A Fault Independent Test Generation Method For Combinational Logic Circuits

by

Mohamed Mahdy Al-Deeb

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

INFORMATION AND COMPUTER SCIENCE

June, 1992

A fault independent test generation method for combinational logic circuits

Al-Deeb, Mohamed Mahdy, M.S.

King Fahd University of Petroleum and Minerals (Saudi Arabia), 1992

A FAULT INDEPENDENT TEST GENERATION METHOD FOR COMBINATIONAL LOGIC CIRCUITS

BY

MOHAMED MAHDY AL-DEEB

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

In

INFORMATION AND COMPUTER SCIENCE

JUNE 1992

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by ***MOHAMED MAHDY AL-DEEB*** under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of ***MASTER OF SCIENCE*** in ***INFORMATION AND COMPUTER SCIENCE.***

THESIS COMMITTEE

[illegible]
Dr. Bassel R. Arafeh (Chairman)

[illegible]
Dr. Mohamed Y. Osman (Co-Chairman)

[illegible]
Dr. Mamdouh M. Najjar (Member)

[illegible]
Dr. Muhammad Shafique (Member)

[illegible]
Dr. Muhammad A. Al-Tayyeb
Department Chairman

[illegible]
Dr. Ala H. Al-Rabeh
Dean, College of Graduate Studies

To My Beloved Parents, Sisters

and to Those Who Shared Their Care and Concern

ACKNOWLEDGEMENT

First of all, I thank " *ALLAH* " ALMIGHTY for all the knowledge and science *HE* has given to mankind including myself. Thanks are also to our Prophet *MOHAMMED, PEACE BE UPON HIM*, who encourages us as Moslems to seek science wherever it can be found.

Acknowledgement is due *King Fahd University of Petroleum and Minerals* for giving me this opportunity to achieve this work through its tremendous facilities.

I would like to express my deep appreciation and gratitude to my thesis advisor and co-advisor, *Dr. Bassel R. Arafeh* and *Prof. Mohamed Y. Osman* for their patient guidance and their generous support and encouragement.

I would like to express my gratefulness to the other committee members, *Dr. Mamdouh M. Najjar* and *Dr. Muhammad Shafique* for their valuable suggestions, helpful remarks and their kind co-operation.

Special thanks are to *Brothers*, *Friends* and many others for the memorable days we shared together.

Finally, I would like to send my sincere gratitude to my father, mother and sisters whose constant encouragement made the completion of this work possible.

TABLE OF CONTENTS

THESIS ABSTRACT

Name	: **Mohamed Mahdy Al-Deeb**
Title	: **A FAULT INDEPENDENT TEST GENERATION METHOD FOR COMBINATIONAL LOGIC CIRCUITS**
Major Field	: **Information and Computer Science**
Date	: **June 1992**

In order to improve the performance of fault independent test generation algorithms for VLSI combinational logic circuits, two new strategies are proposed: Critical Lines Maximization strategy (CLM) and Critical Primary inputs Flipping strategy (CPF). CLM is used to maximize the number of detected faults while generating a test pattern. CPF is used to derive new test pattern(s) from a generated test pattern with little additional effort.

In this thesis, a new backtrace procedure, called CLM-Multiple-Backtrace, based on the CLM strategy as well as multiple backtrace procedure of FAN is introduced. A new fault independent test generation algorithm, called MAX, based on these new strategies as well as many other efficient strategies and procedures of the existing test generation algorithms is introduced and illustrated with examples. Experimental results show that MAX is more efficient than the fault independent test generation algorithms given in the literature.

MASTER OF SCIENCE

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN, SAUDI ARABIA

June 1992

خلاصة الرسالة

إسم الطالب :- **محمد مهدي الديب.**

عنوان الرساله :- **طريقة مستقلة عن الأخطاء لتوليد اختبارات الكشف عن الأخطاء في الدوائر المنطقيه التوافقية.**

التخصص :- **علم الحاسب الآلي والمعلومات.**

تاريخ الشهادة :- **يونيو ١٩٩٢م.**

لتحسين اداء الخوارزميات المستقله عن الأخطاء والتي تستخدم لتوليد اختبارات الكشف عن الأخطاء في الدوائر المنطقيه التوافقية. فقد تم استحداث استراتجيتين جديدتين : تستخدم الاستراتجية الأولى للإكثار من عدد الأخطاء التي يمكن الكشف عنها اثناء توليد إختباراً واحداً ، اما الاستراتجيه الثانية فتستخدم لإشتقاق اختبارات اخرى جديدة من الإختبار المولد بإستخدام الإستراتجية الأولى.

في هذه الرساله ، تم استحداث طريقة تراجعية جديده تجمع بين الإستراتجية الأولى المذكورة اعلاه والطريقة التراجعية المستخدمة في خوارزمية "فان" لتوليد اختبارات الكشف عن الأخطاء . في هذه الرساله ايضاً ، تم استحداث خوارزميه مستقلة عن الأخطاء لتوليد اختبارات الكشف تدعى "ماكس" تجمع بين الإستراتجيات المستحدثه اعلاه وعدد من الإستراتجيات المستخدمه من قبل بعض خوارزميات توليد الإختبارات المستقله وخوارزميات توليد الإختبارات المعتمدة على الأخطاء. أخيراً فإن النتائج العمليه أثبتت أن خوارزمية "ماكس" أسرع من الخوارزميات الأخرى لتوليد الإختبارات المستقلة عن الأخطاء التي تم استحداثها من قبل.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

الظهران - المملكة العربية السعودية

يونيو - ١٩٩٢م

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In this chapter, we give a brief introduction to general fault testing schemes for digital circuits which form the basis for the material discussed in this thesis.

1.1 Digital Testing

The basic function of testing is to verify whether the circuit under test (CUT) is functioning as specified or not. This is done by applying a set of input patterns and observing the output response of the CUT (see Figure 1.1).

1.1.1 Test Pattern Generation

The field of testing developed into two different approaches, the exhaustive approach and the deterministic approach. In the exhaustive approach, every possible input combination is applied to the circuit under test. This gurantess that all the possible fault are detected. With the progress of VLSI technology, the problem of testing logic circuits is becoming more and more difficult. As a result, the exhaustive approach became infeasible and the deterministic approach emerged.

The deterministic approach naturally restricts the possible faults to a manageable size and solve the problem of detecting this set of faults. The main concern here is which inputs should be applied such that the effect of the fault would occur at the output. The set of faults may be anything from the physical structure of the CUT, such as stuck-at faults and bridging faults, or from the function of the CUT, such as next state-faults and output faults, or from the timing of the CUT, such as delay faults and transient faults.

In the deterministic approach, deriving tests for combinational circuits differ from that for sequential circuits. Each kind of circuit, however, has two

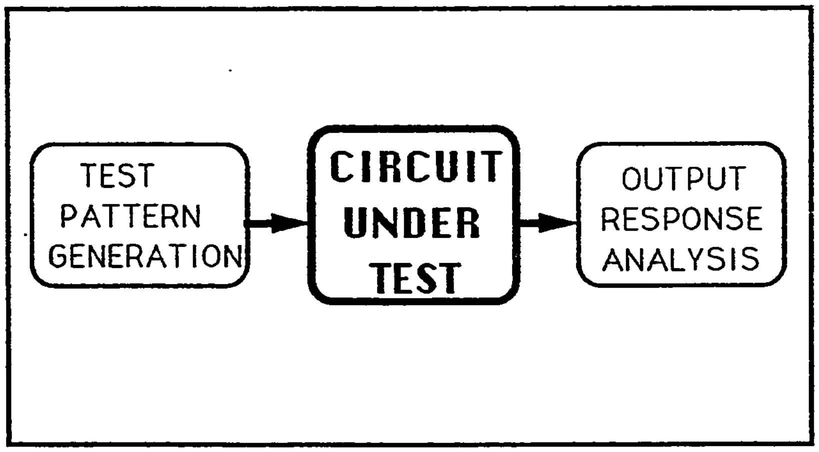


Figure 1.1 Digital System Testing

major approaches for deriving tests, the algebraic approach, which tests from the knowledge of the truth table or state table, and the topological approach, which requires the knowledge of the structure of the CUT.

For combinational circuits, the boolean difference method [13] represents the algebraic method. This is usually manageable for small number of inputs and outputs only. On the other hand, methods that rely on path sensitization represent the topological approach. Topological oriented methods are classified into two types: fault independent and fault oriented test generation methods. Fault independent test generation methods are used to provide a set of tests for a large percentage of the faults of the circuit under test. Example methods include RANDOM [35], RAPS [36], SMART [12]. Fault oriented test generation methods are used to generate tests for specific faults. Some efficient methods have been developed such as PODEM [27], FAN [28], FAST [12]. All these methods shall be presented in Chapter II.

For sequential circuits, checking experiments represent the algebraic approach, while methods such as the extended D-algorithm [46] represent the other approach.

1.1.2 Output Response Analysis

All test pattern generation methods for combinational and sequential circuits would require bit-by-bit comparisons of observed output values with the correct values. This requires a significant amount of memory for saving the correct outputs associated with all test vectors which caused the compression approach to emerge. In this approach the information is saved in a compressed form for the expected test outcome, called a signature. The circuit is then tested by comparing the expected signature with the computed signature. The process of reducing the complete output response to a signature is referred to as output response compression. This approach is simpler and requires less memory storage. The concept is illustrated in Figure 1.2.

Some of the well known compression techniques include transition counting [47], syndrome checking [48], Walsh coefficient [49] , and signature analysis [50]. Unfortunately, all compression techniques have a probability that an erroneous output pattern may have the same signature as the correct output pattern. This is called aliasing.

1.1.3 Design for Testability (DFT)

Previous work [1] has established that the problem of test generation is usually more difficult for sequential circuits than for combinational ones. Testing of sequential circuits can be made comparable to those of testing combinational ones by using certain techniques known as design for testability (DFT) such as LSSD (Level Sensitive Scan Design) and scan path architecture [2,46]. In DFT techniques, additional hardware is added to the original design of the circuit such that the resulting circuit is easily tested.

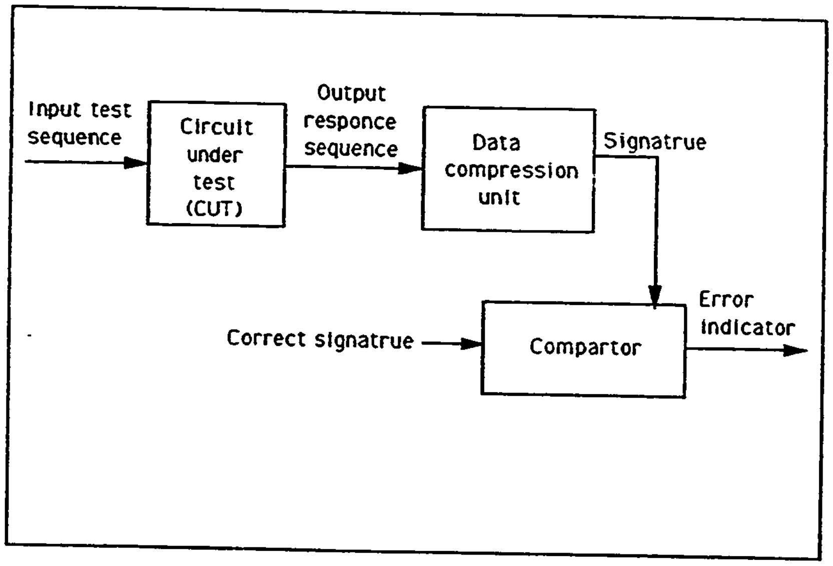


Figure 1.2 Testing Using Test Response Compression

1.2 Thesis Overview

In this thesis, a new efficient fault independent test generation algorithm is developed for combinational logic circuits. In order to develop the new algorithm, some efficient test generation features (i.e, concepts, strategies and procedures) given in the literature as well as our new features are introduced and used. So the developed algorithm (called MAX) is proposed based on all these features. For proving that MAX is more efficient than the other existing fault independent test generation algorithms, both MAX and SMART [12], the most efficient fault independent test generation algorithm in the literature upto date are implemented and run for bench mark test circuits. Experimental results of different runs show that MAX is more efficient than SMART, as it provides test sets in a shorter time.

1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapter II surveys test pattern generation algorithms for combinational logic circuits. Chapter III provides a detailed description of some of the existing test generation features given in the literature. Chapter IV provides a detailed description of our new proposed test generation features. Chapter V outlines the new proposed fault independent test generation algorithm (called MAX) and analyzes a complete circuit example. Chapter VI provides some discussions about the performance of MAX compared to the performance of SMART [12]. Finally, the conclusions are given in Chapter VII.

CHAPTER II

LITERATURE SURVEY

In this chapter, we give a literature survey on fault test generation algorithms for digital circuits. A short background on fault models is also provided. The material presented in this chapter forms the basis for our discussion in later chapters.

2.1 Fault Models

A fault model is a representation of the effect of failures that produce changes in the signal values of the logic circuit. A large variety of models [3-7] are used in digital circuit testing.

The most common fault model used in digital circuit testing is the single stuck-at model [3]. Each circuit node is assumed to have two potential faults: permanently stuck at either logic value "1" (s-a-1) or logic value "0" (s-a-0). Furthermore, at most one fault is assumed to exist in a circuit at any given time. More complex fault models have also been proposed, such as the multiple stuck-at fault model [4] and the bridging fault model [3]. In the multiple stuck-at fault model it is assumed that two or more stuck-at faults may exist at the same time. The bridging fault model assumes that signal lines are connected together so that wired logic occurs.

Another variation of the stuck-at fault model is the unidirectional fault model. It assumes that one or more stuck-at faults may be present, but all the stuck lines have the same logical value. It is used in modeling faults in storage media.

Another fault model, called the pattern-sensitive fault model [5] was developed for RAMs and high density dynamic MOS chips. In this model, the effect of the fault depends on the particular input applied to the chip.

Results [6,7] has shown that test patterns that can detect all single stuck-at faults in a circuit, can also detect many multiple stuck-at faults and bridging faults. All testing methods surveyed in this thesis assume only single stuck-at faults. Based on that, the type of fault model used in the proposed work will be the single stuck at fault model.

2.2 Combinational Logic Circuits Testing

2.2.1 Automatic Test Pattern Generation (ATPG) Systems

Many ATPG systems for combinational logic circuits have been proposed earlier such as LASAR [8], PODEM-X [9], MAHJONG [11], and LAMP2 [12]. These systems usually couple two distinct phases as follows:

***Phase I*:**

In the first phase, a low-cost fault independent method is used to provide a set of tests for a large percentage of the faults. Examples include RANDOM [35], RAPS [36], and SMART [12].

***Phase II*:**

In the second phase, a fault oriented method targets the remaining undetected faults. Suitable methods for this phase include D-Algorithm [22], PODEM [27], FAN [28], FAST [12], TOPS [29], and SLOPE [30].

2.3.2 *Test Pattern Generation Algorithms*

Test pattern generation algorithms can be classified into three main categories:

A) Exhaustive Testing.

B) Fault-Oriented Testing.

C) Fault-Independent Testing.

A. Exhaustive Testing

In the exhaustive testing technique, every possible input combination is applied to the combinational circuit under test. It is feasible only for small circuits, since a large number of test patterns may be required.

McCluskey [10] introduced an important observation that if the combinational circuit under test is first partitioned into subcircuits with a sufficiently small number of inputs, then exhaustive testing will become feasible.

B. Fault-Oriented Testing (FOT)

In this category, the testing methods [12-33] enumerate a target list of faults. For each detectable fault in the list, a test pattern to detect the fault is generated. Below is an account of the major approaches to FOT.

B.1 Algebraic Algorithms

The major characteristic of algebraic methods is generating equations for the fault free circuit and manipulating these equations to generate tests. Several algebraic test generation methods have been developed. These include the boolean difference method of Sellers [13], the prepositional method of Poage [14], the equivalent normal form of Armstrong [15], the cause-effect equation of Bossen and Hong [16], the SPOOF procedure of Clegg [17], and the structure description function of Kinoshita [18]. However, because these methods generate all possible tests for the given fault, they have the disadvantage of requiring a large quantity of computing time and memory, which make them impractical for large circuits.

One disadvantage of the boolean difference method is the difficulty of manipulating algebraic equations. In [19] a tabular method, for generating the boolean difference, suitable for programming has been introduced.

B.2 Topological Search Algorithms

In this sub-category, all the testing methods involve fault activation, fault effect propagation and line justification using a backtracing mechanism.

The single-path sensitization (SPS) method is an extension of Eldred's results [20] and was formulated by Armstrong [15] and Chang [21]. It consists of three basic steps: activation, propagation, and justification. To detect the fault line L s-a-0 (s-a-1), a suitable input pattern that makes L take the opposite value 1 (0) must be found. This process activates the fault. Then the effect of the fault must be made observable at a circuit primary output (PO). This can be done by sensitizing a path from L to a PO. The inputs to the gates along the chosen path must be assigned to values that propagate the fault effect to the output. This process is called the propagation phase of the method. To complete the test, values on the gates inputs needed to sensitize the path are driven back to corresponding values on the circuit inputs. This process is called the line justification phase of the method. However, the method is not complete, since there can exist testable faults in a circuit for which it is impossible to generate tests if only one path is allowed to be sensitized at a time.

The multiple-path sensitization approach, called the D-Algorithm, was formulated by Roth [22,23]. The D-algorithm is formalized in terms of cubical algebra called the D-calculus. Basically, the D-algorithm proceeds in two stages: first the D-drive operation whose objective is to propagate the effect of the fault to a primary output. Then line justification (consistency) operation is performed to justify each internal line that got an assignment in the D-drive operation. Cha [24], Akers [25], and Takamatsu [26] worked on the same line extending the algorithm to a larger D-calculus set. The D-algorithm and its variations are complete algorithms in that they will generate a test for the given fault if a test exists.

A more attractive complete method called PODEM (Path Oriented DEcision Making) was reported by Goel [27]. PODEM has been shown to be more efficient than the D-algorithm especially for error correction and translation (ECAT) circuits. Goel formulated the test generation problem as a search problem of the n-dimensional 0-1 state space of primary input patterns of an n-input combinational logic circuit. Like the D-algorithm, PODEM performs fault propagation operation that is similar to the D-Drive operation of the D-algorithm. But unlike the D-algorithm, PODEM does not perform line justification process. It avoids that by allowing line assignment to primary inputs only, rather than internal lines.

Fujiwara [28] developed a more accelerated complete method called FAN (fan-out-oriented). In FAN, several strategies were introduced such as unique implication, unique sensitization, and multiple backtrace. These strategies try to decrease the number of backtracks. FAN also extended the backtracing concept of PODEM. It allows the backtracing to stop at certain lines called head lines, rather than primary inputs. The authors also introduced an automatic test generation system composed of FAN and a concurrent fault simulator [34]. The fault simulator is run after each test pattern is generated to find what other faults are detected by that test pattern.

Abramovici et al [12] introduced a testing method, called FAST (Fault-Oriented Algorithm for Sensitized-path Testing). It combines and extends features of PODEM [27] and FAN [28]. It also uses controllability/observability cost functions. These functions guide FAST toward decisions less likely to cause conflicts. They used to identify lines whose justification are conflict free.

TOPS (TOPological Search) algorithm was proposed by Kirkland et al [29]. It uses small size search space and improves the ordering of node assignments based on a topological analysis of the circuit under test. Also, it rapidly identifies many redundant faults without search.

Chuang et al [30] presented a test pattern generator called SLOPE (Stop Line Oriented Path End). It combines the advantages of FAN [28] and FAST [12] and utilize a controllability measure and an observability measure. The two measures are used to assist in guiding the test generation process.

A further improvement to the test generation process has been proposed by Lioy [31], where an adaptive backtrace technique is introduced. It avoids unnecessary internal conflicts of previous methods and reduces the number of backtracks while searching for a test.

A distributive D-Algorithm was proposed by Lo et al [32]. It improves the performance of the conventional D-Algorithm. The fault effect is propagated from its location to the primary output through all possible paths. The goal of this process is to reduce the number of backtracks during the test generation process.

Recently, Patil and Banerjee [33] identified the problems inherent in a uniprocessor implementation of a test generation method and proposed a parallel test generation method. It tries to achieve a high fault coverage for hard-to-detect faults in a reasonable amount of time. To do that, a dynamic search space allocation strategy is used. It ensures that the search spaces allocated to different processors are disjoint.

C. Fault-Independent Testing (FIT)

Fault independent testing [12,35,36] is usually used in phase I of a test pattern generation system. Unlike FOT, where the goal is to generate a test pattern for a given fault, FIT objective is to generate an initial test set that covers as many faults as possible of the list of all faults before passing the CUT to phase II.

Fault-independent testing methods can be classified into the following categories: Random Testing and Semi-Deterministic Testing. Since these are the basis of our proposed algorithm a brief discussion of them follows.

C.1 Random Testing

A random test generation method called RANDOM was reported by Schnurmann [35]. The basic idea of RANDOM is to generate a test pattern at random and simulate it to find out which faults are tested. It keeps doing that until a reasonable fault coverage is achieved or a specified time limit is exceeded.

C.2 Semi-Deterministic Testing

Geol [36] introduced a testing method called RAPS (Random Path Sensitization). It combines features of random and deterministic test generation algorithms. Central to RAPS is the concept of objective, (m, v_m), that is setting line m to a desired value v_m. For the circuit under test, RAPS selects a primary output (PO) at random and assigns it a random logic value v_{PO}. Procedure Justify, shown in Figure 2.1, is used to justify the initial objective (PO, v_{PO}) by repeatedly finding a primary input (PI) assignment that is likely to contribute to setting PO to v_{PO}. The mapping of an objective into a PI assignment is recursively done by procedure Backtrace shown in Figure 2.2. Internal line values are generated only by simulating primary inputs assignments. After a test pattern is generated, a fault simulator is invoked to determine the detected faults. RAPS keeps generating test patterns until the incremental number of the new detected faults become very small. For most of logic circuits, RAPS has been shown to be more efficient than RANDOM [36].

```
Justify ($m$, $v_m$ )
 /* Justify attempts to assign the logic value $v_m$ for line m */
{ Repeat
      { ($i$, $v_i$ ) = Backtrace ($m$, $v_m$)
         /* ($i$, $v_i$ ) represents a value $v_i$ of primary input $i$ */
        Simulate ($i$, $v_i$ )
        /* True value simulation */
      }
  Until m has binary value
  /* At this point the assigned value of m is justified */
}
```

Figure 2.1 Justify Procedure

```
Backtrace ( $m$, $v_m$ )
/* Backtarce returns an assignment to a PI line */
{ If ( $m$ is a PI ) Return ( $m$, $v_m$ )
  If ( $m$ is a fanout branch of $j$ ) Return Backtrace ( $j$, $v_m$)
  $i$ = inversion of gate $Q$ that fed line $m$
  Select arbitrarily an unassigned input $j$ of $Q$
  Return Backtrace ( $j$, $v_m \oplus i$ )
}
```

Figure 2.2 RAPS Backtarce Procedure

Abramovici et al [12] presented a testing method called SMART (Sensitizing Method for Algorithmic Random Testing). It combines features of random and deterministic methods. In SMART many concepts are proposed such as critical lines, stop lines, useful lines and restart gates. All these concepts are explained below. Stop lines delimit areas of the circuit under test where additional fault coverage cannot be obtained, while restart gates delimit areas where additional fault coverage is likely to be obtained with little additional effort.

