AN IMPROVED CHIP-LEVEL TEST GENERATION ALGORITHM

by

Michael Douglas O'Neill

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

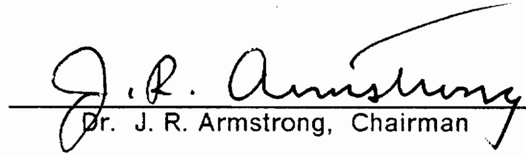
in partial fulfillment of the requirements for the degree of

Master of Science

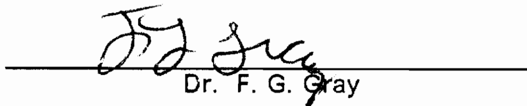
in

Electrical Engineering

APPROVED:



Dr. J. R. Armstrong, Chairman



Dr. F. G. Gray

Dr. D. P. Miller

January, 1988

Blacksburg, Virginia

LD
5655
V855
1988
V645
c.2

AN IMPROVED CHIP-LEVEL TEST GENERATION ALGORITHM

by

Michael Douglas O'Neill

Dr. J. R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

An improved algorithm for the automatic generation of test vectors from chip-level descriptions written in VHDL is described. The method offers an order of magnitude speed improvement over earlier test generation algorithms. The algorithm accepts data flow circuit discriptions written in a subset of VHDL. A fault model which defines faults for the VHDL statements is applied to determine fault cases. Test generation requirements of fault sensitization, value justification, and fault effect propagation are expressed in terms of justification, propagation, and execution goals, rather than in terms of low-level operations. Prolog rules define the way in which the goals are satisfied, using backtracking to select alternative solutions. A method for handling time in absolute, rather than relative, terms is discussed. Comparison of run times for the improved algorithm against those obtained by the previous method is made to demonstrate the speedup. Suggestions for incorporating the algorithm into a test generation system are discussed. A user's guide is given for the current implementation of the method.

Acknowledgements

I would like to thank Dr. Armstrong for providing an excellent environment in which to pursue this project, and for providing guidance during its development. I would also like to thank Dr. Gray and Dr. Miller for serving as members of my committee.

Many thanks to my wife, Stacy, for offering her support during the course of this work.

Table of Contents

1.0 Introduction 1

1.1 Contents 2

2.0 Literature Review 4

3.0 Chip-Level VHDL Models 8

4.0 Fault Model 11

5.0 Previous Method 14

5.1 Method Summary 14

5.2 Problems 17

6.0 Improved Method 19

6.1 New Approach 19

6.1.1 Level of Abstraction 19

6.1.2 Handling Time 23

6.2 Basic Operations 25
6.3 System Description 29
6.3.1 Translating VHDL to Prolog 29
6.3.2 Determining Fault Cases 31
6.3.3 Test Generation 32

7.0 Results 40
7.1.1 ADDER 40
7.1.2 ADDR2 41
7.1.3 CCNT2 42
7.1.4 CKTA 43
7.1.5 CKTCV 44
7.1.6 CNTR 45
7.1.7 CNTRV 45
7.1.8 DFF 46
7.1.9 FNTST 47
7.1.10 PRTY 47
7.1.11 SHFT 48
7.1.12 SHFTV 49
7.1.13 UARTO 49
7.2 Summary 50

8.0 Suggestions 51
8.1.1 Expansion of the VHDL Subset and Fault Model 51
8.1.2 Analyzing Circuit Function 52
8.1.3 Retention of Past Work 53
8.1.4 Avoiding Duplicate Tests 53
8.1.5 Proving Algorithm Validity 54

9.0 Conclusions 55

BIBLIOGRAPHY 56

Appendix A: User's Guide 58

9.1 Introduction 58

9.2 VHDL-to-Prolog Translation 59

9.2.1 Writing the VHDL Model 59

9.2.2 Initial Preprocessing 61

9.2.3 Rule Extraction 61

9.3 Generating a Fault List 63

9.4 Generating Tests 64

9.5 Running in Batch Mode 67

Appendix B: Circuit Models and Fault Lists 68

9.6 Adder VHDL Model 69

9.7 ADDER Fault List 70

9.8 ADDR2 VHDL Model 71

9.9 ADDR2 Fault List 72

9.10 CCNT2 VHDL Model 75

9.11 CCNT2 Fault List 77

9.12 CKTA VHDL Model 78

9.13 CKTA Fault List 79

9.14 CKTCV VHDL Model 81

9.15 CKTCV Fault List 82

9.16 CNTR VHDL Model 84

9.17 CNTR Fault List 85

9.18 CNTRV VHDL Model 87

9.19 CNTRV Fault List 88
9.20 DFF VHDL Model 89
9.21 DFF Fault List 90
9.22 FNTST VHDL Model 92
9.23 FNTST Fault List 93
9.24 PRTY VHDL Model 94
9.25 PRTY Fault List 95
9.26 SHFT VHDL Model 97
9.27 SHFT Fault List 99
9.28 SHFTV VHDL Model 101
9.29 SHFTV Fault List 102
9.30 UARTO VHDL Model 104
9.31 UARTO Fault List 105

Appendix C: Rule Definitions 106

9.32 Introduction 106
9.33 Justification 106
9.34 Propagation 106
9.35 Execution 108

List of Illustrations

Figure 1. Tester timing model 9

Figure 2. Goal tree comparison 22

Figure 3. Propagation example: Propagate 1/0 from IN 28

Figure 4. VHDL fragment for controllability measurement 30

Figure 5. Prolog justification rule. 31

Figure 6. Two-phase tests 38

Figure 7. Directory structure 58

Figure 8. Allowable operations 60

Figure 9. VHDL fragment for DONT_DISTURB example 65

1.0 Introduction

Recent advances in VLSI technology have made possible the development of powerful and complex digital integrated circuits. A single chip may contain several hundred thousand logic gates, performing tasks of high functional complexity. In designing circuits of such high complexity, the amount of detailed information which must be handled quickly becomes unreasonable. For this reason, logic designers have begun to shift to design methodologies which work at higher levels of abstraction whenever possible. Hardware Description Languages (HDL's) provide a convenient tool for functional specification and simulation of logic circuits, reducing the quantity and detail of the information which the designer must manage. Automated circuit synthesis tools have been developed which further remove the designer from the gate-level view of the device. Tools such as these allow a designer to work with a device model which contains only the level of information which his present task requires. This approach is critical if VLSI devices are to be designed and built in reasonable time periods.

While much work is being done to develop high-level design tools, high-level fault models and test generation tools have not been as well developed. Gate-level test generation algorithms, which have worked well in the past on combinational logic or small sequential circuits, require huge amounts of computer resources to handle VLSI devices. In order to avoid the problems associated with gate-level test generation methods, a test engineer often must rely on functional test guidelines, which

attempt to exercise the functionality of the circuit. As a result, the test process may become either quite expensive and time consuming, or it may be somewhat incomplete.

Recently, Barclay developed a test generation algorithm which worked from chip-level device descriptions written in VHDL [1], [2]. A chip-level fault model was defined, and artificial intelligence techniques of goals and goal-solving were used to find test vectors. While the algorithm successfully found tests for faults in a circuits of various types, the speed performance of the algorithm was not satisfactory. This thesis describes an improved algorithm, derived from Barclay's method, which offers better speed performance.

1.1 Contents

Chapter II, "Literature Review", describes briefly some other recent approaches to test generation from HDL models.

Chapter III, "Chip-Level VHDL Models", defines the chip-level models, and the VHDL constructs used to build them, which are used by the algorithm.

Chapter IV, "Fault Model", defines the chip-level fault model used to define fault cases based on the VHDL descriptions.

