The E-Algorithm:
An Automatic Test Generation Algorithm
for Hardware Description Languages

by

Forrest Eugene Norrod

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

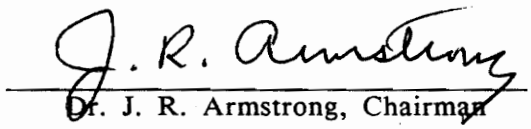
in partial fulfillment of the requirements for the degree of

Master of Science

in

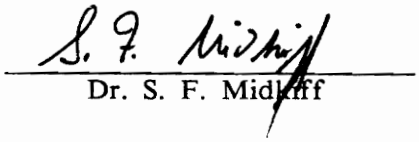
Electrical Engineering

APPROVED:



Dr. J. R. Armstrong, Chairman

Dr. F. G. Gray



Dr. S. F. Midkiff

February, 1988

Blacksburg, Virginia

The E-Algorithm:
An Automatic Test Generation Algorithm
for Hardware Description Languages

by

Forrest Eugene Norrod

Dr. J. R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

Traditional test generation techniques for digital circuits have been rendered inadequate by the increasing levels of integration achieved by VLSI technology. This thesis presents a test generation algorithm, the *E-algorithm*, that generates tests for circuits described using the VHDL Hardware Description Language. A fault model has been developed that addresses data path faults, faults in control structures, and faults in functional operators. The E-algorithm is able to generate tests for all modeled fault types, and handles a wide variety of circuit types, including sequential circuits. The algorithm has been implemented; preliminary results are given.

Acknowledgements

I would like to thank Dr. James Armstrong for the challenging environment he has made available, and the support and guidance he lent this research. Thanks also go to Dr. Gray and Dr. Midkiff for serving on my committee. To Dr. Miller, my gratitude for serving for a time on my committee and for many interesting discussions on artificial intelligence and politics. Erann Gat and Ledlie Klosky also helped keep life interesting with endless debates on innumerable topics.

Finally, my thanks to Carol Smith, who lent invaluable support and encouragement over the past several years.

Table of Contents

Chapter 1
Introduction

VLSI technology has ushered in a new era in integrated circuit sophistication and capabilities. The ability to produce integrated circuits with hundreds of thousands of components has brought unparalleled capabilities to single chips, opening the door to a host of new applications and vastly improved performance to cost ratios. However, the problems inherent in designing and testing these circuits grow ever more intractable as the level of integration increases. At the VLSI level, circuit design involves handling a large amount of detailed structural information. Hardware Description Languages (HDL's) have been introduced as part of a hierarchical design process that eases the amount of detail the designer must consider. HDL's provide a convenient method for specifying intended circuit function at a high level of abstraction, without burdening the designer with the structural details of the circuit implementation.

Circuit designers may use HDL circuit models to specify the circuit function and simulate the actual circuit operation to verify their designs. Automated circuit synthesis tools such as silicon compilers have been developed to transform HDL models to gate-level implementations, relieving the designer of this burden. The major advantage of HDL's is that they allow the designer to consider only the minimum amount of detail which his or her task at hand demands.

While HDL's provide a useful basis for high-level design tools, high-level methods to model faults or generate tests have not been well developed. The traditional gate-level fault models and test generation algorithms are inadequate when confronted by the staggering number of gate-level components in VLSI circuitry. As the computational costs of gate-level test generation have grown excessive, designers and test engineers have turned to other testing methods. Often, these techniques are somewhat ad-hoc such as: write test vectors to "toggle every circuit node." These approaches require a large investment in engineering time and yet fail to guarantee adequate fault coverage, particularly for complex sequential circuits. A popular alternative is to add logic that makes the circuit easier to test; this method is particularly useful for circuits with a number of state machine or other hard-to-test structures.

In recent attempts to devise systematic Automatic Test Pattern Generation (ATPG) systems, functional approaches to circuit description and test generation have been examined. In these functional approaches, the circuit input to the ATPG system is described at the functional or behavioral level by a HDL model rather than at the structural level. This thesis develops an algorithm, the *E-algorithm*, that uses HDL circuit descriptions to develop tests for circuits. Using HDL circuit models greatly decreases the volume of detail that is considered during test generation. The application of a graph transformation to the circuit eases the test generation process by allowing propagation paths through the circuit to be quickly found. The graph model further provides the basis for a complete fault model for the HDL descriptions. The E-algorithm has been implemented and results for fifteen test circuits are given.

1.1 Contents

This thesis describes an algorithm for generating tests for digital circuits described using the VHSIC Hardware Description Language (VHDL). The details of the algorithm are presented as are the allowed modeling constructs, the fault model used, current implementation and results, and an analysis of system limitations and suggestions for future work.

Chapter 2, "Literature review", outlines previous approaches to automatic test generation, considering early gate-level methods as well as recent developments in functional and behavioral paradigms.

Chapter 3, "VHDL Circuit Models", specifies the subset of VHDL used to model circuit behavior and discusses the circuit timing model.

Chapter 4, "Graph Transformation & Fault Model", explains the process of converting VHDL models to a data flow graph representation. A fault model based on this graph structure is proposed.

Chapter 5, "The E-Algorithm", describes the E-algorithm in detail, giving examples of the algorithm and discussing special cases and heuristics which improve efficiency.

Chapter 6, "Results", describes the current implementation and reviews the results of applying the E-algorithm to a number of circuit models.

Chapter 7, "Analysis & Suggestions", discusses the limitations of the current E-algorithm implementation and recommends a number of improvements.

Chapter 8, "Conclusions", draws conclusions concerning the E-algorithm and implementation.

Appendix A, "User's Manual", details the operation of the current E-algorithm implementation.

Appendix B, "Circuit Models and Fault Lists", gives the VHDL models, internal representations, and fault lists (with results) for all test circuits.

Chapter 2

Literature Review

2.1 Digital Circuit Faults

Random defects often occur during fabrication of integrated circuits. These defects can cause *faults* in the circuit which cause the circuit to behave in a manner that its designers had not intended. If such a fault occurs, the circuit cannot be used. *Test programs* are employed to detect circuit faults. A test program consists of a set of data patterns to be applied to the inputs of the circuit, and a set of values which represent the expected patterns at the circuit outputs. Generating a test program for a circuit ideally consists of finding a minimum length set of values such that, when they are applied to the circuit inputs, the outputs of a circuit containing any fault will differ from those of the fault-free circuit. An *ATPG* (Automatic Test Program Generation) system is a computer program that automatically generates tests for circuit faults; usually considering every plausible fault.

The list of faults that may occur in the circuit is derived from the *fault model*. The fault model specifies what potential effects manufacturing defects can have upon the circuit. Test generation systems usually specify a particular fault model and attempt to generate tests for all faults given by the model. It should be

noted that a given fault model does not usually model all possible perturbations caused by manufacturing defects, but merely a reasonable subset so that if the circuit is free of the modeled faults, it will probably work as designed.

2.2 Gate Level Test Generation

Early approaches to test generation were based on gate-level circuit descriptions; the circuit structure was explicitly given as a set of primitive interconnected logic gates, AND, OR, NAND, XOR, etc. A simple fault model based on "stuck-at" faults was used. The stuck-at fault model characterizes faults as producing permanent logic values on interconnection lines; for example, a particular line is "stuck-at-0" (or s-a-0) if the line delivers a constant 0 logic value to connected lines or components. The first systematic way to generate a complete set of tests for gate-level stuck-at faults was the D-algorithm proposed by Roth [1].

The D-algorithm employs a "D-calculus" to move the effects of a fault through the circuit, specifying good and faulty values for any line in which they differed. The symbolic variable "D" was introduced to indicate a value of '1' in the good circuit and '0' in the circuit containing the fault. The D-algorithm uses the stuck-at fault model but can also handle arbitrary faults in gate logic. Beginning at the site of the modeled fault, the effects of the fault (a D or its complement $\overline{D}$) are propagated toward the outputs of the circuit until the effect of the fault can be observed at the outputs. Any conditions required on the propagation path are then satisfied by a consistency check which establishes the input conditions to generate the test. Fault dominance and equivalence concepts were introduced to reduce the number of faults for which the D-algorithm was

applied.

Further gate-level methods have been introduced to speed the test generation process. Notable among these are PODEM [2], which has proven more efficient than the D-algorithm for certain types of circuits, and 9-V [3], which reduces the amount of backtracking necessary in reconvergent fanout circuits.

The problems with the gate-level approaches are [4,5]:

- They are too slow, their cost grows exponentially with circuit size, and, at the gate-level, VLSI circuits are too complex to use this approach.

- The stuck-at fault model was developed to model connection faults in SSI and MSI systems and is an inadequate model for some faults common in newer technologies such as CMOS.

- The entire gate-level circuit is required to generate tests. This is often unavailable if the component is from a vendor. Moreover, during the design cycle, attempting test generation for the circuit can point to hard to test segments where extra test logic may be desirable. Gate-level descriptions are not available until the end of the design cycle, at which point it may be difficult to insert test logic.

Some engineers have abandoned gate-level ATPG and have resorted to ad-hoc approaches to hand-generate tests. These methods are usually in the form of general guidelines such as: "exercise each function" or "toggle each circuit node". These methods typically require man-weeks of valuable engineering time and yet

give no guarantee of reasonable fault coverage.

A few companies have instituted strict guidelines on design methodology to force engineers to design circuits that are easy to test and for which modified gate-level ATPG algorithms may be applied [6,7]. Engineers in general have been reluctant to fully embrace such design-for-testability rules as they are unfamiliar, require valuable chip real estate and I/O pins, and may reduce performance.

Another factor in the declining attractiveness of gate-level ATPG techniques is the increasing use of functional level design and Hardware Description Languages (**HDL's**). Apart from making test generation more difficult, the increasing number of gates in VLSI circuits is causing problems with the traditional gate-level design techniques. Because of the enormous number of gates in current designs, it is extremely difficult to insure design correctness [4,5]. HDL's and functional design techniques are methods of describing a circuit at a higher level of abstraction than the gate-level approach. This higher abstraction reduces the amount of circuit detail the designer must consider and increases confidence in design correctness.

Functional-level design techniques approach circuit design using primitive elements such as adders, shift registers, and multiplexors. Standard-cell designs are examples of functional-level design. HDL's such as **GSP2** [8] and **VHDL** [9] typically resemble computer languages, specifying the behavior of the circuit being designed with little regard to the structure of the implementation. Use of HDL's and functional-level design methods can greatly reduce the length of the design cycle.

With the growing emphasis on these new design methodologies and the inadequacy of gate-level ATPG's, test generation techniques appropriate to the new level of abstraction are required.

2.2 Functional and Behavioral Test Generation

In general, functional and behavioral test generation resemble gate-level methods as they all involve faulting a portion of the circuit and propagating the effects of the fault to an observable output. In functional test generation, however, the faults to be detected are general faults in the functions of components of the circuit. These functional faults often correspond to one or more gate-level stuck-at faults, but they may also be uncharacterizable in the stuck-at model [4,5]. This lack of total equivalence to the gate-level fault coverage is not necessarily a drawback, as functional faults may detect circuit defects that the stuck-at gate-level model can not detect. Examples of faults undetectable to the stuck-at model are certain CMOS bridging and stuck-on faults [10,11].

2.2.1 Structure-Specific Approaches

A number of functional-level test generation methods have been developed for specific types of circuits. For testing microprocessors, several notable methods have been tried. Robach and Saucier [12] developed the Abstract Execution Graph approach, where each instruction was broken into micro-operations and each micro-operation in each instruction tested. Thatte and Abrahams [13] proposed another graph-oriented method. Hunger and Gaertner [14] suggested an approach where microprocessors are tested by a executing an ordered sequence of instructions and monitoring the results.

RAM memories are another type of system for which a number of functional testing approaches have been employed. Abadir and Reghbati discuss the prominent techniques [16].

2.2.2 General Functional Testing Techniques

Several methods have been proposed for general functional test generation, most based on extensions to the D-algorithm that include rules for propagating faults through functional blocks. These techniques are usually based on circuit descriptions resembling gate-level descriptions, but where some primitive elements are complex functional blocks.

Breuer and Friedman [15] propose structures to be used in a functional-level ATPG that are based on extensions to the D-algorithm. They discuss a description language, FSL, that allows circuits to be designed with functional primitives of arbitrary complexity. Each functional primitive is described by a (hopefully) simple algorithm that includes information on how to propagate faults throught the block by manipulating any control inputs to the functional block. Information to aid fault justification is also explicitly specified. Test generation is via a modified D-algorithm where propagation through functional blocks is accomplished by referencing the given propagation information for each functional block. The FSL language includes a syntax to specify the control conditions for each block. This approach doesn't consider how the propagation algorithms may be automatically generated and tests cannot be generated for all types of control faults.

Abadir and Reghbati [16,17] extended Breuer and Friedman's work by introducing binary decision diagrams. Their work demonstrates how faults can be propagated through functional blocks without hand-compiling propagation algorithms for each functional block. The problems raised by propagating fault syndromes through control structures, however, remain.

2.2.3 Register Transfer Level Test Generation

Approaches based on Register Transfer Languages or *RTL's* have been proposed by a number of researchers. RTL's are hardware description languages that describe a circuit in terms of transfers between registers and operations on data stored in registers. Register Transfer level methods model circuits at a slightly higher level than the more basic extended D-algorithm methods. Sequential components are invariably modeled at the functional level, combinational logic may be lumped into functional blocks or left at the gate level. Typically, data transfer is between registers or sequential functional blocks and is done in vector-wide chunks. The register transfer methods may still imply a particular circuit structure, but they are usually implementation independent. Models at this level and above are useful in generating tests for circuits where only specifications on data operations are known. An example is generating tests for a VLSI chip where more stringent tests than those applied by the vendor are desired; in this case, of course, typically no structural information is known about the circuit [4,5].

Shteingart, et al. [18] demonstrated the *RTG* or Register Transfer Level test generator. RTG is aimed at generating tests for circuit boards but is also applicable to LSI and VLSI circuits. The RTG system uses a hybrid model in which all combinational logic is represented at the gate level, i.e. explicit structural information must be given for combinational logic, but sequential elements such as shift registers and counters are modeled at the register transfer level.

Each sequential element in the RTG system has three tables that describe its function, and input/output connectivity. A modified version of the 9-V algorithm is applied to propagate faults through the combinational parts of the circuit, and the table information is used to propagate faults and sensitize paths through sequential blocks. Shteingart also discusses design techniques to ease test

generation. Joobani describes a similar algorithm [19].

Another of the RTL based test generation systems, *SCIRTSS* [20], was developed by Huey and Hill to work with a RTL known as *AHPL*. The SCIRTSS system applies a modified D-algorithm to the AHPL description. For each fault in the fault model of the circuit, the D-algorithm is applied to propagate the effect of the fault to the output. SCIRTSS treats the circuit as a combinational one by only considering one time period during the D propagation. When values are specified at the inputs and registers to sensitize the propagation path, heuristics are applied to try to find a sequence of inputs to put the circuit in the desired state for the test. This approach relies on the fact that the data and control paths are completely distinct and suffers from two severe limitations. First, the heuristics aren't guaranteed to find a valid sensitizing sequence; and second, the method can't develop tests for faults in the control area of the circuit.

Su and others [4,21,22,23,24] have proposed two approaches to test generation at the RTL level. The first, the Data Graph Generation approach [23], transforms the RTL circuit description into a data path graph and applies a modified D-algorithm to generate tests. This approach is roughly equivalent to the SCIRTSS method and suffers from an inability to generate tests for all possible control faults. An adjunct technique to test control faults was developed by Liaw, Su, and Malaiya and is based on modeling state machine faults by invalid transitions in state graphs [21].

Su, Lin, and Hsieh further proposed the symbolic execution approach to functional test generation [22,23,24]. The symbolic execution method, or *S-algorithm*, involves injecting a fault into the RTL model and then simulating the model, using symbolic values instead of actual values. After simulation, the constraints generated by the faulty model simulation are compared to the constraints from the fault-free model simulation and symbolic inequalities are

solved. If the constraints are compatible, a test can be specified for the given fault. The order of fault consideration is important in the S-algorithm; the data paths should be tested before control faults so potential data path faults will not prevent detection of control faults. The computational costs of the S-algorithm are large and the types of circuit control faults which can be modeled are restricted.

Kawai et al. [25] propose a heuristic ATPG system using the FDL Functional description language, a sophisticated RTL. FDL descriptions of circuits incorporate some detail of possible circuit implementations; each FDL statement corresponds to a primitive circuit element, but may be fairly complex. Using FDL circuit descriptions, a test generation algorithm, the P-algorithm, was developed to generate tests for combinational circuits or circuits with a scan path, where all the internal registers form a shift register that can be loaded or read.

The P-algorithm lacks an explicit fault model, relying instead on differences in behavior of fault-free and faulty circuits. The FDL circuit description is translated to a graphic form; each node in the graph representing a FDL operator, and the branches between nodes representing data or control lines between the nodes.

The P-algorithm picks an observable node (output) and sensitizes a path from the observable node to a controllable node (input) by propagating a "P" value backwards through the circuit. Test patterns are then generated which justify the sensitized path. To completely test the circuit, each path through the graph must be stimulated at least once. The process of propagating a P backwards through the network is similar to D propagation, including a backtracking search method to find consistent values with which to sensitize the selected path. Since the P-algorithm exercises each possible path through the graph, often much of the work done in sensitizing previous paths can be applied to new paths. At the

inputs, heuristics are applied to generate the values for P's to provide maximal fault coverage on the data paths.

While the P-algorithm has achieved fairly high fault coverages for some test circuits, its utility is limited by the requirement that circuits be combinational or utilize scan-path design rules. Furthermore, the lack of an explicit fault model makes fault coverage estimates for the P-algorithm difficult to compute.

2.2.4 Behavioral and Chip-Level Methods

The FDL system described above approaches the sophistication of behavioral HDL's, but still incorporates significant elements of the circuit structure. More powerful HDL's allow circuit descriptions at higher levels of abstraction, incorporating no structural information. Alternatively, most HDL's make provisions for specifying circuit implementation details if desired. Any ATPG system that uses HDL circuit descriptions must be able to generate tests without relying on structural information, but should be able to take advantage of circuit structure if available.

Perhaps the most difficult task in developing a HDL-based ATPG is the specification of the fault model. The fault model employed will determine the worth of the ATPG system; if potential faults are not included in the fault model then defective circuits might not be detected by the test set generated by the ATPG. Unfortunately, evaluating the worth of behavioral fault models is difficult, and as yet no study has done a complete comparison of behavioral faults to the full spectrum of possible and probable circuit defects. It must also be noted that some behavioral test generation techniques don't have explicit fault models; evaluation of the test programs produced by such systems must be done on a

case-by-case basis [4].

The HDL's used by behavioral systems incorporate functional level data transformation primitives and advanced control constructs. HDL's in general can be characterized as procedural or non-procedural [26]. In non-procedural HDL's, each statement in a given design is evaluated in parallel, at each time step every statement is checked to see if it executes. In the more abstract procedural HDL's each statement is evaluated in sequence, with execution of a given statement occuring in the time period after the preceeding statement is evaluated. In general, non-procedural HDL's are "closer" to a hardware implementation than procedural HDL's which are often used to describe complex algorithms whose hardware implementations are difficult to derive.

Levendel and Menon [26] demonstrated that the D-algorithm could be applied to circuits specified by a HDL description. They classified three basic fault types in HDL descriptions and showed test generation methods for each type. The three types of faults are:

- **pin fault** - identical to a stuck-at fault, an input, output, or state variable is stuck at a logic value.

- **control fault** - a control variable or expression having the wrong value.

- **general function faults** - faults whose effects are known, but can't be modeled by the other two fault types. An example is an ADD function failing to a SUBTRACT function.

Levendel and Menon develop an algebraic method for generating

D-propagation conditions for HDL constructs and functional primitives. They then apply a D-algorithm with extensions to account for the sequential nature of the circuits to be tested. Tests for the three types of faults can be generated for both procedural and non-procedural HDL descriptions. This algorithm should usually generate good fault coverage as the fault model subsumes all line stuck-at faults, but contains the same backtracking problems inherent to the D-algorithm. Furthermore, for complex functional blocks, the algebraic method for deriving D-propagation conditions is impractical and a control graph approach should be employed.

Several recent behavioral level test generation systems lack explicit fault models. One, developed at GTE by Son and Fong [27] uses generic HDL circuit descriptions that are translated into functional tables. The derived functional table consists of two parts: the first specifies the conditions that determine what actions will be taken, and the second describes the function performed by each action. The test engineer must provide a list of the faults to be considered by the test generation system; this information is then incorporated into the functional table. Condition and Action tables are then generated which allow the system to derive sensitization and propagation conditions for test generation.

The GTE approach needs to have a good fault model coupled with an automatic fault list generator to produce a complete ATPG. The techniques used to generate the function tables also need to be better defined and automated.

An even more abstract method has been developed at Texas Instruments [5]. The TI approach uses a structured, ALGOL-like language to describe intended circuit behavior. The design language is hierarchical in nature, and any function can be expanded to a complete gate-level implementation if desired. As in the GTE approach, the TI system lacks an explicit fault model. The basic testing paradigm is "to insure the circuit is behaviorally exercised." This goal is

accomplished by generating in advance a list of functional components which should be exercised to assure adequate test coverage. The heuristic used to judge which functional components should be exercised roughly corresponds to a critical path analysis, with dominated components being removed from consideration.

The TI system transforms the HDL description into two linked directed graphs, one corresponding to the data paths and the other the control paths of the circuits. Links from the control graph regulate the transversal of nodes in the data graph.

The hierarchical nature of the TI HDL allows tests for the circuit to be generated throughout the design cycle, from behavioral specification to gate-level implementation. This continual feedback helps the designer insure the final implementation will be easy to test. Once again, however, this approach suffers from the lack of a thorough fault model.

Armstrong, Barclay, and Gupta [28,29,30] have proposed a behavioral level test generation system that incorporates an explicit fault model. Gupta and Armstrong used GSP, a hardware simulation language, to describe circuits to be tested [29]. Components are specified either by look-up tables defining arbitrary functions, or by sequences of "micro-operations", roughly analogous to RTL level statements with some control features included.

The basic fault model is the Model Perturbation (*MP*) approach where the function of a component of the circuit is disrupted. In circuits defined by tables, entries in the tables are changed to yield the faulty functions. In circuits characterized as sequences of micro-operations, faults are manifested as a change in some micro-operation to another micro-operation. In the micro-operation models, control faults are obtained by faulting micro-operations in control expressions.

Propagation and justification of the injected faults are carried out by

backtracking consistency and propagation algorithms. For the look-up table circuits, heuristics are applied to choose values with which to fault the tables. Faults should be chosen so total fault coverage is maximized.

Micro-operation faults are usually taken to be a micro-operation faulting to its logical dual or the wrong branch of a control statement executing.

Barclay and Armstrong extended and automated the MP approach [28,30]. Employing a non-procedural subset of VHDL for circuit description, four fault types were characterized (Figure 1). The ATPG implemented with these fault types performed test generation by performing heuristic goal solving. The root goal of the system is to generate a valid test for a given fault. The system breaks this goal into subgoals of propagating the fault syndrome to an output and justifying the sensitized path. These subgoals are in turn further decomposed until goals of manipulating input lines are generated; goals of manipulating inputs are satisfied automatically and are the conditions which are compiled into test vectors. At each stage of the goal decomposition and solving process, heuristics are used to choose the next goal to try to solve. This corresponds to choosing a path through the circuit.

The goal solving process is a search for valid propagation and justification conditions for a test of the fault being examined. A few sample goals are **VIE** - have a given value in an expression at a certain time or **OBSOBJ** - observe the value in a specified data object at a certain time. Note that solving a OBSOBJ goal will always require breaking the goal into subgoals unless the data object is a circuit output; this goal decomposition will result in moving the value in the object closer to the circuit outputs.

STUCKTHEN and STUCKELSE - an IF statement becomes stuck one way or another, as if the conditional expression became stuck true or false.

DEADCLAUSE - a WHEN clause in a CASE statement fails to execute when selected.

ASSNCNTL - an assignment statement fails to execute.

MICRO-OP - a micro-operation fails to some other operation.

Figure 1. Fault types in Barclay / Armstrong model.

Barclay implemented the ATPG system in ProLog and included a fault list generator that produces a list of all modeled faults present in the circuit under test. Simple measures of controllability and observability of the VHDL statements in the circuit description are used to guide the goal-solving process so the most promising paths for fault propagation and justification are tried first.

The ATPG system is unable to generate tests for certain modeled control faults. To observe a fault in a control statement, the execution or non-execution of a assignment statement under control of the control statement must be monitored. Execution is monitored by observing changes in the value of the object, thus for a reliable test, the starting value of the object must be known. Often, however, the only way to preload a known value into the object is by using a statement under the faulty control expression. Barclay's system will not allow fault sites to be used for test justification, so the object cannot be preloaded and the test fails. Barclay's approach further cannot handle reconvergent fanout.

O'Neill [31] enhanced Barclay's system by providing a new method for selecting times when goals must be solved, allowing fault sites to be used for justification in certain cases, and adding overall goals of *Justification*, *Sensitization*, and *Propagation* to help guide the selection of goals to be solved. These extensions resulted in a average performance improvement of an order of magnitude, but the approach still suffers from poor performance, an inability to deal with reconvergent fanout, and the inability to handle some control faults.

Chapter 3

VHDL Circuit Models

The E-algorithm accepts circuit models described using a subset of VHDL (VHSIC Hardware Description Language). Circuit descriptions are limited to block structured, non-procedural data-flow representations. These representations consist of groups of data signal assignments and transformations using functional operators such as ADD, AND, and SUB. Signals can be of the type BIT or BIT-VECTOR, from which other types can be formed. Control statements such as IF..THEN..ELSE and CASE are used to govern the conditional execution of blocks of assignment statements. Branch statements such as GOTO, common in RTL descriptions, are disallowed because of the non-procedural nature of the circuit models. All statements in non-procedural models execute in parallel, with only values of inputs or internal signals used to compute new signal values. Variables which assume a series of values sequentially through the circuit description are not permitted. Algorithmic structures such as FOR loops or REPEAT...UNTIL blocks are excluded by this prohibition on variables.

Data transformation operations are at presented limited to a fairly small set. The basic boolean operations AND, EQ, NOT, OR, and XOR are defined for bits and bit vectors. Operations on bit vectors are: unsigned ADD and SUB; PARITY, which generates even parity for the vector; the less-than-or-equal-to and

greater-than-or-equal-to comparison operations; concatenation of two vectors; and slicing, which extracts a part of the vector. The MULTIPLY operation for bit vectors is at present partially implemented; propagation cubes for some, but not all, input conditions can be derived. The operation set is easy to expand as adding an additional operation requires just a set of primitive fault cubes and propagation conditions. The propagation conditions may be either in the form of propagation cubes or an algorithm that simulates the operation and generates propagated fault syndromes given a set of inputs.

All signals are assumed to have only one driver and only one procedure block is allowed. Extensions of the algorithm to multiple procedure blocks should be simple, but the bus resolution functions required for multiple signal drivers are usually expressed as algorithms and cannot be handled by the E-algorithm.

A simple timing model is employed to permit sequential behavior. All logic is assumed to be synchronous, with time divided into equal periods by a master clock which drives the inputs and outputs. Inputs are set at the beginning of the clock period and outputs sampled at the end. Input and output signals do not have to be synchronous *with respect to each other*, the master clock is assumed to be the clock for the test fixture, not necessarily the circuit under test. Circuit signals do not have to change synchronously with respect to some defined circuit clock, they must merely be synchronized to the tester clock. This assumption permits asynchronous logic to be simulated to the same degree of accuracy that the signals can be driven by the test fixture. Each master clock period is assumed to be long enough for all internal signal propagation to occur and outputs to stabilize.