A line L with value v in a test t is ***critical*** if t detects the fault L stuck at $\bar{v}$ (i.e., L s-a- $\bar{v}$).

A line L is a ***v-stop*** line for a test set T, if T detects all the detectable faults that cause L to take value $\bar{v}$.

In SMART, stop lines are determined according to the following rules:

a) A primary input is a v-stop if it has had a critical value v.

b) The output of a gate with inversion i is a v-stop if it has had a critical value v and all the gate inputs are (i $\oplus$ v)-stops.

c) a fanout branch is a v-stop if it has had a critical value v and its stem is a v-stop.

A line L that is not v-stop is said to be ***v-useful.***

A gate G in a test t is a ***restart gate*** if G satisfies the following conditions:

a) Its output is critical (in t) but none of its inputs is critical.

b) Only one input has the controlling value c (e.g., 0 for an AND gate) and this input is not a c-stop line.

Like RAPS, SMART (see Figure 2.3) starts by justifying a "randomly" chosen value for a "randomly" chosen primary output (PO). But unlike RAPS, whenever an objective (k, v_k) for a gate output is mapped into an objective (j, v_j) for a gate input, SMART uses a Selective-backtrace procedure. It differs from Backtrace of Figure 2.2 in giving preference to the gate inputs that are *v_j-useful* lines (i.e., that are not *v_j-stop* lines) rather than selecting inputs arbitrarily. This way allows detection of new faults and avoids generating repeated test patterns. In the justification process, SMART uses a procedure, called Selective-justify, that is similar to Justify shown in Figure 2.1, except that it uses Selective-backtrace instead of Backtrace. After the PO is driven to a binary value, SMART invokes the Critical Path Tracing (CPT) [38] fault simulator for the partially generated vector. CPT determines critical lines and restart gates for the partially generated vector. Then SMART repeatedly picks a restart gate and tries to justify non-controlling values on its inputs with value x and reruns CPT. This close interaction with CPT greatly contributes to the efficiency of SMART. After a fully generated vector is obtained, CPT determines stop lines based on the above rules. SMART stops generating tests, if the average number of new faults detected by the last n tests is less than 2 (mostly n=30). Finally, results showed that SMART is more efficient than RAPS [36], as it generates smaller test sets in shorter time.

```
SMART( )

{While (Useful POs have X values)
    {Randomly select a useful PO Z with X value
     If (Z is both 0- and 1- useful) randomly select a value v
     Else v = the useful value of Z
     Selective-justify (Z,v)
     /* At this point a partial test vector p is generated */
     CPT( )
      /* CPT( ) determines critical lines and restart gates for p */
     While (The set of Restart-gates ≠ Φ )
         { Remove a gate G from Restart_gates
           c = controlling value of G
           for every input j of G with X value
               Justify ( j, $\bar{c}$ )
           CPT( )
         }
    }
Return vector of PIs value
}
```

Figure 2.3 SMART--A Fault Independent Test Generation Algorithm

CHAPTER III

RELEVANT CONCEPTS AND PROCEDURES

In this chapter, we shall provide a detailed description to a number of relevent concepts and procedures of the existing test generation algorithms. These concepts and procedures will be incoperated into the new proposed fault independent test generation algorithm. Each one of these concepts and procedures will be explained and illustrated by examples. This chapter is orgnized as follows. Section 3.1 presents circuit leveling procedure of Breuer (Circuit_leveling) [41]. Section 3.2 presents controllability measure of FAST (Compute_controllability) [12]. Section 3.3 presents Multiple Backtrace of FAN [28]. Section 3.4 presents Critical Path Tracing Fault Simulator (CPT) [38]. Section 3.5 presents Test Generation Stopping Function of SMART (TGS) [12].

3.1 Circuit Leveling Procedure of Breuer (Circuit_Leveling)

Aim:

Given a circuit to be tested. The aim of this procedure [41] is to assign a certain level number, which will be explained later in this section, to each line of the circuit under test (CUT) based on its position in the circuit. While a test is generated, these level numbers will be used as a guidance measure for the lines selection during the backtracing process.

Description:

The level of a line is recursively defined as follows:

1) The level of a primary input is 0.

2) The level of a gate output is $1 + i_{\max}$ where $i_{\max}$ is the highest level among the levels of the gate inputs.

3) The level of fanout branches is identical to the level of its stem.

Example 3.1

Consider the circuit of Figure 3.1, the result of leveling is given below:

Level 0:	A,B,G,H,I,L
Level 1:	J,K
Level 2:	M,N
Level 3:	Z

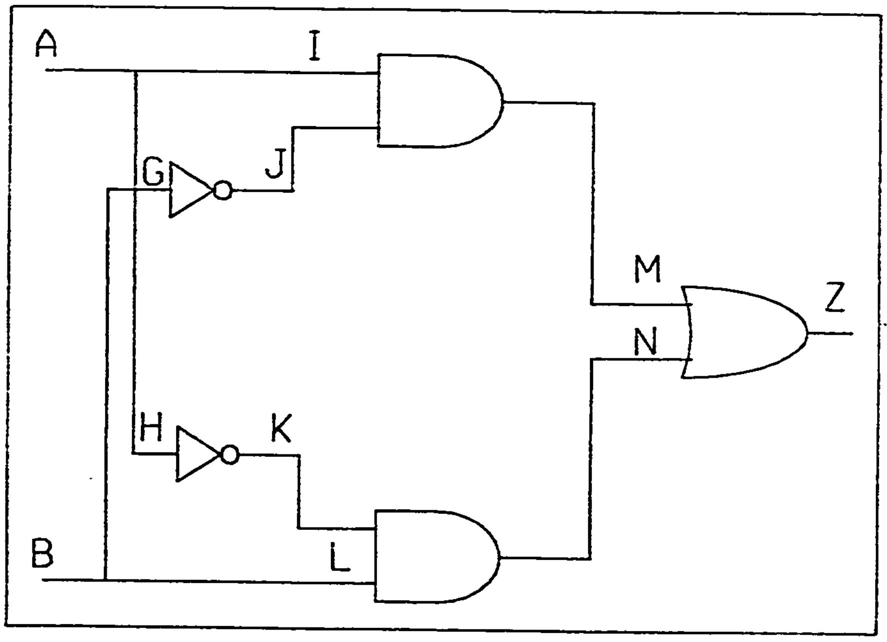


Figure 3.1 Circuit Leveling

3.2 Controllability Measure of FAST (Compute_controllability)

Aim:

Given a circuit to be tested. The aim of using this measure [12] is to provide a quantitative measure of the difficulty of controlling the logic values of internal lines from the topology of the CUT. In other words, it is used to help in making suitable choices in justifying the logic values to the lines of the CUT, hence minimize the required time for the backtracing process.

Description:

For guiding the backtracing process, we need cost functions that measure the difficulty of justifying a logic value to a given line. Typically, the major cost function is the controllability cost function. Controllability cost indicates the relative difficulty of setting a line to a value.

The main problem is to define difficulty in a meaningful way for test generation. If we analyze the execution of a test generation algorithm, we observe that its worst-case behavior is characterized by many wrong decisions from which the algorithm recovers using backtracking. A decision is incorrect if it leads to conflicts (inconsistencies). On the other hand, the best case occurs when the execution completes without backtracking. Hence the objective of cost functions should be to minimize the amount of time required for backtracing by guiding the algorithm toward a decision which is less likely to cause conflicts. The potential for conflicts is directly related to the fanout structure of the circuit and not to its size. For example, no conflicts can occur in any fanout-free circuit or in any circuit without reconvergent fanout.

In FAST algorithm, two controllability cost functions are used, C0(m) and C1(m), to reflect the relative potential for conflicts resulting from trying to justify 0 and 1 value for line m, respectively. These costs functions are derived from those used in [37] and are computed as follows:

1) If m is a primary input

$$C0(m) = C1(m) = 0$$

2) If m is the output of an AND gate

$$C0(m) = \text{MIN}\,\{C0(i)\}$$

$$C1(m) = \sum C1(i).$$

where the minimization and summation are over all the gate inputs i.

3) If m is the output of a NAND gate

$$C0(m) = \sum C1(i).$$

$$C1(m) = \text{MIN}\,\{C0(i)\}$$

4) If m is the output of an OR gate

$$C0(m) = \sum C0(i).$$

$$C1(m) = \text{MIN}\,\{C1(i)\}$$

5) If m is the output of a NOR gate

$C0(m) = MIN\{C1(i)\}$

$C1(m) = \sum C0(i)$.

6) If m is a fanout branch of a stem s whose fanout count is f_i

$C0(m) = C0(s) + f_i - 1$

$C1(m) = C1(s) + f_i - 1$

Finally, the controllability cost functions are also used in FAST to stop backtracing at certain lines, called v-backtrace stop lines ($v = 0$ or 1), which can be assigned without leading to any conflicts (inconsistencies). A controllability cost of 0 should denote an assignment that does not lead to conflicts. This will allow us to identify 0- and 1-backtrace-stop lines among the lines with 0 controllability costs C0 and C1, respectively.

Example 3.2

Consider the given circuit in Figure 3.2, where the controllability values of the line K for the logic "0" and "1" are 0 and 2, respectively. If K is set to the logic value 0, then the backtrace path F will be selected instead of I2 since it is easier to set F tō be 0.

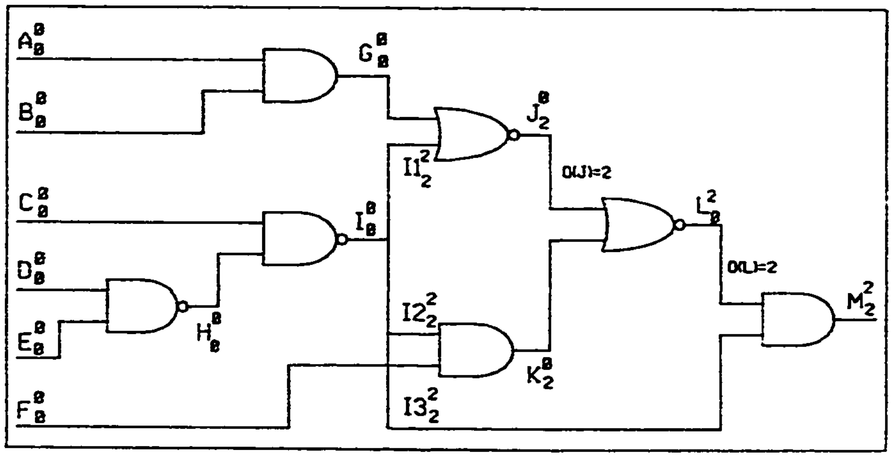


Figure 3.2 A Circuit Example to Demonstrate the Computation of Controllability Costs

3.3 Multiple Backtrace of FAN

***Aim*:**

Multiple backtrace of FAN [28] is used to concurrently backtrace more than one path instead of backtracing a single path.

***Description*:**

In the single backtrace procedure, which is used by many testing algorithms, an objective is defined by a pair consisting of an objective logic value and an objective line. An objective line is the line at which the objective logic value is desired. An objective is defined by a triple:

$$(k, n_0 (k), n_1 (k))$$

where:

k is an objective line,

n_0 (k) is the number of times the objective logic value 0 is required at k, and

n_1 (k) is the number of times the objective logic value 1 is required at k.

which is used in the multiple backtrace of FAN [28].

Multiple backtrace starts with one or more initial objective(s), that is, a set of Initial_objectives. It keeps moving backward and stops at certain internal lines rather than going all the way to primary inputs (PIs). These lines are called head lines. In FAN [28], a head line is defined as a line that is fed by a fanout free subcircuit (a subcircuit that has no fanout lines). The reason for stopping the multiple backtrace at head lines is that any objective at a head line can be backtraced until primary inputs and its value can be justified without any conflicts.

Beginning with the set of Initial_objectives, a set of objectives which appears in the middle is called a set of Current_objectives. A set of objectives which will be obtained at head lines is called a set of Head_objectives. A set of objectives at stems is called a set of Stem_objectives.

An initial objective required to set 0 to a line k is

$$(k, n_0(k), n_1(k)) = (k, 1, 0)$$

and an initial objective required to set 1 to a line k is

$$(k, n_0(k), n_1(k)) = (k, 0, 1).$$

Multiple backtarce works depth-first from the initial objectives backwards to head lines. The next objectives are successively determined from the current objectives by the following six rules:

Rule 1: AND gate.

Let k be an unassigned input that is the easiest to set to 0. Selecting the easiest input is basically based on the controllability costs that are computed using the controllability measure of [43], this is a generalized selection rule for the rest of gate types. Let y be the output of the AND gate,

then

n_0 (k) = n_0 (y), n_1 (k) = n_1 (y),

and for other unassigned inputs K_i,

n_0 (k_i) = 0, n_1 (k_i) = n_1 (y).

Rule 2: OR gate.

Let k be an unassigned input that is the easiest to set to 1. Let y be the output of the OR gate,

then

n_0 (k) = n_0 (y), n_1 (k) = n_1 (y),

and for other inputs K_i,

n_0 (K_i) = n_0 (y), n_1 (K_i) = 0.

Rule 3: NAND gate.

Let k be an unassigned input that is the easiest to set to 0. Let y be the output of the NAND gate,

then

$n_0(k) = n_1(y), n_1(k) = n_0(y)$

and for other inputs K_i,

$n_0(K_i) = 0, n_1(K_i) = n_0(y)$.

Rule 4: NOR gate.

Let k be an unassigned input that is the easiest to set to 0. Let y be the output of the NOR gate,

then

$n_0(k) = n_1(y), n_1(k) = n_0(y)$,

and for other inputs K_i,

$n_0(K_i) = n_1(y), n_1(K_i) = 0$.

Rule 5: NOT gate.

Let k and y be the input and output of NOT gate, respectively,

then

$n_0(k) = n_1(y), n_1(k) = n_0(y)$.

Rule 6: Stem.

Let s be a stem. Let B_i is the set of branches of stem s, then

$n_0(s) = \sum n_0(B_i)$, $n_1(s) = \sum n_1(B_i)$.

Each objective arriving at a stem or a head line stops its backtracing while there exist other current objectives. After the set of Current_objectives becomes empty, a stem p closest to a primary output is taken out, if one exists. Selecting the closest stem to a primary output guarantees that all the objectives that could depend on this stem have been backtraced. If the stem objective satisfies the following contradictory condition: that is, if n_0 (p) and n_1 (p) are nonzero, then stem p will be assigned the value {0 if n_0 (p) $> =$ n_1 (p) or 1 if n_0 (p) $<$ n_1 (p)} and the implication of this assignment will be performed. The reason of assigning a value to a stem is that there might exist a possibility of an inconsistency which the objective in backtracing has an inconsisent requirement such that n_0 (p) and n_1 (p) are nonzero. So as to avoid the fruitless computation, a binary value is assigned to the stem as soon as the objective involves a contradictory requirements.

When an objective at a stem s has no contradiction, that is, either n_0 (s) or n_1 (s) is zero, the backtrace would be continued from the stem. If all the objectives arrive at headlines , that is, both sets of Current_objectives and Stem_objectives are empty, then multiple backtrace terminates and returns a set of Head_objectives. After this, FAN takes out a head line one by one from the set of Head_lines objectives, the corresponding value is assigned to the head line based on its n_0 and n_1 values.

During the backtracing process, FAN creates what is called a decision tree. The decision tree is an ordered list of nodes with each node identifying a current assignment of either 0 or 1 to one head line or one stem that has a condradictory requirements, and the ordering reflects the relative sequence of which the current assignment were made. A node is flagged if the initial assignment has been rejected and the alternative is being tried. When both assignment choices at a node are rejected, then the associated node is removed and the predecessor node's current assignment is also rejected. So the use of the decision tree is to find an assignment for the tree nodes that meets the initial objective(s). After a stem of the decision tree is assigned a value v, an initial objective required to set v to this stem will be added to the set of Current_objectives. The backtrace would then be continued from this stem. The backtracing would be continued till the sets of Current_objectives and Stem_objectives are empty. Finally, an extracted version of multiple backtrace of FAN {38} will be introduced in Chapter 4.

3.4 Critical Path Tracing Fault Simulator (CPT)

Aim:

Given a test pattern t. The aim of CPT [38] is to simulate t and determine all the detected single stuck faults by t.

Description:

CPT consists of simulating the fault-free circuit (true value simulation) and using the computed signal values for tracing paths from primary outputs (POs), towards primary inputs (PIs) to determine the detected faults. Compared with conventional (parallel, deductive and concurrent) fault simulation, the distinctive features of CPT are as follows:

1) It directly identify the faults detected by a test, without simulating the set of all possible faults. Hence, it avoids all the work involved in propagating the effects of faults that are not detected by a test.

2) It deals with faults only implicitly. Therefore, there is no need for fault collapsing.

3) It is based on a path-tracing algorithm that does not require computing values in the faulty circuits by gate evaluations of fault list processing.

The test evaluation method consists of determining paths of critical lines , called critical paths, by a backtracing process starting at the POs. A line L with value v in a test t is ***critical*** if t detects the fault L s-a- $\bar{v}$. By finding the critical lines in a test t, we immediately know the faults detected by t.

Critical path tracing starts after the true-value simulation of the fault-free circuit for a test t has been performed. To aid the backtracing , the sensitive gate inputs are marked during the true-value simulation.

A gate input i is sensitive if complementing the value of i changes the value of the gate output. The sensitive inputs of a gate with two or more inputs are easily determined: If only one input has the controlling value c of the gate, then i is sensitive (AND and NAND gates have c = 0; OR and NOR have c = 1). If all inputs have noncontrolling values ($\bar{c}$), then all inputs are sensitive. Otherwise no input is sensitive.

The following rules provides the basis of the critical path tracing algorithm:

Rule I: If a gate output is critical, then its sensitive inputs, if any, are also critical. (Primary outputs are always critical)

A) Fanout-free circuits

For a fanout-free circuit (which always has a tree structure), critical path tracing is a simple tree traversal procedure that marks critical lines and recursively follows in turn every sensitive input of a gate with critical output.

Example 3.3

Consider the fanout-free circuit of Figure 3.3(a). In this Figure, the results of a true-value simulation is shown with the sensitive inputs marked by dots.

The critical lines are identified by recursive application of Rule I. Figure 3.3(b) shows the critical paths by heavy lines. Note how critical path tracing completely ignores the part of the circuit boarded by the lines B and C, since

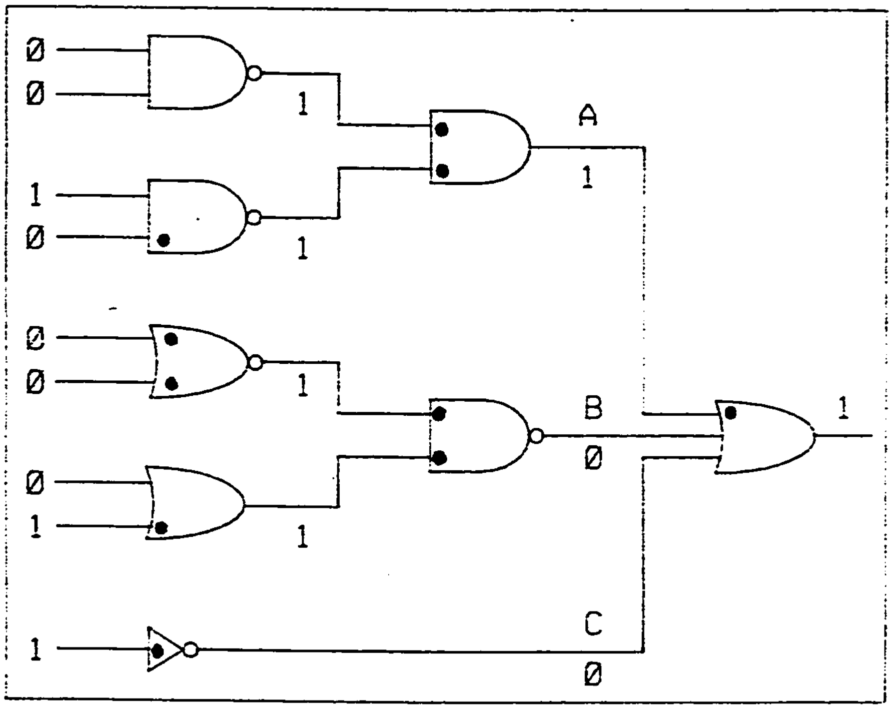


Figure 3.3(a) Sensitive Input Lines

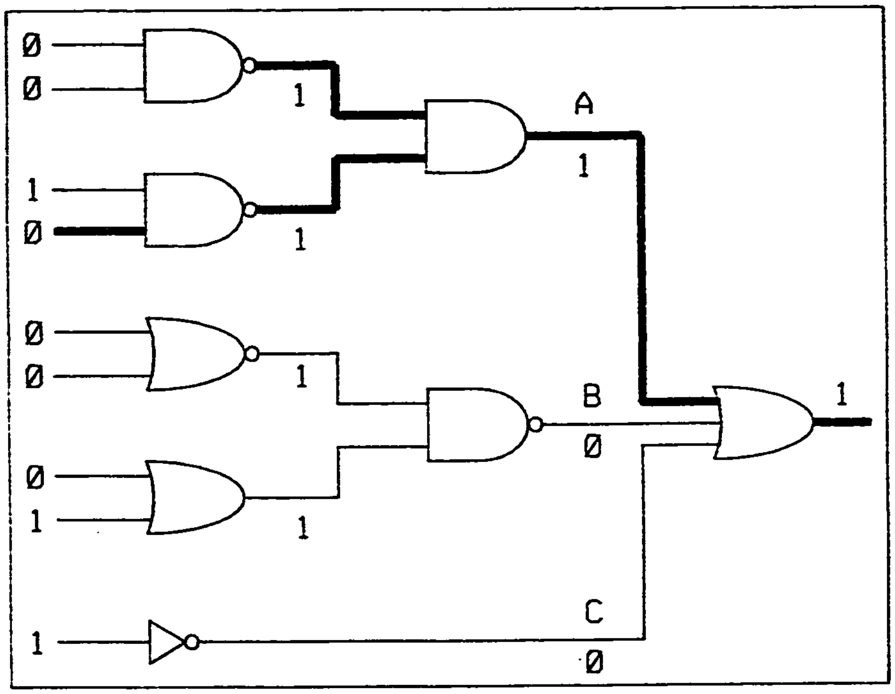


Figure 3.3(b) Critical Lines and Paths

working backward from the primary output, it first determines that B and C are not critical.

B) Circuits with reconvergent fanout

Under the stuck fault model, for a line with fanout we distinguish between its stem and its fanout branches (FOBs). In the presence of reconvergent fanout, a stem may not be critical even if one or more of its FOBs are critical. This is because the effects of a stem fault may propagate along two or more paths cancelling each other when they reconverge. This is called self-masking.

The main problem in circuits with reconvergent fanout is to determine whether a stem is critical, given that some of its FOBs are critical. This problem is called the stem analysis problem.

The rest of this section presents the theoretical basis of the main features of the stem analysis [38].

