad z_j = r_{ij}, \qquad r_{ij} \in \{0, 1\}. \tag{3.27}$$

Using this notation to describe an experiment with the example circuit yields the following two equations (one for each output):

$$m_1(x_1, x_2) = 1 \quad \Rightarrow \quad z_1 = r_{11}, \qquad r_{11} \in \{0, 1\} \tag{3.28}$$

and

$$m_1(x_1, x_2) = 1 \quad \Rightarrow \quad z_2 = r_{12}, \qquad r_{12} \in \{0, 1\}. \tag{3.29}$$

Assume that in the ongoing example the application of the vector [0,1] resulted in a one on both outputs z_1 and z_2 (for the sake of an example we could pick any of the four possible combinations for the outputs). Therefore

$$x_1' x_2 = 1 \quad \Rightarrow \quad z_1 = 1, \tag{3.30}$$

and

$$x_1' x_2 = 1 \quad \Rightarrow \quad z_2 = 1. \tag{3.31}$$

Analysis of Input/Output Experiment Results. This discussion shows how the result of a test is analyzed to determine if a fault has been detected. Analysis is done in two steps. In the first step a term is constructed for each output of the circuit that is accessible by the suspected faulty line. Each term is constructed using two items: the particular minterm that supplied the vector used in the experiment, and the result of the experiment taken from the output being used to construct the term (ie., minterm $m_1(\underline{x})$ combined with result r_{12} which has been read from output z_2). The second analysis step

compares each term to the original circuit description to determine if a fault has occurred. Comparisons are only made with respect to the same output that a term is constructed with. For example, a term constructed with an output z_i will only be compared with the equation from the circuit description that relates the primary inputs to output z_2.

In step one a method developed by Kainec is used to combine the minterm used to get a test vector with the result of applying the vector (18:80-81). A particular vector results from the solution of equation (3.26). Application of a vector derived from this equation implies a result of the form in equation (3.27) (18:80). This implication is shown in equation (3.27).

The combination of the minterm and the result of applicationy begins by complementing the equation that sets the arbitrarily chosen minterm equal to one. Using our chosen example minterm, the complement of the equation containing this minterm is

$$m_1'(x_1, x_2) = 0. \tag{3.32}$$

In the general situation the complement is

$$m_i'(\underline{x}) = 0. \tag{3.33}$$

By equation (A.30) the right side of the implications in equations (3.28) and (3.29) are changed to:

$$z_1 \oplus r_{11} = 0 \tag{3.34}$$

and

$$z_2 \oplus r_{12} = 0. \tag{3.35}$$

To get the general equation we make the change to the right side of the implication in equation (3.27) to get

$$z_i \oplus r_{ij} = 0. \tag{3.36}$$

Rewriting the implications in (3.28) and (3.29) with substitutions (3.32), (3.34) and (3.35)

$$m_1'(x_1, x_2) = 0 \quad \Rightarrow \quad z_1 \oplus r_{11} = 0, \tag{3.37}$$

$$m_1'(x_1, x_2) = 0 \quad \Rightarrow \quad z_2 \oplus r_{12} = 0. \tag{3.38}$$

In general

$$m_i'(\underline{x}) = 0 \quad \Rightarrow \quad z_j \oplus r_{ij} = 0. \tag{3.39}$$

By the Extended Verification Theorem (Appendix A) (3.37) and (3.38) become

$$z_1 \oplus r_{11} \leq m_1'(x_1, x_2) \tag{3.40}$$

$$z_2 \oplus r_{12} \leq m_1'(x_1, x_2). \tag{3.41}$$

The general relation is

$$z_j \oplus r_{ij} \leq m_i'(\underline{x}). \tag{3.42}$$

From equation (A.10) equations (3.40) and (3.41) are equivalent to (18:81)

$$m_1(x_1, x_2) \cdot (z_1 \oplus r_{11}) = 0 \tag{3.43}$$

$$m_1(x_1, x_2) \cdot (z_2 \oplus r_{12}) = 0. \tag{3.44}$$

Generally

$$m_i(\underline{x}) \cdot (z_j \oplus r_{ij}) = 0. \tag{3.45}$$

Therefore when provided with a specific value for a given output resulting from application of a test vector we form one of the following two equations for each output:

$$m_i(\underline{x}) \cdot z_j = 0 \quad (r_{ij} = 0), \tag{3.46}$$

$$m_i(\underline{x}) \cdot z_j' = 0 \quad (r_{ij} = 1). \tag{3.47}$$

We assumed earlier that following application of the test vector $[x_1 = 0, x_2 = 1]$ (obtained using minterm $m_1(x_1, x_2)$) both of the outputs were equal to one. Therefore for the example equation (3.47) yields

$$x_1' x_2 \cdot z_1' = 0, \tag{3.48}$$

$$x_1' x_2 \cdot z_2' = 0. \tag{3.49}$$

Step two of the analysis process shows how these terms are compared to the equations that make up the original circuit description (the output equations) to determine whether or not a fault is present. For example equation (3.48) is compared to the output equation that relates z_1 to the primary inputs. Equation (3.49) is compared to the z_2 output equation.

Each output equation is represented as a function of the output and the primary inputs set equal to one. The output equations for the example are

$$z_1 \odot (x_1 + x_2') = 1, \tag{3.50}$$

$$z_2 \odot (x_1 + x_2) = 1. \tag{3.51}$$

Simplifying these equations gives

$$x_1 z_1 + x_2' z_1 + x_1' x_2 z_1' = 1, \tag{3.52}$$

$$x_1 z_2 + x_2 z_2 + x_1' x_2' z_2' = 1. \tag{3.53}$$

In general the output equation for a particular output is

$$\psi(\underline{x}, z_j) = 1. \tag{3.54}$$

To match the form of the term(s) formed in the first analysis step both sides of this equation are complemented.

The general equation is

$$\psi'(\underline{x}, z_j) = 0. \tag{3.55}$$

The application of a given vector results in one of two conditions. Either a fault is detected or not. The case in which a test results in no fault being detected yields the following implication:

$$\psi'(\underline{x}, z_j) = 0 \quad \Rightarrow \quad m_i(\underline{x}) \cdot (z_j \oplus r_{ij}) = 0. \tag{3.56}$$

This implication states that the original fault-free circuit representation implies a particular fault-free result. By extended verification (see Appendix A) this is equivalent to stating that the term formed in step 1 is logically included in the output function (the left-hand-side of an output equation) associated with that term. This output function is a fault-free circuit description. The general form of this logical inclusion is

$$m_i(\underline{x}) \cdot (z_j \oplus r_{ij}) \leq \psi'(\underline{x}, z_j). \tag{3.57}$$

Using equation (A.10)

$$\psi(\underline{x}, z_j) \cdot [m_i(\underline{x}) \cdot (z_j \oplus r_{ij})] = 0. \tag{3.58}$$

If in fact the vector in question results in the correct output for the specified function then this equality will be satisfied. If evaluation of the left-hand-side is not identically equal to zero then a fault has been detected.

Returning to the example, equation (3.58) is used twice to compare the terms from the left-hand-side of (3.48) and (3.49) with the left-hand-sides of (3.52) and (3.53) to get

$$(x_1 z_1 + x_2' z_1 + x_1' x_2 z_1') \cdot (x_1' x_2 \cdot z_1') = 0, \tag{3.59}$$

$$(x_1 z_2 + x_2 z_2 + x_1' x_2' z_2') \cdot (x_1' x_2 \cdot z_2') = 0. \tag{3.60}$$

Simplification shows that the left-hand-sides of both equations evaluate to zero. If one or both had failed to do so then y_1 would be stuck-at-zero. An interesting result is that when a failure does occur the left-hand-side of the equation that showed the failure evaluates to the term used in the comparison (developed in step one of this section).

Bridge Faults

The process for generating vectors to diagnose bridge faults is very similar to that of single stuck-at fault diagnosis. An overall CCE is derived and combined with an FCE describing the circuit in question. The resulting equation is then manipulated to get an OCE. Variations that exist in this process when diagnosing bridge faults will be made apparent in the following subsection. An example that is based on the circuit in Figure 3.2 will be used to derive the general forms of the CCE and OCE (the FCE does not change).

Note that the section on single fault diagnosis discussed the following three topics: vector generation, running an input/output experiment, and analyzing the results of that experiment. This section only addresses test vector generation because there are no variations in the way the last two activities are conducted.

Test Vector Generation. When considering bridge fault diagnosis the CCE from equation (3.7) in the single fault diagnosis section is similarly derived, only this time with respect to the two lines that are suspected to be bridged. As before we begin by listing the individual CCEs of the circuit. Using the example circuit in Figure 3.2 the individual CCEs are

$$\begin{aligned} y_1 &= x_1' \\ z_1 &= y_1' + x_2' \\ z_2 &= x_2'. \end{aligned} \tag{3.61}$$

For the example circuit the suspected bridged lines are x_1 and x_2. Both lines are "cut" as before and substituted by test variables resulting in the following system of equations:

$$\begin{aligned} y_1 &= TEST_1' \\ z_1 &= y_1' + TEST_2' \\ z_2 &= TEST_2'. \end{aligned} \tag{3.62}$$

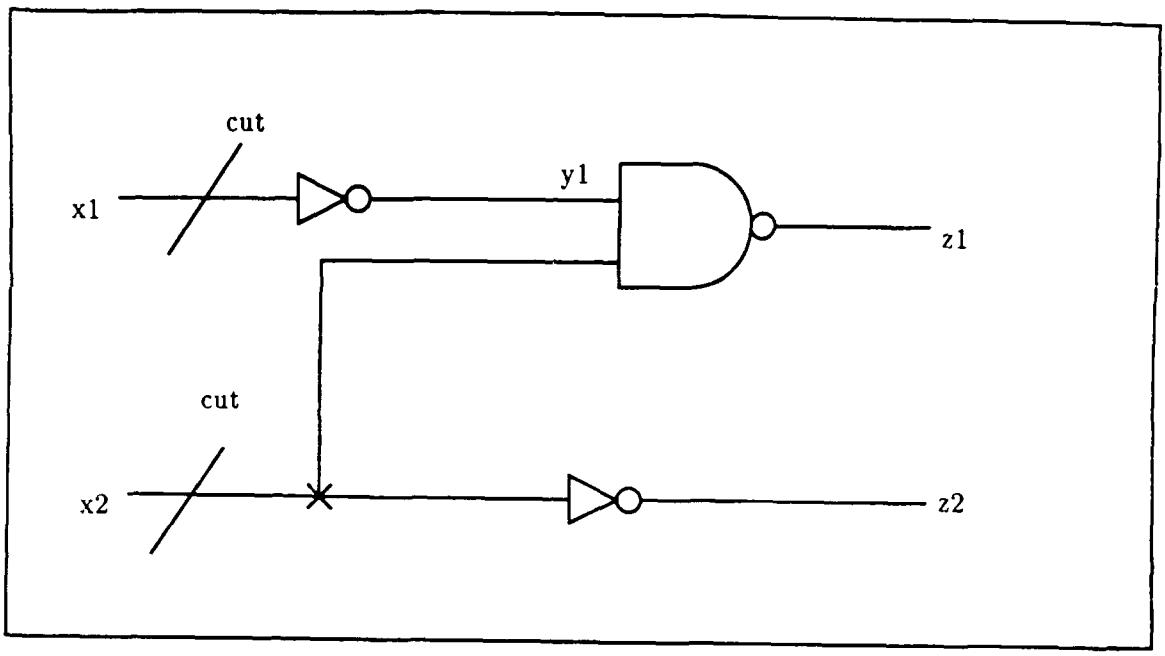


Figure 3.2. Example Circuit for Combinational Bridge Fault Diagnosis

Using equations (A.31), (A.52) and (A.53) this system of equations is equivalent to

$$[y_1 \odot TEST_1'] \cdot [z_1 \odot (y_1' + TEST_2')] \cdot [z_2 \odot TEST_2'] = 1. \tag{3.63}$$

Simplification of this equation yields

$$\begin{aligned} &y_1' z_1 z_2' TEST_1 TEST_2 + y_1' z_1 z_2 TEST_1 TEST_2' + \\ &y_1 z_1' z_2' TEST_1' TEST_2 + y_1 z_1 z_2 TEST_1' TEST_2' = 1. \end{aligned} \tag{3.64}$$

As before the disjunctive eliminant is used to eliminate all internal variables ($\underline{y}$) resulting in the following overall CCE:

$$\begin{aligned} &z_1 z_2' TEST_1 TEST_2 + z_1 z_2 TEST_1 TEST_2' + \\ &z_1' z_2' TEST_1' TEST_2 + z_1 z_2 TEST_1' TEST_2' = 1. \end{aligned} \tag{3.65}$$

In general the overall CCE for bridge fault diagnosis is

$$\phi_{ccf}(\underline{x}, TEST_1, TEST_2, \underline{z}) = 1. \tag{3.66}$$

The FCE is developed by combining the individual output equations as before. The individual output equations for this example are

$$z_1 = x_1 + x_2', \tag{3.67}$$

$$z_2 = x_2'. \tag{3.68}$$

These equations are combined to get

$$[z_1 \odot (x_1 + x_2')] \cdot [z_2 \odot x_2'] = 1. \tag{3.69}$$

After simplification of this equation the FCE for the example circuit is

$$x_1' x_2 z_1' z_2' + x_1 x_2 z_1 z_2' + x_2' z_1 z_2 = 1. \tag{3.70}$$

The OCE is the result of combining the CCE and FCE and then eliminating the output variables from the resulting equation. Using the example the combined CCE and FCE equation is

$$x_1' x_2 z_1' z_2' TEST_1' TEST_2 + x_1 x_2 z_1 z_2' TEST_1 TEST_2 + x_2' z_1 z_2 TEST_2' = 1. \tag{3.71}$$

Eliminating the output variables yields

$$x_1' x_2 TEST_1' TEST_2 + x_1 x_2 TEST_1 TEST_2 + x_2' TEST_2' = 1. \tag{3.72}$$

The general form of the OCE for bridge fault diagnosis is

$$\Theta_{ocf}(\underline{x}, TEST_1, TEST_2) = 1. \tag{3.73}$$

As in the single stuck-at fault vector generation process the complement of the left-hand-side is set equal to one. For the ongoing example

$$x_1 TEST_1' TEST_2 + x_2 + TEST_2' + x_1' TEST_1 TEST_2 + x_2' TEST_2 = 1. \tag{3.74}$$

In general

$$\Theta'_{ocf}(\underline{x}, TEST_1, TEST_2) = 1. \tag{3.75}$$

In the case of bridge faults:

$$TEST_1 = TEST_2. \tag{3.76}$$

Therefore when one variable is zero (one) then the other will be zero (one). Inserting this information into equation (3.74) (7:18)

$$x_2 = 1 \tag{3.77}$$

when both test variables are zero and

$$x_1' + x_2' = 1 \tag{3.78}$$

when both test variables equal one.

These equations are combined to get

$$x_1 x_2 = 1. \tag{3.79}$$

In general the equation used to yield test vectors is

$$\Phi'_{ocf}(\underline{x}, 0, 0) \cdot \Phi'_{ocf}(\underline{x}, 1, 1) = 1. \tag{3.80}$$

Any minterm of the left-hand-side of this equation, when set equal to one and solved, will yield a test vector that can be applied and analyzed as before to detect and isolate a

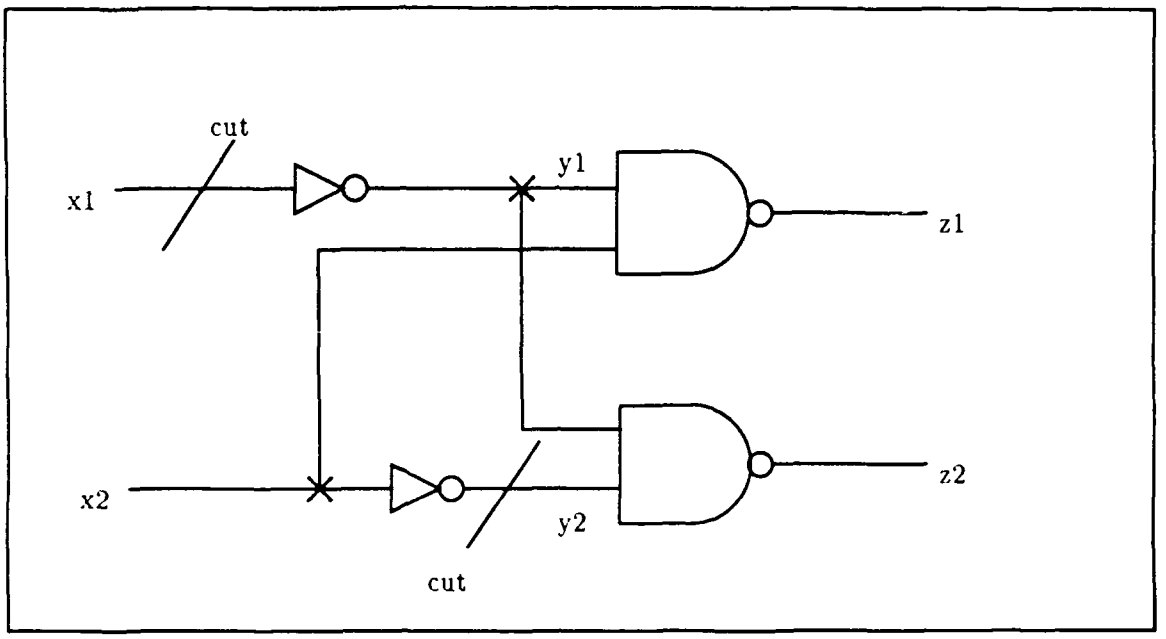


Figure 3.3. Example Circuit for Combinational Multiple SA Fault Diagnosis

bridge fault for the suspected lines. In the example only one minterm remains and therefore the only applicable test vector is $[x_1 = 0, x_2 = 1]$.

Multiple Stuck-at Faults

Multiple stuck-at fault diagnosis is an extension of the single fault case. The variations that exist for test vector generation are illustrated by example (using Figure 3.3) in the following subsection.

Test Vector Generation. The overall CCE is derived with respect to the lines being analyzed. From the example circuit the individual CCEs are

$$\begin{aligned} y_1 &= x_1' \\ y_2 &= x_2' \\ z_1 &= y_1' + x_2' \\ z_2 &= y_1' + y_2'. \end{aligned} \tag{3.81}$$

The suspected faulty lines are x_1 and y_2. Cutting these lines yields

$$\begin{aligned} y_1 &= TEST_1' \\ z_1 &= y_1' + x_2' \\ z_2 &= y_1' + TEST_2'. \end{aligned} \tag{3.82}$$

Combining this system of equations we get

$$[y_1 \odot TEST_1'] \cdot [z_1 \odot (y_1' + x_2')] \cdot [z_2 \odot (y_1' + TEST_2')] = 1. \tag{3.83}$$

Simplifying this equation yields

$$\begin{aligned} y_1' z_1 z_2 TEST_1 + x_2 y_1 z_1' z_2' TEST_1' TEST_2 + x_2 y_1 z_1' z_2 TEST_1' TEST_2' + \\ x_2' y_1 z_1 z_2' TEST_1' TEST_2 + x_2' y_1 z_1 z_2 TEST_1' TEST_2' = 1. \end{aligned} \tag{3.84}$$

As was done before the internal variables ($\underline{y}$) are disjunctively eliminated to get the overall CCE

$$\begin{aligned} z_1 z_2 TEST_1 + x_2 z_1' z_2' TEST_1' TEST_2 + x_2 z_1' z_2 TEST_1' TEST_2' + \\ x_2' z_1 z_2' TEST_1' TEST_2 + x_2' z_1 z_2 TEST_1' TEST_2' = 1. \end{aligned} \tag{3.85}$$

In general the CCE for multiple stuck-at fault diagnosis is

$$\phi_{ccf}(\underline{x}, TEST_1, TEST_2, \ldots, TEST_n, \underline{z}) = 1 \tag{3.86}$$

where n is the number of variables being tested.

Combination of the individual output equations provides the circuit FCE. The output equations for the example are

$$z_1 = x_1 + x_2', \tag{3.87}$$

$$z_2 = x_1 + x_2. \tag{3.88}$$

They are combined to get

$$[z_1 \odot x_1 + x_2'] \cdot [z_2 \odot x_1 + x_2] = 1. \tag{3.89}$$

Simplification yields the following FCE for the example circuit:

$$x_1' x_2 z_1' z_2 + x_1' x_2' z_1 z_2' + x_1 z_1 z_2 = 1. \tag{3.90}$$

The overall CCE is combined with the FCE. For the example this combination results in the following equation:

$$x_1 z_1 z_2 TEST_1 + x_1' x_2 z_1' z_2 TEST_1' TEST_2' + x_1' x_2' z_1 z_2' TEST_1' TEST_2 + \tag{3.91}$$
$$x_1 x_2' z_1 z_2 TEST_2' = 1.$$

The output variables are eliminated as before resulting in the following OCE:

$$x_1 TEST_1 + x_1' x_2 TEST_1' TEST_2' + x_1' x_2' TEST_1' TEST_2 + x_1 x_2' TEST_2' = 1. \tag{3.92}$$

For multiple stuck-at fault diagnosis the general OCE is

$$\Theta_{ocf}(\underline{x}, TEST_1, TEST_2, \ldots, TEST_n) = 1. \tag{3.93}$$

As in the single stuck-at and bridge fault routines the left-hand-side is complemented. For the example circuit the result is

$$x_2 TEST_1' TEST_2 + x_1 TEST_1' TEST_2 + x_1 x_2 TEST_1' + x_1' TEST_1 + x_1' x_2' TEST_2' = 1. \tag{3.94}$$

In general

$$\Theta'_{ocf}(\underline{x}, TEST_1, TEST_2, \ldots, TEST_n) = 1. \tag{3.95}$$

The suspected stuck-at values are substituted for the appropriate test variables in this equation to yield test vectors capable of detecting the particular fault condition specified. For the example we suspect x_1 and y_2 ($TEST_1$ and $TEST_2$) of being stuck-at zero and one, respectively. Substituting these values in equation (3.94) yields

$$x_1 + x_2 = 1. \tag{3.96}$$

Minterms of the left-hand-side of this equation will lead to test vectors that are capable of detecting the specified muliple stuck-at fault condition. The minterms are generated, as before, by expanding with respect to the input variables to get

$$x_1x_2 + x_1x_2' + x_1'x_2 = 1. \tag{3.97}$$

Any of the minterms lead to test vectors. One is arbitrarily chosen and application and analysis proceed as before.

Sequential Circuit Diagnosis

This section addresses the mathematical development of the diagnostic routine for sequential circuits. A given sequential circuit is first modelled as a purely combinational circuit. The process then uses the methods discussed in previous sections to generate test vectors based on the type of fault suspected. Also considered in the generation of test vectors is the current state of the circuit. The state of the circuit is determined by reading the values of the memory elements in the circuit. The circuit in Figure 3.7 will serve as an example for deriving the equations used in the process.

Modelling Sequential Circuits as Combinational Circuits. Figure 3.4 shows a generic model of a synchronous sequential circuit (19:50). The combinational block,

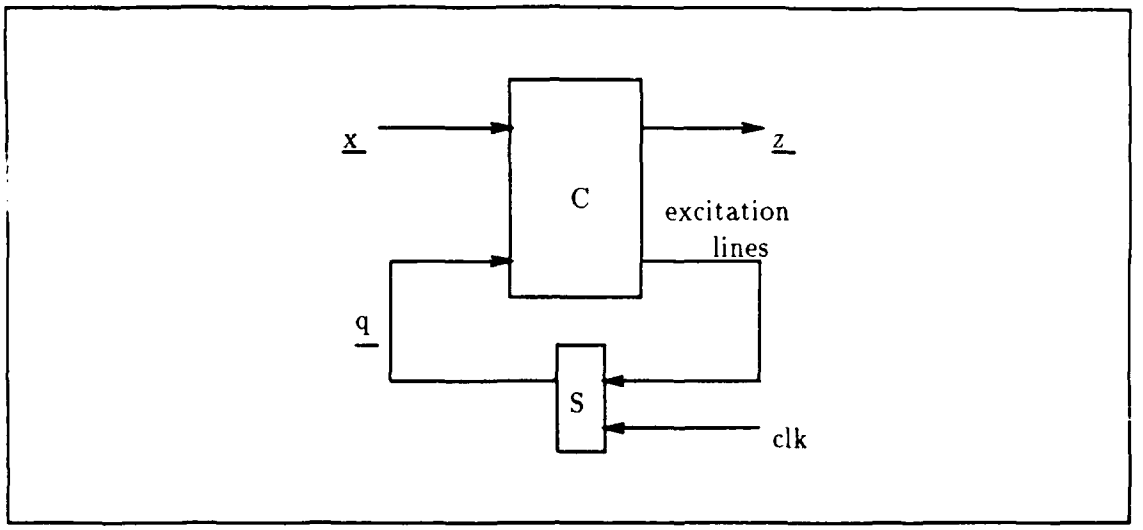


Figure 3.4. Model of a Sequential Circuit

C, represents an arbitrarily complex combinational circuit while the sequential block, S, includes any number of clocked sequential elements, typically flip-flops.

The sequential circuit model is converted into a totally combinational circuit by "flattening" the sequential circuit. The result of flattening the circuit in Figure 3.4 is shown in Figure 3.5 (19:50). This figure illustrates several stages of the circuit, each representing discrete moments in time.

The key to flattening, as illustrated in each stage of Figure 3.5, is to treat the present state of all memory elements as primary inputs to the combinational network. The next state of each element is treated as a primary output. To do this each memory element must be described by its characteristic equation which can be viewed as a combinational circuit description. Fujiwara describes the use of the characteristic equation in this way as a pseudo flip-flop (12:72). An example pseudo JK flip-flop is shown in Figure 3.6.

Vector Generation. Figure 3.7 will serve as an example for development of the procedure for generating test vectors. It contains combinational circuitry and two memory elements (JK flip-flops). The flattened circuit is shown in Figure 3.8.

The generation of test vectors for the flattened circuit begins with Cerny's approach

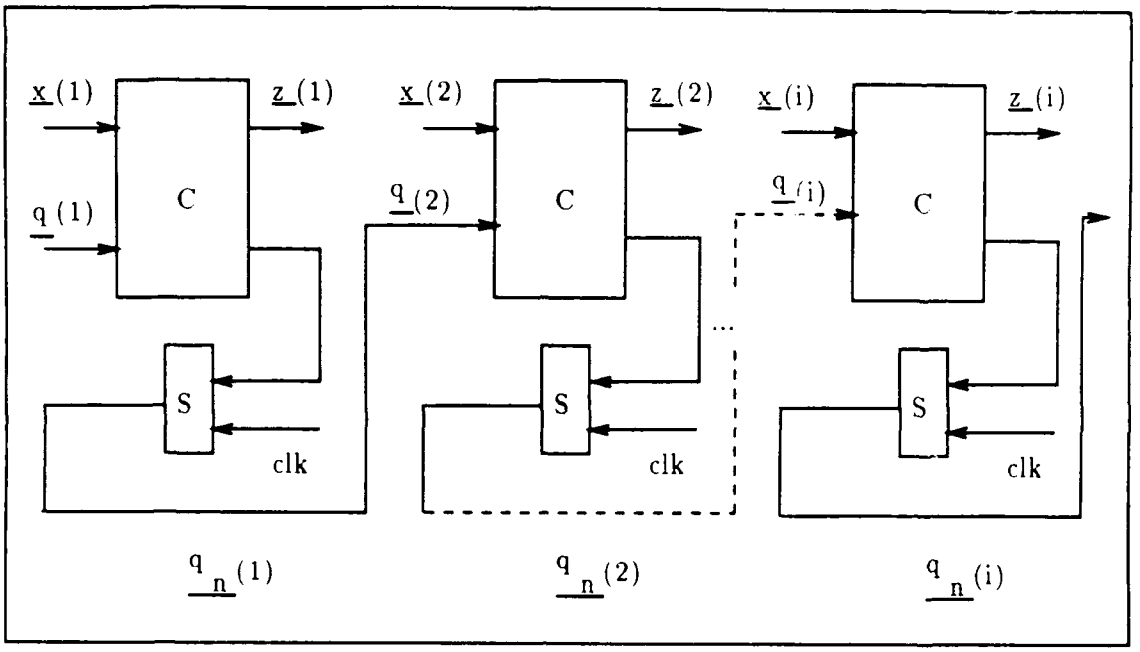


Figure 3.5. Flattened Sequential Circuit Model

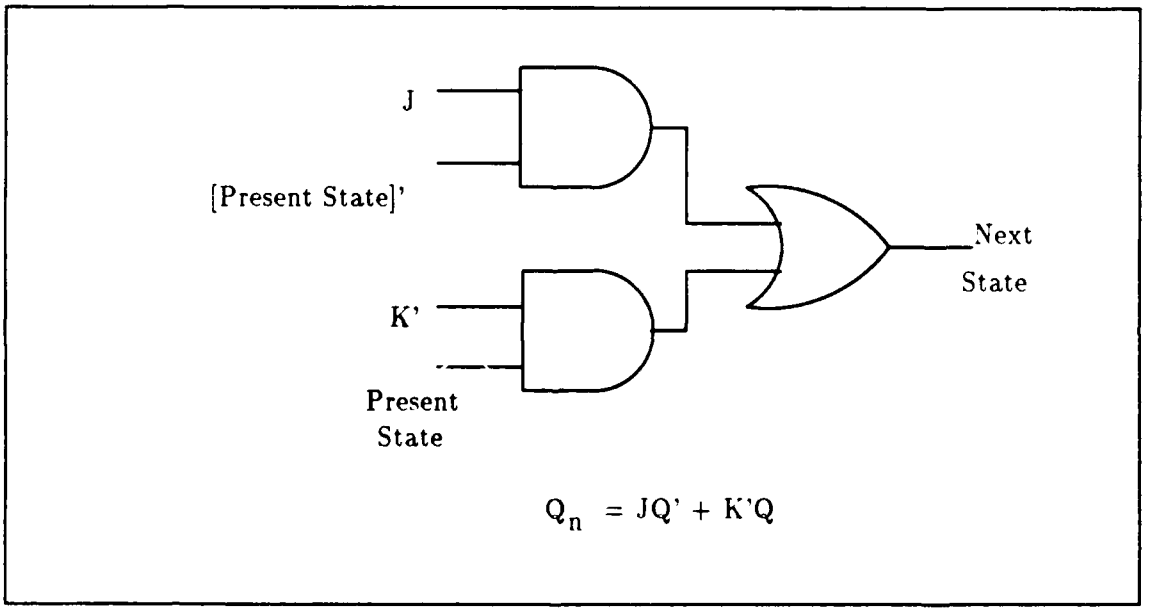


Figure 3.6. Pseudo JK Flip-flop

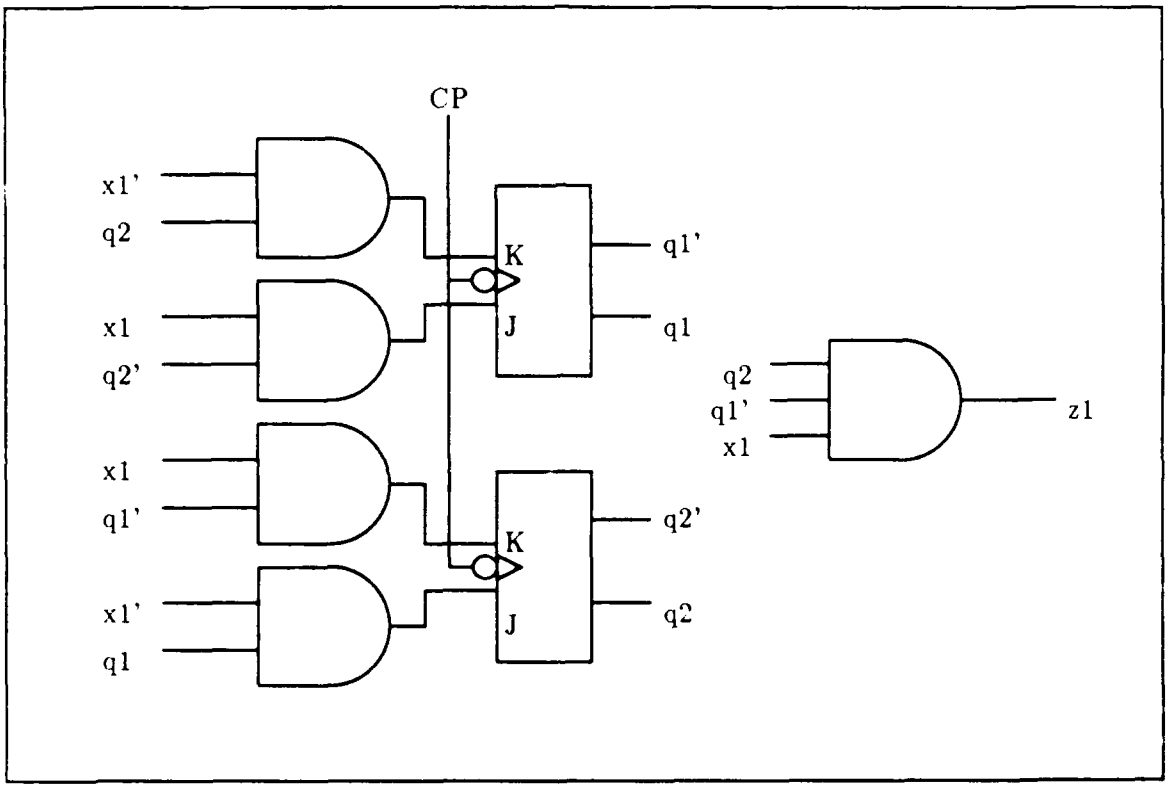


Figure 3.7. Example Sequential Circuit

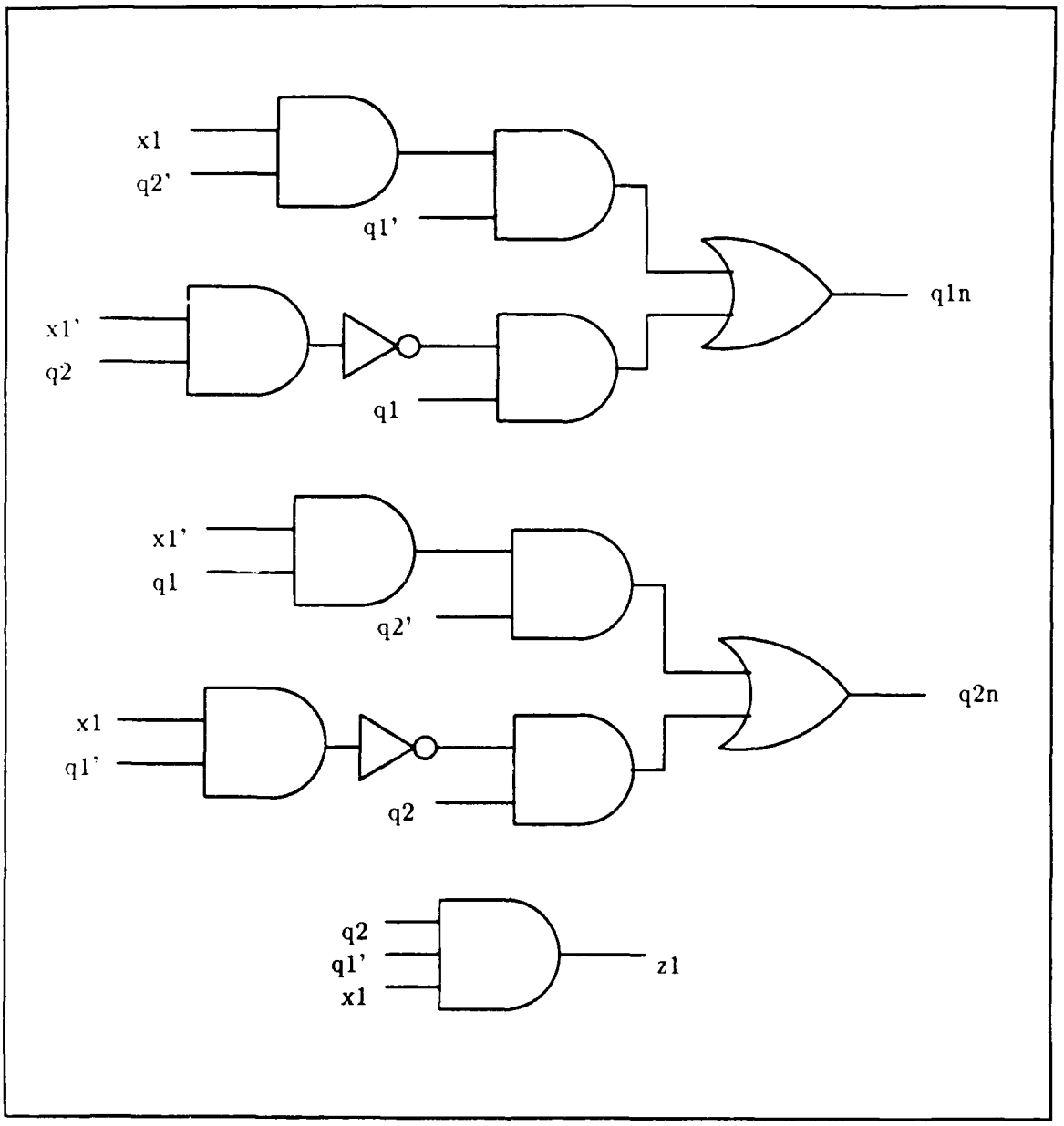


Figure 3.8. Flattened Example Circuit

for generating test vectors. For this example we will generate a vector capable of detecting a single stuck-at-one condition on node x_1. The following individual CCEs from the flattened circuit support the generation of an overall CCE:

$$\begin{aligned} q_{1n} &= j_1 q_1' + k_1' q_1 \\ q_{2n} &= j_2 q_2' + k_2' q_2 \\ z_1 &= q_1' q_2 x_1 \\ j_1 &= x_1 q_2' \\ k_1 &= x_1' q_2 \\ j_2 &= x_1' q_1 \\ k_2 &= x_1 q_1'. \end{aligned} \tag{3.98}$$

Cutting node x_1 leaves

$$\begin{aligned} q_{1n} &= j_1 q_1' + k_1' q_1 \\ q_{2n} &= j_2 q_2' + k_2' q_2 \\ z_1 &= q_1' q_2 TEST \\ j_1 &= TEST q_2' \\ k_1 &= TEST' q_2 \\ j_2 &= TEST' q_1 \\ k_2 &= TEST q_1'. \end{aligned} \tag{3.99}$$

This system of equations is collapsed into one equation just as before, using the exclusive-nor operator and multiplication, resulting in

$$\begin{aligned} &j_1' k_1' j_2' k_2' q_1' q_2' q_{1n}' q_{2n}' z_1' TEST' + j_1' k_1' j_2' k_2 q_1' q_2 q_{1n}' q_{2n}' z_1 TEST + \\ &j_1' k_1 j_2' k_2' q_1' q_2 q_{1n}' q_{2n} z_1' TEST' + j_1' k_1 j_2 k_2' q_1 q_2 q_{1n}' q_{2n} z_1' TEST' + \\ &j_1' k_1' j_2' k_2' q_1 q_2 q_{1n} q_{2n} z_1' TEST + j_1 k_1' j_2' k_2 q_1' q_2' q_{1n} q_{2n}' z_1' TEST + \\ &j_1 k_1' j_2' k_2' q_1 q_2' q_{1n} q_{2n}' z_1' TEST = 1. \end{aligned} \tag{3.100}$$

The internal variables ($\underline{j}$ and $\underline{k}$ in this example) are disjunctively eliminated leaving the overall CCE

$$q_1' q_2' q_{1n}' q_{2n}' z_1' TEST' + q_1' q_2 q_{1n}' q_{2n}' z_1 TEST + q_1' q_2 q_{1n}' q_{2n} z_1' TEST' + \tag{3.101}$$

$$q_1 q_2 q_{1n}' q_{2n} z_1' TEST' + q_1 q_2 q_{1n} q_{2n} z_1' TEST + q_1' q_2' q_{1n} q_{2n}' z_1' TEST +$$

$$q_1 q_2' q_{1n} q_{2n}' z_1' TEST = 1.$$

In general the overall CCE for a sequential circuit test will be

$$\phi_{ccf-seq}(\underline{x}, \underline{q}, \underline{TEST}, \underline{z}, \underline{q_n}) = 1 \tag{3.102}$$

where $\underline{q}$ represents the present state of all memory elements, and $\underline{q_n}$ are the next states. The number of test variables in $\underline{TEST}$ depends on the type of fault being diagnosed (single stuck-at, bridge, multiple stuck-at).