Figure 2 illustrates this timing model with an example of signal timing for a simple D flip flop with an asynchronous clear CLR and a circuit clock signal CLK. Notice that all signals change synchronously with the master clock

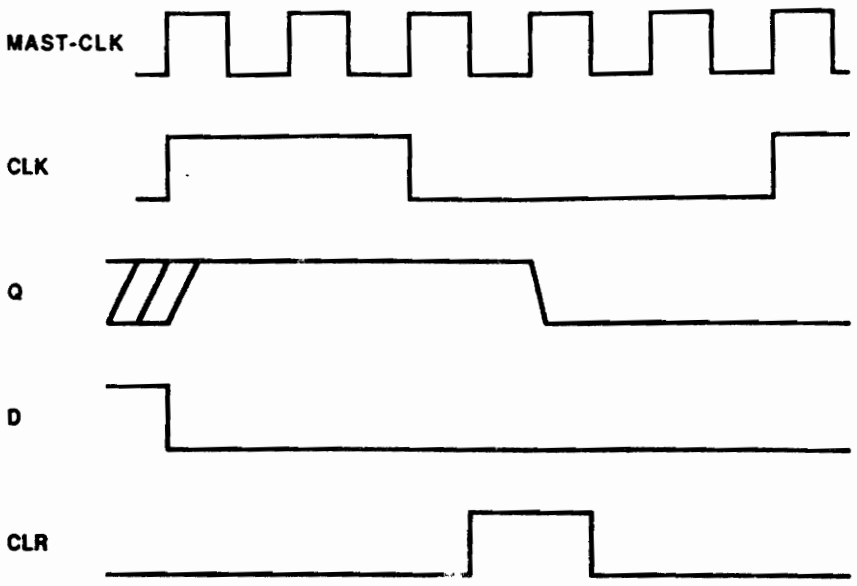


Figure 2. Example of timing model and "asynchronous" logic transitions.

MAST-CLK. In this case, the input D and the output Q change synchronously with the first transition of CLK. The CLR signal, however, goes active (high) in the middle of the CLK period, causing an asynchronous (with respect to CLK) change in Q.

Circuit clocks or signal transitions can be modeled with the *signal*'STABLE boolean operation which is false when *signal* changes value. Thus, the use of the 'STABLE operation implies two time periods. A typical use of 'STABLE is to model clocked logic with statements such as: **IF** CLOCK **AND NOT** CLOCK'**STABLE THEN** ... where signal assignments under control of the IF will execute on the leading edge of CLOCK. The values loaded by the assignments after the clock transition are calculated using the signal values from the period before the clock transition.

An example of a VHDL circuit description which uses the 'STABLE operation to model a circuit clock is the code fragment given in Figure 3 which specifies the behavior of a D flip flop. This code exhibits the timing behavior shown in Figure 2. Although the subset of VHDL used by the E-algorithm is small, complex behavior may easily be specified and few restrictions are placed on the types of circuits which may be considered. However, though the VHDL subset provides a useful basis for designing and simulating circuits, VHDL circuit descriptions in their original form are somewhat ill-suited representations for the execution of a test generation algorithm.

```
IF CLR = '1' THEN
    Q <= '0';
    QBAR <= '1';
ELSE
    IF (NOT CLK'STABLE) AND (CLK = '1') THEN
        Q <= D;
        QBAR <= NOT D;
    ENDIF
ENDIF
```

Figure 3. VHDL code for simple D flip flop.

Chapter 4

Graph Transformation & Fault Model

VHDL circuit models are a useful form for describing the behavior of a circuit but are non-optimal representations for a test generation algorithm. Most of the computational effort expended during test generation is a result of tracing data paths through the circuit or deriving sequences of signal values required to establish and maintain these paths. Therefore, we are motivated to find a representational scheme that eases these tasks. One description paradigm that achieves this goal is a data flow graph representation which explicitly shows all data and control logic paths in the circuit. A fault model based on this representation has been developed.

4.1 Data Flow Graph Representation

As a pre-processing step to the E-algorithm, a data-flow graph of the circuit under test is derived from the VHDL description. The graph contains operation blocks corresponding to the basic data functions, eg. NOR or SUB, specified in the VHDL description. Operation blocks are interconnected by data paths, data buffers, and control lines. Inputs and outputs to the circuit are assigned to I/O

lines in the graph. Signals can be either bits or bit vectors, the connected data paths conform to the width of the signal.

Internal signals in the VHDL description are transformed into graph signal blocks. Each signal assignment in the VHDL model to an internal signal maps to a data buffer connected to the input of the corresponding graph signal block. The input of the data buffer comes from the signal or operation block corresponding to the expression on the right hand side of the VHDL signal assignment statement. The data buffers each have a control port that regulates the flow of information into the connected signal block. Whenever the graph branch attached to the buffer control port has a logic value of '1', the data at the input to the buffer is latched by the signal block. The control branch will have a '1' value when the conditions controlling the data assignment in the VHDL model are met. See Figure 4 for examples of VHDL-to-graph transformations. Notice that the values on the control branches connected to each data buffer are equivalent to the boolean result of the evaluation of the control expression that governs the execution of the signal assignment in the VHDL description.

Feedback paths in the circuit are permitted, but there must be at least one clocked signal assignment in every loop. This restriction requires that all sequential logic be modeled explicitly; cross-connected operation blocks are prohibited. In behavioral models this is no great limitation as sequentiality isn't typically modeled with combinational feedback.

VHDL

```
IF A XOR (B AND C) THEN
     C <= DT1;
ELSE
     C <= DT2;
ENDIF;
OUTPORT <= C;
```

GRAPH MODEL

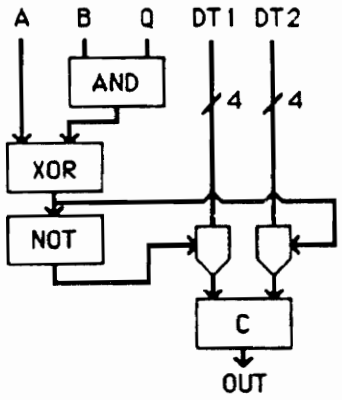


FIGURE 4.a IF...THEN Statement Transformation

```
IF CLOCK AND (NOT CLOCK'STABLE) THEN
   CASE CNTRL is
      WHEN '00' => REG <= DTINP;
      WHEN '01' => REG <= ADD(B,C);
      WHEN '11' => REG <= '0000';
ENDIF;
```

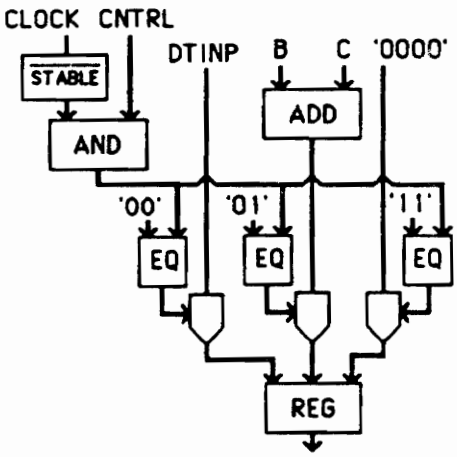


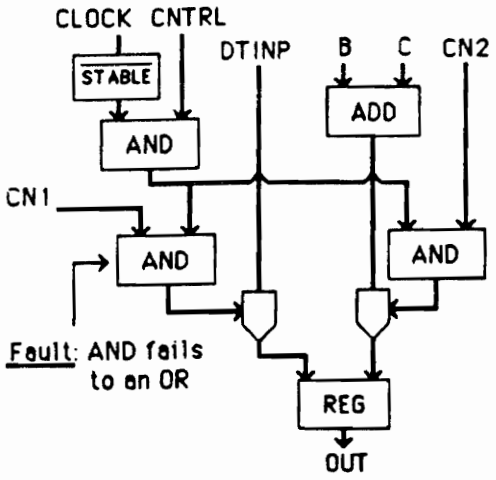
FIGURE 4.b CASE Statement transformation and 'STABLE example

FIGURE 4. VHDL to Graph transformations.

4.2 Fault Model

The fault model for the E-algorithm considers two types of faults: defects in functional operation blocks and stuck-at faults in data or control lines. Each operation block is faulted by having it fail to some other operation; in the current implementation the dual operation is substituted for the faulty case. Faulting an operation to its logical dual, ie. AND -> OR, is a convenient scheme, but the algorithm permits arbitrary faults in the function of any operation. Research [32] has been conducted to find what operation substitution(s) yield the best fault coverage.

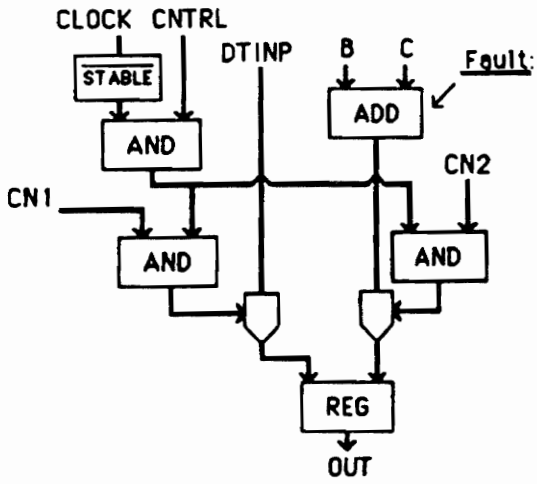
Data and control lines are faulted by impressing stuck-at faults on the lines. For data paths, these faults usually affect the entire path, ie. all bits in a vector-wide data line are simultaneously faulted to a single value although the algorithm can handle individually faulted bits. Control faults are modeled as stuck-at-1 and stuck-at-0 faults on control branches. A stuck-at-0 fault on a control branch will prevent data transfers through the attached control buffer(s) from ever occuring. In the VHDL description this corresponds to an assignment statement or block of assignment statements being prevented from ever executing. Similarly, a stuck-at-1 fault on a control branch causes the associated data transfer(s) to occur at every time period, thus modeling assignment statements which execute continuously (with every master clock) in the VHDL model. (When the branch is under a 'STABLE operation, even if the line is stuck-at-1, the load won't occur until the clock transition.) Each segment of a control branch fan-out tree is individually faulted. Note that faults in operation blocks in the control logic of the circuit can have effects identical to control branch faults, depending on the nature and location of the fault. Figure 5 shows the effect of several faults on the function of a circuit.



The fault of the AND operation to an OR results in the load of DTINP through the data buffer into REG occuring under the wrond set of conditions.

In particular, the load will occur with every transition of CLK when CNTRL is true, regardless of the value of CN1.

Figure 5.a A functional fault in control logic.



The effect of this fault is easy to characterize: the value loaded into REG through this path will be SUB(B,C) instead of ADD(B,C).

Figure 5.b A fault in a data path of the circuit.

Figure 5. Examples of circuit faults.

Comparing the fault model to that proposed by Levendel and Menon [26], their *pin* faults correspond to stuck-at faults on graph data paths and the *general functional faults* are equivalent to faults in the operation blocks. Their *control fault* model is here a result of a stuck-at fault on a control line or a functional fault in the logic that generates control signals, not an entirely separate class of faults.

The effects of the faults in the E-algorithm fault model are similar to those of the fault model developed by Armstrong et al. [28,29,30]. Certain faults in the E-algorithm model, however, have no clear correspondence. For example, a control branch fault can have the effect of a STUCKELSE fault, but still permit the use of the assignments under the THEN clause if the fault occurs in the control logic after the THEN branch. In the Armstrong - Gupta - Barclay model, a STUCKELSE fault precludes the use of the THEN clause during test generation.

Fault equivalence and dominance principles can be applied to reduce the number of faults to be considered. In the initial implementation, however, tests are separately generated for each possible circuit fault.

Chapter 5
E-Algorithm

The test generation approach outlined here is an extension of the D-algorithm [1], called the E-algorithm, that generates tests for digital circuits described by a non-procedural HDL. A graph transformation is applied to the HDL circuit model to yield a representation that eases the description and implementation of the E-algorithm. The graph representation also forms the basis for an intuitively satisfying fault model for control faults.

Following the D-algorithm form, the process of generating a set of test vectors for a given fault in a circuit can be broken into three main steps. The first step is *Sensitization* where the effect of the fault is made to manifest itself at the fault site. The second step is *Fault Propagation* where the effect of the fault is moved through the circuit until it can be observed at a primary output. During propagation, some circuit signals must be set at certain times to establish the path the fault syndrome uses to get to the outputs. Unless all the required signals are primary inputs and thus may be controlled directly, a sequence of input values must be found that set the needed signals at the proper times. The third step in the procedure works these propagation requirements back to the inputs and is called *Justification.*

The E-algorithm employs the standard notation, using the logical values

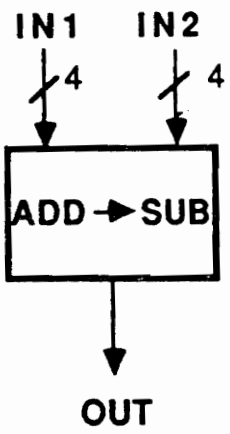
0,1,X,D,$\overline{D}$ where D is herein assumed to = 1 (good signal value) and 0 (faulty value). During test generation, the *D-frontier* lists the signals that manifest the fault as propagation proceeds. A set of symbolic variables E and $\overline{E}$ are introduced to propagate fault syndromes through control structures. The test generation demands of sequential circuits require a further extension to the standard notation: each signal assignment or justification requirement is given a *time tag* which specifies when the assignment is done or the signal value is needed.

5.1 Fault Sensitization

For a test to be generated for a given fault, the fault site must be sensitized so that at least one circuit line or block produces or carries a value containing D or D. Such a condition indicates a value that differs in the faulted circuit from the value in the fault-free circuit. Sensitizing the fault is accomplished by choosing from a fault table a *primitive cube of failure (pdcf)* that specifies what conditions must be set at the inputs to the faulted line or block to produce the given faulty value. The faulty output of the block or line is a value or set of values, containing at least one 'D' or '$\overline{D}$', known as the initial *D-vector* for the fault.

The primitive cubes for each fault can be generated with relative ease. For simple operation blocks such as AND or OR, the pdcf is formed by intersection of the gates' singular covers (the set of all prime implicants for the function). Pdcf's for more complex operations such as ADD and SUB can be formed the same way, but this process becomes expensive for complex operations with vector-wide inputs. Alternatively, pdcf's for these complex vector-wide operations, including ones such as PARITY, are generated by comparing the outputs of the

fault free operation and its dual for a sets of inputs. Input value sets are tried



The Normal operation is ADD(IN1, IN2) , and the faulty operation is SUB(IN1, IN2).
To find a primitive cube of failure for this fault values are cycled through and applied to IN1 and IN2. The operation is simulated and the outputs of the faulty and faulty-free operations are compared. When they differ, a pdcf has been found.

Ex. IN1 = '0000' , IN2 = '0001', produces
OUT = '$\overline{D}\overline{D}\overline{D}1$'

Figure 6. Pdcf for a 4-bit wide ADD failing to a SUB operation.

until one is found that produces D's on the operation block output. This procedure eliminates the need to store a large number of pdcf's for these complex operations. An example of the formation of a pdcf for an ADD failing to a SUB operation is shown in Figure 6.

5.2 Propagation

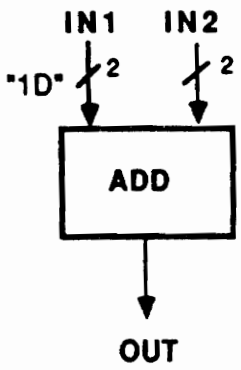
After the fault has been sensitized, unless the fault occurs at a circuit output, the effect of the fault (the *fault syndrome*) must be moved to an output so that it may be observed. Moving the fault syndrome to the output consists of propagating the D-vector of the fault along a path through data operations, signal blocks, and control structures until an output is reached. The D-vector is driven through each successive data operation or signal block in the path by using propagation rules proposed by Levendel and Menon [26]. Propagating fault syndromes through control branches is somewhat more complex and is discussed in the next section.

To propagate a fault syndrome through an operation block, the D-vector is intersected with the *propagation D-cube (pdc)* of the operation. The pdc of an operation specifies the conditions under which a D or $\overline{D}$ at the operation block's inputs can reach the operation block's output (although perhaps as the complemented value D or $\overline{D}$). Usually, these conditions include restraints on the values at the other inputs to the operation block. For simple operations, eg. AND, OR, etc., the pdc's are simple to derive [1,3,26]. More complex operations (ADD, SUB, etc.) require special algorithms which are used to obtain propagated syndromes and justification requirements bit-by-bit [26,15].

An example of propagation through a complex operation, namely an ADD block, is given in Figure 7. (Again notice that the ADD operation is here assumed to be unsigned and the carry is not considered.) The table in Figure 7.a [26] shows the intersection rules which are applied to the two inputs bit-by-bit. To derive a propagation cube for a D-vector on one input, the bits of the faulty input are matched against the table to find a vector to apply at the other input. The chosen vector must permit a D to appear at the ADD output. In general, the selection criteria is based on the zero-controllability and one-controllabilty [33] of the input, i.e. the vector value chosen is the one that is easiest to justify and still produces a D output. The secondary goal during this process is to produce a maximal number of D's in the output.

SUM	0	1	$\overline{D}$	D	X
0	0	1	D	$\overline{D}$	X
1	1	0	$\overline{D}$	D	X
D	D	$\overline{D}$	0	1	X
$\overline{D}$	$\overline{D}$	D	1	0	X
X	X	X	X	X	X

Figure 7.a Bit intersection for propagation through an ADD



A value must be found for IN2 that allows the "D" input at IN1 to propagate through the operation.

Here assume that the one-controllability of IN2 is greater than its zero-controllability.

Intersecting the IN1 syndrome with bits of IN2, trying IN2 = all ones initially, we find a value for IN2 that allows propagation.

IN1 ⊓ IN2 = OUT

1D ⊓ 11 = 0$\overline{D}$ So IN2 = 11 will allow propagation.

Figure 7.b Example of justification value selection.

Figure 7. Fault propagation through an ADD operation.

Usually, the fault syndrome must pass through a number of operation and signal blocks before a primary output is reached. A procedure must be devised to systematically propagate the fault syndrome towards the outputs, identifying a path for the test. The procedure must be able to attempt propagation along many different paths, or even several paths simultaneously. The E-algorithm uses a relatively simple derivative of the D-algorithm to propagate fault syndromes through data structures (data operation or signal blocks):

1. Sensitize the fault: select fault cube; set D-frontier to initial D-vector; set justification list to justification requirements of primitive fault cube; assign initial time-tags to the D-vector and justification list.

2. At limits of the D-frontier, select the next object(s) in path(s) to be propagated through. This selection is based on stored observability measures for each graph block; these measures indicate the relative distances of the objects from the circuit outputs. Typically, single-path propagation will be tried first, but if no single consistent propagation path can be found then multiple paths will be examined.

3. Propagate the fault syndrome through the selected object by intersecting the D-frontier and its time-tags with the propagation requirements of the object. This creates a new D-vector at the outputs of the object. If propagation is through the data path of a control buffer (and under a 'STABLE), then an additional time period is implied and a new time tag is created for the new D-vector.

4. Check propagation requirements for consistency with justification list. If inconsistencies arise, attempt to reschedule data signal loads.

5. If no propagation cube is consistent, try another path. GOTO 2.

6. Add propagation requirements to propagation list, construct a new D-frontier.

7. If primary output has been reached, resolve justification requirements back to primary inputs and STOP - test has been generated. (If justification fails, backup and try another path. GOTO 2)

8. If a primary output has not been reached, continue propagation from the new D-frontier. GOTO 2.

Whenever a path turns out to be a dead-end, the algorithm backs up to the last untried path in the circuit and propagates through that path. The choice of paths is guided primarily by observability measures. The observability measures are rough estimates of the difficulty in reaching the output along a certain path; rules similar to those discussed by Fong [33] are used by a pre-processor to generate the observability of each circuit object prior to running the test algorithm for any faults. In cases where different paths have identical observabilities, the controllability of the signals required to justify each path is considered as a second decision criteria. Though at any point during propagation single paths are usually considered first, heuristics have been added to recognize certain circuit structures where multiple path propagation is often required. In these cases,

multiple path propagation is tried first and single paths traced only if necessary.

The primary example of a situation where multiple path propagation is often necessary is the case where a single signal block is loaded by two or more different data buffers. When a line or operation exhibiting the fault reaches a fanout point in the circuit where the syndrome can reach both buffer ports, multiple path propagation along these two paths is initiated. This situation usually corresponds to assignments being made to the same signal in both the THEN and ELSE clauses of an IF...THEN statement.

Faults are propagated through graph elements by intersecting the pertinent elements of the D-frontier with the gate's propagation cubes or invoking its propagation routine. The object's propagated syndrome and justification requirements are assigned time tags that specify when they are available or required. The time tags will be the same as the tag of the D-frontier elements at the input of the object unless the object is a control buffer or line under a 'STABLE operation. In either of these cases, at least two time periods are implied and new tags are created for the appropriate signals (the output signals will be valid after the clock transition and are assigned the new tag).

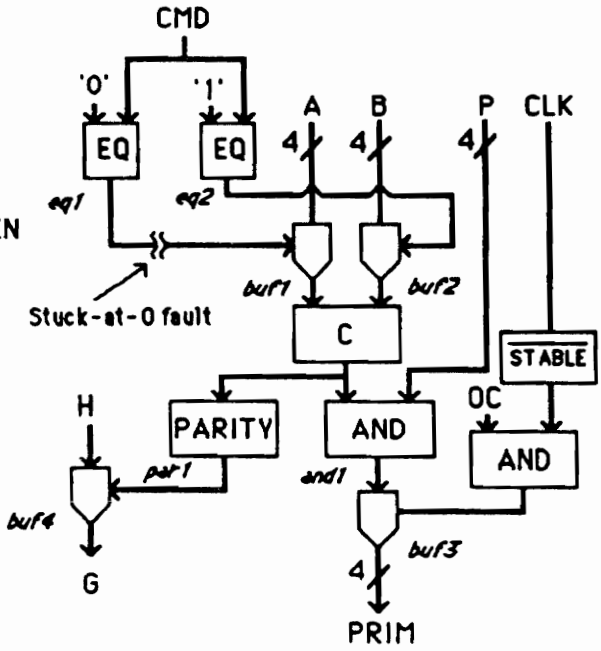
Once the fault syndrome reaches a primary output, a complete propagation path has been formed and the justification algorithm is invoked. If an irresolvable inconsistency is detected during justification, then the propagation routine tries to find a different propagation path. Heuristics are employed to help the E-algorithm quickly backtrack to the point in the propagation path from which the inconsistency arose. Unfortunately, the source of conflict often cannot be pinpointed, thus requiring blind backtracking until a new consistent path is found.

5.3 Example

Figure 8 shows a VHDL fragment and its associated graph. *PRIM* and *G* are primary outputs, while *A*, *B*, *H*, *P*, *OC*, *CLK*, and *CMD* are primary inputs. A fault has occurred which prevents the first clause in the CASE statement from ever executing (a DEADCLAUSE fault). In the graph model, this fault corresponds to the control branch which relates to the boolean evaluation of the execution condition of the faulty clause being stuck-at-0. This fault prevents the connected data buffer from loading the signal C.

To generate a test for this fault, the fault is first sensitized by setting the output of *eq1* to 1 at an initial time *tag*; this produces a *D* on the faulted line. Propagation of the fault syndrome then begins by finding the most observable object with an input connected to the faulted branch. In this case, there exists no path through data paths - the only route is through the control port of *buf1*. Unfortunately, there is no way for the fault syndrome to directly affect the data in the signal block. The effect of the fault therefore can not be propagated by using D-propagation rules and a path for the syndrome must be found some other way.

```
CASE CMD is
     WHEN '0' => C <= A;
     WHEN '1' => C <= B;
ENDCASE;
IF OC AND CLK AND (NOT CLK'STABLE) THEN
     PRIM <= C AND P;
ENDIF;
IF PARITY(C) THEN
     G <= H;
ENDIF;
```



Fault: DEADCLAUSE when CMD = '0'
(Clause will never execute).

Corresponds to a stuck-at-0 fault on the output of Eq1 block.

Figure 8. Example of control branch fault.

5.4 Propagation Through Control Branches

The symbolic variable E and its complement $\overline{E}$ are used in the control branch propagation procedure. The data object directly under control of the control branch exhibiting the fault syndrome is "loaded" with a vector (or bit, if the object is of type BIT) composed of bits set with the initial symbolic value E. These E-bits are then propagated towards the outputs through operations using rules similar to the D-propagation methods. Thus, when a D-vector reaches a control branch, the data path propagation algorithm invokes the control branch propagation process to find a path for the fault:

1. Select a signal block connected to a data buffer whose control port is attached to the branch exhibiting the fault syndrome.

2. Load the selected signal block with a vector (as wide as the signal block) containing subscripted E-bits. The subscripts on the E bits indicate the bits' positions in the signal block.

3. Propagate the *E-vector* towards the circuit outputs using the D-propagation rules, but substituting 'E' for 'D'.

4. When the E-vector reaches the output or a second unavoidable control branch, invoke the *E-Justification* routine to select a value or sequence of values to be instantiated for the E-bits in the original selected signal block.

The E-vector is used to establish a propagation path for the fault syndrome from the signal block under the faulty control branch. As the E-vector is propagated from the signal block, it accumulates information on the effect that values in the signal block have on other circuit elements. Therefore, when the E-vector has reached an output, the output value can be specified as a function of the E-bits of the E-vector (and implicitly of the other signals required to form the propagation path). This procedure is nearly identical to D-propagation, with the exception of propagation through certain complex operation blocks (see section 5.4.1.) The effect of a certain value in the selected data block upon the output can now be discerned.

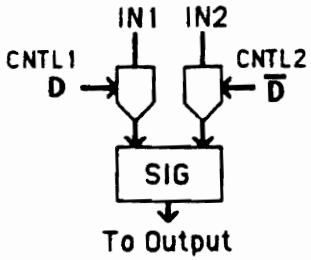
To observe the fault syndrome, conditions must be established so that when the fault is present one value is loaded into the signal block, and when no fault exists a different value is loaded. The two different values are chosen after examining the E-vector at the output to determine which bits of the signal block reach the output and in what form; often only some bits of the E-vector will reach the output. As different signal block values may produce identical output values, an inequality satisfaction routine (see section 5.4.2) is applied to the E-vector at the output to determine what E-bit values in the signal block will produce different outputs. The E-Justification routine chooses the values to be loaded into the block (added to the overall justification list).

The set of good/bad values or sequence of values to be instantiated for the E-vector at the signal block are chosen depending on the topology of the faulty control branch and attached signal block. Figure 9 shows the 5 basic cases of propagation through control branches and the strategy for assigning values to the initial E-vector. Note in Figure 9 that E and $\overline{E}$ correspond to the two different actual values *(derived from the inequality satisfaction routine)* loaded into the bits

of the E-vector which make it to an output - *they do not refer to a specific E-bit. Thus, in Figure 9, E stands for one actual value to be justified for the symbolic E-vector, and* $\overline{E}$ *corresponds to another value which produces a different output response.*

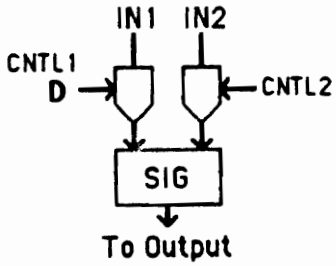
Notice that in most cases sequences of values are loaded into the E-vectors. This is a consequence of the nature of control faults. Suppose a single signal block has separate load buffers, with one controlled by a branch with a value of 'D' and another with a control branch value of '$\overline{D}$' (Figure 9.a) A test that would detect the fault in such a case consists of justifying one value on the input to one buffer, and the inverse value on the input to the other buffer. The value of the output will indicate which load occured, and thus the presence or absence of the fault.