Example 3.4

Consider the circuit and the test given in Figure 3.4(a). Lines F,D,A, and B1 are identified as critical. The effects of the fault B s-a-0 propagate on two paths so that they cancel each other when they reconverge at gate F, therefore, B is not critical. This does not occur for the test shown in Figure 3.4(b), because the propagation of the effect of the stem fault along the path starting at B2 stops at gate E. In this case B is critical.

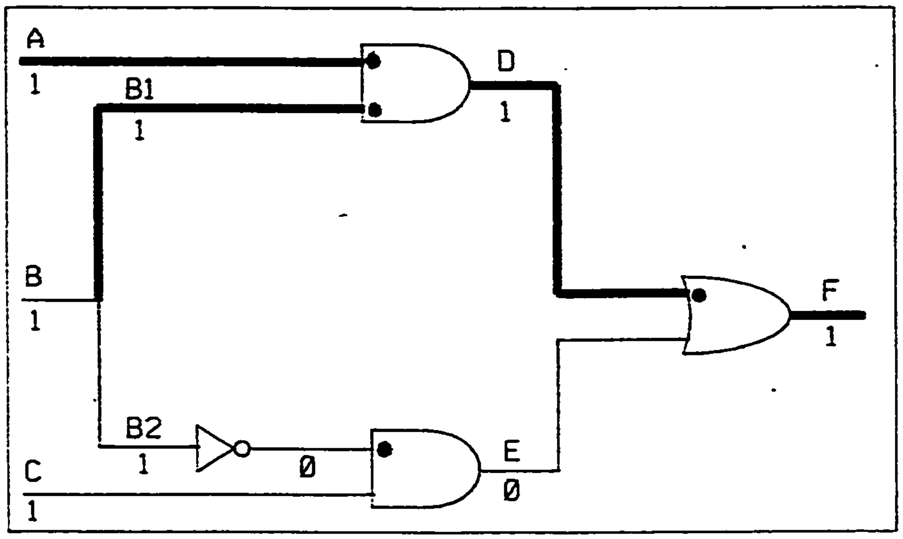


Figure 3.4(a) Simulation of Test Vector 111

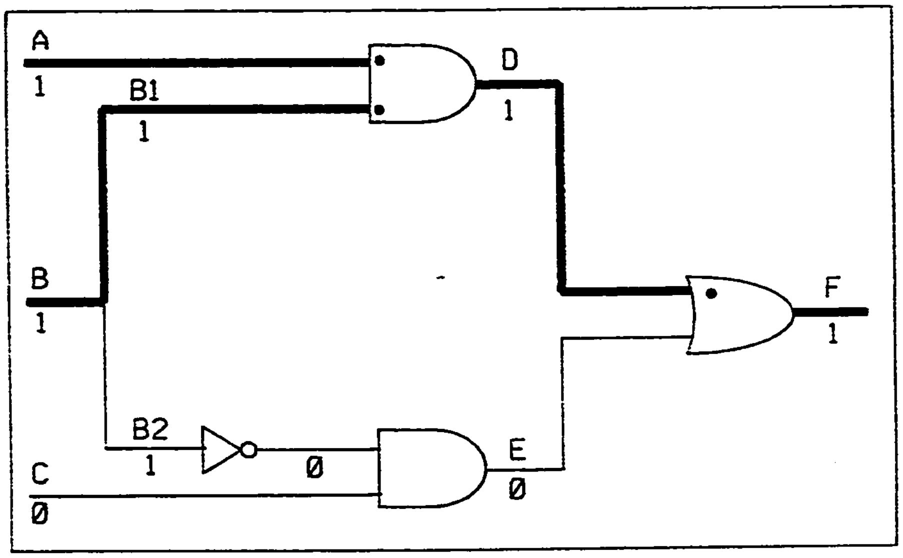


Figure 3.4(b) Simulation of Test Vector 110

C) *Stem analysis*

The stem analysis approach of [38] has the following features:

- can determine the detection of a fault without propagating its effects all the way to a PO.
- implicitly propagates fault effects without computing values in the faulty circuit.
- uses a simple preprocessing technique that allows some stem faults to be easily detected without any propagation of fault effects; and
- can propagate fault effects in a global manner, by leaps and bounds over potentially large areas of the circuit.

C1) *Partial fault effect propagation*

Let us consider some basic concepts in fault detection. If line x has value v in test t, we say that t activates the fault x s-a- $\bar{v}$. A line y is sensitized by t to this fault if the presence of the fault changes the value of y in t. If y is sensitized to the fault on x, we say that the effect of the fault propagates from x to y. Then there exists at least one path P between x and y, so that every line on P is sensitized to the fault x s-a- $\bar{v}$; P is called a sensitized path. The value of y is v + p, where p is the inversion parity of the path P.

Determining the detection of a fault without completely propagating its effects to a PO is based on the concept of a capture line.

Definition 1:

Let t be a test that activates fault f in a single-output circuit. If t sensitizes line y to f, but does not sensitize any other line with the same level as y (see Circuit_leveling procedure of Section 3.1.1), then y is said to be a capture line of f in test t.

A capture line, if one exists, belongs to all the paths on which the effect of f can propagate to the PO in test t. If t detects f, then there exists at least one capture line of f, namely the PO itself. If the effect of f propagates on a single path, then every line on that path is a capture line of f. (see Figure 3.5 which shows the capture lines for the fault b s-a-0).

We can easily show that a test t detects a fault f if and only if all the capture lines of f in t are critical in t. Therefore, we may not need to propagate the effects of the stem fault "all the way" to a PO as done in explicit fault simulation. Rather, it is sufficient to find a capture line of x, say y, whose status (critical or not critical) has already been determined. Then x is critical if y is critical.

C2) *Implicit fault effect propagation*

Propagating fault effects without explicit computation of faulty values is based on the marking of the sensitive gate inputs done during the true value simulation phase. A gate G propagates fault effects if either (1) fault effects arrive only on sensitive inputs of G or (2) G has no sensitive inputs and fault effects arrive on all inputs with controlling value and only on these inputs.

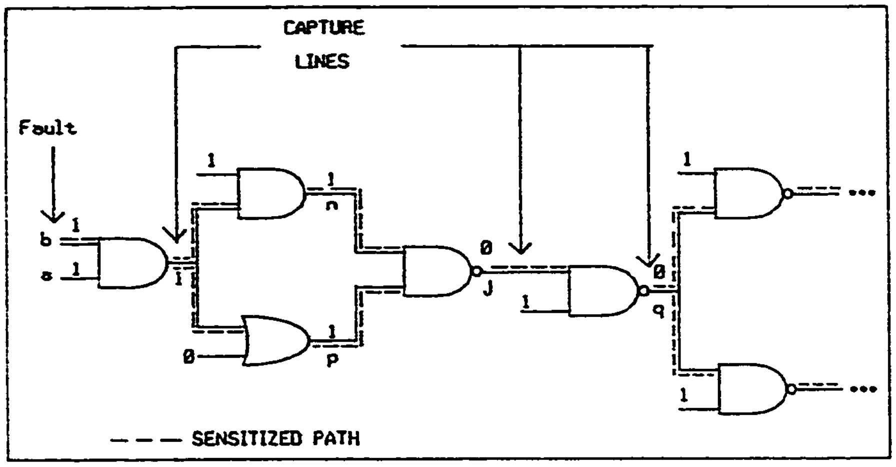


Figure 3.5 Determination of Capture Lines for the Fault b s-a-0

Figure 3.6 shows the different possible cases of propagation and nonpropagation of fault effects through an AND gate.

C3) Stem analysis without fault effect propagation

Definition 2:

In a single output circuit, a line y that belongs to all the paths between the line x and the PO is said to be a cover line of x. If all paths between x and its cover line y have the same inversion parity, then y is said to be an equal parity cover line of x.

Unlike a capture line, a cover line is not defined on the basis of the applied test.

Rule II: A stem that has an equal parity cover line is critical in any test in which one of its FOBs is critical.

This rule is based on the fact that self masking cannot occur in the region between a stem and its equal parity cover line.

C4) Global fault effect propagation

In this section, global properties used to analyze the propagation of fault effects through an entire FFR (fanout-free region) without processing its internal gates are introduced.

Let $\{ x_i \}$ be the set of inputs of an FFR. Let v_i be the value of x_i in

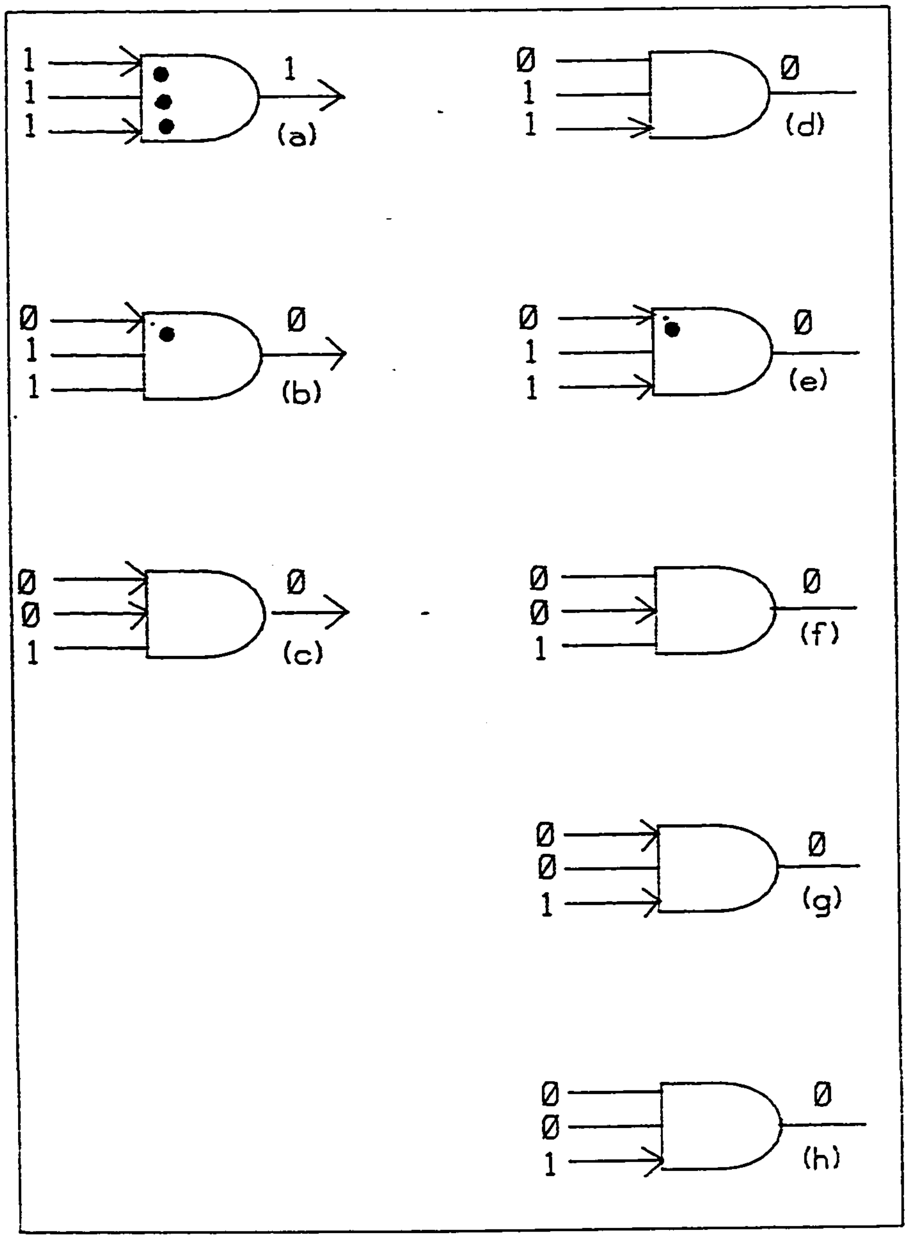


Figure 3.6 Propagation of Fault Effects through an AND gate: (a-c), Propagation; (d-h) Nonpropagation. Faults Effects are Indicated By Arrows

the test t. Let p_i be the inversion parity of the path between x_i and the FFR output.

Property 1:

If fault effects arrive on a subset $\{ x_k \}$ of FFR inputs, so that (1) at least one input in $\{ x_k \}$ is critical and (2) all the inputs in $\{ x_k \}$ have the same sum $p_k + v_k$, then the FFR propagates fault effects.

If we determine the parities p_i by preprocessing, then the conditions of property 1 can be evaluated by checking the FFR inputs. whenever the conditions are satisfied, we conclude that fault effects propagate to the FFR output; then the fault effects can "jump the FFR," regardless of the number of gates it contains.

The following properties can be established in a similar manner.

Property 2:

All critical inputs $\{ x_j \}$ of an FFR have the same sum $p_j \oplus v_j$. This result simplifies the evaluation of the conditions of property 1, since it shows that we need to compute the sum $p_j \oplus v_j$ only for one critical FFR input.

Property 3:

If fault effects arrive only on critical inputs of an FFR, then the FFR propagates fault effects.

Property 4:

Consider an FFR whose output is critical. If a fault affects only one FFR input and that input is noncritical, then the FFR does not propagate the fault effect.

D) *Algorithm for critical path tracing*

D1) *Preprocessing*:

The multiple output circuit is processed to determine its cones. A cone contains all the logic gates feeding one PO and is represented as interconnection of FFRs. Constructing cones and FFRs is done by a simple backtrace procedure [38].

D2) *Algorithm* [38] :

Figure 3.7 outlines the algorithm for evaluating a given test. It assumes that true-value simulation, including the marking of the sensitive gate inputs, has been performed. The algorithm processes each cone starting at its PO and alternates between two main operations: critical path tracing inside an FFR, represented by the procedure Extend, and checking a stem for criticality, done by procedure Critical. If a stem j is found to be critical, critical path tracing continues from j. Figure 3.8 shows the recursive procedure Extend(i) that backtraces all the critical paths inside an FFR starting from a given critical line i by following lines marked as sensitive. Extend stops at FFR inputs and collect all the stems reached in the set Stems_to_check.

Every stem in stems_to_check has at least one critical FOB. The algorithm always selects the highest level stem for analysis and hence guarantees that the status (critical/noncritical) of all its FOBs is known. The key element is the routine Critical(j) that determines whether the stem j is critical.

```
CPT ( )
{
 For every primary output z
     {Stems_to_check = $\Phi$
      Extend(z)
      While (Stems_to_check $\neq \Phi$ )
           { j = the highest level stem in Stems_to_check
             Remove j from Stems_to_check
             If Critical(j) Then Extend (j)
           }
      }.
}
```

Figure 3.7 CPT---An Algorithm for evaluating one test.

```
Extend(b)
{Mark b as critical
 If b is a fanout branch of stem i Then
   Add stem i to Stems_to_check
 Else
   G = a gate that its output is b
   For every input j of G
       If j is sensitive Then Extend(j)
}
```

Figure 3.8 Backtracing inside an FFR

E) Determining the criticality of a stem

In CPT [38] two different procedures for stem analysis are presented. The first one implements the concepts of capture lines and implicit fault effect propagation. In addition to these, the second procedure also uses the identification of stems with equal parity covers and global fault effect propagation. In this thesis, we restrict our discussion to the first procedure.

Stem Analysis Procedure:

The stem analysis procedure [38] is given in Figure 3.9. It implicitly propagates the effects of the fault on stem j in a breadth-first manner. Frontier is the set of all gates currently on the frontier of this propagation. A gate in Frontier has been reached by one or more fault effects from j. The propagation of these effects through the gate is represented by the procedure Propagates. Procedure Propagates determines that by checking if one of the two conditions stated in part C2 is satisfied or not. When the last gate has been removed from Frontier and it propagates fault effects, then its output is a capture line of the stem fault and the stem is critical if its capture line is so.

```
Boolean Critical(j)
{Frontier = {set of front gates that are reachable from fault effects of j}
 Repeat
    { g = lowest level gate in Frontier
      Remove g from Frontier
      If (Frontier ≠Φ ) Then
          If Propagates(g) Then add set of front gates that are reachable
                  from fault effects of the output line of g to Frontier
      Else
          { If Propagates(g) Then
             If (the output line of g is critical) Then Return TRUE
          Return FALSE
          }
    }
}
```

Figure 3.9 Critical(j)

F) Summary of CPT fault simulator [38] **:**

CPT consists of the following steps:

1) Preprocessing the given circuit to determine its cones and FFRs, and to identify stems with equal parity covers.

2) True value simulation of a test and identification of the sensitive gate inputs.

3) Critical path tracing, which is a backtracing that identifies the Critical lines (and hence the detected faults) in the test simulated in step 1.

A variation from CPT has also been introduced in [12]. This variation is called Partial_CPT. The Partial_CPT fault simulator has the capability to do partial fault simulation beginning from assigned primary output(s) points and from certain points called restart points. For determining critical lines, Partial_CPT moves backward from these points towards primary inputs. It does not move from a gate output to its inputs unless all its inputs are assigned. Otherwise, it stops at the gate output and it will be considered as a restart point. So Partial_CPT restarts its backtracing from restart points (if any) and from the new primary outputs that have binary values (if any). Example 3.5 illustrates how the Partial_CPT fault simulator works.

***Example 3.5*:**

Consider the circuit example of Figure 3.10, where PO1 is assigned and lines d, e, f, g have X value. Partial_CPT starts its backtracing from the assigned PO1 for determining critical lines as well as restart points (if any). Figure 3.10(a) shows all the determined critical lines by the first pass of Partial_CPT. Point b becomes a restart point. Assume that some of the circuit lines that have X value are assigned. So Partial_CPT restarts its backtracing to determine the new critical lines (if any). It restarts from the new assigned PO2 and from restart point b. Figure 3.10(b) shows all the determined critical lines by the first and second pass of Partial_CPT. Since at this point all POs are assigned and no more restart points, Partial_CPT stops its backtracing.

3.5 Test Generation Stopping Function of SMART (TGS)

Aim:

TGS [12] is used to stop generating any more tests.

Description:

This function tracks the effectiveness of SMART. It stops SMART based on one of the following criteria:

a) A specified time limit is exceeded.

b) A specified fault coverage is achieved.

c) The average number of new faults of the last n tests is less than 2, results [12] has shown that n is mostly 30 for large circuits.

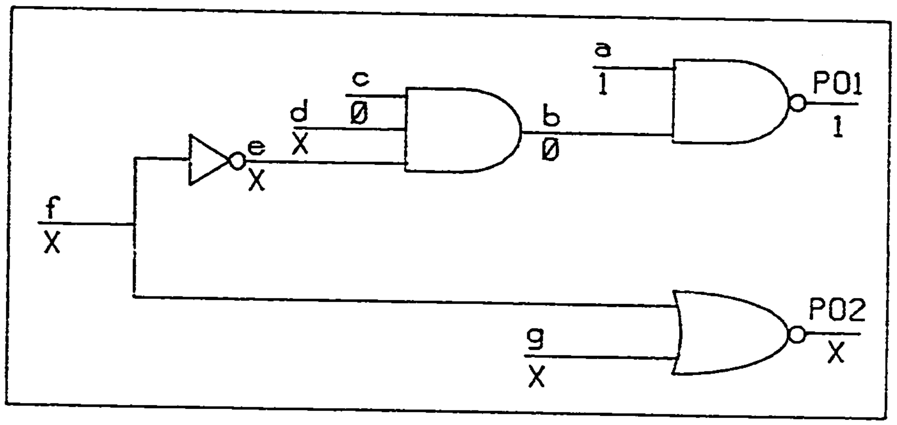


Figure 3.10 A Circuit Example

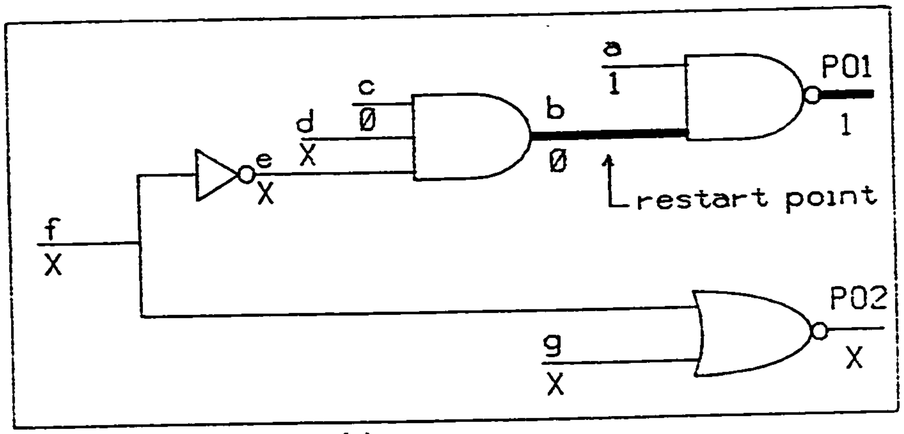


Figure 3.10(a) Partial CPT--First Pass

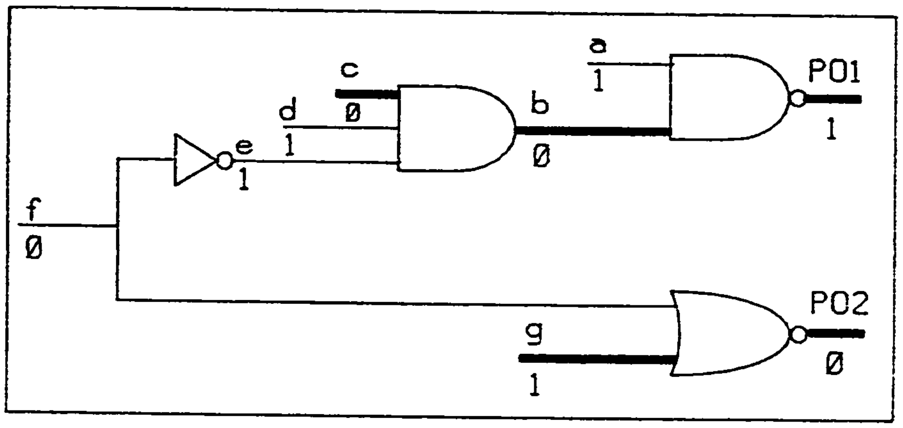


Figure 3.10(b) Partial CPT--Second Pass

CHAPTER IV

NEW STRATEGIES AND PROCEDURES

In this chapter, we give a detailed description of our new proposed features (i.e., strategies and procedures). A new fault independent test generation algorithm, called MAX, is developed and will be introduced in Chapter V. It is based on some new features as well as all the procedures of Chapter III. MAX utilizes a new backtrace procedure, called CLM-Multiple-Backtrace, which is based on some of these strategies and procedures. The detailed description of CLM-Multiple-Backtrace is discussed in this chapter. Each one of the new features are explained and illustrated by examples. In addition, since results [12] has shown that SMART [12] is more efficient than RAPS [36] and RANDOM [35], some differentiations are made between SMART and MAX features. These differentiations show how our proposed strategies and procedures avoid some of the deffeciencies of SMART. They also show how the performance of the test generation process can be improved. This chapter is orgnized as follows. Section 4.1 presents Critical Lines Maximization Strategy (CLM). Section 4.2 presents CLM-Multiple-Backtrace. Section 4.3 presents Critical Primary Inputs Flipping Strategy (CPF).