Chapter V, "Previous Method", summarizes Barclay's test generation algorithm. Major factors affecting algorithm performance are identified and discussed.

Chapter VI, "Improved Method", discusses the approach taken in the new algorithm. The test generation process is described, and both advantages and difficulties with the method are discussed.

Chapter VII, "Results", demonstrates the speed improvement gained by applying the new algorithm to a set of circuit models used by Barclay.

Chapter VIII, "Suggestions", identifies areas in which the test generation process could be enhanced and areas for future work.

Chapter IX, "Conclusions", provides some general conclusions about the new algorithm.

Appendix A, "User's Guide", is designed to be a reference for those using the current implementation of the algorithm.

Appendix B, "Circuit Models and Fault Lists", shows the VHDL models used and gives the fault lists for each model. Individual run times for both Barclay's method and the new algorithm are given.

Appendix C, "Rule Definitions", formally defines the Prolog rules that are used to implement the major functions used in the algorithm: justification, propagation, and execution.

2.0 Literature Review

Gate-level test generation methods are not practical for VLSI circuits for several reasons. First, it is becoming more common for users of digital integrated circuits to purchase off-the-shelf units, for which implementation details may not be available. Second, chip designers typically do not have a gate level model available until late in the design process, at which point changes suggested by testability analysis are likely to be costly. In any case, if a gate-level model is available, the test generation process for large circuits is often made impractical by the computer run time involved.

To address these problems, test generation systems that work from functional descriptions, using HDL's, are being developed. In a functional HDL model, details of physical structure and circuit implementation are not modeled. Faults are no longer modeled as signals stuck-at-zero or stuck-at-one; instead they are defined with respect to the functional specification of the device. Information of this type is available in manufacturer data books for off-the-shelf users, and is typically available early in the design process for chip designers.

In [3], Levendel and Menon propose an extension of the D-algorithm [4] to handle functions modeled in HDL's. A circuit is modeled as a collection of transfer statements (which may include data operations) and control statements. The fault model includes inputs, outputs, and state variables stuck at 0 or 1 values, control faults

(which cause control expressions to have incorrect values, causing improper transfers of execution), and general function faults (in which a specified function fails to some other incorrect function.)

In order to accomplish value propagation, D-propagation cubes are derived for both switching and non-switching functions. Faults are inserted either in a HDL statement or at a HDL block boundary, and the D-propagation cubes are used in combination with the structure of the HDL description to propagate the fault effect to an observable output. All signal decisions made during the process are justified by applying sequences to the primary inputs of the circuit.

In [5], Khorram describes another HDL-based test generation technique. The fault model includes stuck-at faults and operation faults; control faults are not modeled. The algorithm consists of four stages. A statement is selected and the local test inputs and expected result are defined. Test inputs are designed to check the statement's variables and operations. Next, a justification step moves the statement input requirements back to the external inputs. The statement under test is then activated, which may involve setting up conditions that control the execution of the statement. Finally, propagation moves the result of executing the statement to an external output. These steps are accomplished using only the information contained in the HDL statements. To activate a statement, it is necessary to determine the sequence of statements which must execute, and also the values needed in the conditional expressions. Justification and propagation involve locating statements that use the signal involved, and stringing together a number of transfer statements until an external input (or output) is reached.

In [6], Son and Fong describe a technique in which functional tables are created from a hardware description language model. A functional table consists of two parts. One specifies the conditions controlling statement execution, and the other defines the execution operation in algebraic terms. Four fault types are defined: data stuck-at faults, control faults, operation faults, and functional faults. Functional fault modes are specified by the user in a special Functional and Timing Language. For each fault case, values that can detect the fault locally are determined. Value justification and propagation are then accomplished with the aid of the functional tables, which essentially define the present and previous machine states in a form that is easily traversed.

[7] describes fault models and test generation at the chip level, based on the hardware description language GSP (General Simulation Program). Functional faults are classified as operation faults, data faults, control faults, and timing faults. For test generation purposes, timing faults, which cover requirements such as minimum pulse widths and data hold times, are not considered. Fault lists for operation, data, and control faults may be generated by examining the GSP description of the device under test.

Given a fault list, the technique of model perturbation is used. Model perturbation is defined as follows: if C(x) represents the correct model procedure and F(x) is the set of faulty model procedures, then

$$F(x) = [A]\ C(x)$$

represents the transformation from the correct model to a faulty one. The transfer function [A] is then the set of faults injected into the good model C(x). For each faulty model in F(x), the fault is injected into the good model and both are simulated. Inputs which detect the difference between the good and faulty models are taken as test patterns. Although this work does not define an automated test generation algorithm, it provides a foundation for much of the test generation work done with HDL's.

In [8], Norrod describes a test generation method that works with chip-level descriptions written in VHDL. A graphical representation of the VHDL description is extracted from the model. Nodes in the graph represent data items and data operations; arcs indicate data flow. A fault model is defined which includes control faults derived from the graph structure, and a set of data and operation faults. The approach employs a modified D-algorithm to accomplish value justification and propagation through the graph representation.

[1] and [2] fully describe Barclay's test generation algorithm. It is analyzed in Chapter 5 of this thesis.

3.0 Chip-Level VHDL Models

The test generation algorithm described here is designed to work with chip-level models. A chip level model is defined as a behavioral model of a block of logic in which signal path delays are modeled accurately without resorting to lower level descriptions [10]. A behavioral model does not include any implementation details about the device; essentially the chip is modeled by defining its input-output characteristics.

The language used to represent the chip-level description is VHDL (VHSIC Hardware Description Language), developed under the VHSIC program sponsored by the Department of Defense. VHDL provides a powerful and complete set of constructs for modeling at the chip level [11], [12]. At present, the models handled by the test generation algorithm are limited to data flow representations. A data flow model consists of groups of data assignment statements (which may involve data operations), linked together by control statements (IF-THEN-ELSE, CASE), which govern the flow of execution. Features of algorithmic descriptions, such as FOR loops and WAIT statements, are excluded, since a suitable fault model for these constructs has not been developed.

In VHDL, sequential behavior can be modeled through the use of the 'STABLE attribute. An expression containing 'STABLE is false at the time when a change occurs in the expression, and is true otherwise. For example, the expression

```
NOT CLOCK'STABLE
```

is true when the signal CLOCK makes a zero to one transition. This implies signal values in two adjacent time periods, which constitutes sequential behavior. All sequential activity is required to be expressed in this format; cross-connected gates or combinational feedback loops are not handled under the algorithm.

VHDL provides many constructs to perform accurate modeling of timing. However, for test generation we assume a timing model based on a simple tester. It is assumed that time is broken into periods of equal duration. The tester sets input values at the start of each time period, and observes output values at the end of each time period. The time periods must be long enough to allow all signal propagation to reach a steady state value. Figure 1 demonstrates the timing model. Note that when CLK rises, the value of D in the previous time period is sampled, and assigned to Q in the next time period. Also note that the time periods are long enough so that Q takes a 0 value by the end of the time period in which CLRBAR is asserted.

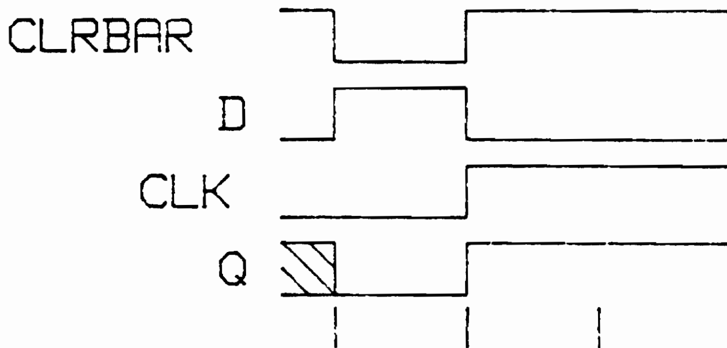


Figure 1. Tester timing model

Certain other restrictions on the VHDL subset which may be used in the circuit models are listed below:

- Objects may be of type BIT, BIT_VECTOR, or INTEGER. Signal values of 0 and 1 are implemented. Other object types (e.g. BOOLEAN signals) must be expressed in terms of these types. (Note: The algorithm also works with X values, which represent "don't care" conditions. A don't care condition is satisfied by either a 1 or a 0 value.)