The next step, as we continue to follow the original Cerny method for generating test vectors, is to derive an FCE for the circuit The FCE is a combination of the output equations in the circuit. To reiterate, the output equations are those equations that set a primary output equal to a function of the primary input variables that feed the output (directly or indirectly). For a flattened sequential circuit the next states of the memory elements are also primary outputs. The present states are primary inputs. The output equations for the example circuit are

$$\begin{aligned} q_{1n} &= x_1 q_1' q_2' + x_1 q_1 + q_1 q_2' \\ q_{2n} &= x_1' q_1 q_2' + x_1' q_2 + q_1 q_2 \\ z_1 &= x_1 q_1' q_2. \end{aligned} \tag{3.103}$$

These equations are combined yielding the FCE

$$x_1' q_1' q_2' q_{1n}' q_{2n}' z_1' + x_1' q_2 q_{1n}' q_{2n} z_1' + x_1 q_1' q_2 q_{1n}' q_{2n}' z_1 + \tag{3.104}$$
$$x_1' q_1 q_2' q_{1n} q_{2n} z_1' + x_1 q_2' q_{1n} q_{2n}' z_1' + x_1 q_1 q_2 q_{1n} q_{2n} z_1' = 1.$$

The general form of the FCE for sequential circuit diagnosis is

$$\psi_{fcf-seq}(\underline{x}, \underline{q}, \underline{z}, \underline{q_n}) = 1. \tag{3.105}$$

The next step is to combine the overall CCE and the FCE, which results in the equation

$$x_1' q_1' q_2' q_{1n}' q_{2n}' z_1' TEST' + x_1 q_1' q_2 q_{1n}' q_{2n}' z_1 TEST + x_1' q_2 q_{1n}' q_{2n} z_1' TEST' + \tag{3.106}$$
$$x_1 q_2' q_{1n} q_{2n}' z_1' TEST + x_1' q_1 q_2' q_{1n} q_{2n} z_1' TEST' + x_1 q_1 q_2 q_{1n} q_{2n} z_1' TEST = 1.$$

The outputs (including the pseudo outputs $\underline{q_n}$) are then eliminated to get the OCE for the given suspected faulty line yielding

$$x_1' q_1' q_2' TEST' + x_1 q_1' q_2 TEST + x_1' q_2 TEST' + x_1 q_2' TEST + \tag{3.107}$$
$$x_1' q_1 q_2' EST' + x_1 q_1 q_2 TEST = 1.$$

The general OCE for a sequential circuit is

$$\Theta_{ocf-seq}(\underline{x}, \underline{q}, \underline{TEST}) = 1. \tag{3.108}$$

The left-hand-side of this equation is complemented and set equal to one. For the example this yields

$$x_1 TEST' + x_1' TEST = 1. \tag{3.109}$$

In general

$$\Theta'_{ocf-seq}(\underline{x}, \underline{q}, \underline{TEST}) = 1. \tag{3.110}$$

The next step to test for a single stuck-at-one fault is to substitute one into this equation for the test variable. The following equation results from the substitution for the example circuit

$$x_1 = 1. \tag{3.111}$$

In general this step depends on the type of fault being diagnosed (details are found in the respective sections of this Chapter for each type of fault). As shown in this example a single stuck-at fault requires the substitution of a logic value (0 for the $SA0$ test, 1 for the $SA1$ test) for the single test variable in equation (3.110). For bridge fault diagnosis there are two test variables and equation (3.110) is used to form a new equation that describes the effects of having the two lines bridged. Multiple stuck-at fault diagnosis involves any number of test variables . Each of the test variables in (3.110) are substituted with the logic values that they are suspected to be stuck at.

Expanding equation (3.111) with respect to the pseudo input variables that are not present ($\underline{q}$) we get

$$x_1 q_1 q_2 + x_1 q_1' q_2 + x_1 q_1 q_2' + x_1 q_1' q_2' = 1. \tag{3.112}$$

Referencing equation (3.111) we have an example that turns out to be rather trivial. The variables ($\underline{q}$) representing the current state of the circuit have completely dropped out of the derivation. As shown in the expansion, when they do all drop out then they can be equal to any logic values to begin testing. In this case the current state of the circuit is irrelevant when testing node x_1 for a single stuck-at fault. It is important to note that on the average the variables ($\underline{q}$) representing the current state of the circuit will not typically drop completely out equation (3.111). In the typical case, when they do survive in equation (3.111) the values that are generated represent a state that the circuit must be in to begin testing. For example, if (3.111) were

$$x_1 q_1 q_2' = 1 \tag{3.113}$$

then the current state of the memory elements would have to be one and zero, respectively before applying the vector $x_1 = 1$.

In any case we see the need to know the current state of the circuit before testing can begin. There are two ways to proceed. The first way is to attempt vector generation by directing the user to read the current values of the memory elements and enter them. The values are substituted into equation (3.112) and the minterm that remains will yield a test vector. The second way to proceed is attempted if the first fails. This alternative takes advantage of the fact that all well designed sequential circuits have a means of being cleared or reset. The user is directed to reset the circuit and input the values that result (typically all zeros). These values are substituted in equation (3.112) and the resulting minterm will yield a vector.

In the example circuit we will begin by assuming that the current values of memory variables q_1 and q_2 are zero and one respectively. These logic values are subsituted into equation (3.112) for the memory variables resulting in

$$x_1 = 1. \tag{3.114}$$

In this case a vector has been generated. If the left-hand-side had reduced to zero we would then reset the circuit and substitute the resulting values of the memory elements (assume zero for both) as follows

$$x_1 = 1. \tag{3.115}$$

The minterm can be taken from the left-hand-side of (3.114) and used to generate a vector. The vector is then applied. Note that if the equation (3.113) had been generated instead of (3.111) then neither of the two attempts would have resulted in a test vector.

Input/Output Experiment and Analysis of Results. Once generated the test vector is applied and the outputs are read as before. After application the outputs of the sequential circuit are in there "next" states and are read and fed back in as primary output results. Terms combining the vector and result of application are formed as before. Only this time before comparison to the original circuit equations the states of the memory variables prior to application are substituted into the original circuit equations. For valid results the original equations must be initialized to match the configuration of the circuit that existed before the vectors were applied. This step is best illustrated by example. In the example the output equations are modified using the exclusive-nor operator to get equations that set functions of the inputs and output variables equal to one as follows

$$\begin{aligned} x_1'q_1'q_{1n}' + x_1'q_2q_{1n}' + q_1'q_2q_{1n}' + x_1q_1q_{1n} + q_1q_2'q_{1n} + x_1q_2'q_{1n} &= 1 \\ x_1'q_1q_{2n} + x_1'q_2q_{2n} + q_1'q_2'q_{2n}' + x_1q_1'q_{2n}' + q_1q_2q_{2n} + x_1q_2q_{2n} &= 1 \\ x_1'z_1' + x_1q_1'q_2z_1 + q_1z_1' + q_2'z_1' &= 1. \end{aligned} \tag{3.116}$$

These equations are initialized with zero and one for the memory variables q_1 and q_2 (since these are the values they held before applying the test vector), respectively to get the following equations:

$$\begin{aligned} q_{1n}' &= 1 \\ x_1 + q_{2n} &= 1 \\ x_1 + z_1' &= 1. \end{aligned} \tag{3.117}$$

These are the equations that are compared to the results gained from applying the test vector.

Another Approach to Vector Generation

This section describes a method that yields the same results as Cerny when one output variable exists. Two descriptions of a given circuit are developed that relate a single

circuit output to the primary inputs. To get the first description we begin by choosing an output accessible by the suspected faulty node (which we will call the test node) and ignore all other circuit outputs. A relationship between this output, the primary inputs and the test node is derived. This relationship uses Cerny's "cut node" idea by ignoring any logic previous to the suspect node and replacing the node with a test variable. The second description is just the basic Boolean equation that relates the primary inputs to the output chosen for the first description.

We will use the example from the single fault vector generation section to explain the process. The suspected faulty node is y_1. Derivation of the first circuit description begins, like Cerny's method, by identifying the individual CCEs of the circuit. These are

$$\begin{aligned} y_1 &= x_1' \\ y_2 &= x_2' \\ z_1 &= y_1' + x_2' \\ z_2 &= y_1' + y_2'. \end{aligned} \tag{3.118}$$

Reviewing this system of equations shows that the test node has access to both outputs. Output z_1 is arbitrarily chosen to form the two circuit descriptions. We will call the equation that has this output as its left-hand-side the output equation. With this in mind all other equations involving outputs are deleted yielding

$$\begin{aligned} y_1 &= x_1' \\ y_2 &= x_2' \\ z_1 &= y_1' + x_2'. \end{aligned} \tag{3.119}$$

Also deleted are those equations that relate the test node to logic feeding the test node leaving

$$\begin{aligned} y_2 &= x_2' \\ z_1 &= y_1' + x_2'. \end{aligned} \tag{3.120}$$

The next modification to the system of equations is to substitute the internal variables in the output equation with any logic functions, equal to these internal variables, that remain in the system of equations. For the example none remain, however if the variable x_2 in the output equation had been a y_2 then this variable would have been replaced with x_2'.

The next step is to delete all equations except for the output equation leaving

$$z_1 = y_1' + x_2'. \tag{3.121}$$

The last step is to replace the suspected faulty node with a test variable as follows

$$z_1 = TEST' + x_2'. \tag{3.122}$$

In general the first description is

$$z_i = f_1(\underline{x}, TEST). \tag{3.123}$$

The second description is the chosen output set equal to the function of primary inputs that feed it. This is

$$z_1 = x_1 + x_2'. \tag{3.124}$$

In general the second desription is

$$z_i = f_2(\underline{x}). \tag{3.125}$$

For a fault to be detectable the faulty description (right-hand-side of 3.123) must exhibit an opposite function than the fault-free description (right-hand-side of 3.125). This point is mathematically realized by setting the negated right-hand-side of (3.123) equal t the right-hand-side of (3.125). For the example

$$(TEST' + x_2')' = x_1 + x_2'. \tag{3.126}$$

Simplifying this equation, using the exclusive-nor operator, yields a function of the inputs and test variable set equal to one as follows

$$x_1' x_2 TEST' + x_1 x_2 TEST = 1. \tag{3.127}$$

In general we have derived what will be called a diagnostic circuit equation (DCE). The DCE is

$$\Psi_{dcf}(\underline{x}, TEST) = 1. \tag{3.128}$$

As we did in Cerny's method substitution of zero and one, respectively, for the test variable will yield minterms that lead to vector(s) capable of detecting stuck-at-zero and one conditions. In the example the result of subtituting zero and one leads to the following two equations

$$x_1' x_2 = 1, (stuck - at - zero) \tag{3.129}$$

$$x_1 x_2 = 1, (stuck - at - one). \tag{3.130}$$

Note that this result is slightly different than the Cerny result given that the Cerny method uses both outputs to sensitize a path for the suspected faulty node.

IV. Mathematical Development of Extensions to Kainec's Diagnostic System

This Chapter discusses the extension to the Kainec diagnostic system (18) which has been achieved. The extension incorporates the capability to detect stuck-at faults in multiple output circuits. The mathematical basis for the development of the extension begins with a discussion of Kainec's original stuck-at fault diagnostic system. Following this review the method for diagnosing multiple output circuits is developed.

Stuck-at Fault Diagnosis

This section reviews the original diagnostic system developed by Kainec for diagnosing multiple stuck-at faults in combinational circuits.

Checkpoint Fault Model. Kainec's system tests for multiple (as well as single) stuck-at fault conditions by designating critical points in a circuit as checkpoints. Checkpoints are defined by Bossen and Hong to be fan-out branches of lines which fan out, and primary input lines that do not fan-out (5:1252). Figure 4.1 identifies the checkpoints for an example circuit. The smaller boxes in this figure label the circuit's checkpoints, and are known as checkpoint logic gates. Checkpoint logic gates form the basis for developing a checkpoint model to describe the possible stuck-at fault conditions of a particular line.

Figure 4.2 shows Kainec's revised version of the checkpoint model for stuck-at faults originally developed by Bossen and Hong (5:1253–1254). For each checkpoint in a circuit the revised model introduces two checkpoint variables[1] to describe the three possible states of the line: normal, stuck-at-0 or stuck-at-1. The elements of Figure 4.2 lead to the following equations:

$$c_{out} = c_1 + c_0' x_{in}, \tag{4.1}$$

$$c_0 c_1 = 0. \tag{4.2}$$

[1]The original model, which introduces three variables, was improved upon by Kainec (18:65–68).

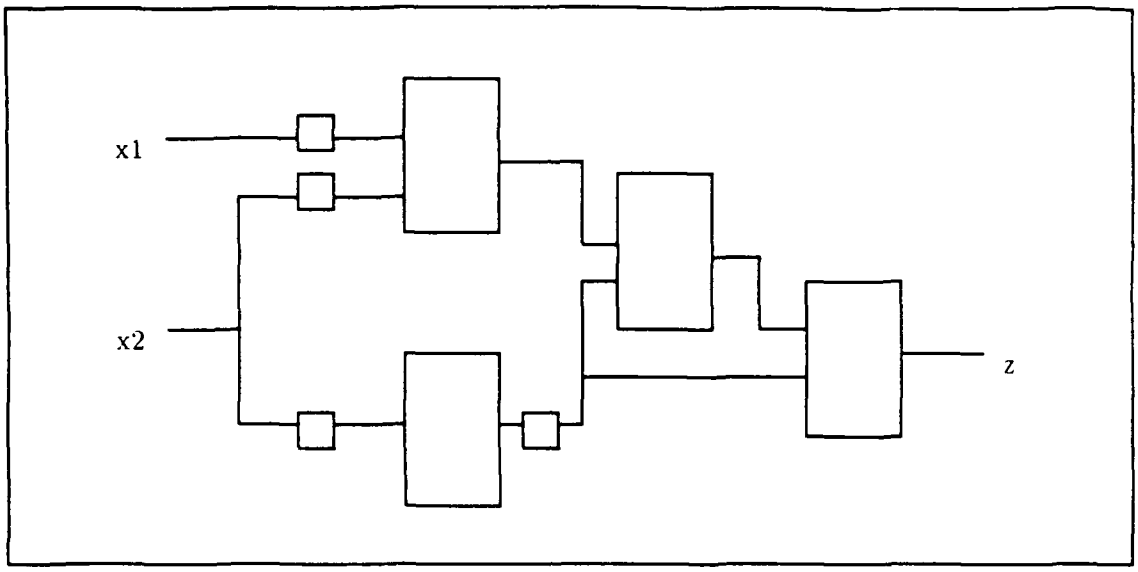
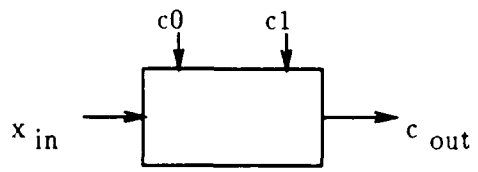


Figure 4.1. Checkpoint Placement

Consequently in a description of a circuit containing a node labeled x (where x is a checkpoint of the circuit), x is replaced by the right-hand-side of equation (4.1) to model the possible stuck-at fault conditions on x. Equation (4.2) establishes the fact that x, and therefore any checkpoint in general, can't be both stuck-at-zero and stuck-at-one. This equation represents a constraint introduced by the checkpoint model and must become part of the circuit description.

Derivation of Characteristic Equation. Having chosen the checkpoint model to describe the possible conditions of the critical points in the circuit, the next step is to develop a characteristic equation for the circuit (18:69–77). The characteristic equation describes the circuit in terms of the primary inputs, checkpoint variables and the circuit output. Once developed it is manipulated to generate test vectors, to determine the location of faults by deducing the logic states of the checkpoint variables, and to find the actual function of the circuit.



a) Checkpoint Logic Gate

c0 c1	Output	Node Conditions
0 0	x_{in}	Normal
0 1	1	Stuck-at-1
1 0	0	Stuck-at-0
1 1	—	Cannot Occur

b) Logic Gate Truth Table

x_{in} \ c0 c1	00	01	11	10
0	0	1	d	0
1	1	1	d	0

$$c_{out} = c1 + c0' \; x_{in}$$
$$c0c1 = 0$$

c) Karnaugh Map

Figure 4.2. Checkpoint Model

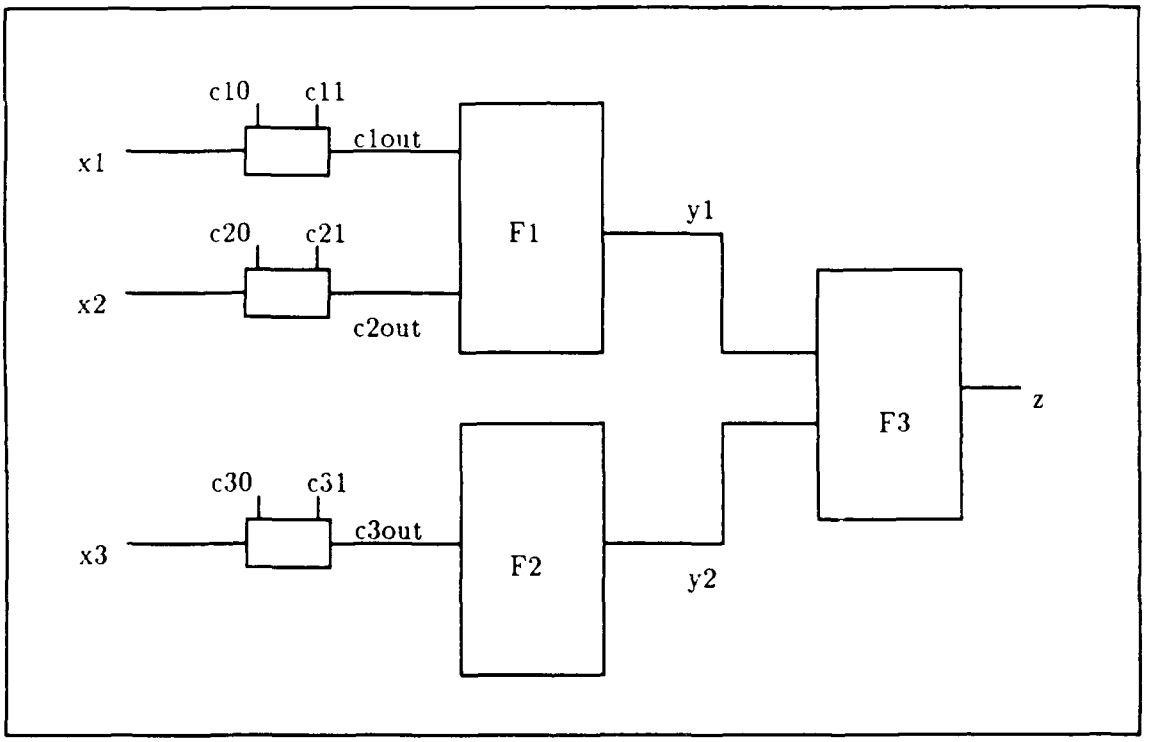


Figure 4.3. Example Circuit

Using Figure 4.3 as an example, the derivation begins with descriptions of each module in the circuit:

$$\begin{aligned} z &= f_3(y_1, y_2) \\ y_1 &= f_1(c_{1out}, c_{2out}) \\ y_2 &= f_2(c_{3out}) \\ c_{1out} &= c_{11} + c'_{10}x_1 \\ c_{2out} &= c_{21} + c'_{20}x_2 \\ c_{3out} &= c_{31} + c'_{30}x_3. \end{aligned} \tag{4.3}$$

Using equations (A.30), (A.48) and (A.49) this system of equations can be combined in a form that sets a function of the inputs, checkpoint variables and output equal to zero.

$$\begin{aligned} z \oplus f(y_1, y_2) + \\ y_1 \oplus f(c_{1out}, c_{2out}) + \\ y_2 \oplus f(c_{3out}) + \\ c_{1out} \oplus (c_{11} + c'_{10}x_1) + \\ c_{2out} \oplus (c_{21} + c'_{20}x_2) + \\ c_{3out} \oplus (c_{31} + c'_{30}x_3) = 0. \end{aligned} \tag{4.4}$$

As in equation (4.2) the following equations constrain the characteristic equation when using the checkpoint fault model.

$$\begin{aligned} c_{11}c_{10} &= 0 \\ c_{21}c_{20} &= 0 \\ c_{31}c_{30} &= 0. \end{aligned} \tag{4.5}$$

These equations are appended to equation (4.5) using equations (A.48) and (A.49).

$$
\begin{aligned}
z \oplus f(y_1, y_2) + \\
y_1 \oplus f(c_{1out}, c_{2out}) + \\
y_2 \oplus f(c_{3out}) + \\
c_{1out} \oplus (c_{11} + c'_{10}x_1) + \\
c_{2out} \oplus (c_{21} + c'_{20}x_2) + \\
c_{3out} \oplus (c_{31} + c'_{30}x_3) + \\
c_{11}c_{10} + c_{21}c_{20} + c_{31}c_{30} = 0.
\end{aligned} \tag{4.6}
$$

As noted before the characteristic equation is a function of the inputs, checkpoint variables and output. Consequently all internal variables are eliminated using the conjunctive eliminant defined in Appendix A. The result of eliminating $c_{1out}, c_{2out}, c_{3out}, y_1$, and y_2 from equation (4.7) leaves the following general form of the characteristic equation:

$$\Phi(\underline{x}, \underline{c}, z) = 0. \tag{4.7}$$

Generation of Vectors. An effective test vector is an input-vector that provides information about the circuit output that could not be gained prior to application of the vector (18:77). Therefore in the search for effective vectors the checkpoint variables are logically eliminated from equation (4.7) using the conjunctive eliminant leaving the equation:

$$\Theta(\underline{x}, z) = 0, \tag{4.8}$$

where

$$\Theta(\underline{x}, z) = ECON(\Phi(\underline{x}, \underline{c}, z), \underline{c}). \tag{4.9}$$

A function $i(\underline{x})$ is defined by

$$i(\underline{x}) = EDIS(\Theta(\underline{x}, z), z). \tag{4.10}$$

Kainec has shown that any solution of the equation

$$i'(\underline{x}) = 1 \tag{4.11}$$

is an effective test vector (18:79).

Incorporation of Input/Output Experiment Results. This discussion shows how the result of a given test vector application is incorporated to later determine if a fault has been detected.

Given $m_i(\underline{x})$, an arbitrarily chosen minterm of $i'(\underline{x})$, the unique solution of

$$m_i(\underline{x}) = 1 \tag{4.12}$$

is an effective test vector (18:80).

The output z obtained from applying this vector will take on one of two logic values, $r \in \{0, 1\}$. Therefore,

$$m_i(\underline{x}) = 1 \quad \Rightarrow \quad z = r \quad r \in \{0, 1\}. \tag{4.13}$$

Equivalently,

$$m_i'(\underline{x}) = 0 \quad \Rightarrow \quad z \oplus r = 0 \tag{4.14}$$

(18:81). Using the Extended Verification Theorem, detailed in Appendix A, this relation is equivalent to the inclusion

$$z \oplus r \leq m_i'(\underline{x}). \tag{4.15}$$

By equation (A.10)

$$m_i(\underline{x}) \cdot (z \oplus r) = 0. \tag{4.16}$$

Therefore,

$$m_i(\underline{x}) \cdot z = 0 \quad (r = 0), \tag{4.17}$$

$$m_r(\underline{x}) \cdot z' = 0 \quad (r = 1) \tag{4.18}$$

(18:81).

Once one of these results is obtained it can be combined with the characteristic equation using equations (A.48) and (A.49). The resulting equation is then used to generate the next test vector. The process is iterated until no further information can be obtained from the process. This condition is evident when the input function $i(\underline{x})$ enlarges to the point that it eventually becomes identically equal to one (18:87). Also, after all of the possible iterations are accomplished a final characteristic equation exists that includes the original characteristic equation and the information gained from all vector applications.

Interpretation of Results. When all possible information has been gained from vector generation and application the next step in Kainec's method is to use this information to determine the actual circuit function (18:87-90). Once this is determined it can then be used to identify the states of the checkpoint variables, and hence the locations of faults.

The final characteristic equation $\Phi_n(\underline{x}, \underline{c}, z) = 0$, which includes the initial characteristic equation (4.7) and the results of all vector applications, is used to arrive at a function $\Theta(\underline{x}, z)$ that relates the inputs to the output of the circuit:

$$\Theta(\underline{x}, z) = ECON(\Phi_n(\underline{x}, \underline{c}, z), \underline{c}). \tag{4.19}$$

Kainec shows that the function that the circuit is actually performing, which is called $F(\underline{x})$, is obtained by setting z equal to zero in $\Theta(\underline{x}, z)$ (18:88,90).

To determine the possible checkpoint states a function $G(\underline{c})$ is defined as

$$G(\underline{c}) = ECON(\Phi_n(\underline{x}, \underline{c}, z), z). \tag{4.20}$$

The possible checkpoint variable states are found as solutions to (18:92)

$$G'(\underline{c}) = 1. \tag{4.21}$$

Extensions for Multiple Output Circuit Diagnosis

From a strictly mathematical (versus implemental) viewpoint Kainec's procedures can be used to test multiple output circuits by simply changing the scalar output z in his derivations to a vector $\underline{z}$. The primary mathematical change described here is done to take advantage of the capability to choose an optimal vector from a group of effective test vectors. This capability is apparently specific to circuits with multiple outputs. Currently no method exists for choosing an optimal vector when diagnosing single output circuits.

As mentioned before, an effective vector is one that provides information about the circuit output(s) that cannot be deduced prior to application of that particular vector. An optimal, or "best", effective vector is one which is part of an experiment that will minimize the number of total test vector applications required to diagnose the faults in a circuit. At each iteration of vector generation a standard can be used to choose the best vector from a group of effective test vectors.

The following discussion will show how and why this standard can be used when diagnosing multiple output circuits as opposed to single output circuits.

Single Output Generation. Kainec's single output vector generation procedure will typically produce a set of effective vectors at each iteration of vector generation (18:80). Jumping ahead in the previously described process for test vector generation equation (4.8) is repeated below. It describes the relationship between the primary inputs and the output after an iteration of the vector generation process.

$$\Theta(\underline{x}, z) = 0. \tag{4.22}$$

Up to this point the only difference between single and multiple output diagnosis is the scalar output z versus the vector $\underline{z}$.

The next step disjunctively eliminates the output variable from the left-hand-side of equation (4.22) to get the input function $i(\underline{x})$. Kainec has shown that the input function is equal to zero (18:79)

$$i(\underline{x}) = 0. \tag{4.23}$$

Complementing this equation results in the following equation:

$$i'(\underline{x}) = 1. \tag{4.24}$$

In single output diagnosis manipulation of equation (4.22) results in (4.24), which involves only those terms from (4.22) that have no information regarding their relationship with the output variable. This point is llustrated by taking a specific example of (4.22), namely

$$abz' = 0 \tag{4.25}$$

where a and b are input variables and z is the output variable. By expanding this equation with respect to the input variables all of the possible input combinations associated with the equation are shown without changing the equation. This is not part of Kainec's original vector generation procedure, but makes it easier to see what is occurring.

$$ab(z') + ab'(0) + a'b(0) + a'b'(0) = 0. \tag{4.26}$$

Equation (4.26) shows that the first term, abz', has complete information relating the specific input combination $a = 1$, $b = 1$ to the output z. Setting the term equal to zero (using equation A.32), $abz' = 0$ yields the information that $a = 1$, $b = 1$ and $z = 0$ is false. Therefore whenever this input combination is applied the output z will equal one. There is no need to apply this input combination because we already know what the result will be.

The last three terms, however, provide no relative information regarding the output. These are the terms that remain when we disjunctively eliminate the output z from the left-hand-side of equation (4.25), set the result equal to one, and complement the resulting equation. This is the process that was done to get equations (4.23) and (4.24). The result

of this process is

$$ab' + a'b + a'b' = 1.$$

Each term, when set equal to one and solved, will yield effective test vectors.

The main point to be seen here is that with a single output there are *two* alternative types of terms that exist in an equation such as (4.22) that relates inputs to output: terms that provide complete information regarding the relationship between a specific input combination and the output, and those that provide no output relationship. Equation (4.22) is manipulated to generate the latter.

Multiple Output Generation. Given the case of multiple outputs, the possible combinations of output variables provide *three* alternative types of terms in an equation relating inputs to outputs: terms that give complete information, terms that give partial information, and terms that give none.

Adding an output v to the example will illustrate these alternatives. Take the equation

$$a'v' + a'b'z = 0.$$

Expanding (a necessary addition to the procedure for multiple outputs), as before, with respect to the primary inputs

$$a'b'(v' + z) + a'b(v') + ab'(0) + ab(0) = 0. \tag{4.27}$$

In the first term of equation (4.27) both outputs are represented and therefore complete information is present. We know exactly what the values of the outputs will be if we apply the vector $a = 0$, $b = 0$. The second term has complete information regarding the relationship of the output v to the input combination $a = 0$, $b = 1$, but has no information describing the relationship with respect to z. This is a partial information term. The last two terms have no information at all. In this example the last three terms would form a

collection of terms that yield effective vectors. The last two terms form a class of terms that are least informative, and therefore yield vectors that will provide greater information once applied. Both are considered optimal.

In general, as more and more information is gained the list of candidates will reduce to two categories: complete information terms and partial information terms. The terms with complete information are not effective and can be ignored. Of the remaining terms the term(s) with the least partial information are optimal.

Mathematically, identifying the terms with the least information is a rank ordering task. The first step, noted previously, is to expand the left-hand-side of equation (4.22) with respect to the input variables. Since this does not change the function it remains set equal to zero. The second step is to complement this equation, which results in the expanded function set equal to one. The new equation is

$$\Theta'(\underline{x}, \underline{z}) = 1. \tag{4.28}$$

From the example

$$a'b'(vz') + a'b(v) + ab'(1) + ab(1) = 1. \tag{4.29}$$

Notice that the single output step of disjunctively eliminating the output has been skipped. The purpose of this step is to isolate the terms that have no output information. Given varying degrees of information this step could be detrimental. Consider the situation in which there are no terms that contain no information relative to the circuit outputs, but partial information terms do exist. The partial information terms, which yield the only effective vectors, will be lost and the diagnostic system will terminate under the assumption that there are no further effective test vectors.

The next step in arriving at a rank ordering of the terms that are candidates for test vector generation is to expand the resulting output terms, which are those inside parentheses in equation (4.29), with respect to the output variables. Each expansion will have a number associated with it that corresponds to the number of terms resulting from

expansion. The following list relates each input combination with the number of expansions described.

- a'b' - 1
- a'b - 2
- ab' - 4
- ab - 4

In general, the vector or vectors associated with the largest value represents the optimal effective vector or group of optimal effective vectors.

V. Implementation of Cerny-based Diagnostic System

This chapter describes the implementation of the extensions made to Cerny's process for generating test vectors. The mathematical basis for this implementation is detailed in Chapter three. The diagnostic system described here extends Cerny's work to allow the analysis of the results from each vector application. Another extension allows the testing of sequential circuits. The system is automated, where Cerny's original work is not.

Figure 5.1 shows the opening menu of the diagnostic system as seen by the user. Six options are provided for testing a circuit. In choosing one of the six testing options the user specifies the type of circuit being tested (combinational of sequential) as well as the class of fault being tested for (single stuck-at fault, bridge fault or multiple stuck-at fault). Regardless of the type of circuit being diagnosed or the targeted fault class, each diagnostic routine has the same underlying architecture. Each routine is decomposed into three functions: an input function, a vector generation function and an analysis function.

Software System Architecture

The three functions that form the architecture of each diagnostic routine are illustrated in Figure 5.2 along with the inputs and outputs for each function. The following subsections describe each function.

Input Function. The input function is exactly the same for each diagnostic routine option and is actually completed before the branch to a specific option takes place. The input format required is the same for all routines, as are the data elements that are extracted from the input and passed on to the vector generation function.

The input format requires a data file containing a system of Boolean equations that use the following set of AND, OR, XOR and NOT operators, respectively: *, +, !, ' (18:100–103). The operations NAND, NOR and XNOR can be represented using the first three operators combined with the NOT operator. The use of juxtaposition, with respect to a set of operands, in place of the AND operator is allowed (ie., $A * B$ can be represented as AB).

```
ENTER NUMERICAL CHOICE OF DIAGNOSTIC ROUTINE:

        1. SINGLE SA FAULT ANALYSIS - COMBINATIONAL CKT
        2. SINGLE SA FAULT ANALYSIS - SEQUENTIAL CKT
        3. BRIDGE FAULT ANALYSIS - COMBINATIONAL CKT
        4. BRIDGE FAULT ANALYSIS - SEQUENTIAL CKT
        5. MULTIPLE SA FAULT ANALYSIS - COMBINATIONAL CKT
        6. MULTIPLE SA FAULT ANALYSIS - SEQUENTIAL CKT
        7. EXIT
 1

**** Enter the filename of your input file ****

Enter the input filename - te.ckt
```

Figure 5.1. Diagnostic System Menu

The system of equations includes a subcircuit equation for each gate in the circuit. The subcircuit equations, which relate the contribution of each logic gate to the overall circuit function, are used to develop the circuit characteristic equation (CCE) detailed in Chapter three. In addition to the subcircuit equations an output equation must be included for each output in the circuit. Each output equation relates a given output to the primary inputs of the circuit. The output equations are used to develop the fault-free circuit equation (FCE) detailed in Chapter three. A circuit with two outputs and three gates, such as the one in Figure 5.3, would have the following five equations in the input file:

$$\begin{aligned} z_1 &= x_1' \\ z_2 &= yx_1 \\ y &= x_2 + x_3 \end{aligned} \tag{5.1}$$

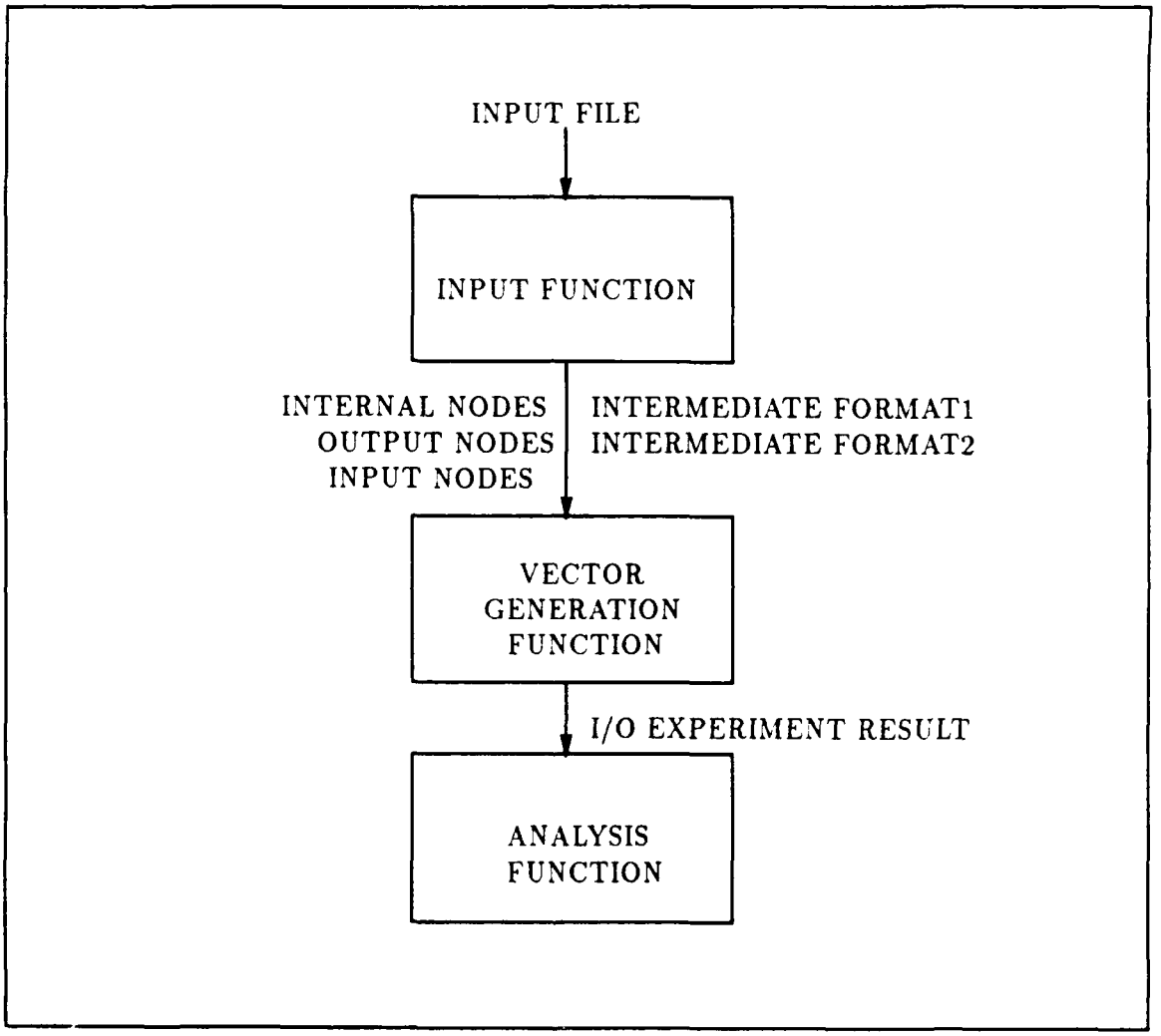


Figure 5.2. Architecture of Diagnostic Routines

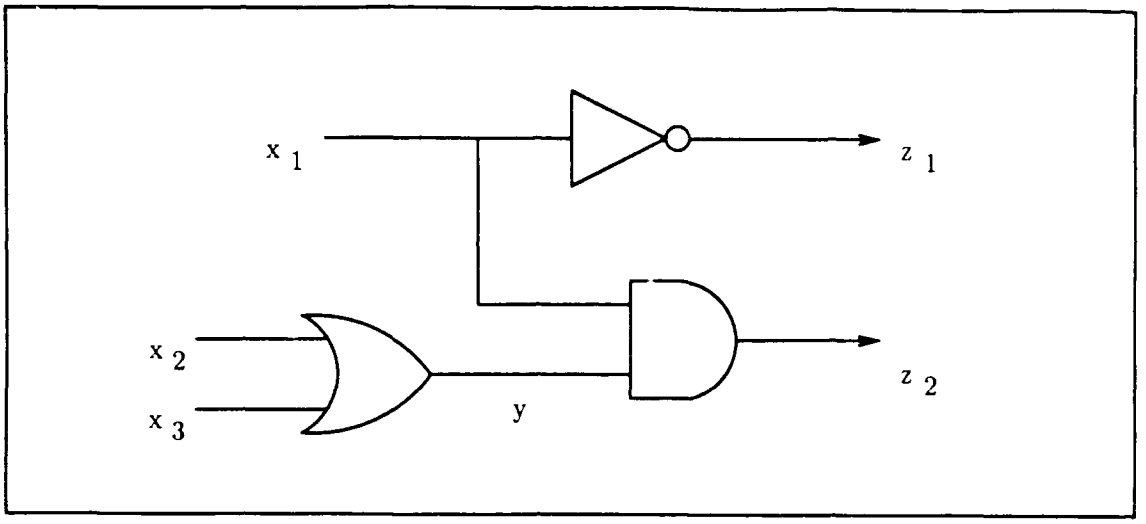


Figure 5.3. Example Circuit

$$z_1 = x_1'$$
$$z_2 = (x_2 + x_3') \cdot x_1.$$

The first three equations are subcircuit equations; the last two are output equations. Notice that all fanout nodes must be explicitly represented in each gate equation that the given fanout feeds (x_1 is a fanout node). Also notice that a subcircuit equation must be present for each gate regardless of its possible repetition in the data file.