4.1 Critical Lines Maximization Strategy (CLM)

Aim:

Given a circuit to be tested. The aim of CLM is to assist MAX in selecting the proper value v_{PO} for a selected PO and to assist CLM-Multiple-Backtrace in maximizing the number of critical lines while generating a test pattern t. This will increase the number of target faults that can be detected by t.

Description:

CLM is used to find the proper assignments of the lines (output and input lines) of the gate under consideration (GUC), such that the number of critical lines is maximized. All typical gate types (NOT, AND, NAND, OR, NOR) are considered. In order to show how CLM works under different conditions, four different cases are discussed below with some examples and comparisons with SMART [12]. These cases cover all the possible conditions that could be satisfied by a GUC. In general for any GUC, there are two major cases: the noncontrolling case (i.e., a 1/0 is assigned to the ouput of an AND/OR gate) and the controlling case of GUC (i.e., a 0/1 is assigned to the ouput of an AND/OR gate). In the noncontrolling case, the output assignment of GUC uniqely implies the noncontrolling value to be assgined to all inputs of the GUC (i.e., 1/0 for an AND/OR gate). So CLM case I handles the noncontrolling case. In the controlling case, at least one the GUC inputs must be assigned to the controlling value of the GUC (i.e., 0/1 for an

AND/OR gate). CLM cases II, III and IV handle the controlling case that works under three different conditions. CLM case II handles the selection of an input of GUC to assigned to the controlling value by giving preference to useful inputs (see details of CLM case II). In case that non of the controlling input lines of GUC is useful, CLM case IV handles the selection of an input of such GUC to be assigned to the controlling value by giving preference to the most controllable stop input line (see details of CLM case IV). After the controlling value is justified to the selected input of CLM case II, CLM case III tries to assign the noncontrolling value to the unassigned inputs of GUC that satisfy conditions of case II. The first and third cases of CLM are used by MAX and the other two cases are used by CLM-Multiple-Backtrace.

Case I:

GUC satisfies the following conditions:

1) GUC is a PO gate.
2) Its output is unassigned (X value).
3) Its output is v and $\bar{v}$ -useful where $v = i + \bar{c}$.

   where i = gate inversion, c = controlling input value of the gate and $\bar{c}$ = noncontrolling input value of the gate (e.g., for a AND/NAND gate $i = 0/1$, $c = 0$, $\bar{c} = 1$ where as for an OR/NOR gate $i = 0/1$, $c = 1$, $\bar{c} = 0$).

CLM objective:

In this case, all the lines of the GUC can potentially be made critical. Therefore, the objective of CLM is to assign a value $v = i \oplus \bar{c}$ to the output of GUC. This implies that all the GUC inputs must be assigned to $\bar{c}$.

Example 4.1:

Consider the PO gate G1 of the circuit shown in Figure 4.1(a) with the assumption that A is 0-useful. Since G1 satisfies the conditions of case I, the CLM objective is to assign 0 to its output A and 1 to its inputs B,C,D,E. Therefore, all the gate lines become critical as shown in Figure 4.1(b). (Critical lines are shown by heavy lines). Based on that assignment, the potential fault domain of the circuit that may be detected by a test pattern t includes the subcircuits S1, S2, S3 and S4. This gives a chance for many faults in S1, S2, S3 and S4 to be detected by t.

A comparison with SMART[12] :

Since SMART starts by justifying a randomly chosen value v to the PO, it might (50% chance) start by a value v = 1 for the PO A of the circuit shown in Figure 4.1(a). This makes at most one of the PO gate inputs critical. This SMART assignment shrinks the potential domain of detected faults of the circuit to only one of the subcircuits S1, S2, S3 or S4 as shown in Figure 4.1(c).

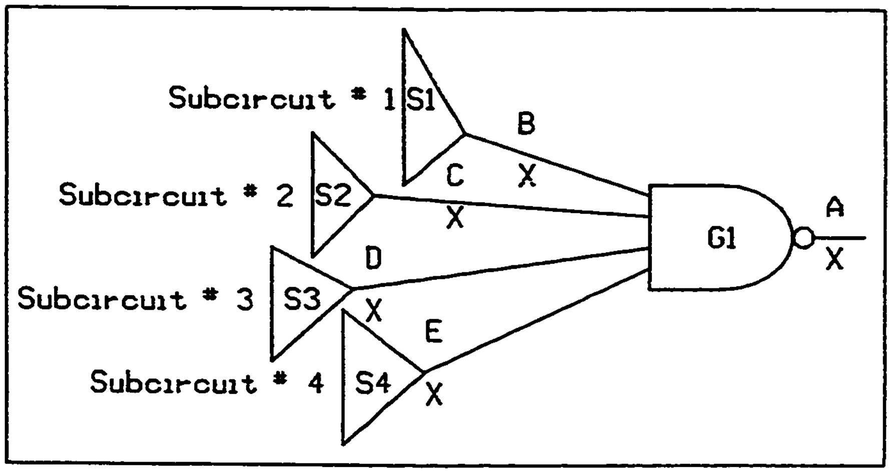


Figure 4.1(a) Example of CLM Case I

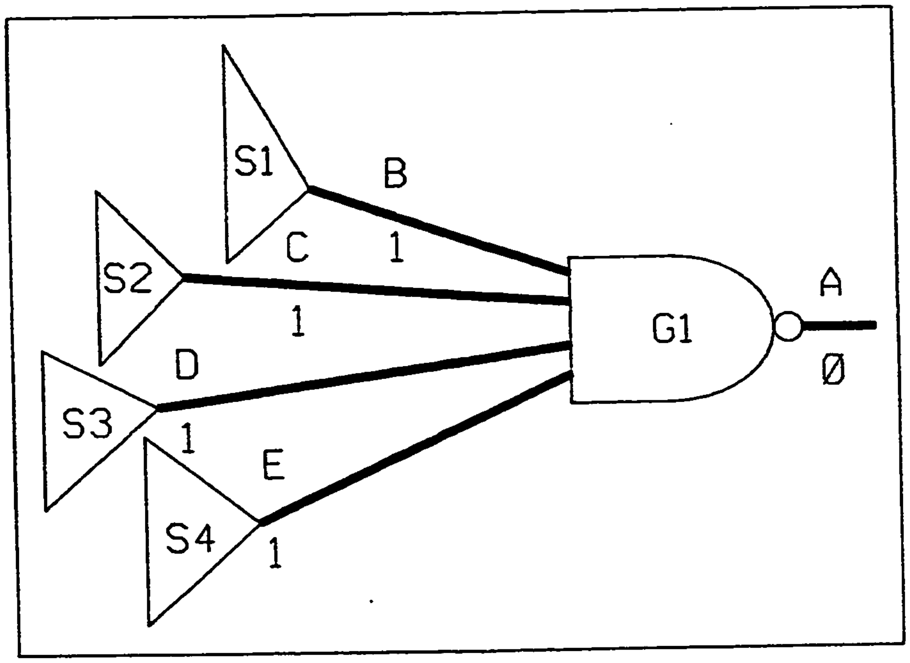


Figure 4.1(b) CLM Objective at G1

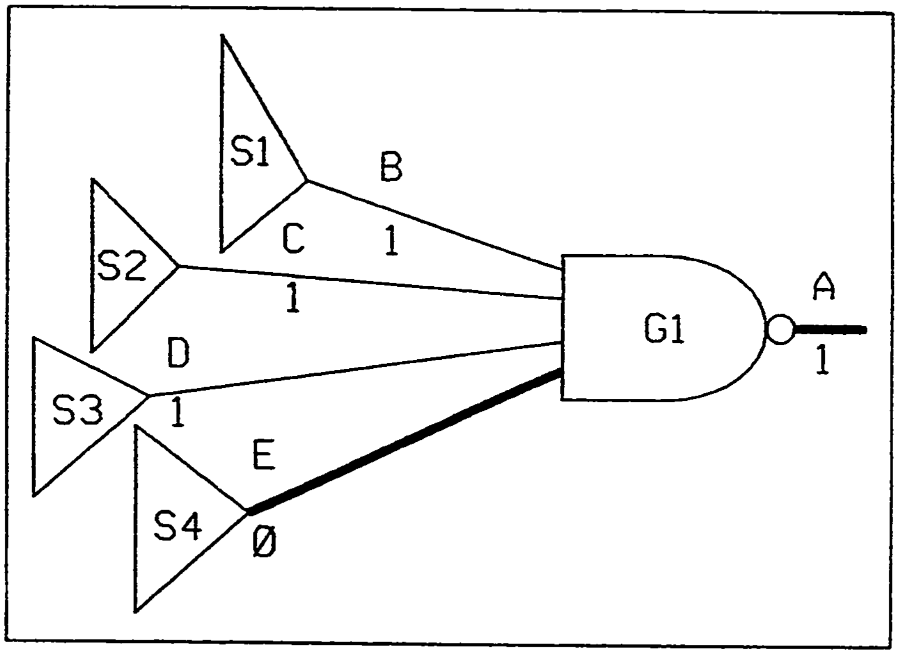


Figure 4.1(c) SMART Objective at G1

Case II:

GUC satisfies the following conditions:

1) Its output is critical and has a value that requires at least one of its inputs to be assigned to the controlling input value c.
2) Some of its inputs have X values.
3) None or some of the rest of its inputs have $\bar{c}$ values.

CLM objective:

In this case, at most one of the GUC inputs can be made critical. Therefore, the objective of CLM is to assign c to at most one of its inputs having X value. While CLM searches for an input to be assigned to c, it gives preference to the following inputs (sorted by preference):

1) Inputs that are c-useful.
2) Inputs that have high level number (i.e. farther to primary inputs).
3) Inputs fed by a gate that satisfies the following: all its inputs can potentially be made critical, if c is assigned to its output.

The CLM objective is then to assign the rest of inputs having x values to $\bar{c}$. The first condition ensures that new faults will be detected (i.e., new test patterns will be generated) and each of the second and third conditions will increase the chance of making more lines to be critical.

Example 4.2:

Consider G2 of the circuit shown in Figure 4.2(a) with an assumption that lines F,G,H are 0-useful. Since G2 satisfies the conditions of case II, the CLM objective is to assign 0 to input F and 1 to the rest of inputs G,H having X values. Therefore, input F becomes critical, while the rest of inputs are not critical (see Figure 4.2(b) for new critical lines). CLM gave preference to the 0-useful input F because assigning F to 0 will make all the three inputs of G4 critical.

A comparison with SMART [12] :

Since all inputs of G2 in Figure 4.2(a) are 0-useful, SMART arbitrarily selects an unassigned input of G2 and assigns the 0 value to it. Therefore, it might assign 0 to H or G (say G). This makes at most one of G5 inputs critical (see Figure 4.2(c)).

Case III (similar to the ***restart*** gate of SMART [12]):

GUC satisfies the following conditions:

1) Its output is critical.
2) At most one of its inputs has c value, and that input is c-useful.
3) Some of the rest of inputs have X values.
4) None or some of the rest of inputs are assigned to $\bar{c}$.

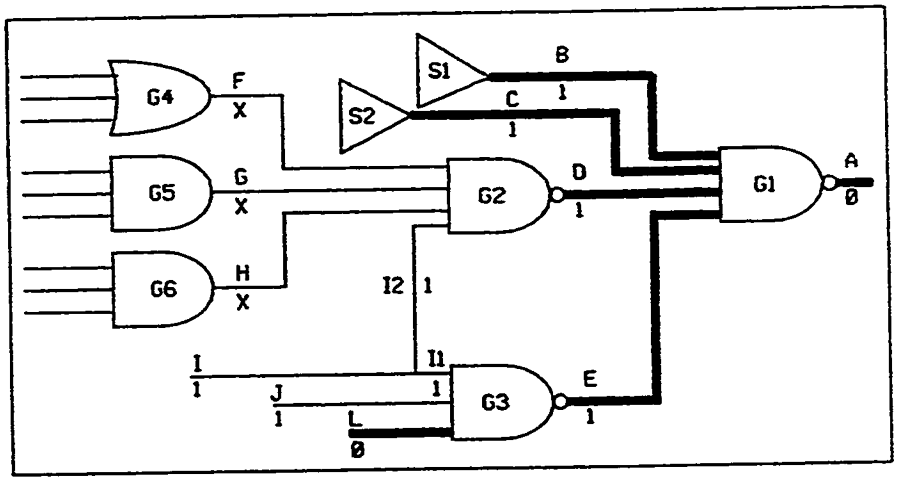


Figure 4.2(a) Example of CLM Case II

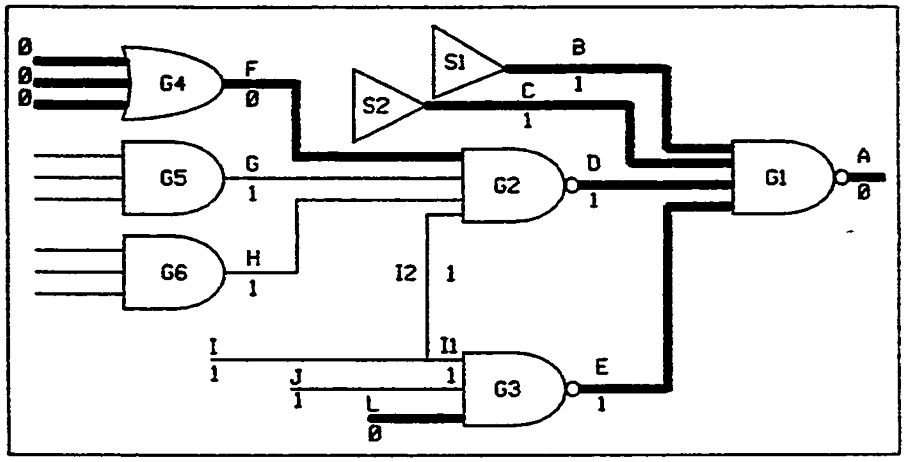


Figure 4.2(b) CLM Objective at G2

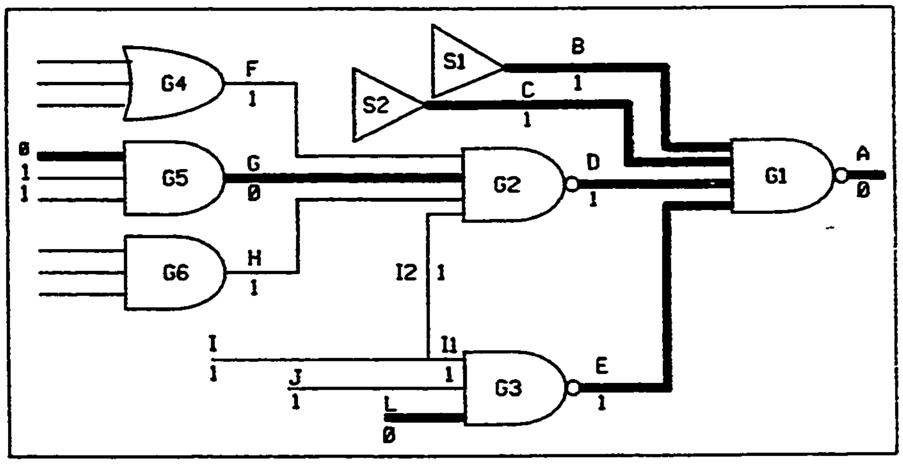


Figure 4.2(c) SMART Objective at G2

CLM objective:

In this case, CLM objective is to assign the rest of inputs having X values to $\bar{c}$. This makes the only input having c value critical.

Example 4.3:

Consider G2 of the circuit shown in Figure 4.3(a). Since G2 satisfies the conditions of case III, CLM assigns 1 to its inputs F,G,H having X values. Therefore, input I2 becomes critical, while the rest of inputs F,G,H are not critical (see Figure 4.3(b) for new critical lines).

Case IV (Stop-gate):

GUC satisfies the following conditions:

1) Its output is critical and has a value that requires at least one of its inputs to be assigned to the controlling input value c.
2) All its unassigned inputs are c-stop.

CLM objective:

Since assigning c to one of the stop gate inputs will not lead to new detected faults in the subcircuit that feeds the selected input, the CLM objective is then to hold the stop gate till CLM objectives for all other non-stop gates are satisfied. By satisfying CLM objectives for non-stop gates, the chance for detecting more faults will be increased. After all the non-stop gates are handled, the CLM objective at a stop gate is then to assign c to one of its

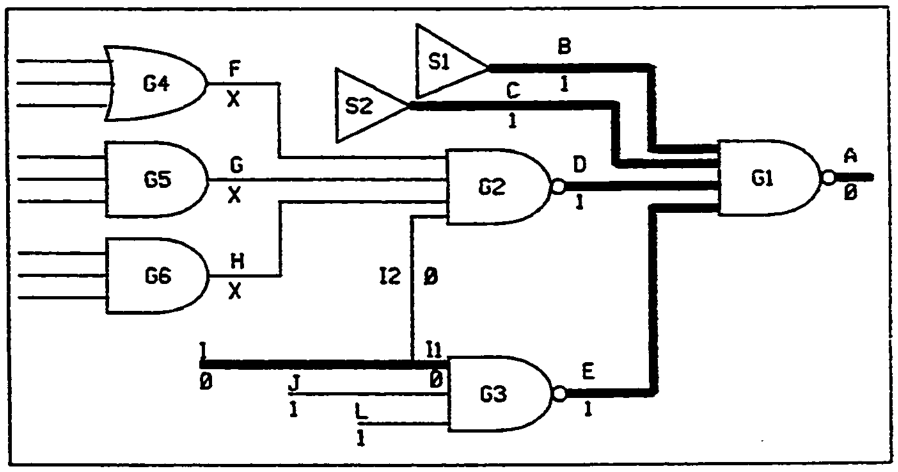


Figure 4.3(a) Example of CLM Case III

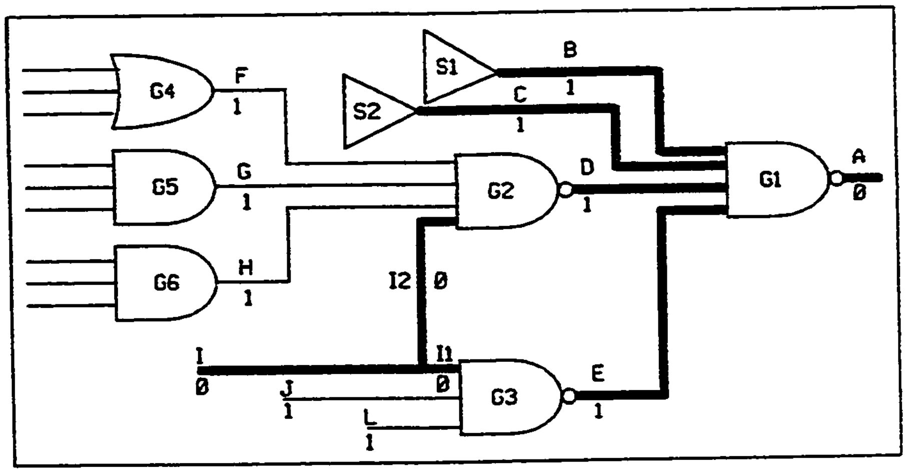


Figure 4.3(b) CLM Objective at G2

inputs (if none of the inputs has already been assigned a value c). While CLM searches for an input to be assigned to c, it gives preference to inputs that have low controllability cost. This decreases the chance of causing any further conflicts (inconsistencies).

***Example 4.4*:**

Consider the circuit shown in Figure 4.4(a) with an assumption that lines D,D1,D2,E are 0-stop and F is not. Since all G2 inputs (D1 and E) are 0-stop, therefore G2 is a stop gate. So the CLM objective is to hold G2 and handle the CLM objective at the other non-stop gates (if any). CLM starts to handle its objective at G3, since G3 is not a stop gate. Since F is the only 0-useful input line, then the CLM objective is to assign 0 to F and 1 to D (see Figure 4.4(b)). By handling G3 first, this gives chance to more faults to be detected in subcircuit S3.

***A comparison with SMART*[12] :**

Since all inputs of G2 in Figure 4.4(a) are 0-stop, SMART arbitrarily selects an unassigned input of G2 and assigns the 0 value to it. Therefore, it might assign 0 to D1 or E (say D1). Since D = D2 = D1 = 0, then the assignment of D = 0 prevents the only useful input line F of G3 to become critical. Also, no more new faults will be detected in subcircuits S1 (see Figure 4.4(c)).

