COMBINATIONAL LOGIC CIRCUITS FOR WHICH TESTS CAN BE GENERATED IN N^2 TIME

BY

BIJAN KARIMI
Bachelor of Science
Aryamehr University of Technology
Tehran, Iran
1977

Master of Science
Oklahoma State University
Stillwater, Oklahoma
1981

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
DECEMBER, 1985

COMBINATIONAL LOGIC CIRCUITS FOR

WHICH TESTS CAN BE GENERATED

IN N^2 TIME

Thesis Approved:

Louis G. Johnson
Thesis Adviser

[signature]

David L. Soldan

John Wolfe

Norman N. Durham
Dean of the Graduate College

ACKNOWLEDGMENT

I would like to thank Dr. Louis Johnson, my advisor, Dr. Rao Yarlagadda, chairman of my committee, Dr. David Soldan, and John Wolfe for all the help they gave me.

I dedicate this piece of work to my wonderful wife Taraneh, my wonderful son Abteen, and the memory of my wonderful father.

TABLE OF CONTENTS

LIST OF TALBES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Digital systems are subject to physical faults during their life time. With the increased complexity of digital systems, with huge numbers of elements in an IC chip, the problem of testing digital systems for reliable performance has become more important. In general, a fault in a system can be considered as anything which makes a system to behave in a different way than for which it was designed. Faults can occur during manufacture, assembly, storage, or operation. Faults which alter circuit parameters such as current, voltage, or speed are known as parametric faults. Faults which alter logical behavior of a circuit are known as logical faults. Since faults can occur in a system at any time, the system must be tested during its life time.

Testing consists of applying a set of logical values to inputs of a circuit and observing the output to see if it is different from what was expected. To test a circuit there must be a fault model to identify the period of time that the fault will be present, the number of such faults present at the same time, and the effect of the fault on operation of the circuit. The most common logical fault model is a single permanent stuck-at model which assumes a line in the circuit is permanently stuck-at-logic zero or logic one (this model will be used throughout this research study).

One way to test a circuit is to apply all possible input combinations and observe the output. This method is not reasonable to apply for circuits with large numbers of inputs because possible input combinations grow exponentially with increasing numbers of inputs. Then it is desirable to find a subset of input combinations which detects all faults in the circuits. In a circuit consisting of N lines (N includes primary inputs and outputs, and internal lines) there are 2N single stuck-at-0/1 type faults.

Attention in this research study is focused on combinational circuits. Basic elements of these kinds of circuits are called gates and there are no feedbacks or memory elements in combinational circuits. Different gates under consideration will be AND, OR, NAND, NOR, and INVERTER. Because of the complex topology (interconnection of lines) that combinational circuits may have, there is no known algorithm which generates tests in polynomial time for an arbitrary combinational circuit. In general, it is accepted that there exist no such algorithms. With the growing number of gates on a single chip, even high order (greater than 2) polynomial in time algorithms are not desirable. Then it becomes important that a designer designs a circuit in such a topological form for which tests can be generated in N or N^2 time.

Redundancy is one of the reasons that test generation is time consuming. A circuit is redundant if one or more lines of it cannot be tested. Redundancy is an unwanted feature in most designs and a good design rarely suffers this problem except in fault tolerant systems. Then if topologies for irredundant circuits can be identified which make circuits testable in N^2 time, a designer may keep circuit topology close to those identified topologies and save a great deal of time in the test

generation process. It would even be more desirable if a method for design can be found which makes any circuit testable in N^2 time. An effort has been made in this research to identify the topology of circuits which can be tested in N^2 time. Also a design method is introduced which makes any circuit (even redundant circuits) testable in N^2 time. A program has been written which generates tests for circuits designed according to the proposed design method.

During test generation for a given fault it is possible that a value assignment on a line be inconsistent with other value assignments in the circuit. Then that value assignment must be removed and another choice must be considered. This process is called backtracking. It is this process which makes the process of test generation exponential in time because without the need for backtracking, each single stuck type fault can be detected by at most N value assignments in the circuit. If there are no reconvergent paths in the circuit then there will be no need for backtracking. For this reason reconvergent paths are the main subject of this research. Every two reconvergent paths will be referred to as a loop.

In this research an attempt has been made to identify relative positions and properties of the loops for which circuit can be tested in time proportional to N^2. The most general topology which has been identified with the above property in this research consists of reconvergent paths which do not reconverge on more than one gate if they originated from the same fanout origin, and they do not share gates with other reconvergent paths if there is no path between their fanout origins. Also it is shown that any circuit can be tested in time proportional to N^2 if certain gates and inputs, called blocking gates and

control inputs respectively, are placed in specific locations in the circuit.

CHAPTER II

REVIEW OF RELATED LITERATURE

The two most widely used methods of test generation for single permanent stuck at logic values (0/1) are the D-algorithm (Roth, 1966) and critical path sensitization (Thomas, 1971). These methods use the path sensitization concept to propagate a fault signal from the sight of the fault to the output(s) of the circuit under test, where it can be compared with the expected value in the normal circuit. Since this concept will be used throughout this research study, it will be reviewed here. Consider Figure 1 and the fault line "a" stuck at 0 (a s-a-0). In fact it must be determined if this line can be set to a logic 1. For this purpose a logic value 1 must be assigned to line "a". Other lines in the circuit must be set to values such that the effect of the value assignment of line "a" can be seen on the output of the circuit. In other words the fault signal can be propagated to the output. To achieve this goal, line 5 must be set to logic 1. If this line is set to 0, then the output of the gate 14 will be 0 regardless of the value assignment on line "a". For the similar reason line 8 must be set to logic 1. With these value assignments the value of the output will be 0 if line "a" is not stuck at 0 and the value of the output will be 1 if line "a" is stuck at 0. This process is called path sensitization. To generate a test for this fault (an input vector); logic values on the inputs of the circuit must be determined such that they set the internal

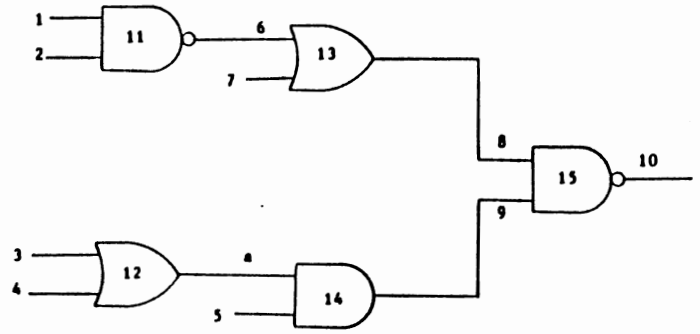


Figure 1. Path Sensitization and Line Justification

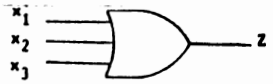


Figure 2. Example for Controllability and Observability

lines of the circuit to the desired values found in the path sensitization process. This is called a justification process. In order to have a 1 on line "a", either line 3 or line 4 can be set to 1. To have a 1 on line 8, either line 6 or line 7 can be set to 1. If line 6 is selected then either line 1 or line 2 (or both) must be set to 0.

It is known that the following fault detection problems:

1. Can all single faults be detected in a combinatioal circuit (is the circuit irredundant)?

2. Can a fault in a particular input line x_i be detected by input-output experiments?

3. Can all single input faults be detected by I/O experiments?

4. Can faults in the output line be detected by I/O experiments?

are NP-complete (Ibarra and Sahni, 1975), i.e. there is a polynomial time algorithm to decide if the above single faults are detectable if and only if there is a polynomial time algorithm for problems such as the traveling salesman problem. Then it seems very unlikely that a polynomial time algorithm can be found (in terms of the number of inputs, gates, or lines) to detect single faults. In fact, it would appear that only algrithms with a computing time linear or at most a square of the number of input lines and gates would be feasible for large combinational circuits (Ibarra and Sahni, 1975). Even for relatively simple circuits such as monotone and unate circuits these problems are NP-complete if the numbers of levels in those circuits are greater than 2 (Fujiwara and Toida, 1982). A circuit is said to be monotone if all the variables appear unnegated in the expression describing the function of the circuit. A circuit is said to be unate

if all the variables appear either negated or unnegated.

In the processes of path sensitization and line justification, it is possible that a test generation algorithm has to select a choice among several choices. Also it is possible that some or all of those choices lead to conflicting assignments of values to nodes in the circuits. Then the algorithms has to backtrack and try different choices until either a test is generated or there is no choice left. This backtracking is the reason that the time complexity of test generation algorithms is exponential, because in general an algorithm has to try an exponential number of value assignment combinations until it finds a test. Sometimes there is no test for a certain fault. In this case the circuit is said to be redundant. If a circuit is not redundant then it is called irredundant (Breuer and Friedman, 1976) or nonredundant. Test generation for redundant circuits is more time consuming because all possible choices must be tried by the algorithm before it can decide that no test exists for a certain fault.

A great deal of work has been done to simplify the process of test generation for logic circuits and several methods of design for testability have been proposed since 1970. There are two key concepts in design for testability, controllability and observability (Williams and Parker, 1983). Controllability is the ability to apply test patterns to internal circuitry by exercising the input pins of that circuitry. Observability is the ability to determine the internal states of a circuit by observing the output pins. All methods of design for testability try to enhance the controllability and observability of a system by some means. To appreciate the problem consider the simple OR gate in Figure 2. In order to generate a test for the input fault X_1 s-a-0, it

is necessary to control X_2 and X_3 to 0 and X_1 to 1. Also it is necessary that z can be observed to determine if this fault actually exists in the circuit. In this case it is possible to control the inputs to the desired values and observe the output. In general it is not always possible to control a line in a circuit or observe states of a circuit on the output. For example in Figure 3, line "a" cannot be controlled to 1 and the effect of any value assignment on "b" cannot be observed from the output. One way to enhance controllability and observability of a circuit is to use test points. If a test point is used as a primary input to the network, then that functions to enhance controllability. If a test point is used as a primary output, then that can be used to enhance the observability of a network. In Figure 3, if the test point "c" is added to the circuit then, the value on line "b" can be observed through "c". The use of input test points has been discussed by Hayes (1974) for circuits consisting of 2-input NAND gates and inverters. An example of the circuits consisting of NAND gates and inverters is shown in Figure 4. Hayes has discussed that if a circuit with a structure like this is changed to another circuit according to the following rules:

1. Every inverter is replaced by an EX-OR gate while the other input of the EX-OR gate is connected to logic 1 for normal operation of the circuit.

2. Each NAND gate has only two input lines.

3. One EX-OR gate is placed on input lines of each NAND gate if no inverter is preceding that line. The other input of the EX-OR gate is connected to logic 0 for the normal operation of the circuit.

then the resulting circuit needs only five tests for complete testing of

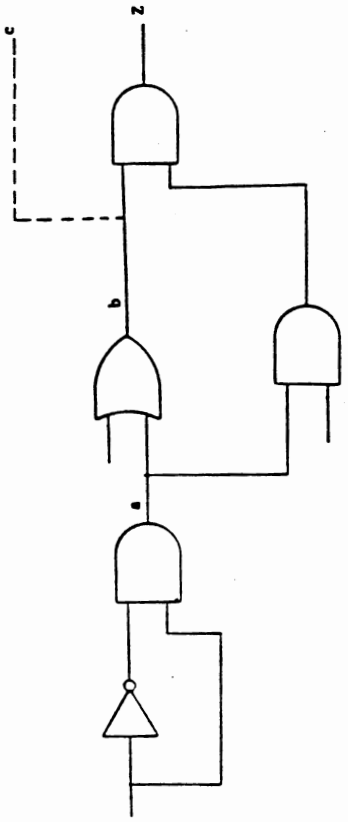


Figure 3. Example for Uncontrollable and Unobservable Points

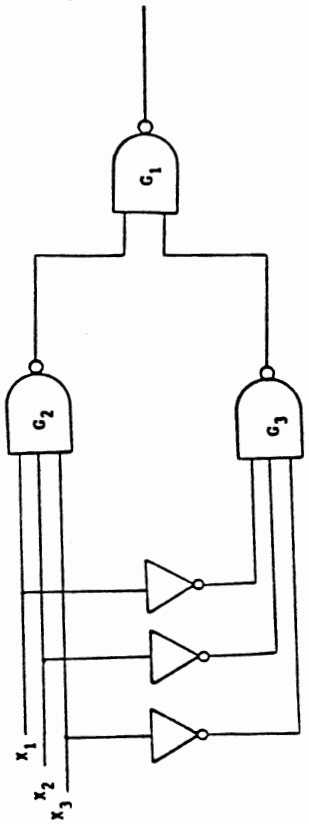


Figure 4. Example of Circuits Consisting of NAND Gates and INVERTER's

the circuit single stuck at faults. The basic building block of such circuits is shown in Figure 5, and the circuit in Figure 4 is shown in Figure 6 after modifications. The second inputs of the EX-OR gates are used as control inputs to put desired values for test generation on the internal lines of the circuit. Drawbacks in this method are that a circuit must be changed to a circuit with the properties mentioned before and a great amount of circuitry must be added to the circuit.

Another method of design for testability is partitioning. Goel (1980) has shown by imperical results that the computer run time to do test generation is approximately proportional to the number of logic gates used in a circuit to the power of 3. Then partioning a circuit into modules which can be tested seperately seems to decrease the time required for test generation (Williams and Parker, 1983). Drawbacks for this method are cost, space, and it is in contradiction with the purpose of integration.

Another method of design for testability, which has received much attention, is Level Sensitive Scan Design (Berglund, 1978). This method of design for testability is for sequential circuits but it is important to be mentioned here because it reduces the complexity of the test generation to that for combinational circuits. This design methodology also uses the concept of controlling inputs. The only type of storage element permitted in this technique is a shift register latch (SRL), which is a pair of D flip-flops, as shown in Figure 7. The output of the first latch (L_1) serves as data input to the second latch (L_2). L_1 is used as storage element and L_2 is used to enhance testing of the circuit. The D input of L_1 comes from the output of a L_2 and the output of L_2 is an input to another L_1. Then all latches in the circuit are

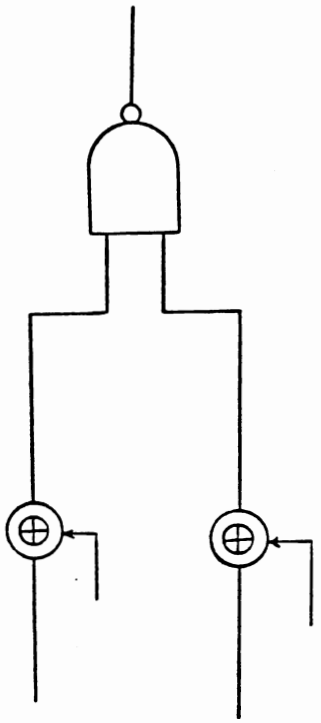


Figure 5. Basic Building Block

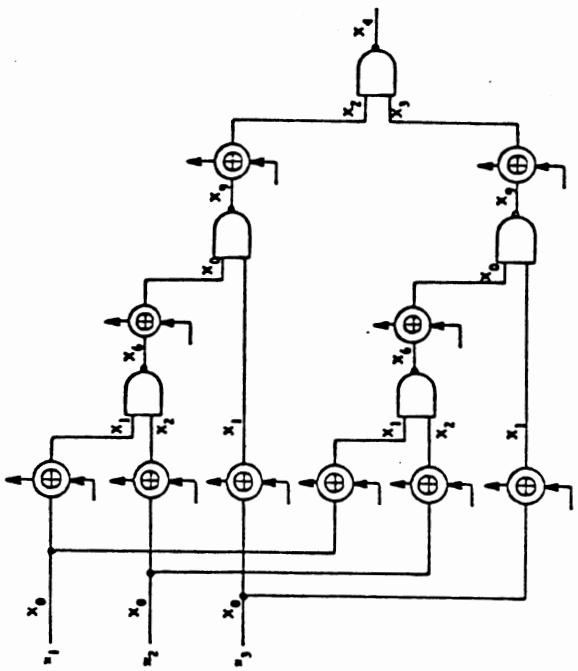


Figure 6. Example of Circuits Consisting of NAND Gates with Added Control Inputs

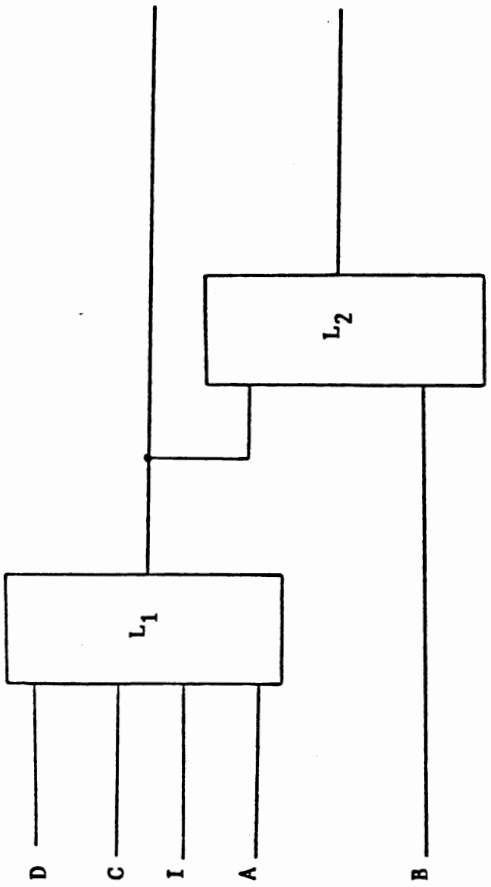


Figure 7. Shift Register Latch

chained together by this scheme. The first L_1 in this chain is connected to an input pin called SDI (Scan Data In) and the output of the last L_2 is connected to an output pin called SDO (Scan Data Out). There are four input to each L_1 which have the following functions. Input D is connected to a L_2 latch, input A is a clock which is used to clock D into L_1, input I is a data line for the use of the designer, and input C is a clock which clocks data from the I into L_1. Input B to L_2 is a clock used to clock the data output from L_1 into L_2. Figure 8 is typical to circuits which use LSSD technique. In this figure, if the output of a combinational circuit is directly connected to a primary output then that output can be used to detect faults in the combinational part, and if the output of the combinational part is input to a latch then this output signal can be run through the chain of latches until it reaches the SDO pin. Thus, using this technique reduces the complexity of testing to that for combinational circuits.

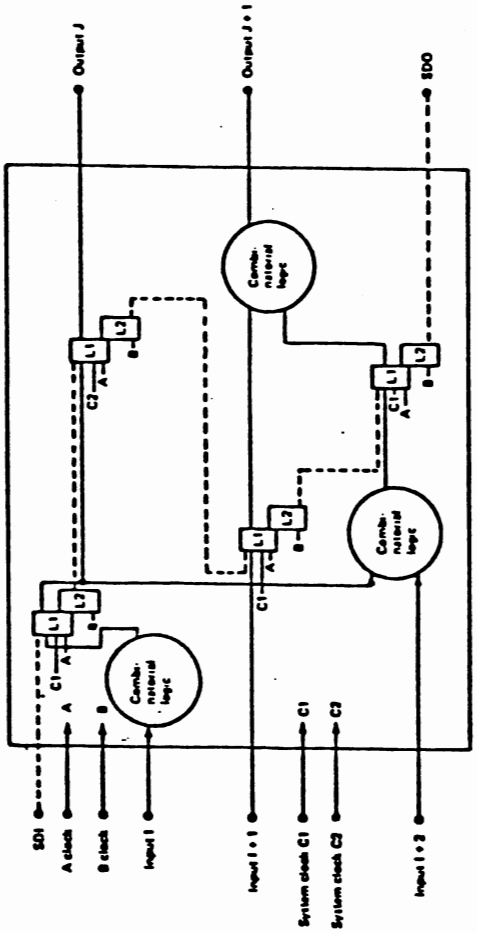


Figure 8. Typical LSSD LSI Chip

CHAPTER III

CLASSIFYING CIRCUIT TOPOLOGIES FOR EASY TESTING

In this chapter different topologies for combinational circuits which make them testable in N^2 time and restrictions on these topologies will be discussed. In chapter IV a simple design method will be presented so that if a combinational circuit is designed according to that method, then it will be testable in N^2 time. There are some terms which will be used throughout this chapter and chapter IV. The following definitions are needed to understand the meaning of each term.

DEFINITION: A propagation value is a value which must be assigned to some inputs of a gate in order that fault(s) on other input(s) of that gate can be propagated through that gate. This value is "0" for OR and NOR gates "1" for AND and NAND gates.

DEFINITION: A path in a circuit from a point to an output is sensitized if all inputs to the gates in that path (other than those on the path) are set to propagation values.

DEFINITION: A point in a circuit is justified for a logic value if inputs of the circuit have values which generate that logic value on that point.

DEFINITION: A circuit is redundant if it contains untestable nodes.

DEFINITION: A circuit is totally irredundant if all subcircuits of that circuit are irredundant. A subcircuit consists of a subset of

gates in the circuit and the inputs to those gates.

DEFINITION: The path sensitization process is the process of sensitizing a single path and finding all the forced values in the circuit because of the assignment of propagation values on the sensitized path.

DEFINITION: The justification process is the process of assigning proper values to the inputs of a circuit in order to justify the values on the outputs of the gates which some values have been assigned to their outputs during the path sensitization process but the inputs to those gates have not been forced to propagation values because of the values on the outputs of those gates.

DEFINITION: A "fanout origin" is a point in a circuit with more than one line exiting from it. Lines which exit from this point are called "fanout branches" of that fanout origin.

DEFINITION: A "reconvergent gate" is a gate that at least two branches from the same fanout origin have a path to that gate.

DEFINITION: A loop is part of a logic circuit which consists of two branches of a fanout origin which reconverge on a gate. This includes the fanout origin, the gates, and the outputs of all gates on the two branches.

DEFINITION: A "simple loop" is a loop which has no fanout origin on outputs of gates on any of its branches and does not share any gate with other loops with different fanout origins or reconvergent gates. Then loops with the same fanout origin and reconvergent gate can share gates. Figure 9 shows a circuit consisting of four simple loops. The four simple loops include the following set of points and the gates between each two points: (E,A,D), (E,A,C), (D,A,C), (G,B,F).

DEFINITION: A point in a circuit is "blocked" for a certain value

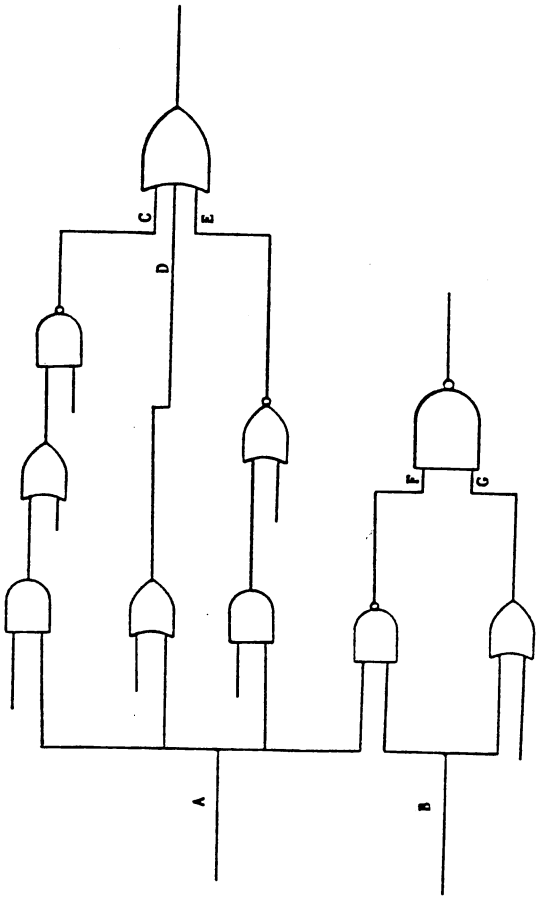


Figure 9. Simple Loops

if that value cannot be propagated through gates to primary outputs.

DEFINITION: A conflict occurs during the path sensitization or justification process if the assignment of a value at some point in the circuit be inconsistent with previous value assignment(s) in the circuit.

DEFINITION: A point which is part of a loop is marked as conflict for a value if assigning that value to that point forces the input to the reconvergent gate on the other branch of the loop to a value which is not a propagation value for that gate.

DEFINITION: A circuit is called "path independent fault detecting" (pifd) if a fault can be detected through any path from the sight of the fault to the primary output(s) without facing any conflict.

LEMMA 1: In any logic circuit if "A is true" implies "B is true" then "complement of B is true" implies "complement of A is true".

PROOF: The proof for this lemma is a direct conclusion from Venn diagrams for logic functions. Figure 10 shows this property.

From Lemma 1 it can be concluded that if assigning a value "a" at some point "A" of a logic circuit forces another point "B" to value "b" then assigning the complement of "b" to "B" forces the value of "A" to complement of "a'.

EXAMPLE: In Figure 11 assigning a logic "1" to "A" forces "B" to logic "0" and assigning logic "1" to "B" forces "A" to logic "0".

THEOREM 1: Consider a circuit with reconvergent fanouts restricted to simple loops and initially all lines have don't care values. If assigning a value on one branch of a fanout origin forces a value at some point of another branch of the same fanout origin then this dependency

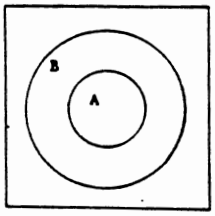


Figure 10. Venn Diagram

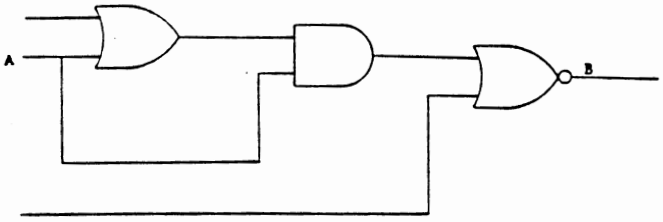


Figure 11. Example for Lemma 1

can be found in time proportional to N where N is the number of lines in a circuit.

PROOF: If a value assignment on one branch of a fanout origin forces a value on the other branch of the same fanout origin, then it must force the fanout origin to some value which in turn forces the point on the other branch to some value. The reason that the fanout origin will be forced to some value is that since initially all lines in the circuit are set to don't care values then the first value assingment on a branch of a fanout origin does not have to satisfy any condition with respect to the other value assignments in the circuit. Then the only way that a point on the other branch can be forced to some value is by forcing the fanout origin to some value first and then propagating the effect of this value on the other branch. Then using Lemma 1, assigning the complement of the value on the fanout origin forces the point in the first branch to the complement of the value it has. According to this conclusion there is a procedure which can find this dependency as follows:

1. Put a "0" on a fanout origin and find forced values on branches of the fanout origin (call them branch one and two).

2. Put a "1" on the fanout origin and find forced values on branches of the fanout origin.

3. Consider the set of points on branch one which were forced to some values in step 1. If the complement of values in step 1 are assigned to any of these points, then by Lemma 1, the fanout origin must be forced to a "1" which forces the points found in step 2 on the second branch to the values found in step 2. The same thing is true for branch 2.

Since this process needs at most 2N value assignments then this process can be done in time proportional to N. This process is called "preprocessing of fanout origins".

EXAMPLE: In Figure 12 assigning a "0" on point "H" forces points "A" and "B" to "0", and points "C" and "D" to "1". Assigning a "1" on point "H" forces points "E" and "F" to "1", and point "G" to "0". Then assigning a "0" on points "C" or "D" or a "1" on points "B" or "A" forces points "E" and "F" to "1" and point "G" to "0".

NOTE 1: Assume only one reconvergent gate exists for branches of some fanout origin. If assigning a value on one branch of a fanout origin which has a path to the reconvergent gate, forces one or more of inputs of the reconvergent gate on other branches to values which are not propagation values, then faults on the original point for the complement of the assigned value cannot be tested through the output of the reconvergent gate. This point is marked as a "conflict" for that value.

NOTE 2: Theorem 1 does not indicate that all forced values in a circuit due to a value assignment can be found in N^2 time because it was assumed that all lines were initially set to don't cares.

NOTE 3: Theorem 1 can be applied to any circuit topology as long as the propagation of a value assignment on one branch of a fanout origin in the forward direction does not force a value on the other branch(es) of the fanout origin. In general the preprocessing of fanout origins can be used to predict some of the sources of backtracking before starting the test generation.

THEOREM 2: The process of test generation, which consists of sensitization and justification of a single path for each fault, for irredundant circuits with reconvergent fanouts restricted to simple loops is

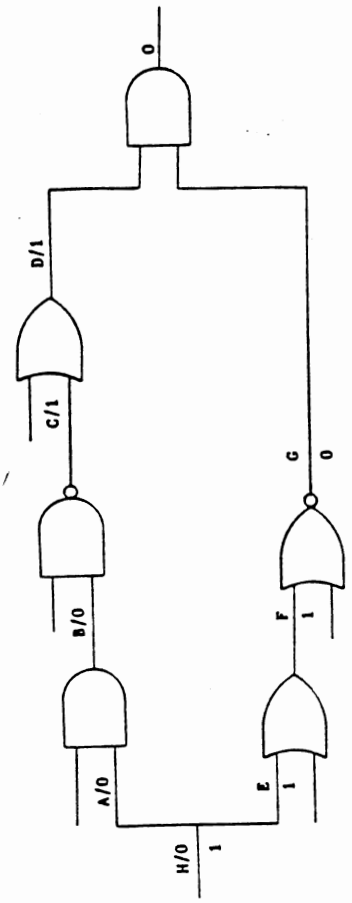


Figure 12. Example for Theorem 1

proportional to N^2 in time where N is the number of inputs to all the gates in a circuit (number of lines). It is assumed that all the lines in the circuit have don't care values prior to test generation and the fault signal on an input of a gate will not be propagated to the output of that gate until the backward effect of value assignments on all inputs of the gate is found throughout the circuit (the two latter conditions will be considered for the other theorems as well).