However, if the D-frontier of the fault doesn't reach at least two buffers of the same signal block, then the fault can't be detected by observing if the wrong load occurs in one time period. Instead, these fault cases are covered by checking if a load under control of the faulty branch can be properly executed. A known value is loaded into the selected signal block, and the output then observed for a change or non-change produced by attempting a load (or trying to prevent one from occuring) with a different value using the data buffer with the faulty control branch. The known value must be loaded first to initialize the signal block. Figures 9.b - 9.e show these cases.



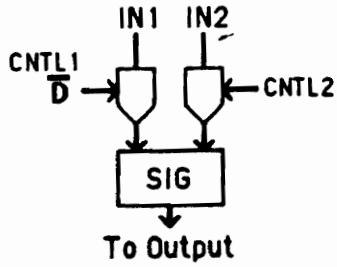
Branch Topology	E-Justification Strategy				
	Time	CNTL1	CNTL2	IN1	IN2	OUTPUT f(SIG)
	t0	1 ⟹ (IMPLIES)	0	E	Ē	E / Ē (good / faulty)

Figure 9.a



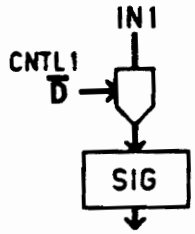
Time	CNTL1	CNTL2	IN1	IN2	OUTPUT f(SIG)
t0	0	1	X	E	E / E (preload)
t1	1	0	Ē	X	Ē / E (test)

Figure 9.b



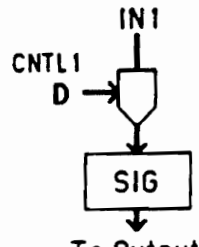
Time	CNTL1	CNTL2	IN1	IN2	OUTPUT f(SIG)
t0	0	1	X	E	E / - (preload)
t1	0	0	Ē	X	E / Ē (test)

Figure 9.c



Time	CNTL1	IN1	OUTPUT f(SIG)
t0	1	E	E / E (preload)
t1	0	Ē	E / Ē (test)

Figure 9.d



Time	CNTL1	IN1	OUTPUT f(SIG)
t0	1	E	E / -
t1	1	Ē	Ē / -

Faulty output value is undetermined, but will n change, and thus cannot b mistaken for fault-free behavior.

Figure 9.e

FIGURE 9. E-Justification strategies for the five basic branch topologies.

In topologies of Figures 9.b and 9.c, there is at least one other data buffer, not exhibiting the fault, which is attached to the signal block under question. Known preload values can be loaded into the signal block using this fault-free buffer. In both 9.b and 9.c, E (one value produced by the inequality satisfaction routine) is loaded into the signal block using the fault-free buffer. In 9.b, an attempt is then made to load $\overline{E}$ into the block through the faulted buffer and observe the effect of the value change at the output; if the fault is present, then the load will not occur and the output will be unchanged. In 9.c, the converse holds; the value $\overline{E}$ is placed at the input to the buffer, but in the fault-free circuit it won't be loaded into the signal block. When the fault D is present at the control port, the $\overline{E}$ will be loaded and the output will show the effect of this value change.

If the faulty control branch is the only one that can be used to load the signal block, then it is first assumed to be fault-free, a preload value loaded, and then a test for incorrect control branch value made. In Figure 9.d, the $\overline{D}$ on the control port makes observing the fault effect easy. Justifying the *CNTL1* line to '1', the E value is loaded into the signal block. Then the E value is justified on the *IN1* line and *CNTL1* set to '0'. If the fault occurs, the $\overline{E}$ will be loaded (as the control branch of the buffer is stuck-at-1) and the value transition observed. In Figure 9.e the observation of the fault is somewhat different. Loads of first the E value and then the $\overline{E}$ value are made. In the fault-free circuit, these loads will cause the output to change from one known value to another. In the faulty circuit, the exact output values are unknown, but, due to the propagation method and the fact that the value in the signal block won't change, the output will not change, permitting the fault to be detected.

Armed with the E-propagation and justification routines, we may now return to the example of Figure 8. The *D* @ *t0* at the input to *buf1* causes *C* to be loaded with $E_{c\{1-4\}}$. This E-vector is now propagated using data path rules. The most observable object under *C* is *and1*; *par1* is not chosen as it requires propagation through another control branch. $E_{c\{1-4\}}$ is intersected with the propagation cubes of *and1*. A '1' in any bit of *P* would allow the syndrome to propagate through *and1* but assume here that the cube chosen for intersection requires *P* = '1111' and yields Output(*and1*) = $E_{c\{1-4\}}$ @ *t0*. Only one route is available for further propagation at this point, so $E_{c\{1-4\}}$ @ *t0* is intersected with the cube of *buf3*, giving *PRIM* = $E_{c\{1-4\}}$ @ *t1*. Note the output has now been expressed as a function of the E-vector $E_{c\{1-4\}}$. The new tag *t1* is produced since *buf3* is under a 'STABLE operation block, causing *buf3* to load its contents into *PRIM* at the leading edge of *CLK*, in the time period after *t0*.

As the E-vector has reached an output, the E-Justification procedure is invoked. The topology of the *C* signal block / data buffer where $E_{c\{1-4\}}$ is loaded corresponds to that of Figure 9.b. With the appropriate substitutions (eg. IN1 = *A*, IN2 = *B* etc.), the sequence of Figure 9.b is used and the values for the E-vectors are chosen by the inequality satisfaction routine based on the constraints accumulated during E-vector propagation. In this case the choice is simple, merely any two different values ex.: E = '1111', $\overline{E}$ = '0000'. After justifying all required signals, a complete test for the modeled fault is given by:

Time	*CLK*	*CMD*	*A*	*B*	*P*	*H*	*PRIM*
t0	0	1	XXXX	1111	1111	X	- / -
t1	1	1	XXXX	1111	1111	X	1111/1111
t2	0	0	0000	XXXX	1111	X	1111/1111
t3	1	0	0000	XXXX	1111	X	0000/1111

If the E-vector encounters another control branch instead of an output, E-Justification of the vector proceeds as before (with slight modifications to some cases) and then a signal block under control of the encountered control branch is loaded with a different E-vector, E_{new}. E-propagation is then initiated. However, since the fault syndrome at the new control branch is not given by D or $\bar{D}$, different strategies must be used during E-Justification to instantiate the E_{new} vector values when the output or *next* control branch is reached. The values are chosen based on the sequence of explicit values the previous E-vector induced at the new control branch. Figure 10 lists the possible control branch conditions and the corresponding E_{new} load strategies.

Time	*Sequence at Branch*	*E-Vector Sequence*	*Output Function of*
		Enew	
t0	1 / 0	E_{new}	E_{new} / -
t1	1 / 0	$\overline{E_{new}}$	$\overline{E_{new}}$ / -
t2	1 / 0	E_{new}	E_{new} / -
t0	1 / -	E_{new}	E_{new} / -
t1	0 / -	E_{new}	E_{new} / -
t2	1 / -	$\overline{E_{new}}$	$\overline{E_{new}}$ / -
t0	1 / 1	E_{new}	E_{new} / E_{new}
t1	1 / 0	$\overline{E_{new}}$	$\overline{E_{new}}$ / E_{new}
t0	1 / 1	E_{new}	E_{new} / E_{new}
t1	0 / 1	$\overline{E_{new}}$	E_{new} / $\overline{E_{new}}$

FIGURE 10. E-Justification strategies for successive control branch propagation

The first strategy of Figure 10 corresponds to the output of the cases in 9.a, or 9.c with the exception that the pattern is held for a longer time than indicated. The E-justification procedure for the first E-vector is modified so that the patterns will match (the propagation path from the signal block and the signal block value are held for additional time periods.) The second strategy of Figure 10 corresponds to the result of strategy 9.e, except that an additional load of the E value is made. Similarly, the third strategy matches 9.d and the fourth matches 9.b. Note the assumption in all these cases is that when the value E is justified for the E-vector it will produce a value of '1' at the next control branch, and a value of '0' will be the result of the E load. If this condition cannot be satisfied, i.e: if the value of the loads is determined by circuit structure, with perhaps a literal value having to be loaded, then the strategy table match values can be complemented.

An example of propagation through a second control branch can be seen from the example of Figure 8. If the path through *buf3* were removed from the circuit then the fault syndrome would have to propagate through the control branch of *buf4*. The E-vector value at the output of *par1* can be given as Output(*par1*) = $E_{c1} \oplus E_{c2} \oplus E_{c3} \oplus E_{c4}$. When the inequality satisfaction routine is invoked, values for the $E_{c\{1-4\}}$ vector loads are chosen to meet this condition, for example : A = 0001 and B = 0000. This yields the sequence at the input to *buf4* of: *0/0 @ t1 - t2; 1/0 @ t3*. In order to match this syndrome (given in Figure 9.b) with those given in Figure 10, the value *1/0* is to be held for two more time periods, giving *1/0 @ t3-t5* and matching the first strategy in Figure 10. This strategy reguires the E_{new} vector to be loaded into G and propagated to an output (a feat of no great difficulty in this case). The values instantiated for E_{new} and $\overline{E_{new}}$ are chosen so as they differ (here they can be any two different values). Thus, H (the load path) will be justified to go through the values H =

E_{new} @ *t3*, *t5*; H = $\overline{E_{new}}$ @ *t4*. The output will follow the same sequence and the syndrome can be observed.

The strategies given in Figure 10 for values to load in the E_{new} vector apply in cases where the new control branch controls a buffer that may be loaded with different values. If, however, the branch controls the loading of a literal (eg. '00'), then the previous E-vectors are altered to produce a continuous '1/0' or '0/1' syndrome, effectively a D or $\overline{D}$, and the strategies of Figure 9 followed.

If the E-vector reaches an output, then the test has been found. If, however, yet another control branch is encountered, the rules of Figure 10 are again applied. and the process continues until an output is reached.

5.4.1 E-Propagation through Complex Operations

The propagation rules for E values are identical to the D-propagation rules except in certain instances of complex operations. In simple cases, as in AND or OR operation blocks, the D-propagation rules apply; when used with E values the result is still a set of conditions that permit the value actually loaded into the E-vector to determine what the output of the block will be. However, in the case of certain blocks such as a PARITY block, the situation is more complex. D-propagation through PARITY blocks is simple since the precise good and bad values for the input vector are implicity specified by the D-vector and the operation can be simulated to determine what the output will be and if the fault can be detected. In the case of E-propagation, the *actual* good and bad values in the E-vector aren't known during propagation, they are undetermined until the inequality satisfaction routine picks them. Therefore, simulation of the PARITY operation with an E-vector input can provide no information.

In these cases, a slightly different approach to E-propagation is taken. A single bit E value is *assumed* at the output of the PARITY operation block. The constraints on the generation of that E value are then stored, specifying the value of the E bit as a function of the bits of the E-vector at the input of the PARITY block. The E-bit is then propagated the rest of the way to an output. When the values to be instantiated for the original E-vector are chosen, the constraints on generating the E-bit are considered.

This process is illustrated by the previous example of propagation through successive control branches. Notice that propagating the E-vector $E_{c\{1-4\}}$ through the *par1* block involved specifying the resultant E-bit at the output of *par1* in terms of the bit values of $E_{c\{1-4\}}$. This condition was examined during inequality satisfaction.

5.4.2 Inequality Satisfaction

In most cases, the generation of a pair of values to be instantiated for an E-vector (such that each value produces a different output) is a simple task. The output is usually given as some function of the bits of the E-vector and inequality satisfaction consists of picking values for the pertinent E-bits that cause the output to have different values. Most of the time only a few of the E-bits must be set, the remainder may be left as "don't care's" ("X" valued bits) to ease the demands of justifying specific values.

In situations where E-vectors have propagated through certain complex blocks (see section 5.4.1), however, inequality satisfaction must consider propagation constraints. In these cases, the routine cycles through possible good/bad E-vector values until a pair is found that satisfies the propagation constraints and produces

different output values.

5.5 Justification

Signal values required to sensitize the fault, establish the propagation path, or load E-vectors are moved to the circuit inputs with the justification procedure. The process begins at the primary output forming the end of the propagation chain and works backwards towards the inputs. The justification procedure for the E-algorithm is similar to the D-algorithm justification method [1]. The primary differences arise due to the need to handle time tags and justify through complex functional blocks.

Signals are worked backwards through simple operations by intersecting them with the singular covers of operation blocks. For operation blocks such as ADD, SUB, or PARITY, routines are invoked which consider the controllability of the block inputs and generate a set of input values (which should be easy to justify) to produce the required output. These block input values are then added to the justification list.

The justification procedure is guided by the controllability measures. If different paths for justifying a signal are available, the more controllable is chosen.

If a signal is justified back through a signal block and buffer, then the value of the signal must be available at the buffer and loaded into the block at *or before* the time the value is needed at the output of the signal block. When a signal is justified through a data buffer under control of a 'STABLE operation, two time periods are implied, the time the input to the buffer is required is the time slice before the output is needed.

A certain measure of flexibility is inherent in justifying through data buffers. As mentioned, the input to the buffer and its control branch signal must be available at the time the output is required (or the time before if under a 'STABLE), but this is not necessarily true. The real requirement is on the output value of the buffer; it must be justified at a certain time. If that value can be loaded into a signal block prior to the required time and held unchanged until it is needed, then the justification requirement will be met. Thus, if a conflict arises due to an attempt to load a value at a certain time, an attempt is made to reschedule the load prior to the required time. The new load time is checked for conflicts and, if none arise, justification continues with the load signals and their new time tags.

If an irresolvable conflict arises, the justification procedure halts and the propagation algorithm attempts to find a different propagation path. The justification procedure tries to find the source of the conflict in order to increase efficiency of the backtracking process by having the propagation algorithm backtrack directly to the source of the conflict and try another path. A primitive conflict diagnosis routine detects attempts to load a signal block using a control branch that must remain unchanged during the time period the load could occur. If the conflict source can be identified in this manner the propagation routine is alerted and made to backtrack to the point where the requirement for the load occured.

When all required values have been moved back to the inputs and no inconsistency has arisen, then a test for the fault has been found.

Chapter 6
Results

The E-algorithm has been implemented in Turbo PASCAL on an IBM PC. The complete system includes a pre-processor that derives the graph from the VHDL model and calculates the controllability and observability measures. A fault list is generated for each circuit that includes faulting each operation block and placing stuck-at-1 and stuck-at-0 faults on all control and data lines. Tests are generated for each fault on the fault list, except for faults which disrupted clocking mechanisms. Appendix A provides instructions for using the current implementation.

The test set of 13 SSI and MSI circuits used by Barclay [28] and O'Neill [31] was used to evaluate performance. Tests were successfully generated for nearly all testable (non-redundant and not 'STABLE) faults in the test set. A few faults proved intractable because the sequence of values required to generate tests to detect them could not be loaded (this usually occured in data signals with particularly poor controllability). Complete results for these circuits are given in Appendix B: VHDL circuit models, internal graph representations, and complete fault lists including results are included.

Performance on Barclay's test set was good, with tests for most faults derived in under 2 seconds. The times required to generate tests for these models were

significantly less than the times reported by Barclay or O'Neill [28,31]. These performance figures cannot be compared directly because of the differences in the host CPU's and implementation languages, but, since the IBM PC MIP rating is less than the Control Data machine used by O'Neill, first pass comparisons are favorable.

In addition, tests for several larger MSI circuits were generated, including a registered ALU and an 8-bit controlled counter. The ALU model used is equivalent to the positive logic portion of a 74181, but with output registers added. The 8-bit controlled counter includes a parallel load mechanism. A sequence detector circuit, STATE_MACHINE, was used to demonstrate the problems with generating tests for circuits with poor observability and complex state transitions. Circuit models and fault lists for these circuits are found in Appendix B.

6.1 Circuit Models and Comments

A total of fifteen circuits were used for test generation to provide initial results. The circuit models (both the VHDL code and an abreviated description of the internal representation) and the complete fault list with results are given in Appendix B.

Faults which disrupt clocking signals in circuits modeling synchronous logic were excluded from test generation. These faults consist of faults in operation blocks that generate the clock signals, i.e. in the VHDL model, faults in a control statement containing a 'STABLE operation are excluded. These faults are not considered as they would cause clocking with infinite frequency. Operation blocks or control statements 'past' the clocking statements can be faulted stuck-at-1, but

the assumption is made that they are clock gating signals, and thus will cause data loads with every clock, not with every tester time period.

ADDER

The ADDER model describes a simple four-bit adder at the vector level. Results for this model were excellent, tests for every modeled fault were found.

ADDR2

The ADDR2 model describes a four-bit full adder at the bit level. In actuality, this model is a full gate-level description, included for comparison. Tests for all faults were generated, and no test took longer than 2 seconds to find.

CCNT2

The CCNT2 model is a two-bit up/down counter with a limit register, enable flag, and direction flag. Tests for all the CCNT2 faults were found, except for excluded faults which cause improper clocking. Performance on this circuit was markedly better than that reported by O'Neill [31]. Also, the E-algorithm's lack of the fault reuse avoidance problem allowed several additional tests to be found.

CCNT8

The CCNT8 model is a eight-bit up/down counter with a parallel load. This model was included to test the propagation and primitive cube routines for wide vector operations. Again, all tests were generated for CCNT8 except for the excluded clocking faults. Performance for CCNT8 was good because of the high

controllability afforded by the parallel load feature.

CKTA

The CKTA model consists of two serially-connected and independently clocked registers. An AND gate forms the ouput based on the register values. This model is excellent for exercising load rescheduling to avoid conflicts. Performance on this circuit was good, though a high percentage of the faults were excluded because of clock interference. This circuit illustrates the ability to pass through the fault site during test generation.

CKTCV

The CKTCV circuit consists of a multiplexor with two input registers, a single output register, and a control register that determines which input will be loaded into the output. This circuit also serves as an excellent example of temporal rescheduling, as well as modeling loads to multiple registers through a single data bus. This circuit also demonstrates the E-algorithm's ability to deal with circuits with poor observability.

CNTR

The CNTR model is a bit-level, three-bit counter model. Two tests failed due to loop limit checks. The E-algorithm permits propagation through an object multiple times, but a boundary has been set to prevent unlimited looping. In these cases, the test generation system attempted to set the high bit of the counter to a '1'. Due to choosing a particularly poor set of values for justification,

backtracking became excessive and the loop cut mechanism aborted test generation. All other non-excluded faults were tested.

CNTRV

The CNTRV model is a three-bit counter described at the vector level. All tests were successful, good choices for justification values prevented excessive looping.

DFF

DFF is simply a D flip-flop with asynchronous set and clear. Not surprisingly, all tests were generated.

FNTST

The FNTST example is a simple gate-level model that demonstrates the E-algorithm's ability to handle reconvergent fanout. Tests for all faults were generated.

PRTY

The PRTY model describes an eight-bit parity generator at the gate-level. Tests for all faults were generated.

SHFT

Tests for this four-bit shift register were quite successful. Again, all

non-excluded faults were successfully processed.

STATE_MACHINE

The STATE_MACHINE model is a serial sequence detector. The implementation of the circuit contains 6 valid and 2 illegal states (combinations of register values). The circuit has extremely poor observability and controllability, with only one serial input, one output, one clear, and one clock. Tests for most faults were generated, though performance on some was very poor, but several failed due to attempts to justify illegal values in the state registers. When the algorithm tried to set the machine to an illegal state for a test, the justification procedure proceeded to set the machine to that state. This, of course, was futile, but there is no mechanism for detecting this problem. If loop checking was disabled, the justification promptly overflowed the system stack.

6.2 Summary

The current implementation of the E-algorithm performed extremely well on the test set of MSI circuits. Test generation performance was directly related to the controllability and observability of the circuit. In general, the more observable a circuit is, the easier it is for the propagation routine to find a path for the test. Similarly, circuits with high controllabilities are easy to justify, usually allowing the first propagation path found to be used.

Tests for some faults in complex sequential logic could not be generated. As shown by STATE_MACHINE, if the circuit under test has illegal states, the E-algorithm can loop indefinitely while trying to find a sequence to initialize to

an illegal state. This limitation arises due to a lack of high-level knowledge concerning the circuit. Suggestions to alleviate this problem are presented in Chapter 7.

Performance was in general acceptable. A test for the average fault took roughly 2 seconds to generate while running on an IBM AT. A fair amount of this time is overhead caused by initialization. Test generation times, even for 'hard' faults, were almost always less than 15 seconds. The primary reasons for these results are the guidance provided by the observability and controllability measures, rapid identification of situations requiring multiple path propagation, and the conflict diagnosis routine. Conflict diagnosis is particularly useful for improving performance. By identifying where irresolvable justification requirements arose, a large amount of futile backtracking on the propagation path was avoided.

Chapter 7
Analysis & Suggestions

The E-algorithm as currently implemented is a powerful test generation tool, but does contain certain deficiencies. This chapter reviews the primary weaknesses and suggests pŏssible solutions. Ideas for further extensions are presented.

7.1 Adding State Transition Knowledge

The primary deficiency of the E-algorithm is the inability to take advantage of state transition information. In generating tests for STATE_MACHINE, the problems inherent in testing complex sequential logic are evident. Mainly, the generation of sequences to put the circuit in a known state is a combinatorially explosive problem when approached without any high-level information. Moreover, the E-algorithm may attempt to generate a test using an illegal machine state. If this occurs, depending on the exact circuit structure, there may be no way for the algorithm to detect that the given state cannot be reached.

What is needed is some way to provide the E-algorithm with high-level guidance. One approach would be a *META-test* routine that monitors the execution of the E-algorithm. Whenever the E-algorithm must choose values for

circuit state variables, the META-test routine consults state-transition information and provides a set of values that satisfy the test generation requirements and don't produce illegal states. Also, homing sequences for circuit states could be stored by META-test and provided as complete justification plans to the E-algorithm whenever a state must be set. A META-test s · .n would greatly improve the performance of test generation for complex sequential circuits.

7.2 Don't Care Value Selection

The Justification and propagation procedures do not usually fully constrain the values of all the signals required for a test. Often, particularly for circuits containing complex operations, some of the bits of vecto values will be "X" 's - for "don't care" bits. These bits may be set to any values without affecting the test results. A possible further use for a META-test routine would be to choose values for these bits that will exercise more circuit logic or data lines. Cho [35] has proposed such a system that maintains a history of the values chosen for each signal line during test generation and sets don't care bits to values that haven't been used before. The effects of the instantiation of the bits are then simulated to produce any additional output values. The motivation for this procedure is the fact that behavioral fault models have a loose (at best) correspondence to the effects actual circuit defects can cause. Setting the don't care bits will exercise further combinations of circuit states and hopefully increase the probability that any faults will be detected, even if they cannot be easily modeled.

7.3 Forward Implication

Forward implication consists of substituting values chosen at each stage of propagation into the circuit to find what other signal values are implied by these choices. Forward implication is used in many test generation systems as a means to improve efficiency by early detection of value conflicts. The E-algorithm employs partial forward implication, values are moved through operation blocks to find any implied values, but implication is not done through data buffers. The problem with implication through data buffers is that often the other input of the buffer (either the control port or the data input) will not be specified, making the implication meaningless. In some cases, however, both values are available, and in these cases implication through data buffers could be performed to further improve efficiency.

7.4 Path Storage

Currently, the implementation of the E-algorithm generates a complete new path during test generation for each fault, without considering past propagation. An alternative would be to store 'propagation plans' which specify a propagation path to the output for each circuit element. Instead of generating a new path, the system could consult a *path library* to determine if any past propagation paths could be used. Unfortunately, generating, storing, and searching this library may be more expensive than its worth. A alternative would be to generate and store paths for only the least observable circuit elements. In these cases, the searching overhead will be less than the cost of finding the paths from scratch.

One other situation where propagation plans can clearly be efficiently applied

is the case where tests for a stuck-at-0 and then a stuck-at-1 fault on the same line are found. In almost all cases, the propagation path found during for the first fault will be applicable to the second as well. Thus, these paths should only be generated once, and simply followed the second time (though consistency checks must be made to insure no value conflicts arise).

Chapter 8
Conclusions

This thesis has presented the E-algorithm, a method for generating tests for general logic circuits modeled using a Hardware Description Language. A fault model based on faulting structures in a graph derived from the HDL model, and a test generation algorithm that reliably generates tests for the modeled faults have been developed and implemented. The E-algorithm has the ability to propagate fault syndromes through high-level control structures as well as data operations. Moreover, tests for sequential logic can be generated.

Preliminary results are highly encouraging, yielding reasonable performance. Limitations arise due to the lack of state transition knowledge, preventing test generation for some faults in complex sequential circuits. Future work will attempt test generation for larger circuits and examine the problem of embedding state-transition knowledge.

Bibliography

[1] J.P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method", *IBM Journal of Research and Development*, vol. 10, pp.278-291, July 1966.

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

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

[4] S.Y.H. Su and T. Lin, "Functional Testing Techniques for Digital LSI/VLSI Systems", *21st Design Automation Conference.*, pp 517-528, June 1984.

[5] W. Johnson, "Behavioral-Level Test Generation", *16th Design Automation Conference*, 1979, pp. 171-179.

[6] E.B. Eichelberger and T.W. Williams, "A Logic Design Structure for LSI Testability", *Journal of Design Automation and Fault-Tolerant Computers*,vol. 2 pp. 165-178, May 1978.

[7] M. Kawai, S. Funastu and A. Yamada, "Application of Shift Register Approach and its Effective Implementation", *1980 IEEE Test Conference*, 1980, pp. 22-25.

[8] "The GSP Manual", Dept. of E.E., VPI & SU, Blacksburg, Dec. 1982.

[9] *VHDL Language Reference Manual, Version 7.3*, IR-MD-045-2, September 1986.

[10] K.W. Chiang and Z. G. Vranesic, "On Fault Detection in CMOS Logic Networks", *20th Design Automation Conference*, 1983, pp. 50-56.

[11] H. Chen et al.,"An Algorithm to Generate Tests for MOS Circuits at the Switch Level", *1985 IEEE Test Conference*, 1985, pp. 304-312.

[12] C. Robach and G. Saucier, "Microprocessor Functional Testing,"*1980 IEEE Test Conference*, 1980.

[13] S. Thatte and J.A. Abraham, "Test Generation for Microprocessors", *IEEE Transactions on Computers*, vol C-29, June 1980, pp. 429-441.

[14] A. Hunger and A. Gaertner, "Functional Characterization of Microprocessors"

1984 IEEE Test Conference, 1984, pp. 794-803.

[15] M.A. Breuer and A.D. Friedman,"Functional Level Primitives in Test Generation", *IEEE Transactions on Computers*,vol C-29, March 1980, pp. 223-235.

[16] M.S. Abadir and H.K. Reghbati,"Functional Test Generation for Digital Circuits Described Using Binary Decision Diagrams", *IEEE Transactions on Computers*, vol C-35, April 1986, pp. 375-379.

[17] -----,"Functional Test Generation for LSI Circuits Described by Binary Decision Diagrams", *1985 IEEE Test Conference*, 1985, pp. 483-492.

[18] S. Shteingart et. al.,"RTG: Automatic Register Level Test Generator", *22nd Design Automation Conference*, 1985, pp. 803-807.

[19] R. Joobbani, "Functional-Level Fault Simulation", *Proc. ICCC, 1980, pp. 1120-1124.*

[20] B.M. Huey and F.J. Hill,"Fault Test Generation Using A Design Language" *Proc. Intl. Symp. CHDL*, 1975, pp. 91-95.

[21] C. Liaw, S.Y.H. Su, and Y.K. Malaiya,"State Diagram Approach for Functional Testing of Control Section", *1981 IEEE Test Conference*, 1981, pp. 433-446.

[22] T. Lin and S.Y.H. Su, "VLSI Functional Test Pattern Generation - A Design and Implementation", *1985 IEEE Test Conference*, 1985, pp. 922-929.

[23] ----,"The S-Algorithm: A Promising Solution for Systematic Functional Test Generation", *IEEE Transactions on Computer-Aided Design*, vol 4, July 1985, pp. 250-263.