- All signal assignments must use the same signal driver. This is accomplished by using only one process in the description.

- Functions and subprograms may not be used.

4.0 Fault Model

Traditional gate-level test generation methods use the stuck-at fault model to define all faults. This fault model was developed to describe fault modes in SSI TTL devices, in which the majority of faults occurred as short- or open-circuits. These faults were accurately modeled as signal lines and logic gates stuck-at-0 or stuck-at-1. However, in a chip-level model the gate-level structure of the circuit is no longer visible. A fault model for chip-level descriptions must be based on the HDL used to express the description.

Such an HDL-based fault model was described in [7] and [8]. The HDL description is viewed as a sequence of micro-operation and control structures. Micro-operations are statements which perform some transformation on data. Control structures govern the flow of execution between blocks of micro-operations. Based on this view, two main fault types are defined: micro-operation faults and control faults.

Since a micro-operation performs some function on data, a micro-operation is faulty if it performs any function other than the correct function. In the current implementation, the test generation algorithm assumes that a micro-operation fails to its logical dual. This is an arbitrary assumption; however, the algorithm is completely independent of the choice of micro-operation failure mode. Other work [13] is currently being done to determine which micro-operation failure mode provides the best fault

coverage when compared to gate-level implementations of the operations. The results of this work may easily be incorporated into the algorithm.

A fault in a control statement implies that control is not correctly transferred when that statement is executed. The wrong group of micro-operations may be executed, or none may execute at all. The fault model used here assumes that an IF statement fails to STUCK-THEN or STUCK_ELSE conditions, in which the set of micro-operations under the THEN (ELSE) clause executes regardless of the value of the control expression. For a case statement, a DEAD-CLAUSE condition is defined, in which no clause executes when the faulty clause is selected by the control expression. As explained in [2], this should provide better fault coverage than a STUCK-CLAUSE fault model. For a case statement with N clauses, N tests are required to detect all DEAD-CLAUSE faults (one for each clause.) Only 2 tests are needed to test for all STUCK-CLAUSE faults: one test executes the first clause, which checks all of the other N-1 clauses for STUCK conditions; the second test executes the second clause to check the first for a STUCK fault. It is reasonable that N chip-level tests will exercise the control logic more fully than 2 tests would.

As each assignment statement controls a data transfer operation, an assignment control (ASSNCNTL) fault is included in the model. It is assumed that in the faulty case the execution of the assignment statement does not result in a transfer of data values.

The fault cases described above are true chip-level faults, since they are derived from the chip-level HDL description. In order to improve the fault coverage along data paths, STUCK-DATA faults are also included in the fault model. Any signal value is

tested for stuck-at-0 and stuck-at-1 fault conditions. Data objects of type BIT_VECTOR are tested for all-zero and all-one STUCK-DATA conditions.

[14] gives the results of an experiment testing the validity of the fault model described above. The test vectors derived to test for all chip-level faults were applied to corresponding gate-level models, and the resulting stuck-at fault coverage measured. It was found that, on average, approximately 92% of gate-level stuck-at faults were covered by the chip-level test vectors, which supports the validity of the fault model.

5.0 Previous Method

5.1 Method Summary

As mentioned above, a chip-level test generation algorithm was developed by Barclay. The requirements for forming a test are expressed as a set of initial goals. Each goal is then broken into a set of subgoals, until the point at which each goal is directly solvable. A goal is considered solvable when it involves the setting of an external input or the observation of an external output, as these are functions under the control of the tester.

Barclay defined the following basic goals, which are used during the solving process:

1. VIO (Value In Object): Place **value** in **object** at a given **time.**
2. VIE (Value In Expression): Place **value** in **expression** at a given **time.**
3. EXEC (Execute): Execute a given **statement** at a given **time.**
4. DNE (Do Not Execute): Avoid executing **statement** at a given **time.**
5. EXG (Execute Given): Execute **statement** at a particular **time**, given that one or more other statements execute.
6. OBSOBJ (Observe Object): Observe **value** in **object** at a given **time.**
7. OBSEXPR (Observe Expression): Observe **value** in **expression** at a given **time.**
8. OBSEXEC (Observe Execution): Observe effect of executing **statement** at a given **time.**

9. DND (Do Not Disturb): Avoid disturbing **value** in **object** at some **time**.
10. TR (Time Relation): Specify a relation between two time periods.

The set of goals listed above are used to realize the operations that accomplish fault sensitization, value justification, and fault effect propagation. The steps involved in the algorithm are explained next.

Since the algorithm is implemented in Prolog, the first step is to translate the VHDL source code into a set of Prolog rules. This involves extracting basic information from the model, such as statement types, signal names and types, and expression information. Following this, a set of fault cases for the circuit is determined, based on the fault model described above. For each fault case, a set of basic test goals is specified. These basic goals define the local operations of fault sensitization, value justification, and fault effect propagation. These three steps are automated and performed once, before the solving for test vectors begins.

The goal-solving algorithm begins by examining the set of basic goals for a given test. An initial goal is selected, and it is solved by breaking it into an appropriate set of subgoals. These subgoals are added to a list, called the unsolved goal list, which contains all goals remaining to be solved. The unsolved goal list is sorted each time new goals are added, and the goal on the top of the list is chosen next for solving. The sort is accomplished based on a set of heuristics developed over a period of time by observing the solving process. The sort attempts to order the unsolved goal list in such a way as to minimize the number of conflicts between goal solutions. (A conflict occurs when two goals attempt to assert mutually exclusive events in the same time period.)

When a conflict occurs, backtracking is used to recover from the bad choice. A backtracking mechanism is built into the Prolog environment. When a goal fails due to a conflict, Prolog will automatically return to the last solved goal, undo it, find an alternative solution, and continue.

The requirements for generating a test, together with all of the alternate solutions that exist, may be represented in an AND-OR goal tree. In such a tree, each goal is represented by a node. Sets of alternative solutions are joined together by an OR node, while AND nodes connect sets of subgoals which all must be satisfied in order to solve a parent goal. In Barclay's implementation, the goal tree is represented explicitly. The unsolved goal list contains goals which must be solved in order to find a test vector (AND-connected nodes in the tree), while the backtracking mechanism is used to traverse alternative solutions (OR-connected nodes) when necessary. The order in which alternative solutions are considered is determined by simple controllability and observability measures, as well as through some heuristic guidance.

When the initial test goals have been reduced to a set of solvable goals, the test vector may be extracted from the goal tree. The final set of goals specify a list of action to be taken (setting input values and observing output values), each assigned to a relative time period. The set of all TR (Time Relation) goals gives a set of constraints on the ordering of the relative time periods. As the final step in the solving process, an actual ordering of the relative time periods must be found. The number of orderings of the events that are compatible with the set of time constraints is often quite large. However, conflicting assignments to the same signal in the same time period are not allowed. Also, it is desirable to find the shortest test vector, in terms

of the number of time periods required to apply the test. These further restrictions make the finding of a time ordering more difficult.

5.2 Problems

Barclay's implementation of his algorithm was able to generate tests for a variety of small circuits. However, the run time required to execute the algorithm was quite large. In this section, two main factors affecting the run time performance of the algorithm are identified.