The software prompts the user for the filename of the file containing the circuit descriptions (last line of Figure 5.1). After being read from the external file the input is used to generate several data elements that are passed to the vector generation function. Two "intermediate formats" are assembled. The intermediate format is a data structure that was first developed by Kainec (18:135) to maintain information about the structure and content of a given circuit. The first intermediate format constructed here contains the subcircuit equations described above. The second contains the output equations. For the example circuit of 5.3 the following intermediate format structures represent the subcircuit and output equations, respectively:

((EQ Z1 (NOT X1)) (EQ Z2 (* X1 Y)) (EQ Y (* X2 X3))),

((EQ Z1 (NOT X1)) (EQ Z2 (* X1 (+ X2 X3)))).

These data structures form lists that are consistent with Scheme syntax rules and order the necessary information in a way that simplifies manipulation.

In addition to the intermediate format structures, lists of the circuit's input variables, internal variables and output variables are generated. All lists are passed to the vector generation function.

Vector Generation Function. Though the vector generation function is accomplished by each type of diagnostic routine, the actual process varies depending on the type of fault and type of circuit. In this subsection the variations among the types of faults are discussed first, followed by the differences that exist when testing sequential, as opposed to combinational, circuits. The figures shown in this subsection refering to the fault-type differences correspond to the circuit in Figure 5.3. Those figures associated with the sequential circuit discussion refer to the circuit in Figure 5.7

As shown in Chapter three, the algorithm is slightly different when considering the three *types of faults* to be diagnosed (single stuck-at, bridge and multiple stuck-at faults). Therefore the implementation of the vector generation function differs among the six routines accordingly.

Single stuck-at fault diagnosis requires that one variable, identifying the suspected faulty node, be cut and replaced (by a "test" variable) when generating the CCE using the first intermediate format. The variable identifying the suspected faulty line is specified by the user following a prompt to the terminal screen, as shown in the first two lines of Figure 5.4. The CCE is combined with the FCE developed from the output equations from the second intermediate format, and the result is manipulated to get the output characteristic equation (OCE). The output characteristic function in this equation is complemented and set equal to one. The resulting equation, which we will call a modified OCE, is used to get the single stuck-at fault test vectors by replacing the test variable with logic one or

```
ENTER THE VARIABLE THAT LABELS THE SUSPECTED FAULTY      (1)
                    LINE:
y                                                        (2)
APPLY THE FOLLOWING VECTOR TO TEST A STUCK-AT-ZERO       (3)
CONDITION ON THE SUSPECTED FAULTY LINE:

X3 = 1                                                   (4)
X1 = 1                                                   (5)
X2 = 1                                                   (6)

INPUT THE RESULT FROM OUTPUT Z1 - 0 OR 1:                (7)
0                                                        (8)
INPUT THE RESULT FROM OUTPUT Z2 - 0 OR 1:                (9)
1                                                        (10)
APPLY THE FOLLOWING VECTOR TO TEST A STUCK-AT-ONE        (11)
CONDITION ON THE SUSPECTED FAULTY LINE:

X1 = 1                                                   (12)
X2 = 0                                                   (13)
X3 = 0                                                   (14)

INPUT THE RESULT FROM OUTPUT Z1 - 0 OR 1:                (15)
0                                                        (16)
INPUT THE RESULT FROM OUTPUT Z2 - 0 OR 1:                (17)
0                                                        (18)
LINE Y IS NORMAL.                                        (19)
WOULD YOU LIKE TO RUN A SINGLE FAULT TEST ON ANOTHER     (20)
NODE IN THE CIRCUIT?
TYPE y(rtn) OR n(rtn).
n                                                        (21)
```

Figure 5.4. Example Single Stuck-at Fault Test

zero (for stuck-at-one or stuck-at-zero tests respectively). The logic values are automatically replaced by the software. By replacing the test variable with the logic value zero in the modified OCE the set of vectors capable of detecting a stuck-at-zero condition are generated. One of these vectors is provided to the user for application (lines 3 through 6 of Figure 5.4. If application of the vector proves that the node is indeed stuck-at-zero, then testing stops and the node is reported as being stuck-at-zero. If not then the vectors capable of detecting the node stuck-at-one are generated by replacement of the logic value one. One is provided to the user for application (lines 11 through 14 of Figure 5.4).

Bridge fault diagnosis cuts and replaces two nodes, corresponding to the lines suspected to be bridged. The user is prompted for the variables labeling the suspected faulty lines (lines 1 through 3, Figure 5.5). Vectors are generated from an equation that combines two versions of the OCE. To be bridged the suspected lines must be simultaneously equal to zero or equal to one. This is represented in an equation containing a version of the OCE for the lines replaced with zeros and a version with the lines replaced by ones. The resulting equation is then solved to get the vectors capable of detecting the specified bridged condition. One vector is presented to the user for application (lines 4 through 7, Figure 5.5).

Multiple stuck-at diagnosis cuts and replaces n variables, where n is the number of variables suspected to be faulty. The routine prompts the user for the suspected faulty lines (lines 1 through 5, Figure 5.6). Logic values that are used for replacement in multiple fault diagnosis are specifically designated by the user via prompts to the terminal screen (lines 6 through 9, Figure 5.6). The logic values provided identify the suspected stuck-at values. The values are substituted into the equation containing the OCE for the n test variables. Solution of the resulting equation leads to the identification of vectors capable of detecting the exact fault situation proposed by the user. One vector is provided to the user (lines 10 through 13, Figure 5.6).

There are situtations when no possible vectors exist for a given specified test in each of the categories. When this occurs a message to that effect is sent to the terminal screen. All routines offer the user the opportunity to run the same routine on the same circuit for a different suspected faulty node.

```
ENTER THE VARIABLES THAT LABEL THE SUSPECTED                (1)
FAULTY LINES. ENTRIES SHOULD BE MADE ONE AT A
TIME WITH (rtn) TYPED BETWEEN EACH ENTRY.

x1                                                          (2)
y                                                           (3)

APPLY THE FOLLOWING VECTOR TO TEST A FAULT CONDITION        (4)
ON THE SUSPECTED FAULTY LINES:

X1 = 1                                                      (5)
X3 = 0                                                      (6)
X2 = 0                                                      (7)


INPUT THE RESULT FROM OUTPUT Z1 - 0 OR 1:                   (8)
0                                                           (9)
INPUT THE RESULT FROM OUTPUT Z2 - 0 OR 1:                   (10)
1                                                           (11)


LINES X1 AND Y ARE BRIDGED.                                 (12)

WOULD YOU LIKE TO RUN A BRIDGE FAULT TEST ON ANOTHER        (13)
SET OF NODES IN THE CIRCUIT?
TYPE y(rtn) OR n(rtn).
n                                                           (14)
```

Figure 5.5. Example Bridge Fault Test

```
ENTER THE NUMBER OF LINES SUSPECTED TO BE FAULTY.          (1)
FOLLOW THE RESPONSE WITH (rtn):

2                                                          (2)

ENTER THE VARIABLES THAT LABEL THE SUSPECTED               (3)
FAULTY LINES. ENTRIES SHOULD BE MADE ONE AT A
TIME WITH (rtn) TYPED BETWEEN EACH ENTRY.

x1                                                         (4)
x3                                                         (5)


ENTER THE SUSPECTED FAULT VALUE FOR VARIABLE X1:           (6)
0                                                          (7)
ENTER THE SUSPECTED FAULT VALUE FOR VARIABLE X3:           (8)
1                                                          (9)


APPLY THE FOLLOWING VECTOR TO TEST A FAULT CONDITION       (10)
ON THE SUSPECTED FAULTY LINES:

X3 = 1                                                     (11)

X2 = 1                                                     (12)

X1 = 1                                                     (13)

INPUT THE RESULT FROM OUTPUT Z1 - 0 OR 1:                  (14)
0                                                          (15)
INPUT THE RESULT FROM OUTPUT Z2 - 0 OR 1:                  (16)
1                                                          (17)

THE FOLLOWING LINES: X1 X3 ARE NOT STUCK AT THE            (18)
SUSPECTED VALUES.

WOULD YOU LIKE TO RUN A MULTIPLE FAULT TEST ON
ANOTHER SET OF NODES IN THE CIRCUIT?
TYPE y(rtn) OR n(rtn).
n                                                          (20)
```

Figure 5.6. Example Multiple Stuck-at Fault Test

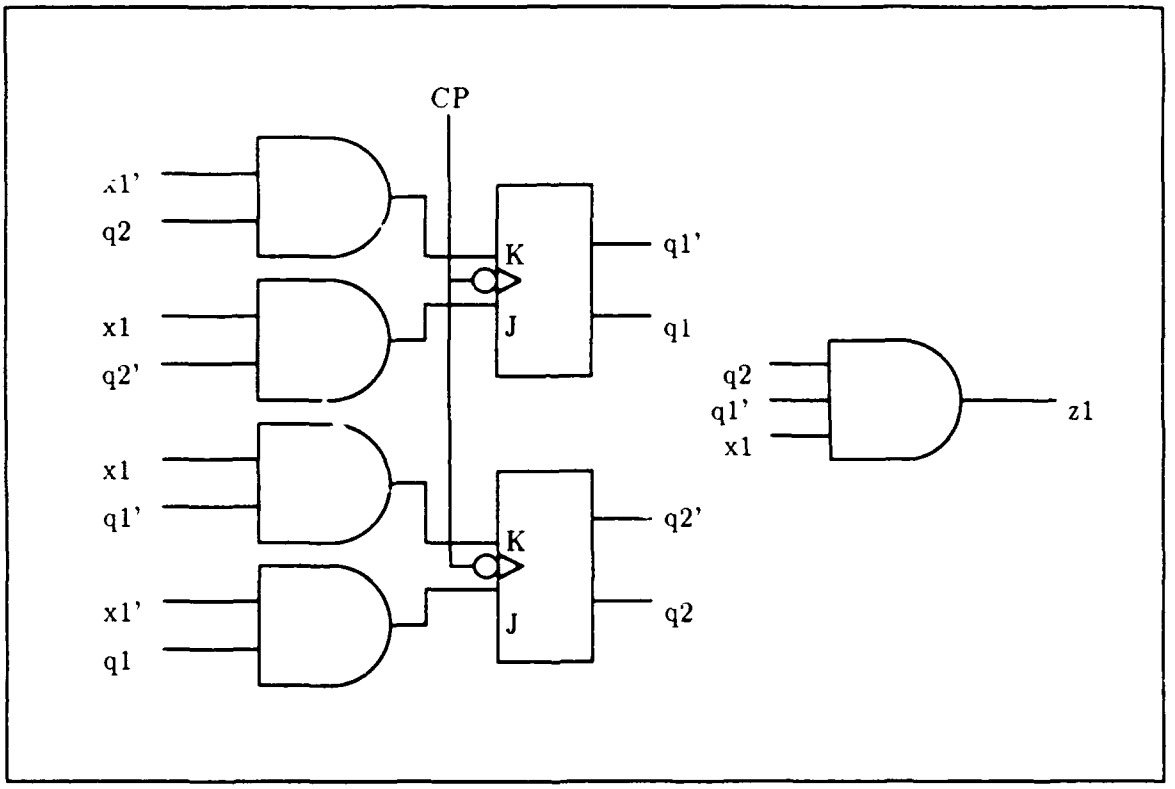


Figure 5.7. Example Sequential Circuit

To this point only combinational vector generation has been addressed. In the case of a *sequential diagnostic routine* (regardless of fault type), vector generation occurs in two stages. The *first stage* generates the modified OCE exactly the same way as in the combinational case. The modified OCE is the equation that includes the complemented version(s) of the OCE. It is this equation that is used to get vectors after replacement of logic values for the test variable(s). For sequential circuits, however, this equation will include the memory variables of the circuit because they have been converted to primary inputs during flattening of the circuit (explained in Chapter 3). To reiterate, the memory variables represent the current state of the memory elements contained in the circuit, and therefore the current state of the sequential circuit. The user is prompted for the identities, current values, and values upon circuit reset of these nodes (lines 3 through 14, Figure 5.8). In the *second* stage the current values of these elements are substituted into the equation from stage one. The substitution should result in a minterm that is then set equal to one to yield a test vector. If it does not then the reset values are subtituted into the modified OCE (the same equation that the first values were substituted into). If this process results in a test vector then the user is directed to reset the circuit. The resulting vector is provided to the user for application.

The output of the vector generation function is a term (or terms if multiple outputs are present in the circuit) that combines the vector applied with the result of application (reference Chapter three discussion regarding this combination). Each of the figures representing output to the user show the prompts to the user to input the resulting outputs following vector application. The following terms could result from the application of the stuck-at-zero vector listed in Figure 5.4:

X1 X2 X3 Z1

X1 X2 X3 Z2'

These terms indicate that both z_1 and z_2 were read as zero when the vector was applied. The result (or results as the case may be) is passed on to the analysis function to determine

```
ENTER THE VARIABLE THAT LABELS THE SUSPECTED FAULTY LINE:    (1)
x                                                            (2)
ENTER THE STATE VARIABLES OF THE CIRCUIT. THESE ARE THE      (3)
VARIABLES THAT LABEL THE OUTPUTS OF SEQUENTIAL ELEMENTS
BEFORE FLATTENING. ENTER THE VARIABLES ONE AT A TIME
FOLLOWED BY (rtn). ENTER "O"(rtn) WHEN DONE.
q1                                                           (4)
q2                                                           (5)
0                                                            (6)
ENTER THE CURRENT VALUE OF STATE VARIABLE Q1. TYPE           (7)
0 or 1 AND (rtn):
0                                                            (8)
ENTER THE CURRENT VALUE OF STATE VARIABLE Q2. TYPE           (9)
0 or 1 AND (rtn):
1                                                            (10)
ENTER THE VALUE OF STATE VARIABLE Q1 WHEN THE CIRCUIT IS     (11)
RESET.
0 or 1 AND (rtn):
0                                                            (12)
ENTER THE VALUE OF STATE VARIABLE Q2 WHEN THE CIRCUIT IS     (13)
RESET.
0 or 1 AND (rtn):
0                                                            (14)
```

Figure 5.8. Example Sequential Circuit Single SA Fault Test

```
APPLY THE FOLLOWING VECTOR TO TEST A STUCK-AT-ZERO CONDITION ON THE SUSPECTED FAULTY LINE:   (15)

X1 = 1                                                                                        (16)


INPUT THE RESULT FROM OUTPUT Z1 - 0 OR 1:                                                     (17)
0                                                                                             (18)
INPUT THE RESULT FROM OUTPUT Q1N - 0 OR 1:                                                    (19)
0                                                                                             (20)
INPUT THE RESULT FROM OUTPUT Q2N - 0 OR 1:                                                    (21)
1                                                                                             (22)

LINE X1 IS STUCK-AT-ZERO.                                                                     (23)


WOULD YOU LIKE TO RUN A SINGLE FAULT TEST ON ANOTHER NODE IN THE CIRCUIT?                     (24)
TYPE n(rtn) OR y(rtn).
n                                                                                             (25)
```

Figure 5.9. Example Sequential Circuit Single SA Fault Test, cont.

if a fault is presnt.

Analysis. Analysis compares a test result to the overall circuit equations to see if the result is logically included in the equations. Comparisons are individually made only with each equation that includes an output accessible by the node being diagnosed. In the case of sequential circuits the original equations must first be initialized with the values that represent the present state of the circuit prior to application.

Results of analysis that find that a term is not logically included are reported as faults, according to the type of test conducted (see Chapter three, Analysis of Input/Output Results).

Results

The software routines described in this chapter have been tested for accuracy and speed of operation on several circuits. Though development was done on an IBM-compatible computer (XT clone), testing was accomplished on a Sun-4 workstation.

The test for accuracy included two areas: generation of a correct test vector, and the correct diagnosis of a circuit. Considering the first area, if a vector was generated it was applied to the circuit in question (on paper) to insure that the suspected faulty node was excited correctly (0 if testing a stuck-at-one condition, 1 if stuck-at-zero) and also to insure that a path was sensitized from the node to a primary output. If a vector was not generated the circuit was reviewed to verify that there was no apparent test for the suspected fault. The second area, testing for correct diagnosis, verified that the results generated by the software were consistent with the results read from the outputs after the vector was applied.

The test for speed was done to determine the relative time it takes for the software to diagnose basic circuits (six to eight gates).

The following two subsections review the results for the two testing areas. The first subsection addresses the combinational diagnostic routines, the second discusses the sequential routines.

Results for Combinational Routines. In all test cases the diagnostic routines correctly generated test vectors and diagnostic results. For single stuck-at fault diagnosis there were no circuits that resulted in a "no test possible" message. This message is generated when no possible test exists for the specified fault condition. Typically most diagnostic sessions for bridge faults produce test vectors capable of detecting the specified faults. A noticeable failure to generate a bridge fault test happened with a particular 4:1 multiplexer. When any two of the inputs to be multiplexed were specified as fault nodes the routine failed to generate a test. The attempt to sensitize a path to the output for any two inputs involves setting the select inputs on the multiplexer to mutually-exclusive values. Of course this is the way the multiplexer operates but it hinders the capability to generate a bridge fault test using this software. Multiple fault tests reliably generate test vectors up to a point. As expected, when four or more nodes are suspected as being faulty it becomes harder and harder to generate a test for their particular configuration of suspected fault values.

For the test to evaluate time of operation, most combinational tests on circuits with ten or fewer gates ran in relatively "real time". By this I mean that the diagnostic process is continuous from a human's perspective. All user inputs are immediately followed by a response and appropriate prompt by the software for the next user input. The time to complete a diagnostic session is largely dependent on the time it takes a user to apply a test vector and report the results. The exceptions for combinational circuits are multiple output circuits with three or more nodes specified as being faulty. In these cases it takes two or three seconds to generate a test vector. For all combinational circuit tests circuits with 10 to 15 gates slows the response time to three to five seconds for vector generation. All in all the software runs very fast.

Results for Sequential Circuits. The sequential circuit tests were also successful from the standpoint of generating correct test vectors and providing accurate test results. On the whole the technique for generating vectors was successful given that only two circuit states were available to do so. Tests were generated about seventy percent of the time. The reset state that was specified most often cleared all sequential elements to zero. This

apparently is a good state to start from when testing.

Since the circuits are first converted to combinational circuits by the user the run times are very similar to the combinational cases described above. In other words the response times are the same. The time for overall testing is increased because more user inputs are required for sequential circuit tests.

VI. Implementation of Extensions to Kainec Diagnostic System

Given that Kainec's diagnostic system is already automated the implementation of the extensions uses the original software architecture as a basis. Changes are made to the original modules that form the architecture to incorporate the extension for testing multiple output circuits.

Original Architecture

The original architecture consists of an input module, an equation generation module, a tester module and an interpretation module (18:97). Figure 6.1 shows the four modules and the data passed between them.

Input Module. Kainec develops the "intermediate format" data structure to maintain the system of Boolean equations that is used to describe a circuit (18:98–103). The system of equations is provided by the user in an external data file. Equation operators for AND and OR operations are represented with conventional symbols (*, +). Juxtaposition can be used to replace the AND operator. The NOT operator is '. The XOR operator is !.

Equation Generation Module. This module accepts the intermediate format from the input module and uses it to generate the charateristic equation developed in Chapter 4 and shown again below (18:109–111).

$$\Phi(\underline{x}, \underline{y}, z) = 0. \tag{6.1}$$

In generating the characteristic equation checkpoint variables, $\underline{y}$, are combined with the system of equations from the intermediate format. The checkpoint variables are added according to the checkpoint model and circuit structure. As described before the checkpoint model designates checkpoints in the circuit at fanout branches of nodes that fan out, and nodes of primary inputs that do not fan out. The checkpoint model equations (4.1) and (4.2) introduce the checkpoint variables to the characteristic equation.

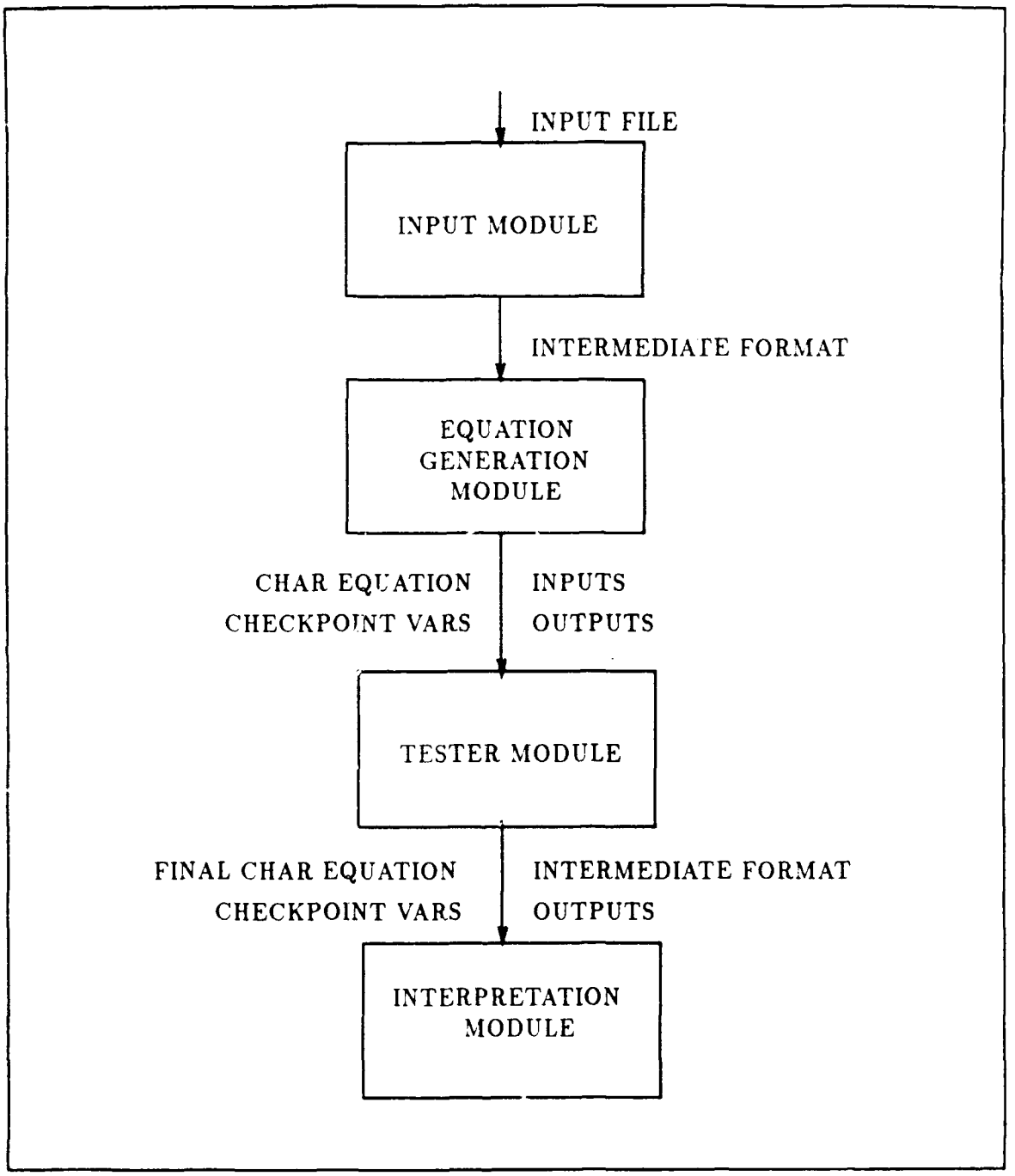


Figure 6.1. Original Kainec Diagnostic System Architecture

Also generated, in addition to the characteristic equation, are a list of the circuit's inputs, checkpoint variables and output. These four data elements are then passed on to the tester module.

Tester Module. The tester module uses the characteristic equation and associated lists from the equation generation module to generate test vectors (18:112–114). As vectors are generated and applied the results of each input/output experiment are added to the characteristic equation to support the generation of successive vectors. This iterative process continues until all possible information is obtained regarding the function of the circuit.

The main data element that is generated and passed to the interpreter module is the final characteristic equation. The final characteristic equation contains all of the information relevant to determining actual circuit function and supporting the diagnosis of existing faults. Additionally the number of tests conducted is passed to the interpretation module to provide a measure of efficiency for the diagnostic system.

Interpretation Module. The interpretation module (18:115–120) performs three functions. The first function generates the actual function of the tested circuit using the final characteristic equation along with circuit checkpoint variables and the output vairable. The second function uses the final characteristic equation and the output variable to deduce the definite and possible faults existing in the circuit. The third function generates metrics to support determination of system performance using the inputs and number of tests conducted. A performance ratio is calculated that compares the number of tests conducted to the number of tests possible.

Changes for Multiple Outputs

The two modules that are changed to accommodate testing of multiple output cricuits are the tester module and the interpretation module. Note that the changes do not alter the system's capability to diagnose faults in single output circuits.

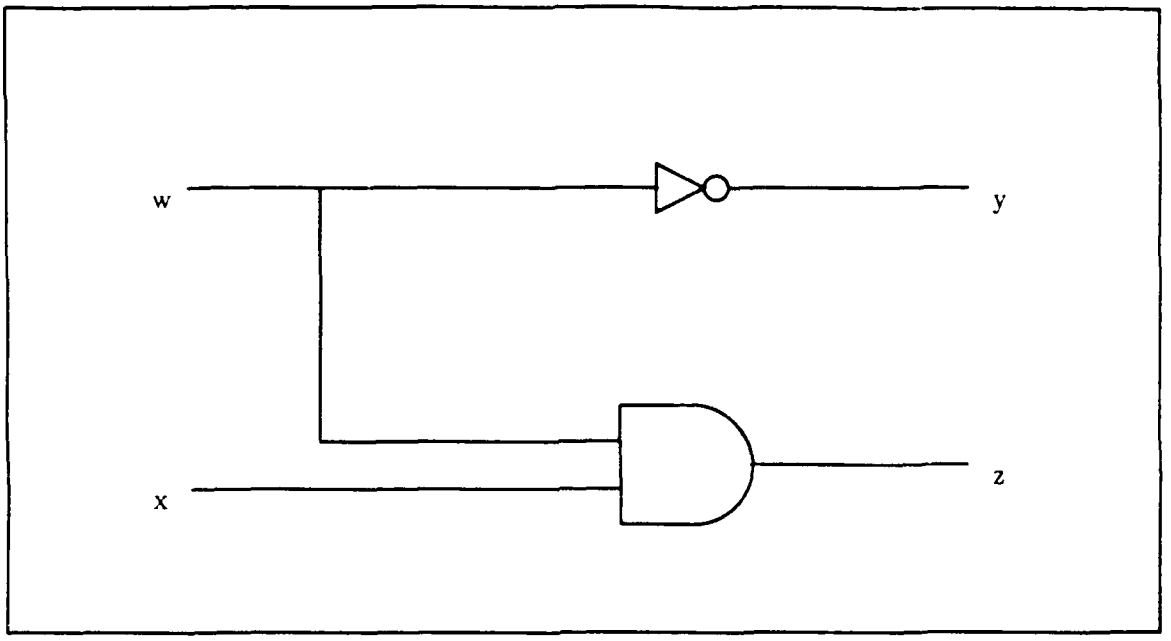


Figure 6.2. Example Multiple Output Circuit

Tester Module Changes. Changes to the tester module incorporate the capability to choose an optimum vector from a group of effective vectors. The process begins with equation (4.8), repeated below with one change; there is now a vector of outputs ($\underline{z}$) as opposed to a single output. This equation is the result of manipulating equation (6.1) to arrive at an equation that is a function of only circuit inputs and outputs.

$$\Theta(\underline{x}, \underline{z}) = 0. \tag{6.2}$$

Software routines added to the tester module expand the complement of this equation with respect to the input variables. The resulting expansion isolates all possible input combinations together with the information that each combination has with respect to the outputs. Each of these groups of information is then expanded by each of the outputs to determine which terms have the least amount of information concerning the state of the outputs in the presence of the respective inputs. The input vector terms associated with the greatest number of output expansions form an class of "best" effective vectors.

```
The Input Equation is: 0 = 0
The Suggested Input is:

    W = 0
    X = 0

If the output was 0, type 0 (rtn), else type 1 and
(rtn).
Enter the Result from Output Y- -                1

Processing....

Enter the Result from Output Z- -                0
```

Figure 6.3. Vector Application Prompt to User

The tester module is also altered to enable processing of multiple output results from the application of a given test vector. Figure 6.3 shows the screen output that the user sees when prompted to apply a particular vector. Notice the prompts to the user for reporting each of the results from the multiple outputs of the circuit shown in Figure 6.2 following application. All terms, which are constructed from an applied vector and its associated output results, are combined into a single function. This single function is then added to the characteristic equation. Previously with one output only one result was added to the characteristic equation.

Interpretation Module Changes. Changes to the interpretation module focus on extending existing functions to account for multiple circuit equations that result from having multiple outputs. The interpretation module derives the actual function and designed function for a given circuit using the final characteristic equation and the intermediate format, respectively. When there are several output for a given circuit several functions

The function(s) that the circuit was designed to perform is:

Y = W'

Z = W X

The function(s) that the circuit is performing is:

Y = W'

Z = W X

The actual circuit IS equivalent to the designed circuit.

Figure 6.4. Report of Results to User

exist corresponding to each output. The existing routines that are used to extract a single function are extended to extract multiple functions, when they exist, from the final characteristic equation and intermediate format. Figure 6.4 shows the report that the user sees upon completion of testing. Notice the two equations describing the designed function of the circuit.

Another function performed by the interpretation module is the comparison of actual and designed circuit functions. This process is extended to account for the possibility of having several functions in each category. All of the actual functions are combined using Boolean reasoning, as are the designed functions. The two resulting combined functions are then tested for equality.

Results

The additions made to Kainec's diagnostic system to allow the diagnosis of multiple output circuits have been tested for accuracy and speed of operation (relative to the original system). Testing was done on a Sun-4 workstation.

The test for accuracy was done to verify two things: the choice of an optimal vector from a set of effective test vectors, and correct diagnosis of circuits (both single and multiple output).

The test for speed was done to provide an estimate of the overhead, if any, that the extension adds to the original sofrware.

Accuracy. The extension works as planned. In all test cases the added software chooses an optimal vector. This was verified by running the software up to the point where vectors are generated. The effective vectors were dumped to the screen along with the choice of an optimal vector.

The system was then exercised as a whole and compared manually (on paper) to verify proper operation.

Speed. The original and extended software systems were run on single output circuits to compare speed of operation. There is no noticeable difference in speed of operation between the two.

VII. Conclusions and Recommendations

This chapter discusses the conclusions and recommendations resulting from this thesis. The first section summarizes the work that has been done. The second section assesses the contributions made in accomplishing the work. The last section identifies recommendations for future research related to this thesis.

Summary

The goal of this thesis was to further research in the area of Boolean-based digital circuit diagnosis. Extensions were made to two research efforts: Cerny's process for generating test vectors (7), and Kainec's diagnostic system(18).

Summary of Extensions to Cerny Research. Cerny developed a process to generate test vectors that are capable of diagnosing faults in combinational circuits. The process generates vectors to test nodes in a circuit for single stuck-at, multiple stuck-at, and bridge faults. Regardless of the type of fault, two circuit descriptions are formed and combined to yield an equation capable of producing test vectors. The circuit descriptions that are combined are the circuit characteristic equation (CCE) and the fault-free characteristic equation (FCE). The CCE describes a given circuit at the gate level, while isolating the identity of a suspected faulty node(s) which is specified by the user of the process. The FCE describes the circuit in terms of its primary inputs and outputs. The CCE is combined with the FCE to get the output characteristic equation (OCE). The OCE is then manipulated to produce an equation that yields vectors capable of detecting a fault on the specified node(s). Manipulation of the OCE is done in different ways, depending on the type of fault being diagnosed. The resulting equation has a function of the primary inputs set equal to one. Any of the minterms of this function can be set equal to one to yield a test vector.

Two extensions were made to Cerny's original process for generating vectors. The first extension adds the capability to automatically determine the presence or absence of the suspected fault, based on the results gained from applying the vector generated by

Cerny's process. To do this a term(s) is formed using two items: the minterm used to get the vector, and a representation of the value of the output that resulted from applying the vector. A term is formed for each output of the circuit (ie., minterm x_1x_2' and outputs $z_1 = 1$ and $z_2 = 0$ would yield the terms: $x_1x_2'z_1'$ and $x_1x_2'z_2$). Each term is then compared to its associated output equation. An output equation is a partial circuit description that includes a particular output and the primary inputs that feed that output. The comparison of the term(s) with the output equation(s) identifies the presence or absence of a fault.

The second extension to Cerny's work implements a diagnostic system for sequential circuits. Cerny's process is used to generate test vectors for a "flattened" sequential circuit. The flattening process converts a sequential circuit into a combinational circuit. This is done by the user prior to creating the input file that contains the circuit description. Flattening the circuit replaces its memory elements with the logic that realizes the characteristic equations of those memory elements. For example, a JK flip-flop is replaced by the logic function $jq' + k'q$, where q is the variable representing the flip-flops current state. The output of this logic would be q_n which represents the next state of the flip-flop. In taking this approach flattening also cuts the feedback path in the circuit which turns the variables representing the current state of the circuit into primary inputs. Cutting this path also creates additional outputs which are the next state variables of the circuit.

The process of generating test vectors using a flattened circuit identifies the values that the current state variables (one for each flip-flop) must be in order to sensitize a path from the site of the suspected fault to an output. Several vectors are typically generated. Each vector will be associated with one of the 2^n specific combinations of current state variables (where n is the number of memory elements). The process that was developed checks the set of generated vectors for two of the possible specific combinations of current state variables. The first possibility is the current state of the elements as determined by the user after probing the circuit. The current values of the current state variables are provided by the user and then are substituted into the equation that yields test vectors. If one of the vectors is associated with the existing state of the circuit then it will be generated. If not then the second alternative combination is tried. The second alternative is the state of the circuit after it has been reset. The user is asked for the values of the

current state variables when the circuit is reset. If the substitution of these values results in a vector then the user is directed to reset the circuit and apply this vector.

Analysis of the test results is conducted much the same way as in the combinational circuit case. With sequential circuits, however, the formation and comparison of terms also uses the new outputs that represent the next state of the memory elements. The outputs of the elements are probed after testing to get these values. Also, prior to the comparison that determines the presence or absence of a fault, the output equations for these next state variables must be initialized with the values that the current state variables held prior to testing.

Cerny's work was not automated. Following the mathematical derivation of the extensions the entire diagnostic system was automated resulting in a program with six separate routines for diagnosing three types of faults (single stuck-at, bridge and multiple stuck-at) in two types of circuits (combinational and sequential).

An alternative routine for generating test vectors was also explored.

Summary of Extension to Kainec Research. Kainec developed an automated diagnostic system for diagnosing multiple faults in combinational circuits with a single output. The approach derives a characteristic equation which is used to generated test vectors. The process goes through several iterations to diagnose all possible stuck-at faults in a circuit. At each iteration the characteristic equation generates a vector that yields information about the circuit output that was not previously known. This vector is applied by the user and the resulting output value is read back into the diagnostic system. This result is used to update the characteristic equation and therefore supports the next iteration for generating a test vector. After all possible information is gained from applying test vectors the last characteristic equation (which is the result of updating the equation used in the last iteration of vector generation with the result of applying the last vector) is manipulated to derive the states of the circuit's internal nodes. Determining the states of these nodes leads to the diagnosis of the circuit's faults. A comparison is also made to determine if the circuit's designed function matches the actual function.

The extension that was made to the Kainec system incorporates the capability to

diagnose faults in multiple output circuits. The appropriate software modules have been changed to account for multiple output equations. For example, the user is prompted to enter results from each output of a multiple output circuit after applying a particular vector. However, the extension was made primarily to take advantage of the fact that multiple output circuits offer the capability to choose an optimum test vector from a set of generated vectors. An optimum vector is one that provides the most information about the circuit outputs when applied. The software routines capable of choosing an optimal vector at each iteration of generation have been incorporated into the main system.

Assessment of Research.

The extensions accomplished in this thesis offer useful improvements specifically to the two existing research efforts (Cerny and Kainec) described, and also to the area of Boolean-based circuit diagnosis in general.

Extension and automation of the Cerny process created a complete diagnostic system capable of detecting faults on specified lines in a circuit. The extension for analyzing the results from applying a particular vector can be used with any of the Boolean-based methods described in Chapter two. The extension for sequential circuit diagnosis offers a significant addition to a very limited research area. A significant part of the diagnostic system is the capability to diagnose bridge faults as opposed to just the classical stuck-at faults.

Certain limitations exist with the Cerny-based diagnostic system. It is restricted to diagnosing circuits described at the gate level. In diagnosing a particular fault on a line the system assumes that the fault being diagnosed is the only fault present in the circuit. This is known as a single fault assumption. It also assumes that if an error is detected in testing, then the fault that has been specified is the cause of the error. This may not always be the case. Take for example the test of an AND gate output for a stuck-at-zero condition. Regardless of the location of the gate in the circuit, an input vector is produced to generate a one on this output by setting the inputs of the gate to one. This is known as exciting the suspected fault site. If the node is stuck-at-zero then the output will not set to one as it would in normal circuit operation and an error is detected. The

problem is that a stuck-at-zero fault on either of the inputs, while the output is normal, will cause the same error. The point is that the system actually isolates faults to a class of possible faulty nodes. A possible improvement to account for this fact is addressed in the recommendations section.

The extension made to Kainec's diagnostic system significantly expands its diagnostic capability to a larger group of circuits. The extension is very useful since integrated circuits typically have multiple outputs. The extension does not change the systems' capability to diagnose circuits with a single output.

Recommendations

There are a number of possible improvements that can be made that use the work done in this thesis as a starting point. Some of the recommendations address the diagnostic routines developed here in general. Other recommendations are specific to the individual extensions that were done.

General Improvements. In general the routines that were programmed as part of this thesis were developed with the primary goal of correct operation. The routines are probably not as optimal as they can be with respect to speed of operation. Analyzing and reworking the software should lead to significant improvements in the speed in which circuits are diagnosed.

Another general improvement that could be made concerns the input modules of each system. The input modules should be changed to accept the user's system of equations describing the circuit via prompts to the terminal screen. This modification would make each diagnostic system completely interactive, eliminating the need to construct an external data file for each circuit to be tested. Another advantage of this improvement is that it would make the systems more practical for use on large circuits that have been partitioned to simplify testing. If a user chooses to test one area of a circuit at a time, given access to internal nodes, he/she can type in the descriptions of these areas as they need to be tested. The interactive nature of the modification eliminates the need to continually exit and enter the diagnostic systems to create an input file for the area of the circuit that the

user decides should be tested next. This improvement should be implemented such that the user has an option for interactive or external file input so that large circuits that are to be tested as a whole can be tested without tediously entering them interactively.

Research should be continued in the area of non-classical fault diagnosis; specifically in the area of diagnosing transistor faults in very large scale integration (VLSI) circuits. As was noted in Chapter one, faults in VLSI circuits are typically stuck-open and stuck-closed transistor faults. With the widespread use of VLSI technology the automated diagnosis of these faults becomes more and more necessary. Jain and Agrawal have formalized a technique for converting stuck-at faults to transistor faults and vice versa (15:65). This technique can be used with the stuck-at fault routines developed here to approach the task of testing VLSI circuits.

Specific Improvements. Several specific modifications to the Cerny extensions can be done. The first one relates to the second general modification described above. To make this second modification useful for the Cerny-based routines they need to be changed to have an option to preprocess and save input information. Preprocessing would construct the necessary equations and lists that are generated using a given circuit description (FCE, OCE, inputs, outputs etc.) and store this information, along with the desription, in secondary storage. Kainec's original system includes this option. In this way the circuit description that is interactively entered by the user can be saved for use after the system is exited.