[24] S.Y.H. Su and Y. Hsieh,"Testing Functional Faults in Digital Systems Described by Register Transfer Language", *1981 IEEE Test Conference*, 1981, pp. 447-457.

[25] M. Kawai et. al. "A High Level Test Pattern Generation Algorithm", *1983 IEEE Test Conference*, 1983, pp. 346-352.

[26] Y. Levendel and P. Menon, "Test Generation Algorithms for Computer Hardware Description Languages", *IEEE Transactions on Computers*, vol C-31, July 1982, pp. 577-588.

[27] K. Son and J.Y.O. Fong,"Automatic Behavioral Test Generation", *1982 IEEE Test Conference*, 1982, pp. 161-165.

[28] D. Barclay, "An Automatic Test Generation Method for Chip-Level Circuit Descriptions," Master's Thesis, Virginia Polytechnic Institute and State University, Jan. 1987.

[29] A.K. Gupta and J.R. Armstrong, "Functional Fault Modeling and Simulation for VLSI Devices", *22nd Design Automation Conference*, 1985, pp.

720-726.

[30] D. Barclay and J.R. Armstrong,"A Heuristic Chip-Level Test Generation Algorithm", *23rd Design Automation Conference*, 1986, pp. 257-262.

[31] M. D. O'Neill,"An Improved Chip-Level Test Generation Algorithm", Master's thesis, Virginia Polytechnic Institute and State University, Jan. 1988.

[32] C.H. Chao and F.G. Gray, "Micro-Operation Perturbations in Chip Level Fault Modeling", **to appear in** *Proc. 25th DAC*, July, 1988.

[33] J.Y.O. Fong, "On Functional Controllability and Observability Analysis", *Proc. Intl. Test Conf.*, 1982, pp. 170-175.

[34] D. Siewiorek and R. Swartz, *The Theory and Practice of Reliable System Design*, Digital Press, N.Y., 1982.

Appendix A

Introduction to User's Guide

This is the user's manual for a preliminary implementation of the E-algorithm. The algorithm has been implemented in Turbo Pascal 4.0 using the IBM PC-DOS operating system. The complete system comprises some 3000 lines of Pascal source code and requires a host computer with at least 512K bytes of memory. The implemented system is a menu-driven program called *EATPG*.

VHDL Subset Allowed

The implementation of the algorithm currently restricts the set of VHDL operations and structures which may be used to model circuits-under-test. The subset consists of non-procedural, data-flow operations. For a complete list of the allowed VHDL structures, please consult Chapter 3, VHDL Circuit Models.

Translating VHDL Models into Internal Form

VHDL circuit descriptions must be transformed into an internal representational format, the VHDL code cannot be used as a direct input at the present time.

The internal form used to describe circuit models consists of a linked-record data structure which describes the data and control flow for the circuit. Each input, output, internal signal, data operation or control buffer is assigned to a separate record. Each record specifies the number and width of the object inputs, outputs, and what operation the object performs. The inputs and outputs of each object are given as pointers in the record to the records corresponding to the appropriate items. Taken as a whole, the linked record representation embodies the structure of the graph representation used by the E-Algorithm.

The VHDL code is transformed into the internal form by identifying each input, output, internal signal, or data operation in the VHDL model, and then giving each a unique identifier, ex. AND3 or XOR6. Control statments are broken down into graph structures composed of data operations and signals which compute the boolean value corresponding to the evaluation of the control expression. Each of these operation blocks are then given identifiers, the result of the expression will be used to control the execution of signal assignments under the control statement. Each signal assignment in the VHDL model under a control statement will be represented by a control buffer record having two inputs. The first input is connected to the evaluation of the right hand side of the signal assignment, and the second input is connected to the evaluation of the control expression as described above. Once unique identifiers have been specified for each circuit object, the internal graph form of the circuit is built by entering each object name, giving it's type, eg. AND, XOR, ADD, etc., and specifying the

names of all its input and output objects and their widths.

An example of a very simple transformation is shown below. Here, a VHDL description of a vector-wide adder is tranformed:

```
    entity ADDER(
        A,
        B       : in bit_vector(3 downto 0);
        C       : out bit_vector(3 downto 0)
        ) is
     end ADDER;

    architecture ARCH of ADDER is

    process(A,B)
    begin
s1: C <= add(A,B)
    end block;
    end ADDER;
```

Internal Form:

```
Input #1 : ADD1    Width : 4
Object name :          A     Object Type : INPUT
Output Width : 4
Output #1 : ADD1

Object name :      ADD1      Object Type : ADD
Input #1 : A    Width : 4
Input #2 : B    Width : 4
Output Width : 4
Output #1 : C

Object name :          B     Object Type : INPUT
Output Width : 4
Output #1 : ADD1

Object name :          C     Object Type : OUTPUT
Input #1 : ADD1       Width : 4
Output Width : 4
```

Further examples of VHDL descriptions and their corresponding internal representations are found in Appendix B.

Running the Program

The main E-algorithm program, EATPG, is executed by typing 'EATPG' at the DOS prompt. The file 'EATPG.EXE' must be in the user's current working directory. Alternatively, the file 'EATPG.PAS' can be read into the Turbo Pascal 4.0 environment, and compiled and run using the compiler.

When the program is executed, the main menu will appear, giving all available options. This menu is given in Figure 11.

Circuit : *Circuit Name*

A - Add an object to circuit
C - Change (Edit) existing circuit
D - Display circuit
E - Execute test generation for all circuit faults
H - Help
I - Input circuit (or Add multiple objects to current)
F - Generate fault list
L - Load circuit from disk
O - Generate Controllability / Observability measures for circuit
P - Print circuit structure to document file
Q - Quit
R - Reset
S - Save circuit to disk
T - Generate test for a selected fault
U - Set trace level

Option: *Selected Function*

Figure 11. EATPG main menu

A - Add an object to circuit

This function allows additional circuit objects to be added to the circuit. The user is prompted for an object identifier. If an object with that name has already been entered, the user is asked if the object is to be modified. When adding a new object, the user is prompted for the object type, number of inputs, source of each input, output width, number of outputs, and output destination.

C - Change (Edit) existing circuit

The user is prompted for the name of an object to be modified. The record of the specified object is displayed, and the user can edit the information contained therein. The user may specify any number of objects to modify, and may also enter new objects.

D - Display Circuit

A shorthand form of the internal representation is displayed. The user may page through the circuit, or ask to see just the representation of certain objects.

E - Execute test generation for all faults

This function generates test vector sequences for all modeled faults, and reports any faults for which tests cannot be devised. The user is prompted for an output

file name '*Outfile*'. The test results are stored in the directory \Results under the name OUTFILE.RES. This file lists all modeled faults, the test sequence for each, and the time required to find each test.

This function first finds all modeled faults, checks if all specified circuit objects have been completely entered, and executes the pre-processor to find the controllability / observability measures if necessary. Then, the circuit faults are systematically considered and tests generated for each one. Any error conditions which may arise are reported and the user given the option of aborting test generation if an error does occur.

If a trace option has been enabled, the system will display (or print) intermediate results during test generation. See the trace option for more details.

I - Input circuit

This function allows the user to systematically enter a circuit description in the internal representation form. The user is prompted for object names and specifications. Any unspecified objects which are used as inputs or outputs to other objects are reported at the enter of circuit entry. The user is also prompted for the name of the circuit.

F - Generate fault list

This function generates a complete list of circuit faults based upon the fault model. The fault list may be displayed or printed to a file.

L - Load circuit from disk

This function allows a circuit model to be loaded from a disk file.

O - Generate Controllabilty / Observability measures

This function generates the observability and controllability measures for the circuit.

P - Print circuit structure

This option allows the user to print the circuit to a file or print device.

Q - Quit

Quit the system. The user will be prompted to store the circuit model.

R - Reset

The reset option erases the circuit model and resets the program.

S - Save circuit to disk

This function allows the user to save the circuit model to a disk file.

T - Generate test for a specified fault

This function allows the user to generate tests for a modeled circuit fault. The fault list may be displayed, or the user can enter the name of the object to be faulted, and select from a list of possible object faults. Intermediate results can be displayed if tracing is enabled and the test can be written to disk.

U - Set trace level

This option allows the user to specify the amount of intermediate results display during test generation. There are three levels of trace detail. Trace level 2 causes intermediate results to be displayed during the propagation and justification phases of test generation, all propagation conditions and all backtracking steps are displayed. Trace level 1 shows all object value assignments during test generation and identifies backtracking points. Trace level 0 disables tracing.

Modifying the Fault Model

The fault model may be modified by changing the faulting routines. For more information, please consult the commented source code.

Appendix B

Circuit Models and Fault Lists

This section contains the VHDL source listings, the abbreviated form of the internal representation, and the fault list with test results for each circuit model. Times are given in CPU seconds on an IBM AT. Faults marked as 'Excl' are faults in statements containing 'STABLE contructs which would cause either continuous clocking, or no clock transitions at all. These faults were excluded from consideration. Faults marked as '- Not a fault or not modeled' are faults such as a line containing a literal '1' being stuck-at-1.

ADDER Model

```
    entity ADDER(
        A,
        B       : in bit_vector(3 downto 0);
        C       : out bit_vector(3 downto 0)
        ) is
     end ADDER;

    architecture ARCH of ADDER is

    process(A,B)
    begin
s1: C <= add(A,B)
    end block;
    end ADDER;
```

ADDER
Object name : A Object Type : INPUT
Output Width : 4
Output #1 : ADD1

Object name : ADD1 Object Type : ADD
Input #1 : A Width : 4
Input #2 : B Width : 4
Output Width : 4
Output #1 : C

Object name : B Object Type : INPUT
Output Width : 4
Output #1 : ADD1

Object name : C Object Type : OUTPUT
Input #1 : ADD1 Width : 4
Output Width : 4

ADDER Fault List

Fault						Time(sec)
Fault #	1	Obj =	A	Output to :	ADD1 Stuck-at-zero	0.251
Fault #	2	Obj =	A	Output to :	ADD1 Stuck-at-one	0.233
Fault #	3	Obj =	ADD1	Fails to :	SUB	1.213
Fault #	4	Obj =	ADD1	Output to :	C Stuck-at-zero	0.719
Fault #	5	Obj =	ADD1	Output to :	C Stuck-at-one	0.789
Fault #	6	Obj =	B	Output to :	ADD1 Stuck-at-zero	0.870
Fault #	7	Obj =	B	Output to :	ADD1 Stuck-at-one	0.741

ADDR2 VHDL Model

```
      entity ADDR2
        (A1,A2,A3,A4,
        B1,B2,B3,B4,
        C0: in BIT;
        S1,S2,S3,S4,
        C4: out BIT) is
      end ADDR2;

      architecture ARCH of ADDR2 is

      process (A1,A2,A3,A4,B1,B2,B3,B4,C0,C1,C2,C3,C4,S2,S3)
      signal
          C1,
          C2,
          C3 : BIT;
      begin
s1:   S1 = A1 xor (B1 xor C0);
s2:   C1 = (A1 and B1) or (C0 and (A1 xor B1) ) ;
s3:   S2 = A2 xor (B2 xor C1);
s4:   C2 = (A2 and B2) or (C1 and (A2 xor B2) ) ;
s5:   S3 = A3 xor (B3 xor C2);
s6:   C3 = (A3 and B3) or (C2 and (A3 xor B3) ) ;
s7:   S4 = A4 xor (B4 xor C3);
s8:   C4 = (A4 and B4) or (C3 and (A4 xor B4) ) ;
       end block;
       end ARCH;
```

ADDR2
Object name : A1 Object Type : INPUT
Output Width : 1
Output #1 : XOR1
Output #2 : AND1
Output #3 : XOR3

Object name : XOR1 Object Type : XOR
Input #1 : A1 Width : 1
Input #2 : B1 Width : 1
Output Width : 1
Output #1 : XOR2

Object name : AND1 Object Type : AND
Input #1 : A1 Width : 1
Input #2 : B1 Width : 1
Output Width : 1
Output #1 : OR1

```
Object name :      XOR3     Object Type : XOR
Input #1 : A1    Width : 1
Input #2 : B1    Width : 1
Output Width : 1
Output #1 : AND2

Object name :        B1     Object Type : INPUT
Output Width : 1
Output #1 : XOR1
Output #2 : AND1
Output #3 : XOR3

Object name :      XOR2     Object Type : XOR
Input #1 : XOR1    Width : 1
Input #2 : C0    Width : 1
Output Width : 1
Output #1 : S1

Object name :       OR1     Object Type : OR
Input #1 : AND1    Width : 1
Input #2 : AND2    Width : 1
Output Width : 1
Output #1 : C1

Object name :      AND2     Object Type : AND
Input #1 : C0    Width : 1
Input #2 : XOR3    Width : 1
Output Width : 1
Output #1 : OR1

Object name :        C0     Object Type : INPUT
Output Width : 1
Output #1 : XOR2
Output #2 : AND2

Object name :        S1     Object Type : OUTPUT
Input #1 : XOR2    Width : 1
Output Width : 1

Object name :        C1     Object Type : SIGNAL
Input #1 : OR1    Width : 1
Output Width : 1
Output #1 : XOR4
Output #2 : AND4

Object name :      XOR4     Object Type : XOR
Input #1 : C1    Width : 1
Input #2 : B2    Width : 1
Output Width : 1
Output #1 : XOR5
```

```
Object name :      AND4     Object Type : AND
Input #1 : C1    Width : 1
Input #2 : XOR6    Width : 1
Output Width : 1
Output #1 : OR2

Object name :        A2     Object Type : INPUT
Output Width : 1
Output #1 : XOR5
Output #2 : AND3
Output #3 : XOR6

Object name :      XOR5     Object Type : XOR
Input #1 : XOR4    Width : 1
Input #2 : A2    Width : 1
Output Width : 1
Output #1 : S2

Object name :      AND3     Object Type : AND
Input #1 : A2    Width : 1
Input #2 : B2    Width : 1
Output Width : 1
Output #1 : OR2

Object name :      XOR6     Object Type : XOR
Input #1 : A2    Width : 1
Input #2 : B2    Width : 1
Output Width : 1
Output #1 : AND4

Object name :        S2     Object Type : OUTPUT
Input #1 : XOR5    Width : 1
Output Width : 0

Object name :        B2     Object Type : INPUT
Output Width : 1
Output #1 : XOR4
Output #2 : AND3
Output #3 : XOR6

Object name :       OR2     Object Type : OR
Input #1 : AND3    Width : 1
Input #2 : AND4    Width : 1
Output Width : 1
Output #1 : C2

Object name :        C2     Object Type : SIGNAL
Input #1 : OR2    Width : 1
Output Width : 1
Output #1 : XOR7
Output #2 : AND6
```

```
Object name :      XOR7     Object Type : XOR
Input #1 : C2    Width : 1
Input #2 : B3    Width : 1
Output Width : 1
Output #1 : XOR8

Object name :      AND6     Object Type : AND
Input #1 : C2    Width : 1
Input #2 : XOR9    Width : 1
Output Width : 1
Output #1 : OR3

Object name :        A3     Object Type : INPUT
Output Width : 1
Output #1 : XOR8
Output #2 : AND5
Output #3 : XOR9

Object name :      XOR8     Object Type : XOR
Input #1 : XOR7    Width : 1
Input #2 : A3    Width : 1
Output Width : 1
Output #1 : S3

Object name :      AND5     Object Type : AND
Input #1 : A3    Width : 1
Input #2 : B3    Width : 1
Output Width : 1
Output #1 : OR3

Object name :      XOR9     Object Type : XOR
Input #1 : A3    Width : 1
Input #2 : B3    Width : 1
Output Width : 1
Output #1 : AND6

Object name :        S3     Object Type : OUTPUT
Input #1 : XOR8    Width : 1
Output Width : 0

Object name :        B3     Object Type : INPUT
Output Width : 1
Output #1 : XOR7
Output #2 : AND5
Output #3 : XOR9

Object name :      OR3     Object Type : OR
Input #1 : AND5    Width : 1
Input #2 : AND6    Width : 1
Output Width : 1
Output #1 : C3
```

```
Object name :        C3     Object Type : SIGNAL
Input #1 : OR3    Width : 1
Output Width : 1
Output #1 : XOR10
Output #2 : AND8

Object name :     XOR10     Object Type : XOR
Input #1 : C3    Width : 1
Input #2 : B4    Width : 1
Output Width : 1
Output #1 : XOR11

Object name :      AND8     Object Type : AND
Input #1 : C3    Width : 1
Input #2 : XOR12    Width : 1
Output Width : 1
Output #1 : OR4

Object name :        A4     Object Type : INPUT
Output Width : 1
Output #1 : XOR11
Output #2 : AND7
Output #3 : XOR12

Object name :     XOR11     Object Type : XOR
Input #1 : XOR10    Width : 1
Input #2 : A4    Width : 1
Output Width : 1
Output #1 : S4

Object name :      AND7     Object Type : AND
Input #1 : A4    Width : 1
Input #2 : B4    Width : 1
Output Width : 1
Output #1 : OR4

Object name :     XOR12     Object Type : XOR
Input #1 : A4    Width : 1
Input #2 : B4    Width : 1
Output Width : 1
Output #1 : AND8

Object name :        S4     Object Type : OUTPUT
Input #1 : XOR11    Width : 1
Output Width : 0

Object name :        B4     Object Type : INPUT
Output Width : 1
Output #1 : XOR10
Output #2 : AND7
Output #3 : XOR12
```

```
Object name :       OR4     Object Type : OR
Input #1 : AND7    Width : 1
Input #2 : AND8    Width : 1
Output Width : 1
Output #1 : C4

Object name :       C4     Object Type : OUTPUT
Input #1 : OR4    Width : 1
Output Width : 1
```

ADDR2 Fault List

Fault			Time(sec)
Fault # 1 Obj =	A1 Output to :	XOR1 Stuck-at-zero	1.037
Fault # 2 Obj =	A1 Output to :	XOR1 Stuck-at-one	0.951
Fault # 3 Obj =	A1 Output to :	AND1 Stuck-at-zero	1.082
Fault # 4 Obj =	A1 Output to :	AND1 Stuck-at-one	0.885
Fault # 5 Obj =	A1 Output to :	XOR3 Stuck-at-zero	1.025
Fault # 6 Obj =	A1 Output to :	XOR3 Stuck-at-one	1.210
Fault # 7 Obj =	XOR1 Fails to :	EQUAL	1.728
Fault # 8 Obj =	XOR1 Output to :	XOR2 Stuck-at-zero	0.619
Fault # 9 Obj =	XOR1 Output to :	XOR2 Stuck-at-one	0.601
Fault # 10 Obj =	AND1 Fails to :	OR	1.406
Fault # 11 Obj =	AND1 Output to :	OR1 Stuck-at-zero	1.541
Fault # 12 Obj =	AND1 Output to :	OR1 Stuck-at-one	1.884
Fault # 13 Obj =	XOR3 Fails to :	EQUAL	1.074
Fault # 14 Obj =	XOR3 Output to :	AND2 Stuck-at-zero	1.882
Fault # 15 Obj =	XOR3 Output to :	AND2 Stuck-at-one	1.560
Fault # 16 Obj =	B1 Output to :	XOR1 Stuck-at-zero	1.683
Fault # 17 Obj =	B1 Output to :	XOR1 Stuck-at-one	1.703
Fault # 18 Obj =	B1 Output to :	AND1 Stuck-at-zero	1.779
Fault # 19 Obj =	B1 Output to :	AND1 Stuck-at-one	1.729
Fault # 20 Obj =	B1 Output to :	XOR3 Stuck-at-zero	1.743
Fault # 21 Obj =	B1 Output to :	XOR3 Stuck-at-one	1.665
Fault # 22 Obj =	XOR2 Fails to :	EQUAL	1.950
Fault # 23 Obj =	XOR2 Output to :	S1 Stuck-at-zero	0.834
Fault # 24 Obj =	XOR2 Output to :	S1 Stuck-at-one	0.873
Fault # 25 Obj =	OR1 Fails to :	AND	1.098
Fault # 26 Obj =	OR1 Output to :	C1 Stuck-at-zero	1.070
Fault # 27 Obj =	OR1 Output to :	C1 Stuck-at-one	1.040
Fault # 28 Obj =	AND2 Fails to :	OR	1.836
Fault # 29 Obj =	AND2 Output to :	OR1 Stuck-at-zero	0.813
Fault # 30 Obj =	AND2 Output to :	OR1 Stuck-at-one	0.846
Fault # 31 Obj =	C0 Output to :	XOR2 Stuck-at-zero	0.634
Fault # 32 Obj =	C0 Output to :	XOR2 Stuck-at-one	0.555
Fault # 33 Obj =	C0 Output to :	AND2 Stuck-at-zero	0.807
Fault # 34 Obj =	C0 Output to :	AND2 Stuck-at-one	0.803
Fault # 35 Obj =	C1 Output to :	XOR4 Stuck-at-zero	2.155
Fault # 36 Obj =	C1 Output to :	XOR4 Stuck-at-one	1.899
Fault # 37 Obj =	C1 Output to :	AND4 Stuck-at-zero	1.442
Fault # 38 Obj =	C1 Output to :	AND4 Stuck-at-one	1.532
Fault # 39 Obj =	XOR4 Fails to :	EQUAL	0.931
Fault # 40 Obj =	XOR4 Output to :	XOR5 Stuck-at-zero	1.965
Fault # 41 Obj =	XOR4 Output to :	XOR5 Stuck-at-one	2.126
Fault # 42 Obj =	AND4 Fails to :	OR	1.840
Fault # 43 Obj =	AND4 Output to :	OR2 Stuck-at-zero	0.977
Fault # 44 Obj =	AND4 Output to :	OR2 Stuck-at-one	0.973
Fault # 45 Obj =	A2 Output to :	XOR5 Stuck-at-zero	0.732
Fault # 46 Obj =	A2 Output to :	XOR5 Stuck-at-one	0.606
Fault # 47 Obj =	A2 Output to :	AND3 Stuck-at-zero	1.847

Fault # 48	Obj =	A2 Output to :	AND3 Stuck-at-one	1.761
Fault # 49	Obj =	A2 Output to :	XOR6 Stuck-at-zero	0.541
Fault # 50	Obj =	A2 Output to :	XOR6 Stuck-at-one	0.558
Fault # 51	Obj =	XOR5 Fails to :	EQUAL	0.812
Fault # 52	Obj =	XOR5 Output to :	S2 Stuck-at-zero	1.712
Fault # 53	Obj =	XOR5 Output to :	S2 Stuck-at-one	1.748
Fault # 54	Obj =	AND3 Fails to :	OR	1.228
Fault # 55	Obj =	AND3 Output to :	OR2 Stuck-at-zero	1.170
Fault # 56	Obj =	AND3 Output to :	OR2 Stuck-at-one	1.346
Fault # 57	Obj =	XOR6 Fails to :	EQUAL	1.993
Fault # 58	Obj =	XOR6 Output to :	AND4 Stuck-at-zero	1.831
Fault # 59	Obj =	XOR6 Output to :	AND4 Stuck-at-one	1.812
Fault # 60	Obj =	B2 Output to :	XOR4 Stuck-at-zero	0.662
Fault # 61	Obj =	B2 Output to :	XOR4 Stuck-at-one	0.731
Fault # 62	Obj =	B2 Output to :	AND3 Stuck-at-zero	2.003
Fault # 63	Obj =	B2 Output to :	AND3 Stuck-at-one	1.751
Fault # 64	Obj =	B2 Output to :	XOR6 Stuck-at-zero	1.342
Fault # 65	Obj =	B2 Output to :	XOR6 Stuck-at-one	1.402
Fault # 66	Obj =	OR2 Fails to :	AND	1.300
Fault # 67	Obj =	OR2 Output to :	C2 Stuck-at-zero	0.897
Fault # 68	Obj =	OR2 Output to :	C2 Stuck-at-one	0.787
Fault # 69	Obj =	C2 Output to :	XOR7 Stuck-at-zero	1.345
Fault # 70	Obj =	C2 Output to :	XOR7 Stuck-at-one	1.347
Fault # 71	Obj =	C2 Output to :	AND6 Stuck-at-zero	1.530
Fault # 72	Obj =	C2 Output to :	AND6 Stuck-at-one	1.613
Fault # 73	Obj =	XOR7 Fails to :	EQUAL	0.826
Fault # 74	Obj =	XOR7 Output to :	XOR8 Stuck-at-zero	1.376
Fault # 75	Obj =	XOR7 Output to :	XOR8 Stuck-at-one	1.153
Fault # 76	Obj =	AND6 Fails to :	OR	1.886
Fault # 77	Obj =	AND6 Output to :	OR3 Stuck-at-zero	1.833
Fault # 78	Obj =	AND6 Output to :	OR3 Stuck-at-one	1.409
Fault # 79	Obj =	A3 Output to :	XOR8 Stuck-at-zero	1.995
Fault # 80	Obj =	A3 Output to :	XOR8 Stuck-at-one	1.828
Fault # 81	Obj =	A3 Output to :	AND5 Stuck-at-zero	1.013
Fault # 82	Obj =	A3 Output to :	AND5 Stuck-at-one	0.940
Fault # 83	Obj =	A3 Output to :	XOR9 Stuck-at-zero	1.575
Fault # 84	Obj =	A3 Output to :	XOR9 Stuck-at-one	1.405
Fault # 85	Obj =	XOR8 Fails to :	EQUAL	0.918
Fault # 86	Obj =	XOR8 Output to :	S3 Stuck-at-zero	1.943
Fault # 87	Obj =	XOR8 Output to :	S3 Stuck-at-one	1.277
Fault # 88	Obj =	AND5 Fails to :	OR	1.690
Fault # 89	Obj =	AND5 Output to :	OR3 Stuck-at-zero	1.506
Fault # 90	Obj =	AND5 Output to :	OR3 Stuck-at-one	1.592
Fault # 91	Obj =	XOR9 Fails to :	EQUAL	1.855
Fault # 92	Obj =	XOR9 Output to :	AND6 Stuck-at-zero	1.527
Fault # 93	Obj =	XOR9 Output to :	AND6 Stuck-at-one	1.737
Fault # 94	Obj =	B3 Output to :	XOR7 Stuck-at-zero	1.243
Fault # 95	Obj =	B3 Output to :	XOR7 Stuck-at-one	1.076
Fault # 96	Obj =	B3 Output to :	AND5 Stuck-at-zero	0.688
Fault # 97	Obj =	B3 Output to :	AND5 Stuck-at-one	0.644
Fault # 98	Obj =	B3 Output to :	XOR9 Stuck-at-zero	1.629
Fault # 99	Obj =	B3 Output to :	XOR9 Stuck-at-one	1.927
Fault #100	Obj =	OR3 Fails to :	AND	1.124