PROOF: Consider Figure 13 which can be part of a larger combinational circuit. Suppose a test is to be generated for a fault on line A by propagating the value on A to the output through the gates $G_3,..,G_R$,OUT. First, only the value assignments necessary for sensitization of the path will be considered. If value assignment on A forces C to some value C_V then A and C must be on a loop with G_R as reconvergent gate. Notice that there are two ways that value assignment on A can force C to some value without being on a loop with it. The first one is to force the output of G_R to some value which in turn forces C to some value. The second one is to force a fanout origin, in the forward direction, to some value which in turn forces C to some value. If the first case happens then it means that there is either a feedback from the output of a gate G_N, to which G_R has a path, to A or the output of the two gates G_N and G_M are connected together as shown in Figure 14. It is obvious that both connections are in contradiction with the definition of (topology of) combinational circuits. For the second case consider Figure 14. If a value assignment on A forces F' to some value which in turn forces C to a nonpropagation value then at least one of the faults on one of the branches of F is undetectable which is in contradiction with the assumptions made in this theorem. If C_V is not a

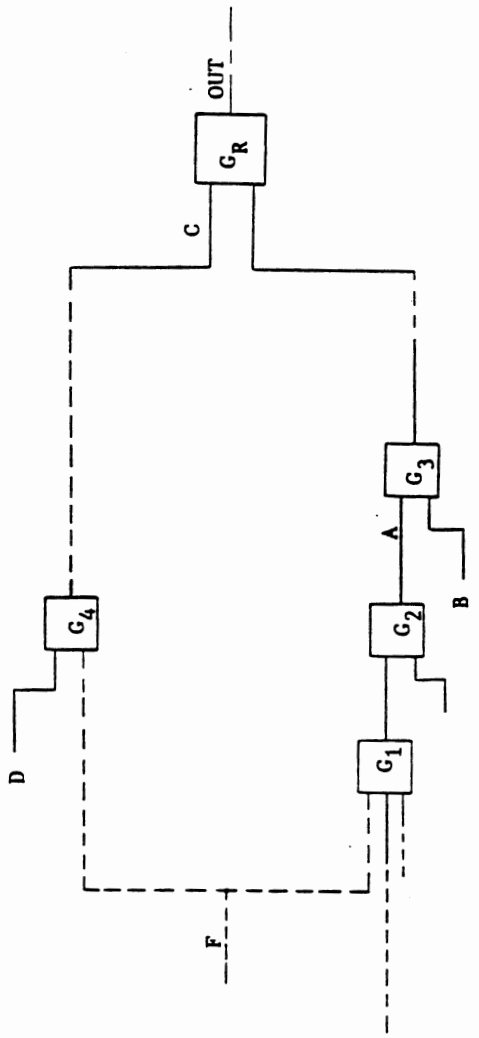


Figure 13. Path Sensitization in Ciruits Consisting of Simple Loops

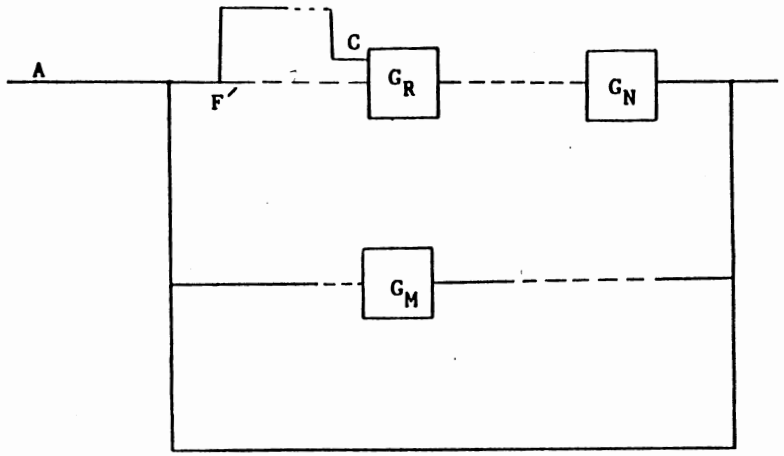


Figure 14. Conflict on a Reconvergent Gate because of Feedback Loops and Fanout Origins

propagation value then the fault on A cannot be propagated through G_R. Since all the loops are simple, then there is no other way for the fault on A to be detected through, which means fault on A cannot be detected which is in contradiction with the assumptions made in this theorem. Notice that inputs like D, which are not part of the loop but are inputs to the gates on the loop, cannot be set to any value becasue if value assignment on A forces D to some value then it means that two inputs to G_4 are on a loop which indicates that two loops with reconvergent gates G_4 and G_R are sharing gates. Then no conflict can occur because of the value assignments on inputs like D. If value assignment on B, or any line which is set to propagation value on the sensitized path, forces C to a non-propagation value then G_R is a reconvergent gate for two different loops, one with (A,C,G_R) and the other one with (B,C,G_R). If A and C are not on a loop but B and C are and C is forced to a nonpropagation value because of the value assignment on B, then at least one fault on B cannot be detected. If a propagation value on a line like B, which is forced to a value in order to satisfy the requirements for path sensitization, needs the requirments which cannot be satisfied, for example if B is the output of an AND gate and has the value "1" but a "1" on one the inputs of this gate forces another input of this gate to a "0", then there is a redundancy in the circuit (line B is stuck at some value) which is in contradiction with the assumptions made in this theorem. Since no value assignment in path sensitization can create conflict then there is no need for backtracking.

Now suppose there are two gates G_1 and G_2 with some values on their outputs but the inputs to G_1 and G_2 are not justified for those values, as shown in Figure 15. Consider one of the inputs to G_1, "A", which is

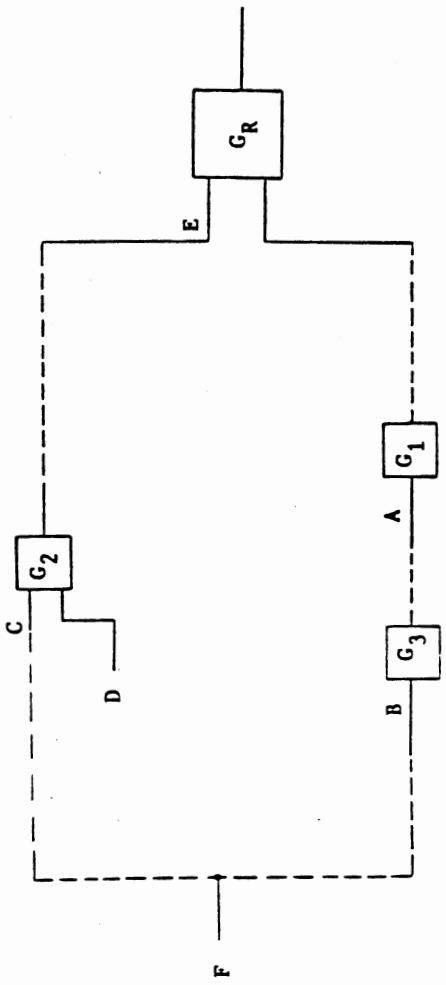


Figure 15. Line Justification in Circuits Consisting of Sample Loops

to take a value in order to partly (part of the condition for justifying the value on the output of G_1) or completely justify the value on the output of G_1. Suppose that because of value assignments from A to a primary input, a value assignment on a point B forces the input C of gate G_2 to some value C_V. Since both G_1 and G_2 have a path to a gate which is (or its output is) part of the sensitized path, and also there is a different path from B to C, then it means that they are on a loop. Since all the loops are simple, again no input such as D can be set to a value by value assignment from A to B or there will be two loops which are sharing gates. If C_V does not have a correct value to justify the value on the output of G_2, then D can be set to that value and no value assignment from A to primary inputs can set D to some other value or loop (E,C,B,A,G_R) is sharing gates with another loop. The reason that only a correct value on one input of G_2 is enough to justify the value on the output of G_2 comes from the fact that the value on the output of G_2 does not force all the inputs to G_2 to propagation values according to the definition of justification process given at the beginning of this chapter. Then a nonpropagation value on one of the inputs of G_2 is enough to justify the value on the output. Then no conflict can occur during the line justification and there is no need for backtracking.

Since both processes of path sensitization and line justification are conflict free then to detect each fault in the circuit not more than N value assignments are necessary and since there can be 2N such faults then the whole process can be done in time proportional to N^2.

EXAMPLE: Consider the circuit in Figure 16. To detect faults on "d", the two other inputs of G_5 must be set to logic "1". To justify a "1"

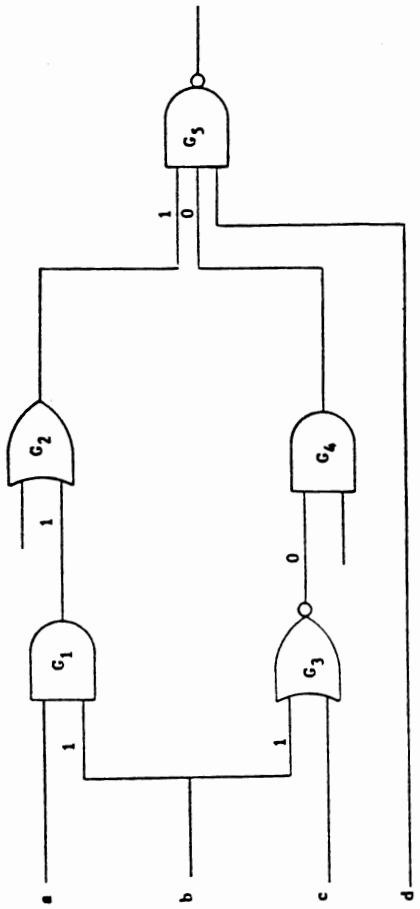


Figure 16. Example of Theorem 2

on the output of G_2, a "1" is required on either of the inputs of G_2. Suppose the output of G_1 is selected in this step. Now to justify a "1" on the output of G_1 requires logic "1" on "a" and "b". A "1" on "b" will force the output of G_3 to logic "0" which in turn will force the output of G_4 to logic "0" which is a conflic for propagating a fault on "d" through G_5. According to theorem 1, such a circuit must be redundant and at least the fault on the input of G_1 which is a branch of "b" cannot be detected through G_5 for the complement value on this line. The value on this input of G_1 is a "1". To detect a s-a-0 on this line a "1" must be assigned to "b". Assigning a "1" to "b" forces the output of G_4 to "0" which violates the propagation rules on G_5. Then the fault s-a-0 on the branch of "b" which is an input to G_1 cannot be detected which makes the circuit redundant.

NOTE 1: In Figure 13 since the fault on "F" can be propagated through both branches, and the type of circuits presented in Theorem 2 consist of only isolated simple loops, then faults on any point of this kind of circuit can be propagated and detected through any path which includes that point. Then this type of circuit is path independent fault detecting.

NOTE 2: There is no limit on the number of fanout branches as long as assumptions made in Theorem 2 hold.

NOTE 3: A circuit with the topology given in Theorem 2 is redundant if a value assignment on a fanout origin forces one or more of the inputs of a reconvergent gate to a value other than the propagation value for that reconvergent gate. This is a direct result from Theorem 2.

Now an upper bound will be found for the number of tests for path independent fault detecting circuits.

LEMMA 2: The number of paths in a combinational circuit with no reconvergent fanout from one of the primary inputs to any of the primary outputs is equal to:

A-B+1

where

A = # of fanout branches on the paths which connect the primary input to primary outputs

B = # of fanout origins on the paths from primary input to primary outputs.

PROOF: A circuit as described above can be considered as a free tree where a free tree is defined to be a finite connected graph with no simple cycle (Standish). The fanout origins and the primary input and outputs are vertices of the tree and lines connecting vertices are edges of the tree.

In a free tree we have the relation e=v-1 where "e" is the number of edges and "v" the number of vertices. In a tree as shown in Figure 17, the number of paths is equal to the number of primary outputs. The reason for this is that any distinct path (two paths are distinct if they are different in at least one edge) originated from the primary input will end with one primary output in a free tree.

If vertices other than primary outputs are called "internal nodes" then we have:

e= # of paths + # of internal nodes - 1

or

# of paths = e - # of internal nodes + 1

but "e" in a combinational circuit is nothing other than the number of fanout branches and internal nodes are fanout origins. Then we have:

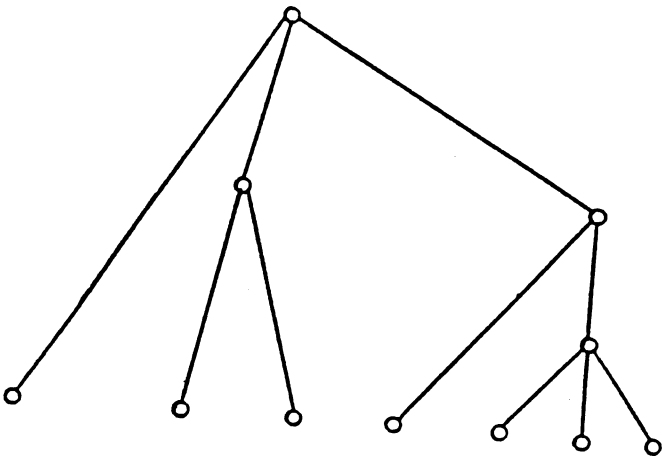


Figure 17. A Free Tree

# of paths = # of fanout branches - # of fanout origins + 1

or

# of paths = A - B + 1

NOTE: In circuits with one primary input and the topology given in Lemma 2, only two tests are necessary to detect all single stuck at faults on a given path. Then, the maximum number of tests in these circuits is as follow:

max # of tests = 2(A - B + 1)

THEOREM 3: The maximum number of tests to detect all single stuck at faults for path independent fault detecting circuits is as follows:

2(# of fanout branches - # of fanout origins + # of primary inputs)

PROOF: Consider Figure 18 which without dotted lines has no reconvergent fanout as required to apply Lemma 2. To detect all single stuck at faults on the dotted sub-path no more than two tests are needed because it is enough to select a path which covers this line from a primary input to a primary output and generate tests for this path. Since one additional fanout branch is added and two more tests are needed then the maximum number of tests for circuit in Figure 3.8 including the dotted line is still :

2(# of fanout branches - # of fanout origins + 1)

Now assume we have a circuit with n primary inputs. For the first input we find all the paths which connect this input to primary outputs. For input i>1, we find all paths which connect that input to primary outputs which have at least one edge that has not appeared in paths found for primary inputs 1 to i-1. There could be edges on the paths originating from input i and merging to one of the paths covered by those for primary inputs 1 to i-1. For each of these edges, which

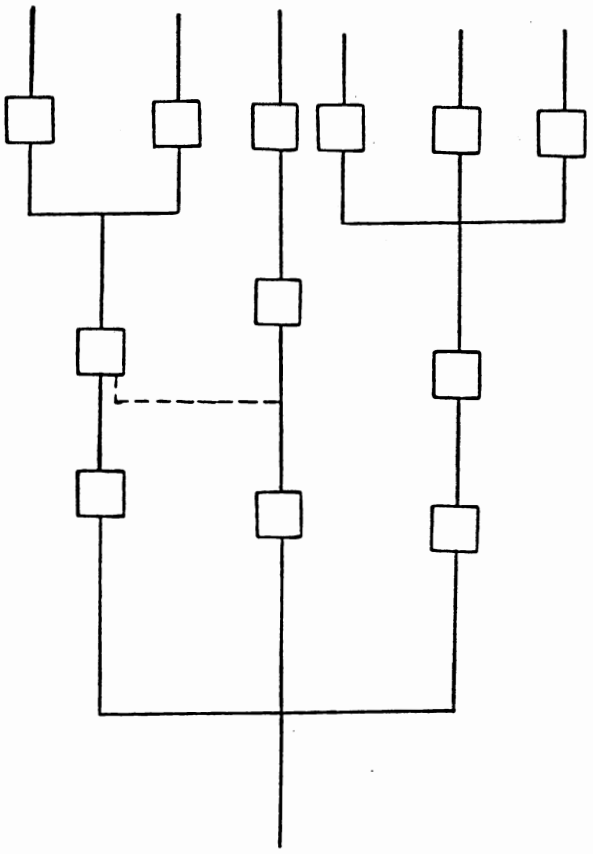


Figure 18. A Logic Circuit with only one input

are fanouts with paths to input i and are not covered by paths found for inputs 1 to i-1, we need two tests (the same thing as for the dotted line). Then there is no line in the circuit which is not covered and no path from primary inputs to primary outputs which has all its edges covered more than one time. Then for the paths found for input i with fanouts which do not merge in to any path covered by another input we have the same relation for maximum number of tests as in Figure 18 and for each fanout merging to another path covered by another input we need at most two tests. Then for the paths found for input i and merging edges on these paths we have:

max # of test(i) = 2(# of fanouts(i) - # of fanout origins(i) + 1)

Adding up "max # of test(i)" for i=1 to n we will have:

max # of tests = 2(# of fanouts - # of fanout origins + # of primary inputs)

Now the process of test generation for redundant circuits with reconvergent fanouts restricted to simple loops will be studied. It should be kept in mind that because of the properties stated in the proof of Theorem 2 for the circuits with the same topology, Theorem 1 can be applied to these circuits.

THEOREM 4: The blocking process (determining the blocked points) for redundant circuits with reconvergent fanouts restricted to simple loops is proportional to N^2 in time.

PROOF: First the preprocessing of fanout oringins presented in Theorem 1 should be applied to all fanout origins of the loops in order to find all the points in a loop which cannot be tested for some values. For each point found this way and marked as conflict for a propagation value consider all the other inputs to the gate that has this point as its

input if those inputs are not affected by the assignment of propagation value on the conflict point. Because this input can be one of the inputs to the reconvergent gate and some of the inputs to the reconvergent gate are affected by this value assignment. Conflicts on the other branch will be found seperately. For each of those inputs travel backward on all the possible paths and mark all those points as "blocked" until a fanout origin is faced. If all branches of a fanout origin are marked as blocked again travel backward and mark those points as "blocked" until another fanout origin or a primary input is faced. Using preprocessing of fanout origins, all the outputs of the reconvergent gates which are stuck at some value can be identified and their effects can be propagated throughout the circuit, and using backward traveling on the inputs of the affected gates the points which are blocked can be identified (if an input of a gate is stuck at a non-propagation value then the other inputs are blocked). If all inputs of a gate are marked as stuck at some value then its output must be marked as "stuck at value", if it is not already marked, and the effect must be propagated throughout the circuit and all blocked points must be found. If all inputs of a gate are marked as "conflict" and "stuck at value" then the input marked as "conflict" (there is only one such input because loops are simple) must be treated as if it were a fanout origin for the loop on which it lies if the output of the gate is not marked as stuck as a value, and preprocessing of fanout origins must be done for that input and all blocked points due to this situation must be identified. Notice that the forced values which are already found do not need to be found again. Since the preprocessing of all fanout origins requires time proportional to N^2 and the rest of the process does not

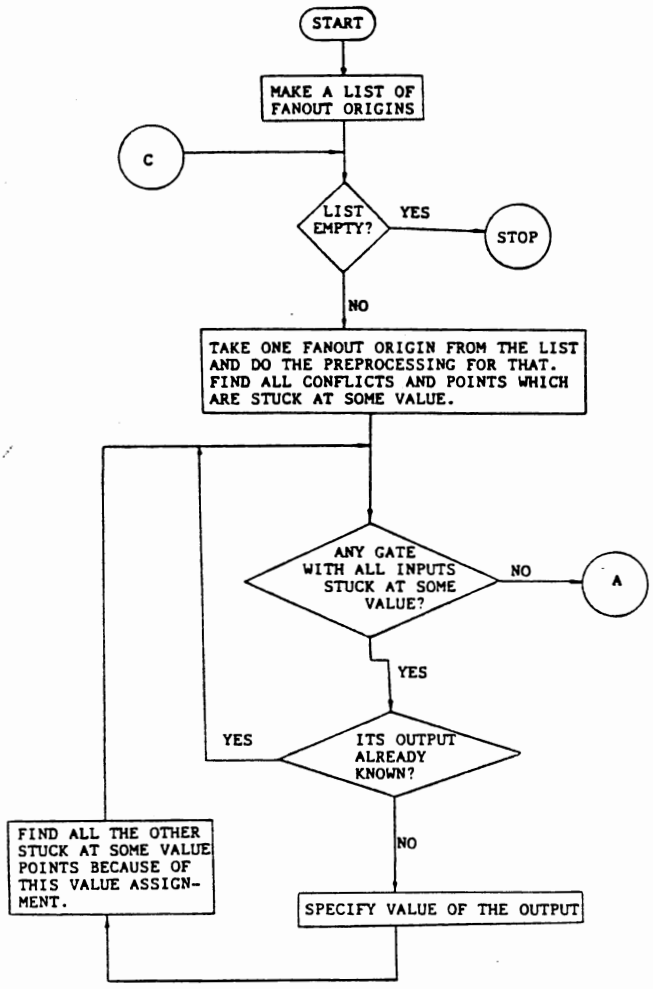


Figure 19. Algorithm for Theorem 4

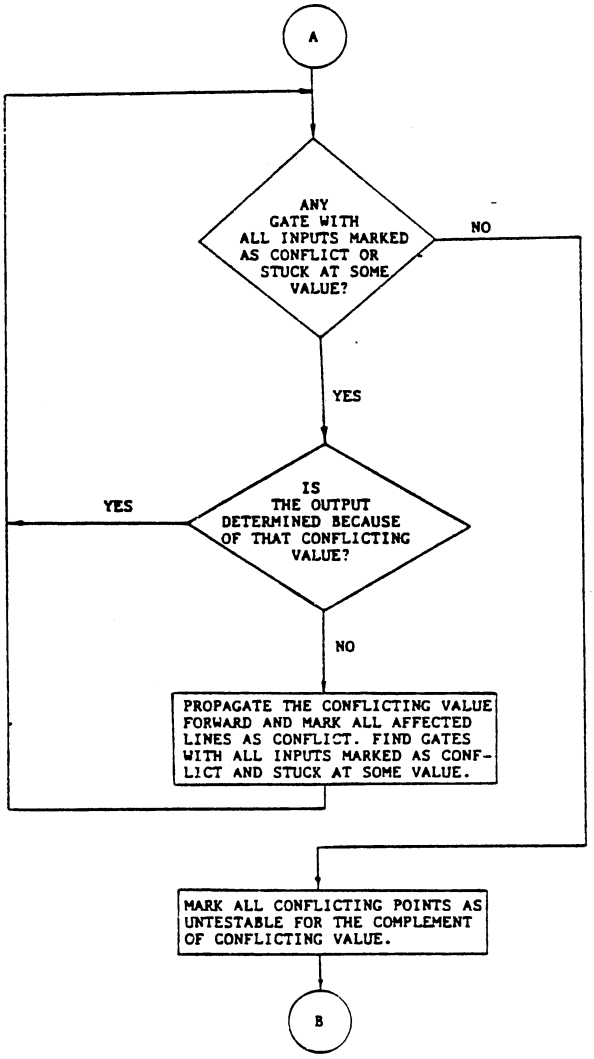


Figure 19. (Continued)

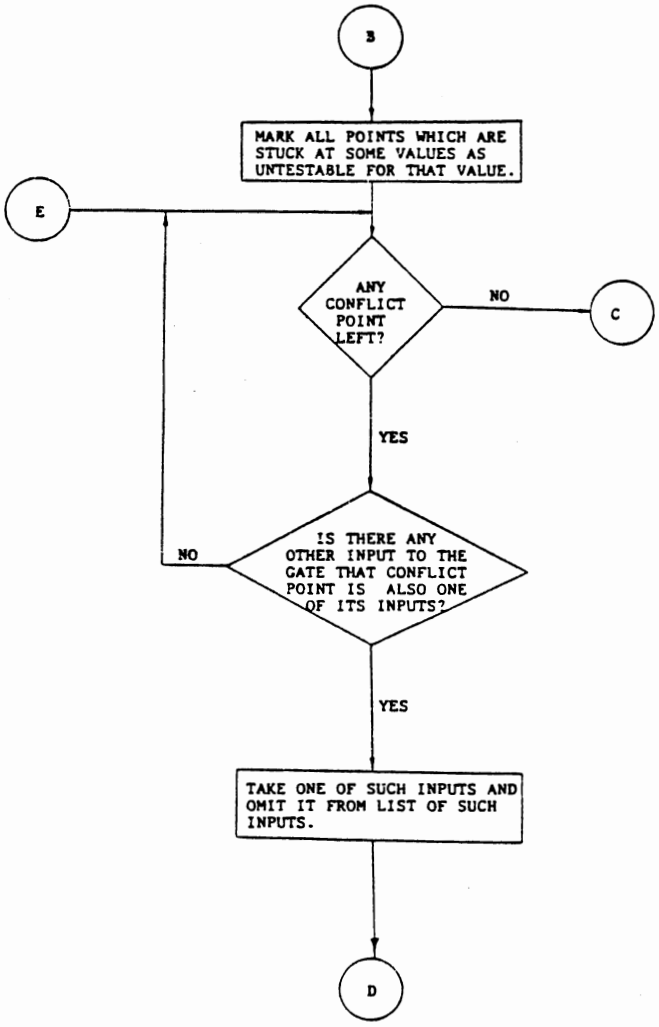


Figure 19. (Continued)

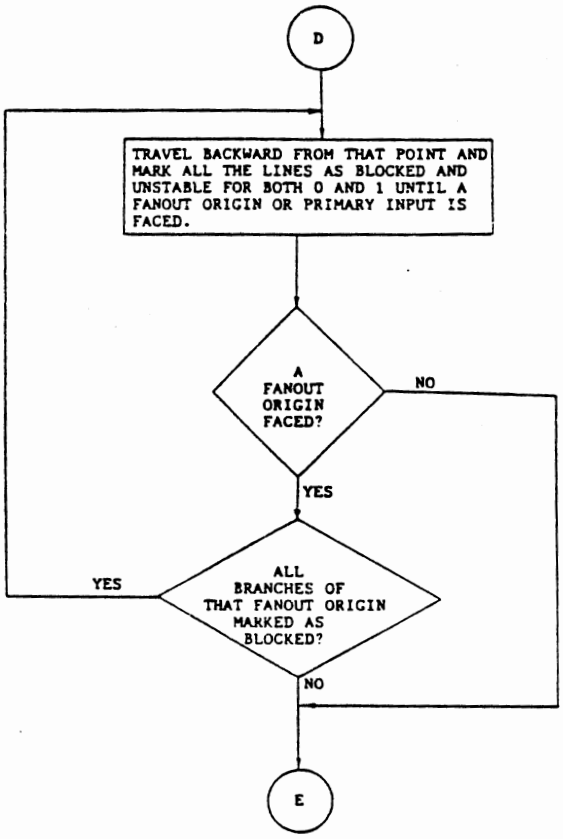


Figure 19. (Continued)

need marking more than N points as "conflict", "stuck at value", or "blocked" then this process is proportional to N^2 in time. An algorithm is given for Theorem 4 in Figure 19.

THEOREM 5: The process of test generation for redundant circits with reconvergent fanouts restricted to simple loops is proportional to N^2 in time.

PROOF: Using "blocking process" presented in Theorem 4, all paths which are blocked for certain faults can be found in time proportional to N. In the justification process there will be no conflict because all the choices for justifying a value on output of a gate which cause conflicts are already marked and will not be chosen. Since to detect each fault no more than N value assignments are required and at most for 2N faults (factor of 2 is for stuck-at-1 and stuck-at-0 faults) tests must be generated seperately, then the time for the whole process is proportional to N^2.

EXAMPLE: Figure 20 shows a redundant circuit with reconvergent fanouts restricted to simple loops. Applying the preprocessing of fanout origins and the blocking process on the circuit in this figure give the results shown in Table I. As can be seen in this figure, all the points which are not testable or must not be chosen in the justification process are marked.

Identifying Reconvergent Gates in Circuits Consisting of Simple Loops:

One of the requirements for making the table in the previous example is to identify reconvergent gates. The following procedure presents a method by which reconvergent gates can be identified in time proportional to N^2:

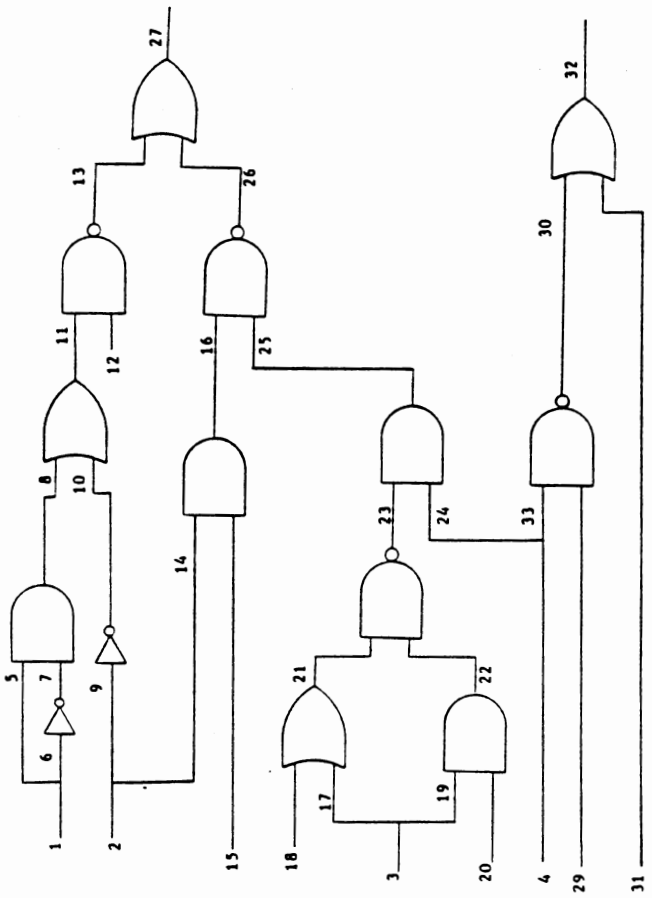


Figure 20. Example of Redundant Circuits Consisting Simple Loops

TABLE I

THE INFORMATION FOUND USING THE ALGORITHM
IN FIGURE 19

LINE #	STUCK AT	BLOCKED FOR	CONFLICT FOR VALUE	CONFLICT ON RECON. GATE	NOT TESTABLE FOR STUCK AT
1	-	-	-	-	-
2	-	-	-	-	-
3	-	1,0	-	-	1,0
4	-	-	-	-	-
5	-	1	1	8	0
6	-	0	0	8	1
7	-	1	1	8	0
8	0	-	-	-	0
9	-	0	0	8	1
10	-	1	1	27	0
11	-	1	1	27	0
12	-	1,0	-	-	1,0
13	-	0	0	27	1
14	-	1	1	27	0
15	-	1,0	-	-	1,0
16	-	1	1	27	0
17	-	1,0	0	23	1,0
18	-	1,0	-	-	1,0
19	-	1,0	-	-	1,0
20	-	1,0	-	-	1,0
21	-	1,0	0	23	1,0
22	-	1,0	-	-	1,0
23	-	1,0	-	-	1,0
24	-	1,0	-	-	1,0
25	-	1,0	-	-	1,0
26	-	0	0	27	1
27	1	0	-	-	1
28	-	-	-	-	-
29	-	-	-	-	-
30	-	-	-	-	-
31	-	-	-	-	-
32	-	-	-	-	-
33	-	-	-	-	-

For all fanout origins do the following:

Travel on all branches of a fanout origin until a primary output, a fanout origin, or a reconvergent gate is faced. Mark all gates in between by the number assigned to the branch which has been traveled. If a gate which is already marked by another branch is faced, mark that as a reconvergent gate of those branches on which this gate lies. Stop the process on that branch and process a new branch.