The second recommended modification to the Cerny work is that it be changed to include an option for diagnosing all possible single stuck-at faults in a circuit. This change would greatly improve its usefulness to a user that has no idea where the fault may be, and wishes to test the entire circuit without specifying and running a test on each node in the circuit. Fault simulation is a process that allows a user to determine the fault coverage that a particular test vector has. Techniques exist (parallel, deductive and concurrent fault simulation) that can deduce which stuck-at faults are detectable given a certain test vector. The Cerny-based diagnostic system can be programmed to randomly choose a node, generate a test for that node, and use fault simulation to see what other faults are

detectable by applying the vector. Faults that have already been covered can be iteratively eliminated from the list of all possible faults until test vectors have been generated and applied to cover all faults, or until a fault has been detected (which ever comes first). This process would also address the limitation decsribed above that concerns the fault isolation capability of the system. In this way the specific members of a class of faults that may have resulted in a detected error can be identified to the user. This technique could be used for bridge and multiple stuck-at fault diagnosis also, but may require too much computation for testing for these faults.

Appendix A. Fundamentals of Boolean Algebra

Definitions

An algebra is characterized by three components:

1. A set, called a *carrier*,
2. *Operations* defined on the carrier, and
3. Distinct members of the carrier which are called *constants* of the algebra.(30:301)

In addition to these components, an algebra has associated *axioms*. A *closed algebraic system* is governed by the Law of Substitution which states that two expressions are said to be equal if one can be replaced by the other (13:55).

A *Boolean algebra* is a closed algebraic system denoted by the quintuple

$$< \mathbf{B}, +, \cdot, 0, 1 > \tag{A.1}$$

where

- $\mathbf{B}$ is the carrier of the algebra,
- $+$ and $\cdot$ are binary operations defined on $\mathbf{B}$, and
- 0 and 1 are the constants of $\mathbf{B}$.

The operator $\cdot$ is called **AND**. An expression of the form $a \cdot b$ is called a *conjunction*. The operator $+$ is called **OR**. An expression of the form $a + b$ is called a *disjunction*. The $*$ symbol is often used in lieu of the $\cdot$ symbol. Additionally, $a \cdot b$ may be replaced by the juxtaposition ab for simplicity.

Axioms

A Boolean algebra is based on a set of axioms known as Huntington's Postulates (14). These axioms are:

1. **Commutative Laws.** For all $a, b \in \mathbf{B}$,

$$a + b = b + a \tag{A.2}$$

$$a \cdot b = b \cdot a. \tag{A.3}$$

2. **Distributive Laws.** For all $a, b, c \in \mathbf{B}$,

$$a + (b \cdot c) = (a + b) \cdot (a + c) \tag{A.4}$$

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c). \tag{A.5}$$

3. **Identities.** For all $a \in \mathbf{B}$,

$$0 + a = a \tag{A.6}$$

$$1 \cdot a = a. \tag{A.7}$$

0 is the identity for the + operator. 1 is the identity for the · operator.

4. **Complements.** For every $a \in \mathbf{B}$, there exists an $a' \in \mathbf{B}$ such that

$$a + a' = 1 \tag{A.8}$$

$$a \cdot a' = 0. \tag{A.9}$$

The " ′ " symbol denotes *complementation*. Note that both equations must hold to prove complementation.

Boolean algebras are governed by the *principle of duality* by which a given valid expression has an associated valid dual expression. The dual of an expression is found by interchanging all + and · operators and interchanging identity elements 0 and 1. Note that

each of the preceding postulates has two expressions; these expressions are duals of each other.

The Inclusion Relation

A relation, $\leq$, is defined as follows. For all $a, b \in \mathbf{B}$

$$a \leq b \quad \Leftrightarrow \quad a \cdot b' = 0 \tag{A.10}$$

(28:8)

The relation $\leq$ is called the *inclusion relation.*

Theorems

Theorems which can be proven from the axioms and the definition of the inclusion relation are:

1. **Associativity.** For all $a, b, c \in \mathbf{B}$,

$$a + (b + c) = (a + b) + c \tag{A.11}$$

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c. \tag{A.12}$$

2. **Idempotence.** For all $a \in \mathbf{B}$,

$$a + a = a \tag{A.13}$$

$$a \cdot a = a. \tag{A.14}$$

3. **Boundedness.** For all $a \in \mathbf{B}$,

$$a + 1 = 1 \tag{A.15}$$

$$a \cdot 0 = 0. \tag{A.16}$$

4. **Absorption.** For all $a, b \in \mathbf{B}$,

$$a + (a \cdot b) = a \tag{A.17}$$

$$a \cdot (a + b) = a. \tag{A.18}$$

5. **Involution.** For all $a \in \mathbf{B}$,

$$(a')' = a. \tag{A.19}$$

6. **DeMorgan's Laws.** For all $a, b \in \mathbf{B}$,

$$(a + b)' = a' \cdot b' \tag{A.20}$$

$$(a \cdot b)' = a' + b'. \tag{A.21}$$

7. For all $a, b \in \mathbf{B}$,

$$a + a' \cdot b = a + b \tag{A.22}$$

$$a \cdot (a' + b) = a \cdot b. \tag{A.23}$$

8. **Consensus.** For all $a, b, c \in \mathbf{B}$,

$$a \cdot b + a' \cdot c + b \cdot c = a \cdot b + a' \cdot c \tag{A.24}$$

$$(a + b) \cdot (a' + c) \cdot (b + c) = (a + b) \cdot (a' + c). \tag{A.25}$$

9. **Interchange.** For all $a, b, c \in \mathbf{B}$,

$$(a \cdot b) + (a' \cdot c) = (a + c) \cdot (a' + b) \tag{A.26}$$

$$(a + b) \cdot (a' + c) = (a \cdot z) + (a' \cdot b). \tag{A.27}$$

10. For all $a, b \in \mathbf{B}$,

$$a \leq a + b \tag{A.28}$$

$$a \cdot b \leq a. \tag{A.29}$$

(17, 20, 13)

Properties

General properties of Boolean algebras which can be proven from the postulates and theorems are:

1.

$$a = b \Leftrightarrow a' \cdot b + a \cdot b' = 0 \tag{A.30}$$

$$a = b \Leftrightarrow a' \cdot b' + a \cdot b = 1. \tag{A.31}$$

$(a' \cdot b + a \cdot b')$ is the exclusive-OR of a and b and is denoted by either $(a \oplus b)$ or a XOR b; $(a' \cdot b' + a \cdot b)$ is the exclusive-NOR of a and b and is denoted be either $(a \odot b)$ or a XNOR b.

2.

$$a = 0 \ \& \ b = 0 \Leftrightarrow a + b = 0 \tag{A.32}$$

$$a = 1 \ \& \ b = 1 \Leftrightarrow a \cdot b = 1. \tag{A.33}$$

Literals, Terms, and Formulas

A *literal* is a variable or complemented variable such as a, b, a', b'. A *term* is a 1, a literal, or a conjunction of two or more literals in which no two literals involve the same variable. Examples of terms include ab', ac, and abc'. An *alterm* is a 0, a literal, or a disjunction of literals in which no two literals involve the same variable. Examples include $(a + b), (a + c')$, and $(a + b + c')$. (6:2.1-1)(20:225)

The set of Boolean ***formulas*** on n symbols $x_1, \ldots, x_n$ is defined by the following:

1. The elements of **B** are Boolean formulas, and
2. The symbols $x_1, \ldots, x_n$ are Boolean formulas, and
3. If f and g are Boolean formulas, then so are
   (a) $f + g$,
   (b) $f \cdot g$,
   (c) f', and

4. A string is a Boolean formula if and only if it is formed by a finite number of applications of the first three rules.

Examples of formulas include $x, x', x + y, (x \cdot (y + z))' + w$.

A *sum-of-products* formula is 0, a single term, or a disjunction of terms. A *product-of-sums* formula is 1, a single alterm, or a conjunction of alterms. (6:2.1-1)

Functions

An n-variable Boolean *function*, $f : \mathbf{B}^n \rightarrow \mathbf{B}$, is the mapping associated with an n-variable Boolean formula. Rudeanu, in his work on Boolean functions and equations, gives an informal definition of a Boolean function:

> Roughly speaking, a *Boolean function* (also called *Boolean polynomial* by certain authors) is a function with arguments and values in a Boolean algebra $\mathbf{B}$, such that f can be obtained from variables and constants of $\mathbf{B}$ by superpositions of the basic operations $+, \cdot$, and $'$ of $\mathbf{B}$. (28:16)

Rudeanu makes a clear distinction between Boolean functions in the general case, and the special case of Boolean functions involving no constants except 0 and 1 which he calls simple Boolean functions (28:xvi). He states:

> In the particular case of the two-element Boolean algebra $\mathbf{B}_2 = \{0, 1\}$, every function $f : \mathbf{B}_2^n \rightarrow \mathbf{B}_2$ is a simple Boolean function and will be termed a *truth function* (also called a "switching function" or "Boolean function" by switching theorists ...) (28:xvi)

The switching theorist point of view is taken in this work; however, all axioms, properties, and theorems discussed in this report hold for Boolean functions in the general case.

Boolean functions may be constructed as follows:

1. For n variables, $x_1, \ldots, x_n$, the *projection function* $f : \mathbf{B}_2^n \rightarrow \mathbf{B}_2$ defined by

$$f(x_1, \ldots, x_n) = x_i \qquad \forall (x_1, \ldots, x_n) \in \mathbf{B}_2^n, \qquad i \in \{1 \ldots n\}, \tag{A.34}$$

is an n-variable Boolean function.

2. If $g, h : \mathbf{B}_2^n \rightarrow \mathbf{B}_2$ are n-variable Boolean functions, then the functions $g + h$, gh, and g' defined by

   (a)
   $$(g + h)(x_1, \ldots, x_n) = g(x_1, \ldots, x_n) + h(x_1, \ldots, x_n) \tag{A.35}$$

   (b)
   $$gh(x_1, \ldots, x_n) = g(x_1, \ldots, x_n)\, h(x_1, \ldots, x_n) \tag{A.36}$$

   (c)
   $$g'(x_1, \ldots, x_n) = (g(x_1, \ldots, x_n))' \tag{A.37}$$

   $\forall (x_1, \ldots, x_n) \in \mathbf{B}_2^n$, are also n-variable Boolean functions.

3. A function is a Boolean function if and only if it is formed by a finite number of applications of the first two rules. (28:17)

Every n-variable Boolean formula maps into a corresponding n-variable Boolean function. A function, $f : \mathbf{B}^n \rightarrow \mathbf{B}$, is a Boolean function if and only if it can be represented by a Boolean formula. Moreover, a Boolean function may have any number of corresponding formulas. Formulas that represent the same function are called *equivalent formulas*. A *function table* or *truth table* is often used to specify a function.

Example A.1:

Given the two-element Boolean algebra, $\mathbf{B} = \{0, 1\}$, a truth table for the three-variable Boolean function $f : \mathbf{B}_2^3 \rightarrow \mathbf{B}_2$ corresponding to the Boolean formula $xyz + x'z' + y'z'$ is given by Table A.1:

$x\ y\ z$	$f(x, y, z)$
0 0 0	1
0 0 1	0
0 1 0	1
0 1 1	0
1 0 0	1
1 0 1	0
1 1 0	0
1 1 1	1

Table A.1. Truth Table for Example A.1

□

Boolean Expansion Theorem

The most important functional theorem is the *Boolean Expansion Theorem*. It is stated as follows:

If f is an n-variable Boolean function, then f has the expansions

$$f(x_1, x_2, \ldots, x_n) = x_1' f(0, x_2, \ldots, x_n) + x_1 f(1, x_2, \ldots, x_n) \tag{A.38}$$

$$f(x_1, x_2, \ldots, x_n) = [x_1' + f(1, x_2, \ldots, x_n)][x_1 + f(0, x_2, \ldots, x_n)]. \tag{A.39}$$

(4)

Extended Verification Theorem

Another important theorem in Boolean algebra is the *Extended Verification Theorem*. It is stated:

Let $f, g : \mathbf{B}^n \rightarrow \mathbf{B}$ be Boolean functions, and assume that the equation $f(X) = 0$ is consistent. Then the following statements are equivalent:

1. $f(X) = 0 \quad \Rightarrow \quad g(X) = 0,$
2. $g(X) \leq f(X) \quad \forall X \in \mathbf{B}^n,$
3. $g(X) \leq f(X) \quad \forall X \in \{0,1\}^n$

(28:100)

Canonical Forms

It is often desirable to use a restricted class of formula in which any Boolean function has only one corresponding formula. Formulas in such classes are called *canonical forms*. Canonical Boolean forms include the *minterm canonical form*, the *maxterm canonical form*, and the *Blake canonical form*.

Minterm Canonical Form. A *minterm* is a term in a formula of n variables which contains all variables of the formula either in complemented or uncomplemented form. A

formula in *minterm canonical form* is a sum-of-products formula in which all of the terms are minterms. A minterm canonical form is also called a *canonical sum-of-products form* or *full disjunctive normal form* (20:225)(13:84).

Example A.2:
Given the three-variable Boolean function $f : \mathbf{B}_2^3 \rightarrow \mathbf{B}_2$ from Example A.1, the following formula in minterm canonical form represents the same function f:

$$xyz + x'yz' + x'y'z' + xy'z'. \tag{A.10}$$

□

Often, a shorthand notation is used to represent a minterm. This form is m_i, where i is the decimal integer of the binary code for the minterm. The shorthand notation for three-variable minterms is given in Table A.2.

Term	Binary Code	Shorthand Notation
$x'y'z'$	0 0 0	m_0
$x'y'z$	0 0 1	m_1
$x'yz'$	0 1 0	m_2
$x'yz$	0 1 1	m_3
$xy'z'$	1 0 0	m_4
$xy'z$	1 0 1	m_5
xyz'	1 1 0	m_6
xyz	1 1 1	m_7

Table A.2. Shorthand Notation for Minterms

Using this notation, the formula in Example A.2 can be written as $f(x, y, z) = m_0 + m_2 + m_4 + m_7$. This notation can be shortened further to *minterm list form*. The function $f(x, y, z)$ is expressed in minterm list form as $f(x, y, z) = \sum m(0, 2, 4, 7)$. (13:85)

Maxterm Canonical Form. A *maxterm* is an alterm in a formula of n variables which contains all variables of the formula either in complemented or uncomplemented form. A formula in *maxterm canonical form* is a product-of-sums formula in which all of the alterms are maxterms. A maxterm canonical form is also called a *canonical product-of-*

sums form or *full conjunctive normal form* (20:225)(13:84). Hence, the maxterm canonical form is analogous to the minterm canonical form where the formula is expressed in product-of-sums form rather than sum-of-products form and terms are replaced by alterms.

Example A.3:

Given the three-variable Boolean function $f : \mathbf{B}_2^3 \rightarrow \mathbf{B}_2$ from Example A.1, the following formula in product-of-sums form represents the same function f:

$$(x + z')(x' + y + z')(x' + y' + z) \tag{A.41}$$

This formula can be transformed to the following formula in maxterm canonical form:

$$(x + y + z')(x + y' + z')(x' + y + z')(x' + y' + z) \tag{A.42}$$

□

As with minterms, a shorthand notation is used to represent maxterms. This form is M_i, where i is the decimal integer of the binary code for the maxterm. The shorthand notation for three-variable maxterms is given in Table A.3. Using this notation, the formula

Alterm	Binary Code	Shorthand Notation
$x + y + z$	0 0 0	M_0
$x + y + z'$	0 0 1	M_1
$x + y' + z$	0 1 0	M_2
$x + y' + z'$	0 1 1	M_3
$x' + y + z$	1 0 0	M_4
$x' + y + z'$	1 0 1	M_5
$x' + y' + z$	1 1 0	M_6
$x' + y' + z'$	1 1 1	M_7

Table A.3. Shorthand Notation for Maxterms

in Example A.3 can be written as $f(x, y, z) = M_1 M_3 M_5 M_6$. This notation can be shortened further to *maxterm list form*. The function $f(x, y, z)$ is expressed in maxterm list form as $f(x, y, z) = \prod M(1, 3, 5, 6)$. (13:88)

. ıke Canonical Form. A term p is called an *implicant* of a Boolean function f if $p \leq f$. When a function f is expressed in sum-of-products form, all terms in the form are implicants of f. A *prime implicant* of a Boolean function f is an implicant of f such that it is no longer an implicant if any of its literals is removed (24). Boolean axioms and theorems such as consensus and absorption are used to reduce a Boolean formula for a function to a form which consists of the prime implicants of the function. An application is minimization, one approach to which is to reduce a Boolean formula to an equivalent formula which includes the smallest number of prime implicants that still represent the same function. The impetus for minimization is to represent a Boolean function by a formula that can be implemented in hardware with the smallest number of components. See (13, 24, 25) for discussions of Boolean minimization. A *prime implicate* is the analog of a prime implicant for the product-of-sums form.

Example A.4.

The only term in the n-variable Boolean formula f given by

$$xyz + x'yz' + x'y'z' + xy'z' \tag{A.43}$$

that is a prime implicant of f is xyz. The formula may be transformed to an equivalent formula consisting of only prime implicants by application of Boolean axioms and theorems. An equivalent formula which consists only of prime implicants is:

$$xyz + y'z' + x'z'. \tag{A.44}$$

□

In the process of reducing a given formula to prime implicants, *superfluous* terms are often generated. A term p is superfluous in a sum-of-products formula, $p + q$, if $p + q$ is equivalent to the formula q (24:522). A literal of a term in a sum-of-products formula is *superfluous* if it can be removed without changing the formula to a non-equivalent formula. Quine called a "formula *irredundant* if it has no superfluous clauses and none of its clauses has superfluous literals (24:523)."

Another application for the prime implicants of a formula is for *Boolean inference*, also called *Boolean reasoning*. Boolean inference is "the extraction of conclusions from a collection of Boolean data" (6:2.0-2). The basis for Boolean inference is the *Blake canonical form*. The Blake canonical form, denoted BCF(f), of a function f is the disjunction of all of the prime implicants of f. The Blake canonical form is a complete and simplified representation of all possible conclusions that can be inferred from a Boolean equation. Methods for generating BCF(f) are by the *exhaustion of implicants*, *iterated consensus*, and *multiplication*. Blake invented the methods of iterated consensus and multiplication (3). Iterated consensus is discussed in (24); the multiplication method is found in (29).

Example A.5.

The n-variable Boolean function defined in Table A.1 and represented by the formula

$$xyz + y'z' + x'z' \tag{A.45}$$

is in Blake canonical form because the formula consists of all of the prime implicants of the function. □

Reduction

Any system of Boolean equations can be reduced to a single Boolean equation of the form $f(\underline{x}) = g(\underline{x})$ where $g(\underline{x})$ is any preassigned Boolean function (28:116-117). In particular, we may choose $g(\underline{x})$ to be 0 or 1. (The notation $\underline{x}$ denotes the vector $(x_1, x_2, \ldots, x_n)$.) The form $f(\underline{x}) = 0$ is derived in the following manner. A system

$$\begin{aligned} g_1(\underline{x}) &= h_1(\underline{x}) \\ g_2(\underline{x}) &= h_2(\underline{x}) \\ &\vdots \\ g_n(\underline{x}) &= h_n(\underline{x}) \end{aligned} \tag{A.46}$$

of Boolean equations can be transformed, using property (A.30), into the equivalent system

$$\begin{aligned} g_1(\underline{x}) \oplus h_1(\underline{x}) &= 0 \\ g_2(\underline{x}) \oplus h_2(\underline{x}) &= 0 \\ &\vdots \\ g_n(\underline{x}) \oplus h_n(\underline{x}) &= 0. \end{aligned} \tag{A.47}$$

This system of equations can then be transformed into a single Boolean equation by property (A.32). Since all of the equations must be simultaneously true, they are "&'ed" together as in equation (A.32). However, the "&" symbol is dropped for notational simplicity. The resulting single Boolean equation is

$$f(\underline{x}) = 0 \tag{A.48}$$

where f is defined by

$$f = \sum_{i=1}^{n} (g_i \oplus h_i), \tag{A.49}$$

i.e.,

$$f = \sum_{i=1}^{n} (g_i' h_i + g_i h_i'). \tag{A.50}$$

The $p(\underline{x}) = 1$ form of a system of equations is similarly derived. The system of equations (A.46) can be transformed into an equivalent system using the property shown by equation (A.31):

$$\begin{aligned} g_1(\underline{x}) \vee h_1(\underline{x}) &= 1 \\ g_2(\underline{x}) \vee h_2(\underline{x}) &= 1 \\ &\vdots \\ g_n(\underline{x}) \vee h_n(\underline{x}) &= 1. \end{aligned} \tag{A.51}$$

This system of equations is transformed into a single Boolean equation by equation (A.33). Again, the "&" symbol is dropped for notational simplicity. The resulting single Boolean equation is

$$p(\underline{x}) = 1 \tag{A.52}$$

where p is defined by

$$p = \prod_{i=1}^{n} (g_i \vee h_i), \tag{A.53}$$

i.e.,

$$p = \prod_{i=1}^{n} (g_i' h_i' + g_i h_i). \tag{A.54}$$

The utility of the choice of the $f(\underline{x}) = 0$ form versus the $p(\underline{x}) = 1$ form is dependent on the application (28:52). Conversion between the two forms is done by complementation of both sides of the equality, i.e.,

$$f'(\underline{x}) = 0 \Leftrightarrow f(\underline{x}) = 1 \tag{A.55}$$

and

$$p'(\underline{x}) = 1 \Leftrightarrow p(\underline{x}) = 0. \tag{A.56}$$

Eliminants

The Conjunctive Eliminant. For an n-variable Boolean function $f : B^n \rightarrow B$ with variables $x_1, \ldots, x_n$ and a subset $\{x_1, x_2\}$ of the variables, the *conjunctive eliminant* of the function with respect to $\{x_1, x_2\}$ is defined as:

$$ECON(f, \{x_1, x_2\}) = \prod_{(x_1, x_2) \in \{0,1\}^2} f(x_1, x_2, x_3, \ldots, x_n). \quad \text{(A.77)}$$

(6:3.8-1)

Although a specific subset of the variables was used in the above definition, the conjunctive eliminant of a function may be found with respect to an arbitrary subset of the variables in the function.

Example A.6:

The conjunctive eliminant of a function $f(x, y, z)$ with respect to z is given by

$$ECON(f(x, y, z), \{z\}) = f(x, y, 0) f(x, y, 1). \quad \text{(A.78)}$$

□

Brown has shown that the conjunctive eliminant of a function in Blake canonical form with respect to a given variable is the sum of terms in the form which do not involve the variable (6:3.8-2). Formally,

$$ECON(f, \{y\}) = \sum (\textit{terms of } BCF(f) \textit{ which do not have a literal } y \textit{ or } y'). \quad \text{(A.79)}$$

The resulting formula is in Blake canonical form.

The Disjunctive Eliminant. For an n-variable Boolean function $f : B^n \rightarrow B$ with variables $x_1, \ldots, x_n$ and a subset $\{x_1, x_2\}$ of the variables, the *disjunctive eliminant* of the function with respect to $\{x_1, x_2\}$ is defined as:

$$EDIS(f, \{x_1, x_2\}) = \sum_{(x_1, x_2) \in \{0,1\}^2} f(x_1, x_2, x_3, \ldots, x_n). \tag{A.60}$$

(6:3.8-1)

As in the conjunctive eliminant, the disjunctive eliminant of a function may be found with respect to an arbitrary subset of the variables in the function.

Example A.7:

The disjunctive eliminant of the function $f(x, y, z)$ with respect to z is

$$EDIS(f(x, y, z), \{z\}) = f(x, y, 0) + f(x, y, 1). \tag{A.61}$$

□

A simple method for deriving the disjunctive eliminant of a Boolean function f is by transforming the formula that represents the function to any equivalent sum-of-products form and then replacing the literals of the variables to be eliminated, whether in complemented or uncomplemented form, by 1 (22).

Elimination

Given a Boolean equation, it is possible to determine constraints on certain variables given the absence of information with respect to the other variables using a process called ***elimination.*** Equations deduced as the result of elimination are called *resultants of elimination.*

Using the definition of the conjunctive eliminant, a variable may be eliminated from an equation to form a new equation:

$$f(\underline{x}) = 0 \Rightarrow ECON(f, \{x_i\}) = 0 \tag{A.62}$$

The equation $ECON(f, \{x_i\}) = 0$ is called the resultant of elimination of x_i from equation $f(\underline{x}) = 0$.

Using the definition of the disjunctive eliminant, a variable may be eliminated from an equation to form a new equation:

$$p(\underline{x}) = 1 \Rightarrow EDIS(p, \{x_i\}) = 1 \tag{A.63}$$

The equation $EDIS(f, \{x_i\}) = 1$ is called the resultant of elimination of x_i from equation $p(\underline{x}) = 1$.

Solutions of Boolean Equations

A solution of the equation $f(\underline{x}) = 0$ is a vector $\underline{a} \in \mathbf{B}^n$ such that $f(\underline{a}) = 0$ is an identity. In general, it is inconvenient to determine solutions of the $f(x, y, z) = 0$ form of an equation. A simple method to find a solution to an equation is first to convert the equation to the equivalent $p(x, y, z) = 1$ form as in equation (A.55), and then express p in minterm canonical form. Solutions are found by inspection of the minterms of $p(x, y, z)$.

Example A.8:

Given the equation

$$xyz' + x'z + y'z = 0, \tag{A.64}$$

the $f(\underline{x}) = 1$ form of this equation is

$$xyz + x'z' + y'z' = 1. \tag{A.65}$$

The minterm canonical form of the left-hand side of this equation is used to form a new equation

$$xyz + x'yz' + x'y'z' + xy'z' = 1. \tag{A.66}$$

By inspection, solutions of the equation are

$$(0,0,0), (0,1,0), (1,0,0), (1,1,1). \tag{A.67}$$

□

An equation typically will have several solutions. Constant vectors, $\underline{a}$ and $\underline{b}$, are called *equivalent* with respect to f if $f(\underline{a}) = f(\underline{b})$. Two equations are called "*equivalent* if they have the same set of solutions" (28:50).

Comparison of Functions

Given two n-variable Boolean functions f and g, a function h can be constructed which shows all circumstances in which functions f and g are different. h is defined in the following way:

$$f \oplus g = h \tag{A.68}$$

Minterms of h define the differences between f and g.

Example A.9:

Given the equations $f(x, y) = x$ and $g(x, y) = y$, $h(x, y)$ is found as follows:

$$\begin{aligned} h(x, y) &= f(x, y) \oplus g(x, y) \\ &= x \oplus y. \end{aligned} \tag{A.69}$$

Minterms of $h(x, y)$ are xy' and $x'y$. The results are summarized in Table A.4. □

$x\ y$	$f(x, y)$	$g(x, y)$	$h(x, y)$
0 0	0	0	0
0 1	1	0	1
1 0	0	1	1
1 1	1	1	0

Table A.4. Results of Example A.9

Appendix B. Cerny-based Diagnostic System Code

This appendix contains the commented code for the diagnostic system described in Chapter 5. Three of the files required are not part of this appendix as they were borrowed in their entirety from Kainec. They are cited below.

The following files comprise the diagnostic system:

- boolean.s (18:335-347)
- prefixer.s (18:234-246)
- tokenize.s (18:225-233)
- menu.s
- test1.s
- test1s.s
- test2.s
- test2s.s
- test3.s
- test3s.s
- utils.s
- utils2.s

The system is entered by typing **menu** and is driven by prompts to the user.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FILENAME:  menu.s
;;
;;  FILES REQUIRED FOR CALLED FUNCTIONS:  test1.s, test2.s,
;;                    test3.s, test1s.s, test2s.s, test3s.s,
;;                    utils.s, prefixer.s, tokenize.s
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  menu
;;
;;  CALLING FUNCTION(S):  single-fault-com, single-fault-seq,
;;                        bridge-fault-com, bridge-fault-seq,
;;                        multiple-fault-com, multiple-fault-seq
;;
;;  CALLED FUNCTIONS(S):  get-tokenized-list, token->prefix
;;                        get-internal-nodes, get-output-nodes,
;;                        get-input-nodes, remove-duplicates,
;;                        get-output-equations,
;;                        remove-output-equations
;;                        single-fault-com, single-fault-seq,
;;                        bridge-fault-com, bridge-fault-seq,
;;                        multiple-fault-com, multiple-fault-seq
;;
;;  PURPOSE:  This is the main routine for the diagnostic system.
;;            It first prompts the user for a choice of diagnostic
;;            routine.  It then uses GET-TOKENIZED-LIST to prompt
;;            the user for the circuit input file (containing the
;;            output-equations and individual circuit
;;            characteristic equations-- CCEs) which is read in and
;;            stored as a list of tokens.  TOKEN->PREFIX converts
;;            the tokenized-list into the intermediate-format data
;;            structure which maintains the structure and identity
;;            of the circuit.  Lists of the nodes in the circuit
;;            are then collected, based on location in the circuit.
;;            The output equations in the circuit are then
;;            collected and removed from the intermediate-format so
;;            that only the individual CCEs remain.  A branch is
;;            then taken based on the diagnostic routine chosen.
;;            The collected information is passed to the called
;;            routine.
;;
```

```
;;  VARIABLES:  none
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (menu)
   (newline)
   (writeln "    ENTER NUMERICAL CHOICE OF DIAGNOSTIC ROUTINE:")
   (newline)
   (writeln "          1.  SINGLE SA FAULT ANALYSIS --
                                    COMBINATIONAL CKT")
   (writeln "          2.  SINGLE SA FAULT ANALYSIS --
                                    SEQUENTIAL CKT")
   (writeln "          3.  BRIDGE FAULT ANALYSIS --
                                    COMBINATIONAL CKT")
   (writeln "          4.  BRIDGE FAULT ANALYSIS --
                                    SEQUENTIAL CKT")
   (writeln "          5.  MULTIPLE SA FAULT ANALYSIS --
                                    COMBINATIONAL CKT")
   (write n "          6.  MULTIPLE SA FAULT ANALYSIS --
                                    SEQUENTIAL CKT")
   (writeln "          7.  EXIT")
   (newline)
   (let* ( (fault-choice (read-line)))
          (if (equal? fault-choice "7")
              (writeln "EXITING DIAGNOSTIC SYSTEM.")
         (let* (
           (intermediate-format1 (token->prefix
                                    (get-tokenized-list)))
           (internal-nodes (get-internal-nodes
                                    intermediate-format1))
           (output-nodes (get-output-nodes intermediate-format1))
           (input-nodes (get-input-nodes intermediate-format1))
           (output-equations (remove-duplicates
                      (get-output-equations intermediate-format1
                                    internal-nodes)))
           (intermediate-format2 (remove-output-equations
                         intermediate-format1 output-equations)))
       (cond ((equal? fault-choice "1")
              (single-fault-com intermediate-format2 internal-nodes
                        output-nodes input-nodes output-equations))
             ((equal? fault-choice "2")
              (single-fault-seq intermediate-format2 internal-nodes
                        output-nodes input-nodes output-equations))

              ((equal? fault-choice "3")
```

```
  (bridge-fault-com intermediate-format2 internal-nodes
           output-nodes input-nodes output-equations))
  ((equal? fault-choice "4")
  (bridge-fault-seq intermediate-format2 internal-nodes
           output-nodes input-nodes output-equations))
  ((equal? fault-choice "5")
(multiple-fault-com intermediate-format2 internal-nodes
           output-nodes input-nodes output-equations))
 ((equal? fault-choice "6")
(multiple-fault-seq intermediate-format2 internal-nodes
           output-nodes input-nodes output-equations))
 (else
   (begin
   (newline)
   (writeln "   INCORRECT DIAGNOSTIC ROUTINE ENTRY!")
   (newline)
   (menu)))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FILENAME:  test1.s
;;
;;  FILES REQUIRED FOR CALLED FUNCTIONS:  boolean.s, utils2.s
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  single-fault-com
;;
;;  CALLING FUNCTION(S):  menu, single-fault-com
;;
;;  CALLED FUNCTION(S):  get-fault-variable, cut-node,
;;                       get-vector-function, combinational-test,
;;                       output-results, again?, single-fault-com,
;;                       menu
;;
;;  PURPOSE:  This function performs diagnosis of single stuck-at
;;            faults in combinational circuits when called upon to
;;            do so by MENU.  The user is prompted for a single
;;            suspected faulty variable using GET-FAULT-VARIABLE.
;;            The fault-variable is used to isolate the faulty line
;;            in the circuit.  CUT-NODE does this by replacing the
;;            variable with a TEST variable in the intermediate
;;            format description.  The intermediate format
;;            description contains the individual CCEs described in
;;            the text of the thesis.  CUT-NODE also deletes any
;;            logic that may be feeding the suspected faulty node.
;;            GET-VECTOR-FUNCTION then derives the Output
;;            Characteristic Function (OCF) that is used to produce
;;            test vectors.  COMBINATIONAL-TEST generates the
;;            stuck-at (first zero, and then one if the zero test
;;            passes) test vectors and then walks the user through
;;            an input/output test procedure.  COMBINATIONAL-TEST
;;            also performs generation and analysis of the i/o test
;;            results to determine the presence or absence of a
;;            fault.  Diagnostic test results are provided to the
;;            user via the OUTPUT-RESULTS function.  AGAIN? provides
;;            the option of testing a different line in the same circuit.
;;
;;  VARIABLES:  intermediate-format --
;;                  data structure containing the system of
;;                  equations describing the circuit at the gate
```

```
;;                  level
;;
;;              internal-nodes --
;;                  list containing internal circuit variables
;;
;;              output-nodes --
;;                  list containing output circuit variables
;;
;;              input-nodes --
;;                  list containing input circuit variables
;;
;;              output-equations --
;;                  overall functional equations relating primary
;;                  inputs to each circuit output
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (single-fault-com intermediate-format internal-nodes
                          output-nodes input-nodes
                          output-equations)
    (newline)
    (let* ( (fault-variable (get-fault-variable))
            (new-format (cut-node intermediate-format input-nodes
                      output-equations fault-variable))
            (vector-function (get-vector-function new-format
                                internal-nodes output-nodes
                                output-equations)))
        (if (null? vector-function)
            (writeln "NO TEST POSSIBLE FOR THIS VARIABLE.")
          (let* (
            (results (combinational-test vector-function input-nodes
                                output-nodes output-equations
                                intermediate-format fault-variable)))
            (output-results results fault-variable)))

        (if (again?) (single-fault-com intermediate-format
                                internal-nodes output-nodes
                                input-nodes output-equations)
              (begin
              (read-line)
              (menu)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
```

```
;;  FUNCTION:  get-fault-variable
;;
;;  CALLING FUNCTION(S):  single-fault-com, single-fault-seq
;;
;;  CALLED FUNCTION(S):  none
;;
;;  PURPOSE:  This function prompts the user to input the suspected
;;            faulty node from the circuit, and reads the entry.
;;
;;  VARIABLES: none
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-fault-variable)
     (newline)
     (writeln "ENTER THE VARIABLE THAT LABELS THE SUSPECTED FAULTY
                LINE:")
     (newline)
     (read))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  cut-node
;;
;;  CALLING FUNCTION(S):  single-fault-com, single-fault-seq
;;                        bridge-fault-com, bridge-fault-seq
;;
;;  CALLED FUNCTION(S):  check-list, replace-variable, delete-ccf
;;
;;  PURPOSE:  This function modifies the original set of individual
;;            CCEs (ccf-list) based on the suspected faulty node.
;;            It first checks to see if the node is an input node
;;            using CHECK-LIST.  If it is an input node then
;;            the function replaces the node in the CCEs with a
;;            TEST variable.  If the node is not an input then it
;;            must either be an internal or output node.  In either
;;            case the function deletes the logic (ccf) feeding the
;;            node using DELETE-CCF, in addition to replacing the
;;            node.  A special case exists when there are no
;;            internal internal nodes in the circuit.  The function
;;            MENU is set up to delete all output equations
;;            from the original input file and will leave no CCEs
;;            when no internal nodes exist.  This is accounted for
;;            here by performing the CUT-NODE function on the
```

```
;;            output equations of the circuit.  In the case where
;;            no internals exist then the output equations also
;;            describe the circuit at the gate level.
;;
;;  VARIABLES:  ccf-list -- the intermediate-format, a list of
;;                          the individual CCEs describing the
;;
;;              input-nodes -- list of circuit inputs
;;
;;              output-equations -- list of circuit output
;;                                  equations
;;
;;              fault-variable -- suspected fault node provided by
;;                                user
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (cut-node ccf-list input-nodes output-equations
                  fault-variable)
     (let* ( (replacement-var 'TEST))
      (if (null? ccf-list)
          (if (check-list input-nodes fault-variable)
             (replace-variable output-equations replacement-var
                               fault-variable)
             (replace-variable (delete-ccf output-equations
                                   fault-variable) replacement-var
                               fault-variable))
          (if (check-list input-nodes fault-variable)
             (replace-variable ccf-list replacement-var
                               fault-variable)
             (replace-variable (delete-ccf ccf-list fault-variable)
                           replacement-var fault-variable)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  check-list
;;
;;  CALLING FUNCTION(S):  cut-node, check-list, cut-node2
;;
;;  CALLED FUNCTION(S):  check-list
;;
;;  PURPOSE:  This function searches a list containing no internal
;;            lists for a particular item.  In all cases here the
;;            lists are lists of variables; the searched for item
```

```
;;            is a suspected fault variable.
;;
;;  VARIABLES:  variable-list -- list to search
;;
;;              fault-variable -- item to search for
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (check-list variable-list fault-variable)
     (cond ((null? variable-list) nil)
           ((equal? (car variable-list) fault-variable) t)
           (else
             (check-list (cdr variable-list) fault-variable))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  replace-variable
;;
;;  CALLING FUNCTION(S):  cut-node, replace-variable, cut-node2
;;
;;  CALLED FUNCTION(S):  replace-variable
;;
;;
;;  PURPOSE:  This function substitutes the specified fault node
;;            with a replacement variable in the list of CCEs
;;            regardless of the depth of a given CCE.
;;
;;  VARIABLES:  ccf-list -- individual circuit characteristic
;;                          equations
;;
;;              replacement -- the variable to substitute in
;;
;;              fault-variable -- the variable to replace
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (replace-variable ccf-list replacement fault-variable)
     (cond
           ((null? ccf-list) nil)
           ((atom? (car ccf-list))
            (cond ((equal? (car ccf-list) fault-variable)
                   (cons replacement (cdr ccf-list)))
                  (else
                   (cons (car ccf-list)
```

```
                        (replace-variable (cdr ccf-list)
                             replacement fault-variable)))))
          (else
           (append (list (replace-variable (car ccf-list)
                             replacement fault-variable))
                   (replace-variable (cdr ccf-list)
                             replacement fault-variable)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  delete-ccf
;;
;;  CALLING FUNCTION(S):  cut-node, cut-node2
;;
;;  CALLED FUNCTION(S):  find-ccf, remove
;;
;;  PURPOSE:  This function finds a particular CCE in a set of
;;            individual CCEs and removes it from the set.  The
;;            search for the CCE to be deleted keys on the
;;            suspected fault node.
;;
;;  VARIABLES:  ccf-list -- individual CCEs
;;
;;              fault-variable -- suspected fault node
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (delete-ccf ccf-list fault-variable)
    (let* ((ccf-to-go (find-ccf ccf-list fault-variable)))
     (if (null? ccf-to-go) nil
        (let* (
          (new-ccf-list (remove ccf-to-go ccf-list)))
       new-ccf-list))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  find-ccf
;;
;;  CALLING FUNCTION(S):  delete-ccf, find-ccf
;;
;;  CALLED FUNCTION(S):  find-ccf
;;
;;  PURPOSE:  This function searches a list of CCEs for a CCE that
```