The first factor concerns the level of abstraction at which the goal space is handled. Barclay opted to represent the goal tree explicitly, storing and manipulating the set of goals to be solved. Essentially, a best-first search is used to traverse the goal space. A certain amount of overhead is incurred in handling the goal list and in applying the evaluation heuristics, since each time a goal is solved, the list must be resorted, checked for new incompatibilities among goals, and simplifications to expressions contained in the goals must be made. As the length of the unsolved goal list grows, this overhead increases at a very rapid rate.

The best-first solving order also has an effect on backtracking performance. Prolog provides a blind backtracking mechanism: when a goal fails, the last solved goal is always undone first. No attempt is made to determine which previously solved goal is related to the conflict. Thus, Prolog's backtracking scheme works best when related goals are solved in sequence. The best-first ordering often means that related goals are not solved in order so that backtracking travels through a number of unre-

lated goals before reaching one that is related to the conflict. This is an important consideration, as unnecessary backtracking adds significantly to run time.

The second factor is the way in which time is handled. During the goal solving process, all time periods assigned are relative. TR goals then specify a set of constraints on the ordering of the relative time periods. For instance, a TR goal may require that t0 occur before t1. As mentioned above, the last step in solving for a test vector is to find a time ordering of test events that does not contain any conflicts. By handling time in this fashion, it is hoped that many conflicts can be avoided during the goal solving. However, as the list of events and the corresponding list of time relations grows longer, the process of finding a suitable ordering requires more and more backtracking. If no proper ordering of the set of events is found, backtracking returns to the goal solving process and finds a new set of events. As noted earlier, excessive backtracking is very costly in terms of run time.

6.0 Improved Method

The new method directly addresses the two main problems identified in the previous chapter. This chapter outlines the approach taken in overcoming these difficulties, defines the basic tools employed by the method, and discusses the test generation system which implements the new approach.

6.1 New Approach

6.1.1 Level of Abstraction

In Barclay's method, the elements manipulated by the algorithm are the ten basic goal types defined previously. Since these goal types describe basic operations such as setting signal values, observing expression values, etc., they are referred to as low-level goals. In order to accomplish the three main tasks involved in generating a test (fault sensitization, value justification, and fault syndrome propagation), the algorithm expresses the tasks in terms of these low-level goals. As discussed above, working at this level involves a considerable amount of overhead, which grows quickly as the number of goals increases. The best-first sort used to order the goals also has an effect on backtracking performance.

In the new approach, a higher level view is taken. The low-level goal types used by Barclay's method are no longer represented explicitly. Instead, the higher level op-

erations of justification, propagation, and execution are considered to be basic elements. (These basic operations are described in more detail in the next section.) Each of these operations is represented explicitly, and the tasks of fault sensitization, value justification, and fault syndrome propagation are expressed in terms of these high-level operations. The solving process then considers only the high-level operations when selecting among alternatives during the generation of a test.

Justification, propagation, and execution are explicitly represented and handled during the solving process. Sets of rules define how each operation is to be carried out under the various circumstances that arise during test generation. The basic goals of Barclay's method are now represented implicitly in the flow of program execution. That is, the operations performed by the basic goals are accomplished during the solving of the high-level rules, but the basic goals are not treated as an elemental unit and are never explicitly manipulated.

Figure 2 on page 22 demonstrates this idea. Goal tree A represents the solution tree as found by Barclay's algorithm. Each node in the tree represents one of the low-level goal types. Children connected to a parent node represent the subgoals which must be solved in order to satisfy the parent goal. Overhead accumulates during the solving process because the list of unsolved goals must be handled at each node.

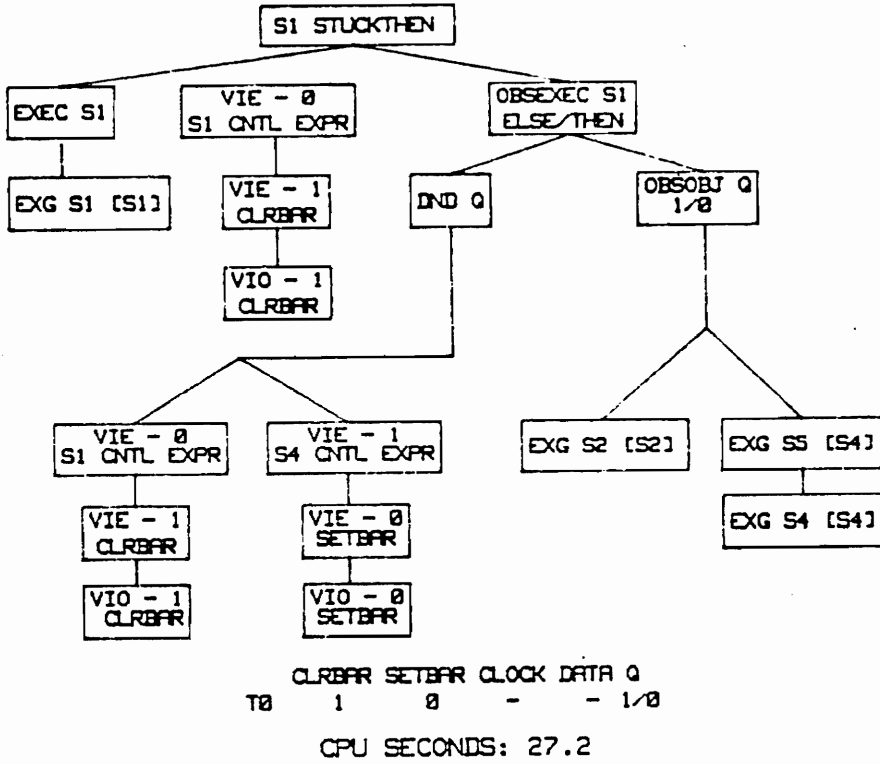
Goal tree B is the solution tree traversed by the new algorithm. Each node in this tree is a high-level operation, which is solved in a depth-first manner. The order in which subgoals are solved is determined by the rules defining each operation, with backtracking used to recover from bad choices. No list of unsolved goals is maintained, thus no overhead due to sorting such a list is incurred. The same result is produced

by both methods, but the new method is able to find a solution more quickly due to the reduced overhead involved in handling goals.

As mentioned above, the requirements for justification, propagation, and execution are solved in a depth-first fashion. This is contrasted with the best-first ordering used in Barclay's approach. In both methods, the built-in backtracking mechanism of Prolog is used to recover from bad choices. Prolog's backtracking method is "blind". Whenever a goal fails do to a conflict, the last solved goal is undone. If an alternative solution exists, it is chosen and goal solving continues. If no alternative solutions exists, Prolog moves to the last goal solved before that one and tries to find an alternate solution. No attempt is made to discover whether or not the solved goal which is being undone has any relation to the conflict which originally caused the failure; hence the backtracking is termed "blind".

A blind backtracking algorithm performs better if goals that are related to each other (and therefore could conflict with one another) are solved in order. Then, if backtracking is necessary, the source of the conflict is uncovered immediately. In the best-first ordering, related goals are not necessarily solved in order. Often, a number of unrelated goals will have to be undone before the goal related to the conflict is reached. The time spent backtracking through unrelated goals is essentially wasted. In a depth-first approach, related goals are solved in sequence, so that backtracking can typically discover the source of the conflict without undoing many unnecessary goals.