As can be seen from the above procedure, at most marking N gates is necessary to identify the reconvergent gate and the corresponding fanout branches of a simple fanout origin, and a reconvergent gate will be marked at most N times. Then this process is proportional to 2N for one fanout origin. Since there are no more than N fanout origins then the whole process can be done in time proportional to N^2.

DEFINITION: A simple nested loop is a simple loop with the exception that it can share gates with loops with different reconvergent gates.

An example of circuits consisting of simple nested loops is given in Figure 21. An example of the topology of the loops in such circuits is given in Figure 22.

THEOREM 6: The process of test generation for irredundant circuits with reconvergent fanouts restricted to simple nested loops is proportional to N^2 in time.

PROOF: Consider Figure 23 in which the fault on line A is supposed to be propagated through the path (G_2,..., G_3,..., G_R,..., OUT). If the value assignment on A forces E to some value E_V then as it was shown in the proof of Theorem 2, A and E must be on a loop with G_R as reconvergent

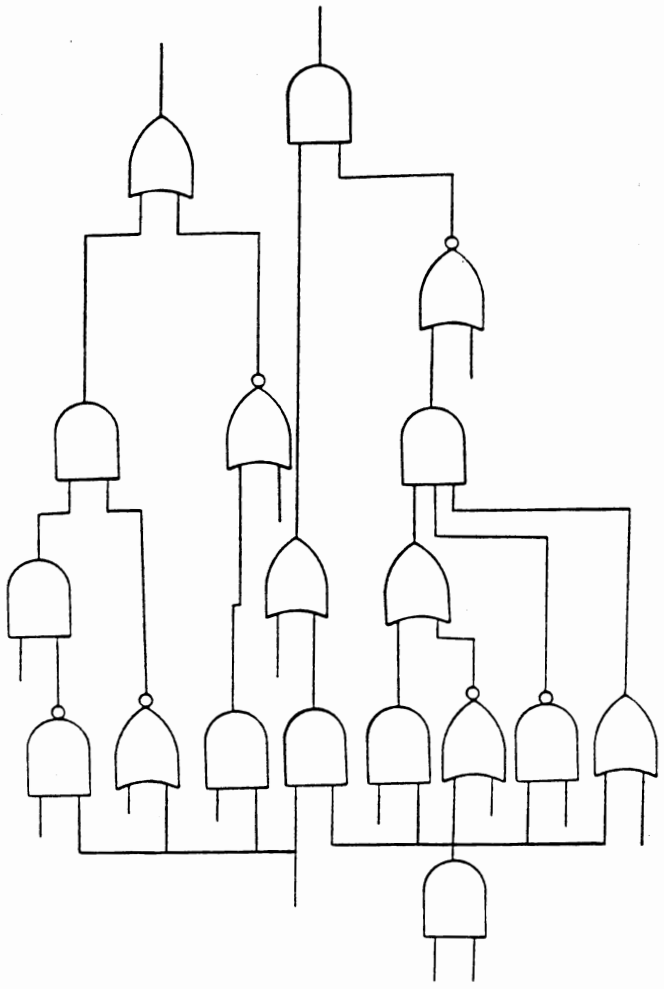


Figure 21. A Circuit Consisting of Simple Nested Loops

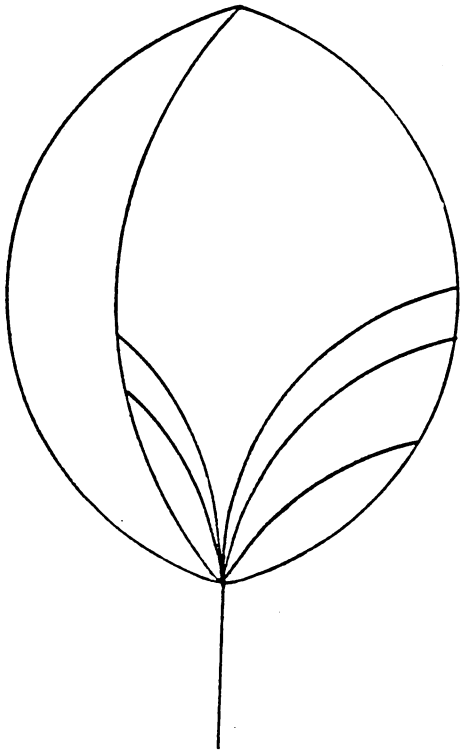


Figure 22. Topology of Simple Nested Loops

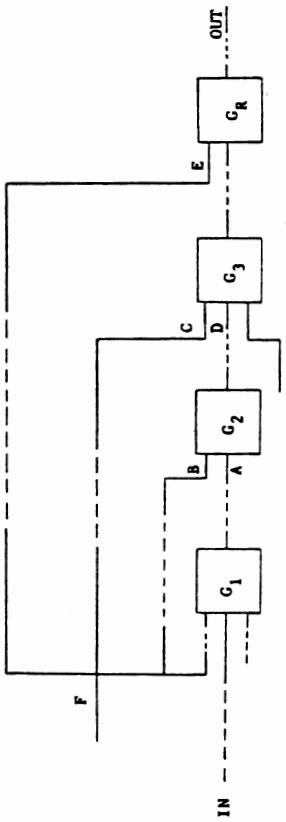


Figure 23. Path Sensitization in Circuits Consisting of Simple Nested Loops

gate. If E_V is not a propagation value then the fault on A will be undetectable because there is no fanout origin between G_2 and G_R which introduces a different path from A to a primary output. It is true because reconvergent fanouts are restricted to simple nested loops. If value assignment on B forces E to a nonpropagation value then at least one fault on B cannot be detected for the same reason stated for A. The same reasoning can be used to show that if value assignment on C or D forces E to a nonpropagation value then there are undetectable faults on C or D. Now suppose that value assignment on E (E_V) is inconsistent with the values on C or D. Then E is on a loop with C or D (or both). The values C_V and D_V on C and D force E to E_V which is a nonpropagation value, then at least one fault on C or D cannot be detected because it cannot be propagated through G_R. The rest of the proof for path sensitization is the same as stated in Theorem 2. Since no value assignment can create a conflict during the path sensitization then there will be no need for backtracking.

For the proof during the justification process, consider Figure 15. Since there are simple nested loops in the circuit, it is possible because of a value assignment on B that both lines C and D be forced to propagation values which are inconsistent with the value on the output of G_2. But if that happens then there is at least one fault on B which cannot be propagated through G_R (notice that the only way that a value assingment on B and the other lines from B to the primary inputs can force C and D to some value is through F) and since due to the topology of the simple nested loops there is no other path for the fault on B to be detected through, this fault is undetectable which means the circuit is redundant, which is in contradiction with the assumptions made in

this theorem. Then there is no need for backtracking in the line justification.

Since there is no need for backtracking in path sensitization and line justification then at most N value assignments in the circuit are necessary to detect a fault. At most there are 2N such faults, then there is no need for more than 2N value assignment in the circuit which means the required time for test generation is proportional to N^2. NOTE: Since the faults on a fanout origin can be propagated and detected through any branch of that fanout origin then circuits of this kind are path independent fault detecting.

The next topology of loops which will be considered is "simple totally nested loops."

DEFINITION: Simple totally nested loops are loops with the following characteristics:

1. They can have fanout origins on their branches providing that the branches of these fanout origins must reconverge on the gates which have paths to the reconvergent gate of the loop from which they are originated.

2. No two loops may share gates if in forward traveling of paths in the circuit there is no path between their fanout origins.

3. No two branches of a fanout origin may reconverge on more than one gate.

An example of the above topology is given in Figure 24 and an example of the circuit consisting of simple totallynested loops is given in Figure 25.

The conditions in the definition of simple totally nested loops eliminate the possibility that if a point on a loop cannot be tested

through the reconvergent gate of that loop then it may be tested through another path. Figure 26 shows an example of what may happen if condition 1 is eliminated. In this figure the fault a-s-0 cannot be propagated through "c" but it can be detected through "b" while the circuit is irredundant. Elimination of condition 2 makes it possible for a gate which is on different loops to be affected by value assignments on fanout origins of those loops. Although all the loops are irredundant, those value assignments may cause a conflict to occur on a reconvergent gate and the test generation process may not be conflict free. An example of such a situation is given in Figure 27. Suppose "a" must be justified for value "1" and arbitrary choices have assigned a "1" on "b", "g", and "c". Then "d" and "e" will be forced to "0" and "1" respectively for a "1" on "h" and "i". These value assignments put a "0" on "f" which is a conflict. Note that no value assignment on a single fanout origin causes a conflict on a reconvergent gate, but to justify "g" for a "1" a certain combination of value assignments on fanout origins are required although the whole circuit is totally irredundant. Condition three guarantees that no two branches of a fanout origin can reconverge on more than one gate because if two branches of a fanout origin reconverge on more than one gate then it is possible that not all paths in a circuit can be sensitized even in totally irredundant circuits as will be discussed later where the definition of simple totally nested loops will be modified for totally irredundant circuits with more complex topology.

THEOREM 7: The process of test generation for irredundant circuits consisting of simple totally nested loops is proportional to N^2 in time.

PROOF: Consider Figure 28 and assume that a test is to be generated for

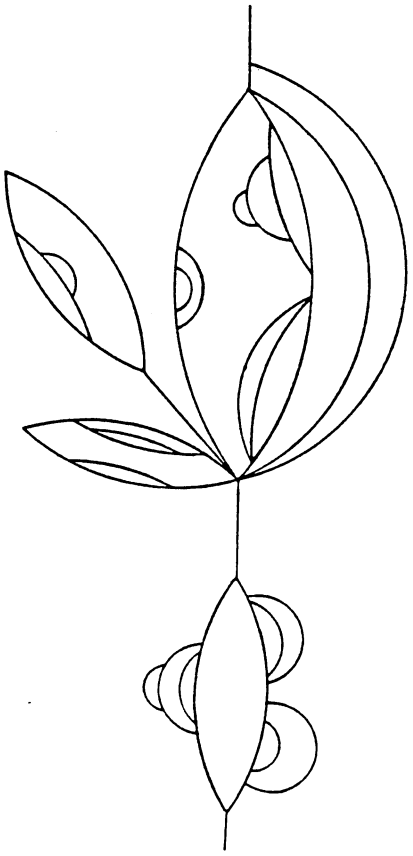


Figure 24. Topology of Simple Totally nested Loops

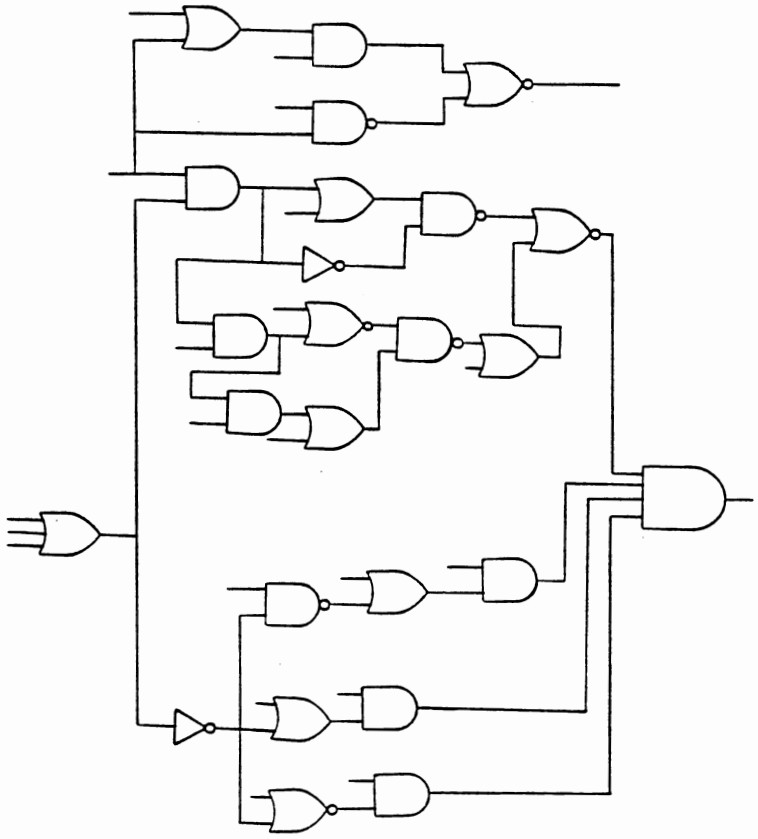


Figure 25. An Example of Circuits Consisting of Simple Totally nested Loops

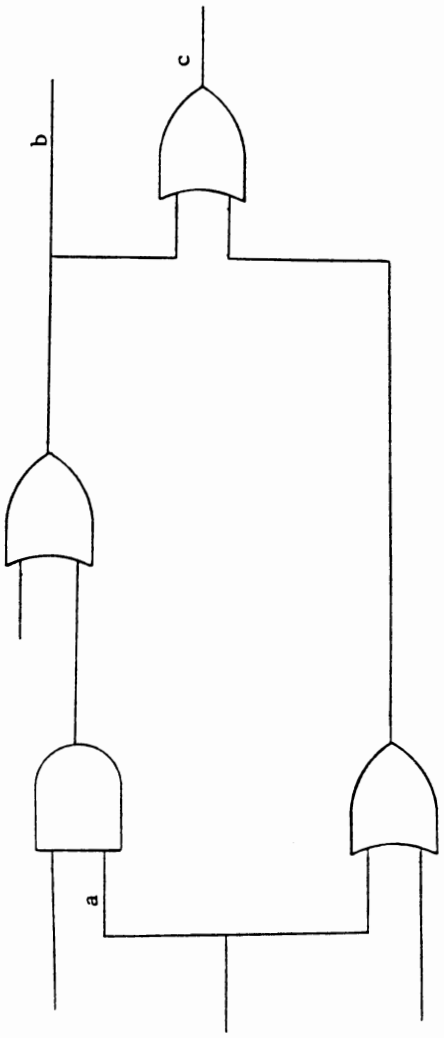


Figure 26. An Irredundant Circuit Including a Redundant Loop

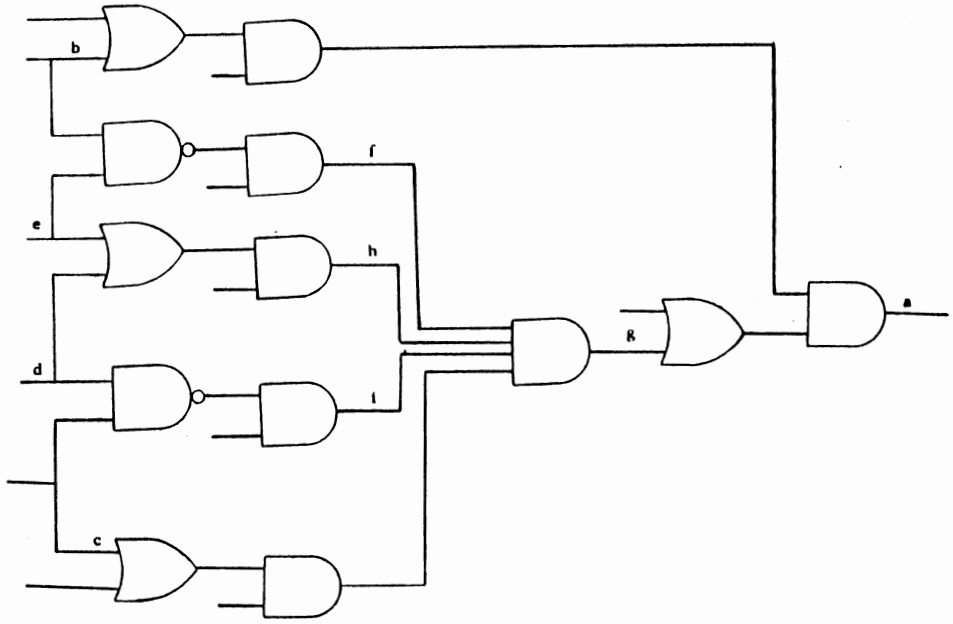


Figure 27. Conflict in Test Generation for the Loops with Unconnected Fanout Origins

the fault on line A. Suppose that the fault signal is to be propagated through G_R. If value assignment on A forces C to a nonpropagation value, then A and C are on a loop as was stated in the proof of Theorem 2. Although there can be fanout origins between A and G_R, according to the definition of simple totally nested loops the fault on A must be propagated through G_R. Then the fault on A will be undetectable which is in contradiction with the assumptions made in this theorem. If value assignment on B forces C to a nonpropagation value then B and C are on a loop and at least one fault on B is undetectable because the fault has to be propagated through G_R. If value assignment on a line between G_1 and G_R forces C to nonpropagation value, such as D or E, since it has to be on a loop with C then at least one fault on that line remains undetectable. This effect is independent of other value assignments during the path sensitization. For example, if value assignment on B forces one input to G_2 to a propagation value and value assignment on D forces the other input of G_2 to a propagation value which forces the output of G_2 to a value which in turn forces C to a nonpropagation value, then it means that there are two loops, (B, G_2, G_R) and (D, G_2, G_R), with unconnected fanout origins which share gates which is in contradiction with the assumptions made in this theorem. Then there is no conflict during the path sensitization. The proof for line justification is similar to the one for Theorem 6. Since there is no backtracking in path sensitization and line justification then only N value assignment is necessary to generate a test for a given fault which makes the time complexity of the test generation proportional to N^2.

NOTE: Irredundant circuits consisting of any combination of topologies discussed so far can be tested in time proportional to N^2 because they

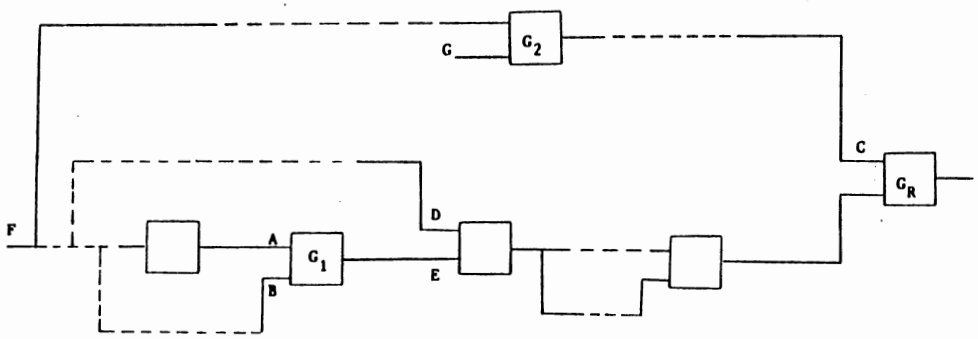


Figure 28. Path Sensitization in Circuits Consisting of Simple Totally nested Loops

all share the property that if a value assignment on one branch of a loop causes a conflict on the reconvergent gate of that loop then that point is not testable for the complement of the value it has.

If the circuits with the topologies discussed so far are irredundant then it means that each loop in the circuit is irredundant. But a circuit can be irredundant with some redundant loops as was shown in Figure 26. Now one of the conditions can be omitted from the definition of simple totallynested loops and still circuit with the topology in the modified definition can be testable in time proportional to N^2 if all loop are irredundant. The condition which can be omitted is condition 1 which expands the topology under consideration to circuits of which one example is given in Figure 29. An example of the topology of circuits with the above definition is given in Figure 30. Notice that condition three in the modified definition is necessary because there are circuits which are totally irredundant but not path-independent fault-detecting since two branches of a fanout origin reconverge on more than one gate. An example of this kind of circuits is given in Figure 31. In this figure the fault a-s-0 cannot be detected through the path (G_1, G_3, G_5) although the circuit is totally irredundant. The class of circuits recognized by the modified definition of simple totally nested loops is called "SIMPLE CONNECTED LOOPS".

THEOREM 8: The process of test generation for the circuits consisting of simple connected loops in which all loops are irredundant is proportional to N^2 in time.

PROOF: Consider Figure 32. Suppose that the fault on A is to be propagated through G_1, G_2, G_3, and G_R. Suppose that G is forced to a nonpropagation value at some point during the path sensitization because

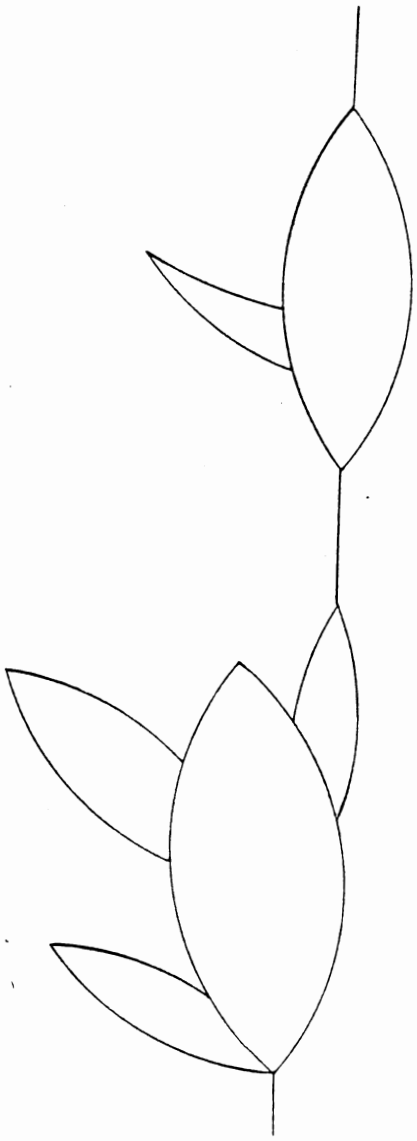


Figure 29. Topology of Simple Connected Loops

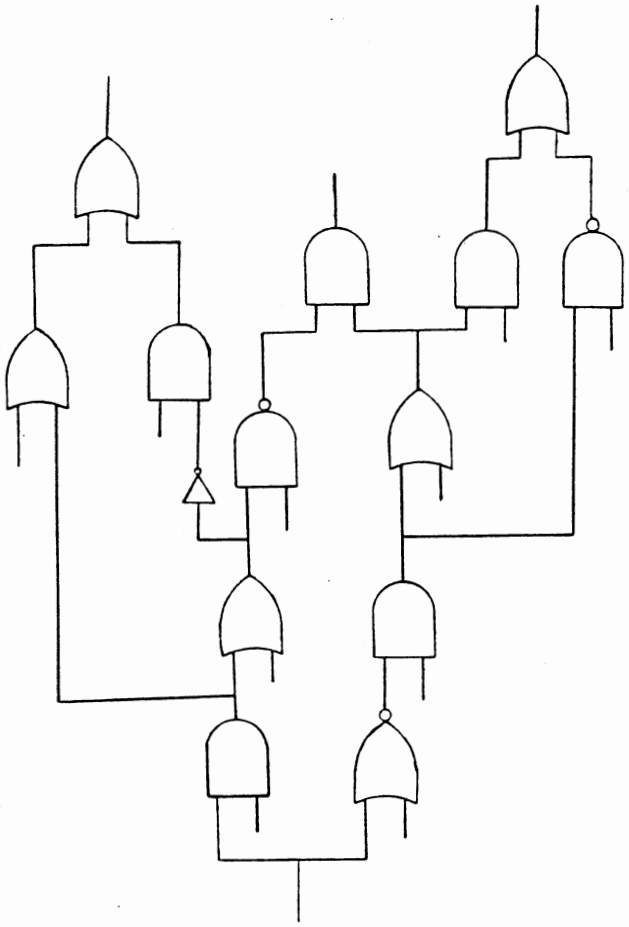


Figure 30. An example of Circuits Consisting of Simple Connected Loops

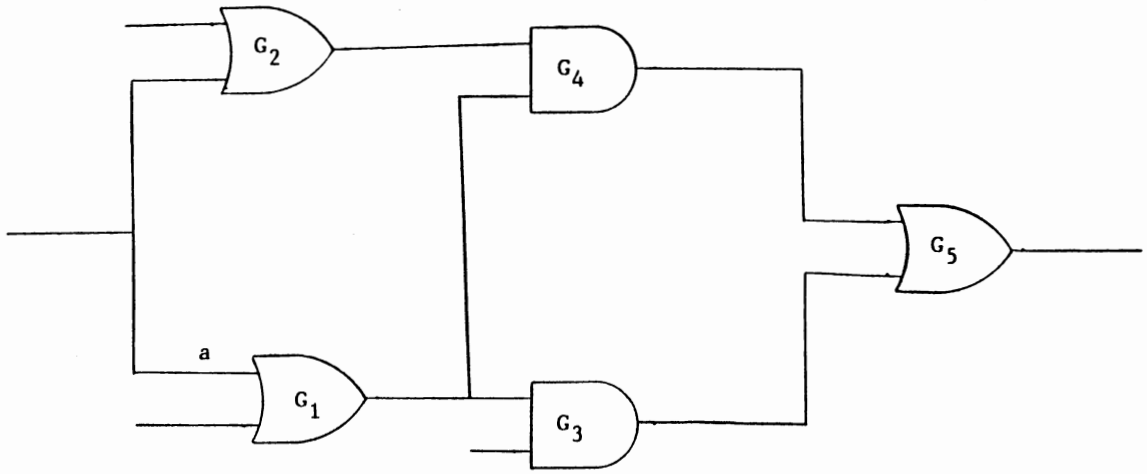


Figure 31. Conflict if two Branches of a Fanout Origin Reconverge on More than one Gate

of value assignment on some lines such as A,B,..., and E. Then either value assignment on one of these lines, for example C, has forced G to the nonpropagation value independent of the other value assignments in the circuit or value assignments on several or all of lines (A,B,..,F) have forced G to that value. In the first case if G and C are on a loop, then at least one fault is not detectable through G_R which is in contradiction with the assumption that all loops are irredundant. If C and G are not on a loop then C has to force a fanout origin to some value which in turn the value assignment on this fanout origin forces G to a nonpropagation value. In this case at least one of the faults on one of the branches of this fanout origin cannot be detected through G_R which means that there is a redundant loop in the circuit. If value assignments on several points forces G to a nonpropagation value and those points are on some loops with G, then as it can be seen from Figure 32, two branches of a fanout origin reconverge on more than one gate, G_4 and G_R, and loops which their fanout origins have no path to each other are sharing gates, which is in contradiction with the assumptions made in this theorem. If all of those value assignments forces only one fanout origin to some value which in turn forces G to a nonpropagation value, then at least one fault on one of the branches of the fanout origin cannot be detected through G_R which means there is at least one redundant loop in the circuit. If A, B,.., and F are not on a loop with G, then either they have to force one fanout origin to some value which in turn creates a conflict on G or they force several fanout origins to some value which in turn force the line G to a nonpropagation value. In the first case there is a redundant loop in the circuit and in the second case one of the rules for simple connected loops has been

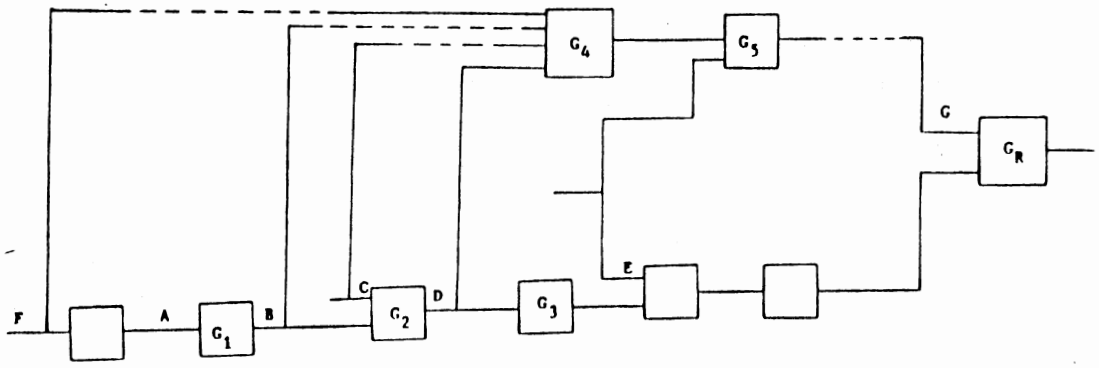


Figure 32. Path Sensitization in Circuits Consisting of Simple Connected Loops

violated. Then there is no need for backtracking in path sensitization.

Now it will be shown that the line justification process is also conflict free. Consider Figure 33 and suppose that the lines A and B are to be justified for the values that they have. Also assume that the lines A and B are outputs of the gates G_A and G_B. Then only assigning nonpropagation values on one of the inputs of G_A and G_B is enough to justify the values on A and B. Suppose that because of the value assignment on a point C, for justifying the value on A, all the inputs to G_B which have don't cares be changed to propagation values. If that happens then it means that A and B are on a loop because both can be merged to the sensitized path through some paths. If the value on C forces the inputs of G_B to some value then it must first forces a fanout origin(s) to a value which in turn forces the inputs of G_B to some value (or other fanout origin(s) which forces the inputs of G_B to some value). If more than one fanout origins, F_1 and F_2, are forced to some values then as it can be seen from the Figure 33 the two loops (A,F_1,B,G_R) and (A,F_2,B,G_R) which have no path between their fanout origins are sharing gates which is in contradiction with the assumptions made in this theorem. Suppose that value assignment on F_1 and F_3 have forced all the inputs of G_B which have don't cares to propagation values. If that happens then consider the other input of G_B, D, which has been assigned a value during the path sensitization or line justification to justify a value on a line E. Then D and E must be on a loop, as it is shown in Figure 33, and the loops (B,D,E,G_Q) and (A,F_1,B,G_R) which have no path between their fanout origins are sharing gates which is in contradiction with the assumptions made in this theorem. Now suppose that during the line justification for point A, a

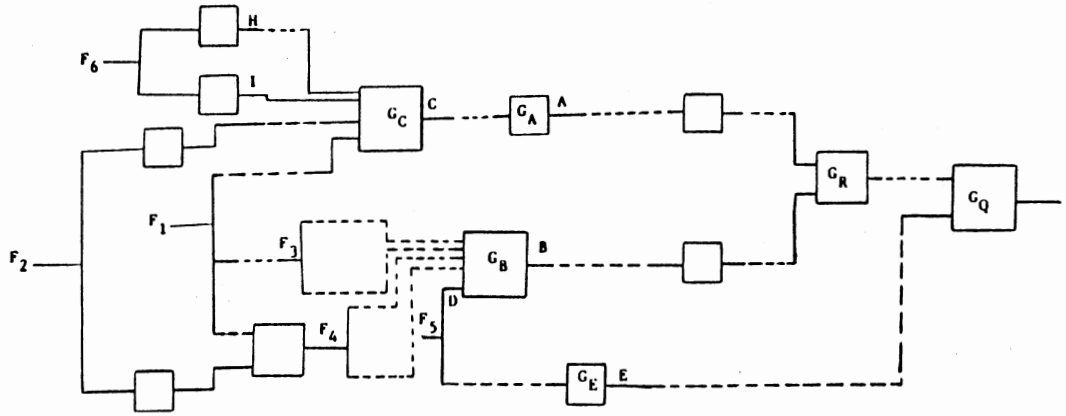


Figure 33. Line Justification in Circuits Consisting of Simple Connected Loops

value assignment on C forces all the inputs to G_C to propagation values but because of the value assignment on H, the input I to G_C be forced to a nonpropagation value. If that happens then at least one fault on the branch of F_6 which has a path to H is undetectable through G_C which means a loop in the circuit is redundant which is in contradiction with the assumptions made in this theorem. Then there is no conflict during the line justification. Since only N value assignments are necessary to generate a test for a given fault and there are no more than 2N such faults in the circuit, then the required time for test generation is proportional to N^2.