```
;;            contains the fault variable as the output of a gate.
;;            It is also the filter to determine if the user has
;;            specified a node that does not exist in the circuit
;;            description.  If all CCEs are searched without
;;            finding the specified node then this is the case.
;;
;;  VARIABLES:  ccf-list -- list of individual CCEs
;;
;;              fault-variable -- suspected fault node
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (find-ccf ccf-list fault-variable)
     (cond ((null? ccf-list)
            (begin
            (writeln "ERROR, ONE OR MORE SPECIFIED VARIABLES DO NOT
         EXIST IN THIS CIRCUIT.") (writeln "PROCESSING.....") nil))
           ((equal? (cadr (car ccf-list)) fault-variable)
            (car ccf-list))
           (else
            (find-ccf (cdr ccf-list) fault-variable))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-vector-function
;;
;;  CALLING FUNCTION(S):  single-fault-com, single-fault-seq
;;                        bridge-fault-com, bridge-fault-seq
;;                        multiple-fault-com, multiple-fault-seq
;;                        get-vector-function
;;
;;  CALLED FUNCTION(S):  get-vector-function, get-vf
;;
;;  PURPOSE:  This function generates the functions that are later
;;            used to generate test vectors.  We talk in terms of
;;            vectors in this case knowing that all functions are
;;            equal to one.  The function iteratively calls GET-VF
;;            to combine the individual CCEs with each output
;;            equation.  Another way to approach this is to combine
;;            all of the output equations and then combine them
;;            with the overall CCE.  The resulting function is the
;;            same.  The process used here results in a function
;;            of functions that is equivalent to using the other
;;            approach.
```

```
;;
;;  VARIABLES:  ccf-list -- modified list of individual CCEs
;;
;;              internal-nodes -- list of circuit internal nodes
;;
;;              output-nodes -- list of circuit output nodes
;;
;;              output-equations -- list of circuit output
;;                                  equations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-vector-function ccf-list internal-nodes output-nodes
                             output-equations)
     (if (null? output-equations) nil
     (let* ( (vf (get-vf ccf-list internal-nodes output-nodes
                         (list (car output-equations))))
             (vf2 (append vf (get-vector-function ccf-list
                                        internal-nodes output-nodes
                                    (cdr output-equations)))))
          vf2)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-vf
;;
;;  CALLING FUNCTION(S):  get-vector-function
;;
;;  CALLED FUNCTION(S):  make-sop, eliminate, simplify
;;
;;  PURPOSE:  This function calls MAKE-SOP to change each prefix-
;;            form equation into a function of variables set equal
;;            to one (the one being implicit, not explicit).  These
;;            equations include the modified individual CCEs and
;;            output equations.  In the process MAKE-SOP also
;;            combines the equations in a given description.  In
;;            this case the individual CCEs and output equatins are
;;            combined into one equation.  ELIMINATE is called to
;;            eliminate the internal variables and output variables
;;            from the final combination.  The order of operations
;;            does not correspond to the order presented in the
;;            mathematical development, but is faster and does the
;;            same job.  COMPLEMENT completes the Cerny routine;
;;            SIMPLIFY reduces the result.
```

```
;;
;;
;;  VARIABLES:  ccf-list -- modified CCEs
;;
;;              internal-nodes -- internal nodes of the circuit
;;
;;              output-nodes -- output nodes of the circuit
;;
;;              output-equation -- a circuit output equation
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;get one vector function
(define (get-vf ccf-list internal-nodes output-nodes
                                                    output-equation)
    (let* ( (new-ccf (append ccf-list output-equation))
            (sop1 (make-sop new-ccf))
            (sop2 (eliminate sop1 internal-nodes))
            (sop3 (eliminate sop2 output-nodes))
            (sop4 (simplify (complement sop3))))
        sop4))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  make-sop
;;
;;  CALLING FUNCTION(S):  get-vf, make-sop, mksops, init-eqs1
;;
;;  CALLED FUNCTION(S):  make-sop, add, mult, xor, xnor, complement
;;
;;  PURPOSE:  Orginally coded by Kainec (18:273), this function was
;;            developed to collapse a system of equations into a
;;            form that sets a function of all the variables equal
;;            to zero.  It is changed to realize a function-set-
;;            equal-to-one form.
;;
;;  VARIABLES:  lst -- a list of the system of equations to be
;;                     collapsed
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (make-sop lst)
   (cond ( (null? lst) '(())
```

```
; if lst is atomic return it in SOP list format
( (atom? lst)
  (list (list lst)) )

; if the first element is atomic, then lst is in prefix
;form
( (atom? (car lst))
  (let ((first-elt (car lst))
        (second-elt (cadr lst)))

     (if (eq? 'NOT first-elt)

         ; if first-elt is NOT, then complement the SOP
         ; form of the second element
         (if (atom? second-elt)
             (list (list (list second-elt)))
             (complement (make-sop second-elt)))

         ; if the first-elt is a valid Boolean operator,
         ; perform the operation on the SOP forms of the
         ; second and third elements
         (cond ( (or (eq? '+ first-elt) (eq? 'OR
                                            first-elt))
                   (add (make-sop second-elt) (make-sop
                                              (caddr lst))) )

               ( (or (eq? '* first-elt) (eq? 'AND
                                            first-elt))
                   (mult (make-sop second-elt) (make-sop
                                              (caddr lst))) )

               ( (or (eq? '! first-elt) (eq? 'XOR
                                            first-elt))
                   (xor (make-sop second-elt) (make-sop
                                              (caddr lst))) )

               ( else
                 '() )))) )

; the input lists is a list of lists - assume that these
; lists are
; a system of equations in prefix form, break up
; accordingly, and make into SOP forms
(else
   (let* ((first-list   (car lst))
```

```
        (rest-of-list (cdr lst))
        (first-elt    (car first-list))
        (second-elt   (cadr first-list))
        (third-elt    (caddr first-list)))

    (cond ( (eq? 'EQ first-elt)
            (mult (xnor (make-sop second-elt)
                        (make-sop third-elt))
                  (make-sop rest-of-list)) )

          ; if first-elt of first-list is LE, then take
          ; MULT the SOP form of second-elt by the
          ; COMPLEMENT of the SOP form of third-elt.
          ; ADD the result to the SOP form of the
          ; rest-of-list
          ( (eq? 'LE first-elt)
            (mult (add (complement (make-sop
                                    second-elt))
                       (make-sop third-elt))
                  (make-sop rest-of-list)) )


          ; if first-elt of first-list is GE, then take
          ; MULT the COMPLEMENT of the SOP form of
          ; second-elt by the SOP form of third-elt.
          ; ADD the result to the SOP form of the
          ; rest-of-list
          ( (eq? 'GE first-elt)
            (mult (add (make-sop second-elt)
                       (complement (make-sop third-elt)))
                  (make-sop rest-of-list)) )

          ; otherwise, assume that each sublist of lst
          ; is a formula and add its SOP form to the SOP
          ; form of the rest-of-list
          (else
             (mult (make-sop first-list)
                   (make-sop rest-of-list)) ))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  eliminate
;;
;;  CALLING FUNCTION(S):  get-vf, eliminate
```

```
;;
;;  CALLED FUNCTION(S):  edis, eliminate
;;
;;  PURPOSE:  Originally coded by Kainec (18:283), this function
;;            was developed to iteratively eliminate a list of
;;            nodes from a function using the conjunctive
;;            eliminant.  It has been changed to reflect the
;;            function-equal-to-one form by using the disjunctive
;;            eliminant for elimination.
;;
;;  VARIABLES:  f -- the function to eliminate the variables from
;;
;;              nodes -- the nodes to eliminate
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (eliminate f nodes)

   (if (null? nodes)
       f
       (let ( (new-f (edis f (list (car nodes)))) )
          (eliminate new-f (cdr nodes)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  combinational-test
;;
;;  CALLING FUNCTION(S):  single-fault-com
;;
;;  CALLED FUNCTION(S):  get-sa-zero, supplement, mksops,
;;                       get-result1, bad-result?, get-sa-one,
;;                       get-result2
;;
;;  PURPOSE:  This function takes the function generated in
;;            GET-VECTOR-FUNCTION and generates test
;;            vectors to be applied to the circuit.  A vector
;;            capable of detecting the specified fault node
;;            stuck-at-zero is generated using GET-SA-ZERO and
;;            SUPPLEMENT.  The user is directed to apply the vector
;;            and is prompted for the resulting outputs in
;;            GET-RESULT1.  GET-RESULT1 also combines the vector
;;            with the resulting outputs to support comparison to
;;            the output equations using BAD-RESULT?.  MKSOPS
;;            changes the output equations from prefix-form into
;;            function-equal-to-one form for the comparison.  If
```

```
;;            BAD-RESULT? determines that a fault exists then the
;;            user is informed that the node is stuck-at-zero.  If
;;            not then a test vector for the stuck-at-one test is
;;            generated, the results are processed and the user is
;;            given a message based on whether the node was
;;            stuck-at-one or not.
;;
;;
;;  VARIABLES:  function -- a function of the primary inputs and
;;                          test variable used to generate test
;;                          vectors
;;
;;              in-nodes -- circuit input nodes
;;
;;              out-nodes -- circuit output nodes
;;
;;              out-eqs -- circuit output equations
;;
;;              int-form -- the original individual CCEs
;;
;;              fault-var -- the suspected fault node
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (combinational-test function in-nodes out-nodes out-eqs int-form
                            fault-var)
     (let* ( (z-vector (supplement (car (get-sa-zero function)) in-nodes))
             (out-eqs2 (mksops out-eqs))
             (result1 (get-result1 z-vector out-nodes t)))
        (if (bad-result? result1 fault-var int-form out-eqs2)
            '(SA0)
            (let* ( (o-vector (supplement (car (get-sa-one function))
                                          in-nodes))
                    (result2 (get-result2 o-vector out-nodes t)))
               (if (bad-result? result2 fault-var int-form out-eqs2)
                   '(SA1)
                   '(NOR)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-sa-zero
;;
;;  CALLING FUNCTION(S):  combinational-test
;;
;;  CALLED FUNCTION(S):  replace-with-zero
```

```
;;
;;  PURPOSE:  Calls REPLACE-WITH-ZERO to replace the TEST variable
;;            with the value zero to get a function of
;;            minterms to test a stuck-at-zero condition.
;;
;;  VARIABLES:  function -- the function in which the substitution
;;                          takes place
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-sa-zero function)
    (let* ( (vect-fun (replace-with-zero function '(TEST))))
        vect-fun))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  supplement
;;
;;  CALLING FUNCTION(S):  combinational-test, sequential-test
;;                        combinational-testb, sequential-testb
;;                        combinational-testm, sequential-testm
;;                        supplement
;;
;;  CALLED FUNCTION(S): bar, supplement
;;
;;  PURPOSE:  When a minterm representing a test vector is
;;            generated it does not always contain all of the
;;            primary inputs of the circuit.  When this happens
;;            it indicates that the missing inputs can be set to
;;            either a logic-zero or logic-one value.  Supplement
;;            fills in the missing variables in their logic-one
;;            representation.  The function searches the vector
;;            comparing all input vectors to determine if they
;;            are present in complemented or uncomplemented form.
;;            If not then the uncomplemented form is tacked on the
;;            end.
;;
;;  VARIABLES:  vector -- the generated test vector
;;
;;              sup-vars -- the list of input variables
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (supplement vector sup-vars)
```

```
  (if (null? vector) nil
    (if (null? sup-vars) vector
        (if (or (member (car sup-vars)  vector)
                (member (bar (car sup-vars)) vector))
            (supplement vector (cdr sup-vars))
            (begin
              (let* ((new-vector (cons (car sup-vars) vector)))
              (supplement new-vector (cdr sup-vars))))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  mksops
;;
;;  CALLING FUNCTION(S):  combinational-test, bridge-fault-com,
;;                        bridge-fault-seq, multiple-fault-com,
;;                        multiple-fault-seq, mksops
;;
;;  CALLED FUNCTION(S):  make-sop, mksops
;;
;;  PURPOSE:  This function takes a list of equations and changes
;;            them individually into sum-of-product form using
;;            MAKE-SOP.  The equations are left separated
;;            as opposed to collapsing them into one equation.
;;
;;  VARIABLES:  out-eqs -- the list of equations to be transformed;
;;                         in this case the list of output
;;                         equations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (mksops out-eqs)
   (if (null? out-eqs) nil
     (let* ((new-eq (make-sop (list (car out-eqs)))))
        (cons new-eq (mksops (cdr out-eqs))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-result1
;;
;;  CALLING FUNCTION(S):  combinational-test, get-result1
;;
;;  CALLED FUNCTION(S):  output-vector, bar, get-result1
;;
```

```
;;  PURPOSE:  This function outputs the generated vector to the
;;            user to apply it to the circuit.  It then iteratively
;;            collects the output results from the user and
;;            combines these with the minterm used to get the
;;            vector.
;;
;;  VARIABLES:  vector -- the minterm representing the test vector
;;
;;              outputs -- a list of the circuit's output nodes
;;
;;              flag -- flag starts as false and is changed to
;;                      true on successive calls to keep from
;;                      sending the "apply vector" part of the
;;                      function to the screen more than once
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-result1 vector outputs flag)
  (if (null? outputs) nil
   (begin
   (if flag
    (begin
    (newline)
    (writeln "APPLY THE FOLLOWING VECTOR TO TEST A STUCK-AT-ZERO
              CONDITION")
    (writeln "ON THE SUSPECTED FAULTY LINE:")
    (newline)
    (output-vector vector)
    (newline)))
    (display "INPUT THE RESULT FROM OUTPUT ")
    (display (car outputs))
    (writeln " -- 0 or 1:")
    (let* ( (ans (read)))
    (if (equal? ans 0)
        (append (list (cons (car outputs) vector)) (get-result1
                                                vector (cdr outputs)
                                                   nil))
        (append (list (cons (bar (car outputs)) vector))
                      (get-result1 vector (cdr outputs) nil)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  output-vector
;;
```

```
;;  CALLING FUNCTIONS(S):  get-result1, get-result2, get-res3,
;;                         get-res4, get-result, get-result-s,
;;                         output-vector
;;
;;  CALLED FUNCTION(S):  output-vector
;;
;;  PURPOSE:  This function sends the test vector to the screen.
;;            If a particular variable is in complemented form
;;            (noted by being enclosed in parens) then it is
;;            set to zero in the circuit; if uncomplemente then
;;            it is set to one.
;;
;;  VARIABLES:  vector -- the minterm representing the test vector
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (output-vector vector)
    (newline)
    (if (null? vector) nil
      (begin
       (if (symbol? (car vector))
           (begin (display (car vector))
                  (writeln " = 1"))
           (begin (display (bar (car vector)))
                  (writeln " = 0")))
       (output-vector (cdr vector)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  bad-result?
;;
;;  CALLING FUNCTION(S):  combinational-test, sequential-test
;;
;;  CALLED FUNCTION(S):  mult, complement, related?, bad-result?
;;
;;  PURPOSE:  This function compares the terms formed by
;;            GET-RESULT1(2) to the output equations associated
;;            with the output used to get the term.  The
;;            association is made by getting the terms in the same
;;            order as the ouput equations are listed in.  The
;;            function uses RELATED? to insure that comparisons
;;            are only made with outputs that the suspected fault
;;            node is accessible to.  The comparison is done by
;;            complementing the output equation and ANDing it with
```

```
;;            the term.  If the result is zero then a fault exists.
;;            This is contrasted with the description in the thesis
;;            which ANDs the output equation with the term, and
;;            interprets a zero result as good.  The results are
;;            the same.  If a fault exists this function returns
;;            true, otherwise false.
;;
;;  VARIABLES:  test-results -- the terms formed in GET-RESULT1(2)
;;
;;              fault-var -- the suspected fault node
;;
;;              inter-form -- the original unmodified individual
;;                            CCEs
;;
;;              output-eqs -- a list of the circuit's output
;;                            equations in sum-of-products form
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;to be determined at fault the node must be related logically to
;output in question and the output must be in error
(define (bad-result? test-results fault-var inter-form output-eqs)
    (if (or (null? output-eqs) (null? test-results)) nil
        (if (and (null? (mult (complement (car output-eqs))
                              (list (car test-results))))
                 (related? fault-var (car output-eqs) inter-form))
            t
            (bad-result? (cdr test-results) fault-var inter-form
                 (cdr output-eqs)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  related?
;;
;;  CALLING FUNCTION(S):  bad-result?, bridged?, faulty?
;;
;;  CALLED FUNCTION(S):  related-2?
;;
;;  PURPOSE:  This function determines whether or not a particular
;;            node has access to a particular output.
;;
;;  VARIABLES:  fault-var -- suspected fault node
;;
;;              output-eq -- the output equation containing the
```

```
;;                          output to be checked
;;
;;              inter-form -- the original individual CCEs
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (related? fault-var output-eq inter-form)
        (related-2? fault-var output-eq inter-form inter-form))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  related-2?
;;
;;  CALLING FUNCTION(S):  related?, related-2?
;;
;;  CALLED FUNCTION(S):  flatten, related-2?
;;
;;  PURPOSE:  This function does the check for access by first
;;            checking the output equation for the variable.  If
;;            it is not directly related to the output in question
;;            then the function checks other gates in the circuit
;;            to see if it is indirectly related.
;;
;;  VARIABLES:  fault-var -- the suspected fault node, could be any
;;                           variable that we choose to check for
;;                           access
;;
;;              output-eq -- the equation containing the output
;;                           node that we are checking for access
;;                           to
;;
;;              inter-form  -- the original individual CCEs
;;
;;              inter-form-u -- the original individual CCEs,
;;                              required twice because the other
;;                              inter-form gets modified during the
;;                              recursion
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (related-2? fault-var output-eq inter-form inter-form-u)
     (if (member fault-var (flatten output-eq)) t
      (if (null? inter-form) nil
         (if (and (member fault-var (flatten (car inter-form)))
```

```
                (not (equal? fault-var (cadr (car inter-form)))))
            (related-2? (cadr (car inter-form)) output-eq
                                inter-form-u inter-form-u)
            (related-2? fault-var output-eq (cdr inter-form)
                                                inter-form-u)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-sa-one
;;
;;  CALLING FUNCTION(S):  combinational-test
;;
;;  CALLED FUNCTION(S):  replace-with-one
;;
;;  PURPOSE:  Calls REPLACE-WITH-ONE to replace the TEST variable
;;            with the value one to get a function of
;;            minterms to test a stuck-at-one condition.
;;
;;  VARIABLES:  function -- the function in which the substitution
;;                          takes place
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (get-sa-one function)
    (let* ( (vect-fun (replace-with-one function '(TEST))))
                vect-fun))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-result2
;;
;;  CALLING FUNCTION(S):  combinational-test, get-result2
;;
;;  CALLED FUNCTION(S):  output-vector, get-result2
;;
;;  PURPOSE:  This function is the same as GET-RESULT1 but is run
;;            for a stuck-at-one test.
;;
;;  VARIABLES:  vector -- the minterm representing the test vector
;;
;;              outputs -- circuit outputs
;;
;;              flag -- used to avoid repeatint "output vector"
;;                      routine
```

```
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-result2 vector outputs flag)
  (if (null? outputs) nil
   (begin
   (if flag
    (begin
    (newline)
    (writeln "APPLY THE FOLLOWING VECTOR TO TEST A STUCK-AT-ONE
                                                        CONDITION")
    (writeln "ON THE SUSPECTED FAULTY LINE:")
    (newline)
    (output-vector vector)
    (newline)))
    (display "INPUT THE RESULT FROM OUTPUT ")
    (display (car outputs))
    (writeln " -- 0 or 1:")
    (let* ( (ans (read)))
    (if (equal? ans 0)
        (append (list (cons (car outputs) vector))
                                 (get-result2 vector (cdr outputs)
                                                        nil))
        (append (list (cons (bar (car outputs)) vector))
                          (get-result2 vector (cdr outputs) nil)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  output-results
;;
;;  CALLING FUNCTION(S):  single-fault-com, single-fault-seq
;;
;;  CALLED FUNCTION(S):  none
;;
;;  PURPOSE:  This function outputs the appropriate test result to
;;            the user.
;;
;;  VARIABLES:  results -- generated by COMBINATIONAL-TEST when
;;                         the final test result is determined
;;
;;              fault-var -- the suspected fault node
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (output-results results fault-var)
     (newline)
     (newline)
     (cond ((member 'SA0 results)
            (begin (display "LINE ") (display fault-var)
            (writeln " IS STUCK-AT-0.")))
           ((member 'SA1 results)
            (begin (display "LINE ") (display fault-var)
            (writeln " IS STUCK-AT-1.")))
           ((member 'NOR results)
            (begin (display "LINE ") (display fault-var)
            (writeln " IS NORMAL.")))
           (else
            (writeln "UNABLE TO GENERATE TEST WITH INFORMATION
                                                    GIVEN."))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  again?
;;
;;  CALLING FUNCTION(S):  single-fault-com, single-fault-seq
;;
;;  CALLED FUNCTION(S):  none
;;
;;  PURPOSE:  This function provides the user with the opportunity
;;            to test the same circuit for another faulty node.
;;
;;  VARIABLES:  none
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (again?)
   (newline)
   (writeln "WOULD YOU LIKE TO RUN A SINGLE FAULT TEST ON ANOTHER")
   (writeln "NODE IN THE CIRCUIT?")
   (writeln "TYPE y<rtn> OR n<rtn>.")
   (let ( (answer (read)))
       (if (equal? answer 'y) t
           nil)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FILENAME:  test1s.s
;;
;;  FILES REQUIRED FOR CALLED FUNCTIONS:  boolean.s, test1.s
;;                                        utils2.s, menu.s
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  single-fault-seq
;;
;;  CALLING FUNCTION(S):  menu
;;
;;  CALLED FUNCTION(S):  get-fault-variable, cut-node,
;;                       get-state-info, get-vector-function,
;;                       sequential-test, output-results, again?
;;                       single-fault-seq, menu
;;
;;  PURPOSE:  This function performs single stuck-at fault
;;            diagnosis in sequential circuits.  Operation is much
;;            the same as SINGLE-FAULT-COM.  The differences are
;;            primarily in SEQUENTIAL-TEST.  The information
;;            obtained in GET-STATE-INFO is used to attempt to
;;            generate test vectors in SEQUENTIAL-TEST.  This
;;            information, obtained from the user, gives us the two
;;            known possibilities for the state of the tested
;;            circuit.  It is necessary to know, or be able to set,
;;            the present state of the circuit.
;;
;;  VARIABLES:  intermediate-format -- the unmodified system of
;;                                     individual CCEs
;;
;;              internal-nodes -- the internal nodes of the circuit
;;
;;              output-nodes -- the output nodes of the circuit
;;
;;              input-nodes -- the input nodes of the circuit
;;
;;              output-equations -- the output equations of the
;;                                  circuit in prefix form
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (single-fault-seq intermediate-format internal-nodes
                output-nodes input-nodes output-equations)
    (newline)
    (let* ( (fault-variable (get-fault-variable))
            (new-format (cut-node intermediate-format input-nodes
                      output-equations fault-variable))
            (state-info (get-state-info))
            (vector-function (get-vector-function new-format
                                internal-nodes output-nodes
                                output-equations))
            (results (sequential-test vector-function state-info
                            input-nodes output-nodes
                            output-equations intermediate-format
                            fault-variable)))

          (begin
          (output-results results fault-variable)
          (if (again?) (single-fault-seq intermediate-format
                                internal-nodes output-nodes
                                input-nodes output-equations)
             (begin
             (read-line)
             (menu))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-state-info
;;
;;  CALLING FUNCTION(S):  single-fault-seq, bridge-fault-seq,
;;                        multiple-fault-seq
;;
;;  CALLED FUNCTION(S):  get-memory-nodes1, get-mem-values1,
;;                       get-mem-values2
;;
;;  PURPOSE:  This function prompts the user for the current-state
;;            variables (the outputs of sequential elements),
;;            their present values and their values upon
;;            resetting the circuit.  These values are used in
;;            SEQUENTIAL-TFST to attempt to generate test vectors.
;;            The present values of the state variables are first
;;            tried, and if unsuccessful the reset values are
;;            tried.  If the latter work then the user is directed
;;            to reset the circuit prior to testing.
;;
```

```
;;  VARIABLES:  none
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-state-info)
      (display "ENTER THE CURRENT STATE VARIABLES OF THE CIRCUIT.
THESE ARE THE VARIABLES
THAT LABEL THE OUTPUTS OF SEQUENTIAL ELEMENTS.  ENTER VARIABLES ONE
AT A TIME FOLLOWED BY <rtn>.  ENTER ''0'', <rtn> WHEN DONE.")
      (writeln)
      (let* ((mem-nodes (get-memory-nodes1 nil)))
              (display mem-nodes)
             (list mem-nodes (get-mem-values1 mem-nodes)
              (get-mem-values2 mem-nodes)) ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-memory-nodes1
;;
;;  CALLING FUNCTION(S):  get-state-info
;;
;;  CALLED FUNCTION(S):  get-memory-nodes1
;;
;;  PURPOSE:  This function gets the current-state variables of the
;;            circuit.
;;
;;  VARIABLES:  nodes -- starts as nil, this variable collects the
;;                       state variables
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-memory-nodes1 nodes)
       (let* ((node (read)))
        (if (equal? node 0) nodes
           (let*(
           (new-nodes (cons node nodes)))
           (get-memory-nodes1 new-nodes)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-mem-values1
;;
;;  CALLING FUNCTION(S):  get-state-info
;;
```

```
;;  CALLED FUNCTION(S):  get-mem-values1
;;
;;  PURPOSE:  This function gets the current values of the state
;;            variables.
;;
;;  VARIABLES:  nodes -- the state variables
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-mem-values1 nodes)
     (if (null? nodes) nil
       (if (atom? (car nodes))
       (begin
       (newline)
       (display "ENTER THE CURRENT VALUE OF STATE VARIABLE ")
       (display (car nodes))
       (display ".")
       (newline)
       (writeln "TYPE 0 OR 1 AND <RTN>:")
       (newline)
       (let* ( (value (read)))
              (cons value (get-mem-values1 (cdr nodes))))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-mem-values2
;;
;;  CALLING FUNCTION(S):  get-state-info
;;
;;  CALLED FUNCTION(S):  get-mem-values2
;;
;;  PURPOSE:  This function gets the reset values of the state
;;            variables.
;;
;;  VARIABLES:  nodes -- state variables
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-mem-values2 nodes)
     (if (null? nodes) nil
       (begin
       (newline)
       (display "ENTER THE VALUE OF STATE VARIABLE ")
       (display (car nodes))
```

```
      (display " WHEN THE CIRCUIT IS RESET.")
      (newline)
      (writeln "TYPE 0 OR 1 AND <RTN>:")
      (newline)
      (let* ( (value (read)))
            (newline)
            (cons value (get-mem-values2 (cdr nodes)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  sequential-test
;;
;;  CALLING FUNCTION(S):  single-fault-seq
;;
;;  CALLED FUNCTION(S):  get-sa-zero, get-terms, remove-mem-nodes
;;                       supplement, init-eqs, get-res3,
;;                       bad-result?, get-mem-values1, get-sa-one
;;                       get-res4
;;
;;  PURPOSE:  This function generates test vectors, prompts the
;;            user to apply them, and processes the results of
;;            application.  GET-TERMS is the function that attempts
;;            to derive the vectors using the two possible circuit
;;            states.  When a minterm representing a vector is
;;            obtained it is supplemented with missing primary
;;            input variables.  Only primary inputs other than
;;            current-state variables are sent to the user for
;;            application.  INIT-EQS initializes the output
;;            equations with the state values used to generate the
;;            test vector.  This must be done before comparison
;;            using BAD-RESULT?.  If the stuck-at-zero test fails
;;            then a stuck-at-one test is conducted.  Since the
;;            stuck-at-zero test possibly changes the values of the
;;            state variables they must be obtained again using
;;            GET-MEM-VALUES1.
;;
;;  VARIABLES:  function -- the function used for generating test
;;                          vectors; derived using
;;                          GET-VECTOR-FUNCTION
;;
;;              state-info -- list of state variables, list of
;;                            current state values, list of reset
;;                            state values
;;
```

```
;;              input-nodes -- list of circuit input variables
;;
;;              output-nodes -- list of circuit output variables
;;
;;              output-equations -- list of circuit output
;;                                  equations
;;
;;              inter-form -- original unmodified individual CCEs
;;
;;              fault-var -- suspected fault node
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (sequential-test function state-info input-nodes
               output-nodes output-equations inter-form fault-var)
    (newline)
    (let* ( (z-function (get-sa-zero function))
            (terms (get-terms z-function state-info))
            (z-term (caar terms)))
            (if (null? z-term) nil
             (let* (
                 (memory-nodes (car state-info))
                 (input-nodes2 (remove-mem-nodes memory-nodes
                                                 input-nodes))
                 (z-vector (supplement z-term input-nodes2))
                 (out-eqs2 (init-eqs output-equations state-info
                                     (cadr terms)))
                 (result1 (get-res3 z-vector output-nodes t)))
                 (if (bad-result? result1 fault-var inter-form
                                                 out-eqs2)
                    '(SA0)
                     (begin
                     (display "THE NODE IS NOT STUCK-AT-ZERO.")
                     (let* (
                       (new-state-info (list memory-nodes
                                 (get-mem-values1 memory-nodes)
                                             (caddr state-info)))
                       (o-fun (get-sa-one function))
                       (o-term (caar (get-terms o-fun
                                                new-state-info))))
                      (if (null? o-term) nil
                          (let* (
                           (o-vector (supplement o-term input-nodes2))
                           (result2 (get-res4 o-vector
                                              output-nodes t)))
```

```
                    (if (bad-result? result2 fault-var
                                     inter-form out-eqs2)
                    '(SA1)
                    '(NOR) ))))))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-terms
;;
;;  CALLING FUNCTION(S):  sequential-test, sequential-testb,
;;                        sequential-testm
;;
;;  CALLED FUNCTION(S):  try-current-state, try-with-reset
;;
;;  PURPOSE:  This function uses the state information from
;;            GET-STATE-INFO along with the function from
;;            SEQUENTIAL-TEST to attempt to generate
;;            a test vector.  The minterms in the function
;;            contain the states that the state variables must be
;;            in to apply the associated primary inputs.  The two
;;            state possibilities are then substituted to see
;;            if they match one of the necessary conditions.  If
;;            successful then the result is a minterm(s)
;;            representing the vector(s) that are
;;            capable of diagnosing the appropriate fault.
;;            The identity of the state that is eventually used to
;;            get the minterm is maintained by tacking a 0 or 1 one
;;            to the end of the generated minterm(s).
;;
;;  VARIABLES:  fun -- the function generated by SEQUENTIAL-TEST
;;                     by either substituting 0 or 1 into the
;;                     GET-VECTOR-FUNCTION function depending on
;;                     the fault being diagnosed
;;
;;              state-info --  list of lists containing the state
;;                             variables and their current and
;;                             reset values
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-terms fun state-info)
     (let* ((v1 (try-current-state fun state-info)))
              (if (null? v1)
              (try-with-reset fun state-info)
```

```
                (cons v1 '(0)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  try-current-state
;;
;;  CALLING FUNCTION(S):  get-terms
;;
;;  CALLED FUNCTION(S):  initializef
;;
;;  PURPOSE:  Attempts to generate minterms for vectors using the
;;            current states of the memory elements.
;;
;;  VARIABLES:   fun -- the function generated by SEQUENTIAL-TEST
;;                      by either substituting 0 or 1 into the
;;                      GET-VECTOR-FUNCTION function depending on
;;                      the fault being diagnosed
;;
;;               state-info --  list of lists containing the state
;;                              variables and their current and
;;                              reset values
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (try-current-state fun state-info)
     (let* ((mem-nodes (car state-info))
            (init-vals (car (cdr state-info)))
            (init-fun  (initializef fun mem-nodes init-vals)))
         init-fun))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  try-with-reset
;;
;;  CALLING FUNCTION(S):  get-terms
;;
;;  CALLED FUNCTION(S):  initializef
;;
;;  PURPOSE:  Attempts to generate minterms using the reset state
;;            of the circuit.
;;
;;  VARIABLES:   fun -- the function generated by SEQUENTIAL-TEST
;;                      by either substituting 0 or 1 into the
```

```
;;                    GET-VECTOR-FUNCTION function depending on
;;                    the fault being diagnosed
;;
;;             state-info --  list of lists containing the state
;;                            variables and their current and
;;                            reset values
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (try-with-reset fun state-info)
     (newline)
     (newline)
     (display "NO VECTOR CAPABLE USING CURRENT STATE, TRYING RESET
               STATE.")
     (newline)
     (newline)
     (let* ((mem-nodes (car state-info))
            (reset-vals (car (cddr state-info)))
            (reset-fun  (initializef fun mem-nodes reset-vals)))
        (if (null? reset-fun) nil
            (begin
            (display "RESET CIRCUIT PRIOR TO APPLYING TEST
                      VECTOR.")
            (newline)
            (cons reset-fun '(1))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  initializef
;;
;;  CALLING FUNCTION(S):  try-current-state, try-with-reset
;;                        intializef
;;
;;  CALLED FUNCTION(S): replace-with-one, replace-with-zero
;;                      intializef
;;
;;  PURPOSE:  This function substitutes a specified list of nodes
;;            in a given function with specified values.  The order
;;            of the list of nodes corresponds to the order of the
;;            values to be substituted in.
;;
;;  VARIABLES:  func -- the function to substitute in
;;
;;              nodes -- the nodes to be replaced
```

```
;;
;;              values -- the values to place in the function
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (initializef func nodes values)
     (if (equal? nodes nil)
         func
         (begin
         (if (equal? (car values) 1)
           (initializef (replace-with-one func (list (car nodes)))
                                (cdr nodes) (cdr values))
           (initializef (replace-with-zero func (list (car nodes)))
                                (cdr nodes) (cdr values))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  remove-mem-nodes
;;
;;  CALLING FUNCTION(S):  sequential-test
;;
;;  CALLED FUNCTION(S):  remove, remove-mem-nodes. bar
;;
;;  PURPOSE:  The original list of input values includes the
;;            current state variables of the circuit.  These must
;;            be removed before this list is used to supplement the
;;            minterm representing the test vector.  This function
;;            is a generic removal function to do so.
;;
;;  VARIABLES:  mem-nodes -- the nodes to go
;;
;;              in-nodes -- the list to remove from
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (remove-mem-nodes mem-nodes in-nodes)
    (if (null? mem-nodes) in-nodes
       (let* ( (rem1 (remove (car mem-nodes) in-nodes))
               (rem2 (remove (bar (car mem-nodes)) rem1)))
         (remove-mem-nodes (cdr mem-nodes) rem2))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;
;;  FUNCTION:  init-eqs
;;
;;  CALLING FUNCTION(S):  sequential-test
;;
;;  CALLED FUNCTION(S):  init-eqs1
;;
;;  PURPOSE:  This function initializes the output equations using
;;            the state of the memory elements that was used to
;;            get the test vector minterm.  This is a necessary
;;            step prior to comparing for a fault condition so that
;;            the configuration of the circuit prior to vector
;;            application is the compared configuration.
;;
;;  VARIABLES:  out-eqs -- the list of circuit outputs
;;
;;              state-info -- list of lists containing the current
;;                            state variables and their present and
;;                            reset states
;;
;;              init-val -- a variable that was tagged on the end
;;                          of the test-vector minterm to maintain
;;                          the identity of the state used to
;;                          generate the minterm
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (init-eqs out-eqs state-info init-val)
     (if (equal? init-val 0)
         (init-eqs1 out-eqs (car state-info) (cadr state-info))
         (init-eqs1 out-eqs (car state-info) (caddr state-info))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  init-eqs1
;;
;;  CALLING FUNCTION(S):  init-eqs
;;
;;  CALLED FUNCTION(S):  make-sop, initializef
;;
;;  PURPOSE:  This function does the actual initialization using
;;            the values passed to it by INIT-EQ.
;;
;;  VARIABLES:   out-eqs -- circuit output equations
```

```
;;
;;              state-vars -- the variables to replace
;;
;;              init-vals -- the values to replace them with
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (init-eqs1 out-eqs state-vars init-vals)
     (if (null? out-eqs) nil
        (let* ((fun (make-sop (list (car out-eqs))))
               (init-fun (initializef fun state-vars init-vals)))
          (cons init-fun (init-eqs1 (cdr out-eqs) state-vars init-vals)))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-res3
;;
;;  CALLING FUNCTION(S):  sequential-test
;;
;;  CALLED FUNCTION(S):  output-vector
;;
;;  PURPOSE:  This function outputs the stuck-at-zero test vector
;;            to the user and forms the terms required to later
;;            compare based on the resulting outputs.
;;
;;  VARIABLES:  vector -- the minterm representing the vector to be
;;                        applied
;;
;;              outputs -- a list of the circuit outputs
;;
;;              flag -- a flag used to avoid sending the "output
;;                      vector" routine more than once
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-res3 vector outputs flag)
  (if (null? outputs) nil
   (begin
   (if flag
    (begin
    (newline)
    (writeln "APPLY THE FOLLOWING VECTOR TO TEST A STUCK-AT-ZERO CONDITION")
    (writeln "ON THE SUSPECTED FAULTY LINE:")
```

```
    (newline)
    (output-vector vector)
    (newline)
    (writeln "NEXT YOU WILL BE PROMPTED FOR THE RESULTING OUTPUT VALUES.")
    (writeln "RECALL THAT OUTPUTS OF SEQUENTIAL ELEMENTS ARE NOW CONSIDERED")
    (writeln "TO BE PRIMARY OUTPUTS.")))
    (display "INPUT THE RESULT FROM OUTPUT ")
    (display (car outputs))
    (writeln " -- 0 or 1:")
    (let* ( (ans (read)))
    (if (equal? ans 0)
        (append (list (cons (car outputs) vector)) (get-res3
                                                      vector
                                                    (cdr outputs) nil))
        (append (list (cons (bar (car outputs)) vector)) (get-res3
                              vector (cdr outputs) nil)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-res4
;;
;;  CALLING FUNCTION(S):  sequential-test
;;
;;  CALLED FUNCTION(S):  output-vector
;;
;;  PURPOSE:  This function outputs the stuck-at-one test vector
;;            to the user and forms the terms required to later
;;            compare based on the resulting outputs.
;;
;;  VARIABLES:   vector -- the minterm representing the vector to be
;;                         applied
;;
;;               outputs -- a list of the circuit outputs
;;
;;               flag -- a flag used to avoid sending the "output
;;                       vector" routine more than once
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-res4 vector outputs flag)
  (if (null? outputs) nil
   (begin
   (if flag
    (begin
```

```
(newline)
(writeln "APPLY THE FOLLOWING VECTOR TO TEST A STUCK-AT-ONE CONDITION")
(writeln "ON THE SUSPECTED FAULTY LINE:")
(newline)
(output-vector vector)
(newline)
(writeln "NEXT YOU WILL BE PROMPTED FOR THE RESULTING OUTPUT VALUES.")
(writeln "RECALL THAT OUTPUTS OF SEQUENTIAL ELEMENTS ARE NOW CONSIDERED")
(writeln "TO BE PRIMARY OUTPUTS.")))
(display "INPUT THE RESULT FROM OUTPUT ")
(display (car outputs))
(writeln " -- 0 or 1:")
(let* ( (ans (read)))
(if (equal? ans 0)
    (append (list (cons (car outputs) vector))
                                (get-res4 vector (cdr outputs)
                                                  nil))
    (append (list (cons (bar (car outputs))
                                     vector)) (get-res4 vector
                                       (cdr outputs) nil)))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FILENAME:  test2.s
;;
;;  FILES REQUIRED FOR CALLED FUNCTIONS:  menu.s, test1.s, utils2.s
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  bridge-fault-com
;;
;;  CALLING FUNCTION(S):  menu
;;
;;  CALLED FUNCTION(S):  get-fault-variables, cut-node,
;;                       get-vector-function, combinational-testb
;;                       mksops, bridged?, output-resultsb
;;                       bridge-fault-com, menu
;;
;;  PURPOSE:  This function performs diagnosis of bridge faults in
;;            combinational circuits.  The user is prompted for the
;;            variables labeling the suspected bridged lines using
;;            GET-FAULT-VARIABLES.  CUT-NODE is used, as before, to
;;            isolate the suspected faulty lines.  The equations
;;            descriptions are combined using GET-VECTOR-FUNCTION.
;;            COMBINATIONAL-TESTB generates vectors, forms the
;;            terms to be compared for diagnosis and passes them
;;            back to the calling routine.  BRIDGED? does the
;;            comparison to the output equations to determine if
;;            a fault has occurred.  The results are then provided
;;            to the user.
;;
;;  VARIABLES:  intermediate-format -- the individual CCEs
;
;;              internal-nodes -- a list of the internal circuit
;;                                nodes
;;
;;              output-nodes -- a list of the output nodes
;;
;;              input-nodes -- a list of the input nodes
;;
;;              output-equations -- a list of the circuit's output
;;                                  equations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (bridge-fault-com intermediate-format internal-nodes
                    output-nodes input-nodes output-equations)
    (newline)
    (let* ( (fault-variables (get-fault-variables))
           (new-format1 (cut-node intermediate-format input-nodes
                    output-equations (car fault-variables)))
           (new-format2 (cut-node new-format1 input-nodes
                output-equations (car (cdr fault-variables))))
           (vector-function (get-vector-function new-format2
                   internal-nodes output-nodes output-equations))
           (test-results (combinational-testb vector-function
                   input-nodes output-nodes output-equations))
           (out-eqs2 (mksops output-equations)))
    (if (null? test-results) (writeln "NO TEST POSSIBLE FOR THIS
                                    COMBINATION.")
        (let* ( (results (bridged? test-results fault-variables
                     intermediate-format out-eqs2)))
        (output-resultsb results fault-variables)))