Fault #101	Obj =	OR3	Output to :	C3	Stuck-at-zero 1.006
Fault #102	Obj =	OR3	Output to :	C3	Stuck-at-one 0.885
Fault #103	Obj =	C3	Output to :	XOR10	Stuck-at-zero 1.367
Fault #104	Obj =	C3	Output to :	XOR10	Stuck-at-one 1.206
Fault #105	Obj =	C3	Output to :	AND8	Stuck-at-zero 1.409
Fault #106	Obj =	C3	Output to :	AND8	Stuck-at-one 1.329
Fault #107	Obj =	XOR10	Fails to :	EQUAL	1.873
Fault #108	Obj =	XOR10	Output to :	XOR11	Stuck-at-zero 1.693
Fault #109	Obj =	XOR10	Output to :	XOR11	Stuck-at-one 1.666
Fault #110	Obj =	AND8	Fails to :	OR	1.418
Fault #111	Obj =	AND8	Output to :	OR4	Stuck-at-zero 1.102
Fault #112	Obj =	AND8	Output to :	OR4	Stuck-at-one 1.220
Fault #113	Obj =	A4	Output to :	XOR11	Stuck-at-zero 1.403
Fault #114	Obj =	A4	Output to :	XOR11	Stuck-at-one 2.180
Fault #115	Obj =	A4	Output to :	AND7	Stuck-at-zero 1.983
Fault #116	Obj =	A4	Output to :	AND7	Stuck-at-one 1.861
Fault #117	Obj =	A4	Output to :	XOR12	Stuck-at-zero 1.632
Fault #118	Obj =	A4	Output to :	XOR12	Stuck-at-one 1.817
Fault #119	Obj =	XOR11	Fails to :	EQUAL	1.835
Fault #120	Obj =	XOR11	Output to :	S4	Stuck-at-zero 2.011
Fault #121	Obj =	XOR11	Output to :	S4	Stuck-at-one 1.811
Fault #122	Obj =	AND7	Fails to :	OR	0.914
Fault #123	Obj =	AND7	Output to :	OR4	Stuck-at-zero 0.853
Fault #124	Obj =	AND7	Output to :	OR4	Stuck-at-one 0.761
Fault #125	Obj =	XOR12	Fails to :	EQUAL	1.144
Fault #126	Obj =	XOR12	Output to :	AND8	Stuck-at-zero 0.693
Fault #127	Obj =	XOR12	Output to :	AND8	Stuck-at-one 0.658
Fault #128	Obj =	B4	Output to :	XOR10	Stuck-at-zero 1.648
Fault #129	Obj =	B4	Output to :	XOR10	Stuck-at-one 1.516
Fault #130	Obj =	B4	Output to :	AND7	Stuck-at-zero 1.622
Fault #131	Obj =	B4	Output to :	AND7	Stuck-at-one 1.840
Fault #132	Obj =	B4	Output to :	XOR12	Stuck-at-zero 1.790
Fault #133	Obj =	B4	Output to :	XOR12	Stuck-at-one 1.424
Fault #134	Obj =	OR4	Fails to :	AND	1.806
Fault #135	Obj =	OR4	Output to :	C4	Stuck-at-zero 1.994
Fault #136	Obj =	OR4	Output to :	C4	Stuck-at-one 1.763

CCNT2 Model

```
        entity CONTROLLED_CTR(
          CLK,
          STRB : in BIT;
          CON : in BIT_VECTOR(1 downto 0);
          DATA : in BIT_VECTOR(1 downto 0);
          COUNT : out BIT_VECTOR(1 downto 0)) is
        end controlled_ctr;

        architecture arch of controlled_ctr is

        process (CLK,STRB,CON,DATA,CONSIG,COUNT,LIM);
        signal
          EN       : BOOLEAN; ??
          DIR      : ?? (up,down)
          LIM      : BITVECTOR(1 downto 0) ;
          LOADFLAG  : BOOLEAN

        begin
s01:    if STRB='1' and not STRB'STABLE then
s02:       case intval(CON) is
              when 0 =>
s03:           COUNT <= "00";
              when 1 =>
s04:           LOADFLAG <= true;
              when 2 =>
s05:           EN <= true;
s06:           DIR <= up;
              when 3 =>
s07:           EN <= true;
s08:           DIR <= down;
           end case;
        end if;
s09:    if (STRB='0' and not STRB'STABLE) and LOADFLAG then
s10:       LIM <= DATA;
s11:       LOADFLAG <= false;
        end if;
s12:    if (CLK='1' and not CLK'STABLE) and EN then
s13:       if DIR=up then
s14:          COUNT <= add(COUNT,"01");
           else
s15:          COUNT <= sub(COUNT,"01");
           end if;
        end if;
s16:    if COUNT = LIM then
s17:       EN <= false;
        end if;
        end process;
        end arch;
```

CCNT2

```
Object name :       CLK     Object Type : INPUT
Output Width : 1
Output #1 : EQ7
Output #2 : STB3

Object name :       EQ7     Object Type : EQUAL
Input #1 : CLK    Width : 1
Input #2 : LIT14    Width : 1
Output Width : 1
Output #1 : AND8

Object name :      STB3     Object Type : STABLE
Input #1 : CLK    Width : 1
Output Width : 1
Output #1 : AND8

Object name :     LIT14     Object Type : LIT:1
Output Width : 1
Output #1 : EQ7

Object name :      AND8     Object Type : AND
Input #1 : EQ7    Width : 1
Input #2 : STB3    Width : 1
Output Width : 1
Output #1 : AND9

Object name :      STRB     Object Type : INPUT
Output Width : 1
Output #1 : EQ1
Output #2 : STB1
Output #3 : EQ6
Output #4 : STB2

Object name :       EQ1     Object Type : EQUAL
Input #1 : STRB    Width : 1
Input #2 : LIT1    Width : 1
Output Width : 1
Output #1 : AND1

Object name :      STB1     Object Type : STABLE
Input #1 : STRB    Width : 1
Output Width : 1
Output #1 : AND6

Object name :       EQ6     Object Type : EQUAL
Input #1 : STRB    Width : 1
Input #2 : LIT12    Width : 1
Output Width : 1
Output #1 : AND6
```

```
Object name :      STB2     Object Type : STABLE
Input #1 : STRB    Width : 1
Output Width : 1
Output #1 : AND6

Object name :      LIT1     Object Type : LIT:1
Output Width : 1
Output #1 : EQ1

Object name :      AND1     Object Type : AND
Input #1 : EQ1     Width : 1
Input #2 : STB1    Width : 1
Output Width : 1
Output #1 : AND2
Output #2 : AND3
Output #3 : AND4
Output #4 : AND5

Object name :      AND6     Object Type : AND
Input #1 : EQ6     Width : 1
Input #2 : STB2    Width : 1
Output Width : 1
Output #1 : AND7

Object name :     LIT12     Object Type : LIT:0
Output Width : 1
Output #1 : EQ6

Object name :      AND2     Object Type : AND
Input #1 : EQ2     Width : 1
Input #2 : AND1    Width : 1
Output Width : 1
Output #1 : BUF1

Object name :      AND3     Object Type : AND
Input #1 : EQ3     Width : 1
Input #2 : AND1    Width : 1
Output Width : 1
Output #1 : BUF2

Object name :      AND4     Object Type : AND
Input #1 : EQ4     Width : 1
Input #2 : AND1    Width : 1
Output Width : 1
Output #1 : BUF3
Output #2 : BUF4

Object name :      AND5     Object Type : AND
Input #1 : EQ5     Width : 1
Input #2 : AND1    Width : 1
Output Width : 1
Output #1 : BUF5
Output #2 : BUF6
```

```
Object name :       CON     Object Type : INPUT
Output Width : 2
Output #1 : EQ2
Output #2 : EQ3
Output #3 : EQ4
Output #4 : EQ5

Object name :       EQ2     Object Type : EQUAL
Input #1 : CON    Width : 2
Input #2 : LIT2   Width : 2
Output Width : 1
Output #1 : AND2

Object name :       EQ3     Object Type : EQUAL
Input #1 : CON    Width : 2
Input #2 : LIT3   Width : 2
Output Width : 1
Output #1 : AND3

Object name :       EQ4     Object Type : EQUAL
Input #1 : CON    Width : 2
Input #2 : LIT4   Width : 2
Output Width : 1
Output #1 : AND4

Object name :       EQ5     Object Type : EQUAL
Input #1 : CON    Width : 2
Input #2 : LIT5   Width : 2
Output Width : 1
Output #1 : AND5

Object name :      LIT2     Object Type : LIT:00
Output Width : 2
Output #1 : EQ2

Object name :      LIT3     Object Type : LIT:01
Output Width : 2
Output #1 : EQ3

Object name :      LIT4     Object Type : LIT:10
Output Width : 2
Output #1 : EQ4

Object name :      LIT5     Object Type : LIT:11
Output Width : 2
Output #1 : EQ5

Object name :      BUF1     Object Type : BUFFER
Input #1 : LIT6   Width : 2
Input #2 : AND2    Width : 1
Output Width : 2
Output #1 : COUNT
```

```
Object name :      DATA     Object Type : INPUT
Output Width : 2
Output #1 : BUF7

Object name :      BUF7     Object Type : BUFFER
Input #1 : DATA    Width : 2
Input #2 : AND7    Width : 1
Output Width : 2
Output #1 : LIM

Object name :      AND7     Object Type : AND
Input #1 : AND6    Width : 1
Input #2 : LOADFLAG    Width : 1
Output Width : 1
Output #1 : BUF7
Output #2 : BUF8

Object name :      LIM      Object Type : SIGNAL
Input #1 : BUF7    Width : 2
Output Width : 2
Output #1 : EQ8

Object name :    COUNT      Object Type : OUTPUT
Input #1 : BUF1    Width : 2
Input #2 : BUF9    Width : 2
Input #3 : BUF10    Width : 2
Output Width : 2
Output #1 : ADD1
Output #2 : SUB1
Output #3 : EQ8

Object name :      BUF9     Object Type : BUFFER
Input #1 : ADD1    Width : 2
Input #2 : AND10    Width : 1
Output Width : 2
Output #1 : COUNT

Object name :    BUF10      Object Type : BUFFER
Input #1 : SUB1    Width : 2
Input #2 : NOT1    Width : 1
Output Width : 2
Output #1 : COUNT

Object name :      ADD1     Object Type : ADD
Input #1 : COUNT    Width : 2
Input #2 : LIT15    Width : 2
Output Width : 2
Output #1 : BUF9
```

```
Object name :      SUB1     Object Type : SUB
Input #1 : COUNT    Width : 2
Input #2 : LIT16    Width : 2
Output Width : 2
Output #1 : BUF10

Object name :       EQ8     Object Type : EQUAL
Input #1 : COUNT    Width : 2
Input #2 : LIM    Width : 2
Output Width : 1
Output #1 : BUF11

Object name :      LIT6     Object Type : LIT:00
Output Width : 2
Output #1 : BUF1

Object name :     AND10     Object Type : AND
Input #1 : DIR    Width : 1
Input #2 : AND9    Width : 1
Output Width : 1
Output #1 : BUF9
Output #2 : NOT1

Object name :      NOT1     Object Type : NOT
Input #1 : AND10    Width : 1
Output Width : 1
Output #1 : BUF10

Object name :     LIT15     Object Type : LIT:01
Output Width : 2
Output #1 : ADD1

Object name :     LIT16     Object Type : LIT:01
Output Width : 2
Output #1 : SUB1

Object name :     BUF11     Object Type : BUFFER
Input #1 : LIT17    Width : 1
Input #2 : EQ8    Width : 1
Output Width : 1
Output #1 : EN

Object name :       EN     Object Type : SIGNAL
Input #1 : BUF3    Width : 1
Input #2 : BUF5    Width : 1
Input #3 : BUF11    Width : 1
Output Width : 1
Output #1 : AND9
```

```
Object name :      BUF3     Object Type : BUFFER
Input #1 : LIT8    Width : 1
Input #2 : AND4     Width : 1
Output Width : 1
Output #1 : EN

Object name :      BUF5     Object Type : BUFFER
Input #1 : LIT10    Width : 1
Input #2 : AND5     Width : 1
Output Width : 1
Output #1 : EN

Object name :      AND9     Object Type : AND
Input #1 : AND8     Width : 1
Input #2 : EN     Width : 1
Output Width : 1
Output #1 : AND10

Object name :      LIT8     Object Type : LIT:1
Output Width : 1
Output #1 : BUF3

Object name :     LIT10     Object Type : LIT:1
Output Width : 1
Output #1 : BUF5

Object name :     LIT17     Object Type : LIT:0
Output Width : 1
Output #1 : BUF11

Object name :       DIR     Object Type : SIGNAL
Input #1 : BUF4    Width : 1
Input #2 : BUF6    Width : 1
Output Width : 1
Output #1 : AND10

Object name :      BUF4     Object Type : BUFFER
Input #1 : LIT9    Width : 1
Input #2 : AND4     Width : 1
Output Width : 1
Output #1 : DIR

Object name :      BUF6     Object Type : BUFFER
Input #1 : LIT11    Width : 1
Input #2 : AND5     Width : 1
Output Width : 1
Output #1 : DIR

Object name :      LIT9     Object Type : LIT:1
Output Width : 1
Output #1 : BUF4
```

```
Object name :     LIT11     Object Type : LIT:0
Output Width : 1
Output #1 : BUF6

Object name : LOADFLAG     Object Type : SIGNAL
Input #1 : BUF2    Width : 1
Input #2 : BUF8    Width : 1
Output Width : 1
Output #1 : AND7

Object name :     BUF2     Object Type : BUFFER
Input #1 : LIT7    Width : 1
Input #2 : AND3    Width : 1
Output Width : 1
Output #1 : LOADFLAG

Object name :     BUF8     Object Type : BUFFER
Input #1 : LIT13    Width : 1
Input #2 : AND7    Width : 1
Output Width : 1
Output #1 : LOADFLAG

Object name :     LIT7     Object Type : LIT:1
Output Width : 1
Output #1 : BUF2

Object name :     LIT13     Object Type : LIT:0
Output Width : 1
Output #1 : BUF8
```

CCNT2 Fault List

Fault					Time(sec)
Fault #	1	Obj =	CLK Output to :	EQ7 Stuck-at-zero	Excl
Fault #	2	Obj =	CLK Output to :	EQ7 Stuck-at-one	Excl
Fault #	3	Obj =	CLK Output to :	STB3 Stuck-at-zero	Excl
Fault #	4	Obj =	CLK Output to :	STB3 Stuck-at-one	Excl
Fault #	5	Obj =	EQ7 Fails to :	XOR	Excl
Fault #	6	Obj =	EQ7 Output to :	AND8 Stuck-at-zero	Excl
Fault #	7	Obj =	EQ7 Output to :	AND8 Stuck-at-one	Excl
Fault #	8	Obj =	LIT14 Output to :	EQ7 Stuck-at-zero	Excl
Fault #	9	Obj =	LIT14 Output to :	EQ7 Stuck-at-one	NAF
Fault #	10	Obj =	AND8 Fails to :	OR	Excl
Fault #	11	Obj =	AND8 Output to :	AND9 Stuck-at-zero	Excl
Fault #	12	Obj =	AND8 Output to :	AND9 Stuck-at-one	Excl
Fault #	13	Obj =	STRB Output to :	EQ1 Stuck-at-zero	Excl
Fault #	14	Obj =	STRB Output to :	EQ1 Stuck-at-one	Excl
Fault #	15	Obj =	STRB Output to :	STB1 Stuck-at-zero	Excl
Fault #	16	Obj =	STRB Output to :	STB1 Stuck-at-one	Excl
Fault #	17	Obj =	STRB Output to :	EQ6 Stuck-at-zero	Excl
Fault #	18	Obj =	STRB Output to :	EQ6 Stuck-at-one	Excl
Fault #	19	Obj =	STRB Output to :	STB2 Stuck-at-zero	Excl
Fault #	20	Obj =	STRB Output to :	STB2 Stuck-at-one	Excl
Fault #	21	Obj =	EQ1 Fails to :	XOR	Excl
Fault #	22	Obj =	EQ1 Output to :	AND1 Stuck-at-zero	Excl
Fault #	23	Obj =	EQ1 Output to :	AND1 Stuck-at-one	Excl
Fault #	24	Obj =	EQ6 Fails to :	XOR	Excl
Fault #	25	Obj =	EQ6 Output to :	AND6 Stuck-at-zero	Excl
Fault #	26	Obj =	EQ6 Output to :	AND6 Stuck-at-one	Excl
Fault #	27	Obj =	LIT1 Output to :	EQ1 Stuck-at-zero	Excl
Fault #	28	Obj =	LIT1 Output to :	EQ1 Stuck-at-one	NAF
Fault #	29	Obj =	AND1 Fails to :	OR	Excl
Fault #	30	Obj =	AND1 Output to :	AND2 Stuck-at-zero	Excl
Fault #	31	Obj =	AND1 Output to :	AND2 Stuck-at-one	Excl
Fault #	32	Obj =	AND1 Output to :	AND3 Stuck-at-zero	Excl
Fault #	33	Obj =	AND1 Output to :	AND3 Stuck-at-one	Excl
Fault #	34	Obj =	AND1 Output to :	AND4 Stuck-at-zero	Excl
Fault #	35	Obj =	AND1 Output to :	AND4 Stuck-at-one	Excl
Fault #	36	Obj =	AND1 Output to :	AND5 Stuck-at-zero	Excl
Fault #	37	Obj =	AND1 Output to :	AND5 Stuck-at-one	Excl
Fault #	38	Obj =	AND6 Fails to :	OR	Excl
Fault #	39	Obj =	AND6 Output to :	AND7 Stuck-at-zero	Excl
Fault #	40	Obj =	AND6 Output to :	AND7 Stuck-at-one	Excl
Fault #	41	Obj =	LIT12 Output to :	EQ6 Stuck-at-zero	NAF
Fault #	42	Obj =	LIT12 Output to :	EQ6 Stuck-at-one	4.629
Fault #	43	Obj =	AND2 Fails to :	OR	5.026
Fault #	44	Obj =	AND2 Output to :	BUF1 Stuck-at-zero	4.068
Fault #	45	Obj =	AND2 Output to :	BUF1 Stuck-at-one	3.937
Fault #	46	Obj =	AND3 Fails to :	OR	10.103
Fault #	47	Obj =	AND3 Output to :	BUF2 Stuck-at-zero	6.225

Fault # 48	Obj =	AND3 Output to :	BUF2 Stuck-at-one	5.109
Fault # 49	Obj =	AND4 Fails to :	OR	3.539
Fault # 50	Obj =	AND4 Output to :	BUF3 Stuck-at-zero	3.639
Fault # 51	Obj =	AND4 Output to :	BUF3 Stuck-at-one	3.940
Fault # 52	Obj =	AND4 Output to :	BUF4 Stuck-at-zero	3.121
Fault # 53	Obj =	AND4 Output to :	BUF4 Stuck-at-one	2.789
Fault # 54	Obj =	AND5 Fails to :	OR	3.694
Fault # 55	Obj =	AND5 Output to :	BUF5 Stuck-at-zero	3.567
Fault # 56	Obj =	AND5 Output to :	BUF5 Stuck-at-one	2.375
Fault # 57	Obj =	AND5 Output to :	BUF6 Stuck-at-zero	2.137
Fault # 58	Obj =	AND5 Output to :	BUF6 Stuck-at-one	2.238
Fault # 59	Obj =	CON Output to :	EQ2 Stuck-at-zero	2.488
Fault # 60	Obj =	CON Output to :	EQ2 Stuck-at-one	2.233
Fault # 61	Obj =	CON Output to :	EQ3 Stuck-at-zero	3.122
Fault # 62	Obj =	CON Output to :	EQ3 Stuck-at-one	3.446
Fault # 63	Obj =	CON Output to :	EQ4 Stuck-at-zero	3.765
Fault # 64	Obj =	CON Output to :	EQ4 Stuck-at-one	3.839
Fault # 65	Obj =	CON Output to :	EQ5 Stuck-at-zero	6.432
Fault # 66	Obj =	CON Output to :	EQ5 Stuck-at-one	6.439
Fault # 67	Obj =	EQ2 Fails to :	XOR	1.458
Fault # 68	Obj =	EQ2 Output to :	AND2 Stuck-at-zero	6.039
Fault # 69	Obj =	EQ2 Output to :	AND2 Stuck-at-one	5.374
Fault # 70	Obj =	EQ3 Fails to :	XOR	5.496
Fault # 71	Obj =	EQ3 Output to :	AND3 Stuck-at-zero	8.011
Fault # 72	Obj =	EQ3 Output to :	AND3 Stuck-at-one	7.770
Fault # 73	Obj =	EQ4 Fails to :	XOR	8.121
Fault # 74	Obj =	EQ4 Output to :	AND4 Stuck-at-zero	5.686
Fault # 75	Obj =	EQ4 Output to :	AND4 Stuck-at-one	5.925
Fault # 76	Obj =	EQ5 Fails to :	XOR	2.887
Fault # 77	Obj =	EQ5 Output to :	AND5 Stuck-at-zero	3.511
Fault # 78	Obj =	EQ5 Output to :	AND5 Stuck-at-one	3.370
Fault # 79	Obj =	LIT2 Output to :	EQ2 Stuck-at-zero	NAF
Fault # 80	Obj =	LIT2 Output to :	EQ2 Stuck-at-one	5.915
Fault # 81	Obj =	LIT3 Output to :	EQ3 Stuck-at-zero	NAF
Fault # 82	Obj =	LIT3 Output to :	EQ3 Stuck-at-one	6.521
Fault # 83	Obj =	LIT4 Output to :	EQ4 Stuck-at-zero	1.878
Fault # 84	Obj =	LIT4 Output to :	EQ4 Stuck-at-one	NAF
Fault # 85	Obj =	LIT5 Output to :	EQ5 Stuck-at-zero	3.686
Fault # 86	Obj =	LIT5 Output to :	EQ5 Stuck-at-one	NAF
Fault # 87	Obj =	BUF1 Output to :	COUNT Stuck-at-zero	1.652
Fault # 88	Obj =	BUF1 Output to :	COUNT Stuck-at-one	0.923
Fault # 89	Obj =	DATA Output to :	BUF7 Stuck-at-zero	5.226
Fault # 90	Obj =	DATA Output to :	BUF7 Stuck-at-one	5.020
Fault # 91	Obj =	BUF7 Output to :	LIM Stuck-at-zero	1.814
Fault # 92	Obj =	BUF7 Output to :	LIM Stuck-at-one	2.057
Fault # 93	Obj =	AND7 Fails to :	OR	Excl
Fault # 94	Obj =	AND7 Output to :	BUF7 Stuck-at-zero	Excl
Fault # 95	Obj =	AND7 Output to :	BUF7 Stuck-at-one	Excl
Fault # 96	Obj =	AND7 Output to :	BUF8 Stuck-at-zero	Excl
Fault # 97	Obj =	AND7 Output to :	BUF8 Stuck-at-one	Excl
Fault # 98	Obj =	LIM Output to :	EQ8 Stuck-at-zero	4.019
Fault # 99	Obj =	LIM Output to :	EQ8 Stuck-at-one	3.473
Fault #100	Obj =	BUF9 Output to :	COUNT Stuck-at-zero	2.485

Fault #101	Obj =	BUF9 Output to :	COUNT Stuck-at-one	2.219
Fault #102	Obj =	BUF10 Output to :	COUNT Stuck-at-zero	5.402
Fault #103	Obj =	BUF10 Output to :	COUNT Stuck-at-one	6.075
Fault #104	Obj =	ADD1 Fails to :	SUB	5.180
Fault #105	Obj =	ADD1 Output to :	BUF9 Stuck-at-zero	6.336
Fault #106	Obj =	ADD1 Output to :	BUF9 Stuck-at-one	6.344
Fault #107	Obj =	SUB1 Fails to :	ADD	4.894
Fault #108	Obj =	SUB1 Output to :	BUF10 Stuck-at-zero	5.771
Fault #109	Obj =	SUB1 Output to :	BUF10 Stuck-at-one	6.438
Fault #110	Obj =	EQ8 Fails to :	XOR	8.501
Fault #111	Obj =	EQ8 Output to :	BUF11 Stuck-at-zero	7.395
Fault #112	Obj =	EQ8 Output to :	BUF11 Stuck-at-one	7.352
Fault #113	Obj =	LIT6 Output to :	BUF1 Stuck-at-zero	NAF
Fault #114	Obj =	LIT6 Output to :	BUF1 Stuck-at-one	1.886
Fault #115	Obj =	AND10 Fails to :	OR	7.988
Fault #116	Obj =	AND10 Output to :	BUF9 Stuck-at-zero	3.396
Fault #117	Obj =	AND10 Output to :	BUF9 Stuck-at-one	3.303
Fault #118	Obj =	AND10 Output to :	NOT1 Stuck-at-zero	5.583
Fault #119	Obj =	AND10 Output to :	NOT1 Stuck-at-one	5.688
Fault #120	Obj =	NOT1 Fails to :	FBUF	9.495
Fault #121	Obj =	NOT1 Output to :	BUF10 Stuck-at-zero	4.209
Fault #122	Obj =	NOT1 Output to :	BUF10 Stuck-at-one	4.030
Fault #123	Obj =	LIT15 Output to :	ADD1 Stuck-at-zero	NAF
Fault #124	Obj =	LIT15 Output to :	ADD1 Stuck-at-one	6.771
Fault #125	Obj =	LIT16 Output to :	SUB1 Stuck-at-zero	NAF
Fault #126	Obj =	LIT16 Output to :	SUB1 Stuck-at-one	3.155
Fault #127	Obj =	BUF11 Output to :	EN Stuck-at-zero	4.858
Fault #128	Obj =	BUF11 Output to :	EN Stuck-at-one	3.705
Fault #129	Obj =	EN Output to :	AND9 Stuck-at-zero	6.609
Fault #130	Obj =	EN Output to :	AND9 Stuck-at-one	7.355
Fault #131	Obj =	BUF3 Output to :	EN Stuck-at-zero	4.421
Fault #132	Obj =	BUF3 Output to :	EN Stuck-at-one	4.105
Fault #133	Obj =	BUF5 Output to :	EN Stuck-at-zero	3.605
Fault #134	Obj =	BUF5 Output to :	EN Stuck-at-one	3.084
Fault #135	Obj =	AND9 Fails to :	OR	Excl
Fault #136	Obj =	AND9 Output to :	AND10 Stuck-at-zero	Excl
Fault #137	Obj =	AND9 Output to :	AND10 Stuck-at-one	Excl
Fault #138	Obj =	LIT8 Output to :	BUF3 Stuck-at-zero	2.026
Fault #139	Obj =	LIT8 Output to :	BUF3 Stuck-at-one	NAF
Fault #140	Obj =	LIT10 Output to :	BUF5 Stuck-at-zero	4.777
Fault #141	Obj =	LIT10 Output to :	BUF5 Stuck-at-one	NAF
Fault #142	Obj =	LIT17 Output to :	BUF11 Stuck-at-zero	NAF
Fault #143	Obj =	LIT17 Output to :	BUF11 Stuck-at-one	2.942
Fault #144	Obj =	DIR Output to :	AND10 Stuck-at-zero	4.329
Fault #145	Obj =	DIR Output to :	AND10 Stuck-at-one	4.737
Fault #146	Obj =	BUF4 Output to :	DIR Stuck-at-zero	3.072
Fault #147	Obj =	BUF4 Output to :	DIR Stuck-at-one	3.447
Fault #148	Obj =	BUF6 Output to :	DIR Stuck-at-zero	2.842
Fault #149	Obj =	BUF6 Output to :	DIR Stuck-at-one	2.477
Fault #150	Obj =	LIT9 Output to :	BUF4 Stuck-at-zero	2.874
Fault #151	Obj =	LIT9 Output to :	BUF4 Stuck-at-one	NAF
Fault #152	Obj =	LIT11 Output to :	BUF6 Stuck-at-zero	NAF
Fault #153	Obj =	LIT11 Output to :	BUF6 Stuck-at-one	2.144

Fault #154	Obj =	LOADFLAG	Output to :	AND7 Stuck-at-zero	4.736
Fault #155	Obj =	LOADFLAG	Output to :	AND7 Stuck-at-one	4.281
Fault #156	Obj =	BUF2	Output to :	LOADFLAG Stuck-at-zero	5.624
Fault #157	Obj =	BUF2	Output to :	LOADFLAG Stuck-at-one	6.451
Fault #158	Obj =	BUF8	Output to :	LOADFLAG Stuck-at-zero	6.352
Fault #159	Obj =	BUF8	Output to :	LOADFLAG Stuck-at-one	7.495
Fault #160	Obj =	LIT7	Output to :	BUF2 Stuck-at-zero	1.987
Fault #161	Obj =	LIT7	Output to :	BUF2 Stuck-at-one	NAF
Fault #162	Obj =	LIT13	Output to :	BUF8 Stuck-at-zero	NAF
Fault #163	Obj =	LIT13	Output to :	BUF8 Stuck-at-one	2.017