CHAPTER IV

DESIGN FOR TESTABILITY

Now attention will be focused on circuits such that two branches of a fanout origin may reconverge on more than one reconvergent gate, and loops whose fanout origins have no path to each other may share gates. Different comments can be made, as design aids, on the topology of these kinds of circuits such that all paths can be sensitized and no conflict be faced in the justification process. For example "loops with unconnected fanout origins must not reconverge on gates which have paths to each other". But none of these comments seems to be easy to apply when designing a circuit and will put restrictions on the topology of a circuit and may not be always applicable. Instead a design method will be introduced which makes any circuit testable in time proportional to N^2.

It is obvious that there cannot be any inconsistency in value assignments in the path sensitization and justification process for the circuits with no reconvergent fanouts providing that any value assignment in the circuit is for the prupose of sensitizing a path or justifying a line. By adding reconvergent fanouts to the circuit, there could be inconsistency in value assignments when generating tests for the circuit. Since this inconsistency in value assignments is only because of the existence of the reconvergent fanouts in the circuit, then any conflict in value assignments can be transfered to a conflict on a

reconvergent gate. And that reconvergent gate is either part of a sensitized path or is to be justified for some value on its output. Then if the value assignments on the inputs of the reconvergent gates in the circuit can be controlled, any inconsistency in value assignments can be avoided. If a reconvergent gate is part of a sensitized path then a value which is not a propagation value for that gate must not reach the gate. If this gate is to be justified for some value on its output which forces all its inputs to propagation values then, like the previous case, no nonpropagation values must reach the gate. If the inputs of a reconvergent gate must be justified for values which are not propagation values then not all the inputs of the reconvergent gate must be forced to propagation values.

To see how the situations mentioned above can be avoided consider the loop in Figure 34 and add two gates after G_{N1} and G_{M2} according to the following rules:

1. If G_R is an OR or NOR gate then the two gates must be AND gates. If G_R is an AND or NAND gate then the two gates must be OR gates. Call these gates "BLOCKING GATES".

2. Each blocking gate has two inputs. One is the output of G_{N1} or G_{M2} and the other input is called the "CONTROL" or "TEST" input. This input can be treated as a primary input.

By adding the blocking gates to the circuit, no inconsistency in value assignments can occur during sensitizing a path because the control inputs can be set to the values needed on the inputs of the reconvergent gates. The same thing is true for the case that the output of a reconvergent gate must be justified for a value which needs assignments of propagation values on all the inputs of that gate. Now

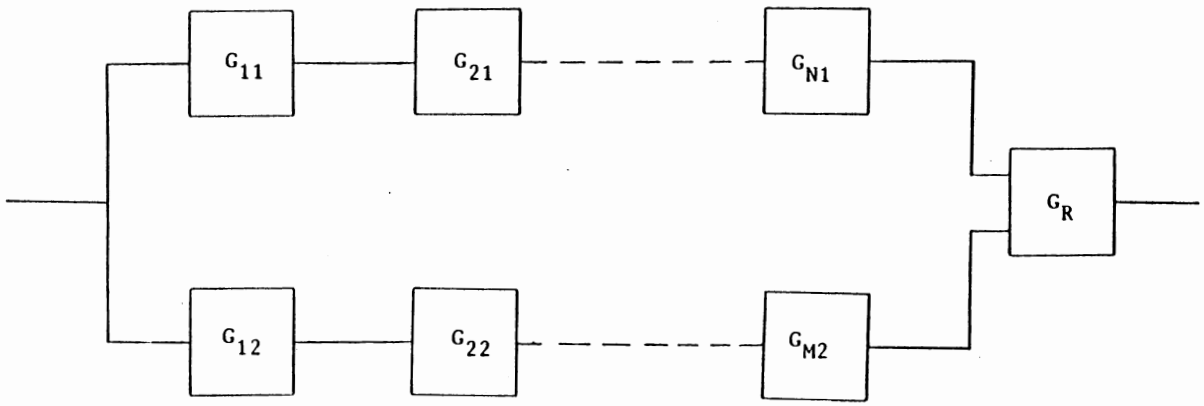


Figure 34. Example of Two Reconvergent Paths

consider a reconvergent gate which has a value on its output that needs at least a nonpropagation value assignment on one of its inputs. Also assume that all the inputs to this gate are set to propagation values because of a value assignment on a point "P" (the last value assignment which set all the inputs of the reconvergent gate with don't care values to propagation values is important, otherwise there are still choices available on the inputs of the reconvergent gate). Since the value on the output of this gate has been determined independent of values on its inputs, then it means that the reconvergent gate itself is on another loop with "P" (if it is not true then either the value assignment on the output of the reconvergent gate or the value assignment on "P" is arbitrary and not forced by the requirments for the path sensitization or justification process). Then this inconsistency or conflict could be transfered to the reconvergent gate of this new loop where it could have been avoided by controlling a test input. This suggests that inproper use of test inputs could cause problems. Notice that adding the blocking gates and the test inputs to a circuit guarantee that no conflict may arise in path sensitization for a certain fault because nonpropagation values can not reach reconvergent gates. But propagation values may reach reconvergent gates and cause conflicts if test inputs are not used properly. An example of such a situation is given in Figure 35. Suppose "a" is to be tested for s-a-0, then a series of value assignments on b, c, and d (all of them have value "1") forces "a" to "0" which is a conflict. This situation can be taken care of and tests can be generated in one of the three following ways:

1. Whenever there is a choice between a test input and the other input of a blocking gate, take the test input. This gives freedom to

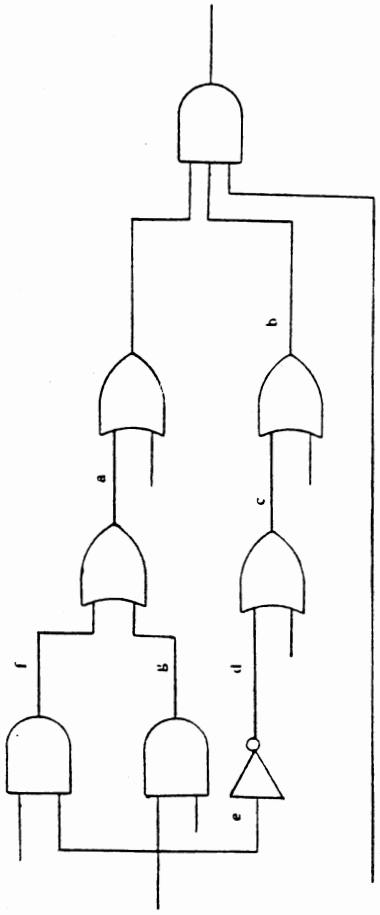


Figure 35. Conflict because of inproper use of Test Inputs

the other input of a blocking gate to be set to any logic value (prevents the inproper use of the test inputs).

2. Completely justify a given point for a value until primary inputs are faced before continuing path sensitization or justifying any other line. Since no value assignment on fanout origins is able to force the inputs of the reconvergent gates in the sensitized path to nonpropagation values, then no conflict occurs and test inputs can be set to appropriate values for the path sensitization or justification process. This solution has the advantage that there will be no need to make special use of test inputs when choices occur.

3. Start test generation for the circuit by sensitizing paths beginning at primary inputs and cover all paths in the circuit. The justification process must be finished entirely for a line before justification of another line is started. This solution has the advantage that longer paths will be covered, the number of tests will be reduced, and there will be no need to make special use of (to keep track of) test inputs when choices occur.

Since there is no conflict in the path sensitization and line justification process, then no more than N value assignments are necessary to generate a test for a given fault. Since there are no more than 2N stuck at 0/1 faults in the circuit then the time complexity of the test generation in the worst case will be proportional to N^2.

In general test inputs can be treated as primary inputs to the circuit, but for chips with built-in test facilities they do not have to appear on the external input pins. This issue will be discussed later. The value of a test input is a propagation value for the normal operation of a circuit.

Note that the number of test inputs and blocking gates cannot be more than N, and consequently, the number of lines in the circuit can not exceed 2N. The only thing needed to identify places where blocking gates and test inputs must be placed is identifying reconvergent gates and inputs to that gate which are part of a loop.

The process of identifying reconvergent gates is proportional to N^2 in time according to the following procedure:

> Take one fanout origin and travel on all paths from that fanout origin to primary outputs and mark all the gates and gates' inputs which are traveled. If in this process a gate which has already been marked is found, mark it as a reconvergent gate. Also mark the inputs to this gate which are on a loop. Repeat this for all fanout origins.

Since there are no more than N fanout origins and the above process for each of them does not need marking more than N gates, then the whole process can be done in time proportional to N^2. The example in Figure 31 is redrawn in Figure 36 with the exception that the blocking gates and the test inputs are added. When it is worthwhile to have built-in test facilities a shift register can be used to load desired values for test inputs when the circuit is under test. In normal operation test inputs have propagation values. Figure 37 demonstrates this scheme. For faster testing, the scheme shown in Figure 38 can be used. The ROM in this figure can be used to save the whole test pattern or only the values of the test inputs for each test. Notice that the latter scheme is faster because all the test inputs can be set to desired values at the same time. In Figure 37 and 38, only one input is added to the pins of the chip.

It is obvious that a designer of a circuit prefers not to add blocking gates and test inputs as much as possible. One way to decrease

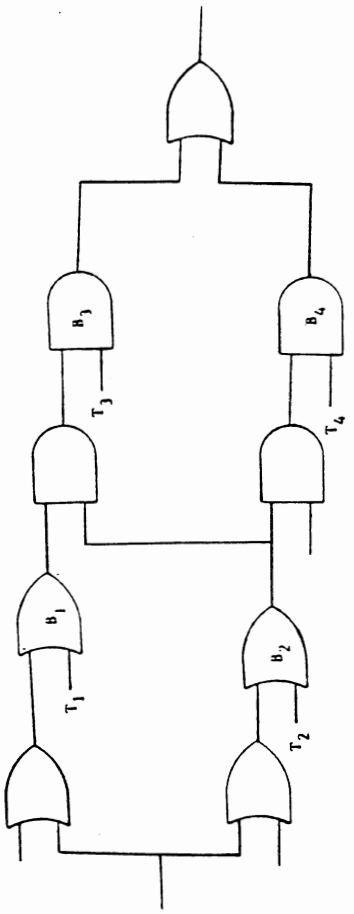


Figure 36. A Circuit with Added Blocking Gates and Test Inputs

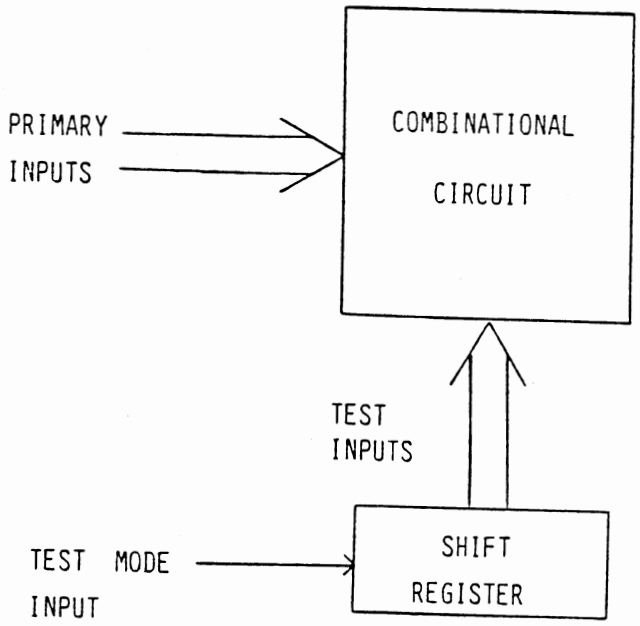


Figure 37. Use of Shift Registers for Test Generation

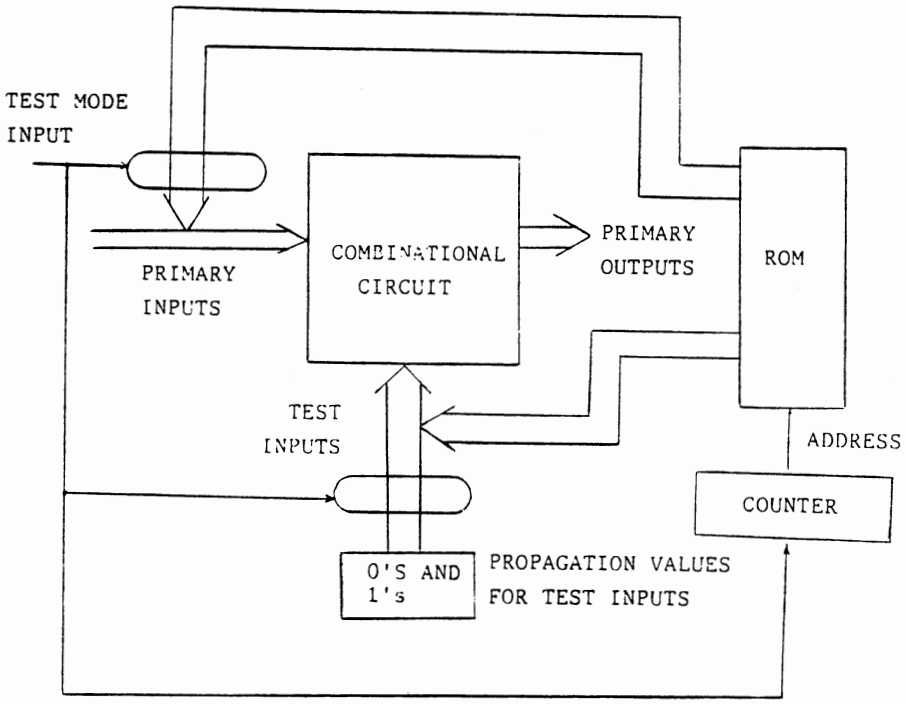


Figure 38. Saving the Test Vectors inside the Chip

the number of blocking gates is to identify the type of the gate placed in a loop immediately before the reconvergent gate. If this gate has the type which matches the required gate type for the blocking gate, then only one extra input need be added to that gate as a test input and there is no need to add an extra blocking gate. An example of this situation is given in Figure 39. In this figure no blocking gates need to be added after gates 1,2,3, and 4. Only on extra input to each gate is enough. There are other alternatives for blocking gates that some of them are shown in Figure 40.

There can be even a more drastic improvement to the design if one of the gates identified in the previous phragraph has an input which is not a part of any loop. Then this input can be considered as a test input and there will be no need to add any extra input to the circuit. An example of such situation is given in Figure 41. In this figure lines "A", "B", "C", and "D" can be considered as test inputs because none of them are on any loop and they can be set to appropriate values to control the values on the inputs of reconvergent gates. These inputs are called "FREE INPUTS".

One thing which can be done to halve the number of attempts to generate tests, and eventually the time required for the test generation, is to set all the control inputs which are not on the sensitized path to their nonpropagation values whenever a blocking gate is faced during the path sensitization process. The reason for this is that control inputs can be either set to nonpropagation values or don't cares and the value assignment on the sensitized path has no effect neither on the set of gates which should be considered for justification process nor on the values on the output of these gates. Then if the program

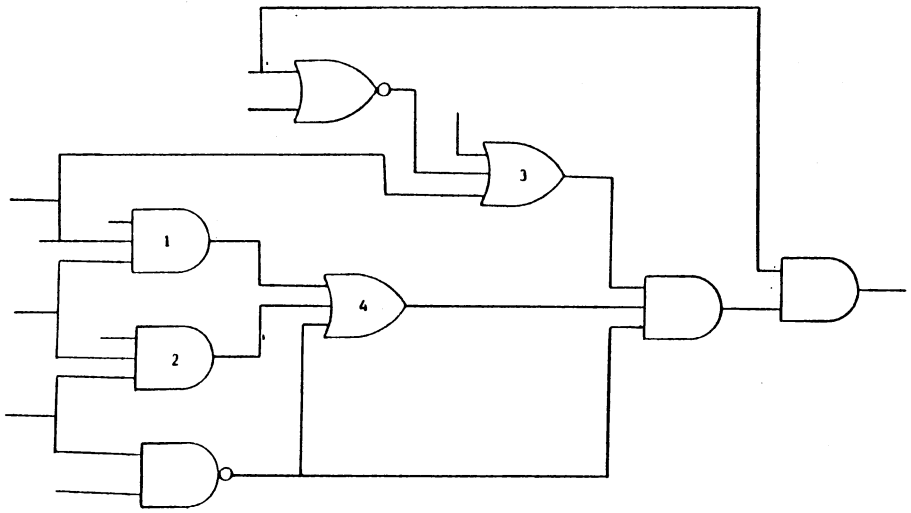


Figure 39. Example of Gates which can be used as Blocking Gates

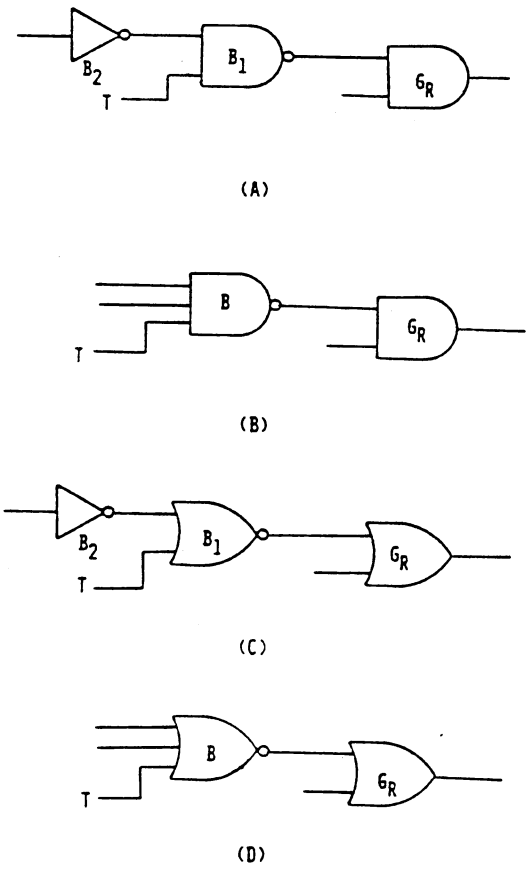


Figure 40. Alternatives for Blocking Gates

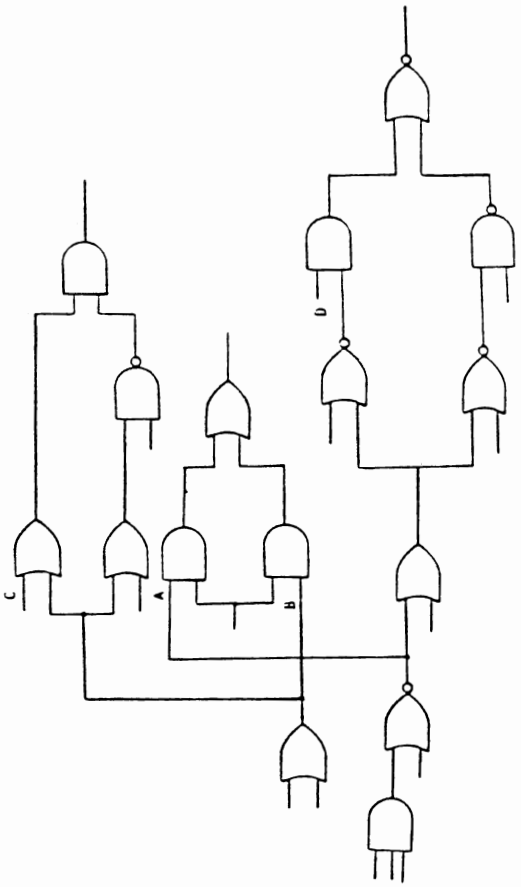


Figure 41. Example of Inputs which can be used as Test Inputs

chooses the same paths to propagate both s-a-0 and s-a-1 faults on a primary input to the outputs of the circuit (like the one written for this research study), a test for primary input s-a-0/1 is the same for the fault s-a-1/0 on the same primary input except that the value on that primary input is complemented.

A program has been written in PASCAL programming language which generates tests for the circuits with the added blocking gates and test inputs, or having the same property. This program starts test generation from the primary inputs and covers all the distinct paths in the circuit. If a conflict is found during the path sensitization, it will be flagged out and another choice will be tried. A choice is either a primary input or a branch of a fanout origin. Since this program generates tests only for the complete paths from the primary inputs to the primary outputs, then if a conflict is found in the path sensitization process, it may be that no tests will be generated for some of the lines on that path. In the other words there is no guarantee that test will be generated for all the testable lines in the circuit. In the justification process, all the choices will be considered until either a test is generated or no test exist for the path. However, any conflict will be reported. The following information should be provided for each gate in the circuit for the use of program by a user:

1. Gate number (an integer)
2. Gate type (ANDE, ORE, NAND, NOR, INV, INPUTE OUTPUTE)
3. # of inputs to the gate
4. Fanin numbers (to what gates the inputs are connected)
5. Number of fanout branches
6. To what gate each fanout branch is connected

It should be kept in mind that the input to an INPUTE gate is itself and the output of an OUTPUTE gate is also itself. A listing of this program is given in appendix A which includes a sample input data in the second page. The general performance of the program can be described as follows. A primary input will be considered as the starting point. It will be tried to find a sensitized path from that input to a primary output. Whenever a value is assigned in this process, the effect will be propagated forward and backward. It means that if a value is assigned to the output of a gate, then it will be determined if any of the inputs to that gate has to be set to a certain value because of the value assignment on the output of that gate. This is called the backward propagation. If any of the inputs of that gate is fanout origin and that input is forced to some value because of the backward propagation of the value on the output of that gate, then effect of that value assignment on that origin must be found on all the other branches of that fanout origin. This is called the forward propagation. If there is no inconsistency in value assignments then the program proceeds to complete the sensitized path, otherwise a flag will be set and another choice will be considered and all the value assignments due to the last choice will be erased. After successful completion of the path sensitization, the gates which have been found during the path sensitization for the justification process will be processed. If a conflict is found in this process then the program reports that conflict and tries other choices until either a test is found or no choice is remained.

Figure 42 shows a redundant circuit and Figure 43 shows the same circuit in Figure 42 with the exception that blocking gates and test

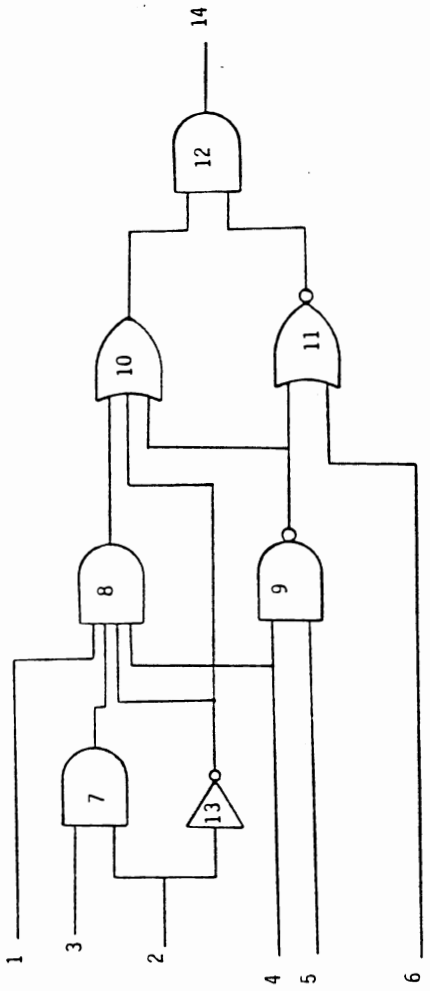


Figure 42. Example of a Combinational Circuit

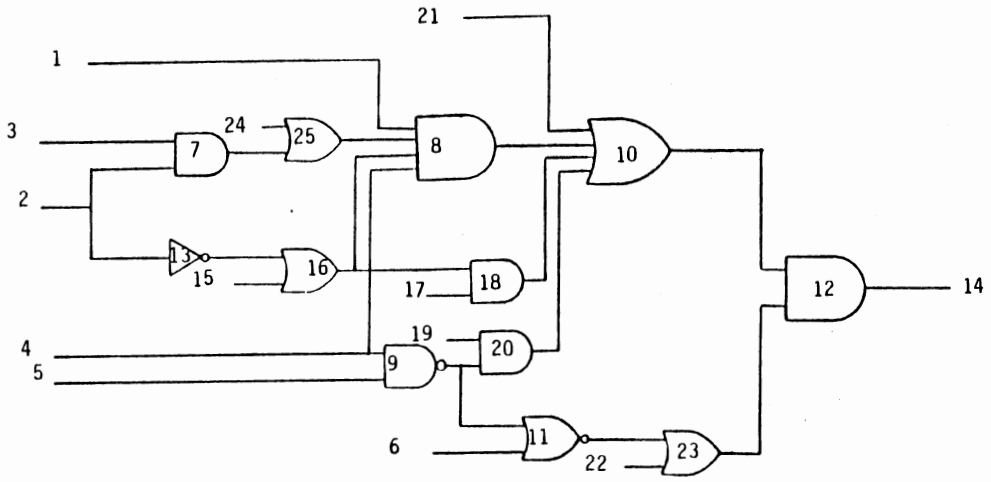


Figure 43. The Circuit in Figure 42 with Added Blocking Gates and Test Inputs

inputs are added. The results from the test generation program after running on these two circuits are given in the appendix C. From those results it can be seen that how the testability of the circuit has been improved.

In Figure 44 the normalized measured times for the circuits, which are designed according to the proposed design method, with different numbers of gates are shown. Figure 45 shows a plot of the data shown in Figure 44. From this figure it can be seen that the required time for test generation is growing proportional to N^2. In Figure 45, the data points marked by circles correspond to the different combinations of TI arithmetic logic unit/function generator, type SN54181, and look-ahead carry generator, type SN54182. Each circuit was changed to a pifd circuit using the program on appendix B. The data points marked by crosses correspond to an arbitrary pifd circuit which was duplicated each time and the outputs of one circuit were used as inputs to some of the gates of the other circuit to make a larger circuit. Each circuit was made a pifd circuit using the program in appendix B. One of the advantages of this method is that a designer can freely design the desired circuit without considering this design method and after the design is complete then necessary blocking gates and test inputs can be added to the circuit. The disadvantage of this design method is the addition of gates and inputs which sometimes can be very large. One way to cope with this problem is to identify the reconvergent gates that most of the conflicts occurs on them and add the blocking gates and test inputs only to those reconvergent gates.

NUMBER OF GATES	NORMALIZED TIME
18	1
36	1.7
78	6.45
209	46.58
458	324.65
875	1153

Figure 44. Timing Results from the Test Generation Program

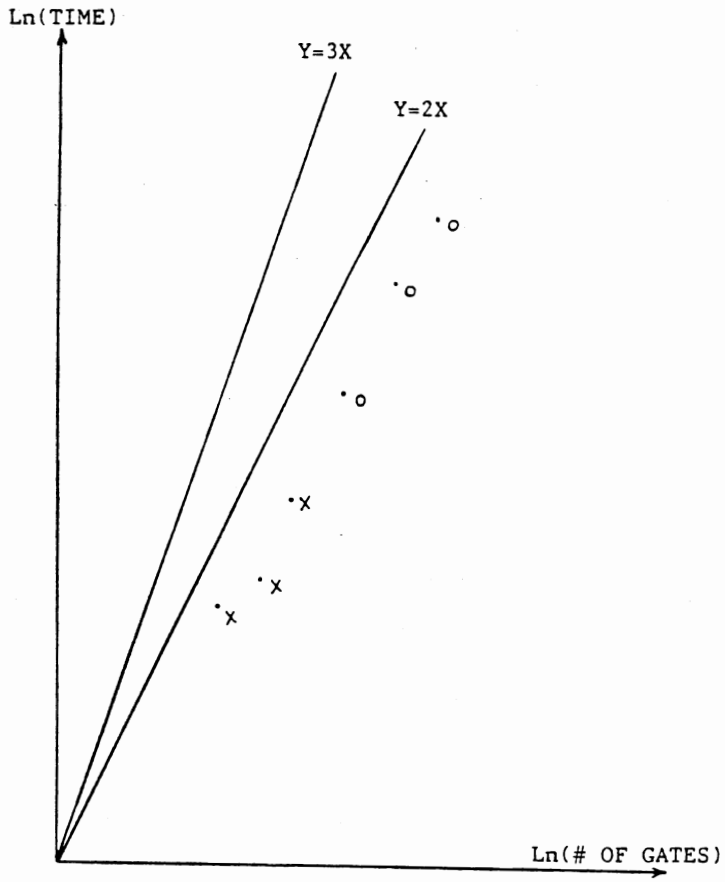


Figure 45. The Plot of the Time for Test Generation Versus Number of Gates (In Ln-Ln Scale). Circles Represent the Data from ALU Funcation Generator. Crosses Represent the Date from Arbitrary Circuits.