        (if (againb?) (bridge-fault-com intermediate-format
                                internal-nodes
                     output-nodes input-nodes output-equations)
            (begin
            (read-line)
            (menu)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-fault-variables
;;
;;  CALLING FUNCTION(S):  bridge-fault-com, bridge-fault-seq
;;
;;  CALLED FUNCTION(S):  none
;;
;;  PURPOSE:  This function prompts the user for the suspected
;;            fault nodes and reads them.
;;
;;  VARIABLES:  none
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-fault-variables)
     (newline)
```

```
     (writeln "ENTER THE VARIABLES THAT LABEL THE SUSPECTED FAULTY
               LINES.")
     (writeln "ENTRIES SHOULD BE MADE ONE AT A TIME WITH <rtn>
               TYPED")
     (writeln "BETWEEN EACH ENTRY.")
     (newline)
     (let* ((input1 (read))
            (inputs (cons input1 (list (read)))))
         (newline)
         (display "PROCESSING......")
         (newline)
         (newline)
         inputs))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  combinational-testb
;;
;;  CALLING FUNCTION(S):  bridge-fault-com
;;
;;  CALLED FUNCTION(S):  replace-with-zero, replace-with-one,
;;                       mult, supplement, get-result
;;
;;  PURPOSE:  This function generates a test vector, prompts the
;;            user to apply it and forms the terms to be later
;;            compared to output equations based on the resulting
;;            output values.
;;
;;  VARIABLES:  function -- the test vector function generated by
;;                          get-vector-function
;;
;;              input-nodes -- list of circuit inputs
;;
;;              output-nodes -- list of circuit outputs
;;
;;              output-eqs -- list of circuit output equations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (combinational-testb function input-nodes output-nodes
                     output-eqs)
   (let* ( (z (replace-with-zero function '(TEST)))
           (o (replace-with-one function ' (TEST)))
           (zo (mult z o))
```

```
        (vector (car zo))
        (vector2 (supplement vector input-nodes)))
  (if (null? zo) nil
      (get-result vector2 output-nodes t))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-result
;;
;;  CALLING FUNCTION(S):  combinational-testb, get-result
;;
;;  CALLED FUNCTION(S):  output-vector, get-result
;;
;;
;;  PURPOSE:  This function outputs the test vector to the user and
;;            accumulates the terms to be later compared to output
;;            equations.
;;
;;  VARIABLES:  vector -- the minterm representing the test vector
;;
;;              outputs -- list of circuit outputs
;;
;;              flag -- a flag used to avoid sending the "output
;;                      vector" message more than once
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-result vector outputs flag)
  (if (null? outputs) nil
   (begin
   (newline)
   (if flag
    (begin
    (newline)
   (writeln "APPLY THE FOLLOWING VECTOR TO TEST A FAULT CONDITION")
    (writeln "ON THE SUSPECTED FAULTY LINES:")
    (newline)
    (output-vector vector)
    (newline)))
    (display "INPUT THE RESULT FROM OUTPUT ")
    (display (car outputs))
    (writeln " -- 0 or 1:")
    (let* ( (ans (read)))
    (if (equal? ans 0)
```

```
        (append (list (cons (car outputs) vector)) (get-result
                                                vector (cdr outputs)
                                                       nil))
        (append (list (cons (bar (car outputs)) vector))
                (get-result vector (cdr outputs) nil)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  bridged?
;;
;;  CALLING FUNCTION(S):  bridge-fault-com, bridge-fault-seq
;;                        bridged?
;;
;;  CALLED FUNCTION(S):  mult, complement, related?, bridged?
;;
;;  PURPOSE:  This function does the comparison with the output
;;            equations to determine if the suspected variables
;;            are bridged.
;;
;;  VARIABLES:  test-results --  the terms formed in
;;                               COMBINATIONAL-TESTB based on the
;;                               test vector minterm and resulting
;;                               output
;;
;;              fault-vars -- the suspected fault nodes
;;
;;              inter-form -- the original individual CCEs
;;
;;              output-eqs -- a list of the circuit's output
;;                            equations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;to be determined at fault either node must be related logically to the
;output in question and the output must be in error
(define (bridged? test-results fault-vars inter-form output-eqs)
    (if (or (null? output-eqs) (null? test-results))
             nil
        (if (and (null? (mult (complement (car output-eqs))
                              (list (car test-results))))
                 (or (related? (car fault-vars) (car output-eqs) inter-form)
                     (related? (car (cdr fault-vars)) (car output-eqs)
                                                      inter-form)))
            '(B)
```

```
            (bridged? (cdr test-results) fault-vars inter-form
                  (cdr output-eqs)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  output-resultsb
;;
;;  CALLING FUNCTION(S):  bridge-fault-com
;;
;;  CALLED FUNCTION(S):  none
;;
;;
;;  PURPOSE:  This function provides the user with the results of
;;            the diagnosis.
;;
;;  VARIABLES:  results -- results determined in BRIDGED? routine
;;
;;              fault-vars -- suspected fault nodes
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (output-resultsb results fault-vars)
     (newline)
     (newline)
     (display "LINES ")
     (display (car fault-vars))
     (display " AND ")
     (display (car (cdr fault-vars)))
     (if (member 'B results)
         (writeln " ARE BRIDGED.")
         (writeln " ARE NORMAL.")))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  againb?
;;
;;  CALLING FUNCTION(S):  bridge-fault-com, bridge-fault-seq
;;
;;  CALLED FUNCTION(S):  none
;;
;;  PURPOSE:  This function gives the user the opportunity to test
;;            the same circuit for a bridge fault on two other
;;            nodes.
```

```
;;
;;  VARIABLES:  none
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (againb?)
   (newline)
   (writeln "WOULD YOU LIKE TO RUN A BRIDGE FAULT TEST ON ANOTHER")
   (writeln "SET OF NODES IN THE CIRCUIT?")
   (writeln "TYPE y<rtn> OR n<rtn>.")
   (let ( (answer (read)))
       (if (equal? answer 'y) t
           nil)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FILENAME:  test2s.s
;;
;;  FILES REQUIRED FOR CALLED FUNCTIONS:  boolean.s, test1.s,
;;                              test2.s, menu.s, test1s.s, utils2.s
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  bridge-fault-seq
;;
;;  CALLING FUNCTION(S):  menu, bridge-fault-seq
;;
;;  CALLED FUNCTION(S):  get-fault-variables, cut-node,
;;                       get-state-info, get-vector-function,
;;                       sequential-testb, init-eqs, bridged?,
;;                       output-resultsb, bridge-fault-seq, menu
;;
;;  PURPOSE:  This function performs diagnosis of bridge faults in
;;            sequential circuits.  It does this much in the same
;;            way as BRIDGE-FAULT-COM, only using the additional
;;            state information to generate test vectors as
;;            described in SINGLE-FAULT-SEQ.
;;
;;  VARIABLES:  intermediate-format -- the list of individual CCEs
;;
;;              internal-nodes -- list of circuit internal nodes
;;
;;              output-nodes -- list of circuit output nodes
;;
;;              input-nodes -- list of circuit input nodes
;;
;;              output-equations -- list of circuit output
;;                                  equations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (bridge-fault-seq intermediate-format internal-nodes
                  output-nodes input-nodes output-equations)
    (newline)
    (let* ( (fault-variables (get-fault-variables))
           (new-format1 (cut-node intermediate-format input-nodes
                     output-equations (car fault-variables)))
```

```
          (new-format2 (cut-node new-format1 input-nodes
                    output-equations (car (cdr fault-variables))))
          (state-info (get-state-info))
          (garb (display "PROCESSING....."))
          (vector-function (get-vector-function new-format2
                              internal-nodes output-nodes
                                          output-equations))
          (test-results (sequential-testb vector-function
                             state-info input-nodes output-nodes
                                          output-equations))
          (out-eqs2 (init-eqs output-equations state-info
                                      (cadr test-results))))

    (if (null? test-results) (writeln "NO TEST POSSIBLE FOR THIS
                                                    COMBINATION.")
        (let* ( (results (bridged? (car test-results)
                    fault-variables intermediate-format out-eqs2)))
        (output-resultsb results fault-variables)))

        (if (againb?) (bridge-fault-seq intermediate-format
                          internal-nodes output-nodes input-nodes
                                                output-equations)
             (begin
             (read-line)
             (menu)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  sequential-testb
;;
;;  CALLING FUNCTION(S):  bridge-fault-seq
;;
;;  CALLED FUNCTION(S):  replace-with-zero, replace-with-one, mult,
;;                       get-terms, remove-mem-nodes, supplement,
;;                       get-result-s
;;
;;  PURPOSE:  This function generates a test vector, prompts the
;;            user to apply it and forms the terms to be used later
;;            in comparison with the output equations.
;;
;;  VARIABLES:  function -- the function generated by
;;                          GET-VECTOR-FUNCTION
;;
;;              state-info -- a list of lists containing the
;;                            current state nodes and their current
```

```
;;                              and reset states
;;
;;              input-nodes -- circuit inputs
;;
;;              output-nodes -- circuit outputs
;;
;;              output-eqs -- circuit output equations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (sequential-testb function state-info input-nodes
            output-nodes output-eqs)
   (let* ((mem-nodes (car state-info))
          (z (replace-with-zero function '(TEST)))
          (o (replace-with-one function ' (TEST)))
          (zo (mult z o))
          (terms (get-terms zo state-info))
          (vector (caar terms)))
  (if (null? vector) nil
    (let* ( (input-nodes2 (remove-mem-nodes mem-nodes input-nodes))
            (vector2 (supplement vector input-nodes2)))
           (cons (get-result-s vector2 output-nodes t)
                          (cdr terms)    )))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-result-s
;;
;;  CALLING FUNCTION(S):  sequential-testb, get-result-s
;;
;;  CALLED FUNCTION(S):  output-vector, get-result-s
;;
;;  PURPOSE:  This function directs the user to apply the test
;;            vector and it then forms the terms to be compared
;;            later to output equations.
;;
;;  VARIABLES:  vector -- the minterm representing the test vector
;;
;;              outputs -- circuit outputs
;;
;;              flag -- a flag used to avoid sending the "output
;;                      vector" message more than once
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (get-result-s vector outputs flag)
  (if (null? outputs) nil
   (begin
   (if flag
    (begin
    (newline)
    (writeln "APPLY THE FOLLOWING VECTOR TO TEST FOR A FAULT
              CONDITION")
    (writeln "ON THE SUSPECTED FAULTY LINES:")
    (newline)
    (output-vector vector)
    (newline)))
    (display "INPUT THE RESULT FROM OUTPUT ")
    (display (car outputs))
    (writeln " -- 0 or 1:")
    (let* ( (ans (read)))
    (if (equal? ans 0)
        (append (list (cons (car outputs) vector)) (get-result-s
                                                      vector
                                                 (cdr outputs) nil))
        (append (list (cons (bar (car outputs)) vector))
                     (get-result-s vector (cdr outputs) nil)))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FILENAME:  test3.s
;;
;;  FILES REQUIRED FOR CALLED FUNCTIONS:  test1.s, test2.s, menu.s,
;;                        boolean.s, utils2.s
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  multiple-fault-com
;;
;;  CALLING FUNCTION(S):  menu, multiple-fault-com
;;
;;  CALLED FUNCTION(S):  get-mfault-variables, cut-nodes,
;;                       get-vector-function, combinational-testm,
;;                       mksops, faulty?, output-resultsm,
;;                       multiple-test-com, againm?, menu
;;
;;  PURPOSE:  This function performs diagnosis of multiple stuck-at
;;            faults in a combinational circuit.  It is much like
;;            the SINGLE-FAULT-COM routine with the additional
;;            software required to handle multiple suspected faulty
;;            nodes and their suspected fault values as specified
;;            by the user.  GET-MFAULT-VARIABLES obtains the
;;            suspected faulty nodes.  GET-VECTOR-FUNCTION and
;;            COMBINATIONAL-TESTM together generate the test
;;            vector.  The results of application are analyzed by
;;            FAULT? and then output to the user.
;;
;;  VARIABLES:  intermediate-format -- the system of individual
;;                                     CCEs
;;
;;              internal-nodes -- list of circuit internal nodes
;;
;;              output-nodes -- list of circuit output nodes
;;
;;              input-nodes -- list of circuit inputs
;;
;;              output-equations -- list of circuit output
;;                                  equations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (multiple-fault-com intermediate-format internal-nodes
                        output-nodes input-nodes output-equations)
    (newline)
    (let* ( (fault-variables (get-mfault-variables))
           (garb (display "PROCESSING....."))
           (new-format (cut-nodes intermediate-format input-nodes
                     output-equations fault-variables)))
     (if (null? new-format)
         (writeln "NO TEST POSSIBLE FOR THIS COMBINATION.")
      (let* (
           (vector-function (get-vector-function new-format
                                internal-nodes output-nodes
                                                output-equations))
           (test-results (combinational-testm vector-function
                          input-nodes output-nodes output-equations
                                fault-variables))
           (out-eqs2 (mksops output-equations)))
    (if (or (null? test-results) (null? vector-function))
             (writeln "NO TEST POSSIBLE FOR THIS COMBINATION.")
        (let* ( (results (faulty? test-results fault-variables
                    intermediate-format out-eqs2)))
          (output-resultsm results fault-variables)))

        (if (againm?) (multiple-fault-com intermediate-format
                                internal-nodes
                        output-nodes input-nodes output-equations)
            (begin
            (read-line)
           (menu)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-mfault-variables
;;
;;  CALLING FUNCTION(S):  multiple-test-com
;;
;;  CALLED FUNCTION(S):  get-vars
;;
;;  PURPOSE:  This function asks the user to provide the number of
;;            fault nodes and then calls GET-VARS to read the
;;            nodes.
;;
;;  VARIABLES:  none
;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-mfault-variables)
     (newline)
     (writeln "ENTER THE NUMBER OF VARIABLES SUSPECTED TO BE
                                                          FAULTY.")
     (writeln "FOLLOW THE RESPONSE WITH <rtn>:")
     (get-vars (read) t))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-vars
;;
;;  CALLING FUNCTION(S):  get-mfault-variables, get-vars
;;
;;  CALLED FUNCTION(S):  get-vars
;;
;;  PURPOSE:  This function prompts the user for the fault nodes
;;            and reads them.
;;
;;  VARIABLES:  num -- a value obtained from the user so the
;;                     routine knows when to stop reading variables
;;
;;              flag -- this flag avoids sending the prompting
;;                      message more than once
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-vars num flag)
   (if (equal? num 0) nil
     (begin
     (if flag
     (begin
      (writeln "ENTER THE VARIABLES THAT LABEL THE FAULTY LINES.
                                                          ENTRIES")
      (writeln "SHOULD BE MADE ONE AT A TIME WITH <rtn> TYPED AFTER
                                                            EACH")
      (writeln "ENTRY.")))
     (cons (read) (get-vars (sub1 num) nil)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  cut-nodes
```

```
;;
;;  CALLING FUNCTION(S):  multiple-fault-com
;;
;;  CALLED FUNCTION(S):  cut-node2, cut-node
;;
;;  PURPOSE:  This function is much like CUT-NODE with the
;;            exception that is cuts several suspected faulty
;;            nodes and replaces them with a number to maintain
;;            their identities.
;;
;;  VARIABLES:  ccf-list -- the system of individual CCEs
;;
;;              input-nodes -- circuit inputs
;;
;;              output-equations -- circuit output equations
;;
;;              fault-variables -- the suspected fault nodes
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (cut nodes ccf-list input-nodes output-equations
                                                    fault-variables)
  (if (null? fault-variables) ccf-list
      (let* ( (replacement-var (length fault-variables))
              (new-ccf (cut-node2 ccf-list input-nodes
                                  output-equations
                          (car fault-variables) replacement-var)))
      (if (null? new-ccf) nil
    (cut-nodes new-ccf input-nodes output-equations
                                        (cdr fault-variables))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  cut-node2
;;
;;  CALLING FUNCTION(S):  cut-nodes
;;
;;  CALLED FUNCTION(S):  check-list, replace-variable, delete-ccf
;;
;;  PURPOSE:  This function cuts one variable at a time.
;;
;;  VARIABLES:  ccf-list -- individual CCEs
;;
;;              input-nodes -- circuit inputs
```

```
;;
;;              output-equations -- circuit output equations
;;
;;              fault-variable -- a suspected fault node from
;;                                CUT-NODES
;;
;;              replacement-var -- the item to replace the fault
;;                                 node with, in this case a number
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (cut-node2 ccf-list input-nodes output-equations
                              fault-variable replacement-var)
      (if (null? ccf-list)
          (if (check-list input-nodes fault-variable)
             (replace-variable output-equations replacement-var
                              fault-variable)
             (replace-variable (delete-ccf output-equations
                                                 fault-variable)
                               replacement-var fault-variable))
          (if (check-list input-nodes fault-variable)
             (replace-variable ccf-list replacement-var fault-variable)
             (replace-variable (delete-ccf ccf-list fault-variable)
                               replacement-var fault-variable))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  combinational-testm
;;
;;  CALLING FUNCTION(S):  multiple-fault-com
;;
;;  CALLED FUNCTION(S):  get-fault-values, rep-function,
;;                       supplement, get-result
;;
;;  PURPOSE:  This function generates a test vector and sends it
;;            to the user for application.  It also forms the
;;            terms used later in comparison to the output
;;            equations.
;;
;;  VARIABLES:  function -- the function generated by
;;                          GET-VECTOR-FUNCTION
;;
;;              input-nodes -- circuit inputs
;;
```

```
;;              output-nodes -- circuit outputs
;;
;;              output-eqs -- circuit output equations
;;
;;              fault-variables -- the suspected fault nodes
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (combinational-testm function input-nodes output-nodes
                             output-eqs fault-variables)
   (newline)
   (let*( (fault-values (get-fault-values fault-variables))
          (new-function (rep-function function fault-values))
          (vector (car new-function))
          (vector2 (supplement vector input-nodes)))
  (if (null? new-function) nil
      (get-result vector2 output-nodes t))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  get-fault-values
;;
;;  CALLING FUNCTION(S):  combinational-testm, get-fault-values
;;
;;  CALLED FUNCTION(S):  get-fault-values
;;
;;  PURPOSE:  This function prompts the user for the values that
;;            he/she suspects the nodes to be stuck at.
;;
;;  VARIABLES:  fault-variables -- suspected fault nodes
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-fault-values fault-variables)
    (if (null? fault-variables) nil
        (begin
        (display "ENTER THE SUSPEC     .ULT VALUE FOR VARIABLE ")
        (display (car fault-variables))
        (display ":")
        (newline)
      (cons (read) (get-fault-values (cdr fault-variables))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
```

```
;;  FUNCTION:  rep-function
;;
;;  CALLING FUNCTION(S):  combinational-testm, rep-function
;;
;;  CALLED FUNCTION(S):  replace-with-zero, replace-with-one
;;
;;  PURPOSE:  This function replaces the numbers in the test
;;            vector function generated by GET-VECTOR-FUNCTION
;;            with the suspected stuck-at values specified by the
;;            user.
;;
;;  VARIABLES:  function -- the function generated by GET-VECTOR-
;;                          FUNCTION
;;
;;              fault-vals -- the suspected stuck-at values
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (rep-function function fault-vals)
    (if (null? fault-vals) function
        (let* ( (replaced-function
                  (cond ((equal? (car fault-vals) 0)
                         (replace-with-zero function
                                     (list (length fault-vals))))
                        (else
                         (replace-with-one function
                                    (list (length fault-vals)))))))
        (rep-function replaced-function (cdr fault-vals)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  faulty?
;;
;;  CALLING FUNCTION(S):  multiple-fault-com, faulty?
;;
;;  CALLED FUNCTION(S):  mult, complement, relatedm?, faulty?
;;
;;  PURPOSE:  This function compares the terms formed in
;;            GET-RESULT with the appropriate output equations.
;;
;;  VARIABLES:  test-results -- the terms formed by GET-RESULT
;;
;;              fault-vars -- the suspected fault nodes
;;
;;              inter-form -- the original system of individual
```

```
;;                            CCEs
;;
;;               output-eqs -- circuit output equations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (faulty? test-results fault-vars inter-form output-eqs)
    (if (or (null? output-eqs) (null? test-results)) nil
        (if (and (null? (mult (complement (car output-eqs))
                          (list (car test-results))))
                 (relatedm? fault-vars (car output-eqs) inter-form))

            '(M)
            (faulty? (cdr test-results) fault-vars inter-form
                  (cdr output-eqs)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  relatedm?
;;
;;  CALLING FUNCTION(S):  faulty?, relatedm?
;;
;;  CALLED FUNCTION(S):  related?, relatedm?
;;
;;  PURPOSE:  This function tests the accessibility of the fault
;;            variables to the output in question.
;;
;;  VARIABLES:  fault-vars -- suspected fault variables
;;
;;              output-eq -- a circuit output equation
;;
;;              inter-form -- the system of CCEs
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (relatedm? fault-vars output-eq inter-form)
   (if (null? fault-vars) nil
       (if (related? (car fault-vars) output-eq inter-form)
           t
           (relatedm? (cdr fault-vars) output-eq inter-form))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
```

```
;;  FUNCTION:  output-resultsm
;;
;;  CALLING FUNCTION(S):  multiple-fault-com
;;
;;  CALLED FUNCTION(S):  show-fault-lines
;;
;;  PURPOSE:  This function sends the diagnostic  results to the
;;            user.
;;
;;  VARIABLES:  results -- the results as generated by FAULTY?
;;
;;              fault-vars -- suspected fault nodes
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (output-resultsm results fault-vars)
     (newline)
     (newline)
     (display "THE FOLLOWING LINES: ")
     (show-fault-lines fault-vars)
     (if (member 'M results)
         (writeln "ARE STUCK AT THE SUSPECTED VALUES.")
         (writeln "ARE NOT STUCK AT THE SUSPECTED VALUES.")))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  show-fault-lines
;;
;;  CALLING FUNCTION(S):  output-resultsm, show-fault-lines
;;
;;  CALLED FUNCTION(S):  show-fault-lines
;;
;;  PURPOSE:  This function sends the suspected faulty lines to the
;;            screen.
;;
;;  VARIABLES:  fault-vars -- suspected fault nodes
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (show-fault-lines fault-vars)
    (if (null? fault-vars) nil
        (begin
        (display (car fault-vars))
        (display " ")
```

```
        (show-fault-lines (cdr fault-vars)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  againm?
;;
;;  CALLING FUNCTION(S):  multiple-fault-com
;;
;;  CALLED FUNCTION(S):  none
;;
;;  PURPOSE:  This function provides the user with the capability
;;            to test the same circuit for a different set of
;;            fault nodes.
;;
;;  VARIABLES:  none
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (againm?)
   (newline)
   (writeln "WOULD YOU LIKE TO RUN A MULTIPLE FAULT TEST ON ANOTHER
         SET OF NODES IN THE CIRCUIT?")
   (writeln "TYPE y<rtn> OR n<rtn>.")
   (let ( (answer (read)))
       (if (equal? answer 'y) t
           nil)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FILENAME:  test3s.s
;;
;;  FILES REQUIRED FOR CALLED FUNCTIONS:  test1s.s, test2s.s,
;;                                        test3.s, test2.s
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  multiple-fault-seq
;;
;;  CALLING  FUNCTION(S):  menu, multiple-fault-s
;;
;;  CALLED FUNCTION(S):  get-mfault-variables, cut-nodes,
;;                       get-state-info, get-vector-function,
;;                       sequential-testm, init-eqs, faulty?,
;;                       output-resultsm, multiple-fault-seq,
;;                       againm?
;;
;;  PURPOSE:  This function performs diagnosis of multiple stuck-at
;;            faults in sequential circuits.  The function operates
;;            much like MULTIPLE-FAULT-COM with the exception that
;;            the current state of the circuits current state
;;            variables must either be known or set by resetting
;;            the circuit.
;;
;;  VARIABLES:  intermediate-format -- system of circuit CCEs
;;
;;              internal-nodes -- circuit internal nodes
;;
;;              output-nodes -- circuit outputs
;;
;;              input-nodes -- circuit inputs
;;
;;              output-equations -- circuit output equations
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (multiple-fault-seq intermediate-format internal-nodes
                    output-nodes input-nodes output-equations)
    (newline)
    (let* ( (fault-variables (get-mfault-variables))
          (garb (display "PROCESSING.....")) )
```

```
        (new-format (cut-nodes intermediate-format input-nodes
                    output-equations fault-variables)))
    (if (null? new-format)
        (writeln "NO TEST POSSIBLE FOR THIS COMBINATION.")
     (let* (
        (state-info (get-state-info))
        (garb (display "PROCESSING....."))
        (vector-function (get-vector-function new-format
               internal-nodes output-nodes output-equations))
        (test-results (sequential-testm vector-function
          state-info input-nodes output-nodes output-equations
                              fault-variables))
        (out-eqs2 (init-eqs output-equations state-info
                                    (cadr test-results))))
   (if (or (null? test-results) (null? vector-function))
         (writeln "NO TEST POSSIBLE FOR THIS COMBINATION.")
      (let* ( (results (faulty? (car test-results)
               fault-variables intermediate-format out-eqs2)))
       (output-resultsm results fault-variables)))

      (if (againm?) (multiple-fault-seq intermediate-format
                                  internal-nodes
                        output-nodes input-nodes output-equations)
          (begin
          (read-line)
         (menu)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  sequential-testm
;;
;;  CALLING  FUNCTION(S):  multiple-fault-seq
;;
;;  CALLED FUNCTION(S):  get-fault-values, rep-function, get-terms
;;                       remove-mem-nodes, supplement, get-result
;;
;;  PURPOSE:  This function generates a test vector, prompts the
;;            user to apply the vector and forms the terms that
;;            are later compared to the output equations to
;;            determine if a fault has occurred.
;;
;;  VARIABLES:  function -- the function generated by GET-VECTOR-
;;                          FUNCTION
;;
```

```
;;              state-info -- the list of lists that contains the
;;                            current state variables of the
;;                            circuit along with their current
;;                            state and reset state values
;;
;;              input-nodes -- circuit inputs
;;
;;              output-nodes -- circuit outputs
;;
;;              output-eqs -- circuit output equations
;;
;;              fault-variables --  suspected fault nodes
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (sequential-testm function state-info input-nodes
                    output-nodes output-eqs fault-variables)
   (newline)
   (let*( (fault-values (get-fault-values fault-variables))
          (mem-nodes (car state-info))
          (new-function (rep-function function fault-values))
          (terms (get-terms new-function state-info))
          (vector (caar terms)))
  (if (null? vector) nil
      (let* ( (input-nodes2 (remove-mem-nodes mem-nodes
                                              input-nodes),
              (vector2 (supplement vector input-nodes2)))
             (cons (get-result vector2 output-nodes t)
                           (cdr terms) )))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FILENAME:  utils.s
;;
;;  NOTE:  The functions in this file have been borrowed from
;;         Kainec (18:262-266, 268, 272).
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;; GET NODES PROCEDURES ;;;;;;;;;;;;;;;;;;;;
;; (GET-INTERNAL-NODES prefix-list)
;;
;; Parameters:
;;    prefix-list - a list of the form:  ((eq _____ _______)
;;                                        (le _____ _______)
;;                                                :        )
;;
;; -- GET-INTERNAL-NODES works by getting first all of the nodes in
;;    the circuit and subtracting the input nodes and the output
;;    nodes.
;; -- GET-ALL-NODES returns all of the nodes in the circuit.
;; -- GET-INPUT-NODES returns the input nodes of the circuit.  GET-
;;    SUBLIST subtracts the input nodes from all of the nodes
;;    leaving the internal nodes and output nodes (NODES-LESS-
;;    INPUT-NODES).
;; -- GET-OUTPUT-NODES returns the output nodes of the circuit.
;;    GET-SUBLIST subtracts the output nodes from the NODES-LESS-
;;    INPUT-NODES leaving the INTERNAL-NODES.

(define (get-internal-nodes prefix-list)
   (let* ( (all-nodes (get-all-nodes prefix-list))

           (nodes-less-input-nodes (get-sublist all-nodes
                                    (get-input-nodes prefix-list)))

           (internal-nodes (get-sublist nodes-less-input-nodes
                                (get-output-nodes prefix-list))) )

      internal-nodes))




;; (GET-ALL-NODES prefix-list)
;;
```

```
;; Parameters:
;;    prefix-list - a list of the form:  ((eq _____ _______)
;;                                       (le _____ _______)
;;                                              :        )
;;
;; -- GET-ALL-NODES calls GET-NODES which returns a list of all of
;;    the nodes in the circuit.  Since GET-NODES does not remove
;;    duplicates of nodes, REMOVE-DUPLICATES is called to remove
;;    duplicates in the list.

(define (get-all-nodes prefix-list)
   (remove-duplicates (get-nodes prefix-list)))



;; (GET-INPUT-NODES prefix-list)
;;
;; Parameters:
;;    prefix-list -- a list of the form:  ((eq _____ _______)
;;                                        (le _____ _______)
;;                                               :        )
;;
;; -- Accepts a list in prefix form and returns a list of the nodes
;;    which are outputs for the given system of equations.  The
;;    equations represent a combinational circuit.
;; -- Input nodes are all nodes which occur only on the right hand
;;    side of the system of equations.
;; -- GET-INPUT-NODES takes each equation, and determines the
;;    symbols on the left hand side by calling GET-NODES-ON-LEFT.
;;    GET-NODES-ON-RIGHT returns the nodes on the right hand side.
;;    The nodes on the left are then subtracted from the nodes on
;;    the right (using GET-SUBLIST) yielding the input nodes.  A
;;    list of the input nodes is returned.

(define (get-input-nodes prefix-list)
   (let ( (nodes-on-left  (remove-duplicates
                            (get-nodes-on-left prefix-list)))

          (nodes-on-right (remove-duplicates
                            (get-nodes-on-right prefix-list))) )

      (get-sublist nodes-on-right nodes-on-left)))
```

```
;; (GET-OUTPUT-NODES prefix-list)
;;
;; Parameters:
;;    prefix-list -- a list of the form:  ((eq _____ _______)
;;                                        (le _____ _______)
;;                                                :       )
;;
;; -- Accepts a list in prefix form and returns a list of the nodes
;;    which are outputs for the given system of equations.  The
;;    equations represent a combinational circuit.
;; -- Output nodes are all nodes which occur only on the left hand
;;    side of the system of equations.
;; -- GET-OUTPUT-NODES takes the equations, and determines the
;;    symbols on the left hand side by calling GET-NODES-ON-LEFT.
;;    GET-NODES-ON-RIGHT determines the nodes on the right hand
;;    side.  The nodes on the right are then subtracted from the
;;    nodes on the left (using GET-SUBLIST) yielding the output
;;    nodes.  A list of the output nodes is returned.

(define (get-output-nodes prefix-list)
   (let ( (nodes-on-left  (remove-duplicates
                            (get-nodes-on-left prefix-list)))

          (nodes-on-right (remove-duplicates
                            (get-nodes-on-right prefix-list))) )

      (get-sublist nodes-on-left nodes-on-right)))



;; (GET-SUBLIST list-1 list-2)
;;
;; Parameters:
;;    list-1 -- an arbitrary list
;;    list-2 -- an arbitrary list
;;
;; -- GET-SUBLIST takes two lists and returns the items in list-1
;;    that are not members of list-2.
;; -- Duplicates are removed from the returned list.

(define (get-sublist list-1 list-2)
   (cond ( (null? list-1) '() )
```

```
        ; the first element of list-1 is an element of list-2
        ( (member (car list-1) list-2)
          (get-sublist (cdr list-1) list-2) )

        ; the first element of list-1 is not an element of list-2
        ( else
            (remove-duplicates
              (cons (car list-1) (get-sublist (cdr list-1) list-2
                                                        ))) )))



;; (GET-NODES-ON-RIGHT prefix-list)
;;
;; Parameters:
;;    prefix-list -- a list of the form:  ((eq _____ _______)
;;                                         (le _____ _______)
;;                                                 :        )
;;
;; -- GET-NODES-ON-RIGHT gets nodes on the right side of the
;;    equations.
;; -- GET-NODES is used to get the nodes from the right hand side
;;    of the prefix-list.  A list of nodes is returned.
;; -- Note: Duplicates are NOT removed from the list.

(define (get-nodes-on-right prefix-list)
   (if (null? prefix-list)
       '()
       (append (get-nodes (caddar prefix-list))
                          (get-nodes-on-right (cdr prefix-list)))))



;; (GET-NODES-ON-LEFT prefix-list)
;;
;; Parameters:
;;    prefix-list -- a list of the form:  ((eq _____ _______)
;;                                         (le _____ _______)
;;                                                 :        )
;;
;; -- GET-NODES-ON-LEFT gets nodes on the left side of the
;;    equation.
```

```
;; -- GET-NODES is used to get the nodes from the left hand side of
;;    the prefix-list.  A list of nodes is returned.
;; -- Note: Duplicates are NOT removed from the list.

(define (get-nodes-on-left prefix-list)
   (if (null? prefix-list)
       '()
       (append (get-nodes (cadar prefix-list))
                          (get-nodes-on-left (cdr prefix-list)))))



;; (GET-NODES lst)
;;
;; Parameters:
;;    lst -- a list in prefix form, i.e., (+ (* A B) (NOT C))
;;
;; -- GET-NODES accepts a list in prefix form and returns a list of
;;    all of the symbols in the list which are atoms, but are not
;;    token symbols.
;; -- TOKEN-SYMBOL? is used to determine if an atom is a token
;;    symbol.
;; -- GET-NODES extracts atoms which are included in nested lists.

(define (get-nodes lst)
   (cond ( (null? lst) '())

         ; if the list is atomic and not a token symbol,
         ; then return it in a list
       ( (and (atom? lst)
              (not (token-symbol? lst)))
         (list lst) )

         ; if the list is atomic and a token symbol, return nil
       ( (and (atom? lst)
              (token-symbol? lst))
         '() )

         ; otherwise, break apart the list
       ( else
           (let ((first-symbol (car lst))
                 (rest-of-list (cdr lst)))

                ; if the first symbol is an atom, determine
                ; the type of symbol--if it is a token
```

```
                    ; symbol, ignore it; if it is not, then add
                    ; it to returned list
                    ; -- make a recursive call either way
               (cond ( (atom? first-symbol)
                     (if (not (token-symbol? first-symbol))
                      (cons first-symbol (get-nodes rest-of-list))
                       (get-nodes rest-of-list)) )

                     ; otherwise, make recursive calls
                     ( else
                        append (get-nodes first-symbol)
                               (get-nodes rest-of-list)) ))) )))
```

```
;; (REMOVE-DUPLICATES lst)
;;
;; Parameters:
;;    lst -- an arbitrary list
;;
;; -- REMOVE-DUPLICATES removes duplicates from the first level
;;    of the input list.

(define (remove-duplicates lst)
   (cond ( (null? lst)
           '() )

         ( (member (car lst) (cdr lst))
           (remove-duplicates (cdr lst)) )

         ( else
           (cons (car lst) (remove-duplicates (cdr lst))) )))
```

```
;; (ON-RIGHT-SIDE? node right-side)
;;
;; Parameters:
;;    node - a node in the circuit
;;    right-side - the right side of an equation
;;
```

```
;; -- ON-RIGHT-SIDE? is a predicate procedure called by REPLACE-
;;    NODE to determine if a given node is on the right-side of an
;;    equation in prefix-form.
;; -- ON-RIGHT-SIDE? is called recursively until the NODE can be
;;    tested for equality against every symbol on the RIGHT-SIDE.