CKTA Model

```
        entity CIRCUITA is
          (DATAIN,
          CLOCK1,
          CLOCK2 : in BIT;
          ANDOUT : out BIT)
        end CIRCUITA;

        architecture ARCH of CIRCUITA is

        process(clock1,CLOCK2,Q1,Q2)
        signal
         Q1,
         Q2 : BIT;

        begin
s1:    if (clock1='1' and notCLOCK1'STABLE) then
s2:       Q1 <= DATAIN;
s3:    if (CLOCK2='1' and not CLOCK2'STABLE) then
s4:       Q2 <= Q1;
s5:       ANDOUT <= Q1 and Q2;
        end process;
        end arch;
```

CKTA

```
Object name :    DATAIN      Object Type : INPUT
Output Width : 1
Output #1 : BUF1

Object name :      BUF1      Object Type : BUFFER
Input #1 : DATAIN    Width : 1
Input #2 : AND1      Width : 1
Output Width : 1
Output #1 : Q1

Object name :      AND1      Object Type : AND
Input #1 : EQ1     Width : 1
Input #2 : STB1    Width : 1
Output Width : 1
Output #1 : BUF1

Object name :        Q1      Object Type : SIGNAL
Input #1 : BUF1    Width : 1
Output Width : 1
Output #1 : BUF3
Output #2 : AND3
```

```
Object name :       EQ1     Object Type : EQUAL
Input #1 : CLOCK1    Width : 1
Input #2 : LIT1    Width : 1
Output Width : 1
Output #1 : AND1

Object name :       STB1     Object Type : STABLE
Input #1 : CLOCK1    Width : 1
Output Width : 1
Output #1 : AND1

Object name :       BUF3     Object Type : BUFFER
Input #1 : AND3    Width : 1
Input #2 : AND2    Width : 1
Output Width : 1
Output #1 : ANDOUT

Object name :       AND3     Object Type : AND
Input #1 : Q1    Width : 1
Input #2 : Q2    Width : 1
Output Width : 1
Output #1 : BUF3

Object name :    CLOCK1     Object Type : INPUT
Output Width : 1
Output #1 : EQ1
Output #2 : STB1

Object name :       LIT1     Object Type : LIT:1
Output Width : 1
Output #1 : EQ1

Object name :    CLOCK2     Object Type : INPUT
Output Width : 1
Output #1 : EQ2
Output #2 : STB2

Object name :       EQ2     Object Type : EQUAL
Input #1 : CLOCK2    Width : 1
Input #2 : LIT2    Width : 1
Output Width : 1
Output #1 : AND2

Object name :       STB2     Object Type : STROBE
Input #1 : CLOCK2    Width : 1
Output Width : 1
Output #1 : AND2

Object name :       LIT2     Object Type : LIT:1
Output Width : 1
Output #1 : EQ2
```

```
Object name :      AND2     Object Type : AND
Input #1 : EQ2    Width : 1
Input #2 : STB2    Width : 1
Output Width : 1
Output #1 : BUF2
Output #2 : BUF3

Object name :      BUF2     Object Type : BUFFER
Input #1 : Q1    Width : 1
Input #2 : AND2    Width : 1
Output Width : 1
Output #1 : Q2

Object name :    ANDOUT     Object Type : OUTPUT
Input #1 : AND3    Width : 1
Output Width : 1

Object name :       Q2     Object Type : SIGNAL
Input #1 : BUF2    Width : 1
Output Width : 1
Output #1 : AND3
```

CKTA Fault List

Fault					Time(sec)
Fault #	1	Obj =	DATAIN Output to :	BUF1 Stuck-at-zero	3.691
Fault #	2	Obj =	DATAIN Output to :	BUF1 Stuck-at-one	3.513
Fault #	3	Obj =	BUF1 Output to :	Q1 Stuck-at-zero	1.622
Fault #	4	Obj =	BUF1 Output to :	Q1 Stuck-at-one	1.796
Fault #	5	Obj =	AND1 Fails to :	OR	Excl
Fault #	6	Obj =	AND1 Output to :	BUF1 Stuck-at-zero	Excl
Fault #	7	Obj =	AND1 Output to :	BUF1 Stuck-at-one	Excl
Fault #	8	Obj =	Q1 Output to :	BUF3 Stuck-at-zero	1.783
Fault #	9	Obj =	Q1 Output to :	BUF3 Stuck-at-one	1.647
Fault #	10	Obj =	Q1 Output to :	AND3 Stuck-at-zero	4.186
Fault #	11	Obj =	Q1 Output to :	AND3 Stuck-at-one	4.090
Fault #	12	Obj =	EQ1 Fails to :	XOR	Excl
Fault #	13	Obj =	EQ1 Output to :	AND1 Stuck-at-zero	Excl
Fault #	14	Obj =	EQ1 Output to :	AND1 Stuck-at-one	Excl
Fault #	15	Obj =	BUF3 Output to :	ANDOUT Stuck-at-zero	3.601
Fault #	16	Obj =	BUF3 Output to :	ANDOUT Stuck-at-one	3.003
Fault #	17	Obj =	AND3 Fails to :	OR	3.158
Fault #	18	Obj =	AND3 Output to :	BUF3 Stuck-at-zero	3.771
Fault #	19	Obj =	AND3 Output to :	BUF3 Stuck-at-one	3.782
Fault #	20	Obj =	CLOCK1 Output to :	EQ1 Stuck-at-zero	Excl
Fault #	21	Obj =	CLOCK1 Output to :	EQ1 Stuck-at-one	Excl
Fault #	22	Obj =	CLOCK1 Output to :	STB1 Stuck-at-zero	Excl
Fault #	23	Obj =	CLOCK1 Output to :	STB1 Stuck-at-one	Excl
Fault #	24	Obj =	LIT1 Output to :	EQ1 Stuck-at-zero	Excl
Fault #	25	Obj =	LIT1 Output to :	EQ1 Stuck-at-one	NAF
Fault #	26	Obj =	CLOCK2 Output to :	EQ2 Stuck-at-zero	Excl
Fault #	27	Obj =	CLOCK2 Output to :	EQ2 Stuck-at-one	Excl
Fault #	28	Obj =	CLOCK2 Output to :	STB2 Stuck-at-zero	Excl
Fault #	29	Obj =	CLOCK2 Output to :	STB2 Stuck-at-one	Excl
Fault #	30	Obj =	EQ2 Fails to :	XOR	Excl
Fault #	31	Obj =	EQ2 Output to :	AND2 Stuck-at-zero	Excl
Fault #	32	Obj =	EQ2 Output to :	AND2 Stuck-at-one	Excl
Fault #	33	Obj =	STB2 Output to :	AND2 Stuck-at-zero	Excl
Fault #	34	Obj =	STB2 Output to :	AND2 Stuck-at-one	Excl
Fault #	35	Obj =	LIT2 Output to :	EQ2 Stuck-at-zero	Excl
Fault #	36	Obj =	LIT2 Output to :	EQ2 Stuck-at-one	NAF
Fault #	37	Obj =	AND2 Fails to :	OR	Excl
Fault #	38	Obj =	AND2 Output to :	BUF2 Stuck-at-zero	Excl
Fault #	39	Obj =	AND2 Output to :	BUF2 Stuck-at-one	Excl
Fault #	40	Obj =	AND2 Output to :	BUF3 Stuck-at-zero	Excl
Fault #	41	Obj =	AND2 Output to :	BUF3 Stuck-at-one	Excl
Fault #	42	Obj =	BUF2 Output to :	Q2 Stuck-at-zero	4.796
Fault #	43	Obj =	BUF2 Output to :	Q2 Stuck-at-one	3.938
Fault #	44	Obj =	Q2 Output to :	AND3 Stuck-at-zero	2.673
Fault #	45	Obj =	Q2 Output to :	AND3 Stuck-at-one	2.762

CKTCV Model

```
        entity REGMUX(
          CLOCK : in BIT;
          CMD,
          INP   : in BIT_VECTOR(1 downto 0);
          C : out vit_VECTOR(1 downto 0)
          ) is
        end REGMUX;

        architecture arch of REGMUX is

        process(CLOCK,CMD,INP,A,B)
        begin
s1:   if CLOCK='1' and not CLOCK'STABLE then
s2:      case CMD is
s3:         when "00" => P <= INP;    -- load control register p
s4:         when "01" => A <= INP;    -- load A register
s5:         when "10" => B <= INP;    -- load b register
             when "11" =>              -- load C according to p
s6:            if P = "00" then
s7:               C <= a;
                else
s8:               C <= b;
                end if;
          end case;
        end if;
        end process;
        end arch;
```

CKTCV
Object name : CLOCK Object Type : INPUT
Output Width : 1
Output #1 : EQ1
Output #2 : STB1

Object name : EQ1 Object Type : EQUAL
Input #1 : CLOCK Width : 1
Input #2 : LIT1 Width : 1
Output Width : 1
Output #1 : AND1

Object name : STB1 Object Type : STABLE
Input #1 : CLOCK Width : 1
Output Width : 1
Output #1 : AND1

Object name : LIT1 Object Type : LIT:1
Output Width : 1
Output #1 : EQ1

```
Object name :      AND1     Object Type : AND
Input #1 : EQ1    Width : 1
Input #2 : STB1    Width : 1
Output Width : 1
Output #1 : AND2
Output #2 : AND3
Output #3 : AND4
Output #4 : AND5

Object name :      CMD     Object Type : INPUT
Output Width : 2
Output #1 : EQ2
Output #2 : EQ3
Output #3 : EQ4
Output #4 : EQ5

Object name :      EQ2     Object Type : EQUAL
Input #1 : CMD    Width : 2
Input #2 : LIT2    Width : 2
Output Width : 1
Output #1 : AND2

Object name :      EQ3     Object Type : EQUAL
Input #1 : CMD    Width : 2
Input #2 : LIT3    Width : 2
Output Width : 1
Output #1 : AND3

Object name :      EQ4     Object Type : EQUAL
Input #1 : CMD    Width : 2
Input #2 : LIT4    Width : 2
Output Width : 1
Output #1 : AND4

Object name :      EQ5     Object Type : EQUAL
Input #1 : CMD    Width : 2
Input #2 : LIT5    Width : 2
Output Width : 1
Output #1 : AND5

Object name :      LIT2     Object Type : LIT:00
Output Width : 2
Output #1 : EQ2

Object name :      AND2     Object Type : AND
Input #1 : AND1    Width : 1
Input #2 : EQ2    Width : 1
Output Width : 1
Output #1 : BUF1

Object name :      LIT3     Object Type : LIT:01
Output Width : 2
```

```
Output #1 : EQ3

Object name :      AND3     Object Type : AND
Input #1 : EQ3    Width : 1
Input #2 : AND1    Width : 1
Output Width : 1
Output #1 : BUF2

Object name :      LIT4     Object Type : LIT:10
Output Width : 2
Output #1 : EQ4

Object name :      AND4     Object Type : AND
Input #1 : EQ4    Width : 1
Input #2 : AND1    Width : 1
Output Width : 1
Output #1 : BUF3

Object name :      LIT5     Object Type : LIT:11
Output Width : 2
Output #1 : EQ5

Object name :      AND5     Object Type : AND
Input #1 : EQ5    Width : 1
Input #2 : AND1    Width : 1
Output Width : 1
Output #1 : AND6

Object name :       INP     Object Type : INPUT
Output Width : 2
Output #1 : BUF1
Output #2 : BUF2
Output #3 : BUF3

Object name :      BUF1     Object Type : BUFFER
Input #1 : INP    Width : 2
Input #2 : AND2    Width : 1
Output Width : 2
Output #1 : P

Object name :      BUF2     Object Type : BUFFER
Input #1 : INP    Width : 2
Input #2 : AND3    Width : 1
Output Width : 2
Output #1 : A

Object name :      BUF3     Object Type : BUFFER
Input #1 : INP    Width : 2
Input #2 : AND4    Width : 1
Output Width : 2
Output #1 : B
```

```
Object name :          P     Object Type : SIGNAL
Input #1 : BUF1    Width : 2
Output Width : 2
Output #1 : EQ6

Object name :          A     Object Type : SIGNAL
Input #1 : BUF2    Width : 2
Output Width : 2
Output #1 : BUF4

Object name :          B     Object Type : SIGNAL
Input #1 : BUF3    Width : 2
Output Width : 2
Output #1 : BUF5

Object name :          C     Object Type : OUTPUT
Input #1 : BUF4    Width : 2
Input #2 : BUF5    Width : 2
Output Width : 2

Object name :      BUF4     Object Type : BUFFER
Input #1 : A    Width : 2
Input #2 : AND6    Width : 1
Output Width : 2
Output #1 : C

Object name :      BUF5     Object Type : BUFFER
Input #1 : B    Width : 2
Input #2 : NOT1    Width : 1
Output Width : 2
Output #1 : C

Object name :      AND6     Object Type : AND
Input #1 : EQ6    Width : 1
Input #2 : AND5    Width : 1
Output Width : 1
Output #1 : BUF4
Output #2 : NOT1

Object name :      NOT1     Object Type : NOT
Input #1 : AND6    Width : 1
Output Width : 1
Output #1 : BUF5

Object name :      EQ6      Object Type : EQUAL
Input #1 : P    Width : 2
Input #2 : LIT6    Width : 2
Output Width : 1
Output #1 : AND6

Object name :      LIT6     Object Type : LIT:00
Output Width : 2
Output #1 : EQ6
```

CKTCV Fault List

Fault					Time(sec)
Fault # 1	Obj =	CLOCK	Output to :	EQ1 Stuck-at-zero	Excl
Fault # 2	Obj =	CLOCK	Output to :	EQ1 Stuck-at-one	Excl
Fault # 3	Obj =	CLOCK	Output to :	STB1 Stuck-at-zero	Excl
Fault # 4	Obj =	CLOCK	Output to :	STB1 Stuck-at-one	Excl
Fault # 5	Obj =	EQ1	Fails to :	XOR	Excl
Fault # 6	Obj =	EQ1	Output to :	AND1 Stuck-at-zero	Excl
Fault # 7	Obj =	EQ1	Output to :	AND1 Stuck-at-one	Excl
Fault # 8	Obj =	LIT1	Output to :	EQ1 Stuck-at-zero	Excl
Fault # 9	Obj =	LIT1	Output to :	EQ1 Stuck-at-one	NAF
Fault # 10	Obj =	AND1	Fails to :	OR	Excl
Fault # 11	Obj =	AND1	Output to :	AND2 Stuck-at-zero	Excl
Fault # 12	Obj =	AND1	Output to :	AND2 Stuck-at-one	Excl
Fault # 13	Obj =	AND1	Output to :	AND3 Stuck-at-zero	Excl
Fault # 14	Obj =	AND1	Output to :	AND3 Stuck-at-one	Excl
Fault # 15	Obj =	AND1	Output to :	AND4 Stuck-at-zero	Excl
Fault # 16	Obj =	AND1	Output to :	AND4 Stuck-at-one	Excl
Fault # 17	Obj =	AND1	Output to :	AND5 Stuck-at-zero	Excl
Fault # 18	Obj =	AND1	Output to :	AND5 Stuck-at-one	Excl
Fault # 19	Obj =	CMD	Output to :	EQ2 Stuck-at-zero	3.809
Fault # 20	Obj =	CMD	Output to :	EQ2 Stuck-at-one	4.097
Fault # 21	Obj =	CMD	Output to :	EQ3 Stuck-at-zero	5.601
Fault # 22	Obj =	CMD	Output to :	EQ3 Stuck-at-one	6.338
Fault # 23	Obj =	CMD	Output to :	EQ4 Stuck-at-zero	6.718
Fault # 24	Obj =	CMD	Output to :	EQ4 Stuck-at-one	6.095
Fault # 25	Obj =	CMD	Output to :	EQ5 Stuck-at-zero	7.506
Fault # 26	Obj =	CMD	Output to :	EQ5 Stuck-at-one	7.893
Fault # 27	Obj =	EQ2	Fails to :	XOR	5.943
Fault # 28	Obj =	EQ2	Output to :	AND2 Stuck-at-zero	8.922
Fault # 29	Obj =	EQ2	Output to :	AND2 Stuck-at-one	7.984
Fault # 30	Obj =	EQ3	Fails to :	XOR	3.297
Fault # 31	Obj =	EQ3	Output to :	AND3 Stuck-at-zero	7.891
Fault # 32	Obj =	EQ3	Output to :	AND3 Stuck-at-one	8.912
Fault # 33	Obj =	EQ4	Fails to :	XOR	6.894
Fault # 34	Obj =	EQ4	Output to :	AND4 Stuck-at-zero	7.497
Fault # 35	Obj =	EQ4	Output to :	AND4 Stuck-at-one	7.290
Fault # 36	Obj =	EQ5	Fails to :	XOR	4.062
Fault # 37	Obj =	EQ5	Output to :	AND5 Stuck-at-zero	3.136
Fault # 38	Obj =	EQ5	Output to :	AND5 Stuck-at-one	2.997
Fault # 39	Obj =	LIT2	Output to :	EQ2 Stuck-at-zero	NAF
Fault # 40	Obj =	LIT2	Output to :	EQ2 Stuck-at-one	3.890
Fault # 41	Obj =	AND2	Fails to :	OR	6.750
Fault # 42	Obj =	AND2	Output to :	BUF1 Stuck-at-zero	3.810
Fault # 43	Obj =	AND2	Output to :	BUF1 Stuck-at-one	4.707
Fault # 44	Obj =	LIT3	Output to :	EQ3 Stuck-at-zero	NAF
Fault # 45	Obj =	LIT3	Output to :	EQ3 Stuck-at-one	4.891
Fault # 46	Obj =	AND3	Fails to :	OR	5.649

Fault # 47	Obj =	AND3 Output to :	BUF2 Stuck-at-zero	8.758
Fault # 48	Obj =	AND3 Output to :	BUF2 Stuck-at-one	7.999
Fault # 49	Obj =	LIT4 Output to :	EQ4 Stuck-at-zero	7.997
Fault # 50	Obj =	LIT4 Output to :	EQ4 Stuck-at-one	NAF
Fault # 51	Obj =	AND4 Fails to :	OR	4.948
Fault # 52	Obj =	AND4 Output to :	BUF3 Stuck-at-zero	5.455
Fault # 53	Obj =	AND4 Output to :	BUF3 Stuck-at-one	4.720
Fault # 54	Obj =	LIT5 Output to :	EQ5 Stuck-at-zero	6.656
Fault # 55	Obj =	LIT5 Output to :	EQ5 Stuck-at-one	NAF
Fault # 56	Obj =	AND5 Fails to :	OR	4.929
Fault # 57	Obj =	AND5 Output to :	AND6 Stuck-at-zero	7.288
Fault # 58	Obj =	AND5 Output to :	AND6 Stuck-at-one	6.397
Fault # 59	Obj =	INP Output to :	BUF1 Stuck-at-zero	3.095
Fault # 60	Obj =	INP Output to :	BUF1 Stuck-at-one	3.527
Fault # 61	Obj =	INP Output to :	BUF2 Stuck-at-zero	6.871
Fault # 62	Obj =	INP Output to :	BUF2 Stuck-at-one	6.701
Fault # 63	Obj =	INP Output to :	BUF3 Stuck-at-zero	5.637
Fault # 64	Obj =	INP Output to :	BUF3 Stuck-at-one	5.195
Fault # 65	Obj =	BUF1 Output to :	P Stuck-at-zero	2.867
Fault # 66	Obj =	BUF1 Output to :	P Stuck-at-one	2.336
Fault # 67	Obj =	BUF2 Output to :	A Stuck-at-zero	2.245
Fault # 68	Obj =	BUF2 Output to :	A Stuck-at-one	1.956
Fault # 69	Obj =	BUF3 Output to :	B Stuck-at-zero	7.765
Fault # 70	Obj =	BUF3 Output to :	B Stuck-at-one	7.727
Fault # 71	Obj =	P Output to :	EQ6 Stuck-at-zero	6.324
Fault # 72	Obj =	P Output to :	EQ6 Stuck-at-one	6.644
Fault # 73	Obj =	A Output to :	BUF4 Stuck-at-zero	2.913
Fault # 74	Obj =	A Output to :	BUF4 Stuck-at-one	2.853
Fault # 75	Obj =	B Output to :	BUF5 Stuck-at-zero	5.637
Fault # 76	Obj =	B Output to :	BUF5 Stuck-at-one	4.658
Fault # 77	Obj =	BUF4 Output to :	C Stuck-at-zero	5.183
Fault # 78	Obj =	BUF4 Output to :	C Stuck-at-one	5.394
Fault # 79	Obj =	BUF5 Output to :	C Stuck-at-zero	4.347
Fault # 80	Obj =	BUF5 Output to :	C Stuck-at-one	4.247
Fault # 81	Obj =	AND6 Fails to :	OR	1.831
Fault # 82	Obj =	AND6 Output to :	BUF4 Stuck-at-zero	7.454
Fault # 83	Obj =	AND6 Output to :	BUF4 Stuck-at-one	7.411
Fault # 84	Obj =	AND6 Output to :	NOT1 Stuck-at-zero	8.385
Fault # 85	Obj =	AND6 Output to :	NOT1 Stuck-at-one	7.432
Fault # 86	Obj =	NOT1 Fails to :	FBUF	7.194
Fault # 87	Obj =	NOT1 Output to :	BUF5 Stuck-at-zero	7.397
Fault # 88	Obj =	NOT1 Output to :	BUF5 Stuck-at-one	8.051
Fault # 89	Obj =	EQ6 Fails to :	XOR	6.331
Fault # 90	Obj =	EQ6 Output to :	AND6 Stuck-at-zero	5.888
Fault # 91	Obj =	EQ6 Output to :	AND6 Stuck-at-one	5.954
Fault # 92	Obj =	LIT6 Output to :	EQ6 Stuck-at-zero	NAF
Fault # 93	Obj =	LIT6 Output to :	EQ6 Stuck-at-one	4.466

CNTR Model

```
        entity COUNTER(
          CLRBAR,
          CLOCK : in BIT;
          Q1,
          Q2,
          Q3    : out BIT) is
        end COUNTER;

        architecture ARCH of COUNTER is

        process(CLRBAR,CLOCK)
        begin
s1:    if CLRBAR = '0' then
s2:       Q1 <= '0';
s3:       Q2 <= '0';
s4:       Q3 <= '0';
        else
s5:       if CLOCK='1' and not CLOCK'STABLE then
s6:          Q1 <= not Q1;
s7:          Q3 <= Q2 xor Q1;
s8:          Q3 <= Q3 xor (Q1 and Q2);
           end if
        end if
        end process;
        end arch;
```

CNTR
Object name : CLRBAR Object Type : INPUT
Output Width : 1
Output #1 : EQ1

Object name : EQ1 Object Type : EQUAL
Input #1 : CLRBAR Width : 1
Input #2 : LIT1 Width : 1
Output Width : 1
Output #1 : BUF1
Output #2 : BUF2
Output #3 : BUF3

Object name : LIT1 Object Type : LIT:0
Output Width : 1
Output #1 : EQ1

Object name : BUF1 Object Type : BUFFER
Input #1 : LIT2 Width : 1
Input #2 : EQ1 Width : 1
Output Width : 1

```
Output #1 : Q1

Object name :      BUF2     Object Type : BUFFER
Input #1 : LIT3    Width : 1
Input #2 : EQ1     Width : 1
Output Width : 1
Output #1 : Q2

Object name :      BUF3     Object Type : BUFFER
Input #1 : LIT4    Width : 1
Input #2 : EQ1     Width : 1
Output Width : 1
Output #1 : Q3

Object name :     CLOCK     Object Type : INPUT
Output Width : 1
Output #1 : EQ2
Output #2 : STB1

Object name :       EQ2     Object Type : EQUAL
Input #1 : CLOCK    Width : 1
Input #2 : LIT5     Width : 1
Output Width : 1
Output #1 : AND1

Object name :      STB1     Object Type : STABLE
Input #1 : CLOCK    Width : 1
Output Width : 1
Output #1 : AND1

Object name :      LIT5     Object Type : LIT:0
Output Width : 1
Output #1 : EQ2

Object name :      AND1     Object Type : AND
Input #1 : EQ2     Width : 1
Input #2 : STB1    Width : 1
Output Width : 1
Output #1 : BUF4
Output #2 : BUF5
Output #3 : BUF6

Object name :        Q1     Object Type : OUTPUT
Input #1 : BUF1    Width : 1
Input #2 : BUF4    Width : 1
Output Width : 1
Output #1 : NOT1
Output #2 : XOR1
Output #3 : AND2

Object name :      BUF4     Object Type : BUFFER
Input #1 : NOT1    Width : 1
Input #2 : AND1    Width : 1
```

```
Output Width : 1
Output #1 : Q1

Object name :      NOT1     Object Type : NOT
Input #1 : Q1    Width : 1
Output Width : 1
Output #1 : BUF4

Object name :      XOR1     Object Type : XOR
Input #1 : Q2    Width : 1
Input #2 : Q1    Width : 1
Output Width : 1
Output #1 : BUF5

Object name :      AND2     Object Type : AND
Input #1 : Q1    Width : 1
Input #2 : Q2    Width : 1
Output Width : 1
Output #1 : XOR2

Object name :      LIT2     Object Type : LIT:0
Output Width : 1
Output #1 : BUF1

Object name :        Q2     Object Type : OUTPUT
Input #1 : BUF2    Width : 1
Input #2 : BUF5    Width : 1
Output Width : 1
Output #1 : XOR1
Output #2 : AND2

Object name :      BUF5     Object Type : BUFFER
Input #1 : XOR1    Width : 1
Input #2 : AND1    Width : 1
Output Width : 1
Output #1 : Q2

Object name :      XOR2     Object Type : XOR
Input #1 : AND2    Width : 1
Input #2 : Q3    Width : 1
Output Width : 1
Output #1 : BUF6

Object name :      LIT3     Object Type : LIT:0
Output Width : 1
Output #1 : BUF2

Object name :        Q3     Object Type : OUTPUT
Input #1 : BUF3    Width : 1
Input #2 : BUF6    Width : 1
Output Width : 1
Output #1 : XOR2
```

```
Object name :      BUF6     Object Type : BUFFER
Input #1 : XOR2    Width : 1
Input #2 : AND1    Width : 1
Output Width : 1
Output #1 : Q3

Object name :      LIT4     Object Type : LIT:0
Output Width : 1
Output #1 : BUF3
```