CHAPTER V

CONCLUSIONS AND RECOMMENDATION

In the last two chapters, several circuit topologies have been identified for which tests can be generated in N^2 time. The concept of preprocessing of fanout origins has been introduced which for a certain type of circuit make the behavior of a circuit completely conflict free regarding the test generation process. The path-independent fault-detecting (pifd) circuits have been introduced for which tests can be generated in N^2 time. Also an upper bound has been found for the number of tests for such circuits. A simple design method has been proposed which can change any arbitrary combinational circuit to a pifd circuit. Also it has been shown that the required time for the test generation will be halved if a circuit is designed according to the proposed design method. It has been shown that by using some of the properties of a circuit, it is possible to reduce the number of gates and inputs which must be added to the circuit. Also it has been shown that it is possible to have only one extra input to a chip for all the added gates and inputs to the circuit.

Experimental results show that the number of gates and inputs added to a circuit using the the proposed design method can be excessive. Further research is needed to extract the properties of pifd circuits which may be used to improve the proposed design method. Also there may be other circuit topologies which are testable in N^2 time for which

further research is needed to identify such topologies. The preprocessing of fanout origins seems to be a powerful tool for predicting the behavior of the circuits regarding the test generation process. In this research, this process was used only for a simple topology for the loops but actually for many of the other circuit topologies this process is applicable. Further research is needed to identify the further application of this process.

REFERENCES

Berglund, N.C. "Processor Development in The LSI Environment." IBM System/38 Technical Development, Dec. 1978.

Breuer, M.A., and Friedman A.D. Diagnosis and Reliable Design Systems. New York: Computer Science Press, Inc., 1976.

Fujiware Hideo., and Toida Shunichi. "The Complexity of Fault Detection Problems." IEEE Trans. on computers, Vol. C-31, No. 6 (1982), pp 555-559.

Goel, P. "Test Generation Costs Analysis and Projections." presented at the 17th Design Automation Conf., Minniapolis, MN. 1980.

Hayes, P. John. "On Modifying Logic Networks to Improve Their Diagnosability." IEEE Trans. on computers, Vol. C-23, No. 1 (1974), pp 56-62.

Ibarra, H. Oscar., and Sahni, K. Sataj. "Pollynomially Complete Fault Detection Problems for Combinational Logic Circuits." IEEE Trans. on computers, Vol. C-24, No. 3 (1975), pp 242-249.

Roth, J. Paul. "Diagnosis of Automate Failure: A Calculus and a Method." IBM Journal of Research & Development, 10 (1966), pp 278-281.

Standish, A. Thomas. Data Structure Techniques. Addison-Wesely Publishing Company, Inc., 1980.

Thomas, J.J. "Automatic Diagnostic Test Program for Digital Networks." Computer Design (1971), pp 63-67.

Williams, W. Thomas. and Parker, P. Kenneth. "Design for Testability-A Survey." Proc. IEEE, Vol. 71, No. 1, Jan. 1983, pp 98-112.

Williams, W. Thomas. and Parker, P. Kenneth. "Testing Logic Networks and Designing for Testability." Computer, Oct. 1979, pp 9-18.