(define (on-right-side? node right-side)
          ; the right-side is nil
   (cond ( (null? right-side) '() )

          ; the right-side is a symbol
         ( (symbol? right-side)
           (if (eq? node right-side)
               t
               '()) )

   ; the head of the right-side is a list
         ( (list? (car right-side))
           (or (on-right-side? node (car right-side))
               (on-right-side? node (cdr right-side))) )

           ; the head of the right-side is a symbol
         ( else
             (if (eq? node (car right-side))
                 t
                 (on-right-side? node (cdr right-side))) )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FILENAME:  utils2.s
;;
;;  FILES REQUIRED FOR CALLED FUNCTIONS:  boolean.s
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  replace-with-zero
;;
;;  CALLING FUNCTION(S):  combinational-test, sequential-test,
;;                        combinational-testb, sequential-testb,
;;                        replace-with-zero
;;
;;  CALLED FUNCTION(S):  rep-complement, replace-with-zero, econ
;;
;;  PURPOSE:  This function replaces a term (which in our case is
;;            usually a single variable) in a function with zero.
;;            First the complemented form of the term is deleted
;;            using REP-COMPLEMENT.  Deletion is the same as
;;            multiplying by one which is what replacement does
;;            with complemented terms.  The remaining terms are
;;            those that are uncomplemented and they can be
;;            replaced by zero using the conjunctive eliminant.
;;
;;  VARIABLES:  f -- function to replace in
;;
;;              term -- term or variable to replace
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (replace-with-zero f term)
  (cond ( (null? term) f)
        ( else
         (econ (replace-with-zero (rep-complement f (car term))
                 (cdr term) )
               term))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  rep-complement
;;
```

```
;;  CALLING FUNCTION(S):  replace-with-zero, rep-complement
;;
;;  CALLED FUNCTION(S):  replace-term1, rep-complement
;;
;;  PURPOSE:  This function replaces the variable in each term of
;;            a function with zero by deleting the complement form
;;            of the variable from the term (same as multiplying by
;;            one).
;;
;;  VARIABLES:  f -- function to replace in
;;
;;              x -- term to replace
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (rep-complement f x)
  (cond ( (null? f) nil)
        ( else
          (cons (replace-term1 (car f) x)
                (rep-complement (cdr f) x) ))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  replace-term1
;;
;;  CALLING FUNCTION(S):  rep-complement, replace-term1
;;
;;  CALLED FUNCTION(S):  replace-term1
;;
;;  PURPOSE:  Used by REP-COMPLEMENT  to replace term in one term
;;            at a time in the function.
;;
;;  VARIABLES:  term -- term in the function to operate on
;;
;;              x -- term to replace
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (replace-term1 term x)
  (cond ( (null? term) nil)
        ( (equal? (car term) (bar x))
                     (cdr term))
        ( else
          (cons (car term)
```

```
                (replace-term1 (cdr term) x) ))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  relpace-with-one
;;
;;  CALLING FUNCTION(S):  combinational-test, sequential-test,
;;                        combinational-testb, sequential-testb,
;;                        replace-with-one
;;
;;  CALLED FUNCTION(S):  rep-uncomplement, replace-with-one, econ
;;
;;  PURPOSE:  This function replaces a term (which in our case is
;;            usually a single variable) in a function with one.
;;            First the uncomplemented form of the term is deleted
;;            using REP-UNCOMPLEMENT.  Deletion is the same as
;;            multiplying by one which is what replacement (with
;;            one) does with uncomplemented terms  The remaining
;;            terms are those that are complemented and they can be
;;            replaced by zero (since they are complemented) using
;;            the conjunctive eliminant.
;;
;;
;;  VARIABLES:  f -- function to replace in
;;
;;              x -- term to replace
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (replace-with-one f term)
  (cond ( (null? term) f)
        ( else
          (econ (replace-with-one (rep-uncomplement f (car term))
                  (cdr term) )
                term))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  rep-uncomplement
;;
;;  CALLING FUNCTION(S):  replace-with-one, rep-uncomplement,
;;
;;  CALLED FUNCTION(S):  replace-term2, rep-uncomplement
```

```
;;
;;  PURPOSE:  This function replaces the variable in each term of
;;            a function with one by deleting the uncomplemented
;;            form of the variable from the term (same as
;;            multiplying by one).
;;
;;  VARIABLES:  f -- function to replace in
;;
;;              x -- term to replace
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (rep-uncomplement f x)
  (cond ( (null? f) nil)
        ( else
          (cons (replace-term2 (car f) x)
                (rep-uncomplement (cdr f) x) ))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  replace-term2
;;
;;  CALLING FUNCTION(S):  rep-uncomplement
;;
;;  CALLED FUNCTION(S):  replace-term2
;;
;;  PURPOSE:  Used by REP-UNCOMPLEMENT  to replace term in one term
;;            at a time in the function.
;;
;;  VARIABLES:  term -- term in the function to operate on
;;
;;              x -- term to replace
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (replace-term2 term x)
  (cond ( (null? term) nil)
        ( (equal? (car term)  x)
          (cdr term) )
        ( else
          (cons (car term)
                (replace-term2 (cdr term) x) ))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;
;;  FUNCTION:  get-output-equations
;;
;;  CALLING FUNCTION(S):  menu, get-output-equations
;;
;;  CALLED FUNCTION(S):  flatten, any-matches, get-output-
;;                       equations,
;;
;;  PURPOSE:  This function extracts the output equations from
;;            the two systems of equations from the input file.
;;            It does so after all equations have been put into
;;            prefix form.  The output equations are recognized
;;            by the fact that they contain no internal variables.
;;
;;  VARIABLES:  intermediate-format -- the prefix form of all
;;              equations in the input file
;;
;;              internal-nodes -- circuit internal nodes
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-output-equations intermediate-format internal-nodes)
     (if (null? intermediate-format) nil
     (let* ( (candidate (car intermediate-format))
             (new-format (flatten candidate))
             (matches (any-matches internal-nodes new-format)))
         (if (null? matches)
             (append (list candidate) (get-output-equations
                                        (cdr intermediate-format)
                                        internal-nodes))
             (get-output-equations (cdr intermediate-format)
                                   internal-nodes)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  any-matches
;;
;;  CALLING FUNCTION(S):  get-output-equations, any-matches
;;
;;  CALLED FUNCTION(S):  any-matches
;;
;;  PURPOSE:  This function checks a particular equation to see
;;            if it includes any internal variables; if not then
;;            it is returned as an output equation.
;;
```

```
;;  VARIABLES:  items -- the variable list that is searched for,
;;                       internal variables in our case
;;
;;              candidate-list -- the flattened equation that is
;;                                searched
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (any-matches items candidate-list)
     (cond ((null? items) nil)
           ((member (car items) candidate-list) t)
           (else
             (any-matches (cdr items) candidate-list))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  FUNCTION:  remove-output-equations
;;
;;  CALLING FUNCTION(S):  menu, remove-output-equations
;;
;;  CALLED FUNCTION(S):  remove-output-equations, remove
;;
;;  PURPOSE:  After the output equations are obtained there is n
;;            need to keep them with the circuit description s
;;            they are removed.
;;
;;  VARIABLES:  intermediate-format -- description to remove them
;;                                     from
;;
;;               output-equations -- items to remove
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (remove-output-equations intermediate-format
                                 output-equations)
     (if (null? output-equations) intermediate-format
     (remove-output-equations
           (remove (car output-equations) intermediate-format)
           (cdr output-equations))))
```

Appendix C Modified Kainec Diagnostic System Code

This appendix includes the [illegible] described in Chapter 6. The modified [illegible] the changes to account for the [illegible] been modified [illegible]

```
(herald testerm
   (env t scheme-syntax scheme-1 scheme-2 boolean interp interp-a))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          Filename: tester.s
;;
;;  This module is the portion of the diagnostic system which provides the
;;  mechanisms for conducting the input-output experiment.  It takes the
;;  single Boolean equation produced by the Equation Generation Module of
;;  the system, lists of the inputs, outputs, and checkpoints, and produces
;;  test vectors for the given circuit.  Once the result of the test is
;;  known, the result is fed back to the system which used it to create
;;  new information about the circuit.  Tests are conducted until it has
;;  been determined that further information cannot be gained from
;;  input-output tests.  At this point, an equation exists which holds all
;;  of the information known about the circuit, including the state of
;;  faults and the actual circuit function.  This equation is returned in
;;  a list with the number of tests that were conducted.
;;
;;  Requires the files: boolean.fsl, interp.fsl, interp-a.fsl
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; (TESTER input)
;;
;; Parameters:
;;    input - The output list from the Equation Generation Module of the
;;            diagnostic system.  It consists of sublists which are the
;;            system equation generated by GENERATE-EQUATION, the INPUTS,
;;            OUTPUTS, and the circuit CHECKPOINTS.
;;
;; -- TESTER decomposes the INPUT into its composite sublists and takes the
;;   Blake Canonical Form of the equation to form a new equation (NEW-EQN).
;; -- A TEST-INPUT is generated from this NEW-EQN by MAKE-TEST-INPUT.
;;    Then TESTER-1 is called to begin the iterative testing process.

(define (tester input)
   (let* ( (equation    (car input))
           (inputs      (cadr input))
           (outputs     (caddr input))
           (checkpoints (cadddr input))
           (new-eqn     (bcf equation))
       (test-input  (make-test-input new-eqn inputs checkpoints outputs)))

      (tester-1 new-eqn test-input inputs checkpoints outputs 0) ))
```

```
;; (TESTER-1 equation test-inp inputs checkpoints outputs test-no)
;;
;; Parameters:
;;    equation - The system equation generated by GENERATE-EQUATION.
;;    test-inp - The first test input generated by the diagnostic system.
;;    inputs - A list of the inputs of the circuit.
;;    checkpoints - A list of the checkpoints in the circuit.
;;    outputs - A list of the outputs of the circuit.
;;    test-no - The current test number.  Initially, this is 1
;;
;; -- TESTER-1 is a helping procedure for TESTER.  However, it is the
;;    module that supervises the input-output experiment.
;;  If the TEST-INP is null, then another test could not be generated from
;;    the system EQUATION.  At this time, a message is output and the
;;    system EQUATION is returned in a list along with the TEST-NO which
;;    indicates the number of tests that occurred.
;; -- The TEST-INP is generated prior to TESTER-1 being called. If it is
;;    not null, then a test was generated.  PRINT-SUGGESTED-INPUT outputs
;;    the list representing the test vector in a user-readable form.
;; -- The user is then prompted for the RESULT of the test.  The RESULT is
;;    combined with the TEST-INP by MAKE-NEW-INFO to make NEW-INFOrmation
;;    which can be added to the EQUATION.  The combination of the EQUATION
;;    and NEW-INFO forms a NEW-EQN.  DCF is a procedure used to generate
;; a "Diagnostic Canonical Form" which is a form of the equation necessary
;;    to generate new test vector inputs.

(define (tester-1 equation test-inp inputs checkpoints outputs test-no)

   (cond ( (null? test-inp)
           (writeln "New information cannot be obtained.")
           (newline)
           (cons test-no equation) )
         ( else
           ; print out the suggested input in a user-readable format
           (print-suggested-input test-inp)
           (newline)

           ; for the first test, give the user instructions
           (if (equal? 0 test-no)
              (writeln "If the output was 0, type 0 and <rtn>, else type 1
                        and <rtn>.")
               '())

           ; prompt for the result
```

```
;(display "Enter the Resulting Output (0 or 1) --> " ****VM*****

       ; read in the result, generate new information from the test input
          ; and the result, and make a NEW-EQN which contains the old
          ; EQUATION plus new information derived from the test
         (let* (            ;(result   (read-line)) ****VM*****
;(new-info (make-new-info test-inp result outputs)) ****VM*****

        ; ****VM***** GET-RESULTS obtains the resulting output values
        ; ****VM***** from the user following vector application.
              (new-inform (get-results  test-inp outputs)) ****VM*****
          (new-eqn  (dcf (append new-inform equation) inputs outputs

             ; make a recursive call using the NEW-EQN, generating a new
             ; TEST-INPUT on the fly, increment the TEST-NO
             (tester-1 new-eqn
                   (make-test-input new-eqn inputs checkpoints outputs)
                       inputs
                       checkpoints
                       outputs
                       (1+ test-no))) )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;                                  ****VM*****
;;
;; GET-RESULTS
;;
;; This function prompts the user to input the resulting outputs
;; following application of a particular test vector represented by
;; the variable TEST-INP.  A prompt is sent for each output
;; contained in the list of circuit OUTPUTS.  As each result is
;; read MAKE-NEW-INFO is called to form the term that combines
;; the vector minterm with the resulting output.  The resulting
;; terms are combined in sum-of-products form and sent to the
;; calling routine TESTER-1.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-results test-inp outputs)
    (if (null? outputs) nil
        (begin
        (display "Enter the Result from Output ")
        (display (car outputs))
        (display " --> ")
```

```
        (let* ( (result (read))
                (new-info (make-new-info test-inp result
                                         (list (car outputs)))))
          (cons new-info (get-results test-inp (cdr outputs)))))))


;; (PRINT-SUGGESTED-INPUT lst)
;;
;; Parameters:
;;    lst - A list of the form ((A--) B-- C--), where the subelements
;;          are literals representing the inputs to the circuit.
;;
;; -- PRINT-SUGGESTED-INPUT prints out a message and then calls
;;  PRINT-SUGGESTED-INPUT-1 which outputs each of the inputs individually.

(define (print-suggested-input lst)
   (writeln "The Suggested Input is: ")
   (newline)
   (print-suggested-input-1 lst) )


;; (PRINT-SUGGESTED-INPUT-1 lst)
;;
;; Parameters:
;;    lst - A list of the form ((A--) B-- C--), where the subelements
;;          are literals representing the inputs to the circuit.
;;
;; -- PRINT-SUGGESTED-INPUT-1 prints out the suggested input in a
;;    user-readable format.  The input LST is of the form ((A--) B-- C--),
;;    where each symbol is an input to the circuit.  If a literal is
;;    enclosed in a sublist, then it should be set to 0. Otherwise, if it
;;    exists in the top-level of the list, then it should be set to 1.
;; -- In each call to PRINT-SUGGESTED-INPUT-1, one of the suggested inputs
;;  is output.  Recursive calls are made until all of the suggested inputs
;;    have been output.
;; -- CONVERT-NODE-BACK is called to eliminate the suffix from each symbol.
;;  The symbol is then of the form that was originally input to the system
;;    by the user.

(define (print-suggested-input-1 lst)
   (if (null? lst)
       '()
       (let ( (first-term (car lst))
              (rest       (cdr lst)) )
```

```
          ; if the first-term is a symbol, it should be set to 1
          ; otherwise, if in a sublist, it should be set to 0
          (if (symbol? first-term)
              (begin
                 (writeln "    " (convert-node-back first-term) " = 1")
                 (print-suggested-input-1 rest))

              (begin
           (writeln "    " (convert-node-back (car first-term)) " = 0")
                 (print-suggested-input-1 rest)) ))))


;; (MAKE-NEW-INFO test-input result outputs)
;;
;; Parameters:
;;    test-input - A list of the form ((A--) B-- C--) which was the test
;;                 vector generated by TEST-INPUT.
;;    result - A string representing the result of the test; either
;;             "1" or "0".
;;    outputs - A list of the outputs of the circuit.
;;
;; -- MAKE-NEW-INFO combines the TEST-INPUT with the OUTPUTS to make new
;;    information about the state of the circuit.
;; -- The new information is based on the mathematical model that:
;;        TEST-INPUT ==> OUTPUT
;;    Translated into Boolean Algebra, this would be modeled
;;        TEST-INPUT s OUTPUT
;;    This is then converted to the form
;;        TEST-INPUT * OUTPUT' = 0
;;    Lists are built appropriately to implement this last equation.
;;This list is then added to the old equation to form an updated equation.
;; -- As currently implemented, it is assumed that OUTPUTS is a list of
;;    a single element representing a single output of the circuit.

; ****VM**** perform function on one output at a time
(define (make-new-info test-input result output) ; ****VM****

   (newline)
   (newline)
   (writeln "Processing....")
   (newline)
   (cond ( (equal? result 1)   ; ****VM****
           (append test-input (list output)) ) ; ****VM****
         ( (equal? result 0)   ; ****VM****
           (append test-input output) )))  ; ****VM****
```

```
;; (DCF equation inputs outputs)
;;
;; Parameters:
;;    equation - The new equation formed by adding the new information
;;        generated by an input-output test to the old system equation.
;;              This equation is in f=0 form.
;;    inputs - A list of the inputs of the circuit.
;;    outputs - A list of the outputs of the circuit.
;;
;; --DCF generates the "Diagnostic Canonical Form" of the system equation.
;; --First, the Blake Canonical Form (BCF) is taken of the input EQUATION.
;;   This generates all of the possible consensus terms from the EQUATION.
;; --The aim of the Diagnostic Canonical Form is to get the equation into
;;   the following form:
;;
;;     A(x,y) z' + B(x,y) z + G(y)  = 0  where x represents the circuit
;;                                       inputs, z the circuit outputs,
;;                                       and y the checkpoint variables
;;
;; -- However, after getting the Blake Canonical Form of this equation,
;;    it may be in the form:
;;
;;          A(x,y) z' + B(x,y) z + H(x,y)  = 0
;;
;; -- The G(y) term is made up of the elements of H(x,y) which have had
;;    the input variables stripped, or SIFTed, off.  This can be done
;;    because the input variables are not constrained due to independence.
;;    Thus, the checkpoint variables they are combined with to form a term
;;    must be identically equal to 0.
;; -- SIFT forms the terms in G(y) which are added to the input EQUATION.
;;    UNABSORB is then called to execute absorptions caused by these new
;;    terms.

(define (dcf equation inputs outputs)
  (unabsorb (sift (bcf equation) inputs outputs)) )


;; (SIFT equation inputs outputs)
;;
;; Parameters:
;;    equation - The new equation formed by adding the new information
;;           generated by an input-output test to the old system equation.
;;               This equation is in f=0 form.
```

```
;;    inputs - A list of the inputs of the circuit.
;;    outputs - A list of the outputs of the circuit.
;;
;; -- SIFT is a helping procedure for the DCF procedure.  It generates the
;;    G(y) terms from the H(x,y) terms in the equations listed above.
;; -- COMMON-ARGS? is used to determine whether any OUTPUTS are in a given
;;    term of the EQUATION.  If they are, then this term is simply ignored.
;;  If they are not, then the INPUTS are disjunctively eliminated from the
;;    term to yield a term that is composed only of checkpoint variables.
;; -- SIFT calls itself recursively until all terms of the input EQUATION
;;    have been checked and modified if appropriate.

(define (sift equation inputs outputs)
  (cond ( (null? equation)
          '() )
        ( (not (common-args? outputs (car equation)))
          (cons (car (edis (list (car equation)) inputs))
                (sift (cdr equation) inputs outputs) ))
        ( else
          (cons (car equation)
                (sift (cdr equation) inputs outputs) ))))


;; (MAKE-TEST-INPUT equation inputs checkpoints outputs)
;;
;; Parameters:
;;    equation - The new equation formed by adding the new information
;;           generated by an input-output test to the old system equation.
;;               This equation is in f=0 form.
;;    inputs - A list of the inputs of the circuit.
;;    outputs - A list of the outputs of the circuit.
;;    checkpoints - A list of the checkpoints of the circuit.
;;
;; -- MAKE-TEST-INPUT uses EQUATION, the CHECKPOINTS, and the OUTPUTS,
;;    to generate a test vector input.
;; --MAKE-INPUT-EQUATION is passed the EQUATION, CHECKPOINTS, and OUTPUTS.
;;    Boolean elimination is used to remove the CHECKPOINTS and OUTPUTS
;;    from the EQUATION to get an INPUT-EQUATION in f=0 format.
;;    Solving this equation yields an effective input that will yield
;;    new information about the circuit.
;; -- Because it is difficult to solve an equation in f=0 format, i.e.
;;    all terms must be set to 0, the INPUT-EQUATION is complemented
;;    to get the f=1 form.  Then, only a single TERM need be set to 1 to
;;    solve the equation.  DISPLAY-CIRCUIT-FUNCTION-1 is called to
;;    display the f=0 equation that must be solved.
```

```
;; -- When the INPUT-EQUATION becomes equal to 1, or in the representation
;;    used in this system, '(()), then further effective inputs cannot
;;    be generated.  At this time '() is returned.
;; -- All of the INPUTS may not exist as literals of TERM.  COMBINE is
;;    used to insert the INPUTS that are not literals of TERM into term.
;;    Due to the nature of Boolean Algebra, these missing literal can
;;    be arbitrarily set to 0 or 1.  In this implementation, the missing
;;    literals are set to 1.  SORT-TERM is called to generate a test
;;    vector in sorted order.

(define (make-test-input equation inputs checkpoints outputs)

;  ****VM**** MAKE-INPUT-EQUATION no longer needs circuit OUTPUTS
;             because they will no longer be eliminated
  (let* ( (input-equation (make-input-equation equation checkpoints))

;  ****VM**** GET-VECTOR obtains an optimal minterm (test vector)
;             from a set of effective test vectors.  Though all
;             possible input combinations are processed, the
;             function stops short of picking a non-effective
;             vector.
          (term (get-vector input-equation inputs outputs)) )

     ; force 17 newlines to the screen to reduce clutter
;     (do ( (i 1 (1+ i)) )
;         ( (> i 17) '())
;         (newline) )

     (newline)
     ; if the input function was 0, then any input is an effective input
     ; i.e. there are no constraints on input variables that are required
     ; to yield new information about the circuit
     (if (null? input-equation)
         (writeln "The Input Equation is: 0 = 0")
         (begin
           (display "The Input Equation is: ")
           (display-circuit-function-1 input-equation)
           (writeln "= 0")))

     ; if the input was 1 i.e. '(()), then return nil to signify that a
     ; new input function cannot be generated.  Otherwise, take the term,
  ; fill in the missing literals, sort is alphabetical order, and return.

; ****VM**** SORT-TERM will return null if term is null to indicate
;            that no further info can be gained.
```

```
        (sort-term term)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;                                    ****VM****
;; GET-VECTOR
;;
;; This function generates a minterm (test vector) by choosing the
;; optimal vector from a set of vectors.  The input FUNCTION is
;; complemented to attain the f = 1 sum-of-products form.  The
;; result is EXPANDed with respect to the INPUT-NODES to enumerate
;; all of the possible input combinations.  The resulting terms
;; combine each input combination with a function of the output
;; nodes.  These output functions may include anywhere from none or
;; all of the outputs.  Those terms with the least number of
;; outputs in their output functions represent optimal vectors
;; because the input combinations associated with them stand to
;; gain the most information when applied.  EXPAND1 associates a
;; number with each input combination based on the expansion of the
;; output functions with respect to the OUTPUT-NODES.  The
;; output functions with the least number of outputs will result in
;; the largest number of terms when expanded.  PICK-LARGE chooses
;; the largest number generated.  PICK-ONE takes one of the input
;; combinations associated with the largest number.  If the largest
;; number generated is 1 then all possible info has been gained and
;; this function returns nil.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-vector function input-nodes output-nodes)
     (let* ( (function2 (complement function))
             (function3 (expand function2 input-nodes))
             (candidates (expand1 function3 output-nodes))
             (large-num (pick-large candidates 0))
            (choice (pick-one candidates large-num)))
          (if (equal? large-num 1)
              nil
              choice)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;                                    ****VM****
;;
;; EXPAND
;;
```

```
;; This is a special boolean expansion wrt the input variables.
;; It calls DIVIDE1 to get the functions required to complete
;; expansion (functions w/ the var set equal to one and zero).
;; DIVIDE2 actually performs the calculation, while DIVIDE1 insures
;; that the output function is enclosed in parentheses when after
;; the last input variable is used for expansion.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (expand func arg-list)
  (if (null? arg-list) (list func)
      (let* ( (num (length arg-list))
              (arg (car arg-list))
              (narg (bar arg))
              (f0 (divide1 func narg num))
              (f1 (divide1 func arg num))
              (m0 (expand f0 (cdr arg-list)))
              (m1 (expand f1 (cdr arg-list))))
            (append (prefix narg m0)
                    (prefix arg m1)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;                                ****VM****
;;
;; DIVIDE1
;;
;; This function calls DIVIDE2 to get the functions required to
;; complete expansion.  If we're on the last input variable then
;; a list is formed a with the next expansion which should be a
;; list of arguments containing just output vars.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (divide1 f x num)
    (let* ( (result (divide2 f x)))
        (if (equal? num 1) (list result)
            result)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;                                ****VM****
;;
;; DIVIDE2
```

```
;;
;; This function generates the functions that set a given variable
;; equal to zero and one, respectively.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (divide2 f x)
  (cond ((null? f) nil)
        ((member (bar x) (car f))
         (divide2 (cdr f) x) )
        (else
         (cons (remove x (car f))
               (divide2 (cdr f) x) ))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;                                  ****VM****
;;
;; EXPAND1
;;
;; This function expands the output functions which are inside
;; parentheses, (lp), wrt output nodes and returns the number of
;; expansions along side the input combination associated with the
;; output function.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (expand1 func arg-list)
    (if (null? func) nil
      (let* ( (lp (last-pair (car func)))
              (expansion (remove-duplicates (expand2 (car lp)
                                                     arg-list)))
              (minterm (remove (car lp) (car func)))
              (num (list (length expansion))))
          (append (list (append num minterm))
                  (expand1 (cdr func) arg-list)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;                                  ****VM****
;;
;; EXPAND2
;;
;; This function performs a normal boolean expansion on a function
;; wrt a specified list of variables using DIVIDE2 to generate the
```

```
;; functions needed for expansion.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (expand2 func arg-list)
  (if (null? arg-list) func
      (let* ( (arg (car arg-list))
              (narg (bar arg))
              (f0 (divide2 func narg))
              (f1 (divide2 func arg))
              (m0 (expand2 f0 (cdr arg-list)))
              (m1 (expand2 f1 (cdr arg-list))))
             (append (prefix narg m0)
                     (prefix arg m1)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;                                ****VM****
;; PICK-LARGE
;;
;; This fucntion picks the largest number generated by EXPAND1 to
;; aid in finding an optimal vector.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (pick-large term-list start-num)
    (if (null? term-list)  start-num
      (let ((new-num (caar term-list)))
        (if (> new-num start-num)
            (pick-large (cdr term-list) new-num)
            (pick-large (cdr term-list) start-num)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;                                ****VM****
;; PICK-ONE
;;
;; This functions uses the largest number to choose one of the
;; minterms that represents an optimal test vector.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (pick-one terms num)
```

```
    (let ( (candidate (caar terms)))
        (if (equal? candidate num)
            (remove num (car terms))
            (pick-one (cdr terms) num))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;                                  ****VM****
;;
;; REMOVE-DUPLICATES
;;
;; This is a helping function that removes duplicate items from a
;; list.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (remove-duplicates lst)
   (cond ( (null? lst)
           '() )

         ( (member (car lst) (cdr lst))
           (remove-duplicates (cdr lst)) )

         ( else
           (cons (car lst) (remove-duplicates (cdr lst))) )))


;; (MAKE-INPUT-EQUATION equation checkpoints outputs)
;;
;; Parameters:
;;    equation - The new equation formed by adding the new information
;;           generated by an input-output test to the old system equation.
;;               This equation is in f=0 form.
;;    checkpoints - A list of the checkpoints of the circuit.
;;    outputs - A list of the outputs of the circuit.
;;
;; -- MAKE-INPUT-EQUATION accepts an equation of the form:
;;
;;         P(x,y,z) = 0 where x are the inputs of the circuit,
;;                           y are the checkpoints of the circuit,
;;                           and z the outputs of the circuit.
;;
;; -- Conjunctive ELIMINATion is used to remove the checkpoints from
;;    the equation.  This leaves an equation of the following form:
;;
```

```
;;         A(x) z' + B(x) z = 0
;;
;; -- Disjunctive elimination, performed by EDIS, yields an equation of the
;;    form:
;;
;;         A(x) + B(x) = 0
;;
;; -- The Blake Canonical Form of this equation, generated by BCF, is
;;    then formed and returned.

;; ****VM**** this function has been changed to delete the
;;            elimination of output variables

(define (make-input-equation equation checkpoints)
  (bcf (eliminate equation checkpoints)) )




;; (COMBINE term inputs)
;;
;; Parameters:
;;   term - A term from the f=1 form of the INPUT-EQUATION.
;;   inputs - A list of the inputs of the circuit.
;;
;; -- All of the INPUTS may not exist as literals of TERM.  COMBINE is
;;    used to insert the INPUTS that are not literals of TERM into term.
;;    Due to the nature of Boolean Algebra, these missing literal can
;;    be arbitrarily set to 0 or 1.  In this implementation, the missing
;;    literals are set to 1.
;; -- For example, if the inputs were (A B C), and term
;;    were A B = 1, then the equations A B C = 1 or A B C' = 1 would both
;;    satisfy the constraints imposed by TERM.  Thus, C can be arbitrarily
;;    chosen.  COMBINE sets C to 1.

(define (combine term inputs)
  (cond ( (null? inputs)
          '() )
        ( (member (bar (car inputs)) term)
          (cons (bar (car inputs))
                (combine term (cdr inputs)) ))
        ( else
          (cons (car inputs)
                (combine term (cdr inputs)) ))))
```

```
;;; T specific call required for file compilation:

(herald interpm
   (env t scheme-syntax scheme-1 scheme-2 boolean
        eqn-gen eqn-gena tokenize interp-a) )

;;; Modification -- replaced #T with T to denote true
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        Filename: interp.s
;;
;;  This module provides the facilities to interpret the output equation
;;  from the TESTER.S module of the system.  The facilities provided
;;  include a procedure to compare the designed circuit to the function
;;  that the circuit is actually performing, an interpretation of the
;;  faults in the circuit, and a summary of system metrics.
;;
;;  NOTE:  This implementation is based on the assumption of a single
;;         output circuit.  Procedures must be revised to accomodate
;;         multiple output circuit diagnosis.
;;
;;  Requires the files: boolean.fsl, eqn-gen.fsl, eqn-gena.fsl,
;;                      tokenize.fsl, interp-a.fsl
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; (INTERPRET intermediate-format phi tester-output)
;;
;; Parameters:
;;    intermediate-format - The data structure, a list in prefix-form that
;;                          was returned by procedure RUN-INPUT-_ ULE.
;;                          This list is used to determine the appropriate
;;                        gate a given fanout node is associated with when
;;                          printing out results for each node.
;;    phi - The data returned by GENERATE-EQUATION.  The information
;;        provided by this list includes the circuit INPUTS, OUTPUTS, and
;;          CHECKPOINTS.
;;    tester-output - The data returned by TESTER.  This includes the
;;                    NO-OF-TESTS that were conducted as well as the
;;                    FINAL-EQUATION generated by TESTER.  This equation
;;                  is solved to yield the circuit FUNCTION as well as the
;;                    FAULT-CLASSES in the circuit.
;;
;; -- INTERPRET takes the output from all of the other major modules and
;;    interprets the information to obtain the results of the diagnostic
;;    test.
```

```
;;--DISPLAY-FUNCTIONS is called to determine the function that the circuit
;;    performed based on the diagnostic test, as opposed to the function
;;    that it was designed to perform.  An equivalency check is made to
;;    compare the actual to the designed function.
;; -- INTERPRET-FAULTS is called to derive the faults in the circuit,
;;    both those that can be positively determined as well as cases of
;; faults that may have occurred, but cannot be determined with certainty.
;; -- DISPLAY-SYSTEM-METRICS is used to make a quick determination of the
;;    a performance metrics of the diagnostic system.
;; -- Finally, the user is asked whether he would like to diagnose another
;;    circuit.  The REPLY, in the form of #T or '() is returned by
;;    INTERPRET to the calling procedure where it is used to determine
;;    whether to reexecute the calling module, or return to the main menu
;;    of the diagnostic system.


(define (interpret intermediate-format phi tester-output)
           ; break down information from input parameters
    (let* ( (inputs         (cadr phi))
            (outputs        (caddr phi))
            (checkpoints    (cadddr phi))
            (no-of-tests    (car tester-output))
            (final-equation (cdr tester-output))

;; ****VM****  this function includes all actual output functions
;;             as determined by the test experiments
            (int-a-function (eliminate final-equation checkpoints))

;; ****VM****  SOLVE-FCNS generates a list of all output functions
;;             for a multiple output circuit
            (a-functions     (solve-fcns (bcf int-a-function)
                                         outputs outputs))

;; ****VM**** this function includes all designed functions taken
;;            from the original circuit description
            (int-d-function (eliminate (simplify
                                 (make-sop intermediate-format))
                          (get-internal-nodes intermediate-format)
                                  ))

;; ****VM**** check the equivalence of the actual and design
;;            functions
          (equivalence-result (xor int-a-function int-d-function))

;; ****VM**** generate a list of the designed output functions
```

```
          (d-functions     (solve-fcns (bcf int-d-function)
                                       outputs  outputs))
          (fault-classes  (solve-cps final-equation outputs)) )

      (newline)
      (writeln "          ********* Results *********")

      ; print out the function that the circuit is performing, the
      ; function that it is supposed to perform, and whether the
      ; two functions are equivalent

;; ****VM**** changed to account for several possible functions
    (display-functions equivalence-result a-functions d-functions
                                                        outputs)
      ; print out the possible faults in the circuit
      (interpret-faults checkpoints fault-classes intermediate-format)
      ; display the performance metrics of the system
      (display-system-metrics inputs no-of-tests)
      (writeln "Would you like to try another circuit? ")
      (writeln "If so, type yes and <rtn>, else type no and <rtn>.")
      (writeln "A reply of no returns you to the main menu.")
      (display "Enter yes or no (default is no) --> ")

      (let ( (reply (read-line)) )
         (if (equal? reply "yes")
             T
             '() ))))


;; (SOLVE-FCN equation checkpoints outputs)
;;
;; Parameters:
;;    equation - The final equation produced by procedure TESTER.  This
;;            equation holds all information about the state of the system
;;               after it has been determined that no new information can
;;               be determined from further input-output tests.
;;    checkpoints - A list of the checkpoint variables introduced into
;;                  the equation.
;;    outputs - A list of the output nodes of the circuit.
;;
;; -- SOLVE-FCN is used to generate the equation that the circuit is
;;    performing based on the results of the input-output experiments.
;; -- The input EQUATION is of the form:
;;
;;        EQUATION(x,y,z) = 0  where x is the input variables,
```

```
;;                              y is the checkpoint variables,
;;                              and z is a single output variable
;;
;; -- This EQUATION must then be converted to the form:
;;
;;     R(x) z' + S(x) z + T(y) = 0  where R(x) & S(X) are functions of the
;;                                        input variables, and T(y) is a
;;                                       function of the checkpoint variables
;;
;; -- R(x) yields the actual circuit function.  To obtain R(x), the
;;    OUTPUTS can be DIVIDEd into the EQUATION using Boolean division.
;;    This leaves an equation in terms of inputs and checkpoints.
;;    Then the CHECKPOINTS can be removed using conjunctive ELIMINATion
;;    to yield the single formula R(x).

;; ****VM**** this function has been created to process several output
;;            functions
(define (solve-fcns equation outputs outputs2)
  (if (null? outputs) nil
      (let* ( (solved-fcn (solve-fcn equation
                                (list (car outputs)) outputs2)))
     (append  (list solved-fcn)
              (solve-fcns equation (cdr outputs) outputs2)))))

;; ****VM**** changed to extract one output function at a time

(define (solve-fcn equation output outputs)
    (let* ( (new-outputs (remove (car output) outputs))
            (eqn (divide equation output))
            (eqn-minus-outputs (eliminate eqn new-outputs)))
        eqn-minus-outputs))



;; (SOLVE-CPS equation outputs)
;;
;; Parameters:
;;    equation - The final equation produced by procedure TESTER.  This
;;            equation holds all information about the state of the system
;;               after it has been determined that no new information can
;;               be determined from further input-output tests.
;;    outputs - A list of the output nodes of the circuit.
;;
;; -- SOLVE-CPS is used to generate the equation which can be solved to
;;    determine the possible faults in the circuit.  This equation is
```

```
;;    based on the results of the input-output experiment.
;; -- EQUATION is of the form:
;;
;;        EQUATION(x,y,z) = 0  where x is the input variables,
;;                             y is the checkpoint variables,
;;                             and z is a single output variable
;;
;; -- This EQUATION must then be converted to the form:
;;
;;     R(x) z' + S(x) z + T(y) = 0  where R(x) & S(X) are functions of the
;;                                    input variables, and T(y) is a
;;                                   function of the checkpoint variables
;;
;; -- T(y) yields the possible faults function.  To obtain T(y), the
;;    OUTPUTS can be ELIMINATEd from the EQUATION using conjunctive
;;    elimination.  This leaves an equation in terms of the checkpoints.
;; -- This equation is in f=0 form which is difficult to solve to
;;    determine the states of the checkpoint variables.  Thus, the
;;    equation is COMPLEMENT to get the f=1 form.  Then, this equation is
;;    SIMPLIFied to yield an equation in which the terms represent the
;;    possible faults in the circuit.
;; -- Literals that exist in each of the terms are variables the state
;;    of which has been positively determined.  When these variables are
;;    removed, the terms left represent the possible faults that may exist
;;    in the circuit.

(define (solve-cps equation outputs)
    (simplify (complement (eliminate equation outputs))) )


;; (DISPLAY-FUNCTIONS function outputs intermediate-format)
;;
;; Parameters:
;;    function - The function that the circuit is performing as determined
;;               by SOLVE-FCN.
;;    outputs - A list of the outputs of the circuit.
;;    intermediate-format - The data structure, a list in prefix-form that
;;                          was returned by procedure RUN-INPUT-MODULE.
;;                          This list is used to determine the function
;;                          that the circuit was designed to perform.
;;
;; -- DISPLAY-FUNCTIONS determines the circuit's ACTUAL-FUNCTION, the
;;    circuit's DESIGNED-FUNCTION and prints these functions to the
;;    screen in the form of a Boolean equation.
;; -- An equivalency test is made to determine if these functions are
```

```
;;    equivalent.  The result of this test is output to the screen.
;; -- The FUNCTION is XORed with the OUTPUTS to get ACTUAL-FUNCTION in
;;    f=0 form.  The prefix-form of the circuit, represented by the
;;    INTERMEDIATE-FORMAT is used to determine the DESIGNED-FUNCTION.
;;    The prefix-form must be reduced by MAKE-SOP and INTERNAL-NODES
;;    must be ELIMINATEd to yield an equation in the form of inputs
;;    and outputs without internal nodes.
;; -- FUNCTION-D the DESIGNED-FUNCTION in the same form as the input
;;    parameter FUNCTION to allow use of a single procedure,
;;    DISPLAY-CIRCUIT-FUNCTION, in displaying the circuit function.
;; -- EQUIVALENCE-RESULT is the result of XORing the DESIGNED-FUNCTION with
;;    the ACTUAL-FUNCTION.  When two f=0 equation are XORed together, if
;;    the result is 0, or in this representation '(), then the equations
;;    are equivalent.

;; ****VM**** changed to display several functions when multiple
;;            outputs exist
(define (display-functions equivalence-result a-function d-function
                                                    outputs)
     (newline)
 (writeln "The function(s) that the circuit was designed to perform is: ")
     (newline)
     (display-circuit-functions d-function outputs)

     (newline)
     (writeln "The function(s) that the circuit is performing is: ")
     (newline)
     (display-circuit-functions a-function outputs)

     (newline)
     (if (equal? equivalence-result '())
         (begin
            (display "The actual circuit IS equivalent to the ")
            (writeln "designed circuit."))
         (begin
            (display "The actual circuit IS NOT equivalent to the ")
            (writeln "designed circuit.")) ))


;; (DISPLAY-CIRCUIT-FUNCTION function outputs)
;;
;; Parameters:
;;    function - An equation representing the function of the circuit.
;;    outputs - The outputs of the circuit.
;;
```

```
;; -- DISPLAY-CIRCUIT-FUNCTION takes an equation representing the
;;    function that the circuit is performing, an displays this equation.
;; -- CONVERT-NODE-BACK is used to remove the suffix from the output node
;;  symbol so that it is output in the form of the original output symbol
;;    that was used by the user.  This node is DISPLAYed followed by an
;;    equals sign.
;; -- DISPLAY-CIRCUIT-FUNCTION-1 is called to display the FUNCTION
;;    which is only in terms of the inputs.

;; ****VM**** displays functions by calling
;;            DISPLAY-CIRCUIT-FUNCTION-1 to generate one function
;;            at a time

(define (display-circuit-functions functions outputs)
  (if (null? outputs) nil
   (let ( (output-node (convert-node-back (car outputs))) )
      (newline)
      (display "    ")
      (display output-node)
      (display " = ")
      (display-circuit-function-1 (car functions))
      (newline)
      (display-circuit-functions (cdr functions) (cdr outputs)))))


;; (DISPLAY-CIRCUIT-FUNCTION-1 function)
;;
;; Parameters:
;;    function - A formula representing the function of the circuit.
;;
;; -- DISPLAY-CIRCUIT-FUNCTION-1 displays the circuit function.
;; -- FUNCTION is a list of the form:
;;
;;       ((X1 (X2) X3) ((X1) X4) (X5) ((X6)))
;;
;;    which represents the formula:
;;
;;        X1 X2'X3 + X1'X4 + X5 + X6'
;;
;;    Each of the top-level sublists is a term of this formula.  If a
;; literal exists in the top-level sublist in the form of a sublist, then
;;    it exists logically in complemented form; uncomplemented otherwise.
;; -- FIRST-TERM is CARed from the FUNCTION and displayed by DISPLAY-TERM.
;; --If there are remaining terms in FUNCTION, then a + sign is DISPLAYed,
;;    and DISPLAY-CIRCUIT-FUNCTION-1 is called recursively to display the
```

```
;;    remaining terms of the formula.

(define (display-circuit-function-1 function)
   (if (null? function)
       (display-term function)
       (let ( (first-term (car function)) )
          (display-term (list first-term))
          (if (not (null? (cdr function)))
              (begin
                 (display "+ ")
                 (display-circuit-function-1 (cdr function)))
              '()) )))


;; (DISPLAY-TERM term)
;;
;; Parameters:
;;    term - a list of the form (((X1) X2 (X3))) where each of the
;;           top level elements represents a term of a Boolean equation.
;;           The example list represents a single term X1'X2 X3'.
;;
;; -- DISPLAY-TERM prints a "1" if the term is of the form '(()) which
;;    represents a Boolean 1.
;; -- DISPLAY-TERM prints a "0" if the term is of the form '() which
;;    represents a Boolean 0.
;;--If TERM is not of this form, DISPLAY-TERM-1 is called to display TERM.

(define (display-term term)
  (cond ( (member nil term)
          (princ "1 ") )
        ( (null? term)
          (princ "0 ") )
        ( else
          (display-term-1 term) )))

;; (DISPLAY-TERM-1 term)
;;
;; Parameters: a list of the form (((X1) X2 (X3))) where each of the
;;           top level elements represents a term of a Boolean equation.
;;           The example list represents a single term X1'X2 X3'.
;;
;; -- If TERM is nil, then DISPLAY-TERM-1 returns '().  Otherwise, the
;;    TERM is sorted by SORT-TERM from file boolean.s.  Then, the first
;;    term is displayed by DISPLAY-TERM-2.  The remaining terms are
;;    displayed by a recursive call to DISPLAY-TERM-1.
```

```
(define (display-term-1 term)
  (cond ( (null? term)
          '() )
        ( else
            (display-term-2 (sort-term (car term)))
            (display-term-1 (cdr term)) )))

;; (DISPLAY-TERM-2 term)
;;
;; Parameters:
;;    term - a list of the form ((X1) X2 (X3)) representing the term
;;           X1'X2 X3'.
;;
;; -- DISPLAY-TERM-2 takes a list representing a term and prints out
;;    each of the literals until the entire term has been output.
;;--If a literal exists in the term as a sublist, then it is complemented,
;;    and a "'" (prime) is output immediately after the literal.
;;    Otherwise, a space is output after the literal.  DISPLAY-TERM-2
;;    is called recursively to output the remaining literals of the TERM.
;; -- CONVERT-NODE-BACK is called to remove the suffix from the nodes
;;    so that they are output in the form of the original node symbols
;;    used by the user.