CNTR Fault List

Fault					Time(sec)
Fault # 1	Obj =	CLRBAR Output to :	EQ1 Stuck-at-zero		2.017
Fault # 2	Obj =	CLRBAR Output to :	EQ1 Stuck-at-one		1.753
Fault # 3	Obj =	EQ1 Fails to :	XOR		1.264
Fault # 4	Obj =	EQ1 Output to :	BUF1 Stuck-at-zero		1.877
Fault # 5	Obj =	EQ1 Output to :	BUF1 Stuck-at-one		1.557
Fault # 6	Obj =	EQ1 Output to :	BUF2 Stuck-at-zero		2.019
Fault # 7	Obj =	EQ1 Output to :	BUF2 Stuck-at-one		2.122
Fault # 8	Obj =	EQ1 Output to :	BUF3 Stuck-at-zero		1.389
Fault # 9	Obj =	EQ1 Output to :	BUF3 Stuck-at-one		1.323
Fault # 10	Obj =	LIT1 Output to :	EQ1 Stuck-at-zero		NAF
Fault # 11	Obj =	LIT1 Output to :	EQ1 Stuck-at-one		2.342
Fault # 12	Obj =	BUF1 Output to :	Q1 Stuck-at-zero		1.413
Fault # 13	Obj =	BUF1 Output to :	Q1 Stuck-at-one		1.615
Fault # 14	Obj =	BUF2 Output to :	Q2 Stuck-at-zero		2.810
Fault # 15	Obj =	BUF2 Output to :	Q2 Stuck-at-one		1.621
Fault # 16	Obj =	BUF3 Output to :	Q3 Stuck-at-zero		Fail
Fault # 17	Obj =	BUF3 Output to :	Q3 Stuck-at-one		0.943
Fault # 18	Obj =	CLOCK Output to :	EQ2 Stuck-at-zero		Excl
Fault # 19	Obj =	CLOCK Output to :	EQ2 Stuck-at-one		Excl
Fault # 20	Obj =	CLOCK Output to :	STB1 Stuck-at-zero		Excl
Fault # 21	Obj =	CLOCK Output to :	STB1 Stuck-at-one		Excl
Fault # 22	Obj =	EQ2 Fails to :	XOR		Excl
Fault # 23	Obj =	EQ2 Output to :	AND1 Stuck-at-zero		Excl
Fault # 24	Obj =	EQ2 Output to :	AND1 Stuck-at-one		Excl
Fault # 25	Obj =	LIT5 Output to :	EQ2 Stuck-at-zero		NAF
Fault # 26	Obj =	LIT5 Output to :	EQ2 Stuck-at-one		Excl
Fault # 27	Obj =	AND1 Fails to :	OR		Excl
Fault # 28	Obj =	AND1 Output to :	BUF4 Stuck-at-zero		2.456
Fault # 29	Obj =	AND1 Output to :	BUF4 Stuck-at-one		2.612
Fault # 30	Obj =	AND1 Output to :	BUF5 Stuck-at-zero		3.394
Fault # 31	Obj =	AND1 Output to :	BUF5 Stuck-at-one		3.218
Fault # 32	Obj =	AND1 Output to :	BUF6 Stuck-at-zero		4.532
Fault # 33	Obj =	AND1 Output to :	BUF6 Stuck-at-one		4.781
Fault # 34	Obj =	BUF4 Output to :	Q1 Stuck-at-zero		1.959
Fault # 35	Obj =	BUF4 Output to :	Q1 Stuck-at-one		2.104
Fault # 36	Obj =	NOT1 Fails to :	FBUF		1.904
Fault # 37	Obj =	NOT1 Output to :	BUF4 Stuck-at-zero		2.143
Fault # 38	Obj =	NOT1 Output to :	BUF4 Stuck-at-one		2.146
Fault # 39	Obj =	XOR1 Fails to :	EQUAL		1.726
Fault # 40	Obj =	XOR1 Output to :	BUF5 Stuck-at-zero		2.534
Fault # 41	Obj =	XOR1 Output to :	BUF5 Stuck-at-one		1.684
Fault # 42	Obj =	AND2 Fails to :	OR		1.164
Fault # 43	Obj =	AND2 Output to :	XOR2 Stuck-at-zero		1.951
Fault # 44	Obj =	AND2 Output to :	XOR2 Stuck-at-one		2.056
Fault # 45	Obj =	LIT2 Output to :	BUF1 Stuck-at-zero		NAF
Fault # 46	Obj =	LIT2 Output to :	BUF1 Stuck-at-one		1.507
Fault # 47	Obj =	BUF5 Output to :	Q2 Stuck-at-zero		1.746

Fault # 48	Obj =	BUF5 Output to :	Q2 Stuck-at-one	1.781
Fault # 49	Obj =	XOR2 Fails to :	EQUAL	2.970
Fault # 50	Obj =	XOR2 Output to :	BUF6 Stuck-at-zero	1.517
Fault # 51	Obj =	XOR2 Output to :	BUF6 Stuck-at-one	1.579
Fault # 52	Obj =	LIT3 Output to :	BUF2 Stuck-at-zero	NAF
Fault # 53	Obj =	LIT3 Output to :	BUF2 Stuck-at-one	Fail
Fault # 54	Obj =	BUF6 Output to :	Q3 Stuck-at-zero	Fail
Fault # 55	Obj =	BUF6 Output to :	Q3 Stuck-at-one	1.136
Fault # 56	Obj =	LIT4 Output to :	BUF3 Stuck-at-zero	NAF
Fault # 57	Obj =	LIT4 Output to :	BUF3 Stuck-at-one	0.683

CNTRV Model

```
entity COUNTERV (
  CLRBAR,
  CLOCK : in BIT;
  COUNT : out BIT_VECTOR(2 downto 0)
  ) is
end COUNTERV;

architecture ARCH of COUNTERV is

process(CLRBAR,CLOCK)
begin
s1:   if CLRBAR='0' then
s2:      COUNT <= "000";
      else
s3:      if CLOCK='1' and not CLOCK'STABLE then
s4:         COUNT <= bvadd(COUNT,"001")
         end if
      end if
      end block;
      end process ARCH;
```

CNTRV
Object name : CLRBAR Object Type : INPUT
Output Width : 1
Output #1 : EQ1

Object name : EQ1 Object Type : EQUAL
Input #1 : CLRBAR Width : 1
Input #2 : LIT1 Width : 1
Output Width : 1
Output #1 : BUF1
Output #2 : NOT1

Object name : LIT1 Object Type : LIT:0
Output Width : 1
Output #1 : EQ1

Object name : BUF1 Object Type : BUFFER
Input #1 : LIT2 Width : 3
Input #2 : EQ1 Width : 1
Output Width : 3
Output #1 : COUNT

Object name : NOT1 Object Type : NOT
Input #1 : EQ1 Width : 1
Output Width : 1
Output #1 : AND1

```
Object name :     CLOCK     Object Type : INPUT
Output Width : 1
Output #1 : EQ2
Output #2 : STB1

Object name :       EQ2     Object Type : EQUAL
Input #1 : CLOCK    Width : 1
Input #2 : LIT3     Width : 1
Output Width : 1
Output #1 : AND1

Object name :      STB1     Object Type : STABLE
Input #1 : CLOCK    Width : 1
Output Width : 1
Output #1 : AND1

Object name :      LIT3     Object Type : LIT:1
Output Width : 1
Output #1 : EQ2

Object name :      AND1     Object Type : AND
Input #1 : NOT1     Width : 1
Input #2 : EQ2      Width : 1
Input #3 : STB1     Width : 1
Output Width : 1
Output #1 : BUF2

Object name :     COUNT     Object Type : OUTPUT
Input #1 : BUF1     Width : 3
Input #2 : BUF2     Width : 3
Output Width : 3
Output #1 : ADD1

Object name :      BUF2     Object Type : BUFFER
Input #1 : ADD1     Width : 3
Input #2 : AND1     Width : 1
Output Width : 3
Output #1 : COUNT

Object name :      ADD1     Object Type : ADD
Input #1 : COUNT    Width : 3
Input #2 : LIT4     Width : 3
Output Width : 3
Output #1 : BUF2

Object name :      LIT2     Object Type : LIT:000
Output Width : 3
Output #1 : BUF1

Object name :      LIT4     Object Type : LIT:001
Output Width : 3
Output #1 : ADD1
```

CNTRV Fault List

Fault					Time(sec)
Fault #	1	Obj =	CLRBAR Output to :	EQ1 Stuck-at-zero	1.052
Fault #	2	Obj =	CLRBAR Output to :	EQ1 Stuck-at-one	0.934
Fault #	3	Obj =	EQ1 Fails to :	XOR	1.778
Fault #	4	Obj =	EQ1 Output to :	BUF1 Stuck-at-zero	1.448
Fault #	5	Obj =	EQ1 Output to :	BUF1 Stuck-at-one	1.313
Fault #	6	Obj =	EQ1 Output to :	NOT1 Stuck-at-zero	1.505
Fault #	7	Obj =	EQ1 Output to :	NOT1 Stuck-at-one	1.529
Fault #	8	Obj =	LIT1 Output to :	EQ1 Stuck-at-zero	NAF
Fault #	9	Obj =	LIT1 Output to :	EQ1 Stuck-at-one	2.467
Fault #	10	Obj =	BUF1 Output to :	COUNT Stuck-at-zero	1.706
Fault #	11	Obj =	BUF1 Output to :	COUNT Stuck-at-one	1.629
Fault #	12	Obj =	NOT1 Fails to :	FBUF	1.222
Fault #	13	Obj =	NOT1 Output to :	AND1 Stuck-at-zero	2.692
Fault #	14	Obj =	NOT1 Output to :	AND1 Stuck-at-one	2.664
Fault #	15	Obj =	CLOCK Output to :	EQ2 Stuck-at-zero	Excl
Fault #	16	Obj =	CLOCK Output to :	EQ2 Stuck-at-one	Excl
Fault #	17	Obj =	CLOCK Output to :	STB1 Stuck-at-zero	Excl
Fault #	18	Obj =	CLOCK Output to :	STB1 Stuck-at-one	Excl
Fault #	19	Obj =	EQ2 Fails to :	XOR	Excl
Fault #	20	Obj =	EQ2 Output to :	AND1 Stuck-at-zero	Excl
Fault #	21	Obj =	EQ2 Output to :	AND1 Stuck-at-one	Excl
Fault #	22	Obj =	LIT3 Output to :	EQ2 Stuck-at-zero	Excl
Fault #	23	Obj =	LIT3 Output to :	EQ2 Stuck-at-one	NAF
Fault #	24	Obj =	AND1 Fails to :	OR	Excl
Fault #	25	Obj =	AND1 Output to :	BUF2 Stuck-at-zero	2.145
Fault #	26	Obj =	AND1 Output to :	BUF2 Stuck-at-one	3.109
Fault #	27	Obj =	BUF2 Output to :	COUNT Stuck-at-zero	2.415
Fault #	28	Obj =	BUF2 Output to :	COUNT Stuck-at-one	2.221
Fault #	29	Obj =	ADD1 Fails to :	SUB	3.500
Fault #	30	Obj =	ADD1 Output to :	BUF2 Stuck-at-zero	2.523
Fault #	31	Obj =	ADD1 Output to :	BUF2 Stuck-at-one	1.729
Fault #	32	Obj =	LIT2 Output to :	BUF1 Stuck-at-zero	NAF
Fault #	33	Obj =	LIT2 Output to :	BUF1 Stuck-at-one	0.937
Fault #	34	Obj =	LIT4 Output to :	ADD1 Stuck-at-zero	NAF
Fault #	35	Obj =	LIT4 Output to :	ADD1 Stuck-at-one	1.077

DFF VHDL Model

```
        entity DFF(
          CLRBAR,
          SETBAR,
          DATA,
          CLOCK: in BIT;
          Q,
          QBAR: out BIT
          ) is
        end DFF;

        architecture arch of DFF is

        process(CLRBAR,SETBAR,DATA,CLOCK)
        begin
s1:    if CLRBAR = 0 then
s2:       Q <= 0;
s3:       QBAR <= 1;
        else
s4:       if SETBAR = 0 then
s5:          Q <= 1;
s6:          QBAR <= 0;
           else
s7:          if (not CLOCK'STABLE) and (CLOCK = 1) then
s8:             Q <= DATA;
s9:             QBAR <= not DATA;
              end if
           end if
        end if
        end process;
        end arch;
```

```
DFF
Object name :    CLRBAR      Object Type : INPUT
Output Width : 1
Output #1 : EQ1

Object name :       EQ1      Object Type : EQUAL
Input #1 : CLRBAR    Width : 1
Input #2 : LIT1     Width : 1
Output Width : 1
Output #1 : BUF1
Output #2 : BUF2
Output #3 : NOT1

Object name :       LIT1      Object Type : LIT:0
Output Width : 1
Output #1 : EQ1
```

```
Object name :      BUF1     Object Type : BUFFER
Input #1 : LIT2    Width : 1
Input #2 : EQ1     Width : 1
Output Width : 1
Output #1 : Q

Object name :      BUF2     Object Type : BUFFER
Input #1 : LIT3    Width : 1
Input #2 : EQ1     Width : 1
Output Width : 1
Output #1 : QBAR

Object name :      NOT1     Object Type : NOT
Input #1 : EQ1     Width : 1
Output Width : 1
Output #1 : AND1

Object name :      LIT2     Object Type : LIT:0
Output Width : 1
Output #1 : BUF1

Object name :        Q      Object Type : OUTPUT
Input #1 : BUF1    Width : 1
Input #2 : BUF3    Width : 1
Input #3 : BUF5    Width : 1
Output Width : 1

Object name :      LIT3     Object Type : LIT:1
Output Width : 1
Output #1 : BUF2

Object name :      QBAR     Object Type : OUTPUT
Input #1 : BUF2    Width : 1
Input #2 : BUF4    Width : 1
Input #3 : BUF6    Width : 1
Output Width : 1

Object name :      AND1     Object Type : AND
Input #1 : NOT1    Width : 1
Input #2 : EQ2     Width : 1
Output Width : 1
Output #1 : BUF3
Output #2 : BUF4
Output #3 : NOT2

Object name :   SETBAR      Object Type : INPUT
Output Width : 1
Output #1 : EQ2

Object name :      EQ2      Object Type : EQUAL
Input #1 : SETBAR  Width : 1
Input #2 : LIT4    Width : 1
Output Width : 1
```

```
Output #1 : AND1

Object name :      LIT4     Object Type : LIT:0
Output Width : 1
Output #1 : EQ2

Object name :      BUF3     Object Type : BUFFER
Input #1 : LIT5    Width : 1
Input #2 : AND1    Width : 1
Output Width : 1
Output #1 : Q

Object name :      BUF4     Object Type : BUFFER
Input #1 : LIT6    Width : 1
Input #2 : AND1    Width : 1
Output Width : 1
Output #1 : QBAR

Object name :      NOT2     Object Type : NOT
Input #1 : AND1    Width : 1
Output Width : 1
Output #1 : AND3

Object name :      DATA     Object Type : INPUT
Output Width : 1
Output #1 : BUF5
Output #2 : NOT3

Object name :      BUF5     Object Type : BUFFER
Input #1 : DATA    Width : 1
Input #2 : AND3    Width : 1
Output Width : 1
Output #1 : Q

Object name :      NOT3     Object Type : NOT
Input #1 : DATA    Width : 1
Output Width : 1
Output #1 : BUF6

Object name :      AND3     Object Type : AND
Input #1 : NOT2    Width : 1
Input #2 : AND2    Width : 1
Output Width : 1
Output #1 : BUF5
Output #2 : BUF6

Object name :      BUF6     Object Type : BUFFER
Input #1 : NOT3    Width : 1
Input #2 : AND3    Width : 1
Output Width : 1
Output #1 : QBAR
```

```
Object name :      AND2     Object Type : AND
Input #1 : STB1    Width : 1
Input #2 : EQ3     Width : 1
Output Width : 1
Output #1 : AND3

Object name :     CLOCK     Object Type : INPUT
Output Width : 1
Output #1 : STB1
Output #2 : EQ3

Object name :      STB1     Object Type : STABLE
Input #1 : CLOCK    Width : 1
Output Width : 1
Output #1 : AND2

Object name :       EQ3     Object Type : EQUAL
Input #1 : CLOCK    Width : 1
Input #2 : LIT7     Width : 1
Output Width : 1
Output #1 : AND2

Object name :      LIT7     Object Type : LIT:1
Output Width : 1
Output #1 : EQ3

Object name :      LIT5     Object Type : LIT:1
Output Width : 1
Output #1 : BUF3

Object name :      LIT6     Object Type : LIT:0
Output Width : 1
Output #1 : BUF4
```

DFF Fault List

Fault				Time(sec)
Fault # 1	Obj =	CLRBAR Output to :	EQ1 Stuck-at-zero	0.601
Fault # 2	Obj =	CLRBAR Output to :	EQ1 Stuck-at-one	0.557
Fault # 3	Obj =	EQ1 Fails to :	XOR	0.600
Fault # 4	Obj =	EQ1 Output to :	BUF1 Stuck-at-zero	0.799
Fault # 5	Obj =	EQ1 Output to :	BUF1 Stuck-at-one	0.900
Fault # 6	Obj =	EQ1 Output to :	BUF2 Stuck-at-zero	1.561
Fault # 7	Obj =	EQ1 Output to :	BUF2 Stuck-at-one	1.405
Fault # 8	Obj =	EQ1 Output to :	NOT1 Stuck-at-zero	1.315
Fault # 9	Obj =	EQ1 Output to :	NOT1 Stuck-at-one	1.249
Fault # 10	Obj =	LIT1 Output to :	EQ1 Stuck-at-zero	NAF
Fault # 11	Obj =	LIT1 Output to :	EQ1 Stuck-at-one	0.882
Fault # 12	Obj =	BUF1 Output to :	Q Stuck-at-zero	1.553
Fault # 13	Obj =	BUF1 Output to :	Q Stuck-at-one	1.588
Fault # 14	Obj =	BUF2 Output to :	QBAR Stuck-at-zero	0.725
Fault # 15	Obj =	BUF2 Output to :	QBAR Stuck-at-one	0.737
Fault # 16	Obj =	NOT1 Fails to :	FBUF	2.087
Fault # 17	Obj =	NOT1 Output to :	AND1 Stuck-at-zero	1.297
Fault # 18	Obj =	NOT1 Output to :	AND1 Stuck-at-one	1.185
Fault # 19	Obj =	LIT2 Output to :	BUF1 Stuck-at-zero	NAF
Fault # 20	Obj =	LIT2 Output to :	BUF1 Stuck-at-one	1.405
Fault # 21	Obj =	LIT3 Output to :	BUF2 Stuck-at-zero	1.222
Fault # 22	Obj =	LIT3 Output to :	BUF2 Stuck-at-one	NAF
Fault # 23	Obj =	AND1 Fails to :	OR	1.431
Fault # 24	Obj =	AND1 Output to :	BUF3 Stuck-at-zero	1.784
Fault # 25	Obj =	AND1 Output to :	BUF3 Stuck-at-one	1.909
Fault # 26	Obj =	AND1 Output to :	BUF4 Stuck-at-zero	1.273
Fault # 27	Obj =	AND1 Output to :	BUF4 Stuck-at-one	1.165
Fault # 28	Obj =	AND1 Output to :	NOT2 Stuck-at-zero	0.836
Fault # 29	Obj =	AND1 Output to :	NOT2 Stuck-at-one	0.760
Fault # 30	Obj =	SETBAR Output to :	EQ2 Stuck-at-zero	1.574
Fault # 31	Obj =	SETBAR Output to :	EQ2 Stuck-at-one	1.681
Fault # 32	Obj =	EQ2 Fails to :	XOR	1.365
Fault # 33	Obj =	EQ2 Output to :	AND1 Stuck-at-zero	1.583
Fault # 34	Obj =	EQ2 Output to :	AND1 Stuck-at-one	1.774
Fault # 35	Obj =	LIT4 Output to :	EQ2 Stuck-at-zero	NAF
Fault # 36	Obj =	LIT4 Output to :	EQ2 Stuck-at-one	1.761
Fault # 37	Obj =	BUF3 Output to :	Q Stuck-at-zero	1.476
Fault # 38	Obj =	BUF3 Output to :	Q Stuck-at-one	1.612
Fault # 39	Obj =	BUF4 Output to :	QBAR Stuck-at-zero	1.254
Fault # 40	Obj =	BUF4 Output to :	QBAR Stuck-at-one	1.307
Fault # 41	Obj =	NOT2 Fails to :	FBUF	1.737
Fault # 42	Obj =	NOT2 Output to :	AND3 Stuck-at-zero	0.810
Fault # 43	Obj =	NOT2 Output to :	AND3 Stuck-at-one	0.739
Fault # 44	Obj =	DATA Output to :	BUF5 Stuck-at-zero	1.584
Fault # 45	Obj =	DATA Output to :	BUF5 Stuck-at-one	1.380
Fault # 46	Obj =	DATA Output to :	NOT3 Stuck-at-zero	1.453
Fault # 47	Obj =	DATA Output to :	NOT3 Stuck-at-one	1.601

Fault # 48	Obj =	BUF5 Output to :	Q Stuck-at-zero	1.422
Fault # 49	Obj =	BUF5 Output to :	Q Stuck-at-one	1.366
Fault # 50	Obj =	NOT3 Fails to :	FBUF	0.784
Fault # 51	Obj =	NOT3 Output to :	BUF6 Stuck-at-zero	1.512
Fault # 52	Obj =	NOT3 Output to :	BUF6 Stuck-at-one	1.245
Fault # 53	Obj =	AND3 Fails to :	OR	Excl
Fault # 54	Obj =	AND3 Output to :	BUF5 Stuck-at-zero	2.019
Fault # 55	Obj =	AND3 Output to :	BUF5 Stuck-at-one	1.873
Fault # 56	Obj =	AND3 Output to :	BUF6 Stuck-at-zero	2.111
Fault # 57	Obj =	AND3 Output to :	BUF6 Stuck-at-one	1.922
Fault # 58	Obj =	BUF6 Output to :	QBAR Stuck-at-zero	1.731
Fault # 59	Obj =	BUF6 Output to :	QBAR Stuck-at-one	1.879
Fault # 60	Obj =	AND2 Fails to :	OR	Excl
Fault # 61	Obj =	AND2 Output to :	AND3 Stuck-at-zero	Excl
Fault # 62	Obj =	AND2 Output to :	AND3 Stuck-at-one	Excl
Fault # 63	Obj =	CLOCK Output to :	STB1 Stuck-at-zero	Excl
Fault # 64	Obj =	CLOCK Output to :	STB1 Stuck-at-one	Excl
Fault # 65	Obj =	CLOCK Output to :	EQ3 Stuck-at-zero	Excl
Fault # 66	Obj =	CLOCK Output to :	EQ3 Stuck-at-one	Excl
Fault # 67	Obj =	EQ3 Fails to :	XOR	2.187
Fault # 68	Obj =	EQ3 Output to :	AND2 Stuck-at-zero	1.038
Fault # 69	Obj =	EQ3 Output to :	AND2 Stuck-at-one	0.885
Fault # 70	Obj =	LIT7 Output to :	EQ3 Stuck-at-zero	Excl
Fault # 71	Obj =	LIT7 Output to :	EQ3 Stuck-at-one	NAF
Fault # 72	Obj =	LIT5 Output to :	BUF3 Stuck-at-zero	1.053
Fault # 73	Obj =	LIT5 Output to :	BUF3 Stuck-at-one	NAF
Fault # 74	Obj =	LIT6 Output to :	BUF4 Stuck-at-zero	NAF
Fault # 75	Obj =	LIT6 Output to :	BUF4 Stuck-at-one	1.517

FNTST Model

```
        entity FNTST
          (IN : in BIT;
          OUT : out BIT) is
        end FNTST;

        architecture ARCH of FNTST is

        process(IN,A,X,Y)
        signal A,X,Y : BIT;
        begin
s1:     A <= IN;
s2:     X <= A;
s3:     Y <= A;
s4:     OUT <= X and Y;
        end block;
        end arch;
```

```
FNTST
Object name :        IN     Object Type : INPUT
Output Width : 1
Output #1 : A

Object name :         A     Object Type : SIGNAL
Input #1 : IN    Width : 1
Output Width : 1
Output #1 : X
Output #2 : Y

Object name :         X     Object Type : SIGNAL
Input #1 : A     Width : 1
Output Width : 1
Output #1 : AND1

Object name :         Y     Object Type : SIGNAL
Input #1 : A     Width : 1
Output Width : 1
Output #1 : AND1

Object name :      AND1     Object Type : AND
Input #1 : X     Width : 1
Input #2 : Y     Width : 1
Output Width : 1
Output #1 : OUT

Object name :       OUT     Object Type : OUTPUT
Input #1 : AND1    Width : 1
Output Width : 1
```

FNTST Fault List

Fault					Time(sec)
Fault #	1	Obj =	IN Output to :	A Stuck-at-zero	0.839
Fault #	2	Obj =	IN Output to :	A Stuck-at-one	0.888
Fault #	3	Obj =	A Output to :	X Stuck-at-zero	0.475
Fault #	4	Obj =	A Output to :	X Stuck-at-one	0.523
Fault #	5	Obj =	A Output to :	Y Stuck-at-zero	0.569
Fault #	6	Obj =	A Output to :	Y Stuck-at-one	0.601
Fault #	7	Obj =	X Output to :	AND1 Stuck-at-zero	0.370
Fault #	8	Obj =	X Output to :	AND1 Stuck-at-one	0.350
Fault #	9	Obj =	Y Output to :	AND1 Stuck-at-zero	0.660
Fault #	10	Obj =	Y Output to :	AND1 Stuck-at-one	0.658
Fault #	11	Obj =	AND1 Fails to :	OR	0.618
Fault #	12	Obj =	AND1 Output to :	OUT Stuck-at-zero	0.735
Fault #	13	Obj =	AND1 Output to :	OUT Stuck-at-one	0.611

PRTY Model

```
        entity PARITY(
          A : in BIT_VECTOR(7 downto 0)
          P : out BIT) is
        end PARITY

        architecture ARCH of PARITY is

        process(A)
        begin
s1:     out <= ((a(0) xor a(1)) xor (a(2) xor a(3))) xor
                  ((a(4) xor a(5)) xor (a(6) xor a(7)));
        end process;
        end arch;
```

PRTY
Object name : A0 Object Type : INPUT
Output Width : 1
Output #1 : XOR1

Object name : XOR1 Object Type : XOR
Input #1 : A0 Width : 1
Input #2 : A1 Width : 1
Output Width : 1
Output #1 : XOR5

Object name : A1 Object Type : INPUT
Output Width : 1
Output #1 : XOR1

Object name : XOR5 Object Type : XOR
Input #1 : XOR1 Width : 1
Input #2 : XOR2 Width : 1
Output Width : 1
Output #1 : XOR7

Object name : A2 Object Type : INPUT
Output Width : 1
Output #1 : XOR2

Object name : XOR2 Object Type : XOR
Input #1 : A2 Width : 1
Input #2 : A3 Width : 1
Output Width : 1
Output #1 : XOR5

Object name : A3 Object Type : INPUT
Output Width : 1
Output #1 : XOR2

```
Object name :        A4     Object Type : INPUT
Output Width : 1
Output #1 : XOR3

Object name :      XOR3     Object Type : XOR
Input #1 : A4    Width : 1
Input #2 : A5    Width : 1
Output Width : 1
Output #1 : XOR6

Object name :        A5     Object Type : INPUT
Output Width : 1
Output #1 : XOR3

Object name :      XOR6     Object Type : XOR
Input #1 : XOR3    Width : 1
Input #2 : XOR4    Width : 1
Output Width : 1
Output #1 : XOR7

Object name :        A6     Object Type : INPUT
Output Width : 1
Output #1 : XOR4

Object name :      XOR4     Object Type : XOR
Input #1 : A6    Width : 1
Input #2 : A7    Width : 1
Output Width : 1
Output #1 : XOR6

Object name :        A7     Object Type : INPUT
Output Width : 1
Output #1 : XOR4

Object name :      XOR7     Object Type : XOR
Input #1 : XOR5    Width : 1
Input #2 : XOR6    Width : 1
Output Width : 1
Output #1 : OUT

Object name :      OUT      Object Type : OUTPUT
Input #1 : XOR7    Width : 1
Output Width : 1
```