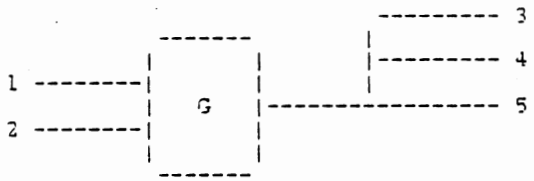
APPENDIXES

APPENDIX A

LISTING OF THE TEST GENERATION PROGRAM

```
PROGRAM TESTLOGICCIRCUIT(INPUT,OUTPUT,INFILE,OUTFILE);

(*THE PURPOSE OF THIS PROGRAM IS TO GENERATE TESTS FOR LOGIC
CIRCUITS IN WHICH   NO BACK-TRACKING IS NEEDED NEITHER IN PATH
SENSITIZATION NOR IN JUSTIFICATION   PROCESSES.THIS PROGRAM
GENERATES TESTS ONLY FOR COMBINATIONAL CIRCUITS. THIS   PROGRAM
STARTS GENERATING TESTS FROM PRIMARY INPUTS.BUT IT WILL COVER ALL
THE   DISTINCT PATHS IN THE CIRCUIT.IF A CONFLICT IS FOUND DURING
THE PATH SENSITIZATION   IT FLAGS OUT THAT CONFLICT AND TRIES
ANOTHER CHOICE.IN THE JUSTIFICATION PROCESS   ALL THE CHOICES WILL
BE CONSIDERED UNTIL A TEST IS GENERATED.HOWEVER,ANY CONFLICT WILL
BE REPORTED.THIS PROGRAM DOES NOT GENERATE TESTS FOR THE REMAINIG
NETS WHICH   HAVE NOT BEEN TESTED EVEN THERE EXIST TESTS FOR
THEM.THE WAY THAT CHOICES ARE   MADE IN PATH SENSITIZATION IS AS
FOLLOWING.THE FIRST CHOICES ARE PRIMARY INPUTS.   WHENEVER A FANOUT
ORIGIN IS FACED,DEPENDING ON THE NUMBER OF INPUTS TO THE GATE
WHICH HAS THAT FANOUT ORIGIN ON ITS OUTPUT OR NUMBER OF FANOUT
BRANCHES WHICH   HAVE NOT BEEN TESTED YET,ONE OR MORE BRACHES OF THE
FANOUT ORIGIN WILL BE ADDED   TO THE CHOICES.FOR EXAMPLE CONSIDER
THE FOLLOWING GATE WITH 2 INPUTS AND THE   FANOUT ORIGIN WITH 3
BRANCHES.IF LINE 1 IS UNDER TEST AND LINE 2 HAS NOT BEENTED   TESTED
BEFORE,THEN THE TWO OF BRANCHES WILL BE CONSIDERED AS CHOICES FOR
LINE   1 AND THE THIRD BRANCH WILL BE CONSIDERED WHEN LINE 2 IS
GOING TO BE TESTED.
```




```
 THEN FOR THE FIRST TIME IF THE NUMBER OF FANOUT BRANCHES ARE MORE
THAN THE NIMBER  OF INPUTS TO THE GATE,#OF FANOUT BRANCHES-#OF
INPUTS+1 OF FANOUT BRANCHES WILL BE  CONSIDERED FOR ONE OF THE
INPUTS AND EACH OF THE REMAINING INPUTS WILL TAKE ONE  OF THE
REMAINIG FANOUT BRANCHES WHICH HAS NOT BEEN TESTED BEFORE.
WHENEVER AN INPUT TO A GATE IS CONSIDERED FOR PATH
SENSITIZATION,FIRST THE OTHER  INPUTS WILL BE SET TO PROPAGATION
VALUES ONE AT A TIME AND THE EFFECT OF THIS  VALUE ASSIGNMENT WILL
BE FOUND FORWARD AND BACKWARD IN THE CIRCUIT.IF NO CONFLICT  IS
FACED THEN THE ERROR SIGNAL WILL BE PROPAGATED TO THE OUTPUTOF THE
GATE AND  THE PROCESS CONTINUES,OTHERWISE ALL THE VALUE ASSIGNMENTS
SINCE THE LAST CHOICE  WILL BE ERASED AND ANOTHER CHOICE WILL BE
CONSIDERED.AT THE END USER WILL BE  PROVIDED WITH THE FOLLOWING
INFORMATION.
  1.TESTS GENERATED.
  2.CRITICAL VALUES AND NON-CRITICALVALUES OF ALL INTENAL LINES OF
    THE CIRCUIT FOR EACH TEST.
  3.ALL THE CONFLICTS FACED DURING THE PATH SENSITIZATION AND THE
    JUSTIFICATION PROCESS.
  4.LIST OF THE INCOMLETE TESTS.
  5.LIST OF LINES THAT NO TESTS HAVE BEEN GENERATED FOR THEM.

##############################################################################
 INPUT FORMAT:
    FOR EACH GATE IN THE CIRCUIT THE FOLLOWING INFORMATION MUST BE
 PROVIDED BY  THE USER TO THE PROGRAM:
 1.LINE NUMBER
 2.GATE TYPE(ANDE,ORE,NAND,NOR,INV,INPUTE,OUTPUTE)
```

```
 3.#OF INPUTS
 4.TO WHAT GATE EACH INPUT IS CONNECTED
 5.#OF FANOUT BRANCHES
 6.TO WHAT GATE EACH FANOUT BRACH FANS IN

 EXAMPLE:
         3 ANDE 2 8 9 4 10 15 25 30
         8 INPUTE 1 8 2 7 27
         14 OUTPUTE 1 12 1 14

 IN THE INPUT FILE TWO OTHER ITEMS MUST ALSO APPEAR BEFORE ANYTHING
ELSE.THE  FIRST ONE IS NAME OF THE CIRCUIT AND THE SECOND ONE THE
NIMBER OF GATES IN THE  CIRCUIT,WHICH MUST BE AN INTEGER. THE
PROGRAM WILL PROMPT A MESSAGE ASKING FOR  THE NAME OF THE INPUT
FILE.
################################################################################
 OUTPUT:
    THE OUTPUT INCLUDES ALL THE INFORMATION MENTIONED ABOVE,AND THE
 NAME OF OUTPUT  FILE IS 'OUTFILE'.
################################################################################
 DATA STRUCTURE:
   TEST' ARRAY:
     FOR EACH GATE IN THE INPUT FILE A RECORD IS DEFINED WHICH KEEPS
ALL THE INFORMATION  PROVIDED BY THE USER AND OTHER INFORMATION
PROVIDED BY THE PROGRAM WHEN EXECUTED.  THESE INFORMATION CONSIST OF
THE VALUE ON THE OUTPUT OF EACH GATE,THE VALUE(S) THAT  THAT OUTPUT
HAS BEEN TESTED FOR,AND THE STATUS OF THE VALUE ON THE OUTPUT OF THE
 GATE(WHETHER OR NOT THAT VALUE IS A CRITICAL VALUE) FOR A
PARTICULAR TEST.AN ARRAY  OF THIS RECORD TYPE KEEPS SUCH INFORMATION
ON ALL THE GATES IN THE CIRCUIT.THIS  ARRAY IS CALLED 'TEST'.

   'WLIST' ARRAY:
     AN IMPORTANT ARRAY USED BY THIS PROGRAM IS
 'WLIST(WAITINGLIST)'.AT THE  BEGINNING IT CONTAINS INPUTE-GATES IN A
CODED FORM(INPUTE#*MAX3+INPUT#).THE REASON  FOR ENCODING THE INPUTE
GATES IS COMPATIBILITY WITH THE OTHER INFORMATION WHICH  WILL BE
ADDED TO THE  WLIST' ENCODED IN THE SAME FORM.AS THE PROGRAM FACES
DIFFERENT  CHOICES IN THE PATH SENSITIZATION,IT WILL ADD THEM TO THE
 'WLIST'.IF A PATH HAS  BEEN SENSITIZED SUCCESSFULLY THEN THE
JUSTIFICATION PROCESS START.THE SAME 'WLIST'  ARRAY WILL BE USED TO
KEEP TRACK OF THE CHOICES ENCOUNTERED IN THIS PROCESS,AND  THE
PROGRAM REMEMBER WHERE IT LEFT THE PATH SENSITIZATION PROCESS.THE
POINTER TO  THE 'WLIST' ARRAY WILL BE RESTORED WHEN THE
JUSTIFICATION PROCESS IS FINISHED.  THE POINTER TO THIS ARRAY IS
CALLED 'FWLIST'.



  CONST  MAX1=10;
         MAX2=100;
         MAX3=1000;


  TYPE   GTYPE=(ANDE,ORE,NAND,NOR,INPUTE,OUTPUTE,INV);

          CIRCUITDES=
                   RECORD
                         GATETYPE    :GTYPE;
                         GATENUM     :1..MAX2;
```

```
                    NOINPUTS    :1..MAX1;
                    INPUTS      :ARRAY[1..MAX1] OF INTEGER;
                    INVALUE     :ARRAY[1..MAX1] OF INTEGER;
                    FANOUTNUM   :1..MAX1;
                    FANOUTS     :ARRAY[1..MAX1] OF INTEGER;
                    OUTVALUE    :INTEGER;
                    CRITICAL    :ARRAY[1..MAX1] OF INTEGER;(*0&1
                                 FOR CRITICAL 0&1.2 FOR BOTH*)
                    TEMPCV      :ARRAY[1..MAX1] OF INTEGER;
                END;


VAR  TEST         :ARRAY[1..MAX2] OF CIRCUITDES;
     WLIST        :ARRAY[1..MAX2] OF INTEGER;(*KEEPS TRACK OF LINES
                   WAITING TO BE USED IN PATH SENSITIZATION OR
                   JUSTIFICATION PROCESS.*)
     FWLIST       :INTEGER; (*POINTER TO WLIST*)
     ADAR         :ARRAY[1..MAX2] OF INTEGER;(*THIS ARRAY KEEPS THE
                   STARTING ADDRESS OF THE SET OF LINES
                   ON 'ASAR' FOR EACH ENTRY OF 'WLIST'.*)
     ADJADAR      :ARRAY[1..MAX2] OF INTEGER;
     ASAR         :ARRAY[1..MAX2] OF INTEGER;(*KEEPS TRACK OF THE
                   LINES THAT SOME VALUES ARE ASSIGNED TO THEM.*)
     FASAR        :INTEGER;(*POINTER TO ASAR*)
     MCRITVAL     :INTEGER;
     CRITVAL      :INTEGER;
     I1,J1,K1     :INTEGER;
     FLAGP        :INTEGER;(*THIS FLAG IS SET TO '1' IF THERE IS A
                   CONFLICT.*)
     NOOFNODES    :INTEGER;
     FARRAY       :ARRAY[1..MAX2]OF INTEGER;(*KEEPS THE LIST OF
                   FANOUTS TO BE IMPLEMENTED FOR FORWARD
                   PROCEDURE.*)
     FPOINTER     :INTEGER;(*POINTER TO FARRAY*)
     ADJUST       :ARRAY[1..MAX2]OF INTEGER;(*KEEPS TRACK OF THE
                   NODES WITH THE VALUE OF THE OUTPUTS SPECIFIED
                   BUT THE INPUTS ARE NOT JUSTIFIED FOR THAT
                   VALUE.*)
     FADJUST      :INTEGER;(*POINTER TO ADJUST ARRAY.*)
     INPUTLIST    :ARRAY[1..MAX2] OF INTEGER;
     INPUTCOUNT   :INTEGER;
     GUT          :INTEGER;
     IUT          :INTEGER;
     FLAGC        :INTEGER;
     CADAR        :ARRAY[1..MAX2] OF INTEGER;(*FOR EACHENTRY OF THE
                   WLIST KEEPS POINTER TO   'CGATES' ARRAY WHERE THE
                   CRITICAL GATES ADDED TO 'CGATES' ARRAY AFTER THAT
                   ENTRY MUST BE ERASED WHEN THAT ENTRY OF THE 'WLIST'
                   IS GOING TO BE PROCESSED.*)
     CGATES       :ARRAY[1..MAX2] OF INTEGER;(*KEEPS TRACK OF
                   CRITICAL GATES.*)
     CVALUES      :ARRAY[1..MAX2] OF INTEGER;(*KEEPS THE OUTPUT
                   VALUE OF CGATES.*)
     FCGATES      :INTEGER;(*POINTER TO 'CGATES' ARRAY.*)
     TEMPCRITVAL:INTEGER;
     NAME         :PACKED ARRAY[1..40] OF CHAR;
     PCOUNT       :INTEGER;(*KEEPS TRACK OF TEST-NUMBER IN
                   PRINTTEST PROCEDURE.*)
     INFILE       :TEXT;
     OUTFILE      :TEXT;
```

```
PROCEDURE STATUS;

 VAR PS,FS:INTEGER;
 BEGIN
      FOR PS:=1 TO NOOFNODES DO
          BEGIN
               WRITELN('TEST[',PS:2,'].OUTVALUE',TEST[PS].OUTVALUE:2);
               FOR FS:=1 TO TEST[PS].NOINPUTS DO
                   BEGIN
                        WRITELN('TEST[',PS:2,'].INVALUE[',FS:2,']=',
                                TEST[PS].INVALUE[FS]:2,'----',
                                'TEST[',PS:2,'].TEMPCV[',FS:2,']=',
                                TEST[PS].TEMPCV[FS]:2,'----',
                                'TEST[',PS:2,'].CRITICAL[',FS:2,']=',
                                TEST[PS].CRITICAL[FS]:2);
                   END;
          END;
      FOR PS:=1 TO FWLIST DO
          WRITELN('WLIST[',PS:2,']=',WLIST[PS]:6,'----','ADAR[',PS:2,']='
                  ,ADAR[PS]:2);
      FOR FS:=1 TO FASAR DO
          WRITELN(FS:2,'-----','ASAR[',FS:2,']=',ASAR[FS]:6);
  END;

PROCEDURE INITIALIZE;

   VAR  CIRCUITNAME:PACKED ARRAY[1..40] OF CHAR;
        I,J,K        :INTEGER;


   BEGIN
        FWLIST:=0;
        FASAR:=0;
        FADJUST:=0;
        FGATES:=0;
        FCOUNT:=0;
        READ(INFILE,CIRCUITNAME);
        WRITELN(OUTFILE,CIRCUITNAME);
        READ(INFILE,NOOFNODES);
        FOR K:=1 TO NOOFNODES DO
             BEGIN
                 READ(INFILE,I);
                 TEST[I].GATENUM:=I;
                 READ(INFILE,TEST[I].GATETYPE,TEST[I].NOINPUTS);
                 IF(TEST[I].GATETYPE=INPUTE)THEN
                   BEGIN
                        FWLIST:=FWLIST+1;
                        WLIST[FWLIST]:=I*MAX3+I;
                        INPUTLIST[FWLIST]:=I*MAX3+I;
                        WRITELN('WL=',WLIST[FWLIST]);
                   END;
                 FOR J:=1 TO TEST[I].NOINPUTS DO
                 READ(INFILE,TEST[I].INPUTS[J]);
                 READ(INFILE,TEST[I].FANOUTNUM);
                 FOR J:=1 TO TEST[I].FANOUTNUM DO
                   READ(INFILE,TEST[I].FANOUTS[J]);
             END;
         INPUTCOUNT:=FWLIST;
         FOR J:=1 TO NOOFNODES DO
```

```
                BEGIN
                     FOR K:=1 TO MAX1 DO
                        BEGIN
                             TEST[J].CRITICAL[K]:=-1;
                             TEST[J].TEMPCV[K]:=-1;
                             TEST[J].INVALUE[K]:=-1
                        END;
                     TEST[J].OUTVALUE:=-1
                END;
           FOR J:=1 TO FWLIST DO
                BEGIN
                     ADAR[J]:=1;
                     ADJADAR[J]:=0;
                     CADAR[J]:=0;
                END;
      END;

(*THE FOLLOWING PROCEDURE ADDS ONE ELEMENT TO 'ASAR' ARRAY AND
ASSIGNS THE DESIRED VALUE TO THAT LINE. GATENUMBER=X,GATEINPUT=Y*)

   PROCEDURE ADDONETOASAR(VAR
             GATENUMBER,GATEINPUT,FLAGCORRECT:INTEGER);
      VAR JAA  :INTEGER;

      BEGIN
           JAA:=1;
   WRITELN('GATENUMBER=',GATENUMBER:3,
           '     ','GATEINPUT=',GATEINPUT:3);
           WHILE(TEST[GATENUMBER].INPUTS[JAA]<>GATEINPUT)DO
                JAA:=JAA+1;
           TEST[GATENUMBER].INVALUE[JAA]:=TEST[GATEINPUT].OUTVALUE;
           FASAR:=FASAR+1;
           ASAR[FASAR]:=FLAGCORRECT*(GATENUMBER*MAX3+GATEINPUT);
      END;


   PROCEDURE MAINFORWARD(VAR GATENO,INPUTNUM:INTEGER);FORWARD;
   PROCEDURE BACKWARD(VAR BGUT,BVALUE:INTEGER);FORWARD;

(*THE FOLLOWING PROCEDURE CHECKS THE VALUE ASSIGNMENT ON THE INPUT
OF A GATE   (FCGATE=X) TO SEE IF IT IS COMPATIBLE WITH THE OUTPUT
VALUE OF THAT GATE WHICH   IS NOT A PROPAGATION VALUE. THIS
PROCEDURE IS CALLED FROM THE 'FIRSTFORWARD'   PROCEDURE.
FCGATE=X,FCINPUT=Y,FCINVAL=0 FOR AND & NAND,1 FOR OR & NOR.*)

   PROCEDURE FORWARDCORRECTION(VAR FCGATE,FCINPUT,FCINVAL:INTEGER);
     VAR JFC,COUNTF,KFC:INTEGER;

       BEGIN
            FLAGF:=0;
            JFC:=1;
            WHILE((TEST[FCGATE].INVALUE[JFC]<>FCINVAL)AND
                  (JFC<TEST[FCGATE].NOINPUTS))DO
                 JFC:=JFC+1;
                 IF(TEST[FCGATE].INVALUE[JFC]<>FCINVAL)THEN
                   BEGIN
                        COUNTF:=0;
                        FOR JFC:=1 TO TEST[FCGATE].NOINPUTS DO
                             IF(TEST[FCGATE].INVALUE[JFC]=-1)THEN
                               COUNTF:=COUNTF+1;
```

```
                    IF(COUNTF=1)THEN(*THIS CASE MAY
                       NEVER OCCURES.JUST FOR INSURANCE*)
                      BEGIN
                           IF(TEST[FCINPUT].OUTVALUE
                              =FCINVAL)THEN
                             BEGIN
                                  KFC:=-1;
                                  ADDONETOASAR(FCGATE,
                                               FCINPUT,KFC);
                             END
                            ELSE
                                FLAGP:=1;
                      END
                    ELSE(*IF COUNTF>=1*)
                        BEGIN
                             IF(COUNTF=2)THEN
                               BEGIN
                                    IF(TEST[FCINPUT].OUTVALUE=
                                       FCINVAL)THEN
                                      BEGIN
                                           KFC:=-1;
                                           ADDONETOASAR(FCGATE,
                                                   FCINPUT,KFC);
                                      END
                                     ELSE
                                         BEGIN
                                              KFC:=-1;
                                              ADDONETOASAR(FCGATE,
                                                     FCINPUT,KFC);
                                              JFC:=1;
                                              WHILE(TEST[FCGATE]
                                                .INVALUE[JFC]<>-1)DO
                                                   JFC:=JFC+1;
                                              TEST[TEST[FCGATE].INPUTS
                                              [JFC]].OUTVALUE:=FCINVAL;
                                              KFC:=-1;
                                              ADDONETOASAR(FCGATE,
                                               TEST[FCGATE].
                                               INPUTS[JFC],KFC);
                                              FOR KFC:=1 TO TEST[TEST
                                               [FCGATE].INPUTS[JFC]]
                                               .FANOUTNUM DO
                                                  IF(TEST[TEST[FCGATE]
                                                    .INPUTS[JFC]].
                                                    FANOUTS[KFC]<>
                                                    FCGATE)THEN
MAINFORWARD(TEST[FCGATE].INPUTS[JFC].TEST[TEST[FCGATE].
            INPUTS[JFC]].FANOUTS[KFC]);
BACKWARD(TEST[FCGATE].INPUTS[JFC],FCINVAL);
                                                        END;(*OF ELSE*)
                                      END(*OF COUNTF=2*)
                                    ELSE(*COUNTF>2*)
                                        BEGIN
                                             KFC:=-1;
                                             ADDONETOASAR(FCGATE,
                                                     FCINPUT,KFC);
                                        END;
                                  END;(*OF IF COUTF>1*)
                        END
                       ELSE
                           BEGIN
```

```
                        KFC:=-1;
                        ADDONETOASAR(FCGATE,FCINPUT,KFC);
                    END;
 END;(*END OF FORWARDCORRECTION*)


(*THE FOLLOWING PROCEDURE CHECKS THE VALUE ASSIGNMENT ON ONE INPUT
OF A GATE  TO SEE IF IT IS COMPATIBLE WITH THE OUTPUT VALUE OF THAT
GATE WICH HAS BEEN  ALREADY ASSIGNED AND IT IS A PROPAGATION
VALUE.*)

   PROCEDURE FIRSTFORWARDCORRECTION(VAR
                                FFCGATE,FFCINPUT,FFCINVAL:INTEGER);

      VAR KFFC:INTEGER;

      BEGIN
           IF((TEST[FFCGATE].GATETYPE=ANDE)
             OR(TEST[FFCGATE].GATETYPE=ORE))THEN
             BEGIN
                  IF(TEST[FFCGATE].OUTVALUE=1-FFCINVAL)THEN
                    BEGIN
                         IF(TEST[FFCINPUT].OUTVALUE=FFCINVAL)THEN
                           FLAGP:=1
                         ELSE
                             BEGIN
                                  KFFC:=-1;
                                  ADDONETOASAR(FFCGATE,FFCINPUT,KFFC);
                             END;
                    END
                  ELSE
                      FORWARDCORRECTION(FFCGATE,FFCINPUT,FFCINVAL);
             END
           ELSE(*NAND&NOR*)
               BEGIN
                    IF(TEST[FFCGATE].OUTVALUE=FFCINVAL)THEN
                      BEGIN
                           IF(TEST[FFCINPUT].OUTVALUE=FFCINVAL)THEN
                             FLAGP:=1
                           ELSE
                               BEGIN
                                    KFFC:=-1;
                                    ADDONETOASAR(FFCGATE,FFCINPUT,KFFC);
                               END;
                      END
                    ELSE
                        FORWARDCORRECTION(FFCGATE,FFCINPUT,FFCINVAL);
                END;
      END;(*END OF FIRSTFORWARDCORRECTION*)


(*THE FUNCTION OF THE FOLLOWING PROCEDURE IS AS FOLLOWS:
  IF OUTPUT OF GATE Y IS INPUT TO GATE X; THEN ACCORDING TO VALUES
ON Y AND OTHER INPUTS TO X; THE VALUE ON THE OUTPUT OF THE GATE X
WILL BE DETERMINED AND THE LINE CONNECTING Y TO X WILL BE KEPT ON
ARRAY 'ASAR'. WHEN THIS ENTRY OF 'ASAR' IS TO BE REMOVED;  THEN IF
THE OUTPUT OF X WAS FORCED TO SOME VALUE BECAUSE OF THE VALUE
ASSIGNMENT ON THE LINE CONNECTING X TO Y; THE VALUE ON THE OUTPUT OF
THE X SHOULD BE ERASED AS WELL AS THE VALUE ON THE LINE CONNECTING
X AND Y. ONLY THE VALUE ON THE LINE CONNECTING X TO Y MUST BE
ERASED OTHERWISE. VARIABLE FLAGR IS SET TO 1 AND -1 TO INDICATE
```

```
WHETHER THE OUTPUT OF X IS FORCED TO SOME VALUE BY THE VALUE
ASSIGNMENT ON THE LINE CONNECTING Y TO X(THE OUTPUT OF Y). ENTRIES
OF THE 'ASAR' ARRAY HAVE THE FOLLOWING FORM:(X*MAX3+Y). FGATENO=X,
FGATEINPUT=Y*)

   PROCEDURE FORWARD(VAR FGATENO,FGATEINPUT:INTEGER);

       VAR   J,INVAL,FLAGR      :INTEGER;


           BEGIN
                IF((TEST[FGATENO].OUTVALUE<>-1)AND
                  (TEST[FGATENO].GATETYPE<>OUTPUTE)) THEN
                  BEGIN
                       IF((TEST[FGATENO].GATETYPE=ANDE)OR
                          (TEST[FGATENO].GATETYPE=NAND))THEN
                         INVAL:=0
                        ELSE
                            INVAL:=1;
                            FIRSTFORWARDCORRECTION(
                                  FGATENO,FGATEINPUT,INVAL);
                  END
                ELSE(*TEST[FGATENO].OUTVALUE=-1*)
                  BEGIN
                    IF(TEST[FGATENO].GATETYPE=OUTPUTE)THEN
                      BEGIN
                           TEST[FGATENO].OUTVALUE:=
                            TEST[FGATEINPUT].OUTVALUE;
                           TEST[FGATENO].INVALUE[1]:=
                            TEST[FGATEINPUT].OUTVALUE;
                           FASAR:=FASAR+1;
                           ASAR[FASAR]:=FGATENO*MAX3+FGATEINPUT;
                      END;
                       IF(TEST[FGATEINPUT].OUTVALUE=0)THEN
                         BEGIN
                              IF(TEST[FGATENO].GATETYPE=ANDE)THEN
                                BEGIN
                                     TEST[FGATENO].OUTVALUE:=0;
                                     J:=1;
                                     WHILE(TEST[FGATENO].INPUTS[J]<>
                                           FGATEINPUT)DO
                                          J:=J+1;
                                     TEST[FGATENO].INVALUE[J]:=0;
                                     FASAR:=FASAR+1;
                                     ASAR[FASAR]:=FGATENO*MAX3+FGATEINPUT;
                                END;
                               IF((TEST[FGATENO].GATETYPE=NAND)OR
                                  (TEST[FGATENO].GATETYPE=INV))THEN
                                 BEGIN
                                      TEST[FGATENO].OUTVALUE:=1;
                                      J:=1;
                                      WHILE(TEST[FGATENO].INPUTS[J]<>
                                            FGATEINPUT)DO
                                           J:=J+1;
                                      TEST[FGATENO].INVALUE[J]:=0;
                                      FASAR:=FASAR+1;
                                      ASAR[FASAR]:=FGATENO*MAX3+FGATEINPUT;
                                 END;
                               IF((TEST[FGATENO].GATETYPE=ORE)OR
                                  (TEST[FGATENO].GATETYPE=NOR))THEN
                                 BEGIN
```

```
                    FLAGR:=1;
                    J:=1;
                    WHILE(TEST[FGATENO].INPUTS[J]<>
                          FGATEINPUT)DO
                         J:=J+1;
                    TEST[FGATENO].INVALUE[J]:=0;
                    FOR J:=1 TO TEST[FGATENO].NOINPUTS DO
                        BEGIN
                            IF(TEST[FGATENO].INVALUE[J]
                               <>0)THEN
                              FLAGR:=-1;
                        END;
                     IF(FLAGR=1)THEN
                       IF(TEST[FGATENO].GATETYPE=NOR)THEN
                          TEST[FGATENO].OUTVALUE:=1
                       ELSE
                           TEST[FGATENO].OUTVALUE:=0;
                       FASAR:=FASAR+1;
                       ASAR[FASAR]:=
                        FLAGR*(FGATENO*MAX3+FGATEINPUT);
                END;

          END;
    IF(TEST[FGATEINPUT].OUTVALUE=1)THEN
      BEGIN
           IF(TEST[FGATENO].GATETYPE=ORE)THEN
             BEGIN
                  TEST[FGATENO].OUTVALUE:=1;
                  J:=1;
                  WHILE(TEST[FGATENO].INPUTS[J]
                        <>FGATEINPUT)DO
                       J:=J+1;
                  TEST[FGATENO].INVALUE[J]:=1;
                  FASAR:=FASAR+1;
                  ASAR[FASAR]:=FGATENO*MAX3+FGATEINPUT;
             END;
           IF((TEST[FGATENO].GATETYPE=NOR)OR
              (TEST[FGATENO].GATETYPE=INV))THEN
             BEGIN
                  TEST[FGATENO].OUTVALUE:=0;
                  J:=1;
                  WHILE(TEST[FGATENO].INPUTS[J]<>
                        FGATEINPUT)DO
                       J:=J+1;
                  TEST[FGATENO].INVALUE[J]:=1;
                  FASAR:=FASAR+1;
                  ASAR[FASAR]:=FGATENO*MAX3+FGATEINPUT
             END;
           IF((TEST[FGATENO].GATETYPE=ANDE)OR
              (TEST[FGATENO].GATETYPE=NAND))THEN
             BEGIN
                  FLAGR:=1;
                  J:=1;
                  WHILE(TEST[FGATENO].INPUTS[J]<>
                        FGATEINPUT)DO
                       J:=J+1;
                  TEST[FGATENO].INVALUE[J]:=1;
                  FOR J:=1 TO TEST[FGATENO].NOINPUTS DO
                      BEGIN
                           IF(TEST[FGATENO].INVALUE[J]
                              <>1)THEN
```

```
                                        FLAGR:=-1
                                    END;
                                IF(FLAGR=1)THEN
                                  IF(TEST[FGATENO].GATETYPE=ANDE)THEN
                                    TEST[FGATENO].OUTVALUE:=1
                                  ELSE
                                    TEST[FGATENO].OUTVALUE:=0;
                                 FASAR:=FASAR+1;
                                 ASAR[FASAR]:=
                                  (FGATENO*MAX3+FGATEINPUT)*FLAGR;
                            END;
                      END;

                END;

          END; (*END OF FORWARD*)


(*THE FUCTION OF THE FOLOOWING PROCEDURE IS TO TAKE THE VALUE ON ONE
BRANCH OF A FANOUT ORIGIN AND PROPAGATE IT FORWARD AS FAR AS
POSSIBLE. ENTRIES OF THE 'FARRAY HAVE THE FOLLOWING
FORMAT:(INPUTNUM*MAX3+GATENO)*)

  PROCEDURE MAINFORWARD;(*INPUTNUM=X,GATENO=Y*)

            VAR X,Y,FLAG,J    :INTEGER;


        BEGIN
             FPOINTER:=0;
             FPOINTER:=FPOINTER+1;
             FARRAY[FPOINTER]:=INPUTNUM*MAX3+GATENO;
             WHILE(FPOINTER>0)DO
                  BEGIN
                       X:=TRUNC(FARRAY[FPOINTER]/MAX3);
                       Y:=FARRAY[FPOINTER]-X*MAX3;
(*           WRITELN('X:=',X:2,'Y:=',Y:2);*)
                       FPOINTER:=FPOINTER-1;
                       FLAG:=0;
                       IF(TEST[X].OUTVALUE=-1)THEN
                         FLAG:=1;
                       FORWARD(X,Y);
                       IF((FLAG=1)AND(TEST[X].OUTVALUE<>-1))THEN
                         BEGIN
                              IF(X<>TEST[X].FANOUTS[1])THEN
                                   (*IF X IS NOT AN OUTPUT*)
                                FOR J:=1 TO TEST[X].FANOUTNUM DO
                                 BEGIN
                                      FPOINTER:=FPOINTER+1;
                                      FARRAY[FPOINTER]:=
                                       TEST[X].FANOUTS[J]*MAX3+X;
                                 END;
                            END;
                  END;
        END;(*END OF MAINFORWARD*)


(*THE FOLLOWING PROCEDURE "BACKWARD" IS A RECURSIVE PROCEDURE WHICH
TAKES A GATE AND FINDS THE EFFECT OF THE VALUE ASSIGNMENT ON THE
OUTPUT OF THAT GATE BACKWARD AS FAR AS POSSIBLE. WHEN A FANOUT
ORIGIN IS FACED THE EFFECT WILL BE FOUND FORWARD ON THE BRANCHES
```

```
OF THAT FANOUT ORIGIN. IF A VALUE ASSIGNMENT ON THE OUTPUT OF A
GATE CANNOT BE PROPAGATED BACKWARD ANYMORE,THEN THAT GATE WILL BE
ADDED TO ADJUSTMENT ARRAY FOR JUSTIFICATION PROCESS.*)


 PROCEDURE BACKWARD;(*BGUT:A GATE NUMBER,BVALUE:VALUE OF OUTPUT OF
                      BGUT*)

          VAR  JB,JJB,AVAL,BCOUNT    :INTEGER;


        BEGIN
             IF(TEST[BGUT].GATETYPE=INPUTE)THEN
               BEGIN
                    TEST[BGUT].INVALUE[1]:=BVALUE;
                    FASAR:=FASAR+1;
                    ASAR[FASAR]:=BGUT*MAX3+BGUT;
               END
             ELSE(*TEST[BGUT].GATETYPE<>INPUTE*)
                 BEGIN
                      IF(BVALUE=0)THEN
                        BEGIN
                             IF((TEST[BGUT].GATETYPE=ORE)OR
                               (TEST[BGUT].GATETYPE=NAND)OR
                               (TEST[BGUT].GATETYPE=INV))THEN
                               BEGIN
                                    FOR JB:=1 TO
                                       TEST[BGUT].NOINPUTS DO
                                       BEGIN
                                            IF(TEST[BGUT].
                                               INVALUE[JB]=-1)THEN
                                              BEGIN
                                                   IF(TEST[BGUT].
                                                     GATETYPE=ORE)THEN
                                                     BEGIN
                                                          TEST[BGUT].
                                                            INVALUE[JB]:=0;
                                                          AVAL:=0;
                                                          TEST[TEST[BGUT].
                                                            INPUTS[JB]].
                                                            OUTVALUE:=0;
                                                          FASAR:=FASAR+1;
                                                          ASAR[FASAR]:=
                                                            BGUT*MAX3+
                                                            TEST[BGUT].
                                                            INPUTS[JB];
                                                     END;
                                                   IF((TEST[BGUT].
                                                      GATETYPE=NAND)OR
                                                     (TEST[BGUT].
                                                      GATETYPE=INV))THEN
                                                     BEGIN
                                                          TEST[BGUT].
                                                            INVALUE[JB]:=1;
                                                          TEST[TEST[BGUT].
                                                            INPUTS[JB]].
                                                            OUTVALUE:=1;
                                                          AVAL:=1;
                                                          FASAR:=FASAR+1;
                                                          ASAR[FASAR]:=
                                                            BGUT*MAX3+
```

```
                                                        TEST[BGUT].
                                                        INPUTS[JB];
                                                    END;
        FOR JJB:=1 TO TEST[TEST[BGUT].INPUTS[JB]].FANOUTNUM DO
         IF FLAGP=0 THEN
         IF(TEST[TEST[BGUT].INPUTS[JB]].FANOUTS[JJB]<>BGUT)THEN
           MAINFORWARD(TEST[BGUT].INPUTS[JB],TEST[TEST[BGUT].
                       INPUTS[JB]].FANOUTS[JJB]);
         IF FLAGP=0 THEN
           BACKWARD(TEST[BGUT].INPUTS[JB],AVAL);
                                                END;
                                    END;
                               END

                          ELSE(*BVALUE=0 AND TEST[BGUT].GATETYPE=NOR,ANDE*)
                              BEGIN
                                   BCOUNT:=0;
                                   FOR JB:=1 TO TEST[BGUT].NOINPUTS DO
                                        BEGIN
                                             IF(TEST[BGUT].INVALUE[JB]<>-1)THEN
                                               BCOUNT:=BCOUNT+1;
                                        END;
                                   IF((TEST[BGUT].NOINPUTS-BCOUNT)<>1)THEN
                                        (*NOT ALL INPUTS OF 'BGUT' CAN BE
                                          SPECIFIED NOW.THEN ADD IT TO THE
                                          ADJUSTMENT ARRY FOR JUSTIFICATION
                                          PROCESS.*)

                                     BEGIN
                                          FADJUST:=FADJUST+1;
                                          ADJUST[FADJUST]:=BGUT;(*ADD ONE
                                                GATE TO ADJUSTMENT ARRAY*)
                                          FASAR:=FASAR+1;
                                          ASAR[FASAR]:=BGUT; (*PUT ONLY
                                              THE GATE NUMBER OF BGUT
                                              IN 'ASAR' ARRAY TO INDICATE
                                              THAT ONLY THE VALUE ASSIGNMENT
                                              ON THE OUTPUT 'BGUT' MUST BE
                                              ERASED WHEN 'ERASE' PROCEDURE
                                              IS CALLED.*)
                                     END
                                   ELSE
                                       BEGIN
                                            JB:=1;
                                  WRITELN('BGUT=',BGUT:3);
                                            WHILE(TEST[BGUT].INVALUE[JB]
                                                  <>-1)DO
 BEGIN
                                               JB:=JB+1;
      WRITELN('BACKWARDJB=',JB:2);
END;
                                            IF(TEST[BGUT].GATETYPE=ANDE)THEN
                                              BEGIN
                                                   TEST[BGUT].INVALUE[JB]:=0;
                                                   AVAL:=0;
                                                   TEST[TEST[BGUT].INPUTS[JB]].
                                                    OUTVALUE:=0;

                                                   FASAR:=FASAR+1;
                                                   ASAR[FASAR]:=BGUT*
                                                    MAX3+TEST[BGUT].INPUTS[JB];
```

```
                                        END
                                       ELSE
                                          BEGIN
                                               TEST[BGUT].INVALUE[JB]:=1;
                                               AVAL:=1;
                                               TEST[TEST[BGUT].INPUTS[JB]].
                                                OUTVALUE:=1;
                                               FASAR:=FASAR+1;
                                               ASAR[FASAR]:=BGUT*MAX3+
                                                TEST[BGUT].INPUTS[JB];
                                          END;
                                      FOR JJB:=1 TO TEST[TEST[BGUT].
                                          INPUTS[JB]].FANOUTNUM DO
                                       IF FLAGP=0 THEN
                                       IF(TEST[TEST[BGUT].INPUTS[JB]].
                                          FANOUTS[JJB]<>BGUT)THEN
   MAINFORWARD(TEST[BGUT].INPUTS[JB],
               TEST[TEST[BGUT].INPUTS[JB]].FANOUTS[JJB]);
                                            IF FLAGP=0 THEN
   BACKWARD(TEST[BGUT].INPUTS[JB],AVAL);
                                  END;
               END;
      END


            ELSE
                BEGIN
                     IF(BVALUE=1)THEN
                       BEGIN
                            IF((TEST[BGUT].GATETYPE=ANDE)OR
                              (TEST[BGUT].GATETYPE=NOR)OR
                              (TEST[BGUT].GATETYPE=INV))THEN
                              BEGIN
                                   FOR JB:=1 TO TEST[BGUT].
                                                NOINPUTS DO
                                       BEGIN
                                            IF(TEST[BGUT].
                                               INVALUE[JB]=-1)THEN
                                              BEGIN
                                                   IF(TEST[BGUT].
                                                      GATETYPE=ANDE)THEN
                                                     BEGIN
                                                          TEST[BGUT].
                                                          INVALUE[JB]:=1;
                                                          AVAL:=1;
                                                          TEST[TEST[BGUT].
                                                           INPUTS[JB]].
                                                           OUTVALUE:=1;
                                                          FASAR:=FASAR+1;
                                                          ASAR[FASAR]:=
                                                           BGUT*MAX3+
                                                           TEST[BGUT].
                                                           INPUTS[JB];
                                                     END;
                                                   IF((TEST[BGUT].GATETYPE
                                                      =NOR)OR(TEST[BGUT].
                                                       GATETYPE=INV))THEN
                                                     BEGIN
                                                          TEST[BGUT].
                                                           INVALUE[JB]:=0;
                                                          TEST[TEST[BGUT].
```

```
                                                    INPUTS[JB]].
                                                    OUTVALUE:=0;
                                                   AVAL:=0;
                                                   FASAR:=FASAR+1;
                                                   ASAR[FASAR]:=
                                                    BGUT*MAX3+
                                                    TEST[BGUT].
                                                    INPUTS[JB];
                                                END;
        FOR JJB:=1 TO TEST[TEST[BGUT].INPUTS[JB]].FANOUTNUM DO
         IF FLAGP=0 THEN
         IF(TEST[TEST[BGUT].INPUTS[JB]].FANOUTS[JJB]<>BGUT)THEN
           MAINFORWARD(TEST[BGUT].INPUTS[JB],TEST[TEST[BGUT].
                       INPUTS[JB]].FANOUTS[JJB]);
         IF FLAGP=0 THEN
           BACKWARD(TEST[BGUT].INPUTS[JB],AVAL);
                                          END;
                                  END;
                            END

                       ELSE (*BGUT IS 'OR' GATE OR 'NAND' GATE*)
                            BEGIN
                                 BCOUNT:=0;
                                 FOR JB:=1 TO TEST[BGUT].NOINPUTS DO
                                     BEGIN
                                          IF(TEST[BGUT].INVALUE[JB]<>-1)THEN
                                            BCOUNT:=BCOUNT+1;
                                     END;
                                 IF(TEST[BGUT].NOINPUTS-BCOUNT<>1)THEN
                                   BEGIN
                                        FADJUST:=FADJUST+1;
                                        ADJUST[FADJUST]:=BGUT;
                                        FASAR:=FASAR+1;
                                        ASAR[FASAR]:=BGUT;
                                   END
                                 ELSE
                                     BEGIN
                                          JB:=1;
                                          WHILE(TEST[BGUT].INVALUE[JB]<>-1)DO
                                               JB:=JB+1;
                                          IF(TEST[BGUT].GATETYPE=ORE)THEN
                                            BEGIN
                                                 TEST[BGUT].INVALUE[JB]:=1;
                                                 AVAL:=1;
                                                 TEST[TEST[BGUT].INPUTS[JB]].
                                                  OUTVALUE:=1;

                                                 FASAR:=FASAR+1;
                                                 ASAR[FASAR]:=BGUT*MAX3+
                                                  TEST[BGUT].INPUTS[JB];
                                            END
                                           ELSE
                                              BEGIN
                                                   TEST[BGUT].INVALUE[JB]:=0;
                                                   AVAL:=0;
                                                   TEST[TEST[BGUT].
                                                    INPUTS[JB]].OUTVALUE:=0;
                                                   FASAR:=FASAR+1;
                                                   ASAR[FASAR]:=BGUT*MAX3+
                                                    TEST[BGUT].INPUTS[JB];
                                              END;
```

```
                                        FOR JJB:=1 TO TEST[TEST[BGUT].
                                             INPUTS[JB]].FANOUTNUM DO
                                          IF FLAGP=0 THEN
                                          IF(TEST[TEST[BGUT].INPUTS[JB]].
                                               FANOUTS[JJB]<>BGUT)THEN
  MAINFORWARD(TEST[BGUT].INPUTS[JB],
        TEST[TEST[BGUT].INPUTS[JB]].FANOUTS[JJB]);
                                                  IF FLAGP=0 THEN
  BACKWARD(TEST[BGUT].INPUTS[JB],AVAL);
                                              END;
                        END;
              END;
        END;

     END;
  END;(*END OF BACKWARD*)


(*THE FUNCTION OF THE FOLLOWINGPROCEDURE IS AS FOLLOWS: SUPPOSE THAT
THE OUTPUT OF THE GATE Y=GUTC IS CONNECTED TO THE INPUT OF THE GATE
X=IUTC. THIS PROCEDURE PUTS PROPAGATION VALUES ON THE INPUTS OF X
OTHER THAN THE ONE CONNECTED TO Y, AND FINDS THE EFFECT OF THESE
VALUE ASSIGNMENTS FORWARD AND BACKWARD. IF NO CONFLICT IS FOUND THEN
VALUE ON THE OUTPUT OF X WILL BE DETERMINED AND GATE X WILL BE ADDED
TO  THE CRITICAL GATE ARRAY(CGATES). THE CORRESPONDING CRITICAL
VALUE WILL BE KEPT ON 'CVALUES' ARRAY. THE INPUT OF THE X CONNECTED
TO Y WILL  BE MARKED AS CRITICAL FOR THE VALUE ON THE OUTPUT OF
Y(CRITICAL VALUE) IN THE CORRESPONDING FIELD IN THE RECORD OF EACH
GATE.*)


  PROCEDURE CRITICALPATH(VAR GUTC,IUTC,CVALC:INTEGER);

    VAR JCP,KCP :INTEGER;

     BEGIN
   WRITELN('CRITICALPATH ENTERY');
          FLAGP:=0;
          IF(TEST[IUTC].OUTVALUE<>-1)THEN
        BEGIN
            FLAGP:=1;(*THIS PATH CANNOT BE SESITIZED.*)
       WRITELN('FLAGP=',FLAGP:1,'     ','TEST[',IUTC:1,'].OUTVALUE=',
                TEST[IUTC].OUTVALUE:1);
       END
          ELSE
              BEGIN
                   IF(TEST[IUTC].GATETYPE=INV)THEN
                     BEGIN
                          TEST[IUTC].INVALUE[1]:=CVALC;
                          TEST[IUTC].OUTVALUE:=1-CVALC;
                          TEST[IUTC].TEMPCV[1]:=CVALC;
                          FCGATES:=FCGATES+1;
                          CGATES[FCGATES]:=IUTC;
                          CVALUES[FCGATES]:=1-CVALC;
                          FASAR:=FASAR+1;
                          ASAR[FASAR]:=IUTC*MAX3+TEST[IUTC].INPUTS[1];
                          CRITVAL:=1-CVALC;
                     END
                   ELSE
                       BEGIN
                            JCP:=1;
```

```
                        WHILE((JCP<=TEST[IUTC].NOINPUTS)AND(FLAGP=0))DO
                             BEGIN
                                  IF(TEST[IUTC].INPUTS[JCP]<>GUTC)THEN
                                    BEGIN
                                         WRITELN('E1');
                                         IF((TEST[IUTC].GATETYPE=
                                             ANDE)OR(TEST[IUTC].
                                             GATETYPE=NAND))THEN
                                           BEGIN
                                             WRITELN('E2');
                                             IF(TEST[TEST[IUTC].
                                                INPUTS[JCP]].
                                                OUTVALUE=0)THEN
                                               FLAGP:=1
                                             ELSE
                                                 BEGIN
                                                     WRITELN('E3');
          IF(TEST[TEST[IUTC].INPUTS[JCP]].OUTVALUE=-1)THEN
                                                       BEGIN
                                                      WRITELN('E4');
             FASAR:=FASAR+1;
             ASAR[FASAR]:=-(IUTC*MAX3+TEST[IUTC].INPUTS[JCP]);
             TEST[IUTC].INVALUE[JCP]:=1;
             TEST[TEST[IUTC].INPUTS[JCP]].OUTVALUE:=1;
          FOR KCP:=1 TO TEST[TEST[IUTC].INPUTS[JCP]].FANOUTNUM DO
 IF(TEST[TEST[IUTC].INPUTS[JCP]].FANOUTS[KCP]<>IUTC)THEN
MAINFORWARD(TEST[IUTC].INPUTS[JCP],
          TEST[TEST[IUTC].INPUTS[JCP]].FANOUTS[KCP]);
           KCP:=1;
           IF FLAGP=0 THEN
             BACKWARD(TEST[IUTC].INPUTS[JCP],KCP);
                                                       END;
                                                    JCP:=JCP+1;
                                                 END;
                                           END
                                      ELSE(*IF(TEST[IUTC].GATETYPE=ORE
                                           OR NOR)THEN*)
                                          BEGIN
                                              IF(TEST[TEST[IUTC]
                                                 .INPUTS[JCP]].
                                                 OUTVALUE=1)THEN
                                                FLAGP:=1
                                              ELSE
                                                BEGIN
          IF(TEST[TEST[IUTC].INPUTS[JCP]].OUTVALUE=-1)THEN
                                                    BEGIN
   FASAR:=FASAR+1;
   ASAR[FASAR]:=-(IUTC*MAX3+TEST[IUTC].INPUTS[JCP]);
   TEST[IUTC].INVALUE[JCP]:=0;
   TEST[TEST[IUTC].INPUTS[JCP]].OUTVALUE:=0;
   FOR KCP:=1 TO TEST[TEST[IUTC].INPUTS[JCP]].FANOUTNUM DO
     IF(TEST[TEST[IUTC].INPUTS[JCP]].FANOUTS[KCP]<>IUTC)THEN
MAINFORWARD(TEST[IUTC].INPUTS[JCP],
          TEST[TEST[IUTC].INPUTS[JCP]].FANOUTS[KCP]);
     KCP:=0;
     IF FLAGP=0 THEN
       BACKWARD(TEST[IUTC].INPUTS[JCP],KCP);
                                                    END;
                                                 JCP:=JCP+1;
                                              END;
                                        END;
```

```
                                        END
                                      ELSE
                                        JCP:=JCP+1;
                                    END;
                                IF(FLAGP=0) THEN
                                  BEGIN
      IF((TEST[IUTC].GATETYPE=ANDE)OR(TEST[IUTC].GATETYPE=ORE))THEN
                                    BEGIN
                                        CRITVAL:=CVALC;
                                        TEST[IUTC].OUTVALUE:=CVALC;
                                        FCGATES:=FCGATES+1;
                                        CGATES[FCGATES]:=IUTC;
                                        CVALUES[FCGATES]:=CVALC;
                                     END;
      IF((TEST[IUTC].GATETYPE=NAND)OR(TEST[IUTC].GATETYPE=NOR))THEN
                                     BEGIN
                                         TEST[IUTC].OUTVALUE:=1-CVALC;
                                         CRITVAL:=1-CVALC;
                                         FCGATES:=FCGATES+1;
                                         CGATES[FCGATES]:=IUTC;
                                         CVALUES[FCGATES]:=1-CVALC;
                                     END;
                                  FASAR:=FASAR+1;
                                  ASAR[FASAR]:=IUTC*MAX3+GUTC;
                                  JCP:=1;
                                  WHILE(TEST[IUTC].INPUTS[JCP]<>GUTC)DO
                                       JCP:=JCP+1;
                                  TEST[IUTC].INVALUE[JCP]:=CVALC;
                                  TEST[IUTC].TEMPCV[JCP]:=CVALC;
                               END;
                  END;
             END;
    END;(*END OF CRITICALPATH*)


  PROCEDURE ERASE;

    VAR  CHECKFLAG,JE,M,N,Q,WLISTEMPTY:INTEGER;

    BEGIN
(*  STATUS;*)
        IF(FWLIST=0)THEN(*THIS OCCURES WHEN NO TEST EXIST FOR LAST
                          ENTRY OF WLIST.*)
          BEGIN
               WLISTEMPTY:=1;
               FWLIST:=1;
               ADAR[FWLIST]:=1;
               ADJADAR[FWLIST]:=0;
               CADAR[FWLIST]:=0;
          END;
        FOR JE:=ADAR[FWLIST] TO FASAR DO
            BEGIN
                 IF((ASAR[JE]<MAX3)AND(ASAR[JE]>0))THEN
                   BEGIN
                   TEST[ASAR[JE]].OUTVALUE:=-1
                   END
                 ELSE
                     BEGIN
                          CHECKFLAG:=1;
                          IF(ASAR[JE]<0)THEN
                          CHECKFLAG:=-1;
```

```
                              M:=TRUNC((CHECKFLAG*ASAR[JE])/MAX3);
                              N:=CHECKFLAG*ASAR[JE]-M*MAX3;
                              Q:=1;
                              WHILE(TEST[M].INPUTS[Q]<>N)DO
                                   Q:=Q+1;
                              TEST[M].INVALUE[Q]:=-1;
                              IF(CHECKFLAG=1)THEN
                                BEGIN
                                     TEST[M].TEMPCV[Q]:=-1;
                                     TEST[M].OUTVALUE:=-1;
                                END;
                         END;
                  END;
              FASAR:=ADAR[FWLIST]-1;
              FADJUST:=ADJADAR[FWLIST];
              FCGATES:=CADAR[FWLIST];
              IF WLISTEMPTY=1 THEN
                 FWLIST:=0;
              WLISTEMPTY:=0;
    END; (*END OF 'ERASE'*)


  PROCEDURE PRINTTEST;FORWARD;

  PROCEDURE ADJUSMENT;(*CALLED FROM PROCEDURE TESTGENERATION*)

   LABEL 100;
   VAR ADJFASAR,ADJFWLIST,ADJUSTVALUE,TEMPGATE,TEMPINPUT
       ,MAINADJVALUE,ADJGATE,JADJ:INTEGER;

   BEGIN
      (*  ADJFASAR:=FASAR;*)
        ADJFWLIST:=FWLIST;
        FLAGP:=0;
        WHILE((FADJUST>0)AND(FLAGP=0))DO (*WHILE NOT ALL GATES
                    WAITING FOR JUSTIFICATION ARE PROCESSED DO*)
             BEGIN
                  ADJGATE:=ADJUST[FADJUST];
                  FADJUST:=FADJUST-1;
                  IF((TEST[ADJGATE].GATETYPE=ANDE)OR
                    (TEST[ADJGATE].GATETYPE=NAND))THEN
                    ADJUSTVALUE:=0
                  ELSE
                      ADJUSTVALUE:=1;
                 MAINADJVALUE:=ADJUSTVALUE;
                 JADJ:=1;
                  WHILE((TEST[ADJGATE].INVALUE[JADJ]<>ADJUSTVALUE)AND
                        (JADJ<TEST[ADJGATE].NOINPUTS))DO
                       JADJ:=JADJ+1;
                  IF(TEST[ADJGATE].INVALUE[JADJ]<>ADJUSTVALUE)THEN
                    FOR JADJ:=1 TO TEST[ADJGATE].NOINPUTS DO
                         IF(TEST[ADJGATE].INVALUE[JADJ]=-1)THEN
                           BEGIN
                                FWLIST:=FWLIST+1;
                                WLIST[FWLIST]:=ADJGATE*MAX3+
                                 TEST[ADJGATE].INPUTS[JADJ];
                                ADAR[FWLIST]:=FASAR+1;
                                ADJADAR[FWLIST]:=FADJUST;(*FOR USE
                                                    OF 'ERASE' ONLY*)
                           END;
100:            IF(FWLIST>ADJFWLIST)THEN
```

```
                    BEGIN
                         FLAGP:=0;
                         TEMPGATE:=TRUNC(WLIST[FWLIST]/MAX3);
                         TEMPINPUT:=WLIST[FWLIST]-TEMPGATE*MAX3;
                         FWLIST:=FWLIST-1;
                         IF((TEST[TEMPGATE].GATETYPE=ANDE)OR
                            (TEST[TEMPGATE].GATETYPE=NAND))THEN
                           ADJUSTVALUE:=0
                         ELSE
                             ADJUSTVALUE:=1;
                         IF(TEST[TEMPINPUT].OUTVALUE=1-ADJUSTVALUE)THEN
(*THIS CHECK IS NECESSARY BECAUSE MAY BE IN ADJUSTMENT PROCESS SOME
OF THE VALUES ON INPUTS OF GATES FOR ADJUSTMENT ARE CHANGED TO
VALUES OTHER THAN DON'T CARES.*)
                           GOTO 100
                         ELSE
                             IF(TEST[TEMPINPUT].OUTVALUE=-1)THEN
                               BEGIN
                                    JADJ:=1;
                                    WHILE(TEST[TEMPGATE].INPUTS[JADJ]
                                          <>TEMPINPUT)DO
                                         JADJ:=JADJ+1;
                                    TEST[TEMPGATE].INVALUE[JADJ]:=ADJUSTVALUE;
                                    TEST[TEMPINPUT].OUTVALUE:=ADJUSTVALUE;
                                    FASAR:=FASAR+1;
                                    ASAR[FASAR]:=TEMPGATE*MAX3+TEMPINPUT;
                                    JADJ:=1;
                                    WHILE(JADJ<=TEST[TEMPINPUT].
                                          FANOUTNUM)AND(FLAGP=0) DO
                                        BEGIN
                MAINFORWARD(TEMPINPUT,TEST[TEMPINPUT].FANOUTS[JADJ]);
                                             JADJ:=JADJ+1;
                                        END;
                                    IF(FLAGP=0)THEN
                                      BACKWARD(TEMPINPUT,ADJUSTVALUE);
                                    IF(FLAGP=1)THEN
                                      IF(FWLIST>ADJFWLIST)THEN
                                        BEGIN
WRITELN(OUTFILE,'CONFLICT FOUND INJUSTIFICATION PROCESS BETWEEN GATES',
        TEMPINPUT:2,' ',TEMPGATE:2,' FOR VALUE ', ADJUSTVALUE:1,'
        IN JUSTIFYING ',ADJGATE:2,' FOR VALUE ',MAINADJVALUE:1);
                                             ERASE;
                                             GOTO 100;
                                        END
                                      ELSE
                    WRITELN(OUTFILE,'GATE ',ADJGATE:2,
                           ' WAS NOT JUSTIFIED FOR ',MAINADJVALUE:1,'.');
                                        END;
                    END;
               END;
          PRINTTEST;
          FWLIST:=ADJFWLIST;
  END;(*END OF ADJUSTMENT*)


PROCEDURE ADDONEFANOUTTOWLIST(VAR GUTAW,JAF:INTEGER);

      BEGIN
           FWLIST:=FWLIST+1;
```

```
            WLIST[FWLIST]:=(TEST[GUTAW].FANOUTS[JAF])*MAX3+GUTAW;
            ADAR[FWLIST]:=FASAR+1;
            ADJADAR[FWLIST]:=FADJUST;
            CADAR[FWLIST]:=FCGATES;
       END;


FUNCTION FINDFANOUTFORWLIST(VAR GATE:INTEGER):INTEGER;

   VAR  JA,KA    :INTEGER;

    BEGIN
         JA:=1;
         WHILE(JA<=TEST[GATE].FANOUTNUM)DO
              BEGIN
                   KA:=1;
                   WHILE(TEST[TEST[GATE].FANOUTS[JA]].INPUTS[KA]<>GATE)DO
                        KA:=KA+1;
                   IF((TEST[TEST[GATE].FANOUTS[JA]].CRITICAL[KA]=CRITVAL)
                       OR(TEST[TEST[GATE].FANOUTS[JA]].CRITICAL[KA]=2))THEN
                     BEGIN
                          IF(JA=TEST[GATE].FANOUTNUM)THEN
                            FINDFANOUTFORWLIST:=JA;
                          JA:=JA+1;
                     END
                   ELSE
                       BEGIN
                            FINDFANOUTFORWLIST:=JA;
                            JA:=TEST[GATE].FANOUTNUM+1;
                       END;
              END;
    END;



PROCEDURE  PUTFANOUTSINWLIST(VAR GUTP:INTEGER);

  VAR JP,KP:INTEGER;

    BEGIN
    WRITELN('ENTERED PUTFANOUTSINWLIST');
         JP:=1;
         IF((TEST[GUTP].GATETYPE=ANDE)OR(TEST[GUTP].GATETYPE=ORE)
             OR(TEST[GUTP].GATETYPE=INPUTE))THEN
           BEGIN
                WHILE(((TEST[GUTP].CRITICAL[JP]<>CRITVAL)AND
                       (TEST[GUTP].CRITICAL[JP]<>2))
                        AND(JP<TEST[GUTP].NOINPUTS))DO
                     JP:=JP+1;
                IF((TEST[GUTP].CRITICAL[JP]=CRITVAL)OR
                   (TEST[GUTP].CRITICAL[JP]=2)
                    OR((TEST[GUTP].FANOUTNUM-TEST[GUTP].NOINPUTS)<=0))THEN
                  BEGIN
                       KP:=FINDFANOUTFORWLIST(GUTP);
                WRITELN('GUTP=',GUTP:3,'    ','KP=',KP:3);
                       ADDONEFANOUTTOWLIST(GUTP,KP);
                  END
                ELSE
                    FOR JP:=1 TO (TEST[GUTP].FANOUTNUM-
                                  TEST[GUTP].NOINPUTS+1)DO
```

```
                        BEGIN
                             KP:=JP;
                             ADDONEFANOUTTOWLIST(GUTP,KP);
                        END;
            END
          ELSE(*TEST[GUTP].GATETYPE=NAND,NOR,INVERTER*)
              BEGIN
    (* WRITELN('CRITVAL=',CRITVAL:2,'     ','TEST[',GUTP:2,'].GATETYPE=',
             TEST[GUTP].GATETYPE);*)
                 WHILE((TEST[GUTP].CRITICAL[JP]<>
                        1-CRITVAL)AND(TEST[GUTP].CRITICAL[JP]<>2)
                        AND(JP<TEST[GUTP].NOINPUTS))DO
                       JP:=JP+1;
    (* WRITELN('JP=',JP:2);*)
                 IF((TEST[GUTP].CRITICAL[JP]=
                     1-CRITVAL)OR(TEST[GUTP].CRITICAL[JP]=2)
                     OR((TEST[GUTP].FANOUTNUM-
                         TEST[GUTP].NOINPUTS)<=0))THEN
                   BEGIN
                      KP:=FINDFANOUTFORWLIST(GUTP);
     (*WRITELN('KP=',KP:2);*)
                      ADDONEFANOUTTOWLIST(GUTP,KP);
                   END
                 ELSE
                     FOR JP:=1 TO (TEST[GUTP].FANOUTNUM+1-
                                   TEST[GUTP].NOINPUTS)DO
                        BEGIN
                           KP:=JP;
     (*WRITELN('KPALL=',KP:2);*)
                           ADDONEFANOUTTOWLIST(GUTP,KP);
                        END;
              END;
    END;


PROCEDURE MAKECRITICAL;

  VAR JM,KM:INTEGER;

  BEGIN
       FOR JM:=1 TO NOOFNODES DO
           FOR KM:=1 TO TEST[JM].NOINPUTS DO
               IF((TEST[JM].TEMPCV[KM]<>-1)
                  AND(TEST[JM].CRITICAL[KM]<>2))THEN
                 IF((TEST[JM].CRITICAL[KM]<>-1)AND
                    (TEST[JM].TEMPCV[KM]<>TEST[JM].CRITICAL[KM]))THEN
                   TEST[JM].CRITICAL[KM]:=2
                 ELSE
                     TEST[JM].CRITICAL[KM]:=TEST[JM].TEMPCV[KM];
  END;


PROCEDURE PRINTTEST;
  VAR JPR:INTEGER;

  BEGIN
       WRITELN(OUTFILE,'****************************************');
       IF(FLAGP=1)THEN
         WRITELN(OUTFILE,'CRITICAL PATH NOT COMPLETED FOR THE FOLLOWING GATES:');
```

```
      IF(FLAGP=0)THEN
        BEGIN
             PCOUNT:=PCOUNT+1;
             WRITELN(OUTFILE,'TEST NUMBER ',PCOUNT,':');
        END;
      WRITELN(OUTFILE,'CRITICAL-GATE NUMBER','      ','CRITICAL-GATE OUTPUT-VALUE')
      FOR JPR:=1 TO FCGATES DO
         WRITELN(OUTFILE,'       ',CGATES[JPR]:3,'                              ',CVALUES
      WRITELN(OUTFILE,'GATE NUM','      ','OUTPUT VALUE');
      FOR JPR:=1 TO NOOFNODES DO
         WRITELN(OUTFILE,JPR:3,'              ',TEST[JPR].OUTVALUE:1);
      IF FLAGP=0 THEN
         BEGIN
              WRITELN(OUTFILE,'TEST VECTOR:');
              WRITELN(OUTFILE,'INPUT NUMBER----------------VALUE');
              FOR JPR:=1 TO INPUTCOUNT DO
                  BEGIN
                       I1:=TRUNC(INPUTLIST[JPR]/MAX3);
                       WRITELN(OUTFILE,I1:2,'---------------->',TEST[I1].INVALUE[
                  END;
        END;
  END;(*END OF PRINTTEST*)


PROCEDURE TESTGENERATION;
 VAR JT,JTT:INTEGER;
  BEGIN
          FOR MCRITVAL:=0 TO 1 DO
          BEGIN
               IF(MCRITVAL=1)THEN
       BEGIN
                 FOR JT:=1 TO INPUTCOUNT DO
                   BEGIN
                        FWLIST:=FWLIST+1;
                        WLIST[FWLIST]:=INPUTLIST[JT];
                        ADAR[FWLIST]:=1;
                        CADAR[FWLIST]:=0;
                        ADJADAR[FWLIST]:=0;
                    END;
       END;

              WHILE(FWLIST>0)DO
                   BEGIN
                    GUT:=WLIST[FWLIST]-TRUNC(WLIST[FWLIST]/MAX3)*MAX3;
                    IUT:=TRUNC(WLIST[FWLIST]/MAX3);
                    IF(TEST[IUT].GATETYPE<>INPUTE)THEN
                      CRITVAL:=TEST[GUT].OUTVALUE;
                    IF(TEST[IUT].GATETYPE=INPUTE)THEN
                      BEGIN
                           CRITVAL:=MCRITVAL;
                           FWLIST:=FWLIST+1;
                           TEST[GUT].INVALUE[1]:=CRITVAL;
                           TEST[GUT].OUTVALUE:=CRITVAL;
                           TEST[GUT].TEMPCV[1]:=CRITVAL;
                           FCGATES:=FCGATES+1;
                           CGATES[FCGATES]:=GUT;
                           CVALUES[FCGATES]:=CRITVAL;
                           FASAR:=FASAR+1;
                           ASAR[FASAR]:=IUT*MAX3+IUT;
```

```
                    PUTFANOUTSINWLIST(IUT);(*ADD FANOUTS OF THIS INPU
                END
              ELSE
                  BEGIN
                    IF(TEST[IUT].GATETYPE=OUTPUTE)THEN
                      BEGIN
                          TEST[IUT].INVALUE[1]:=CRITVAL;
                          TEST[IUT].OUTVALUE:=CRITVAL;
                          TEST[IUT].TEMPCV[1]:=CRITVAL;
                          FCGATES:=FCGATES+1;
                          CGATES[FCGATES]:=IUT;(*ADD
                           THE GATE MADE CRITICAL TO
                           'CGATES' ARRAY.*)
                          CVALUES[FCGATES]:=CRITVAL;(*KEEP
                           THE CRITICAL VALUE OF THIS GATE
                           IN 'CVALUES' ARRAY.*)
                          FASAR:=FASAR+1;
                          ASAR[FASAR]:=IUT*MAX3+GUT;(*ADD
                           THIS OUTPUT TO 'ASAR' ARRAY.*)
                          ADJUSMENT;(*DO THE JUSTIFICATION*)
                          IF FLAGP=0 THEN
                            MAKECRITICAL;(*MARK ALL THE LINES
                             WHICH HAVE BEEN MADE CRITICAL
                             AND SEE IF THEY TESTED
                             COMPLETELY.*)
                          FWLIST:=FWLIST-1;
                          ERASE;
                          FLAGP:=0;
                      END
                    ELSE(*IT MEANS THAT ONE BRANCH OF A FANOUT
                          ORIGIN IS GOING TO BE TAKEN.*)
                        BEGIN
                          FWLIST:=FWLIST-1;
                          FOR JT:=1 TO TEST[GUT].FANOUTNUM DO
   (*FIND THE INPUT TO THE 'IUT' WHICH IS CONNECTED TO THE FANOUT ORIGIN AND
FOR THE OTHER BRANCH OF THAT FANOUT ORIGIN PROPAGATE THE VALUE FORWARD.*)
                            IF(TEST[GUT].FANOUTS[JT]
                              <>IUT)THEN
                              MAINFORWARD(GUT,TEST[GUT].
                                FANOUTS[JT]);
                          IF FLAGP=0 THEN
                            BEGIN
                              TEMPCRITVAL:=CRITVAL;
                          CRITICALPATH(GUT,IUT,TEMPCRITVAL);
                            END;
 IF(FLAGP=1)THEN (*A CONFLICT WAS FOUND IN THE PATH SENSITIZATION PROCESS.*)
                            BEGIN
                              PRINTTEST;
                              ERASE;
                              FLAGP:=0;
                            END
                          ELSE(*IF(FLAGP=0)THEN*)
                            PUTFANOUTSINWLIST(IUT);
                        END;
                  END;
              END;
          END;
    END;

(* PROCEDURE CPUTIMER;EXTERN;*)
```

```
(*MAIN PROGRAM*)

  BEGIN

      (* CPUTIMER;*)
       WRITELN('TYPE NAME OF THE INPUT FILE:');
       READ(NAME);
       OPEN(FILE_VARIABLE:=INFILE,FILE_NAME:=NAME,HISTORY:=OLD);
       RESET(INFILE);
       REWRITE(OUTFILE);
       INITIALIZE;
       TESTGENERATION;
      (* CPUTIMER;*)
    (*    FOR I1:=1 TO NOOFNODES DO
       BEGIN
       WRITE(TEST[I1].GATENUM:2,TEST[I1].GATETYPE:8,TEST[I1].NOINPUTS:2);
       FOR J1:=1 TO TEST[I1].NOINPUTS DO
       WRITE(TEST[I1].INPUTS[J1]:2,'  ',TEST[I1].CRITICAL[J1]:3,'  ',
             TEST[I1].TEMPCV[J1]:3,'  ','INVALUE[',J1:2,']='
            ,TEST[I1].INVALUE[J1]:2);
       WRITE('  ',TEST[I1].FANOUTNUM:2);
       FOR J1:=1 TO TEST[I1].FANOUTNUM DO
       WRITE('  ',TEST[I1].FANOUTS[J1]:2);
       WRITE('  ',TEST[I1].OUTVALUE:3);
       WRITELN;
       END;*)
       WRITELN(OUTFILE,'LIST OF NODES WHICH ARE NOT COMPLETELY TESTED:');
       WRITELN(OUTFILE,'            FROM GATE
             TO GATE                VALUE TESTED FOR');
       K1:=0;
       FOR I1:=1 TO NOOFNODES DO
           FOR J1:=1 TO TEST[I1].NOINPUTS DO
               BEGIN
                    IF(TEST[I1].CRITICAL[J1]<>2)THEN
                      BEGIN
                           WRITELN(OUTFILE,'   ',TEST[I1].INPUTS[J1]:2,
                                   '-------------- ',I1:2,
                                   '-----------------  ',
                                   TEST[I1].CRITICAL[J1]:2);
                           K1:=K1+1;
                      END;
               END;
      IF(K1=0)THEN
        WRITELN(OUTFILE,'CIRCUIT WAS COMPLETELY TESTED.');
  END.
```