The result of the depth-first ordering is that when an operation is chosen for solving, the algorithm will quickly accomplish the task, or discover that it cannot be performed; contrast this with a best-first ordering, in which several operations are con-



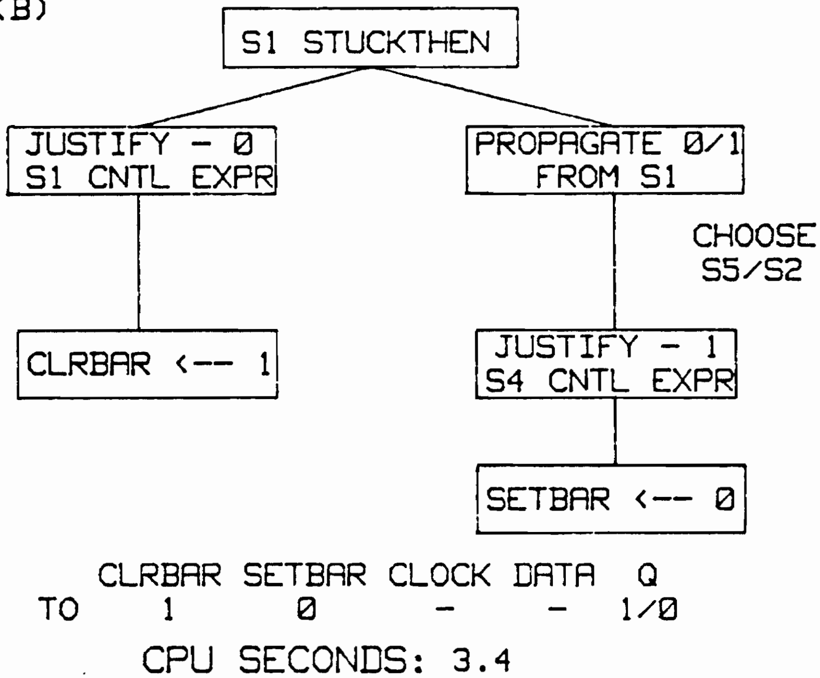


Figure 2. Goal tree comparison

sidered simultaneously, and it may take extra backtracking to find a solution or determine that a given operation is impossible. If, however, after the depth-first method completes an operation it is determined that the operation must be undone in its entirety, the backtracking mechanism will cause all alternative solutions to be tried before the operation is completely undone. This is due to the blind nature of the backtracking mechanism, and cannot be avoided without modifying the nature of the backtracking mechanism.

6.1.2 Handling Time

In Barclay's method, time is handled in relative terms during the goal solving process. Actual time periods are fixed after all operations required for the test have been determined. The benefit of using relative time periods during goal solving is that it is often possible to avoid excessive numbers of conflicting assignments; i.e. it may be easier to determine the operations that must be performed first, and then to order them in time later. As discussed in Chapter 5, however, the ordering process often consumes a large amount of time.

To address this problem, the new method handles time in absolute terms during goal solving. An event queue is maintained, and events are inserted into the time queue as soon as they arise. Each event in the queue is assigned a time tag. Thus, when the goal solving process is complete, the events in the queue have a fixed time ordering, and no further processing is required.

In order to properly maintain the event queue, the algorithm must have the ability to insert new events at appropriate points in the queue. An appropriate point is one

which does not generate any conflicting assignments, makes the value which it is assigning available when needed, and does not destroy the value established by another event before it is used. If no conflicts are generated, more than one event may be assigned to the same slot in the queue. Since each slot represents one time period in the test, combining events will reduce the length of the final test vector, which is desirable. When this is not possible, empty slots are created between existing events, and new events are inserted at this point. However, care must be taken so that events which must remain adjacent are not split apart. For example, if the expression NOT CLOCK'STABLE is to be true at t_2, the signal CLOCK must take the value 0 at t_1 and the value 1 at t_2. If any events are inserted between t_1 and t_2, the value in the expression will be disturbed. To avoid this, time periods which must remain adjacent are considered *linked*, and the algorithm will not be allowed split them.

In order to allow events to be inserted between time periods that are not linked, "empty" events are placed in between existing, non-linked events. When inserting new events, there is the possibility that the new event will destroy the effect of an existing event. Thus, after a given value is justified in an expression, a check is run to see that the loading of the objects involved in the expression did not disrupt each other.

Handling time in this fashion greatly reduces the time required to determine an ordering of the events that make up a test. When conflicts arise during the solving process, however, there are now two possible causes: either the value involved is incorrect, or the time at which the operation occurs is incorrect. When a conflict occurs, the algorithm cannot determine which problem has caused it. Therefore, every

effort is made to select a good time period in which to attempt an operation, and all possible value combinations will be tried before a new time is tried.

6.2 Basic Operations

This section describes the three basic operation employed by the new test generation algorithm. The tasks of justification, propagation, and execution are used to specify the initial requirements for a test, and they are also the basic goals manipulated by the algorithm during the solving process.

Justification : Given an expression (which may be a complex combination of objects or as simple as a single object), and a desired value, the justification operation works backwards to determine the set of external input values which will result in placing the value in the expression. For single objects, the set of rules formed in the pre-processing stage are used to accomplish the justification. Each rule specifies the justification requirements that will set the source expression to the proper value, and the statement which must be executed in order to transfer the value from the source expression to the object.

For expressions, the first step is to determine a set of values for the objects involved in the expression such that the desired result is produced when the expression is evaluated. Often there will be more than one possible set of values to accomplish this; the alternate choices are considered when backtracking occurs. Once values have been chosen for the objects that make up the expression, the justification rules for single objects are called.

An initial justification goal can require solving several justification and execution subgoals. These subgoals are satisfied in depth-first fashion, and the order in which they are attacked is completely specified by the justification rules. This removes any overhead involved in sorting the subgoals to determine the next subgoal to be solved. It does require, however, that the set of rules governing justification allow enough flexibility to recover from a incorrect first choice in solving order.

When a justification goal has been completely satisfied, the result is a set of input values which, when applied to the device, will cause the given value to be placed in the desired expression.

Figure 3 on page 27 demonstrates a justification example. Assume that the value '1' is to be placed in the signal OUT, using statement s5. The first step is to determine the values that must be placed in the signals that make up the source expression of statement s5. In order for the expression Q1 and Q2 to have a value of '1', the signals Q1 and Q2 must take the value '1'. Working with Q1 first, statement s2 can be used to place the value '1' in Q1. The source expression of statement s2, which is the single signal A, must have the value '1'. Since A is an input, we require a value of '1' on this input pin. To execute statement s2, the IF statement in statement s1 must be executed with the control expression set to a true ('1') value. This is accomplished by causing a zero-to-one transition on the CLK1 signal. Similarly, a value of '1' can be placed in the signal Q2 by placing a '1' value on the input pin D2, and causing a zero-to-one transition on the signal CLK2. Note that statement s5 always executes, so that the signal OUT takes the value '1' as soon as both Q1 and Q2 have the '1' value. The table in Figure 3 on page 27 summarizes the justification operation.

```
s1:     if CLK1='1' and not CLK1'STABLE then
s2:       Q1 <= D1;
s3:     if CLK2='1' and not CLK2'STABLE then
s4:       Q2 <= D2;
s5:     OUT <= Q1 and Q2;
```

CLK1	CLK2	D1	D2	OUT
0	0	1	1	-
1	1	-	-	1

Propagation : In order to observe any object value or the result of executing a statement, a value must be moved to an external output. Propagation accomplishes this task. Given a good/bad value pair (good = value which appears if the fault is not present, bad = value which appears if the fault is present), and the initial location in the VHDL model, propagate selects a path to an external output, and specifies all inputs necessary to move the value pair along this path. The easiest propagation method is to determine a series of assignment statements which will transfer values to an output. If such a path exists, all statements are executed in order. Each assignment statement may have an expression involved, and the value being propagated will only be one part of the expression. Values must be chosen for the other objects in the expression such that the good/bad result will be maintained. Justification routines are called to supply these values.