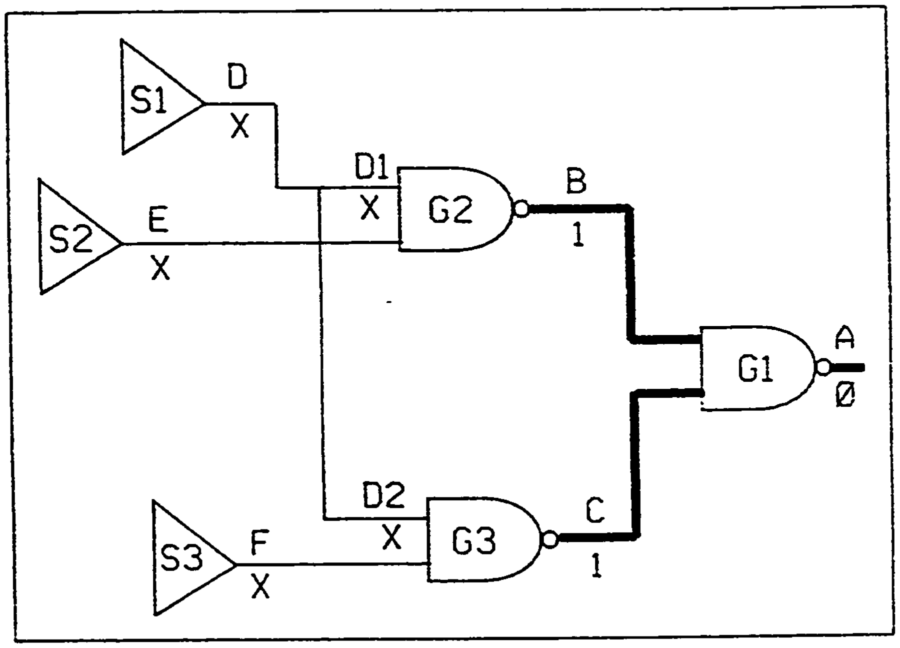


Figure 4.4(a) Example of CLM Case IV

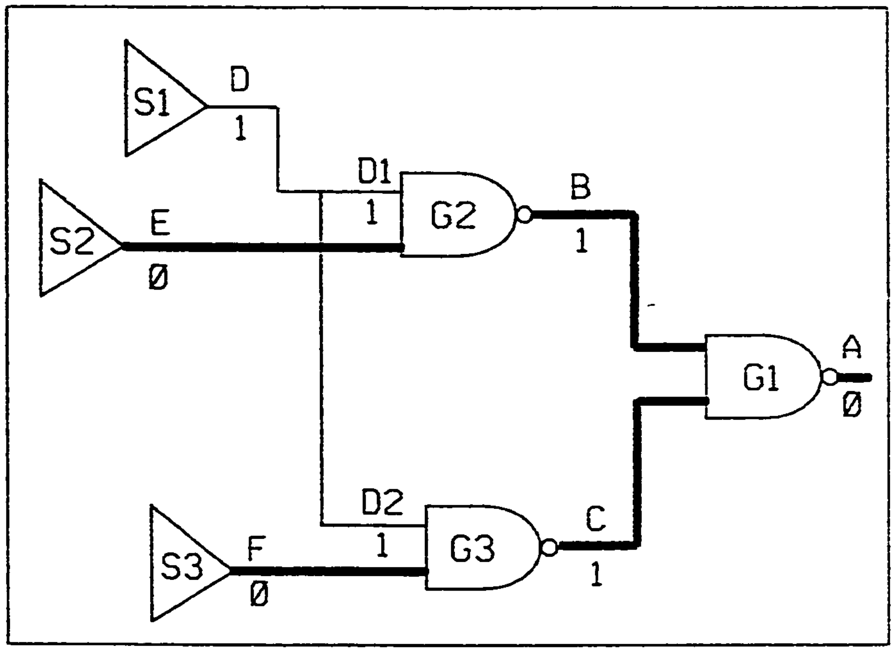


Figure 4.4(b) CLM Objective at G2 and G3

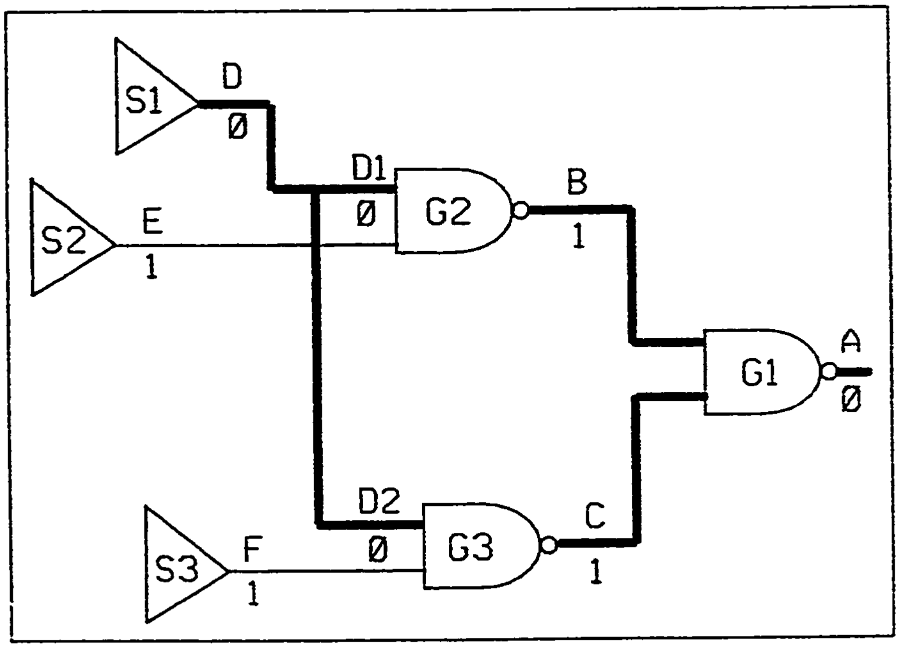


Figure 4.4(c) SMART Objective at G2 and G3

4.2 CLM-Multiple-Backtrace

***Aim*:**

While a test pattern is generated, CLM-Multiple-Backtrace is used to concurrently backtrace more than one path instead of backtracing along single paths and to maximize the number of critical lines (i.e. detected faults) by repeated usage of different cases of the CLM strategy.

***Description*:**

CLM-Multiple-Backtrace is an integrated version of multiple backtrace of FAN [28] and our new proposed strategy CLM. Like multiple backtarce of FAN [28], CLM-Multiple-Backtrace starts with one or more initial objective(s), that is, a set of Initial_objectives. It keeps moving backward and stops at head lines. We introduce the following definition for a head line:

***Definition*:** a head line is a line that is both 0- and 1-backtarce stop line.

Unlike multiple backtrce of FAN [28], CLM-Multiple-Backtarce is using six different rules that are modified from six rules of multiple backtrace of FAN [28] and integrated with CLM cases. The six rules are set as follows:

***Rule 1*: *AND gate*.**

***Rule 1(a)*: *Controlling case*.**

Let k be the unassigned input which is selected to be set to the controlling value c = 0 of the AND gate based on the three preferences listed in the objective part of CLM case II. Let y be the output of the AND gate (see Figure 4.5 case (a)),

then

$n_0(k) = n_0(y), n_1(k) = n_1(y)$.

If all unassigned gate inputs are 0-stop then add it to the set of Stop_gates.

Rule 1(b): Noncontrolling case.

Let y be the output of the AND gate which is selected to be set to 1. Let K_i be the set of all the unassigned inputs (see Figure 4.5 case (b)).

then

$n_0(K_i) = 0, n_1(K_i) = n_1(y)$.

Rule 2: OR gate.

Rule 2(a): Controlling case.

Let k be the unassigned input which is selected to be set to the controlling value c = 1 of the OR gate based on the three preferences listed in the objective part of CLM case II. Let y be the output of the OR gate (see Figure 4.5 case (c)),

then

$n_0(k) = n_0(y), n_1(k) = n_1(y)$.

If all unassigned gate inputs are 1-stop then add it to the set of Stop_gates.

Rule 2(b) Noncontrolling case.

Let y be the output of the OR gate which is selected to be set to 0. Let K_i be the set of all the unassigned inputs (see Figure 4.5 case (d)),

then

$n_0(K_i) = n_0(y), n_1(K_i) = 0.$

Rule 3: NAND gate.

Rule 3(a): Controlling case.

Let k be the unassigned input which is selected to be set to the controlling value c = 0 of the NAND gate based on the three preferences listed in the objective part of CLM case II. Let y be the output of the NAND gate (see Figure 4.5 case (e)),

then

$n_0(k) = n_1(y), n_1(k) = n_0(y).$

If all unassigned gate inputs are 0-stop then add it to the set of Stop_gates.

Rule 3(b): Noncontrolling case.

Let y be the output of the NAND gate which is selected to be set to 1. Let K_i

be the set of all the unassigned inputs (see Figure 4.5 case (f)),

then

$n_0(K_i) = 0, n_1(K_i) = n_0(\mathrm{y})$.

Rule 4: NOR gate.

Rule 4(a): Controlling case.

Let k be the unassigned input which is selected to be set to the controlling value c = 1 of the NOR gate based on the three preferences listed in the objective part of CLM case II. Let y be the output of the NOR gate (see Figure 4.5 case (g)),

then

$n_0(\mathrm{k}) = n_1(\mathrm{y}), n_1(\mathrm{k}) = n_0(\mathrm{y})$.

If all unassigned gate inputs are 1-stop then add it to the set of Stop_gates.

Rule 4(b): Noncontrolling case.

Let y be the output of the NOR gate which is selected to be set to 0. Let K_i be the set of all the unassigned inputs (see Figure 4.5 case (h)).

then

$n_0(K_i) = n_1(\mathrm{y}), n_1(K_i) = 0$.

Rule 5: NOT gate.

Let k and y be the input and output of NOT gate, respectively (see Figure 4.5 case (j)),

then

n_0 (k) = n_1 (y), n_1 (k) = n_0 (y).

Rule 6: Stem.

Let s be a stem. Let B_i is the set of branches of stem s (see Figure 4.5 case (i)),

then

n_0 (s) = $\sum n_0 (B_i)$, n_1 (s) = $\sum n_1 (B_i)$.

The pseudo code of Figure 4.6 describes CLM-Multiple-Backtrace procedure. Each objective arriving at a stem or a head line stops its backtracing while there exist other current objectives. After the set of Current_objectives becomes empty, a stem s closest to a primary output is taken out, if one exists. If the stem has contradictory condition, i.e., n_0 (s) and n_1 (s) are nonzero, then unlike multiple backtrace of FAN [28], n_0 (s) and n_1 (s) will be reassigned the values [1 and 0 if n_0 (s) $> =$ n_1 (s) otherwise 0 and 1 if n_0 (s) $<$ n_1 (s)]. CLM-Multiple-Backtrace would then be continued from this stem with the new values of n_0 (s) and n_1 (s).

When an objective at a stem s has no contradiction, that is, either n_0 (s) or n_1 (s) is zero, the backtrace would be continued from the stem. If all the objectives arrive at headlines , that is, both sets of Current_objectives and Stem_objectives are empty, then CLM-Multiple-Backtrace terminates and returns a set of Head_objectives and Stop_gates.

A variation from CLM-Multiple-Backtrace, called CLM-Multiple-Backtrace-I, is also introduced. This variation is different from the CLM-Multiple-Backtrace in the following:

1) At restart gates, CLM-Multiple-Backtrace-I starts its backtracing from all the unassigned noncontrolling inputs of the restart gate. While CLM-Multiple-Backtrace-I is moving backward, it gives preference to the gate inputs that have low controllability costs (i.e. easier ways to primary inputs).

2) At stop gates, CLM-Multiple-Backtrace-I starts its backtracing from the selected input of the stop gate. While it is moving backward, it gives preference to the gate inputs that have low controllability costs.

Two other variations, called CLM-Multiple-Backtrace-II and CLM-Multiple-Backtrace-III, are also introduced. The two variations are similar to CLM-Multiple-Backtrace and CLM-Multiple-Backtrace-I, respectively. But they start moving backward from head lines and stop at primary input lines.

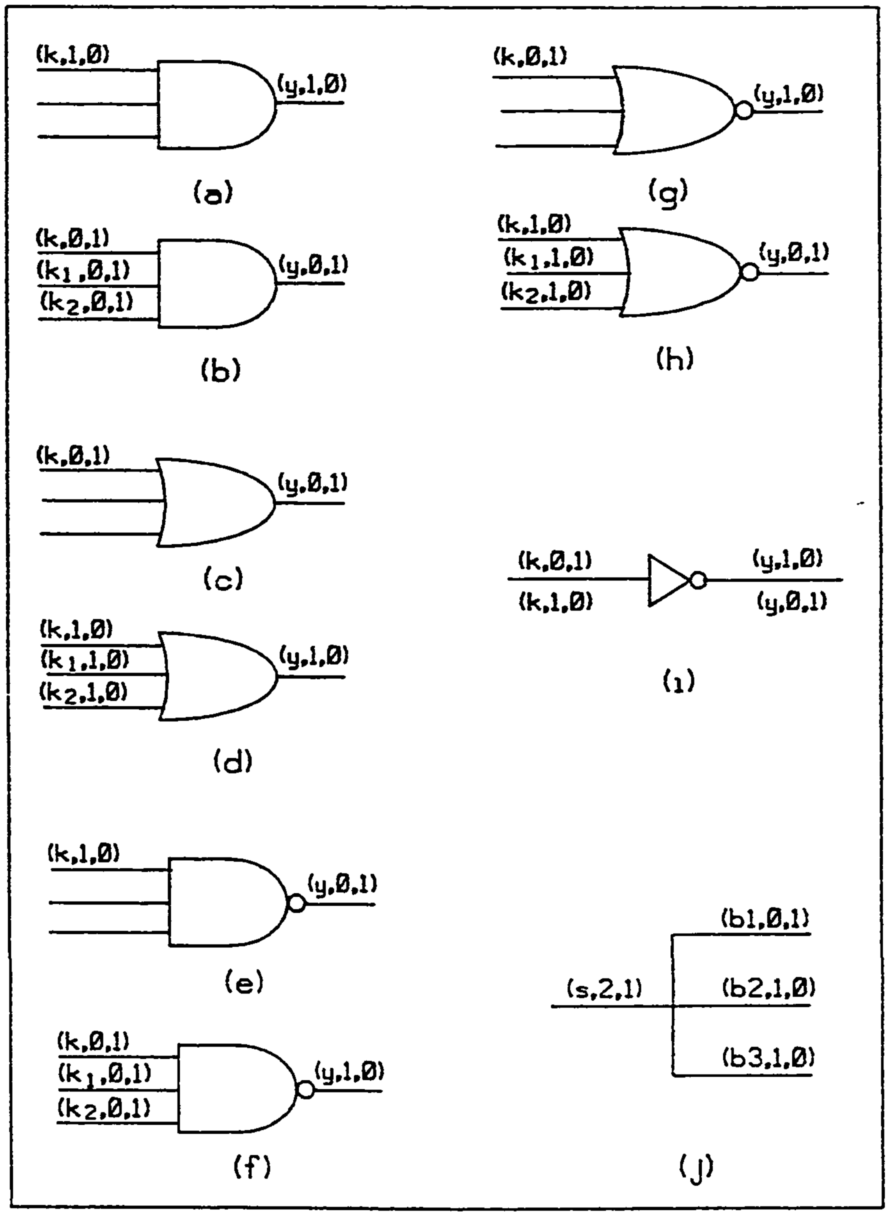


Figure 4.5 Multiple Objectives at Different Gates

CLM-Multiple-Backtrace (Initial_objectives)

```
{Current_objectives = Initial_objectives

 While (Current_objectives $\neq \Phi$ )

     { Remove one entry ($k$, $n_0(k)$, $n_1(k)$ ) from Current_objectives

       If (k is a head line)
           Add ($k$, $n_0(k)$, $n_1(k)$ ) to Head_objectives

       Else { If (k is a fanout branch)

                 { $s$ = stem of $k$

                   Update $n_0(s)$ and $n_1(s)$ values
                    based on $n_0(k)$ and $n_1(k)$ values

                   If (s is not in Stem_objectives)
                       Add ($s$, $n_0(s)$, $n_1(s)$) to Stem_objectives

                 }

              Else {  G = a gate that fed k

                      $i$ = inversion of G

                      $c$ = controlling value of G

                      If ( $n_0((k) > n_1(k))$ $v_k = 0$   else ( $v_k = 1$)

                      If ( $v_k \oplus i = c$ )

                         { If (G is a stop gate)
                               Add G to Stop_gates

                           Else { select an input h of G with value X
                                      based on the preferences of CLM case II

                                  Compute $n_0(h)$, $n_1(h)$ in terms of $n_0(k)$, $n_1(k)$
```

```
                        Add ($h$, $n_0(h)$, $n_1(h)$ ) to Current_objectives
                      }
                  }
                Else { For every input h of G with value X
                        Compute $n_0(h)$, $n_1(h)$
                      Add ($h$, $n_0(h)$, $n_1(h)$)
                      to Current_objectives
                  }
            }
        } /* End of Current_objectives While loop */
If (Stem_objectives $\neq \Phi$ )
   { Remove the closest stem s to a primary output
     If ( s has contradictory requirements )
        { If ( $n_0(s) >= n_1(s)$ )
             { Assign 1 to $n_0(s)$
               Assign 0 to $n_1(s)$
             }
          Else { Assign 0 to $n_0(s)$
                 Assign 1 to $n_1(s)$
               }
        }
     Add ( $s$, $n_0(s)$, $n_1(s)$ ) to Current_objectives
     Return CLM-Multiple-Backtrace (Current_objectives)
   } /* End of checking a stem objective */
```

```
 Return ( Head_objectives and Stop_gates )
} /* End of CLM-Multiple Backtrace */
```

Figure 4.6 CLM-Multiple-Backtrace

Comparisons with the Selective-backtrace of SMART [12] :

CLM-Multiple-Backtrace is different from the Selective-backtrace of SMART in the following:

1) It tries to maximize the number of critical lines by repeated usage of cases II and IV of the CLM strategy.

2) It concurrently traces more than one path rather than tracing single paths as in Selective-backtrace of SMART.

4.3 Critical Primary Inputs Flipping Strategy (CPF)

***Aim*:**

Given a generated test pattern t. The aim of CPF is to derive new test pattern(s) from t, with little additional effort.

***Description*:**

A test t_i is derived from t as follows: CPF flips the logic value v of a critical primary input line L in t if the conditions of one of the following two cases are satisfied:

***Case I*:** (L does not fanout)

(i) L is critical in t.

(ii) L is not $\bar{v}$-*stop* line.

Case II: (L has fanout lines)

(i) L is critical in t.

(ii) At least one of the critical branches of L is not $\bar{v}$-*stop* line.

The first condition of both cases ensures that the output is sensitive to changes in L. While the second condition ensures that new faults are detected by t_i. CPF repeats the above for each primary input (one at a time) that satisfies the two conditions of one of the above two cases. Each new derived test pattern t_i is different from t in one bit (flipped bit) only. After a new test pattern is derived, CPT [38] fault simulator is invoked to determine the new detected faults and the new stop lines

Example 4.5:

Consider the circuit and test shown in Figure 4.7(a). The test t = 010 detects the faults (A/1,B/0,B2/0,C/1,C1/1,D/1,D2/1,F/0,G/1,H/0), where L/i represents the fault "line L stuck at i". Since the critical PIs A, C are not 1-stop lines and the critical PI B is not 0-stop line, CPF flips A to 1, B to 0 and C to 1 one at a time. In this process, three new test patterns (t1 = 110, t2 = 000, t3 = 011) are derived. Then CPT is invoked to simulate these patterns (see Figures 4.7(b), (c), (d)). Critical paths are shown in these Figures and the sets of the new faults detected by t1, t2 and t3 are (A/0,H/1), (B/1,B2/1,C2/1,D/0,D1/0,D2/0,E/1,F/1,G/0) and (B1/0,C/0,C1/0) respectively.

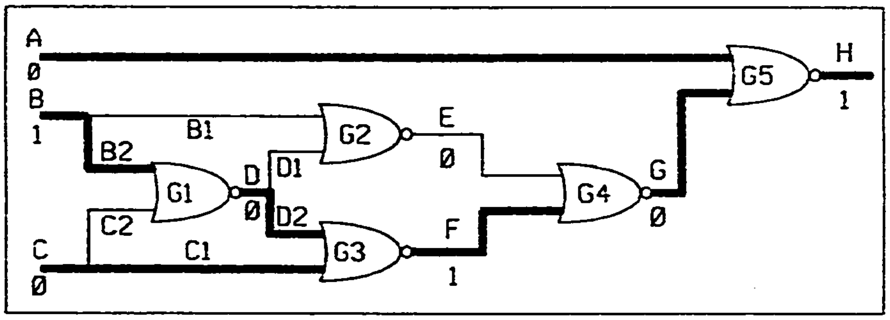


Figure 4.7(a) Simulation of t = 010

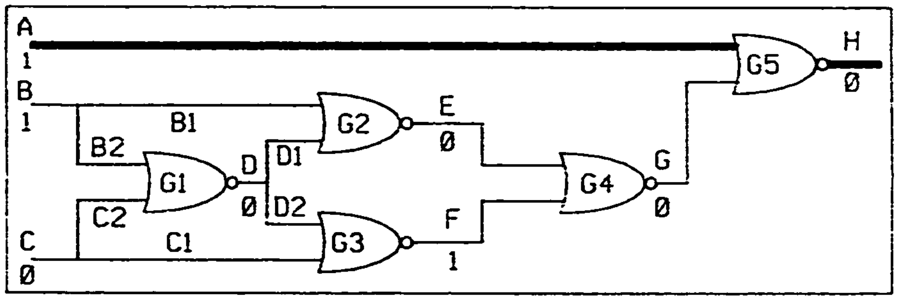


Figure 4.7(b) Simulation of t1 = 110

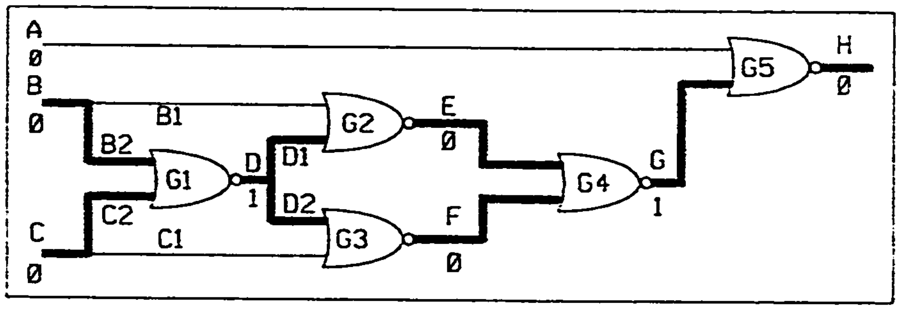


Figure 4.7(c) Simulation of t2 = 000

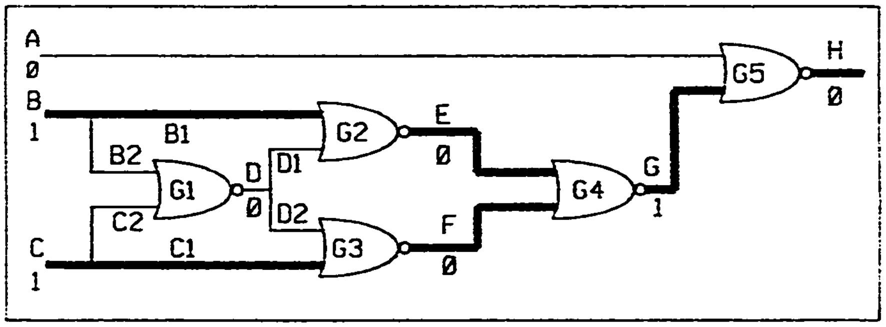


Figure 4.7(d) Simulation of t3 = 011

A comparison with SMART[12] :

Suppose that SMART worked on the same circuit of Figure 4.7(a). SMART would be required to generate the three new test patterns (t1,t2,t3), while CPF quickly derives (t1,t2,t3) without having to run a test patterns generation algorithm. Therefore, using CPF has the potential advantage of saving on computing time, since the time required for CPF is very small compared with a test patterns generation algorithm.

CHAPTER V

NEW PROPOSED ALGORITHM

In this chapter, we discuss the detailed description of our new proposed fault independent test generation algorithm MAX and illustrate it through an example.

5.1 Preliminary

In the proposed fault independent test generation algorithm MAX, many features (i.e. concepts, strategies and procedures) given in the literature have been extended and many other features have been introduced by us. These features are used in MAX to improve the performance of fault independent test generation process. Before MAX starts generating test patterns, two preprocessing steps should be performed: the circuit leveling step and controllability computing step. For generating a test pattern that maximizes the number of detected faults, MAX mainly uses CLM-Multiple-Backtrace. After a test pattern is generated, MAX uses CPF to derive new test patterns from the generated one with a little additional effort. For determining all single stuck faults detected by a test pattern, MAX uses CPT {38} fault simulator that shows better performance than the conventional fault simulators (i.e. parallel and concurrent). The detailed steps of MAX are provided below.

5.2 MAX Algorithm

Figure 5.1 outlines the MAX algorithm which proceeds as follows:

***Preprocessing steps*:**

1) Invoke circuit_leveling for leveling the circuit under test (CUT).

2) Invoke Compute_controllability for determining the controllability costs for all lines of the CUT.

Test Generation steps:

1) PO selection (Figure 5.2): Select a primary output (PO) Z from the set of POs having X value. While selecting a PO, give preference to the following POs:

a) POs that are both 0- and 1-useful, are not in the set of Blocked_PO, and having higher level numbers.

b) POs that are a-useful ($a = i \oplus \bar{c}$), are not in the set of Blocked_PO, and having higher level numbers
Where i = PO gate inversion; $\bar{c}$ = noncontrolling input value of the PO gate.

c) POs that are b-useful where $b = \bar{a}$, are not in the set of Blocked_PO, and having higher level numbers.