(define (display-term-2 term)
  (cond ( (null? term)
          '() )
        ( (atom? (car term))
          (princ (convert-node-back (car term))) (princ " ")
          (display-term-2 (cdr term)) )
        ( else
          (princ (convert-node-back (car (car term)))) (princ "'")
          (display-term-2 (cdr term)) )))

;; (CONVERT-NODE-BACK node)
;;
;; Parameters:
;;    node - A symbol of the form ABC--.
;;
;; -- CONVERT-NODE-BACK accepts a NODE of the given form, removing the
;;    last two characters and returning a symbol of the form ABC.

(define (convert-node-back node)
   (let* ( (node-l (string->list (symbol->string node)))
           (node-less-suffix (remove-suffix node-l)) )
```

```
      (string->symbol (list->string node-less-suffix)) ))

;; (INTERPRET-FAULTS checkpoints fault-classes intermediate-format)
;;
;; Parameters:
;;    checkpoints - A list of the checkpoint variables generated by the
;;                  system.
;;    fault-classes - A list of lists representing different fault cases
;;                    that may occur.
;;    intermediate-format - The data structure, a list in prefix-form that
;;                          was returned by procedure RUN-INPUT-MODULE.
;;                          This list is used to determine the appropriate
;;                        gate a given fanout node is associated with when
;;                          printing out faults for each node.
;;
;; -- INTERPRET-FAULTS is called to derive the faults in the circuit,
;;    both those that can be positively determined as well as cases of
;; faults that may have occurred, but cannot be determined with certainty.
;; -- REMOVE-LAST-CHAR-FROM-ALL-ELTS accepts the list of CHECKPOINTS which
;;    is of the form (AX0 AX1 B00 B01 B10 B11 CX0 CX1) and returns a list
;;    of the form (AX B0 B1 CX).  This latter list represents the actual
;;    checkpoints in the circuit.  GET-INPUT-CHECKPOINTS accepts the new
;;    list and returns a list of the INPUT-CHECKPOINTS which is a list of
;;    the form (AX CX).  GET-SUBLIST subtracts the INPUT-CHECKPOINTS list
;;    from the new list to form the FANOUT-CHECKPOINTS list, which in this
;;example would be (B0 B1).  The fanout checkpoints must be distinguished
;;    from the input checkpoints because the output of the faults for
;;    these two distinct types of checkpoints is different.  The input
;;    nodes be only listed.  The fanout node faults must have the gate
;;    displayed also so the user knows which fanout stem may have a fault.
;; -- GET-NORMAL-NODES accepts the list of FAULT-CLASSES and determines
;;    the normal nodes in the list.  The second parameter is the list
;;    of the nodes to check for normality.  In the first call to
;; GET-NORMAL-NODES, the INPUT-CHECKPOINTS are checked to see if they are
;;    normal.  In the second call, the FANOUT-CHECKPOINTS are checked.
;; -- REMOVE-NORMAL-NODES is called to remove the NORMAL-INPUT-NODES and
;;    the NORMAL-FANOUT-NODES from the fault classes, producing
;;    FAULT-CLASSES-1 and FAULT-CLASSES-2, respectively.
;;--GET-COMMON-NODES gets all of the literals common to each of the terms
;;    after the normal nodes have been removed.  REMOVE-COMMON-NODES
;;    removes the COMMON-NODES from FAULT-CLASSES-2 to produce
;;    FAULT-CLASSES-3 which is a list of terms which have no literals
;;    (sublists) in common.  Each of these terms represents a different
;;    fault that may have occurred in the circuit.
;; -- GET-COMMON-INPUT-NODES extracts the COMMON-INPUT-NODES from
```

```
;;    the COMMON-NODES.  GET-SUBLISTS subtracts the COMMON-INPUT-NODES
;;    from the COMMON-NODES to get the COMMON-FANOUT-NODES.  The
;;    COMMON-INPUT-NODES and COMMON-FANOUT-NODES are used to get the
;;    stuck-at-0, stuck-at-1, not-stuck-at-0, and not-stuck-at-1 nodes
;;    for both the input and fanout nodes.  Lists are made for each case.
;;    In many cases, these list may be nil.
;; -- SHOW-LIST-OF-NODES is called to print out the input nodes for
;;    the appropriate fault.  SHOW-FANOUTS is called to print out the
;;    fanout nodes for the appropriate fault.  SHOW-FANOUTS outputs the
;; appropriate node as well as the gate that the node is associated with.
;;    A given fanout node may have a fanout of three, each of which has
;;    an associated checkpoint.  Thus, the gate must be associated with the
;;    checkpoint when the checkpoint fault status is output.
;; -- Remaining fault cases, those that represent different faults that
;;    may be occurring in the circuit are interpreted by a call to
;;    INTERPRET-FAULT-CASES.  CHECKPOINTS-1, the INTERMEDIATE-FORMAT, and
;;    FAULT-CLASSES-3 are passed to INTERPRET-FAULT-CASES.


(define (interpret-faults checkpoints fault-classes intermediate-format)

  (let* ( (checkpoints-1   (remove-last-char-from-all-elts checkpoints))
          (input-checkpoints    (get-input-checkpoints checkpoints-1))
          (fanout-checkpoints   (get-sublist checkpoints-1
                                             input-checkpoints))
          (normal-input-nodes   (get-normal-nodes input-checkpoints
                                                  fault-classes))
          (fault-classes-1      (remove-normal-nodes normal-input-nodes
                                                     fault-classes))
          (normal-fanout-nodes  (get-normal-nodes fanout-checkpoints
                                                  fault-classes-1))
          (fault-classes-2      (remove-normal-nodes normal-fanout-nodes
                                                     fault-classes-1))

          (prefix-list       (make-unique-fanouts intermediate-format))
          (common-nodes         (get-common-nodes fault-classes-2))
          (fault-classes-3      (remove-common-nodes common-nodes
                                                     fault-classes-2))
          (common-input-nodes   (get-common-input-nodes common-nodes))
          (common-fanout-nodes  (get-sublist common-nodes
                                             common-input-nodes))

        (input-nodes-s-a-0    (get-stuck-at-0-nodes common-input-nodes))
        (input-nodes-s-a-1    (get-stuck-at-1-nodes common-input-nodes))
      (input-nodes-n-s-a-0  (get-not-stuck-at-0-nodes common-input-nodes))
      (input-nodes-n-s-a-1  (get-not-stuck-at-1-nodes common-input-nodes))
```

```
     (fanout-nodes-s-a-0   (get-stuck-at-0-nodes common-fanout-nodes))
     (fanout-nodes-s-a-1   (get-stuck-at-1-nodes common-fanout-nodes))
 (fanout-nodes-n-s-a-0 (get-not-stuck-at-0-nodes common-fanout-nodes))
 (fanout-nodes-n-s-a-1 (get-not-stuck-at-1-nodes common-fanout-nodes))
                                                                      )

    (newline)
    (newline)
    (display "**** The following information is certain ")
    (writeln "about the circuit **** ")

    (newline)
    (writeln "Input nodes (which do not fanout) that are normal:")
    (newline)
    (if (null? normal-input-nodes)
        (writeln "    --none--")
        (show-list-of-nodes normal-input-nodes))

    (newline)
    (writeln "Input nodes (which do not fanout) that are stuck-at-0:")
    (newline)
    (if (null? input-nodes-s-a-0)
        (writeln "    --none--")
        (show-list-of-nodes input-nodes-s-a-0))

    (newline)
    (writeln "Input nodes (which do not fanout) that are stuck-at-1:")
    (newline)
    (if (null? input-nodes-s-a-1)
        (writeln "    --none--")
        (show-list-of-nodes input-nodes-s-a-1))
    (newline)
(writeln "Input nodes (which do not fanout) that are NOT stuck-at-0:")
    (newline)
    (if (null? input-nodes-n-s-a-0)
        (writeln "    --none--")
        (show-list-of-nodes input-nodes-n-s-a-0))
    (newline)
(writeln "Input nodes (which do not fanout) that are NOT stuck-at-1:")
    (newline)
    (if (null? input-nodes-n-s-a-1)
        (writeln "    --none--")
        (show-list-of-nodes input-nodes-n-s-a-1))
    (newline)
    (newline)
```

```
        (writeln "Fanout nodes that are normal:")
        (newline)
        (if (null? normal-fanout-nodes)
            (writeln "    --none--")
            (show-fanouts normal-fanout-nodes prefix-list))
        (newline)
        (writeln "Fanout nodes that are stuck-at-0:")
        (newline)
        (if (null? fanout-nodes-s-a-0)
            (writeln "    --none--")
            (show-fanouts fanout-nodes-s-a-0 prefix-list))
        (newline)
        (writeln "Fanout nodes that are stuck-at-1:")
        (newline)
        (if (null? fanout-nodes-s-a-1)
            (writeln "    --none--")
            (show-fanouts fanout-nodes-s-a-1 prefix-list))
        (newline)
        (writeln "Fanout nodes that are NOT stuck-at-0:")
        (newline)
        (if (null? fanout-nodes-n-s-a-0)
            (writeln "    --none--")
            (show-fanouts fanout-nodes-n-s-a-0 prefix-list))
        (newline)
        (writeln "Fanout nodes that are NOT stuck-at-1:")
        (newline)
        (if (null? fanout-nodes-n-s-a-1)
            (writeln "    --none--")
            (show-fanouts fanout-nodes-n-s-a-1 prefix-list))
        ; interpret the remaining cases, if they exist
        (newline)
        (if (not (equal? fault-classes-3 '(())) )
            (interpret-fault-cases checkpoints-1
                                   fault-classes-3
                                   intermediate-format)) ))

;; (SHOW-LIST-OF-NODES nodes)
;;
;; Parameters:
;;    nodes - a list of nodes
;;
;; -- SHOW-LIST-OF-NODES accepts a list of the form (AX BX CX) and
;;    removes the last character from each of the symbols to produce a
;;    MODIFIED-LIST of the form (A B C).
;;--SHOW-NODES is then called to display each of the nodes in the new list.
```

```
(define (show-list-of-nodes nodes)
   (let* ( (modified-list (remove-last-char-from-all-elts nodes)) )
      (show-nodes modified-list)))

;; (SHOW-NODES lst)
;;
;; Parameters:
;;    lst - an arbitrary list
;;
;; -- SHOW-NODES displays each of the elements of LST on a separate line,
;;    until there are no further elements to display.

(define (show-nodes lst)
   (if (null? lst)
       '()
       (begin
          (display "    ")
          (writeln (car lst))
          (show-nodes (cdr lst)))))

;; (REMOVE-LAST-CHAR-FROM-ALL-ELTS lst)
;;
;; Parameters:
;;    lst - an arbitrary list
;;
;; -- REMOVE-LAST-CHAR-FROM-ALL-ELTS removes that last character from every
;;    symbol in an arbitrary list.  The procedure breaks down sublists to
;;    change every symbol at any level.
;; -- It is assumed that every symbol has two or more characters.

(define (remove-last-char-from-all-elts lst)
   (cond ( (null? lst)
           '())
         ( (symbol? lst)
           (remove-last-char-from-symbol lst) )
         ( (symbol? (car lst))
           (remove-duplicates
              (cons (remove-last-char-from-symbol (car lst))
                    (remove-last-char-from-all-elts (cdr lst)))) )
         ( else
             (remove-duplicates
                (cons (remove-last-char-from-all-elts (car lst))
                      (remove-last-char-from-all-elts (cdr lst)))) )))
```

```
;; (SHOW-FANOUTS fanout-nodes prefix-list)
;;
;; Parameters:
;;    fanout-nodes - a list of nodes of the form (B1 B0)
;;    prefix-list - The prefix list of the input circuit; this list
;;                  was modified to MAKE-UNIQUE-FANOUTS of each of the
;;                  fanout nodes.  This is necessary to distinguish the
;;             fanouts and associate them with the list of fanout nodes.
;;
;; -- SHOW-FANOUT takes the PREFIX-LIST and removes that last char from
;;    each of the NODE-SYMBOLS.  This leaves a list of the form:
;;
;;       ((EQ E- (NOT (* AX B0)))
;;        (EQ F- (NOT B1))
;;        (EQ Z- (NOT (* E- F-))))
;;
;; -- SHOW-FANOUTS-1 is then passed the list of FANOUT-NODES and the
;;    new prefix list.

(define (show-fanouts fanout-nodes prefix-list)
(let ((prefix-list-1 (remove-last-char-from-node-symbols prefix-list)))
      (show-fanouts-1 fanout-nodes prefix-list-1) ))

;; (SHOW-FANOUTS-1 fanout-nodes prefix-list)
;;
;; Parameters:
;;    fanout-nodes - A list of fanout nodes
;;    prefix-list - the modified prefix-list from SHOW-FANOUTS
;;
;; -- SHOW-FANOUTS-1 iteratively outputs each of the FANOUT-NODES in the
;;    input list, displaying in sequence the NODE and then the GATE
;;    associated with that fanout node.
;; -- GET-GATE returns the EQUATION associated with the fanout node.
;;    For example, if node B0 were to be displayed, then the EQUATION
;; returned by GET-GATE would be (EQ E- (NOT (* AX B0))).  This equation
;;    is displayed by SHOW-EQUATION.
;;--SHOW-FANOUTS-1 calls itself recursively until all of the FANOUT-NODES
;;    in the original list have been displayed with the appropriate gate.

(define (show-fanouts-1 fanout-nodes prefix-list)
   (if (null? fanout-nodes)
       '()
       (let ( (equation (get-gate (car fanout-nodes) prefix-list)) )
          (display "    Node ")
          (display (remove-last-char-from-symbol (car fanout-nodes)))
```

```
        (display " of gate: ")
        (show-equation equation)
        (show-fanouts-1 (cdr fanout-nodes) prefix-list) )))


;; (SHOW-EQUATION equation)
;;
;; Parameters:
;;    equation - a list of the form (EQ E- (NOT (* AX BO)))
;;
;; -- SHOW-EQUATION displays the above equation in the form E = A * B.
;; -- First, the output node for the gate is display followed by an equals
;;    sign.  Then either SHOW-NEGATED-EQUATION, or SHOW-FORMULA are called
;;    depending on whether the gate is of the NEGATED variety, i.e. NAND.

(define (show-equation equation)
   (let* ( (output (cadr equation))
           (input  (caddr equation)) )
      (display (remove-last-char-from-symbol output))
      (display " = ")
      (if (equal? (car input) 'NOT)
          (show-negated-equation (cadr input))
          (show-formula input))
      (newline) ))


;; (INTERPRET-FAULT-CASES checkpoints fault-classes intermediate-format)
;;
;; Parameter:
;;    checkpoints - A list of the form (AX BO B1 CX).
;;    fault-classes - A list of lists in which each sublist is a term
;;                    representing a distinct fault class.
;;    intermediate-format - the intermediate-format from RUN-INPUT-MODULE.
;;
;; -- INTERPRET-FAULT-CASES prints out an introductory message and then
;;    calls INTERPRET-FAULT-CASES-1.  All parameters are passed.  A new
;;  parameter, the number 1 is passed to INTERPRET-FAULT-CASES which uses
;;    this number to keep track of the different fault cases.

(define (interpret-fault-cases checkpoints fault-classes
                               intermediate-format)
   (newline)
   (writeln "**** One of the following cases holds for the circuit ****")
   (newline)
(interpret-fault-cases-1 checkpoints fault-classes intermediate-format 1))
```

```
;; (INTERPRET-FAULT-CASES-1 checkpoints fault-classes intermediate-format
;;                                                          case-number)
;;
;; Parameters:
;;    checkpoints - A list of the form (AX B0 B1 Cλ).
;;    fault-classes - A list of lists in which each sublist is a term
;;                    representing a distinct fault class.
;;    intermediate-format - The intermediate-format from RUN-INPUT-MODULE.
;;    case-number - An integer.  Initially, this number is 1.  Every time
;;                  that INTERPRET-FAULT-CASES-1 is called recursively to
;;                  interpret another case, this number is incremented.
;;
;; -- INTERPRET-FAULT-CASES-1 operates similarly to INTERPRET-FAULTS.
;;    However, it is tailored to interpreting distinct fault cases that
;;    may have occurred in the circuit.
;;--The first term (FIRST-CASE) is removed from the list of FAULT-CLASSES.
;;    FIRST-CASE is examined to determine the types of faults associated
;;    with each of the nodes in the term.  The order this is done is the
;;    same as in INTERPRET-FAULTS.  There is no need to check for
;;    COMMON-NODES, because no common nodes exist between terms of
;;    FAULT-CLASSES when this procedure is invoked.
;;--The only case found different in this procedure than in INTERPRET-
;;  FAULTS
;;    is that nodes may be found that have been interpreted to be
;;    stuck-at-0 or stuck-at-1 in which the complementary not-stuck-at-1
;;  or not-stuck-at-0, respectively, variable is not found.  In this case,
;;    lists are made of "only" stuck-at-0 or "only" stuck-at-1 nodes.
;;    However, the display procedures do not differentiate between
;;    stuck-at-0 and only-stuck-at-0 and stuck-at-1 and only-stuck-at-1.
;; -- For each case of faults, the input node and fanout node faults
;;    are displayed together.  SHOW-INPUT-NODES is called to display
;;    input nodes, and SHOW-FANOUT-NODES is called to display the
;;    fanout nodes and associated gates.
;;--After a case is interpreted and displayed, then INTERPRET-FAULT-
;;  CASES-1
;;    calls itself recursively until all cases have been displayed.
;;    CASE-NUMBER is incremented with each recursive call.

(define (interpret-fault-cases-1 checkpoints
                                 fault-classes
                                 intermediate-format
                                 case-number)

   (if (not (null? fault-classes))
```

```
(let* ( (first-case           (list (car fault-classes)))
        (input-checkpoints    (get-input-checkpoints checkpoints))
        (fanout-checkpoints   (get-sublist checkpoints
                                           input-checkpoints))
        (normal-input-nodes   (get-normal-nodes input-checkpoints
                                                first-case))
      (first-case-1         (remove-normal-nodes normal-input-nodes
                                                 first-case))
        (normal-fanout-nodes  (get-normal-nodes fanout-checkpoints
                                                first-case-1))
      (first-case-2         (remove-normal-nodes normal-fanout-nodes
                                                 first-case-1))
    (prefix-list          (make-unique-fanouts intermediate-format))
        (prefix-list-1        (remove-last-char-from-node-symbols
                                                   prefix-list))
        (common-nodes         (get-common-nodes first-case-2))
      (input-faults         (get-common-input-nodes common-nodes))
     (fanout-faults        (get-sublist common-nodes input-faults))
        (input-nodes-s-a-0    (get-stuck-at-0-nodes input-faults))
        (input-nodes-s-a-1    (get-stuck-at-1-nodes input-faults))
     (input-nodes-n-s-a-0  (get-not-stuck-at-0-nodes input-faults))
     (input-nodes-n-s-a-1  (get-not-stuck-at-1-nodes input-faults))
    (input-nodes-o-s-a-0  (get-only-stuck-at-0-nodes input-faults))
    (input-nodes-o-s-a-1  (get-only-stuck-at-1-nodes input-faults))
        (fanout-nodes-s-a-0   (get-stuck-at-0-nodes fanout-faults))
        (fanout-nodes-s-a-1   (get-stuck-at-1-nodes fanout-faults))
    (fanout-nodes-n-s-a-0 (get-not-stuck-at-0-nodes fanout-faults))
    (fanout-nodes-n-s-a-1 (get-not-stuck-at-1-nodes fanout-faults))
   (fanout-nodes-o-s-a-0 (get-only-stuck-at-0-nodes fanout-faults))
   (fanout-nodes-o-s-a-1 (get-only-stuck-at-1-nodes fanout-faults))
                                                                   )

   (writeln "    **** Case #" case-number " ****")
   (newline)

   (if (not (null? normal-input-nodes))
       (show-input-nodes normal-input-nodes 'normal))

   (if (not (null? input-nodes-s-a-0))
       (show-input-nodes input-nodes-s-a-0 'stuck-at-0))

   (if (not (null? input-nodes-s-a-1))
       (show-input-nodes input-nodes-s-a-1 'stuck-at-1))

   (if (not (null? input-nodes-n-s-a-0))
```

```
    (show-input-nodes input-nodes-n-s-a-0
                      'not-stuck-at-0))
(if (not (null? input-nodes-n-s-a-1))
    (show-input-nodes input-nodes-n-s-a-1
                      'not-stuck-at-1))
(if (not (null? input-nodes-o-s-a-0))
    (show-input-nodes input-nodes-o-s-a-0 'only-stuck-at-0))
(if (not (null? input-nodes-o-s-a-1))
    (show-input-nodes input-nodes-o-s-a-1 'only-stuck-at-1))
(if (not (null? normal-fanout-nodes))
    (show-fanout-nodes normal-fanout-nodes
                       'normal
                       prefix-list-1))
(if (not (null? fanout-nodes-s-a-0))
    (show-fanout-nodes fanout-nodes-s-a-0
                       'stuck-at-0
                       prefix-list-1))
(if (not (null? fanout-nodes-s-a-1))
    (show-fanout-nodes fanout-nodes-s-a-1
                       'stuck-at-1
                       prefix-list-1))

(if (not (null? fanout-nodes-n-s-a-0))
    (show-fanout-nodes fanout-nodes-n-s-a-0
                       'not-stuck-at-0
                       prefix-list-1))
(if (not (null? fanout-nodes-n-s-a-1))
    (show-fanout-nodes fanout-nodes-n-s-a-1
                       'not-stuck-at-1
                       prefix-list-1))
(if (not (null? fanout-nodes-o-s-a-0))
    (show-fanout-nodes fanout-nodes-o-s-a-0
                       'only-stuck-at-0
                       prefix-list-1))
(if (not (null? fanout-nodes-o-s-a-1))
    (show-fanout-nodes fanout-nodes-o-s-a-1
                       'only-stuck-at-1
                       prefix-list-1))
(newline)
(display "Press <return> to continue.")
(pause)
(newline)
(newline)
(interpret-fault-cases-1 checkpoints
                         (cdr fault-classes)
```

```
                                    intermediate-format
                                    (1+ case-number)) )))

;; (REMOVE-LAST-CHAR-FROM-SYMBOL symbol)
;;
;; Parameters:
;;    symbol - an arbitrary symbol
;;
;; -- REMOVE-LAST-CHAR-FROM-SYMBOL decomposes the symbol, drops the
;;    last character from the symbol, reassembles the symbol and
;;    returns the NEW-SYMBOL.

(define (remove-last-char-from-symbol symbol)
   (let* ( (symbol-l (string->list (symbol->string symbol)))
           (new-list (drop-last-char symbol-l))
           (new-symbol (string->symbol (list->string new-list))) )
      new-symbol ))

;; (GET-INPUT-CHECKPOINTS checkpoints)
;;
;; Parameters:
;;    checkpoints - A list of the form (AX B0 B1 CX).
;;
;; -- GET-INPUT-CHECKPOINTS returns a list in which the last character of
;; every symbol is an X.  The distinguishes primary input checkpoints from
;; other checkpoints in the system.  LAST-CHAR-EQ-X? is used to determine
;;    whether a given symbol has a last character of X.
;; -- For the given list, (AX CX) would be returned.

(define (get-input-checkpoints checkpoints)
   (cond ( (null? checkpoints)
           '())
         ( else
           (if (last-char-eq-x? (car checkpoints))
               (cons (car checkpoints)
                     (get-input-checkpoints (cdr checkpoints)))
               (get-input-checkpoints (cdr checkpoints))) )))

;; (LAST-CHAR-EQ-X? symbol)
;;
;; Parameter:
;;    symbol - an arbitrary symbol
;;
;; -- LAST-CHAR-EQ-X? decomposes the symbol.  GET-LAST-ELT is used to
;;    get the last element from the list of characters (SYMBOL-L) that
```

```
;;    comprise the SYMBOL.  If the LAST-CHAR equals X, then #T is returned.
;;    Otherwise, '() is returned.

(define (last-char-eq-x? symbol)
   (let* ( (symbol-l  (string->list (symbol->string symbol)))
           (last-char (get-last-elt symbol-l)) )

      (equal? last-char '(#\X)) ))


;; (GET-NORMAL-NODES checkpoints fault-classes)
;;
;; Parameters:
;;    checkpoints - A list of checkpoints, either of the form (AX BX)
;;                  or (C0 C1).
;;    fault-classes - A list of lists in which each top level sublist
;;                    represents a set of faults that may have occurred
;;                    in the circuit.
;;
;; -- GET-NORMAL-NODES takes a node from the lists of checkpoints and
;;    tests to see whether it is normal by calling NORMAL-NODE?.
;; -- If the given checkpoint is normal, it is added to the list that is
;;    returned.  Otherwise, it is not.

(define (get-normal-nodes checkpoints fault-classes)
   (cond ( (null? checkpoints)
           '())
         ( (normal-node? (car checkpoints) fault-classes)
           (cons (car checkpoints)
                 (get-normal-nodes (cdr checkpoints) fault-classes)) )
         ( else
           (get-normal-nodes (cdr checkpoints) fault-classes) )))


;; (NORMAL-NODE? checkpoint fault-classes)
;;
;; Parameters:
;;    checkpoint - a single checkpoint symbol
;;    fault-classes - A list of lists in which each top level sublist
;;                    represents a set of faults that may have occurred
;;                    in the circuit.
;;
;; -- NORMAL-NODE? takes a checkpoint symbol of the form AX or B0 and
;;    creates the symbols AX0 and AX1, or B00 and B01, respectively.
;; -- Then MEMBER-ALL-LISTS? is called to see if the complemented form of
```

```
;;    each of these variables is in every one of the sublists.
;; -- If both complemented forms are in every sublist, then and only then
;;    is that node normal.

(define (normal-node? checkpoint fault-classes)
   (let* ( (checkpoint-1 (string->list (symbol->string checkpoint)))
           (checkpoint-0 (append checkpoint-1
                                 (string->list (number->string 0 '(int)))))
           (checkpoint-1 (append checkpoint-1
                                 (string->list (number->string 1 '(int)))))
           (symbol-0 (string->symbol (list->string checkpoint-0)))
           (symbol-1 (string->symbol (list->string checkpoint-1))) )

      (and (member-all-lists? (list symbol-0) fault-classes)
           (member-all-lists? (list symbol-1) fault-classes)) ))

;; (REMOVE-NORMAL-NODES normal-nodes fault-classes)
;;
;; Parameters:
;;    normal-nodes - A list produced by GET-NORMAL-NODES of the form
;;                   (AX BX) or (C0 C1).
;;    fault-classes - A list of lists in which each top level sublist
;;                    represents a set of faults that may have occurred
;;                    in the circuit.
;;
;; -- REMOVE-NORMAL-NODES removes the NORMAL-NODES from the FAULT-CLASSES
;;    to produce a new list of fault classes with all of the NORMAL-NODES
;;    removed.
;; -- REMOVE-NORMAL-NODES-1 is called to modify the list of FAULT-CLASSES
;;    for a single node.  REMOVE-NORMAL-NODES calls itself recursively
;;until all NORMAL-NODES have been removed from the list of FAULT-CLASSES.

(define (remove-normal-nodes normal-nodes fault-classes)
   (if (null? normal-nodes)
       fault-classes
       (remove-normal-nodes (cdr normal-nodes)
                            (remove-normal-nodes-1 (car normal-nodes)
                                                   fault-classes))))

;; (REMOVE-NORMAL-NODES-1 node fault-classes)
;;
;; Parameters:
;;    node - A node of the form AX or B0.
;;    fault-classes - The list of lists representing fault classes.
;;
```

```
;;-REMOVE-NORMAL-NODES-1 takes the input NODE and creates the appropriate
;;    checkpoint symbols, for AX this would be AX0 and AX1, and uses a
;;    Boolean DIVIDE to remove these symbols from every term.

(define (remove-normal-nodes-1 node fault-classes)
   (let* ( (node-l   (string->list (symbol->string node)))
      (node-0   (append node-l (string->list (number->string 0 '(int)))))
      (node-1   (append node-l (string->list (number->string 1 '(int)))))
           (symbol-0 (list (string->symbol (list->string node-0))))
           (symbol-1 (list (string->symbol (list->string node-1)))) )

      (divide (divide fault-classes symbol-0) symbol-1) ))


;; (GET-COMMON-NODES lst)
;;
;; Parameters:
;;    lst - A list of lists representing different fault classes.
;;
;; -- GET-COMMON-NODES returns a list of those items common to all
;;    of the top-level sublists.  If there is only one top-level sublist,
;;    then it is returned.  If there are more than one, then
;;  GET-COMMON-NODES-1 is called and passed both the FIRST-LST as well as
;;    the REST of the sublists.

(define (get-common-nodes lst)
   (let ( (first-lst (car lst))
          (rest      (cdr lst)) )
      (if (null? rest)
          first-lst
          (get-common-nodes-1 first-lst rest) )))



;; (GET-COMMON-NODES-1 first-lst list-of-lists)
;;
;; Parameters:
;;    first-lst - One of the fault cases.
;;    list-of-lists - All of the remaining fault cases.
;;
;; -- GET-COMMON-NODES-1 works by taking each element of the FIRST-LST
;;    and checks to see if an element is the MEMBER-ALL-LISTS? of each
;;    of the other lists of faults.  If it is, then that element is
;;    common to all of the fault cases.
;; --If an element is not a MEMBER-ALL-LISTS?, then it is not common to
```

```
;;    all of the fault cases.  Only those elements that are common to all
;;    of the fault cases are returned.
;;--GET-COMMON-NODES-1 calls itself recursively until all of the elements
;;    of FIRST-LST in the initial call to GET-COMMON-NODES-1 have been
;;    checked with respect to the other lists.

(define (get-common-nodes-1 first-lst list-of-lists)
   (if (null? first-lst)
       '()
       (let* ( (first-elt (car first-lst))
               (rest      (cdr first-lst)) )
          (if (member-all-lists? first-elt list-of-lists)
              (cons first-elt
                    (get-common-nodes-1 rest list-of-lists))
              (get-common-nodes-1 rest list-of-lists)))))


;; (MEMBER-ALL-LISTS? elt list-of-lists)
;;
;; Parameters:
;;    elt - an arbitrary element
;;    list-of-lists - an arbitrary list of lists
;;
;; -- MEMBER-ALL-LISTS? works by determining whether the element is a
;;    member of the first list in the LIST-OF-LISTS.  If it is, then
;;    MEMBER-ALL-LISTS? calls itself recursively.  If it calls itself
;;    until LIST-OF-LISTS is exhausted, then ELT had to be a member of
;;    all of the sublists in LIST-OF-LISTS.

(define (member-all-lists? elt list-of-lists)
   (if (null? list-of-lists)
       T
       (if (member elt (car list-of-lists))
           (member-all-lists? elt (cdr list-of-lists))
           '())))


;; (REMOVE-COMMON-NODES lst list-of-lists)
;;
;; Parameters:
;;    lst - A list of elements common to each sublist of LIST-OF-LISTS
;;          that are to be removed from LIST-OF-LISTS.
;;    list-of-lists - An arbitrary list of lists.
;;
;; -- REMOVE-COMMON-NODES removes all of the elements of LST from each of
;;    the top-level sublists of LIST-OF-LISTS.  The first element of LST
```

```
;;    is DIVIDEd into LIST-OF-LISTS to form a new list of lists.
;; -- This new list, with the remaining elements of LST are then passed
;;    to a recursive call of REMOVE-COMMON-NODES.  This continues until
;;    the elements of LST have been exhausted.

(define (remove-common-nodes lst list-of-lists)
   (if (null? lst)
       list-of-lists
       (remove-common-nodes (cdr lst) (divide list-of-lists (car lst)))))


;; (REMOVE-LAST-CHAR-FROM-NODE-SYMBOLS lst)
;;
;; Parameters:
;;    lst - a list of the form:
;;
;;       ((EQ E-- (NOT (* AX- BO-)))
;;        (EQ F-- (NOT B1-))
;;        (EQ Z-- (NOT (* E-- F--))))
;;
;; -- REMOVE-LAST-CHAR-FROM-NODE-SYMBOLS takes the LST and removes the last
;;    character from each of the node symbols.  This leaves a list of the
;;    form:
;;
;;       ((EQ E- (NOT (* AX BO)))
;;        (EQ F- (NOT B1))
;;        (EQ Z- (NOT (* E- F-))))
;;
;; -- The LST is decomposed recursively until a symbol is reached.  Then if
;;    a node symbol is detected, the last character is removed.  Otherwise,
;;    the symbol is unchanged.  The returned list is the original list
;;    reassembled with the last character removed from each of the node
;;    symbols.

(define (remove-last-char-from-node-symbols lst)
   (cond ( (null? lst)
           '())
         ( (symbol? lst)
           (if (good-symbol? lst)
               (remove-last-char-from-symbol lst)
               lst) )
         ( (symbol? (car lst))
           (if (good-symbol? (car lst))
               (cons (remove-last-char-from-symbol (car lst))
                     (remove-last-char-from-node-symbols (cdr lst)))
```

```
      (cons (car lst)
         (remove-last-char-from-node-symbols (cdr lst))) )
( else
  (cons (remove-last-char-from-node-symbols (car lst))
        (remove-last-char-from-node-symbols (cdr lst))) )))
```

Vita

Captain Reginald H. Gilyard [redacted] After graduating from Eisenhower H.S. in June 1981 he attended the U.S. Air Force Academy at Colorado Springs. Upon graduation from the Academy he was commissioned a Second Lieutenant in the U.S. Air Force and awarded a B.S. degree. Captain Gilyard went on to complete a tour as an acquisition officer at Norton AFB, CA prior to attending the Air Force Institute of Technology.

[redacted]

Bibliography

1. Vishwani D. Agrawal and Sharad C. Seth. *Test Generation For VLSI Chips*. IEEE Computer Society Press, Washington, D. C., 1981.
2. Sami A. Al-Arian and Dharma P. Agrawal. Physical failures and fault models of cmos circuits. In *IEEE Transactions on Circuits and Systems*, pages 269–279, 1987.
3. Archie Blake. *Canonical Expressions in Boolean Algebra*. PhD thesis, University of Chicago, Chicago, Illinois, 1937.
4. George Boole. *An Investigation of the Laws of Thought*. Macmillan, London, 1854.
5. Douglas C. Bossen and Se Jung Hong. Cause-effect analysis for multiple fault detection. In *IEEE Transactions on Computers*, pages 1252–1257, 1971.
6. F. M. Brown. Boolean reasoning with applications in logical design. Unpublished Textbook, 1989.
7. E. Cerny. Application of a boolean-equation-based methodology to the detection of faults in combinational switching circuits. Unpublished Report R76-84, IEEE Computer Repository, 1976.
8. Kuang-Wei Chiang and Zvonko G. Vranesic. Test generation for MOS complex gate networks. In *Fault Tolerant Computer Symposium Digest of Papers*, pages 149–157, 1982.
9. Kuang-Wei Chiang and Zvonko G. Vranesic. On fault detection in CMOS logic networks. In *IEEE 20th Design Automation Conference*, pages 50–56, 1983.
10. R. I. Damper and N. Burgess. MOS test pattern generation using path algebras. In *IEEE Transactions on Computers*, pages 1123–1128, 1987.
11. Robert J. Feugate and Steven M. McIntyre. *Introduction to VLSI Testing*. Prentice Hall, Englewood Cliffs, NJ, 1988.
12. Hideo Fujiwara. *Logic Testing and Design for Testability*. MIT Press, Cambridge, Massachusetts, 1985.
13. Jr. H. Troy Nagle et al. *An Introduction to Computer Logic*. Prentice-Hall, Englewood Cliffs, 1975.
14. E.V. Huntington. Sets of independent postulates for the algebra of logic. In *Transactions of the American Mathematical Society*, pages 288–309, 1904.
15. Sunil K. Jain and Vishwani D. Agrawal. Test generation for mos circuits using d-algorithm. In *IEEE 20th Design Automation Conference*, pages 64–70, 1983.
16. Barry W. Johnson. *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.
17. E.L. Johnson and M.A. Karim. Digital design: A pragmatic approach. PWS Engineering, Boston, 1987.

18. Capt James J. Kainec. A diagnostic system using boolean reasoning. Master's thesis, AFIT/ENG, Wright-Patterson AFB OH, December 1988.

19. Parag K. Lala. *Fault Tolerant and Fault Testable Hardware Design.* Prentice-Hall International, Inc., London, 1985.

20. Seymour Lipschutz. *Discrete Mathematics.* McGraw-Hill, New York, 1976.

21. M. Morris Mano. *Digital Logic and Computer Design.* Prentice-Hall, Inc., Englewood Cliffs N.J., 1979.

22. O.H. Mitchell. On a new algebra of logic. In C.S. Peirce, editor, *Studies in Logic.* Little, Brown, Boston, 1883.

23. Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, New York, 1987.

24. W.V. Quine. The problem of simplifying truth functions. *American Mathematical Monthly*, 59:521-531, October 1952.

25. W.V. Quine. A way to simplify truth functions. *American Mathematical Monthly*, 62:627-631, November 1955.

26. Hassan K. Reghbati. *Tutorial: VLSI Testing and Validation Techniques.* IEEE Computer Society Press, Washington, D. C., 1985.

27. Scott H. Robinson and John P. Shen. Towards a switch level test pattern generation program. In *International Conference on Computer Aided Design*, pages 39-41, 1985.

28. Sergiu Rudeanu. *Boolean Functions and Equations.* North Holland, Amsterdam, 1974.

29. E.W. Samson and B.E. Mills. *Circuit Minimization: Algebra and Algorithms for New Boolean Canonical Expressions.* Air Force Cambridge Research Center, Cambridge, Massachusetts, 1954.

30. Donald F. Stanat and David F. McAllister. *Discrete Mathematics in Computer Science.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1977.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution Unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/89D-4	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology (AU) Wright-Patterson AFB, Ohio 45433-6533	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Microelectronics Division, Design Branch 8b. OFFICE SYMBOL (If applicable) WRDC/ELED	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB, OH 45433	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)
Boolean Approaches In Digital Diagnosis (UNCLASSIFIED)

12. PERSONAL AUTHOR(S)
Reginald H. Gilyard, Captain, US Air Force

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	891204	245

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Boolean Algebra Fault Diagnosis
09	02		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Advisor: DR. FRANK BROWN, PhD
Professor, Department of
Electrical and Computer Engineering

(see reverse)

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Dr. Frank Brown, PhD	22b TELEPHONE (Include Area Code) 22c OFFICE SYMBOL AFIT/ENG

Abstract

The goal of this thesis is to review and improve two existing methods that use Boolean reasoning as a basis for testing digital circuits. Extensions are made to research done by both Cerny and Kainec in this area.

The method developed by Cerny to generate test vectors capable of detecting single stuck-at, bridge and multiple stuck-at faults is reviewed and then extended in two ways. The first extension incorporates the capability to automatically analyze the results gained from applying a given vector. The second extension allows the diagnosis of sequential circuits. Since Cerny's original method was not automated the entire process is updated to include the extensions and then programmed.

Kainec developed an automated diagnostic system to test for multiple faults in combinational circuits. The original system is restricted to diagnosing faults in circuits with one output. An extension is designed and programmed to incorporate the capability to diagnose multiple output circuits. The extension shows that multiple output circuits offer the added advantage of being able to choose an optimal test vector from a set of genrated vectors, thereby shortening the required testing time for a given circuit.

The software routines are programmed in PC-Scheme (a dialect of LISP) on an IBM microcomputer. Due to a conversion program written by Kainec the software can also be run on a Sun-4 workstation in the T environment. T is derived from Scheme.