If a series of assignment statements cannot be found, a control statement must be used to propagate the value pair. To accomplish this, the expression in the control statement is evaluated to see which clause executes if the fault is present, and which clause executes if it is not. Assignment statements under both the good and bad clauses are then chosen and used to continue the propagation of the good/bad pair.

If a suitable assignment statement does not exist under both the good and bad clauses, one statement is chosen for propagation. A justification operation then places the bad value in the object which the assignment statement transfers data to, and then the chosen statement is executed. The result of this operation is that the assignment statement will be executed if the fault is not present, and good value is observed; if the fault is present, the assignment statement will not be executed, and the bad value will remain stored in the object.

Figure 3 demonstrates a propagation operation. Assume that the good/bad value pair 1/0 is present in the signal IN. To move the effect of this value pair to the output Q, s5 is executed, transferring the 1/0 pair to ENABLE. Then, statement s10 is chosen to propagate the value in ENABLE. Since there are no assignment statements in the ELSE clause of s10, the value Q is first preloaded with a zero value by executing statement s2. The 0 now serves as the bad value. Setting DATA to 1, we can observe the output Q: if the fault is not present, s11 executes, and Q will have value 1. If the fault exists, s11 will not execute, and Q will retain the 0 value.

```
s1:     if CLEAR='0' then
s2:       Q <= 0;
s3:     if (CLOCK='1' and not CLOCK'STABLE) then
s4:       case CONTROL is
            when"00" =>
s5:           ENABLE <= IN;
              .
              .
s10:    if ENABLE='1' then
s11:      Q <= DATA;
```

CLEAR	CLOCK	CONTROL	IN	DATA	Q	
0	-	-	-	-	-	: Clear (Q <= 0)
-	0	00	1	-	-	: Set ENABLE to 1
-	1	-	-	1	1/0	: Observe Q

Figure 3. Propagation example: Propagate 1/0 from IN

Execution : Given a statement, the execution operation determines the sequence of control statements (if any) which govern the execution of the given statement. Then, the values to which each control expression must be set are determined. For each control expression, a justification operation is specified, placing the required value in the expression. When all values for control expressions have been properly justified to the external inputs, the result is a sequence of input values which will cause the specified statement to execute.

6.3 System Description

The new algorithm has been implemented in Prolog as part of a test generation package. The system is organized so that the user may provide a VHDL source file as input, and receive a list of faults and corresponding test vectors as output. This section describes the test generation system.

6.3.1 Translating VHDL to Prolog

Since the test generation system is written in Prolog, the information contained in the VHDL model is first translated to a Prolog representation. There are two steps in the translation. First, basic information is extracted from the VHDL statements. A list of statements, the type of each statement (IF, CASE, ASSIGNMENT), the expressions involved in each statement, the number, type, and use of signals, etc. is compiled. Also, the controllability and observability of each signal is measured. The controllability/observability of a signal is measured by counting the number of statements which must be execute in order to control/observe the signal. Each statement which must be executed is assigned a weight based on the estimated difficulty of

actually executing the statement. For simplicity, a value of 5 is added for each control statement which must be set to execute a given assignment statement. For example, in Figure 4 on page 30, the signal B would have a controllability of 7, since two assignment statements (s2 and s10) plus one control statement (s1) must be used to move a value from an external input (DATA_IN) to B. Controllability/observability measures are used to choose paths for value justification and fault syndrome propagation. Thus, they play an important role in determining the efficiency of the algorithm.

```
s1:  if CLOCK='1' and not CLOCK'STABLE then
s2:     A <= DATA_IN;
        .
        .
s10: B <= A and C;
        .
        .
```

Figure 4. VHDL fragment for controllability measurement

The second stage in the translation involves extracting a set of justification rules from the VHDL model. For each assignment statement, a rule is extracted which details the conditions under which the statement may be used to perform value justification. During the solving process, these rules provide a concise method of expressing, at a high level, justification requirements. Such rules are handled with a minimum of overhead, as Prolog will search the rule database and find the appropriate rule when performing justification. Figure 5 on page 31shows a sample VHDL fragment and the justification rule derived from it. The rule states that in order to place the value **Value** into the object **Q** at time **Time**, **Value** must first be placed in the source expression of statement s2, and this value must also be justified to external inputs. After the proper value has been placed in the source expression, the statement must be executed in order to accomplish the data transfer. The last portion of the rule will insure

that other statements in the model which could corrupt the value placed in **Q** will not be executed.

In addition to justification rules for each assignment statement, rules are formed for each input signal in the model. Since input signals are directly controllable, no further justification or execution of statements is required. Instead, a note is made (in the form of a Prolog fact inserted into the data base) concerning the name of the input signal, the value to which it must be set, and the time at which to perform the action.

VHDL Fragment:

```
s1:if CLOCK='1' and not CLOCK'STABLE then
s2:  Q <= OBJECT_1 xor OBJECT_2;
```

Prolog Justification Rule

```
q(Value,Time) :-
    justify(Value, [xor [obj, OBJECT_1], [obj, OBJECT_2]], Time),
    execute(s1, Time),
    dont_disturb(q, Time).
```

Figure 5. Prolog justification rule.

Both translation steps are automated. The first translation stage is accomplished by a program written in the C programming language. A Prolog program performs the second stage. These two preprocessing steps a performed one time for each model, and the results are stored in files of Prolog rules.

6.3.2 Determining Fault Cases

After preprocessing the model into a Prolog format, a list of fault cases for the model is determined. Based on the fault model described in Chapter 3, the following fault types are extracted:

- ASSNCNTL - An assignment control fault is specified for each assignment statement in the description.
- STUCKTHEN - For each IF statement, a STUCK-THEN fault case is listed.
- STUCKELSE - For each IF statement, a STUCK-ELSE fault case is listed.
- DEADCLAUSE - A DEAD-CLAUSE fault condition is specified for each clause of each CASE statement in the model.
- MICROOP - Every operation in each expression is faulted to its dual operation.
- STUCKDATA - Each data instance (signal and operation result) is faulted to stuck-at-0 and stuck-at-1 fault cases. (For vectors, stuck-at-all-zeros and stuck-at-all-ones are used.)

The first five fault types are true chip-level faults, derived from the VHDL model itself. STUCKDATA faults correspond to traditional stuck-faults on data paths, and are included to improve the total fault coverage.

The list of faults for the model is extracted once, and stored in a data file.

6.3.3 Test Generation

The test generation algorithm works with one fault at a time, and derives a test vector for the given fault. To begin the goal solving process, the fault sensitization, initial value propagation, and fault syndrome requirements for the test are specified in

terms of the high-level operations of justification, propagation, and execution. A discussion of the basic requirements for each fault case follows:

ASSNCNTL : In order to determine if an assignment statement accomplishes its data transfer when executed, a value must be placed in the source expression of the statement, and the statement must then be forced to execute. The destination object of the assignment may then be observed to see if a transfer of data occurred. If the transfer is successful, the value placed in the source expression will be observed. However, if the transfer does not take place, the value of the object will be undefined, as we do not know the "old" value of the object. To allow detection of the ASSNCNTL fault, a value *different from the good value* must first be placed in the destination object. Then, should the transfer fail to occur, the tester will know what value to look for as an indication of the faulty condition. The placing of the bad value in the destination object before testing the assignment statement is called *preloading*.