The concept of Blocked_POs is used to eliminate the initial selection of the same PO in two consecutive trials for generating test vectors, if there are other candidate POs. So this leads to new faults to be detected.

2) Selection of a PO value (Figure 5.2): Based on the type of the selected PO gate, deterministically select a value v_Z to the PO Z of step 1 as follows: if Z is a-useful then select value a to be the initial value of Z. Otherwise, select the other useful value b.

The selection of higher level number POs (i.e. farther from primary inputs) gives chance to more lines to be critical (i.e. longer critical paths). The selection of step 1(a) gives chance to more lines to be critical (see CLM case I), since in this case all lines of the selected PO gate will become critical (i.e., assigned to $\bar{c}$). It also gives chance for CPF to derive new test patterns, since the implication of the value of the flipped PI will causes the selected PO to take the opposite value (i.e., b value). The selection of step 1(b) has same advantages of part (a) except the advantage of using CPF.

3) Justification process: CLM-Justify, shown in Figure 5.3 , is used to justify the above selected value v_Z by repeatedly finding a head line assignment that is likely to contribute to setting Z to v_Z. The mapping of an objective into a head line assignment is recursively done by CLM-Multiple-Backtrace procedure shown in Figure 4.5. Internal line values are generated only by performing implication of the assigned headlines.

4) Handling Stop gates: CLM-Justify of step 3 might mark some gates as stop gates. So each one of these gates (if any) will be handled as shown in Figure 5.4. Only one of the selected stop gate inputs will be selected to be set to the controlling value c of the gate. CLM-Justify-1, shown in Figure 5.7, is then used to justify value c to the selected input.

5) Checking the selected PO value: If the selected PO Z of step 3 is still having X value then repeat steps 3 and 4.

6) Determination of detected faults, critical lines, and restart gates: partial-CPT fault simulator will be invoked to determine detected faults, critical

lines, and restart gates up to this point.

7) Restart gates (Figure 5.6): Pick a restart gate R, if any, such that its output has the highest level number (i.e., the farthest from primary inputs). Then invoke CLM-Justify-I to justify the noncontrolling value $\bar{c}$ of R to all the unassigned inputs of R.

8) Checking of useful POs: If any unassigned POs are still 0- or 1-useful, keep repeating steps 1-7.

9) Justification of headlines assignment (Figure 5.8): For all the assigned headlines, CLM-Justify is used to justify the headlines values by repeatedly finding a primary inputs assignment to set the corresponding values to the headlines.

10) Handling Stop gates: Similar to step 4.

11) Determination of detected faults, critical lines, and restart gates: Similar to step 6.

12) Restart gates (Figure 5.6): Similar to step 7.

13) Determination of unassigned primary inputs value (Figure 5.10): For all the unassigned primary inputs, if any, a random logic value will be assigned to each one of them. At this point all the PIs are assigned. So, a test vector t is generated and all detected faults by t are determined.

14) Monitoring the effectiveness of MAX: If a certain specified criteria is

achieved then stop MAX.

15) Determination of stop lines: Determine all stop lines by using the rules shown in chapter II.

16) Derivation of new tests: Invoke CPF to derive new tests, if any, from t.

17) Fault simulation: For each one of the new derived tests of step 16, perform the following:

    17.1) Invoke CPT fault simulator to determine all the detected faults.

    17.2) Determine all stop lines by using the rules shown in Chapter II.

    17.3) If a specified criteria is satisfied, stop MAX.

18) Keep repeating steps 1-17.

```
MAX ( )
{circuit_leveling( )
 Compute_controllability( )
 Blocked_POs = $\Phi$
 While (Useful POs)
      {Headlines_flag = FALSE
       Primary_inputs_flag = FALSE
       Add_PO = TRUE
       Initialize all the lines value to X value
       While (Useful POs have X values)
           {$(Z, v_Z)$ = Deterministically_select_a_primary_output_and_a_value( )
           Headlines_flag = TRUE
           CLM-Justify ($Z, v_Z$)
           Partial_CPT( )
           Restart_gates( )
          }
      Justify_headlines_assignment( )
      Assign_unassigned_primary_inputs( )
      /* At this point a test vector t is generated */
      TGS( )
      CPF(t)
```

```
        For each new derived test $t_i$ by CPF
            { CPT( $t_i$ )
              TGS( )
            }
        }
}
```

Figure 5.1 MAX--A Fault Independent Test generation Algorithm

```
Deterministically_select_a_primary_output_and_a_value( )
{  For all POs that have X value
      { $i$ = inversion of gate G of a PO Z
        $c$ = controlling input value of G
        $a = i \oplus c$
        $b = i \oplus \bar{c}$
        If ( Z is both 0- and 1-useful )
           Add Z to the set of Both_Useful
        Else { If ( Z is a-useful )
                    Add Z to the set of A_Useful
               Else { If ( Z is b-useful )
                           Add Z to the set of B_Useful
                    }
             }
      }
   If ( Both_Useful )
      { If ( All Both_Useful POs are BLocked ) Blocked_POs = $\Phi$
        Pick a PO Z from Both_Useful that has the highest level
                    number and Z is not in Blocked_POs
        If (Add_PO)
           { Add Z to Blocked_POs
             Add_PO = FALSE
           }
        Return ( Z , a )
      }
```

```
If ( A_Useful )
    { If ( All A_Useful POs are BLocked ) Blocked_POs = $\Phi$
      Pick a PO Z from A_Useful that has the highest level
                number and Z is not in Blocked_POs
      If (Add_PO)
          { Add Z to Blocked_POs
            Add_PO = FALSE
          }
      Return ( Z , a )
    }
If ( B_Useful )
    { If ( All B_Useful POs are BLocked ) Blocked_POs = $\Phi$
      Pick a PO Z from B_Useful that has the highest level
                number and Z is not in Blocked_POs
      If (Add_PO)
          { Add Z to Blocked_POs
            Add_PO = FALSE
          }
      Return ( Z , b )
    }
```

Figure 5.2 Selection of a primary output

```
CLM-Justify (m , $v_m$ )

  { While ( line m has X value)
          { Let $n_0(m) = \bar{v}_m$ and $n_1(m) = v_m$
            If (Headlines_flag)
               {CLM-Multiple-Backtrace (m, $n_0(m)$ , $n_1(m)$ )
                Stop_gates( )
                Assign_head_lines_and_simulate( )
               }
            Else { If (Primary_inputs_flag)
                          {CLM-Multiple-Backtrace-II (m, $n_0(m)$ , $n_1(m)$ )
                            Stop_gates( )
                            Assign_primary_inputs_and_simulate( )
                          }
                 }
            }
     Headlines_flag = FALSE
     Primary_inputs_flag = FALSE
  }
```

Figure 5.3 CLM-Justify

```
Stop_gates( )
 { While (Stop_gates)
        { Remove a stop gate G
          h = the selected unassigned input of G that has the lowest
                controllabiliy cost
          c = controlling value of G
          CLM-Justify-I ( h,c)
        }
 }
```

Figure 5.4 Stop gates

```
Assign_head_lines_and_simulate( )

 { While (Head_objectives)
         { Remove a head line objective (h, $n_0$ (h), $n_1$ (h))
           If (h is a primary input) Add objective (h, $n_0$ (h), $n_1$ (h)) to the
                                     set of Primary_input_objectives
           Else {If ( $n_0$ (h) > $n_1$ (h) ) Assign 0 to h
                Else Assign 1 to h
                 Add h to the set of Assigned_headlines
               }
         }
   Perform implication of all the assigned headlines
 }
```

Figure 5.5 Determination of headlines assignment

```
Restart_gates( )
  { While (Restart_gates)
        { Remove a restart gate G that its output has the highest level number
          c = controlling value of G
          For the set of inputs J of G that have X values
                CLM-Justify-1 ( J, $\bar{c}$ )
                Partial_CPT( )
        }
  }
```

Figure 5.6 Restart gates

```
CLM-Justify-I (J , $v_J$ )

 { While ( all or some lines J have X value)
         { $S$ = a subset of lines J that still have X value
           For every line s in S
             {If ( $v_s$ = = 0 ) $n_0(s) = \bar{v}_s$ and $n_1(s) = v_s$
              Else $n_0(s) = v_s$ and $n_1(s) = \bar{v}_s$
              If (Headlines_flag)
                 {CLM-Multiple-Backtrace-I (S, $n_0(S)$ , $n_1(S)$ )
                  Assign_head_lines_and_simulate( )
                 }
              Else { If (Primary_inputs_flag)
                        {CLM-Multiple-Backtrace-III (S, $n_0(S)$ , $n_1(S)$ )
                          Assign_primary_inputs_and_simulate( )
                        }
                   }
             }
         }
   Headlines_flag = FALSE
   Primary_inputs_flag = FALSE
 }
```

Figure 5.7 CLM-Justify-I

```
Justify_headlines_assignment( )
 { While ( Assigned_headlines)
        { Primary_inputs_flag = TRUE
          Remove a headline h which has a value $v_h$
          CLM-Justify (h, $v_h$ )
          Assign_primary_inputs_and_simulate( )
          Partial_CPT( )
          Restart_gates( )

        }
 }
```

Figure 5.8 Justifying assigned headlines

```
Assign_primary_inputs_and_simulate( )
 { While (Primary_inputs_objectives)
          { Remove a primary input objective (i, $n_0$ (i), $n_1$ (i))
            If ( $n_0$ (i) > $n_1$ (i) ) Assign 0 to i
            Else { If ( $n_0$ (i) < $n_1$ (i) ) Assign 1 to i }
          }
   Perform true value simulation for all the assigned primary inputs
 }
```

Figure 5.9 Determination of primary inputs assignment

```
Assign_unassigned_primary_inputs( )
  { While ( unassigned primary inputs )
        { Remove an unassigned primary input i
           Assign i a random logic value
        }
    Perform true value simulation for the assigned primary inputs
  }
```

Figure 5.10 Determination of unassigned primary inputs value

5.3 Illustrated Example:

Consider the circuit example shown in Figure 5.11. The stopping criteria for this example is to achieve 95% fault coverage. Figures 5.11(a) and (b) show the computed controllability costs and level number for all the circuit lines of Figure 5.11 respectively. Since all the circuit lines are initially unassigned and both of PO lines K and L are 0- and 1-useful , PO line K that has the highest level number among circuit POs is selected. K is added to set of Blocked_POs (see Figure 5.2). Since gate G7 of PO line K is a NAND gate, the value a $= i \oplus \bar{c} = 1 \oplus 1 = 0$ is deterministically selected to PO line K (see CLM case I). The selected value a $= 0$ of PO line K is mapped to the initial objective (K,1,0) which implies the following ordered set of current objectives: (E2,0,1) and (J,0,1) , see CLM-Multiple-Backtrace (CMB) rule 1(b).

Figure 5.11(c) shows all different objectives that appeared during the justification process of the initial objective (K,1,0). On the left side of an objective, there is a number that indicates the sequence of deriving such objective. On the other side, there is a letter that indicates the type of such objective (i.e., (C) current, (S) stem, (H) head and (PI) primary input objective). A stack data structure is used to store and handle each type of objectives. So the general policy of handling stack objectives is in the order of Last In First Out (LIFO).

Based on the LIFO policy, the current objective (J,0,1) is handled first which implies current objective (I,1,0), see CMB rule 5. Current objective

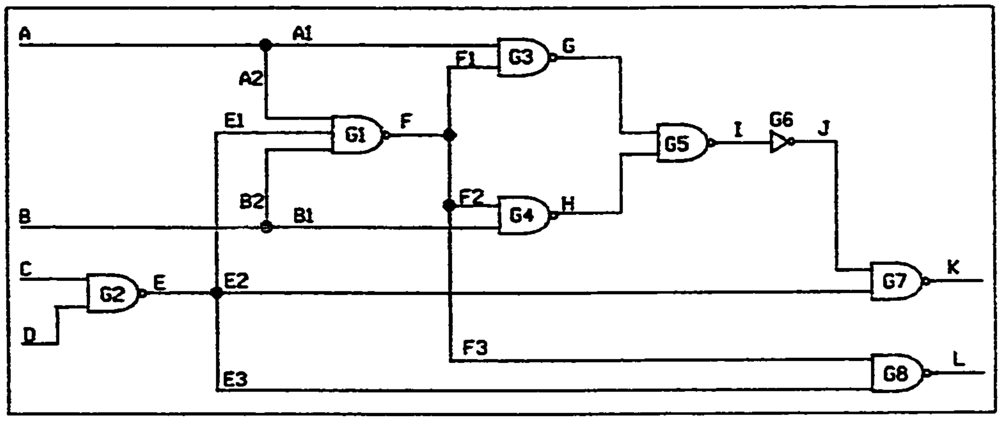
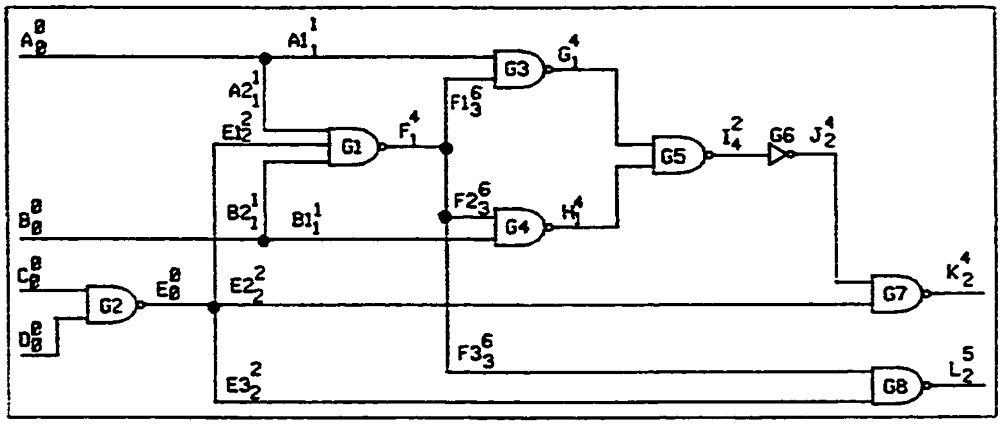
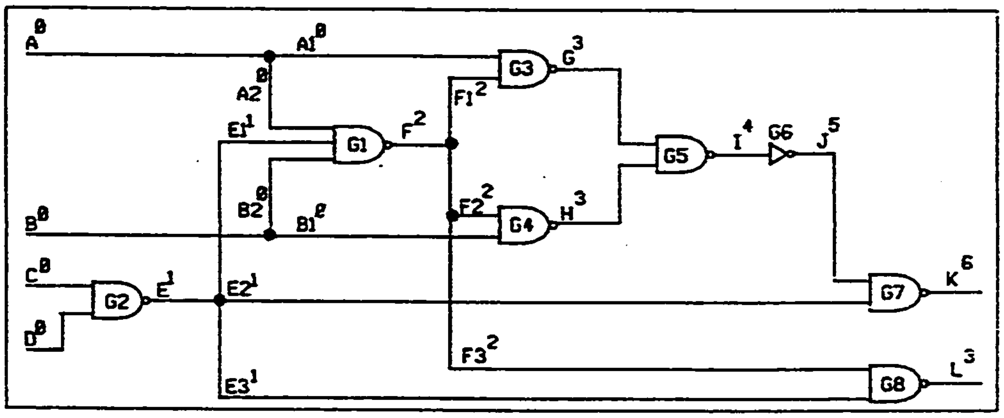


Figure 5.11 Circuit Example



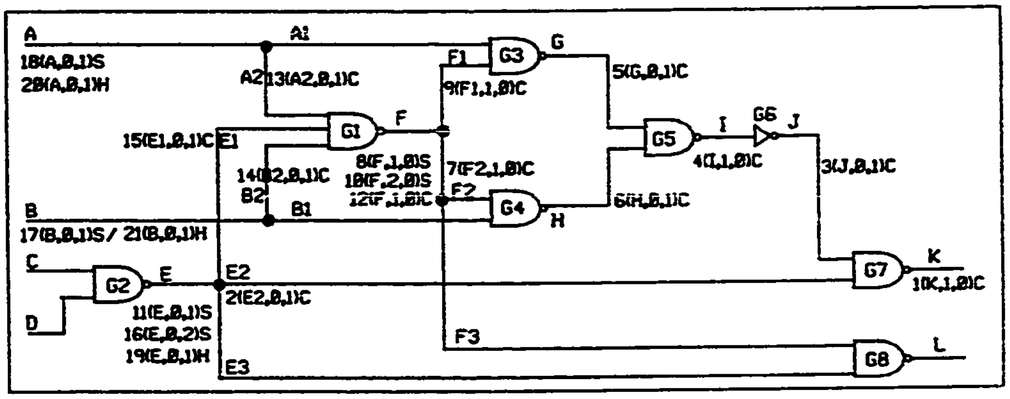
. L^{C0}_{C1} : C0 = 0-Controllability Cost of Line L
C1 = 1-Controllability Cost of Line L

Figure 5.11(a) Controllability Costs



. L^i : i = Level Number of Line L

Figure 5.11(b) Level Numbers



S(L, n_0 (L), n_1 (L))T : S = Sequence of Objective (L, n_0 (L), n_1 (L))
T = Type of Objective as follows:
(C) Current , (S) Stem , (H) Headline ,
(PI) Primary Input.

Figure 5.11(c) Justification of Initial Objective (K,1,0)

(I,1,0) implies ordered current objectives (G,0,1) and (H,0,1), see CMB rule 1(b). The current objective (H,0,1) is handled next. The unassigned input F2 of G4 is selected, since it is 0-useful and has the highest level number among G4 inputs (B1 and F2), see CMB rue 1(a). So the new generated current objective is (F2,1,0). The objective (F2,1,0) is mapped to stem objective (F,1,0) which is added to the set of stem objectives. The current objective (G,0,1) is handled next. The unassigned input F1 of G3 is selected, since it is 0-useful and has the highest level number among G3 inputs (A1 and F1), see CMB rule 1(a). So the new current objective is (F1,1,0). The obective (F1,1,0) updates the stem objective (F,1,0) to (F,2,0). The remaining current objective (E2,0,1) is handled and mapped to stem objective (E,0,1).

At this point, the set of current objectives is empty and the set of stem objectives has two stem objectives (F,2,0) and (E,0,1). Since F has higher level number than E, (F,2,0) is handled first. Stem objective (F,2,0) is updated and mapped to current objective (F,1,0). Current objective (F,1,0) implies ordered current objectives (A2,0,1), (B2,0,1) and (E1,0,1), see CMB case 1(b). The objective (E1,0,1) is handled first and (E1,0,1) updates stem objective (E,0,1) to (E,0,2). The objective (B2,0,1) is handled and mapped to stem objective (B,0,1). Then (A2,0,1) is handled and mapped to stem objective (A,0,1).

At this point, the set of current objectives is empty and the set of stem objectives has three stem objectives (A,0,1), (B,0,1) and (E,0,2). Since E has highest level number among A, B and E, (E,0,2) is handled first. Since E is a headline (i.e., both 0- and 1-backtrace stop line), stem objective (E,0,2) is updated and mapped to headline objective (E,0,1). Similarly, stem objective (B,0,1) is mapped to headline objective (B,0,1) and stem objective (A,0,1) is mapped to headline objective (A,0,1).

At this point, the sets of current and stem objectives are empty and the set of head objectives has three headline objectives: (A,0,1), (B,0,1) and (E,0,1). The three headlines A,B and C are assigned based on their corresponding objective values. Since n_1 (A) = 1 > n_0 (A) = 0, a value 1 is assigned to A. Similarly, a value 1 is assigned to B, and a value 1 is assigned to E.

Figure 5.11(d) shows the implication of the assigned headlines A,B and E. As shown in this figure, all the circuit POs (k and L) are assigned. Partial-CPT fault simulator is run to determine all the critical lines (i.e., detected faults) up to this point. (critical lines are shown by heavy lines in Figure 5.11(d)). For justifying the headlines assignment (A = 1 , B = 1 , E = 1), CLM-Multiple-Backtrce starts its backtracing process from each headline till PI lines (see Figure 5.11(e)). Since headlines A and B are PIs, A = 1 and B = 1 are automatically justified by mapping the head objectives (A,0,1) and (B,0,1) to primary input objectives (A,0,1) and (B,0,1). For justifying the assignment E = 1, this assignment is mapped to the current objective (E,0,1). At this point, the unassigned input C of G2 will be selected arbitrarily, since both of G2 inputs (C and D) are 0-useful and have same level numbers. So the new current objective is (C,1,0). Current objective (C,1,0) is mapped to primary input objective (C,1,0). At this point a value 0 is assigned to C (see Figure 5.11(f)), since n_0 (C) = 1 > n_1 (C) = 0. Partial-CPT fault simulator is

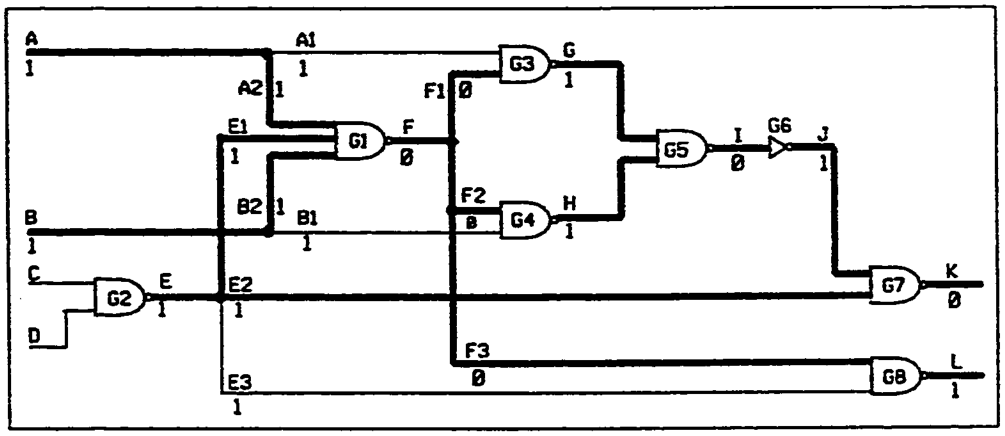


Figure 5.11(d) Implication and Simulation of Headlines (A,B,E) Assignment

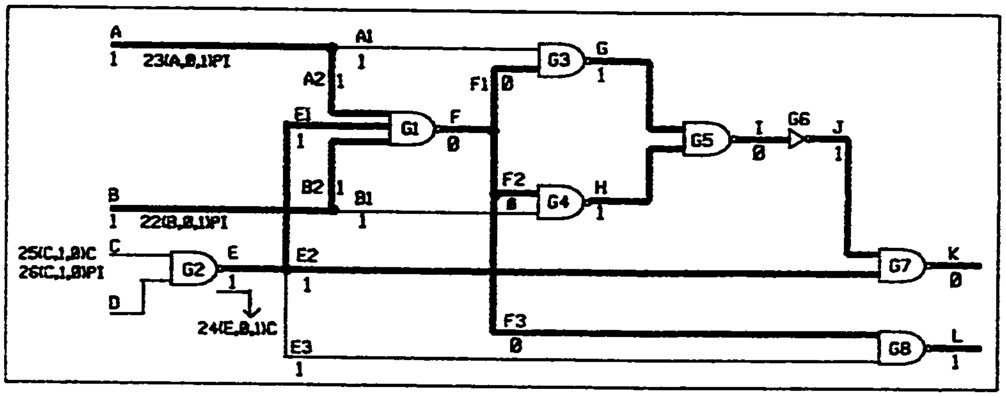


Figure 5.11(e) Justification of Headlines (A,B,E) Assignment

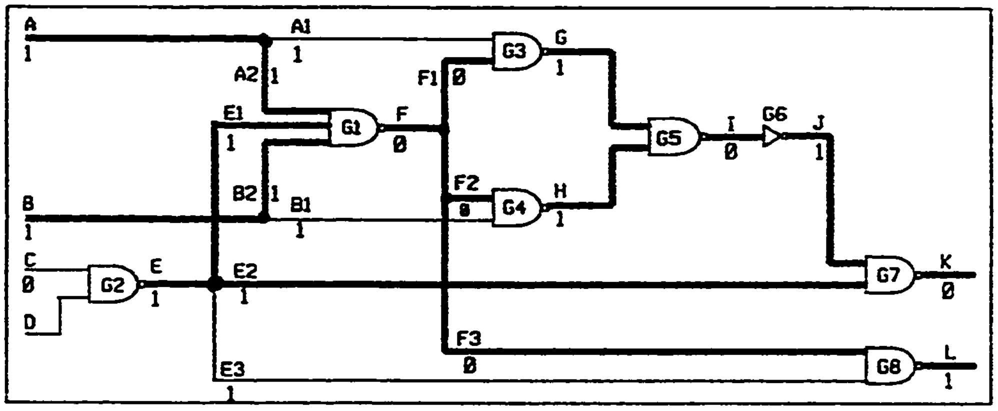


Figure 5.11(f) Implication and Simulation of Primary Inputs (A,B,C) Assignment

rerun. As a result of this run, gate G2 becomes a restart gate. So the CLM objective at G2 is to justify the noncontrolling value of G2 (i.e., 1) to D. The selected value is mapped to current objective (D,0,1) which is mapped to primary input objective (D,0,1), see Figure 5.11(g). At this point D is assigned a value 1. Figure 5.11(h) shows the implication of the assigned primary inputs A,B,C and D. Partial-CPT fault simulator is rerun to determine new critical lines (if any). So, as a result of D assignment and Partial-CPT run, PI C becomes critical, see Figure 5.11(h).