APPENDIX B

LISTING OF THE PROGRAM WHICH CHANGES A COMBINATIONAL LOGIC CIRCUIT TO A PIFD LOGIC CIRCUIT

```
PROGRAM FINDLOOPS(INPUT,OUTPUT);

(*THE PURPOSE OF THIS PROGRAM IS TO IDENTIFY THE RECONVERGENT GATES
AND ADD THE BLOCKING GATES AND TEST INPUTS WHERE THEY ARE NEEDED.THE
INPUT FILE MUST HAVE THE FORMAT AS THOSE NEEDED FOR THE TEST GENERATION
PROGRAM.THE OUTPUT FILE WILL BE IN A FORMAT USABLE BY THE TEST GENERATION
PROGRAM.AT THE END OF THE OUTPUT FILE THERE WILL BE SOME INFORMATION ABOUT
THE ADDED BLOCKING GATES AND TEST INPUTS.THIS PROGRAM ALWAYS PROCESSES
FIRST THE BRANCHES OF FANOUT ORIGINS WHICH ARE FACED FIRST IN THE FORWARD
TRAVELING OF THE CIRCUIT.THIS GUARANTEES THAT THE RECONVERGENT GATES WILL
BE FOUND IN TIME PROPORTIONAL TO N2.ALSO IT GUARANTEES THAT THE GATES OR
LINES WHICH ARE PART OF LOOP CAN BE FOUND IN TIME PROPORTIONAL TO N2.*)
CONST  MAX1=20;
       MAX2=4000;


TYPE   GTYPE=(ANDE,ORE,NAND,NOR,INPUTE,OUTPUTE,INV);

(*THE RECORD WHICH KEEPS NECESSARY INFORMATION FOR EACH GATE.*)
       CIRCUITDES=
                RECORD
                     GATETYPE     :GTYPE;
                     GATENUM      :1..MAX2;
                     NOINPUTS     :1..MAX1;
                     FACED        :INTEGER;
                     INPUTS       :ARRAY[1..MAX1] OF INTEGER;
                     FACEDINPUT   :ARRAY[1..MAX1] OF INTEGER;
                     LOOP         :ARRAY[1..MAX1] OF INTEGER;
                     FANOUTNUM    :1..MAX1;
                     FANOUTS      :ARRAY[1..MAX1] OF INTEGER;
                     PROCESSED    :INTEGER;
                     LOOPS        :INTEGER;(*'1' IF THE GATE IS ON A LOOP*)
                     RECON        :INTEGER;(*'1' IF THE GATE IS A
                                            RECONVERGENT GATE*)
                     PERRECON     :INTEGER;(*KEEP THE NUMBER OF TIMES THAT
                                            A GATE HAS BEEN MARKED AS
                                            RECONVERGENT GATE*)
                     PERLOOPS     :INTEGER;(*KEEP THE NUMBER OF TIMES THAT
                                             A GATE HAS BEEN MARKED AS BEING
                                             ON A LOOP*)
                     PERLOOP      :ARRAY[1..MAX1] OF INTEGER;
                     MAKELOOP     :ARRAY[1..MAX1] OF INTEGER;
                                  (*'1' IF IT IS ONE OF THE BRANCHES OF
                                  A FANOUT ORIGIN WHICH CREATES A
                                  RECONVERGENT PATH*)
                END;


      RECONGATES=
                RECORD
                     RG   :INTEGER;
                     BG   :INTEGER;
                     FG   :INTEGER;
                END;


      FANBRANCH=
                RECORD
                     FAN       :INTEGER;
                     BRANCH    :INTEGER;
                END;
```

```
VAR   TEST          :ARRAY[1..MAX2] OF CIRCUITDES;
      RGATE         :ARRAY[1..MAX2] OF RECONGATES;
      FOUTLIST      :ARRAY[1..MAX2] OF INTEGER;
      FANOUT        :ARRAY[1..MAX2] OF FANBRANCH;
      NAME          :PACKED ARRAY[1..40] OF CHAR;
      INFILE        :TEXT;
      OUTFILE       :TEXT;
      I,J,K,L,M,N   :INTEGER;
      FANOUTP       :INTEGER; (*POINTER TO FANOUT ARRAY*).
      FOUTLISTP     :INTEGER; (*POINTER TO FOUTLIST ARRAY*)
      FRONTIERGATE  :INTEGER; (*THE HEAD GATE UNDER PROCESS ONA PATH*)
      BLOCKGATE     :INTEGER; (*THE GATE BEFORE FRONTIERGATE ON A PATH*)
      NOOFNODES     :INTEGER; (*NUMBER OF GATES IN THE CIRCUIT*)
      OLDNOOFNODES  :INTEGER; (*# OF NODES BEFORE ADDING ANY TEST INPUT
                               OR BLOCKING GATE*)
      FOGATE        :INTEGER; (*A FANOUT GATE*)
      BACKTRACE     :INTEGER; (*KEEPS THE GATE IN BACKTRACING*)
      RGATEP        :INTEGER; (*POINTER TO RGATE ARRAY*)
      P,B           :INTEGER;
      LFO           :INTEGER; (*LAST FANOUT ORIGIN UNDER PROCESS*)
      FORMERBT      :INTEGER; (*FORMER BACKTRACE GATE*)


PROCEDURE INITIALIZE;

(*INITIALIZE THE NECESSARY RECORDS,FILES,AND VARIABLES.*)
  BEGIN
       FOUTLISTP:=0;
       READ(INFILE,NAME);
       WRITELN(OUTFILE,NAME);
       READ(INFILE,NOOFNODES);
       FOR K:=1 TO NOOFNODES DO
         BEGIN
              READ(INFILE,I);
              TEST[I].GATENUM:=I;
              TEST[I].FACED:=0;
              TEST[I].PROCESSED:=0;
              TEST[I].LOOPS:=0;
              TEST[I].PERLOOPS:=0;
              TEST[I].RECON:=0;
              TEST[I].PERRECON:=0;
              READ(INFILE,TEST[I].GATETYPE,TEST[I].NOINPUTS);
              FOR J:=1 TO TEST[I].NOINPUTS DO
                BEGIN
                     READ(INFILE,TEST[I].INPUTS[J]);
                     TEST[I].LOOP[J]:=0;
                     TEST[I].PERLOOP[J]:=0;
                     TEST[I].FACEDINPUT[J]:=0;
                END;
              READ(INFILE,TEST[I].FANOUTNUM);
              FOR J:=1 TO TEST[I].FANOUTNUM DO
               BEGIN
                    READ(INFILE,TEST[I].FANOUTS[J]);
                    TEST[I].MAKELOOP[J]:=0;
               END;
         END;
       FOR K:=1 TO NOOFNODES DO
         IF TEST[K].GATETYPE=INPUTE THEN
```

```
            BEGIN
                 B:=K;
                 WHILE (TEST[B].FANOUTNUM=1) AND
                       (TEST[B].GATETYPE<>OUTPUTE) DO
                   B:=TEST[B].FANOUTS[1];
                 IF TEST[B].FANOUTNUM>1 THEN
                   BEGIN
                        FOUTLISTP:=FOUTLISTP+1;
                        FOUTLIST[FOUTLISTP]:=B;
                   END;
            END;
  END; (*INITIALIZE*)


PROCEDURE BACKTRACING;
(*IF A RECONVERGENT GATE IS FOUND, THEN TRAVEL BACKWARD AND MARK
ALL THE GATE AND LINES WHICH HAVE BEEN FACED IN THE LAST ATTEMPT
AS BEING ON A LOOP UNTIL A FANOUT ORIGIN, A RECONVERGENT GATE, OR
A PRIMARY INPUT IS FACED.*)
BEGIN
     FOR L:=1 TO TEST[FRONTIERGATE].NOINPUTS DO
        IF (TEST[FRONTIERGATE].FACEDINPUT[L]=1)
           AND  (TEST[FRONTIERGATE].LOOP[L]=0) THEN
           BEGIN
                J:=0;
                TEST[FRONTIERGATE].LOOP[L]:=1;
                BACKTRACE:=TEST[FRONTIERGATE].INPUTS[L];
                WHILE (TEST[BACKTRACE].RECON=0) AND
                      (BACKTRACE<>LFO) AND (BACKTRACE<>FOGATE) DO
                    BEGIN
                         J:=1;
                         M:=1;
                         WHILE NOT(TEST[BACKTRACE].FACEDINPUT[M]=1) DO
                           M:=M+1;
                         TEST[BACKTRACE].LOOP[M]:=1;
                         TEST[BACKTRACE].LOOPS:=1;
                         FORMERBT:=BACKTRACE;
                         BACKTRACE:=TEST[BACKTRACE].INPUTS[M];

                    END;
(*                 IF BACKTRACE=LFO THEN
                    BEGIN
                         IF J=0 THEN
                           FORMERBT:=FRONTIERGATE;
                         M:=1;
                         WHILE NOT(TEST[BACKTRACE].FANOUTS[M]=FORMERBT) DO
                           M:=M+1;
                         TEST[BACKTRACE].MAKELOOP[M]:=1;
                    END;*)
           END;
END; (*BACKTRACING*)


(*THE FOLLOWING PROCEDURE IS SUPPOSED TO IDENTIFY THE INPUTS TO THE BLOCKING
  GATES (THE GATES RIGHT BEFORE A RECONVERGENT GATE WHICH ARE ON A LOOP WITH
  THAT RECONVERGENT GATE) WHICH ARE ON A LOOP. ALSO IT IDENTIFIES ALL THE OTHER
  GATES AND GATES' INPUTS WHICH ARE ON A LOOP.*)

PROCEDURE LOOP;
```

```
LABEL 1000;

  BEGIN
       FOR I:=1 TO FOUTLISTP DO
         BEGIN
              FOGATE:=FOUTLIST[I];
              FANOUTP:=0;
              FOR M:=1 TO NOOFNODES DO
                BEGIN
                     FOR L:=1 TO TEST[M].NOINPUTS DO
                       BEGIN
                            TEST[M].PERLOOP[L]:=TEST[M].PERLOOP[L]+
                                                TEST[M].LOOP[L];
                            TEST[M].LOOP[L]:=0;
                            TEST[M].FACEDINPUT[L]:=0;
                       END;
                     TEST[M].PERLOOPS:=TEST[M].PERLOOPS+TEST[M].LOOPS;
                     TEST[M].LOOPS:=0;
                     TEST[M].FACED:=0;
                     TEST[M].PERRECON:=TEST[M].PERRECON+TEST[M].RECON;
                     TEST[M].RECON:=0;
                END;
              IF TEST[FOUTLIST[I]].PROCESSED=0 THEN
                BEGIN
                     TEST[FOUTLIST[I]].PROCESSED:=1;
                     FOR J:=1 TO TEST[FOGATE].FANOUTNUM DO
                        BEGIN
                             FANOUTP:=FANOUTP+1;
                             FANOUT[FANOUTP].BRANCH:=TEST[FOGATE].FANOUTS[J];
                             FANOUT[FANOUTP].FAN:=FOGATE;
                        END;
                     N:=0;
                     WHILE NOT(N=FANOUTP) DO
                        BEGIN
                             N:=N+1;
                             BLOCKGATE:=FANOUT[N].FAN;
                             LFO:=FANOUT[N].FAN;
                             TEST[FANOUT[N].FAN].PROCESSED:=1;
                             FRONTIERGATE:=FANOUT[N].BRANCH;
1000:                        IF TEST[FRONTIERGATE].FACED>1 THEN
                     (*THIS GATE HAS BEEN FACED AT LEAST TWO MORE TIMES WHEN PROCESSING
                       THE SAME FOGATE. THEN IT HAS BEEN ALREADY MARKED AS RECON. GATE*)
                                BEGIN
                     (*KEEP TRACK OF RECONVERGENT GATES AND THEIR CORRESPONDING BLOCKING
                       GATES. IF THE BLOCKING GATE HAS MOR THAN ONE FANOUT, THEN MAY BE
                       IT CANOT BE CONSIDERED AS A BLOCKING GATE ALTHOUGH IT MAY HAVE
                       THE GATE TYPE. THEN KEEP ITS GATE NUMBER NEGATED IN ORDER TO
                       REMEMBER THIS CASE.*)
                                     RGATEP:=RGATEP+1;
                                     RGATE[RGATEP].RG:=FRONTIERGATE;
                                     M:=1;
                                     WHILE (TEST[FRONTIERGATE].INPUTS[M]<>
                                           BLOCKGATE) OR
                                           (TEST[FRONTIERGATE].FACEDINPUT[M]=1)
                                           DO
                                       M:=M+1;
                                     TEST[FRONTIERGATE].FACEDINPUT[M]:=1;
                                     IF TEST[BLOCKGATE].FANOUTNUM=1 THEN
                                       RGATE[RGATEP].BG:=BLOCKGATE
                                     ELSE
```

```
                                        RGATE[RGATEP].BG:=-BLOCKGATE;
                                      TEST[FRONTIERGATE].RECON:=
                                        TEST[FRONTIERGATE].RECON+1;
                                      BACKTRACING;
                                  END;
                                IF TEST[FRONTIERGATE].FACED=1 THEN
   (*RG HAS BEEN FACED JUST ONE TIME BUT IT WAS NOT MARKED AS RG&BLOCKGATE*)
                               BEGIN
                                      M:=1;
                                      WHILE (TEST[FRONTIERGATE].INPUTS[M]<>
                                             BLOCKGATE) OR
                                            (TEST[FRONTIERGATE].FACEDINPUT[M]=1)
                                             DO
                                        M:=M+1;
                                      TEST[FRONTIERGATE].FACEDINPUT[M]:=1;
                                      FOR K:=1 TO TEST[FRONTIERGATE].NOINPUTS DO
                                       IF (TEST[TEST[FRONTIERGATE].INPUTS[K]]
                                           .FACED>0) OR
                                           (TEST[FRONTIERGATE].INPUTS[K]=FOGATE)
                                            OR
                                           (TEST[FRONTIERGATE].INPUTS[K]=LFO)
                                          THEN
                                         BEGIN
                                             RGATEP:=RGATEP+1;
                                             RGATE[RGATEP].RG:=FRONTIERGATE;
                                             RGATE[RGATEP].FG:=FOGATE;
                                             IF TEST[TEST[FRONTIERGATE].
                                              INPUTS[K]].FANOUTNUM=1 THEN
                                               RGATE[RGATEP].BG:=
                                                TEST[FRONTIERGATE].INPUTS[K]
                                             ELSE
                                               RGATE[RGATEP].BG:=
                                                -TEST[FRONTIERGATE].INPUTS[K];
                                               TEST[FRONTIERGATE].RECON:=1;
                                               TEST[FRONTIERGATE].LOOPS:=
                                                TEST[FRONTIERGATE].LOOPS+1;
                                               BACKTRACING;
                                          END;
                                       TEST[FRONTIERGATE].FACED:=
                                         TEST[FRONTIERGATE].FACED+1;
                                  END;
                                IF TEST[FRONTIERGATE].FACED=0 THEN
                                  BEGIN
                                       WHILE (TEST[FRONTIERGATE].FACED=0) DO
                                          BEGIN
                                               TEST[FRONTIERGATE].FACED:=1;
                                               M:=1;
                           WHILE (TEST[FRONTIERGATE].INPUTS[M]<>BLOCKGATE) OR
                         (TEST[FRONTIERGATE].FACEDINPUT[M]<>0) DO
BEGIN
     WRITELN('FR=',FRONTIERGATE:4,'  M=',M:2,'BLOCKGATE=',BLOCKGATE:4);
     WRITELN('TFIM=',TEST[FRONTIERGATE].INPUTS[M]);
                                  M:=M+1;
END;
                                          TEST[FRONTIERGATE].FACEDINPUT[M]:=1;
                                            IF TEST[FRONTIERGATE].GATETYPE
                                                <>OUTPUTE THEN
                                               BEGIN
                                                    BLOCKGATE:=FRONTIERGATE;
                                                    IF TEST[FRONTIERGATE].
                                                      FANOUTNUM>1 THEN
```

```
                                                    FOR L:=2 TO TEST
                                                     [FRONTIERGATE].FANOUTNUM DO
                                                     BEGIN
                                                          FANOUTP:=FANOUTP+1;
                              FANOUT[FANOUTP].BRANCH:=TEST[FRONTIERGATE].FANOUTS[L];
                              FANOUT[FANOUTP].FAN:=FRONTIERGATE;
                                                     END;
                                       FRONTIERGATE:=TEST[FRONTIERGATE].FANOUTS[1];
                                                   END;
                                              END;
                                    IF TEST[FRONTIERGATE].GATETYPE<>OUTPUTE THEN
                                              GOTO 1000;
                                      END;
                               END;

                        END;
                END;
END;



PROCEDURE ADDTESTINPUT;
(*IF THE GATES BEFORE RECONVERGENT GATES HAVE THE RIGHT TYPE FOR
BLOCKING GATES, THEN ONLY ADD TEST INPUTS TO THEM.*)
  BEGIN
       TEST[B].NOINPUTS:=TEST[B].NOINPUTS+1;
       NOOFNODES:=NOOFNODES+1;
       TEST[B].INPUTS[TEST[B].NOINPUTS]:=NOOFNODES;
       TEST[NOOFNODES].GATENUM:=NOOFNODES;
       TEST[NOOFNODES].GATETYPE:=INPUTE;
       TEST[NOOFNODES].NOINPUTS:=1;
       TEST[NOOFNODES].INPUTS[1]:=NOOFNODES;
       TEST[NOOFNODES].FANOUTNUM:=1;
       TEST[NOOFNODES].FANOUTS[1]:=B;
   END;


PROCEDURE ADDBLOCKINGGATE;
(*ADD BLOCKING GATES BEFORE RECONVERGENT GATES.*)

 BEGIN
      FOR J:=1 TO TEST[B].FANOUTNUM DO
        IF TEST[B].FANOUTS[J]=R THEN
          BEGIN
               NOOFNODES:=NOOFNODES+1;
               IF (TEST[R].GATETYPE=ORE) OR
                  (TEST[R].GATETYPE=NOR) THEN
                 TEST[NOOFNODES].GATETYPE:=ANDE
               ELSE
                 TEST[NOOFNODES].GATETYPE:=ORE;
               TEST[B].FANOUTS[J]:=NOOFNODES;
               TEST[NOOFNODES].GATENUM:=NOOFNODES;
               TEST[NOOFNODES].NOINPUTS:=2;
               TEST[NOOFNODES].INPUTS[1]:=B;
               TEST[NOOFNODES].INPUTS[2]:=NOOFNODES+1;
               TEST[NOOFNODES].FANOUTNUM:=1;
               TEST[NOOFNODES].FANOUTS[1]:=R;
               M:=1;
               WHILE TEST[R].INPUTS[M]<>B DO
                M:=M+1;
```

```
                TEST[R].INPUTS[M]:=NOOFNODES;
                NOOFNODES:=NOOFNODES+1;
                TEST[NOOFNODES].GATETYPE:=INPUTE;
                TEST[NOOFNODES].GATENUM:=NOOFNODES;
                TEST[NOOFNODES].NOINPUTS:=1;
                TEST[NOOFNODES].INPUTS[1]:=NOOFNODES;
                TEST[NOOFNODES].FANOUTNUM:=1;
                TEST[NOOFNODES].FANOUTS[1]:=NOOFNODES-1;
          END;
END;


PROCEDURE INSERTGATES;
(*INSERT BLOCKING GATES AND TEST INPUTS WHERE THEY ARE NEEDED.*)
BEGIN
     OLDNOOFNODES:=NOOFNODES;
     FOR I:=1 TO NOOFNODES DO
       TEST[I].PROCESSED:=0;
     WHILE RGATEP>0 DO
       IF(TEST[RGATE[RGATEP].RG].PROCESSED=1) AND
         (TEST[ABS(RGATE[RGATEP].BG)].PROCESSED=1) THEN
         RGATEP:=RGATEP-1
       ELSE
         BEGIN
              R:=RGATE[RGATEP].RG;
              B:=ABS(RGATE[RGATEP].BG);
              TEST[R].PROCESSED:=1;
              TEST[B].PROCESSED:=1;
              K:=0;
              FOR J:=1 TO TEST[B].FANOUTNUM DO
                IF TEST[B].FANOUTS[J]=R THEN
                  K:=K+1;
              N:=0;
              IF (((TEST[R].GATETYPE=ORE) OR (TEST[R].GATETYPE=NOR))
                    AND (TEST[B].GATETYPE=ANDE)) OR
                 (((TEST[R].GATETYPE=ANDE) OR (TEST[R].GATETYPE=NAND))
                    AND (TEST[B].GATETYPE=ORE)) THEN
                IF K=1 THEN
                 BEGIN
                      FOR M:=1 TO TEST[B].FANOUTNUM DO
                        IF (TEST[TEST[B].FANOUTS[M]].PERLOOPS>0) AND
                           (TEST[B].FANOUTS[M]<>R) THEN
                          FOR J:=1 TO TEST[TEST[B].FANOUTS[M]].NOINPUTS DO
                            IF (TEST[TEST[B].FANOUTS[M]].INPUTS[J]=B) AND
                               (TEST[TEST[B].FANOUTS[M]].PERLOOP[J]>0) THEN
                                N:=1;
                      IF N=0 THEN (*NO ADDITIONAL BLOCKING GATE IS NECESSARY.*)
                       BEGIN
                            L:=1;
                            WHILE (TEST[B].PERLOOP[L]>0) AND
                                  (L<=TEST[B].NOINPUTS) DO
                              L:=L+1;
                            IF L>TEST[B].NOINPUTS THEN
                (*ADD AN EXTRA INPUT TO THE BLOCKING GATE.*)
                              ADDTESTINPUT;
                       END;
                      IF N>0 THEN (*ADD BLOCKING GATE AND TEST INPUT*)
                        ADDBLOCKINGGATE;
                 END
                ELSE
                 ADDBLOCKINGGATE
```

```
            ELSE
              ADDBLOCKINGGATE;
            RGATEP:=RGATEP-1;
        END;
    END;


(**********************MAIN PROGRAM**********************)

BEGIN
     WRITELN('TYPE NAME OF THE INPUTFILE');
     READLN(NAME);
     OPEN(FILE_VARIABLE:=INFILE,FILE_NAME:=NAME,HISTORY:=OLD);
     RESET(INFILE);
     WRITELN('TYPE NAME OF THE OUTPUTFILE');
     READLN(NAME);
     OPEN(FILE_VARIABLE:=OUTFILE,FILE_NAME:=NAME,HISTORY:=NEW);
     REWRITE(OUTFILE);
     INITIALIZE;
     LOOP;
(*FOR I:=1 TO RGATEP DO
WRITELN(OUTFILE,'RGATE.RG=',RGATE[I].RG:3,'  RGATE.BG=',RGATE[I].BG:3);*)
     INSERTGATES;
     WRITELN(OUTFILE,NOOFNODES);
     FOR I:=1 TO NOOFNODES DO
       BEGIN
            WRITE(OUTFILE,TEST[I].GATENUM:5,TEST[I].GATETYPE,
                  TEST[I].NOINPUTS:2);
            FOR J:=1 TO TEST[I].NOINPUTS DO
            WRITE(OUTFILE,TEST[I].INPUTS[J]:5,'    ');
            WRITE(OUTFILE,TEST[I].FANOUTNUM:3);
            FOR J:=1 TO TEST[I].FANOUTNUM DO
            WRITE(OUTFILE,TEST[I].FANOUTS[J]:5,'    ');
            WRITELN(OUTFILE);
       END;
     WRITELN(OUTFILE,
      ALL THE GATES WITH THE GATE NUMBER GREATER THAN',OLDNOOFNODES);
     WRITELN(OUTFILE,
      ARE THE GATES ADDED TO THE ORIGINAL CIRCUIT. THEN THE INPUTS');
     WRITELN(OUTFILE,
      WITH THE GATE NUMBERS GREATER THAN THIS VALUE ARE TEST_INPUTS.');

END.
```