Based on this discussion, the basic requirements for an ASSNCNTL test are:

1. Preload the destination object with a bad value.

2. Place the good value in the source expression of the assignment statement and justify it.

3. Execute the assignment statement.

4. Propagate the good/bad value pair to an external output.

STUCKTHEN : A test for a STUCKTHEN fault condition attempts to execute the ELSE clause of the IF statement, and then finds a way to observe whether the ELSE clause actually is executed (in the good case), or whether the THEN clause executes (in the faulty case). However, a STUCKTHEN fault condition actually behaves as if the expression controlling the IF statement were stuck at a value of 1 (since the THEN clause is executed when the control expression evaluates to 1). In order to accomplish the test, all that is required is to propagate the effect of a 1/0 fault syndrome from the control expression to an external output. The rules governing propagation will select appropriate statements for observation, and will take care of causing the IF statement to be executed.

STUCKELSE : The STUCKELSE test follows the same procedure as the STUCKTHEN test. The goal is to attempt execution of the THEN clause, and then to observe which clause is actually executed. This may be specified as a 0/1 good/bad value pair in the control expression of the if statement. Propagation of the good/bad value will therefore define the STUCKELSE test.

DEADCLAUSE : To detect a DEAD-CLAUSE condition, the clause under test should be caused to execute. A data object under the clause may then be observed to determine whether or not the clause actually executes. In order to differentiate between the good and faulty cases, the data object chosen for observation must be preloaded with a bad value. In essence, this is the procedure for testing an assignment statement under the dead clause for an assignment control fault. In fact, this is the way in which a DEADCLAUSE test is specified. The dead clause is examined, and the most observable assignment statement under it is chosen. An assignment control test is then run on this statement. Note that if no assignment statement exists under the

clause being tested, there is no way to detect the fault. In this case, however, the execution of the clause in the good case produces no result, so a DEAD-CLAUSE condition for such a clause is not a fault.

MICROOP : A test for a faulty microoperation involves finding inputs to the operation which will sensitize the faulty behavior of the operation. The fault model dictates that the faulty behavior of an operation is its dual. A look up table is maintained which stores the fault case for each operation in the VHDL subset. Also, the inputs which sensitize the faulty operation mode, and the resulting good/bad output values, are stored in the table. For any given microoperation fault, then, the test requirements are as follows:

1. Retrieve the fault sensitization requirements for the operation from the table.

2. Justify the proper values from the operation inputs to external inputs.

3. Specify a propagation operation to move the good/bad output of the operation to an external output.

STUCKDATA : A STUCKDATA fault case may be viewed as the degenerate case of a MICROOP fault. Any value which is different from the stuck value will sensitize the fault. The basic test requirements are:

1. Select as a test value any value which is different from the stuck-value.

2. Specify a justification operation to justify the test value from the stuck position to external inputs.

3. Define a propagation task to move the good/bad value (good value = test value; bad value = stuck value) pair from the stuck position to an external output.

Once the test for a given fault has been described in terms of the basic operations, the rules for justification, propagation, and execution are called to satisfy the initial requirements. After the initial specification of goals, the flow of execution is left up to the depth-first technique built into the Prolog interpreter. Thus, all overhead involved in determining which goal to solve next is avoided. Also, since the list of unsolved goals is not stored by the algorithm, the new method requires less memory space while running. Some of the space savings is offset by the increased size of the rule file created during preprocessing, but the additional file space requirements are quite small in comparison with the run-time space requirements.

The next section of this chapter deals with several concerns that affect the way in which the algorithm proceeds during the goal solving process.

Two-phase tests : All tests require a known good value and a known bad value so that the absence or presence of the fault may be determined by observing an external output. As mentioned previously, in HDL models it is necessary to preload objects with a bad value when testing assignment control faults. This results in a test which requires (at least) two time periods; one for preloading and one for the actual test. This situation is called a *two-phase* test.

If a faulty assignment statement is used to accomplish the justification of a signal value, the transfer of data performed by the statement may not actually occur if the fault is present. Therefore, if a faulty statement is used for justification the transfer must be verified by propagating a good/bad value from the assignment statement to

an output. In order to do this, a two-phase test is required to create known good/bad values. In Barclay's method, this resulted in test vectors similar to the example in Figure 6 on page 38. The test method would like to perform the test by preloading C with the bad value "1111", and then use the good value "0000" to perform the test. If the only statement assigning to C is the faulty statement, the result of the preload operation must be checked. Barclay's method will then perform a two-phase test on the preload operation, which requires that the value "1111" be preloaded into C. However, this preload operation is not necessary. As shown in part B of Figure 6 on page 38, a single two-phase test may be used, provided that the result of the first step is observed. The difference lies in the fact that the first observation will check only for the good value; any value other than the good value indicates a fault. The second observation will then check for a fault case in which the faulty value is the preloaded bad value. The new method implements this type of two phase-testing.

Choice of test values : In almost every instance where a signal value is used, there is some choice in selecting the value. The choice of values is limited by any literal values involved, and by the requirements of the test; for instance, we always require that a good test value be different from the bad test value. Often the initial selection of the test value will have an effect on the time required to generate the test vector. For instance, in circuits with clear operations built in, 0 is usually a good choice, since the clear operation is typically very simple. In counters, values near the limits of the counter (i.e., "000", "001", and "111" for a three-bit counter) can be reached in fewer steps than values in the middle of the range. The nature of the algorithm is such that it will try every available option in attempting to perform the test with the selected test value before giving up and choosing a different test value. However, the test generation method does not have the ability to perform any analysis of the

VHDL STATEMENT:

```
C <= BVADD(A, B);
```

BARCLAY'S
TWO-PHASE TEST:

	A	B	C
t0	"0000"	"1111"	—
t1	"0000"	"0000"	"0000/1111"
t2	"0000"	"1111"	"1111/0000"

NEW
TWO-PHASE TEST:

	A	B	C
t0	"0000"	"0000"	"0000/XXXX"
t1	"0000"	"1111"	"1111/0000"

Figure 6. Two-phase tests

functionality of the circuit on which it is operating. For this reason, general guidelines, such as those mentioned above, are applied, and some tests which are found by the algorithm may not be optimal in length.

Expression checks: Because it deals with time in an absolute manner, the test generation algorithm must split time periods apart in order to insert new events between existing ones. During the justification process, the loading of one object may be inserted between the loading of other objects, and the possibility arises that one load could destroy the value established by another. Therefore, each time a value is justified in an expression, the expression is evaluated at the end of the operation to ensure that it still holds the proper value. The checking routine operates at the time period when the justification has been completed, and scans back in time to determine the most recent value assigned to each object in the expression. If any of the values are incorrect, backtracking will undo steps in the justification operation, attempting to correct the disturbance.

'Do not disturb' operation: In his method, Barclay defined a "do not disturb" goal type. The purpose of this goal was to block other statements from executing and destroying a value which had been established. For example, if a value has been placed in a flip-flop, the CLEAR input must be set so that the statement which performs the CLEAR operation does not execute. In the new method, this is built into the justification rules. Thus, a justification operation is not successfully completed until all statements which could affect the justified value have been examined.

7.0 Results

In this chapter, the results obtained by applying the new algorithm to thirteen circuit models used to test Barclay's method are presented. For each model, the number of tests successfully generated using Barclay's method, and the total amount of CPU time required, are listed. The total CPU time required by the new method in generating the same set of tests is given, and a speed-up measure is computed by taking the ratio of the CPU times. Also, the number of tests found by the new method which were not generated by Barclay's method is given Note that only chip-level faults which Barclay's method did not generate are considered; stuck-data faults are not included. The fault cases of note are those for which the Barclay's test generation algorithm overflowed; the increased speed performance of the new method allows the tests to be produced in most cases.

Appendix B contains listings of the VHDL source code for each circuit model. The fault list associated with each model is supplied, with an indication of which faults were successfully handled by each of the two methods.