Since at this point non of the circuit PIs is unassigned, then the resulting test pattern r = 1101 is generated. Its simulation by CPT is shown in Figure 5.11(h). The faults detected by r are (A/0,A2/0,B/0,B2/0,C/1,E/0,E1/0,E2/0, F/1,F1/1,F2/1,F3/1,G/0,H/0,I/1,J/0,K/1,L/0). At this point 41% fault coverage is achieved.

Since PIs A, B are critical and are not 0-stop and PI C is critical and is not 1-stop, CPF flips the logic values of A, B, C one at a time. The new derived tests are (r1 = 0101, r2 = 1001 , r3 = 1111) and their simulation by CPT is shown in Figures 5.11(i), 5.11(j) and 5.11(k) respectively. The new detected faults by r1, r2 and r3 are (A/1, A2/1, B1/0, E3/0, F/0, F2/0, F3/0, H/1, I/0, J/1, K/0, L/1) , (A1/0,B/1,B2/1,F1/0,G/1) and (C/0,D/0,E/1,E3/1) respectively. Lines (A,A2,B,B2,C,E,E3) become 0-stop lines and lines (A,A1,A2,B,B1,B2,C,D) become 1-stop lines. At this point 87.6 % fault coverage is achieved.

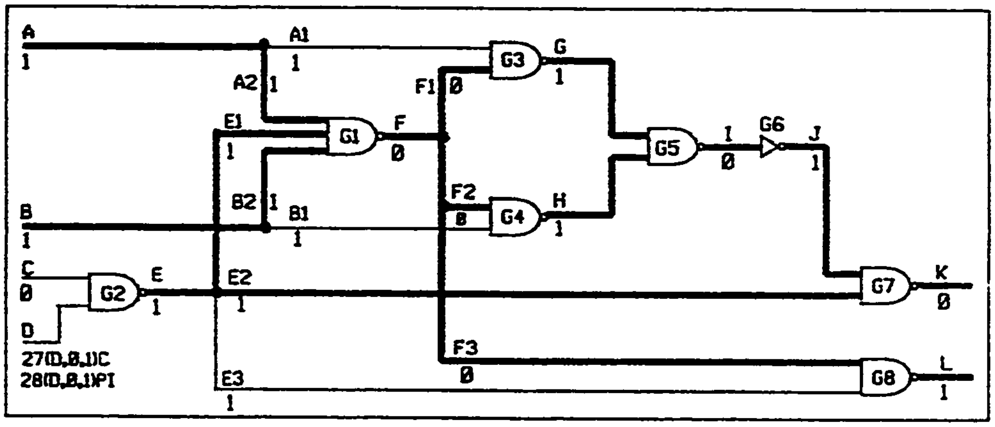


Figure 5.11(g) Handling of Restart Gate G2

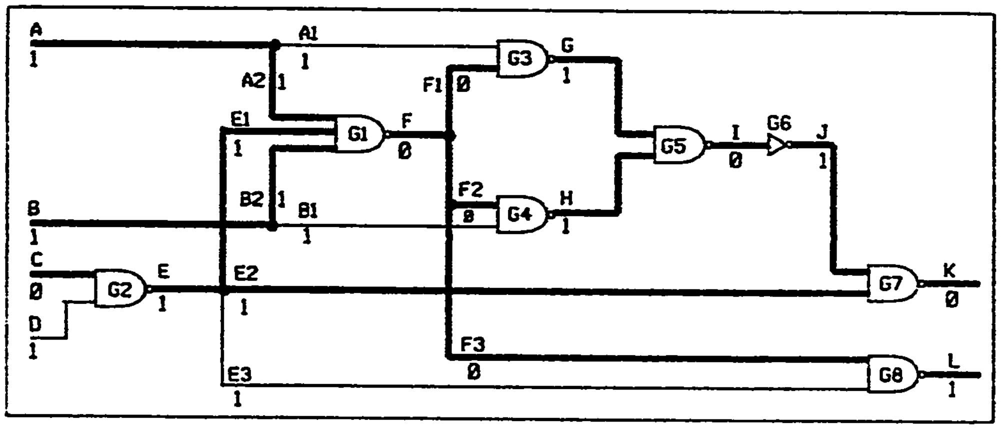


Figure 5.11(h) Generated Test Vector r = 1101

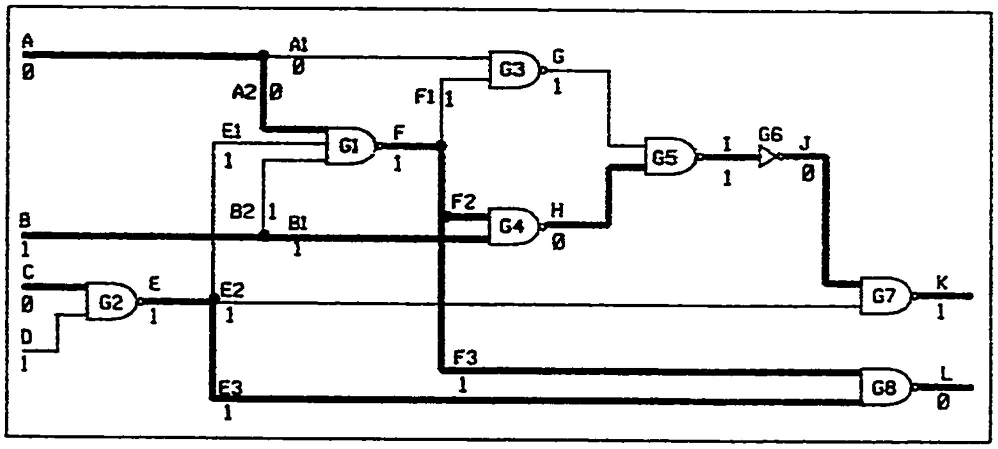


Figure 5.11(i) Simulation of Derived Test Vector r1 = 0101

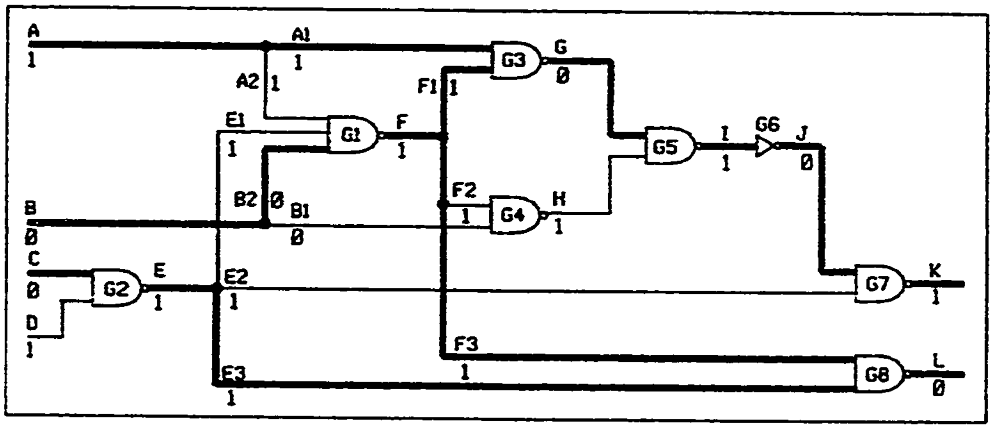


Figure 5.11(j) Simulation of Derived Test Vector r2 = 1001

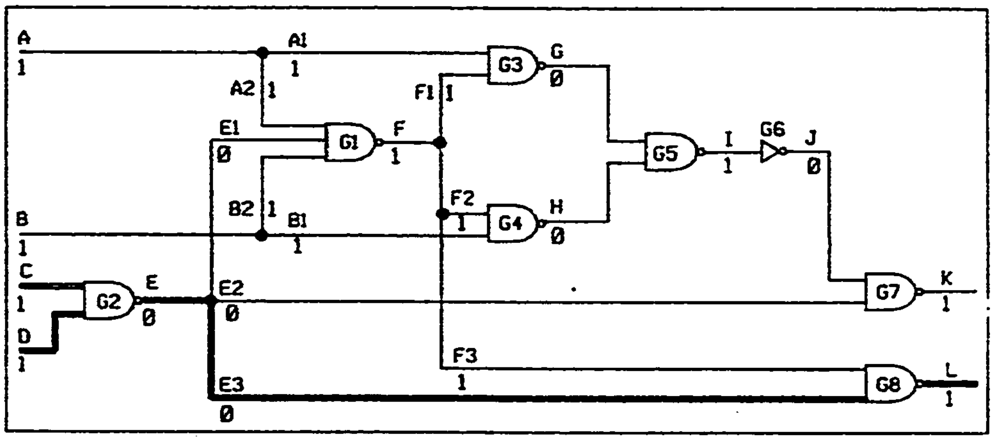


Figure 5.11(k) Simulation of Derived Test Vector r3 = 1111

For generating additional test patterns, all the circuit lines are re-initialized to X value and the process is repeated, except that the assignments of stop lines will be avoided. Since circuit POs (K and L) are still both 0- and 1-useful and K is in the set of Blocked_POs, L is the only candidate for selection, see Figure 5.2. L is added to set of Blocked_POs. Since gate G8 of PO line L is a NAND gate, the value $a = i \oplus \bar{c} = 1 \oplus 1 = 0$ is deterministically selected to PO line L. The selected value a = 0 of PO line L is mapped to the initial objective (L,1,0) which implies the following ordered set of current objectives: (E3,0,1) and (F3,0,1), see CMB rule 1(b).

Figure 5.11(l) shows all different objectives that appeared during the justification process of the initial objective (L,1,0). Based on LIFO stack policy, the current objective (F3,0,1) is handled first. (F3,0,1) is mapped to stem objective (F,0,1), see CMB rule 6. At this point the remaining current objective (E3,0,1) is handled. (E3,0,1) is mapped to stem objective (E,0,1).

At this point, the set of current objectives is empty and the set of stem objectives has two stem objectives (F,0,1) and (E,0,1). Since F has higher level number than E, (F,0,1) is handled next. Stem objective (F,0,1) is mapped to current objective (F,0,1). At this point, the unassigned input E1 of G1 is selected, since it is the only 0-useful input line among G1 input lines (A2, B2 and E1), see CMB rule 1(a). So the new generated current objective is (E1,1,0). (E1,1,0) updates stem objective (E,1,0) to (E,1,1), see CMB rule 6.

At this point, the set of current objectives is empty and the set of

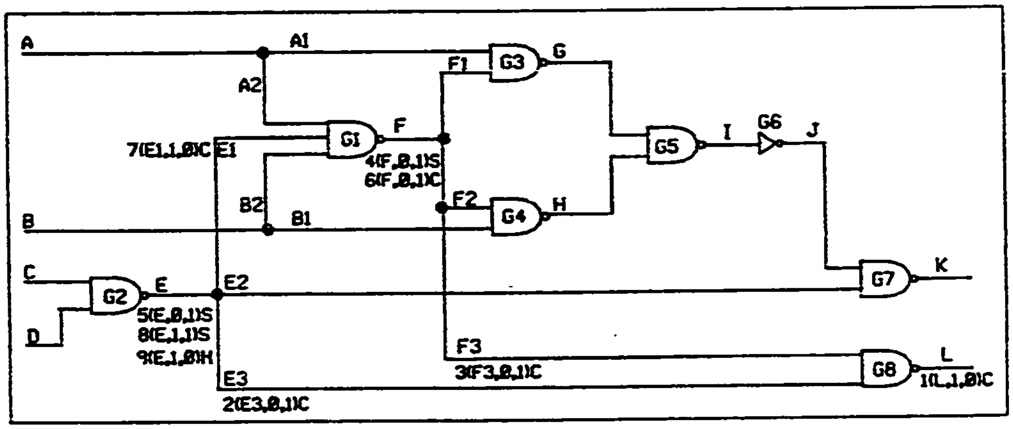


Figure 5.11(l) Justification of Initial Objective (L,1,0)

stem objectives has one stem objective (E,1,1). Since n_0 (E) and n_1 (E) are nonzero, a conflict has occurred. Since n_0 (E) $= 1 >= n_1$ (E) $= 1$, the conflict is resolved by updating (E,1,1) to (E,1,0), (see Figure 4.6). Since E is a headline (i.e., 0- and 1-backtrace stop line), (E,1,0) is mapped to headline objective (E,1,0).

At this point, the sets of current and stem objectives are empty and the set of head objectives has one headline objective (E,1,0). The headline E is assigned a 0 value based on its corresponding objective values.

Figure 5.11(m) shows the implication of the assigned headline E. As shown in this figure, all the circuit POs (k and L) are assigned. Partial-CPT is run to determine all the critical lines (i.e., detected faults) and restart gates up to this point. (critical lines are shown in Figure 5.11(m)). As a result of Partial-CPT run, gate G7 becomes a restart gate. So the CLM objective at G7 is to justify the noncontrolling value of G7 (i.e., 1) to J, see Figure 5.11(n). The selected value of J is mapped to current objective (J,0,1) which is mapped to current objective (I,1,0), see CMB rule 5. Current objective (I,1,0) implies ordered current objectives (G,0,1) and (H,0,1), see CMB rule 1(b). The current objective (H,0,1) is handled next. So the unassigned input B1 of G4 is selected, since it is the only unassigned input of G4, see CMB rule 1(a). So the new generated current objective is (B1,1,0). Current objective (B1,1,0) is mapped to the stem objective (B,1,0). The remaining current objective (G,0,1) is handled next. The objective (G,0,1) is mapped to the current objective (A1,1,0), since A1 is the only unassigned input of G3. The objective (A1,1,0)

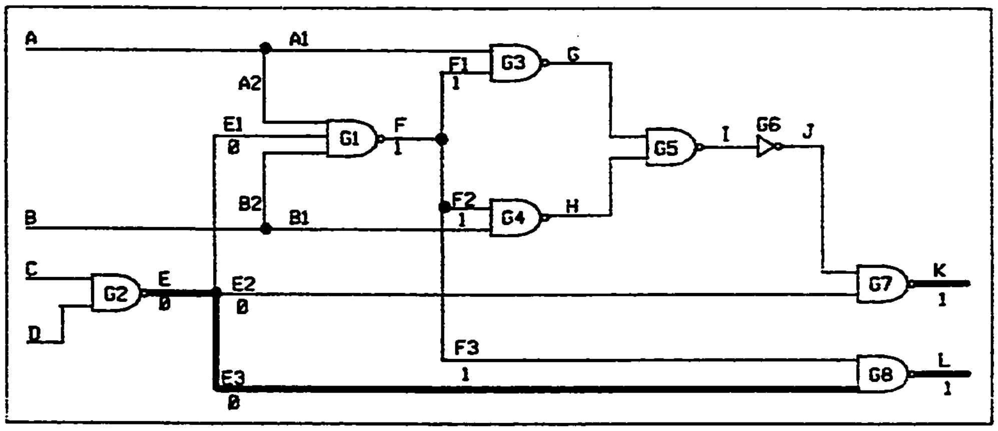


Figure 5.11(m) Implication and Simulation of Headline E Assignment

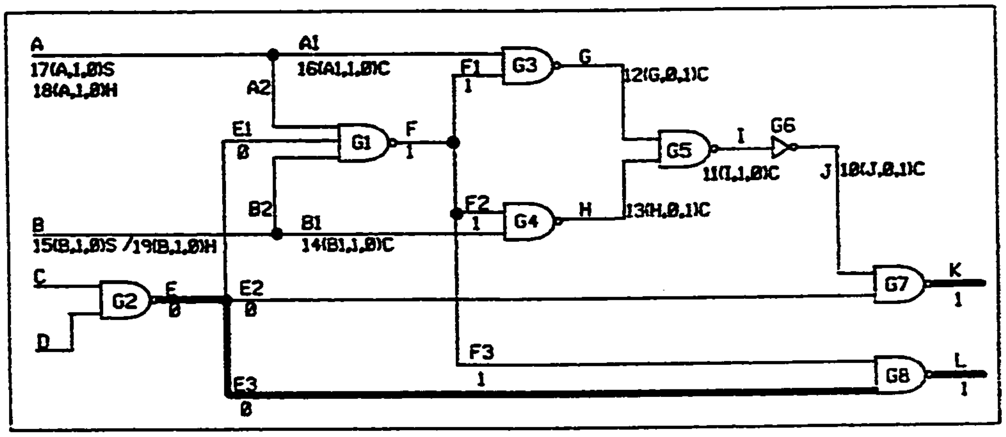


Figure 5.11(n) Handling of Restart Gate G7

is mapped to the stem objective (A,1,0).

At this point, the set of current objectives is empty and the set of stem objectives has two stem objectives: (A,1,0) and (B,1,0). Since A and B have same level number, any one of them can be handled first (say A). Since A is a headline, then stem objective (A,1,0) is mapped to headline objective (A,1,0). Similarly, stem objective (B,1,0) is mapped to headline objective (B,1,0).

At this point, the sets of current and stem objectives are empty and the set of head objectives has two headline objectives (A,1,0) and (B,1,0). Two headlines A and B are assigned based on their corresponding objective values. Since n_0 (A) = 1 > n_1 (A) = 0, a value 0 is assigned to A. Similarly, a value 0 is assigned to B. Figure 5.11(o) shows the implication of the new assigned headlines A and B. Partial-CPT fault simulator is rerun to determine new critical lines (if any). So as a result of A and B assignment and Partial-CPT run, E2 of G7 becomes critical. see Figure 5.11(o).

For justifying the headlines assignment (A = 0 , B = 0 , E = 0), CLM-Multiple-Backtrce starts its backtracing process from each headline till PI lines, see Figure 5.11(p). Since headlines A and B are PIs, A = 1 and B = 1 are automatically justified by mapping the head objectives (A,1,0) and (B,1,0) to primary input objectives (A,1,0) and (B,1,0). For justifying the assignment E = 0, this assignment is mapped to the current objective (E,1,0). Current objective (E,1,0) implies ordered current objectives (C,0,1) and (D,0,1). Current objective (C,0,1) is mapped to primary input objective (C,0,1). Also,

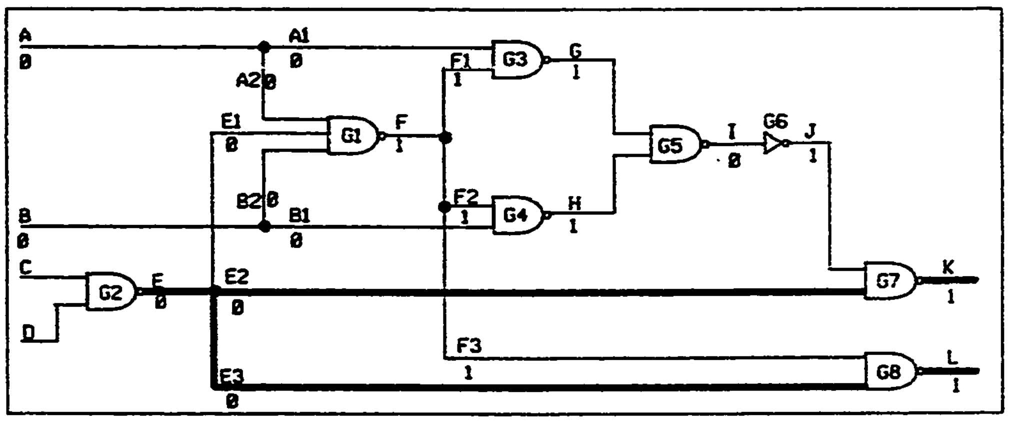


Figure 5.11(o) Implication and Simulation of Headlines (A,B) Assignment

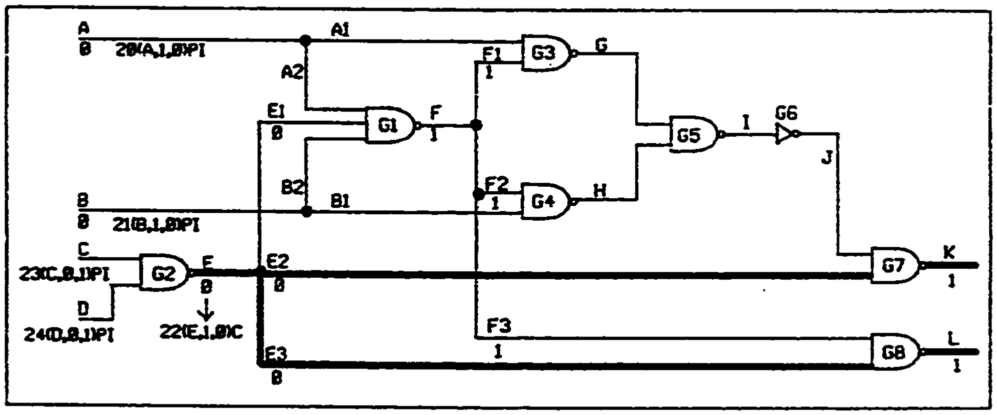


Figure 5.11(p) Justification of Headlines (A,B,E) Assignment

(D,0,1) is mapped to primary input objective (D,0,1). At this point all primary inputs are handled. So a value 1 is assigned to D, since n_1 (D) = 1 > n_0 (D) = 0. Similarly, a value 1 is assigned to C and a value 0 is assigned to both A and B. Figure 5.11(q) shows the implication of the assigned PIs A,B,C and D. Partial-CPT fault simulator is rerun to determine new critical lines (if any). So as a result of C and D assignments and Partial-CPT run, C and D become critical, see Figure 5.11(q). Since at this point non of the circuit PIs is unassigned, then the resulting test pattern s = 0011 is generated. Its simulation by CPT is shown in Figure 5.11(q). The new faults detected by s is (E2/1).