APPENDIX C

COMPUTER RESULTS FROM THE TEST GENERATIONS PROGRAM FOR TWO EXAMPLES

TES.DAT;3 16-JAN-1985 18:51 Page 1

```
SAMPLE
14
1    INPUTE    1    1    1    8
2    INPUTE    1    2    2    7    13
3    INPUTE    1    3    1    7
4    INPUTE    1    4    2    8    9
5    INPUTE    1    5    1    9
6    INPUTE    1    6    1    11
7    ANDE      2    2    3    1    8
8    ANDE      4    1    7    4    13    1    10
9    NAND      2    4    5    2    10    11
10   ORE       3    8    13   9    1     12
11   NOR       2    9    6    1    12
12   ANDE      2    10   11   1    14
13   INV       1    2    1    10
14   OUTPUTE   1    12   1    14
```

```
OUTFILE.DAT;3                    16-JAN-1985 18:17                Page 1

SAMPLE
**************************************
TEST NUMBER          1:
CRITICAL-GATE NUMBER      CRITICAL-GATE OUTPUT-VALUE
         6                          0
        11                          1
        12                          1
        14                          1
GATE NUM      OUTPUT VALUE
   1              -1
   2               0
   3              -1
   4               1
   5               1
   6               0
   7               0
   8               0
   9               0
  10               1
  11               1
  12               1
  13               1
  14               1
 TEST VECTOR:
 INPUT NUMBER----------------VALUE
  1---------------->-1
  2---------------> 0
  3---------------->-1
  4---------------> 1
  5---------------> 1
  6---------------> 0
 **************************************
 CRITICAL PATH NOT COMPLETED FOR THE FOLLOWING GATES:
 CRITICAL-GATE NUMBER      CRITICAL-GATE OUTPUT-VALUE
         5                          0
         9                          1
        10                          1
 GATE NUM      OUTPUT VALUE
   1              -1
   2               1
   3              -1
   4               1
   5               0
   6              -1
   7              -1
   8               0
   9               1
  10               1
  11               0
  12               0
  13               0
  14               0
 **************************************
 CRITICAL PATH NOT COMPLETED FOR THE FOLLOWING GATES:
 CRITICAL-GATE NUMBER      CRITICAL-GATE OUTPUT-VALUE
         4                          0
         9                          1
        10                          1
 GATE NUM      OUTPUT VALUE
   1              -1
   2               1
```

```
OUTFILE.DAT;3                                                    Page 2

  3              -1
  4               0
  5               1
  6              -1
  7              -1
  8               0
  9               1
 10               1
 11               0
 12               0
 13               0
 14               0
****************************************
CRITICAL PATH NOT COMPLETED FOR THE FOLLOWING GATES:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
         4                          0
GATE NUM     OUTPUT VALUE
  1              1
  2              1
  3              1
  4              0
  5             -1
  6             -1
  7              1
  8             -1
  9              1
 10              1
 11              0
 12              0
 13              0
 14              0
****************************************
CRITICAL PATH NOT COMPLETED FOR THE FOLLOWING GATES:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
         3                          0
         7                          0
GATE NUM     OUTPUT VALUE
  1              1
  2              1
  3              0
  4              1
  5             -1
  6             -1
  7              0
  8             -1
  9             -1
 10             -1
 11             -1
 12             -1
 13              0
 14             -1
****************************************
TEST NUMBER          2:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
         2                          0
        13                          1
        10                          1
        12                          1
        14                          1
GATE NUM     OUTPUT VALUE
  1             -1
```

```
  2              0
  3              -1
  4              1
  5              1
  6              0
  7              0
  8              0
  9              0
 10              1
 11              1
 12              1
 13              1
 14              1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1----------------->-1
 2----------------->  0
 3----------------->-1
 4----------------->  1
 5----------------->  1
 6----------------->  0
**************************************
CRITICAL PATH NOT COMPLETED FOR THE FOLLOWING GATES:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
        2                          0
        7                          0
        8                          0
GATE NUM      OUTPUT VALUE
  1              1
  2              0
  3              1
  4              1
  5              -1
  6              -1
  7              0
  8              0
  9              -1
 10              1
 11              -1
 12              -1
 13              1
 14              -1
**************************************
CRITICAL PATH NOT COMPLETED FOR THE FOLLOWING GATES:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
        1                          0
GATE NUM      OUTPUT VALUE
  1              0
  2              1
  3              1
  4              1
  5              -1
  6              -1
  7              1
  8              -1
  9              -1
 10              -1
 11              -1
 12              -1
 13              0
 14              -1
```

```
**************************************
TEST NUMBER          3:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
         6                        1
        11                        0
        12                        0
        14                        0
GATE NUM      OUTPUT VALUE
  1              -1
  2               0
  3              -1
  4               1
  5               1
  6               1
  7               0
  8               0
  9               0
 10               1
 11               0
 12               0
 13               1
 14               0
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1---------------->-1
 2----------------> 0
 3---------------->-1
 4----------------> 1
 5----------------> 1
 6----------------> 1
**************************************
TEST NUMBER          4:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
         5                        1
         9                        0
        10                        0
        12                        0
        14                        0
GATE NUM      OUTPUT VALUE
  1              -1
  2               1
  3              -1
  4               1
  5               1
  6               0
  7              -1
  8               0
  9               0
 10               0
 11               1
 12               0
 13               0
 14               0
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1---------------->-1
 2----------------> 1
 3---------------->-1
 4----------------> 1
 5----------------> 1
 6----------------> 0
```

```
        16                          0
         8                          0
        10                          0
        12                          0
        14                          0
GATE NUM      OUTPUT VALUE
  1              1
  2              1
  3             -1
  4              1
  5             -1
  6             -1
  7             -1
  8              0
  9             -1
 10              0
 11             -1
 12              0
 13              0
 14              0
 15              0
 16              0
 17             -1
 18              0
 19              0
 20              0
 21              0
 22              1
 23              1
 24              1
 25              1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1-----------------> 1
 2-----------------> 1
 3----------------->-1
 4-----------------> 1
 5----------------->-1
 6----------------->-1
15-----------------> 0
17----------------->-1
19-----------------> 0
21-----------------> 0
22-----------------> 1
24-----------------> 1
****************************************
TEST NUMBER          7:
CRITICAL-GATE NUMBER      CRITICAL-GATE OUTPUT-VALUE
         6                          0
        11                          1
        23                          1
        12                          1
        14                          1
GATE NUM      OUTPUT VALUE
  1             -1
  2             -1
  3             -1
  4              1
  5              1
  6              0
  7             -1
```

```
  8              -1
  9              0
 10              1
 11              1
 12              1
 13              -1
 14              1
 15              -1
 16              -1
 17              -1
 18              -1
 19              -1
 20              0
 21              1
 22              0
 23              1
 24              -1
 25              -1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1---------------->-1
 2---------------->-1
 3---------------->-1
 4---------------> 1
 5---------------> 1
 6---------------> 0
15---------------->-1
17---------------->-1
19---------------->-1
21---------------> 1
22---------------> 0
24---------------->-1
****************************************
TEST NUMBER          8:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
         5                        0
         9                        1
        20                        1
        10                        1
        12                        1
        14                        1
GATE NUM      OUTPUT VALUE
  1              -1
  2              1
  3              -1
  4              1
  5              0
  6              -1
  7              -1
  8              0
  9              1
 10              1
 11              0
 12              1
 13              0
 14              1
 15              0
 16              0
 17              0
 18              0
 19              1
```

```
 20              1
 21              0
 22              1
 23              1
 24              -1
 25              -1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1---------------->-1
 2---------------->  1
 3---------------->-1
 4---------------->  1
 5---------------->  0
 6---------------->-1
15---------------->  0
17---------------->  0
19---------------->  1
21---------------->  0
22---------------->  1
24---------------->-1
**************************************
TEST NUMBER          9:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
        4                        0
        9                        1
       11                        0
       23                        0
       12                        0
       14                        0
GATE NUM      OUTPUT VALUE
  1              -1
  2              -1
  3              -1
  4              0
  5              1
  6              0
  7              -1
  8              0
  9              1
 10              1
 11              0
 12              0
 13              -1
 14              0
 15              -1
 16              -1
 17              -1
 18              -1
 19              -1
 20              -1
 21              1
 22              0
 23              0
 24              -1
 25              -1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1---------------->-1
 2---------------->-1
 3---------------->-1
 4---------------->  0
```

```
 5-----------------> 1
 6-----------------> 0
15-----------------> -1
17-----------------> -1
19-----------------> -1
21-----------------> 1
22-----------------> 0
24-----------------> -1
****************************************
TEST NUMBER          10:
CRITICAL-GATE NUMBER      CRITICAL-GATE OUTPUT-VALUE
         4                          0
         8                          0
        10                          0
        12                          0
        14                          0
GATE NUM      OUTPUT VALUE
  1               1
  2               -1
  3               -1
  4               0
  5               -1
  6               -1
  7               -1
  8               0
  9               1
 10               0
 11               0
 12               0
 13               -1
 14               0
 15               1
 16               1
 17               0
 18               0
 19               0
 20               0
 21               0
 22               1
 23               1
 24               1
 25               1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1-----------------> 1
 2-----------------> -1
 3-----------------> -1
 4-----------------> 0
 5-----------------> -1
 6-----------------> -1
15-----------------> 1
17-----------------> 0
19-----------------> 0
21-----------------> 0
22-----------------> 1
24-----------------> 1
****************************************
TEST NUMBER          11:
CRITICAL-GATE NUMBER      CRITICAL-GATE OUTPUT-VALUE
         3                          0
         7                          0
```

```
        25                      0
         8                      0
        10                      0
        12                      0
        14                      0
GATE NUM      OUTPUT VALUE
  1             1
  2             1
  3             0
  4             1
  5            -1
  6            -1
  7             0
  8             0
  9            -1
 10             0
 11            -1
 12             0
 13             0
 14             0
 15             1
 16             1
 17             0
 18             0
 19             0
 20             0
 21             0
 22             1
 23             1
 24             0
 25             0
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1-----------------> 1
 2-----------------> 1
 3-----------------> 0
 4-----------------> 1
 5----------------->-1
 6----------------->-1
15-----------------> 1
17-----------------> 0
19-----------------> 0
21-----------------> 0
22-----------------> 1
24-----------------> 0
****************************************
TEST NUMBER          12:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
         2                      0
        13                      1
        16                      1
         8                      1
        10                      1
        12                      1
        14                      1
GATE NUM      OUTPUT VALUE
  1             1
  2             0
  3            -1
  4             1
  5            -1
```

```
  6              -1
  7              0
  8              1
  9              -1
 10              1
 11              -1
 12              1
 13              1
 14              1
 15              0
 16              1
 17              0
 18              0
 19              0
 20              0
 21              0
 22              1
 23              1
 24              1
 25              1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1---------------> 1
 2---------------> 0
 3---------------> -1
 4---------------> 1
 5---------------> -1
 6---------------> -1
15---------------> 0
17---------------> 0
19---------------> 0
21---------------> 0
22---------------> 1
24---------------> 1
**************************************
TEST NUMBER          13:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
         2                        0
         7                        0
        25                        0
         8                        0
        10                        0
        12                        0
        14                        0
GATE NUM     OUTPUT VALUE
  1              1
  2              0
  3              1
  4              1
  5              -1
  6              -1
  7              0
  8              0
  9              -1
 10              0
 11              -1
 12              0
 13              1
 14              0
 15              -1
 16              1
```

```
 17              0
 18              0
 19              0
 20              0
 21              0
 22              1
 23              1
 24              0
 25              0
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1----------------> 1
 2----------------> 0
 3----------------> 1
 4----------------> 1
 5----------------->-1
 6----------------->-1
15----------------->-1
17----------------> 0
19----------------> 0
21----------------> 0
22----------------> 1
24----------------> 0
****************************************
TEST NUMBER          14:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
         1                        0
         8                        0
        10                        0
        12                        0
        14                        0
GATE NUM      OUTPUT VALUE
  1               0
  2              -1
  3              -1
  4               1
  5              -1
  6              -1
  7              -1
  8               0
  9              -1
 10               0
 11              -1
 12               0
 13              -1
 14               0
 15               1
 16               1
 17               0
 18               0
 19               0
 20               0
 21               0
 22               1
 23               1
 24               1
 25               1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1----------------> 0
 2----------------->-1
```

```
 3----------------->-1
 4-----------------> 1
 5----------------->-1
 6----------------->-1
15-----------------> 1
17-----------------> 0
19-----------------> 0
21-----------------> 0
22-----------------> 1
24-----------------> 1
****************************************
TEST NUMBER          15:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
       24                          1
       25                          1
        8                          1
       10                          1
       12                          1
       14                          1
GATE NUM      OUTPUT VALUE
   1              1
   2             -1
   3              0
   4              1
   5             -1
   6             -1
   7              0
   8              1
   9             -1
  10              1
  11             -1
  12              1
  13             -1
  14              1
  15              1
  16              1
  17              0
  18              0
  19              0
  20              0
  21              0
  22              1
  23              1
  24              1
  25              1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1-----------------> 1
 2----------------->-1
 3-----------------> 0
 4-----------------> 1
 5----------------->-1
 6----------------->-1
15-----------------> 1
17-----------------> 0
19-----------------> 0
21-----------------> 0
22-----------------> 1
24-----------------> 1
****************************************
TEST NUMBER          16:
```

```
CRITICAL-GATE NUMBER      CRITICAL-GATE OUTPUT-VALUE
        22                          1
        23                          1
        12                          1
        14                          1
GATE NUM      OUTPUT VALUE
   1              -1
   2              -1
   3              -1
   4              -1
   5              -1
   6               1
   7              -1
   8              -1
   9              -1
  10               1
  11               0
  12               1
  13              -1
  14               1
  15              -1
  16              -1
  17              -1
  18              -1
  19              -1
  20              -1
  21               1
  22               1
  23               1
  24              -1
  25              -1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
  1----------------->-1
  2----------------->-1
  3----------------->-1
  4----------------->-1
  5----------------->-1
  6----------------->  1
 15----------------->-1
 17----------------->-1
 19----------------->-1
 21----------------->  1
 22----------------->  1
 24----------------->-1
**************************************
TEST NUMBER          17:
CRITICAL-GATE NUMBER      CRITICAL-GATE OUTPUT-VALUE
        21                          1
        10                          1
        12                          1
        14                          1
GATE NUM      OUTPUT VALUE
   1              -1
   2               1
   3              -1
   4              -1
   5              -1
   6              -1
   7              -1
   8               0
```

```
  9              -1
 10              1
 11              -1
 12              1
 13              0
 14              1
 15              0
 16              0
 17              0
 18              0
 19              0
 20              0
 21              1
 22              1
 23              1
 24              -1
 25              -1
TEST VECTOR:
INPUT NUMBER------------------VALUE
 1----------------->-1
 2----------------->  1
 3----------------->-1
 4----------------->-1
 5----------------->-1
 6----------------->-1
15----------------->  0
17----------------->  0
19----------------->  0
21----------------->  1
22----------------->  1
24----------------->-1
**************************************
TEST NUMBER          18:
CRITICAL-GATE NUMBER      CRITICAL-GATE OUTPUT-VALUE
        19                          1
        20                          1
        10                          1
        12                          1
        14                          1
GATE NUM      OUTPUT VALUE
  1               -1
  2               1
  3               -1
  4               -1
  5               0
  6               -1
  7               -1
  8               0
  9               1
 10               1
 11               0
 12               1
 13               0
 14               1
 15               0
 16               0
 17               0
 18               0
 19               1
 20               1
 21               0
```

```
 22              1
 23              1
 24             -1
 25             -1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1---------------->-1
 2---------------->  1
 3---------------->-1
 4---------------->-1
 5---------------->  0
 6---------------->-1
15---------------->  0
17---------------->  0
19---------------->  1
21---------------->  0
22---------------->  1
24---------------->-1
**************************************
TEST NUMBER          19:
CRITICAL-GATE NUMBER      CRITICAL-GATE OUTPUT-VALUE
       17                          1
       18                          1
       10                          1
       12                          1
       14                          1
GATE NUM      OUTPUT VALUE
   1              -1
   2              -1
   3              -1
   4              0
   5              -1
   6              -1
   7              -1
   8              0
   9              1
  10              1
  11              0
  12              1
  13              -1
  14              1
  15              1
  16              1
  17              1
  18              1
  19              0
  20              0
  21              0
  22              1
  23              1
  24              -1
  25              -1
 TEST VECTOR:
 INPUT NUMBER-----------------VALUE
  1---------------->-1
  2---------------->-1
  3---------------->-1
  4---------------->  0
  5---------------->-1
  6---------------->-1
 15---------------->  1
```

```
17-----------------> 1
19-----------------> 0
21-----------------> 0
22-----------------> 1
24-----------------> -1
****************************************
TEST NUMBER          20:
CRITICAL-GATE NUMBER      CRITICAL-GATE OUTPUT-VALUE
        15                          1
        16                          1
        18                          1
        10                          1
        12                          1
        14                          1
GATE NUM      OUTPUT VALUE
   1              -1
   2               1
   3              -1
   4               0
   5              -1
   6              -1
   7              -1
   8               0
   9               1
  10               1
  11               0
  12               1
  13               0
  14               1
  15               1
  16               1
  17               1
  18               1
  19               0
  20               0
  21               0
  22               1
  23               1
  24              -1
  25              -1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
  1----------------> -1
  2----------------> 1
  3----------------> -1
  4----------------> 0
  5----------------> -1
  6----------------> -1
 15----------------> 1
 17----------------> 1
 19----------------> 0
 21----------------> 0
 22----------------> 1
 24----------------> -1
****************************************
TEST NUMBER          21:
CRITICAL-GATE NUMBER      CRITICAL-GATE OUTPUT-VALUE
         6                          1
        11                          0
        23                          0
        12                          0
```

```
        14                                   0
GATE NUM      OUTPUT VALUE
  1                -1
  2                -1
  3                -1
  4                 1
  5                 1
  6                 1
  7                -1
  8                -1
  9                 0
 10                 1
 11                 0
 12                 0
 13                -1
 14                 0
 15                -1
 16                -1
 17                -1
 18                -1
 19                -1
 20                 0
 21                 1
 22                 0
 23                 0
 24                -1
 25                -1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1----------------->-1
 2----------------->-1
 3----------------->-1
 4-----------------> 1
 5-----------------> 1
 6-----------------> 1
15----------------->-1
17----------------->-1
19----------------->-1
21-----------------> 1
22-----------------> 0
24----------------->-1
***************************************
TEST NUMBER          22:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
        5                         1
        9                         0
       20                         0
       10                         0
       12                         0
       14                         0
GATE NUM      OUTPUT VALUE
  1                -1
  2                 1
  3                -1
  4                 1
  5                 1
  6                -1
  7                -1
  8                 0
  9                 0
 10                 0
```

```
 11              -1
 12              0
 13              0
 14              0
 15              0
 16              0
 17              0
 18              0
 19              1
 20              0
 21              0
 22              1
 23              1
 24              -1
 25              -1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1----------------->-1
 2----------------> 1
 3----------------->-1
 4----------------> 1
 5----------------> 1
 6----------------->-1
15----------------> 0
17----------------> 0
19----------------> 1
21----------------> 0
22----------------> 1
24----------------->-1
**********************************
TEST NUMBER         23:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
         4                        1
         9                        0
        11                        1
        23                        1
        12                        1
        14                        1
GATE NUM      OUTPUT VALUE
  1              -1
  2              -1
  3              -1
  4              1
  5              1
  6              0
  7              -1
  8              -1
  9              0
 10              1
 11              1
 12              1
 13              -1
 14              1
 15              -1
 16              -1
 17              -1
 18              -1
 19              -1
 20              0
 21              1
 22              0
```

```
 23              1
 24              -1
 25              -1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1---------------->-1
 2---------------->-1
 3---------------->-1
 4----------------> 1
 5----------------> 1
 6----------------> 0
15---------------->-1
17---------------->-1
19---------------->-1
21----------------> 1
22----------------> 0
24---------------->-1
**************************************
TEST NUMBER          24:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
         4                        1
         8                        1
        10                        1
        12                        1
        14                        1
GATE NUM      OUTPUT VALUE
  1              1
  2              -1
  3              -1
  4              1
  5              -1
  6              -1
  7              -1
  8              1
  9              -1
 10              1
 11              -1
 12              1
 13              -1
 14              1
 15              1
 16              1
 17              0
 18              0
 19              0
 20              0
 21              0
 22              1
 23              1
 24              1
 25              1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1----------------> 1
 2---------------->-1
 3---------------->-1
 4----------------> 1
 5---------------->-1
 6---------------->-1
15----------------> 1
17----------------> 0
```

```
19-----------------> 0
21-----------------> 0
22-----------------> 1
24-----------------> 1
****************************************
TEST NUMBER          25:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
         3                        1
         7                        1
        25                        1
         8                        1
        10                        1
        12                        1
        14                        1
GATE NUM      OUTPUT VALUE
  1               1
  2               1
  3               1
  4               1
  5              -1
  6              -1
  7               1
  8               1
  9              -1
 10               1
 11              -1
 12               1
 13               0
 14               1
 15               1
 16               1
 17               0
 18               0
 19               0
 20               0
 21               0
 22               1
 23               1
 24               0
 25               1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1-----------------> 1
 2-----------------> 1
 3-----------------> 1
 4-----------------> 1
 5----------------->-1
 6----------------->-1
15-----------------> 1
17-----------------> 0
19-----------------> 0
21-----------------> 0
22-----------------> 1
24-----------------> 0
****************************************
TEST NUMBER          26:
CRITICAL-GATE NUMBER     CRITICAL-GATE OUTPUT-VALUE
         2                        1
        13                        0
        16                        0
        18                        0
```

```
       10                          0
       12                          0
       14                          0
GATE NUM      OUTPUT VALUE
  1               -1
  2               1
  3               -1
  4               -1
  5               -1
  6               -1
  7               -1
  8               0
  9               -1
 10               0
 11               -1
 12               0
 13               0
 14               0
 15               0
 16               0
 17               1
 18               0
 19               0
 20               0
 21               0
 22               1
 23               1
 24               -1
 25               -1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1----------------->-1
 3----------------->-1
 4----------------->-1
 5----------------->-1
 6----------------->-1
15----------------> 0
17----------------> 1
19----------------> 0
21----------------> 0
22----------------> 1
24----------------->-1
****************************************
TEST NUMBER          27:
CRITICAL-GATE NUMBER      CRITICAL-GATE OUTPUT-VALUE
        2                          1
        7                          1
       25                          1
        8                          1
       10                          1
       12                          1
       14                          1
GATE NUM      OUTPUT VALUE
  1               1
  2               1
  3               1
  4               1
  5               -1
  6               -1
  7               1
```

```
  8              1
  9             -1
 10              1
 11             -1
 12              1
 13              0
 14              1
 15              1
 16              1
 17              0
 18              0
 19              0
 20              0
 21              0
 22              1
 23              1
 24              0
 25              1
TEST VECTOR:
INPUT NUMBER-----------------VALUE
 1----------------> 1
 2----------------> 1
 3----------------> 1
 4----------------> 1
 5----------------> -1
 6----------------> -1
15----------------> 1
17----------------> 0
19----------------> 0
21----------------> 0
22----------------> 1
24----------------> 0
**************************************
TEST NUMBER          28:
CRITICAL-GATE NUMBER      CRITICAL-GATE OUTPUT-VALUE
          1                          1
          8                          1
         10                          1
         12                          1
         14                          1
GATE NUM      OUTPUT VALUE
  1               1
  2              -1
  3              -1
  4               1
  5              -1
  6              -1
  7              -1
  8               1
  9              -1
 10               1
 11              -1
 12               1
 13              -1
 14               1
 15               1
 16               1
 17               0
 18               0
 19               0
 20               0
```

2

VITA

Bijan Karimi

Candidate for the Degree of

Doctor of Philosophy

Thesis: COMBINATIONAL CIRCUITS FOR WHICH TESTS CAN BE GENERATED IN N^2 TIME

Major Field: Engineering

Biographical:

Personal Data: Born in Tehran, Iran, December 20, 1952, the son of Mr. and Mrs. A. Karimi.

Education: Graduated from Aryamehr University of Technology, Tehran Iran, in January 1977; received Master of Science degree in Electrical Engineering from Oklahoma State University in May 1981; Completed requirements for the Doctor of Philosophy degree at Oklahoma State University in December 1985.

Professional Experience: Electronics Instructor, Academy of Army, Tehran, Iran; Graduate Teaching and Research Assistant; Department of Electrical and Computer Engineering, Oklahoma State University, 1980-1985; Design Engineer, Texas Analytical Control Inc., 1985.