PRTY Fault List

Fault					Time(sec)
Fault #	1	Obj =	A0 Output to :	XOR1 Stuck-at-zero	0.603
Fault #	2	Obj =	A0 Output to :	XOR1 Stuck-at-one	0.675
Fault #	3	Obj =	XOR1 Fails to :	EQUAL	0.338
Fault #	4	Obj =	XOR1 Output to :	XOR5 Stuck-at-zero	1.190
Fault #	5	Obj =	XOR1 Output to :	XOR5 Stuck-at-one	1.250
Fault #	6	Obj =	A1 Output to :	XOR1 Stuck-at-zero	1.260
Fault #	7	Obj =	A1 Output to :	XOR1 Stuck-at-one	1.255
Fault #	8	Obj =	XOR5 Fails to :	EQUAL	1.343
Fault #	9	Obj =	XOR5 Output to :	XOR7 Stuck-at-zero	0.408
Fault #	10	Obj =	XOR5 Output to :	XOR7 Stuck-at-one	0.414
Fault #	11	Obj =	A2 Output to :	XOR2 Stuck-at-zero	1.406
Fault #	12	Obj =	A2 Output to :	XOR2 Stuck-at-one	1.202
Fault #	13	Obj =	XOR2 Fails to :	EQUAL	0.374
Fault #	14	Obj =	XOR2 Output to :	XOR5 Stuck-at-zero	0.909
Fault #	15	Obj =	XOR2 Output to :	XOR5 Stuck-at-one	0.979
Fault #	16	Obj =	A3 Output to :	XOR2 Stuck-at-zero	0.789
Fault #	17	Obj =	A3 Output to :	XOR2 Stuck-at-one	0.724
Fault #	18	Obj =	A4 Output to :	XOR3 Stuck-at-zero	0.566
Fault #	19	Obj =	A4 Output to :	XOR3 Stuck-at-one	0.553
Fault #	20	Obj =	XOR3 Fails to :	EQUAL	1.170
Fault #	21	Obj =	XOR3 Output to :	XOR6 Stuck-at-zero	0.327
Fault #	22	Obj =	XOR3 Output to :	XOR6 Stuck-at-one	0.309
Fault #	23	Obj =	A5 Output to :	XOR3 Stuck-at-zero	1.066
Fault #	24	Obj =	A5 Output to :	XOR3 Stuck-at-one	0.937
Fault #	25	Obj =	XOR6 Fails to :	EQUAL	0.907
Fault #	26	Obj =	XOR6 Output to :	XOR7 Stuck-at-zero	0.421
Fault #	27	Obj =	XOR6 Output to :	XOR7 Stuck-at-one	0.419
Fault #	28	Obj =	A6 Output to :	XOR4 Stuck-at-zero	0.652
Fault #	29	Obj =	A6 Output to :	XOR4 Stuck-at-one	0.571
Fault #	30	Obj =	XOR4 Fails to :	EQUAL	1.113
Fault #	31	Obj =	XOR4 Output to :	XOR6 Stuck-at-zero	0.680
Fault #	32	Obj =	XOR4 Output to :	XOR6 Stuck-at-one	0.580
Fault #	33	Obj =	A7 Output to :	XOR4 Stuck-at-zero	0.879
Fault #	34	Obj =	A7 Output to :	XOR4 Stuck-at-one	1.001
Fault #	35	Obj =	XOR7 Fails to :	EQUAL	0.528
Fault #	36	Obj =	XOR7 Output to :	OUT Stuck-at-zero	1.556
Fault #	37	Obj =	XOR7 Output to :	OUT Stuck-at-one	1.356

SHFT VHDL Model

```
        entity SHIFT(
          CLOCK,
          CLEAR,
          LEFTIN,
          RIGHTIN,
          CONTROL,
          D0,
          D1,
          D2,
          D3    : in BIT
          RIGHTOUT,
          LEFTOUT : out BIT) is
        end SHIFT

        architecture arch of shft is

        process(CLEAR,CLOCK,Q0,Q3)
        signal
              Q0,
              Q1,
              Q2,
              Q3 : BIT
        begin
s1:     if CLEAR='0' then
s2:        Q0 <= '0';
s3:        Q1 <= '0';
s4:        Q2 <= '0';
s5:        Q3 <= '0';
        else
s6:        if not CLOCK'STABLE and CLOCK='1' then
s7:           case CONTROL is
                 when "00" =>  -- hold
                    null;
                 when "01" =>  -- SHIFT left
s8:                Q3 <= Q2;
s9:                Q2 <= Q1;
s10:               Q1 <= Q0;
s11:               Q0 <= LEFTIN;
                 when "10" =>  -- SHIFT right
s12:               Q3 <= RIGHTIN;
s13:               Q2 <= Q3;
s14:               Q1 <= Q2;
s15:               Q0 <= Q1;
                 when "11" =>  -- parallel load
s16:               Q3 <= D3;
s17:               Q2 <= D2;
s18:               Q1 <= D1;
s19:               Q0 <= D0;
              end case;
```

```
          end if;
        end if;
s20:  LEFTOUT <= Q3;              -- explicit output
s21:  RIGHTOUT <= Q0;             -- explicit output
        end process;
        end arch;
```

SHFT

```
Object name :     CLOCK     Object Type : INPUT
Output Width : 1
Output #1 : STB1
Output #2 : EQ2

Object name :      STB1     Object Type : STABLE
Input #1 : CLOCK    Width : 1
Output Width : 1
Output #1 : AND1

Object name :       EQ2     Object Type : EQUAL
Input #1 : CLOCK    Width : 1
Input #2 : LIT6     Width : 1
Output Width : 1
Output #1 : AND1

Object name :      AND1     Object Type : AND
Input #1 : NOT1     Width : 1
Input #2 : STB1     Width : 1
Input #3 : EQ2      Width : 1
Output Width : 1
Output #1 : AND2
Output #2 : AND3
Output #3 : AND4

Object name :      LIT6     Object Type : LIT:1
Output Width : 1
Output #1 : EQ2

Object name :     CLEAR     Object Type : INPUT
Output Width : 1
Output #1 : EQ1

Object name :       EQ1     Object Type : EQUAL
Input #1 : CLEAR    Width : 1
Input #2 : LIT1     Width : 1
Output Width : 1
Output #1 : BUF13
Output #2 : BUF14
Output #3 : BUF15
Output #4 : BUF16
Output #5 : NOT1
```

```
Object name :      LIT1      Object Type : LIT:0
Output Width : 1
Output #1 : EQ1

Object name :     BUF13      Object Type : BUFFER
Input #1 : LIT2    Width : 1
Input #2 : EQ1     Width : 1
Output Width : 1
Output #1 : Q0

Object name :     BUF14      Object Type : BUFFER
Input #1 : LIT3    Width : 1
Input #2 : EQ1     Width : 1
Output Width : 1
Output #1 : Q1

Object name :     BUF15      Object Type : BUFFER
Input #1 : LIT4    Width : 1
Input #2 : EQ1     Width : 1
Output Width : 1
Output #1 : Q2

Object name :     BUF16      Object Type : BUFFER
Input #1 : LIT5    Width : 1
Input #2 : EQ1     Width : 1
Output Width : 1
Output #1 : Q3

Object name :      NOT1      Object Type : NOT
Input #1 : EQ1     Width : 1
Output Width : 1
Output #1 : AND1

Object name :    LEFTIN      Object Type : INPUT
Output Width : 1
Output #1 : BUF10

Object name :     BUF10      Object Type : BUFFER
Input #1 : LEFTIN    Width : 1
Input #2 : AND2      Width : 1
Output Width : 1
Output #1 : Q0

Object name :      AND2      Object Type : AND
Input #1 : EQ3     Width : 1
Input #2 : AND1      Width : 1
Output Width : 1
Output #1 : BUF1
Output #2 : BUF4
Output #3 : BUF7
Output #4 : BUF10
```

```
Object name :        Q0     Object Type : SIGNAL
Input #1 : BUF10    Width : 1
Input #2 : BUF11    Width : 1
Input #3 : BUF12    Width : 1
Input #4 : BUF13    Width : 1
Output Width : 1
Output #1 : BUF7
Output #2 : RIGHTOUT

Object name :  RIGHTIN     Object Type : INPUT
Output Width : 1
Output #1 : BUF2

Object name :      BUF2     Object Type : BUFFER
Input #1 : RIGHTIN    Width : 1
Input #2 : AND3    Width : 1
Output Width : 1
Output #1 : Q3

Object name :      AND3     Object Type : AND
Input #1 : EQ4    Width : 1
Input #2 : AND1    Width : 1
Output Width : 1
Output #1 : BUF2
Output #2 : BUF5
Output #3 : BUF8
Output #4 : BUF11

Object name :        Q3     Object Type : SIGNAL
Input #1 : BUF1    Width : 1
Input #2 : BUF2    Width : 1
Input #3 : BUF3    Width : 1
Input #4 : BUF16    Width : 1
Output Width : 1
Output #1 : BUF5
Output #2 : LEFTOUT

Object name :  CONTROL     Object Type : INPUT
Output Width : 2
Output #1 : EQ3
Output #2 : EQ4
Output #3 : EQ5

Object name :      EQ3     Object Type : EQUAL
Input #1 : CONTROL    Width : 2
Input #2 : LIT7    Width : 2
Output Width : 1
Output #1 : AND2

Object name :      EQ4     Object Type : EQUAL
Input #1 : CONTROL    Width : 2
Input #2 : LIT8    Width : 2
Output Width : 1
```

```
Output #1 : AND3

Object name :       EQ5     Object Type : EQUAL
Input #1 : CONTROL    Width : 2
Input #2 : LIT9    Width : 2
Output Width : 1
Output #1 : AND4

Object name :      LIT7     Object Type : LIT:01
Output Width : 2
Output #1 : EQ3

Object name :      LIT8     Object Type : LIT:10
Output Width : 2
Output #1 : EQ3

Object name :      LIT9     Object Type : LIT:11
Output Width : 2
Output #1 : EQ5

Object name :      AND4     Object Type : AND
Input #1 : EQ5    Width : 1
Input #2 : AND1    Width : 1
Output Width : 1
Output #1 : BUF3
Output #2 : BUF6
Output #3 : BUF9
Output #4 : BUF12

Object name :       D0     Object Type : INPUT
Output Width : 1
Output #1 : BUF12

Object name :     BUF12     Object Type : BUFFER
Input #1 : D0    Width : 1
Input #2 : AND4    Width : 1
Output Width : 1
Output #1 : Q0

Object name :       D1     Object Type : INPUT
Output Width : 1
Output #1 : BUF9

Object name :      BUF9     Object Type : BUFFER
Input #1 : D1    Width : 1
Input #2 : AND4    Width : 1
Output Width : 1
Output #1 : Q1

Object name :       Q1     Object Type : SIGNAL
Input #1 : BUF7    Width : 1
Input #2 : BUF8    Width : 1
Input #3 : BUF9    Width : 1
```

```
Input #4 : BUF14    Width : 1
Output Width : 1
Output #1 : BUF4
Output #2 : BUF11

Object name :        D2     Object Type : INPUT
Output Width : 1
Output #1 : BUF6

Object name :      BUF6     Object Type : BUFFER
Input #1 : D2    Width : 1
Input #2 : AND4    Width : 1
Output Width : 1
Output #1 : Q2

Object name :        Q2     Object Type : SIGNAL
Input #1 : BUF4    Width : 1
Input #2 : BUF5    Width : 1
Input #3 : BUF6    Width : 1
Input #4 : BUF15    Width : 1
Output Width : 1
Output #1 : BUF1
Output #2 : BUF8

Object name :        D3     Object Type : INPUT
Output Width : 1
Output #1 : BUF3

Object name :      BUF3     Object Type : BUFFER
Input #1 : D3    Width : 1
Input #2 : AND4    Width : 1
Output Width : 1
Output #1 : Q3

Object name : RIGHTOUT     Object Type : OUTPUT
Input #1 : Q0    Width : 1
Output Width : 1

Object name :     BUF11     Object Type : BUFFER
Input #1 : Q1    Width : 1
Input #2 : AND3    Width : 1
Output Width : 1
Output #1 : Q0

Object name :      BUF7     Object Type : BUFFER
Input #1 : Q0    Width : 1
Input #2 : AND2    Width : 1
Output Width : 1
Output #1 : Q1

Object name :  LEFTOUT     Object Type : OUTPUT
Input #1 : Q3    Width : 1
Output Width : 1
```

```
Object name :      BUF1     Object Type : BUFFER
Input #1 : Q2    Width : 1
Input #2 : AND2    Width : 1
Output Width : 1
Output #1 : Q3

Object name :      BUF5     Object Type : BUFFER
Input #1 : Q3    Width : 1
Input #2 : AND3    Width : 1
Output Width : 1
Output #1 : Q2

Object name :      LIT5     Object Type : LIT:0
Output Width : 1
Output #1 : BUF16

Object name :      BUF8     Object Type : BUFFER
Input #1 : Q2    Width : 1
Input #2 : AND3    Width : 1
Output Width : 1
Output #1 : Q1

Object name :      BUF4     Object Type : BUFFER
Input #1 : Q1    Width : 1
Input #2 : AND2    Width : 1
Output Width : 1
Output #1 : Q2

Object name :      LIT3     Object Type : LIT:0
Output Width : 1
Output #1 : BUF14

Object name :      LIT2     Object Type : LIT:0
Output Width : 1
Output #1 : BUF13

Object name :      LIT4     Object Type : LIT:0
Output Width : 1
Output #1 : BUF15

Object name :     BUF 4     Object Type : BUFFER
Input #1 : Q1    Width : 1
Input #2 : AND2    Width : 1
Output Width : 1
Output #1 : Q2
```

SHFT Fault List

Fault					Time(sec)
Fault # 1	Obj =	CLOCK	Output to :	STB1 Stuck-at-zero	Excl
Fault # 2	Obj =	CLOCK	Output to :	STB1 Stuck-at-one	Excl
Fault # 3	Obj =	CLOCK	Output to :	EQ2 Stuck-at-zero	Excl
Fault # 4	Obj =	CLOCK	Output to :	EQ2 Stuck-at-one	Excl
Fault # 5	Obj =	EQ2	Fails to :	XOR	Excl
Fault # 6	Obj =	EQ2	Output to :	AND1 Stuck-at-zero	Excl
Fault # 7	Obj =	EQ2	Output to :	AND1 Stuck-at-one	Excl
Fault # 8	Obj =	AND1	Fails to :	OR	Excl
Fault # 9	Obj =	AND1	Output to :	AND2 Stuck-at-zero	Excl
Fault # 10	Obj =	AND1	Output to :	AND2 Stuck-at-one	Excl
Fault # 11	Obj =	AND1	Output to :	AND3 Stuck-at-zero	Excl
Fault # 12	Obj =	AND1	Output to :	AND3 Stuck-at-one	Excl
Fault # 13	Obj =	AND1	Output to :	AND4 Stuck-at-zero	Excl
Fault # 14	Obj =	AND1	Output to :	AND4 Stuck-at-one	Excl
Fault # 15	Obj =	LIT6	Output to :	EQ2 Stuck-at-zero	1.809
Fault # 16	Obj =	LIT6	Output to :	EQ2 Stuck-at-one	NAF
Fault # 17	Obj =	CLEAR	Output to :	EQ1 Stuck-at-zero	0.873
Fault # 18	Obj =	CLEAR	Output to :	EQ1 Stuck-at-one	0.781
Fault # 19	Obj =	EQ1	Fails to :	XOR	3.233
Fault # 20	Obj =	EQ1	Output to :	BUF13 Stuck-at-zero	1.610
Fault # 21	Obj =	EQ1	Output to :	BUF13 Stuck-at-one	1.669
Fault # 22	Obj =	EQ1	Output to :	BUF14 Stuck-at-zero	2.185
Fault # 23	Obj =	EQ1	Output to :	BUF14 Stuck-at-one	1.307
Fault # 24	Obj =	EQ1	Output to :	BUF15 Stuck-at-zero	1.356
Fault # 25	Obj =	EQ1	Output to :	BUF15 Stuck-at-one	2.371
Fault # 26	Obj =	EQ1	Output to :	BUF16 Stuck-at-zero	1.908
Fault # 27	Obj =	EQ1	Output to :	BUF16 Stuck-at-one	1.704
Fault # 28	Obj =	EQ1	Output to :	NOT1 Stuck-at-zero	1.745
Fault # 29	Obj =	EQ1	Output to :	NOT1 Stuck-at-one	1.654
Fault # 30	Obj =	LIT1	Output to :	EQ1 Stuck-at-zero	NAF
Fault # 31	Obj =	LIT1	Output to :	EQ1 Stuck-at-one	0.415
Fault # 32	Obj =	BUF13	Output to :	Q0 Stuck-at-zero	2.421
Fault # 33	Obj =	BUF13	Output to :	Q0 Stuck-at-one	2.711
Fault # 34	Obj =	BUF14	Output to :	Q1 Stuck-at-zero	1.036
Fault # 35	Obj =	BUF14	Output to :	Q1 Stuck-at-one	2.924
Fault # 36	Obj =	BUF15	Output to :	Q2 Stuck-at-zero	1.750
Fault # 37	Obj =	BUF15	Output to :	Q2 Stuck-at-one	1.730
Fault # 38	Obj =	BUF16	Output to :	Q3 Stuck-at-zero	1.369
Fault # 39	Obj =	BUF16	Output to :	Q3 Stuck-at-one	1.423
Fault # 40	Obj =	NOT1	Fails to :	FBUF	1.306
Fault # 41	Obj =	NOT1	Output to :	AND1 Stuck-at-zero	1.979
Fault # 42	Obj =	NOT1	Output to :	AND1 Stuck-at-one	1.929
Fault # 43	Obj =	LEFTIN	Output to :	BUF10 Stuck-at-zero	2.984
Fault # 44	Obj =	LEFTIN	Output to :	BUF10 Stuck-at-one	1.055
Fault # 45	Obj =	BUF10	Output to :	Q0 Stuck-at-zero	2.053
Fault # 46	Obj =	BUF10	Output to :	Q0 Stuck-at-one	1.705
Fault # 47	Obj =	AND2	Fails to :	OR	1.919

Fault # 48	Obj =	AND2 Output to :	BUF1 Stuck-at-zero	1.552
Fault # 49	Obj =	AND2 Output to :	BUF1 Stuck-at-one	1.742
Fault # 50	Obj =	AND2 Output to :	BUF4 Stuck-at-zero	1.470
Fault # 51	Obj =	AND2 Output to :	BUF4 Stuck-at-one	1.448
Fault # 52	Obj =	AND2 Output to :	BUF7 Stuck-at-zero	2.082
Fault # 53	Obj =	AND2 Output to :	BUF7 Stuck-at-one	2.015
Fault # 54	Obj =	AND2 Output to :	BUF10 Stuck-at-zero	1.925
Fault # 55	Obj =	AND2 Output to :	BUF10 Stuck-at-one	1.842
Fault # 56	Obj =	Q0 Output to :	BUF7 Stuck-at-zero	1.839
Fault # 57	Obj =	Q0 Output to :	BUF7 Stuck-at-one	1.693
Fault # 58	Obj =	Q0 Output to :	RIGHTOUT Stuck-at-zero	1.570
Fault # 59	Obj =	Q0 Output to :	RIGHTOUT Stuck-at-one	0.625
Fault # 60	Obj =	RIGHTIN Output to :	BUF2 Stuck-at-zero	1.651
Fault # 61	Obj =	RIGHTIN Output to :	BUF2 Stuck-at-one	1.870
Fault # 62	Obj =	BUF2 Output to :	Q3 Stuck-at-zero	2.291
Fault # 63	Obj =	BUF2 Output to :	Q3 Stuck-at-one	2.095
Fault # 64	Obj =	AND3 Fails to :	OR	0.483
Fault # 65	Obj =	AND3 Output to :	BUF2 Stuck-at-zero	2.177
Fault # 66	Obj =	AND3 Output to :	BUF2 Stuck-at-one	2.256
Fault # 67	Obj =	AND3 Output to :	BUF5 Stuck-at-zero	1.201
Fault # 68	Obj =	AND3 Output to :	BUF5 Stuck-at-one	1.173
Fault # 69	Obj =	AND3 Output to :	BUF8 Stuck-at-zero	1.274
Fault # 70	Obj =	AND3 Output to :	BUF8 Stuck-at-one	1.229
Fault # 71	Obj =	AND3 Output to :	BUF11 Stuck-at-zero	1.497
Fault # 72	Obj =	AND3 Output to :	BUF11 Stuck-at-one	1.406
Fault # 73	Obj =	Q3 Output to :	BUF5 Stuck-at-zero	1.864
Fault # 74	Obj =	Q3 Output to :	BUF5 Stuck-at-one	1.973
Fault # 75	Obj =	Q3 Output to :	LEFTOUT Stuck-at-zero	1.565
Fault # 76	Obj =	Q3 Output to :	LEFTOUT Stuck-at-one	1.405
Fault # 77	Obj =	CONTROL Output to :	EQ3 Stuck-at-zero	1.841
Fault # 78	Obj =	CONTROL Output to :	EQ3 Stuck-at-one	2.031
Fault # 79	Obj =	CONTROL Output to :	EQ4 Stuck-at-zero	1.963
Fault # 80	Obj =	CONTROL Output to :	EQ4 Stuck-at-one	1.914
Fault # 81	Obj =	CONTROL Output to :	EQ5 Stuck-at-zero	1.498
Fault # 82	Obj =	CONTROL Output to :	EQ5 Stuck-at-one	1.573
Fault # 83	Obj =	EQ3 Fails to :	XOR	3.061
Fault # 84	Obj =	EQ3 Output to :	AND2 Stuck-at-zero	1.679
Fault # 85	Obj =	EQ3 Output to :	AND2 Stuck-at-one	1.473
Fault # 86	Obj =	EQ4 Fails to :	XOR	2.079
Fault # 87	Obj =	EQ4 Output to :	AND3 Stuck-at-zero	0.930
Fault # 88	Obj =	EQ4 Output to :	AND3 Stuck-at-one	1.033
Fault # 89	Obj =	EQ5 Fails to :	XOR	1.564
Fault # 90	Obj =	EQ5 Output to :	AND4 Stuck-at-zero	1.173
Fault # 91	Obj =	EQ5 Output to :	AND4 Stuck-at-one	1.019
Fault # 92	Obj =	LIT7 Output to :	EQ3 Stuck-at-zero	NAF
Fault # 93	Obj =	LIT7 Output to :	EQ3 Stuck-at-one	1.687
Fault # 94	Obj =	LIT8 Output to :	EQ3 Stuck-at-zero	2.656
Fault # 95	Obj =	LIT8 Output to :	EQ3 Stuck-at-one	NAF
Fault # 96	Obj =	LIT9 Output to :	EQ5 Stuck-at-zero	0.725
Fault # 97	Obj =	LIT9 Output to :	EQ5 Stuck-at-one	NAF
Fault # 98	Obj =	AND4 Fails to :	OR	1.038
Fault # 99	Obj =	AND4 Output to :	BUF3 Stuck-at-zero	2.464
Fault #100	Obj =	AND4 Output to :	BUF3 Stuck-at-one	2.797

Fault #101	Obj =	AND4 Output to :	BUF6 Stuck-at-zero	1.908
Fault #102	Obj =	AND4 Output to :	BUF6 Stuck-at-one	1.942
Fault #103	Obj =	AND4 Output to :	BUF9 Stuck-at-zero	1.473
Fault #104	Obj =	AND4 Output to :	BUF9 Stuck-at-one	1.502
Fault #105	Obj =	AND4 Output to :	BUF12 Stuck-at-zero	2.111
Fault #106	Obj =	AND4 Output to :	BUF12 Stuck-at-one	2.067
Fault #107	Obj =	D0 Output to :	BUF12 Stuck-at-zero	1.461
Fault #108	Obj =	D0 Output to :	BUF12 Stuck-at-one	1.203
Fault #109	Obj =	BUF12 Output to :	Q0 Stuck-at-zero	1.464
Fault #110	Obj =	BUF12 Output to :	Q0 Stuck-at-one	1.475
Fault #111	Obj =	D1 Output to :	BUF9 Stuck-at-zero	2.300
Fault #112	Obj =	D1 Output to :	BUF9 Stuck-at-one	2.177
Fault #113	Obj =	BUF9 Output to :	Q1 Stuck-at-zero	1.123
Fault #114	Obj =	BUF9 Output to :	Q1 Stuck-at-one	0.999
Fault #115	Obj =	Q1 Output to :	BUF4 Stuck-at-zero	1.157
Fault #116	Obj =	Q1 Output to :	BUF4 Stuck-at-one	1.286
Fault #117	Obj =	Q1 Output to :	BUF11 Stuck-at-zero	1.125
Fault #118	Obj =	Q1 Output to :	BUF11 Stuck-at-one	1.019
Fault #119	Obj =	D2 Output to :	BUF6 Stuck-at-zero	2.028
Fault #120	Obj =	D2 Output to :	BUF6 Stuck-at-one	1.836
Fault #121	Obj =	BUF6 Output to :	Q2 Stuck-at-zero	1.438
Fault #122	Obj =	BUF6 Output to :	Q2 Stuck-at-one	1.287
Fault #123	Obj =	Q2 Output to :	BUF1 Stuck-at-zero	1.041
Fault #124	Obj =	Q2 Output to :	BUF1 Stuck-at-one	1.138
Fault #125	Obj =	Q2 Output to :	BUF8 Stuck-at-zero	1.382
Fault #126	Obj =	Q2 Output to :	BUF8 Stuck-at-one	1.317
Fault #127	Obj =	D3 Output to :	BUF3 Stuck-at-zero	1.242
Fault #128	Obj =	D3 Output to :	BUF3 Stuck-at-one	1.404
Fault #129	Obj =	BUF3 Output to :	Q3 Stuck-at-zero	2.484
Fault #130	Obj =	BUF3 Output to :	Q3 Stuck-at-one	0.835
Fault #131	Obj =	BUF11 Output to :	Q0 Stuck-at-zero	1.930
Fault #132	Obj =	BUF11 Output to :	Q0 Stuck-at-one	1.250
Fault #133	Obj =	BUF7 Output to :	Q1 Stuck-at-zero	2.030
Fault #134	Obj =	BUF7 Output to :	Q1 Stuck-at-one	1.047
Fault #135	Obj =	BUF1 Output to :	Q3 Stuck-at-zero	1.551
Fault #136	Obj =	BUF1 Output to :	Q3 Stuck-at-one	1.590
Fault #137	Obj =	BUF5 Output to :	Q2 Stuck-at-zero	2.084
Fault #138	Obj =	BUF5 Output to :	Q2 Stuck-at-one	2.098
Fault #139	Obj =	LIT5 Output to :	BUF16 Stuck-at-zero	NAF
Fault #140	Obj =	LIT5 Output to :	BUF16 Stuck-at-one	1.930
Fault #141	Obj =	BUF8 Output to :	Q1 Stuck-at-zero	1.735
Fault #142	Obj =	BUF8 Output to :	Q1 Stuck-at-one	1.437
Fault #143	Obj =	BUF4 Output to :	Q2 Stuck-at-zero	1.701
Fault #144	Obj =	BUF4 Output to :	Q2 Stuck-at-one	1.641
Fault #145	Obj =	LIT3 Output to :	BUF14 Stuck-at-zero	NAF
Fault #146	Obj =	LIT3 Output to :	BUF14 Stuck-at-one	1.560
Fault #147	Obj =	LIT2 Output to :	BUF13 Stuck-at-zero	NAF
Fault #148	Obj =	LIT2 Output to :	BUF13 Stuck-at-one	2.817
Fault #149	Obj =	LIT4 Output to :	BUF15 Stuck-at-zero	NAF
Fault #150	Obj =	LIT4 Output to :	BUF15 Stuck-at-one	1.517
Fault #151	Obj =	BUF 4 Output to :	Q2 Stuck-at-zero	1.922
Fault #152	Obj =	BUF 4 Output to :	Q2 Stuck-at-one	1.966

Vita

Forrest E. Norrod was born June 13, 1965 in Sacramento, California. He graduated with honors from Thomas Dale High School, Chester, Virginia, in June 1982. He was chosen as one of the 1982 Presidential Scholars and was a National Merit Scholar. Forrest entered Virginia Polytechnic Institute and State University (Virginia Tech) in September 1982, receiving a Marshall T. Hahn scholarship and graduating with a Bachelor of Science degree in Electrical Engineering in June 1986. He attended graduate school at Virginia Tech from September 1986 to February 1988, receiving a Master's of Science degree in Electrical Engineering in February 1988.

Forrest worked for Computer Sciences Corporation, Falls Church, Virginia as an applications programmer during the summers of 1982 - 84. He worked as a digital design engineer for Licom, Inc., Herndon, Virginia during the summers of 1985 and 1986.

Forrest has been employed by Hewlett Packard, Fort Collins, Colorado since March, 1988.

Forrest is a member of the IEEE and IEEE Computer Society, the ACM, and the AAAS.