At this point, both of PIs C and D are critical but PI D is the only candidate for flipping. Since D is critical and is not 0-stop , CPF flips the logic value of D. The new derived test pattern is s1 = 0010 (see Figure 5.11(r) for its simulation by CPT). The new faults detected by s1 are (D/1,A1/1,B1/1). (At this point 97.73 % fault coverage is achieved).

Since the achieved fault coverage (97.73 %) exceeds the specified fault coverage (95 %), MAX is stopped at this point.

In what follows is a summary of test generation statistics of this circuit example:

1) Total number of single stuck at faults of this circuit example is 44.

2) Two test patterns are generated, r and s.

3) Three test patterns (r1,r2,r3) are derived from r.

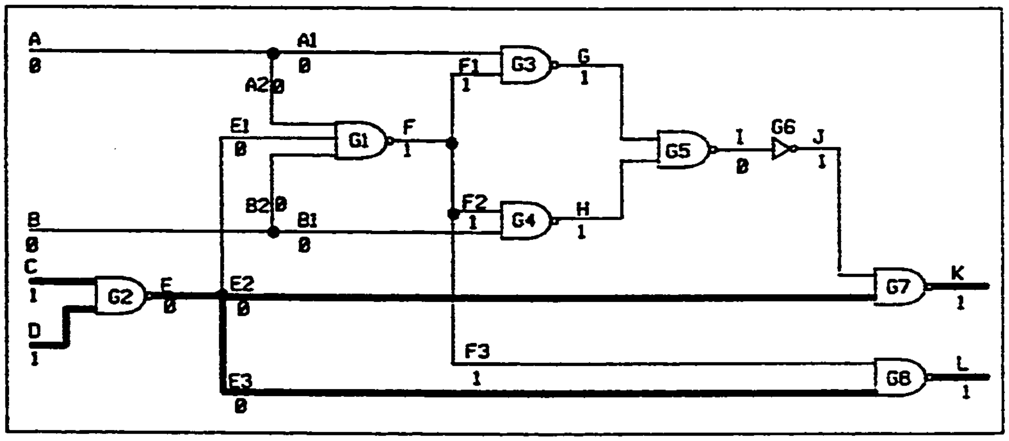


Figure 5.11(q) Implication and Simulation of Primary Inputs (A,B,C,D) Assignment

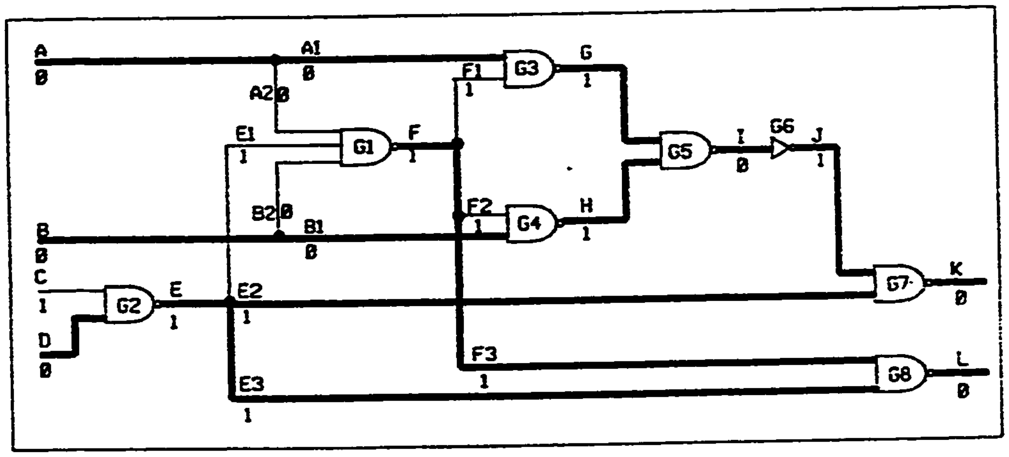


Figure 5.11(r) Simulation of Derived Test Vector s1 = 0010

4) One test pattern (s1) is derived from s.

5) 18 new faults are detected by r.

6) 12 new faults are detected by r1.

7) 5 new faults are detected by r2.

8) 4 new faults are detected by r3.

9) A total number of 39 faults are detected by r,r1,r2,r3, (i.e., 88.6% fault Coverage is achieved).

10) 1 new fault is detected by s.

11) 3 new faults are detected by s1.

12) A total number of 43 faults are detected by r,r1,r2,r3,s,s1, (i.e., 97.73 % fault Coverage is achieved).

CHAPTER VI

PERFORMANCE COMPARISONS

We summarize in this chapter the major differences between MAX and SMART [12]. Both MAX and SMART are implemented in C and run for the bench mark test circuits [11,39,40]. Discussions about the performance of both MAX and SMART is also provided.

6.1 Comparisons with SMART [12]

In this section, all the major differences between MAX and SMART [12] is highlighted. we show how the MAX features improve the fault independent test generation process. MAX is different from SMART in the following:

1) MAX starts by justifying a deterministically chosen value for a deterministically selected primary output (PO) line based on the PO gate type rather than justifying a randomly chosen value for a randomly selected PO as in SMART. The deterministic selection increases the chance of making more lines to be critical (i.e. more detected faults).

2) MAX uses CLM-Multiple-Backtrace instead of Selective-backtrace of SMART. Using CLM-Multiple-Backtrace accelerates the line justification process and gives chance to more faults to be detected.

3) MAX uses the CPF strategy for deriving test patterns from a generated one with a little additional effort.

6.2 Experimental Results

For purposes of comparison, we have implemented MAX and SMART [12] in C under ULTRIX on a VAX station 3100. All comparisons refer to these implementations.

We used the available ISCAS'85 benchmark combinational circuits on the VAX stations 3100 of our college (CCSE). These circuits were proposed by

Breglz [39,40]. In addition, a 4-bit Arithmetic and Logical Unit (ALU4) circuit proposed by Wei [11] is also used. Table 6.1 summarizes the characteristics of these circuits. All these circuits are used as test cases for comparing the performance of MAX and SMART.

Table 6.2 compares SMART and MAX runs that were allowed to work until generating more tests become no longer effective based on the average criteria of TGS function (i.e. the average number of new faults of the last 25 test pattern is less than 2). It is noted that all entries are normalized to the run time of MAX and all runs were done on VAX station 3100.

Tables 6.3 - 6.6 compare SMART and MAX runs that were allowed to proceed only until a specific fault coverage (i.e. 60%, 65%, 70% and 80%) is achieved.

Based on the tabulated results in Tables 6.2 - 6.6 the performance of MAX algorithm is generally better than the performance of SMART [12] for most of test circuits. For six out of eight bench mark test circuits, MAX shows better results than SMART. But for the other two circuits, C2670 and C5315, SMART shows better results than MAX. For similar fault coverages, the run time for the MAX algorithm is lower than the SMART algorithm. So based on the experimental results, MAX is more efficient than SMART, as it generates test sets in a shorter time for most of test circuits.

CIRCUIT NAME	TOTAL GATES	TOTAL LINES	INPUT LINES	OUTPUT LINES	TOTAL FAULTS
ALU4	88	241	14	8	482
C880	383	880	60	26	1760
C1355	546	1355	41	32	2710
C1908	880	1908	33	25	3816
C2670	1193	2670	233	140	5340
C3450	1669	3450	50	22	7080
C5315	2307	5315	178	123	10630
C7552	3512	7552	207	108	15104

Table 6.1 Characteristic of Test Circuits

CIRCUIT	ALGORITHM	TESTS	TESTS CLASIFICATION	FAULT COVERAGE	NORMALIZED RUN TIME
ALU4	MAX	24	7G + 17D	92.32	1
	SMART	30	30G	87.97	2.67
C880	MAX	67	6G + 61D	74.49	1
	SMART	63	63G	75.51	1.93
C1355	MAX	79	13G + 66D	88.93	1
	SMART	73	73G	67.68	1.37
C1908	MAX	53	13G + 40D	79.80	1
	SMART	47	47G	80.53	6.1
C2670	MAX	104	4G + 100D	68.35	1
	SMART	67	67G	81.26	3.94
C3450	MAX	131	16G + 115D	80.82	1
	SMART	141	141G	81.38	11.2
C5315	MAX	186	6G + 180D	76.39	1
	SMART	142	142G	93.74	3.72
C7552	MAX	235	6G + 229D	78.09	1
	SMART	290	290G	70.31	2.6

G: A Generated Test Vector by MAX/SMART
D: A Derived Test Vector by CPF of MAX

Table 6.2 Comparisons Between SMART and MAX Based on the Average Criteria

CIRCUIT	ALGORITHM	TESTS	TESTS CLASIFICATION	NORMALIZED RUN TIME	FAULT COVERAGE
ALU4	MAX	10	1G + 9D	1	60%
	SMART	7	7G	2.84	
C880	MAX	36	1G + 35D	1	
	SMART	21	21G	1.43	
C1355	MAX	38	2G + 36D	1	
	SMART	52	52G	3.84	
C1908	MAX	16	1G + 15D	1	
	SMART	12	12G	2.5	
C2670	MAX	94	3G + 91D	1.4	
	SMART	13	13G	1	
C3450	MAX	54	3G + 51D	1	
	SMART	45	45G	6.1	
C5315	MAX	125	2G + 123D	1.53	
	SMART	11	11G	1	
C7552	MAX	140	1G + 139D	1	
	SMART	183	183G	3.05	

Table 6.3 Comparisons Between SMART and MAX Based on the 60 Percentage Fault Coverage Criteria

CIRCUIT	ALGORITHM	TESTS	TESTS CLASIFICATION	NORMALIZED RUN TIME	FAULT COVERAGE
ALU4	MAX	10	1G + 9D	1	65%
	SMART	8	8G	3.84	
C880	MAX	44	3G + 41D	1	
	SMART	30	30G	1.7	
C1355	MAX	39	2G + 37D	1	
	SMART	66	66G	4.33	
C1908	MAX	25	1G + 24D	1	
	SMART	14	14G	2.17	
C2670	MAX	103	4G + 99D	1.21	
	SMART	20	20G	1	
C3450	MAX	66	3G + 63D	1	
	SMART	55	55G	6.2	
C5315	MAX	147	2G + 145D	1.56	
	SMART	16	16G	1	
C7552	MAX	146	2G + 144D	1	
	SMART	213	213G	3.77	

Table 6.4 Comparisons Between SMART and MAX Based on the 65 Percentage Fault Coverage Criteria

CIRCUIT	ALGORITHM	TESTS	TESTS CLASIFICATION	NORMALIZED RUN TIME	FAULT COVERAGE
ALU4	MAX	14	2G + 12D	1	70%
	SMART	10	10G	3.4	
C880	MAX	53	4G + 49D	1	
	SMART	39	39G	1.83	
C1355	MAX	41	2G + 39D	1	
	SMART[1]	73	73G	> 4.7	
C1908	MAX	33	2G + 31D	1	
	SMART	20	20G	2.4	
C2670	MAX[1]	104	4G + 100D	> .94	
	SMART	32	32G	1	
C3450	MAX	79	4G + 75D	1	
	SMART	73	73G	7	
C5315	MAX	167	4G + 163D	1.52	
	SMART	23	23G	1	
C7552	MAX	214	2G + 212D	1	
	SMART	269	269G	3.96	

1: It Does not Reach 70% Fault Coverage

Table 6.5 Comparisons Between SMART and MAX Based on the 70 Percentage Fault Coverage Criteria

CIRCUIT	ALGORITHM	TESTS	TESTS CLASIFICATION	NORMALIZED RUN TIME	FAULT COVERAGE
ALU4	MAX	19	3G + 16D	1	80%
	SMART	21	21G	5.1	
C880	MAX[1]	67	6G + 61D	1	
	SMART[1]	63	63G	1.93	
C1355	MAX	59	7G + 52D	1	
	SMART[1]	73	73G	> 2.6	
C1908	MAX	53	13G + 40D	1	
	SMART	47	47G	6	
C2670	MAX[1]	104	4G + 100D	> 0.33	
	SMART	58	58G	1	
C3450	MAX	126	14G + 112D	1	
	SMART	125	125G	10.6	
C5315	MAX[1]	186	6G + 180D	> 1.33	
	SMART	40	40G	1	
C7552	MAX[1]	--	--	--	
	SMART[1]	--	--	--	

1: It Does not Reach 80% Fault Coverage

Table 6.6 Comparisons Between SMART and MAX Based on the 80 Percentage Fault Coverage criteria

CHAPTER VII

CONCLUSIONS

We summarize in this chapter the results obtained in this thesis. Suggestions for future work on this line of investigation is provided.

7.1 Summary of Results

In this thesis, the following results have been obtained.

1) Two new strategies, CLM and CPF, are proposed. These strategies improve the performance of the fault independent test patterns generation algorithms.

2) A new backtrace procedure is presented, called CLM-Multiple-Backtrace, which is based on CLM strategy as well as the multiple backtrace of FAN [28].

3) A new fault independent test generation algorithm, called MAX, is proposed based on CLM-Multiple-Backtrace as well as many other efficient features (i.e. strategies, procedures and functions) of the existing fault independent and fault orinted test generation algorithms.

4) Experimental results show that MAX is more efficient than the existing fault independent test generation algorithms given in the literature, as it generates test sets in a shorter time for most of test circuits.

7.2 Future Work

As a continuation of this line of investigation, the proposed algorithm MAX can be integrated into an automatic test patterns generation (ATPG) system with one of the most efficient fault oriented testing algorithms such as D-Algorithm [22], PODEM [27], FAN [28] , FAST [12] , TOPS [29] and SLOPE [30]. The performance of the new ATPG system can be compared with the performance of other ATPG systems such as LASAR [8], PODEM-X [9] , MAHJONG [11] and LAMP2 [12].

REFERENCES

[1] E. J. McCluskey, Logic Design Principles, Prentice-Hall 1986.

[2] R. G. Bennetts, Design of Testable Logic Circuits, Addison-Wesley 1984.

[3] K. C. Mei, "Bridging and Stuck-At Faults," IEEE Transactions on Computers, Vol. C-27, pp.720-727, July 1978.

[4] J. W. Gualt, J. P. Robinson, and S. M. Reddy, "Multiple Fault Detection In Combinational Networks, " IEEE Transactions on Computers, Vol. C-21, pp.31-36, January 1972.

[5] J. P. Hayes, "Detection of Pattern-Sensitive Faults in Random Access Memories," IEEE Transactions on Computers, Vol. C-24, pp.150-157 , Feburary 1975.

[6] V. K. Agrawal and S. F. Fung, "Multiple Fault Testing of Large Circuits by Single Fault test sets," IEEE Transactions on Computers, Vol. C-30, pp.855-865, November 1981.

[7] V. K. Agrawal and G. M. Masson, "Generic Fault Characterizations for Table Look-Up Bounding," IEEE Transactions on Computers, Vol. C-29, pp.288-299, April 1980.

[8] J. J. Thomas, "Automated Diagnostic Test Programs for Digital

Networks," Computer Design, pp.63-67, August 1971.

[9] B. C. Rosales, "PODEM-X : An Automatic Test Generation System for VLSI Logic Structures," Proceedings 18th Design Automation Conference, pp.260-268, 1981.

[10] E. J. McCluskey, "Verification Testing-A Pseudo Exhaustive Test Technique," IEEE Transactions on Computers, Vol. C-33, pp.541-546, June 1984.

[11] R. S. Wei, "Logic Verification and Test Generation for VLSI Circuits," Ph.D. dissertation, EECS Department, University of California at Berkely, November 1, 1986.

[12] M. Abramovici, J. J. Kulikowski, P. R. Menon and D. T. Miller, "Test Generation in LAMP2: Concepts and Algorithms", Proceedings of IEEE International Test Conference, pp.49-56, 1985.

[13] F. F. Sellers, M. Y. Hiso and C. L. Bernson, "Analyzing Errors with the Boolean Difference," IEEE Transactions on Computers, Vol. C-17, pp.676-683, July 1968.

[14] J. F. Poage, "Derivation of Optimum Tests to Detect Faults in Combinational Circuits". Mathematical Theory and Automation. New York: Polytechnic Press 1963.

[15] D. B. Armstrong, "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets", IEEE Transactions on Computers,

Vol. C-15, pp.66-73, February, 1966.

[16] D. C. Boseen and S. J. Hong, "Cause and Effect Analysis for Multiple Fault Detection in Combinational Networks," IEEE Transactions on Computers, Vol. C-20, pp.1252-1257, November 1971.

[17] F. W. Clegg, "Use of SPOOFs in the Analysis of Faulty Logic Network," IEEE Transactions on Computers, Vol. C-22, pp.229-234, March 1973.

[18] K. Y. Kinoshita, Y. Takamstsu, and M. Shibata, "Test Generation for Combinational Circuits by Structure Description - Functions," Proceedings 10th International Symposium Fault Tolerant Computing, pp.152-154, 1980.

[19] M. Y. Osman and M. M. Al-Deeb, "A Tabular Method for Generating The Boolean Difference and Its Application for Testing Logic Circuits," To be appear.

[20] R. D. Eldred, "Test Routines based on Symbolic Logic Statements," J. Ass. Comput. Mach., Vol. 6, No. 1, pp.33-36, 1959.

[21] H. Y. Chang, E. G. Manning, and G. Metze, Fault Diagnosis of Digital Systems, New York : Willey Interscience, 1970.

[22] J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," IEEE Transactions on Computers, Vol. C-15, pp.278-291, July 1966.

[23] J. P. Roth, W. G. Bouricius, and P. R. Cshneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish between Failures in Logic Circuits," IEEE Transactions on Computers, Vol. C-16, pp.567-580, October 1967.

[24] C. W. Cha, W. Donath, and F. Ozguner, "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits." IEEE Transactions on Computers, Vol. C-27, pp.193-200, March 1978.

[25] S. B. Akers, "A Logic System for Fault Test Generation." IEEE Transactions on Computers, Vol. C-25, pp.620-629, June 1976.

[26] Y. M. Takamatsu, M. Hiraishi, and K. Kinoshita, "Test Generation for Combinational Circuits by a Ten-Valued Calculus," Transactions of Information Processing Society of Japan, 24(4), pp.542-548, 1983.

[27] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," IEEE Transactions on Computers, Vol. C-30, pp.215-222, March 1981.

[28] H. Fujiwara and T. Shimono, " On the Acceleration of Test Generation Algorithms," IEEE Transactions on Computers, Vol. C-32, pp.1137-1144, December 1983.

[29] T. Kirkland and M. R. Mercer, "A Topological search algorithm for ATPG", IEEE Design Automation Conference, Paper 28.2, pp.502-508,

1987.

[30] Shih-Jen Chuang, Chung-Len Lee, Wen-Zen Shen. Chein-Wei Jen, Jwu-E Chen, Sen-Chung Jiang, and Ming-Der Chen. "SLOPE: A Test Pattern Generator based on Stop Line Oriented Path End Algorithm," Proceedings of IEEE International Symposium of Circuits and Systems, pp.437-439, 1988.

[31] A. Lioy, "Adaptive Backtrace and Dynamic Partitioning Enhance ATPG," Proceedings of IEEE International Symposium of Circuits and Systems , pp.62-65, 1988.

[32] Hao-Yung Lo and Chien-Chun Su, "A Distributive D-Algorithm for Generating the Test Pattern for Faulty Combinational circuit," International Journal of ELectronics, Vol. 66, No. 1, pp.35-42, 1989.

[33] S. Patil and P. Banerjee. "A Parallel Branch and Bound Algorithm for Test Generation," IEEE Transactions on Computer-Aided Design, Vol.9, No. 3, pp.313-320, March 1990.

[34] E. G. Ulrich and T. Baker, "The Concurrent Simulation of Nearly Identical Digital Networks," In Proc. 10th Design Automation Workshop, pp.145-150, June 1973.

[35] H. D. Schnurmann, E. Lindbloom, and R. G. Carpenter, "The Weighted Random Test Pattern Generation," IEEE Transactions on Computers , Vol. C-24, pp.695-700, July 1975.

[36] P. Goel, "RAPS Test Pattern Generator," IBM Technical Disclosure Bulletin, Vol. 21 No.7, pp.2787-2791, December 1978.

[37] R. A. Rutman, "Fault Detection Test Generation for Sequential Logic by Heuristic Tree Search," IEEE Computer Group Repository, Paper No. R-72, pp.187, 1972.

[38] M. Abramovici, P. R. Menon, and D. T. Miller, "Critical Path Tracing : An Alternative to Fault Simulation," IEEE Transactions on Computers, Vol. C-33, pp.83-93, February 1984.

[39] F. Brglez and F. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," Proceedings of IEEE International Symposium of Circuits and Systems, pp. 705-712, 1985.

[40] B. C. Rosales and P. Goel," Results from Application of a commercial ATP System to Large-Scale Combinational Circuits," Proceedings of IEEE International Symposium of Circuits and Systems, pp.667-670, 1985.

[41] M. A. Breuer and A. D. Friedman, Diagonsis and Reliable Design of Digital Systems, Computer Science Prss, Potomac, MD, 1976.

[42] F. Brglez, P. Pownal and R. Hum, "Applications of Testability Analysis: from ATPG to Critical Delay Path Tracing," Proceeding of IEEE International Test Conference, pp.705-712, 1984.

[43] L. H. Goldstein, "Controllability / Observability Analysis for Digital Circuits," IEEE Transactions on Circuits and Systems, CAS-26, No.9, pp.685-693, 1979.

[44] I. M. Ratiu, "VICTOR: A Fast VLSI Testability Analysis Program," Proceedings IEEE International Test Conference, Paper 13.4, pp.397-401, 1982.

[45] A. Ivanon and V. K. Agarawal, "Testability Measures - What They Do for ATPG?," Proceedings IEEE International Test Conference, pp.129-138, 1986.

[46] H. Fujiwara , Logic Testing and Design for Testability, MIT Press, 1990.

[47] J. P. Hayas, "Transiition Count Testing of Combinational Logic Circuits," IEEE Transactions on Computers, Vol. C-25, pp. 613-620, June 1976.

[48] G. Markowsky, "Syndrome-Teasability Can Be Achieved by Circuit Modification," IEEE Transactions on Computers, Vol. C-30, pp. 604-606, August 1981.

[49] T. C. Hiso and S. C. Seth, "An Analysis of the Use of Rademacher-Walsh Spectrum in Compact Testing," IEEE Transactions on Computers, Vol. C-33, pp. 934-937, October 1984.

[50] K. S. Bhaskar, "Signatrue Analysis: Yet Another Perspective," Digest of Papers 1982 international Test Conference, pp. 132-134, November 1982.