7.1.1 ADDER

The ADDER circuit is a simple behavioral model of a four-bit adder. It is modeled by a single ADD statement; the work involved in finding tests is mainly determination of input values. Since very little, if any, time ordering is performed in generating the

tests, the speed-up is mainly due to the reduction in the overhead involved in working with low-level goal types.

Number of tests	: 9
Total CPU time (Barclay's)	: 415.9 seconds
Total CPU time (New)	: 16.4 seconds
Speed-up	: 25.4
Additional tests found	: 0

7.1.2 ADDR2

ADDR2 is a gate-level model of a four-bit adder. The circuit is combinational, and no control statements are involved. However, propagation and justification must travel through several statements which involve complex expressions. Some tests required a relatively large amount of backtracking in order to determine a suitable set of inputs that would justify the required values through the series of expressions. Again, as very little time ordering is performed, the speed-up is mainly due to the reduction in overhead. A few of the tests took longer to run under the new algorithm than under Barclay's. This is due to the selection of signal values when choices exist. In the new method, if an incorrect choice is made for a given signal, the method will try all options to make the selection succeed, and if it fails on the first attempt, a second attempt is made in a different time slot. The extra backtracking required to find the correct value accounts for the extra time.

Number of tests	: 144

Total CPU time (Barclay's)	: 11553.7 seconds
Total CPU time (New)	: 2839.9 seconds
Speed-up	: 4.1
Additional tests found	: 0

7.1.3 CCNT2

CCNT2 is a controlled up/down counter, with a limit register. (See Appendix B for the circuit description.) The use of multiple flags to control the operation of the counter make it necessary to execute many statements in order to establish the conditions required to execute the statements which increment and decrement the counter. In Barclay's method, a very large number of low-level goals must be solved in order to accomplish this; the new method demonstrates a marked performance improvement due in part to the fewer number of goals which must be manipulated.

CCNT2 also points out a problem which results from the lack of high-level information about the function of the circuit. The statements controlling the incrementing/decrementing of the counter require that the EN (counter enable) and DIR (direction - up/down) flags be set. Therefore, each time the counter is incremented or decremented, a separate justification operation sets the flags, even though setting one flag automatically sets the other (because the statements which set them are contained in the same clause.) Because the algorithm does not know that the flags may already be set, the final test vector will set the flags repeatedly. In Barclay's method, this creates such a large time penalty that most tests exceeded the CPU time limit for batch jobs or overflowed the Prolog interpreter. The new method does not

pay such a high penalty in terms of run time, so the extra operations do not have such a drastic effect. However, if a test is generated by Barclay's method, the redundant flag operations are mapped to a single time period so that they only occur once (the time required to perform the time ordering is quite large, however.) Since the new method does not perform any processing after all goals have been solved, extra flag operations remain in the test vector.

Many of the tests caused Barclay's method to overflow the interpreter. The increased speed performance of the new method allowed these tests to run to completion in reasonable periods of time.

Number of tests	: 1
Total CPU time (Barclay's)	: 6004.3 seconds
Total CPU time (New)	: 108.0 seconds
Speed-up	: 54.2
Additional tests found	: 19

7.1.4 CKTA

CKTA is a circuit consisting of two D flip-flops, with separate clocks and no preset/clear, connected in series. The outputs of the flip-flops are ANDed together to produce the circuit output (See appendix B for the circuit model.) The serial connection requires that the first flip-flop be used to load the second.

The serial connection of the two flip-flops creates the potential for events to be incorrectly inserted into the queue. Loading Q_2 requires that Q_1 be loaded, and then the second flip-flop must be clocked. These two events (loading Q_1 and clocking Q_2) must have their time periods linked so that a future cannot split them and destroy the value in Q_1 before it is loaded into Q_2 The algorithm must move any new operation involving Q_1 either before or after the loading of Q_2

Number of tests : 9

Total CPU time (Barclay's) : 5225.8 seconds

Total CPU time (New) : 129.2 seconds

Speed-up : 40.4

Additional tests found : 3

7.1.5 CKTCV

CKTCV is a multiplexor with input, output, and control registers modeled by vectors. All registers are loaded from a single input bus, which causes load operations to take several steps in order to load registers. In Barclay's method, this requires a large number of low-level goals. Much overhead is involved in handling the goal list, but more importantly, the time required to find a suitable ordering of the events after the solving process completes is very large. A great deal of backtracking takes place during the ordering of relative time periods. This step is not needed in the new method; hence a large speedup is obtained.

Number of tests : 27

Total CPU time (Barclay's)	: 73526.9 seconds
Total CPU time (New)	: 520.4 seconds
Speed-up	: 141.3
Additional tests found	: 1

7.1.6 CNTR

CNTR is a bit model of a three-bit counter with a clear operation. (A bit-level model treats each bit of the counter as a separate data signal instead of grouping the bits together as a bit vector.) The speedup in this case is typical of models which require a moderate number of low-level goals under Barclay's method.

Number of tests	: 18
Total CPU time (Barclay's)	: 7855.0 seconds
Total CPU time (New)	: 155.1 seconds
Speed-up	: 50.6
Additional tests found	: 10

7.1.7 CNTRV

CNTRV is the same three-bit counter modeled in CNTR, except that the counter is modeled as a bit vector instead of a set of individual bits. Barclay's method produced invalid tests for some faults due to the incomplete implementation of the ADD function for X (don't care) values. The new method was able to generate tests in these

cases, although the ADD function still does not handle X values. This is because the new method picked completely defined test values for the ADD function.

Number of tests	: 4
Total CPU time (Barclay's)	: 3140.6 seconds
Total CPU time (New)	: 35.2 seconds
Speed-up	: 89.2
Additional tests found	: 5

7.1.8 DFF

DFF is a D flip-flop model, with asynchronous clear and set operations. Some of the speedup obtained by the new method is related to the improved two-phase tests used in the new algorithm. The new two-phase test requires only two justification operations, while the old test required three. Also, the improved controllability/observability measure allows the method to pick the clear and set operations when loading values, instead of using the clock mechanism. All statements had an equal controllability and observability under Barclay's method.

Number of tests	: 39
Total CPU time (Barclay's)	: 5370.9 seconds
Total CPU time (New)	: 215.1 seconds
Speed-up	: 25.0
Additional tests found	: 0

7.1.9 FNTST

FNTST is a combinational circuit model designed to demonstrate the difficulties in handling reconvergent fanout. The new method does not improve on the performance of the test generation process for circuits with reconvergent fanout. Two-phase tests which occur are handled more efficiently under the new algorithm. The relatively small number of low-level goals required, combined with the fact that most tests occur in a single time period so that ordering of time periods is not required, causes the speedup factor to be correspondingly smaller.

Number of tests	: 7
Total CPU time (Barclay's)	: 348.0 seconds
Total CPU time (New)	: 28.0 seconds
Speed-up	: 12.4
Additional tests found	: 0

7.1.10 PRTY

PRTY is a gate-level, eight-bit parity generator. Complex expressions are involved in each statement, hence justification and propagation operations have many choices when assigning values to the objects involved in the expressions. As in the ADDR2 model, the speedup factor is small, since there are relatively few low-level goals required, and most tests run in a single time period. This neutralizes to a certain degree the two main areas of in which the new method makes improvements. Also, the fact that the new algorithm will try justifying a value twice (the second time in a new time

slot) before selecting a new test value causes some unnecessary backtracking in these types of strictly combinational circuits.

Number of tests : 54

Total CPU time (Barclay's) : 7079.7 seconds

Total CPU time (New) : 1172.5 seconds

Speed-up : 6.0

Additional tests found : 0

7.1.11 SHFT

SHFT is a four-bit bidirectional shift register with parallel load and clear operations. The register is modeled at the bit level. The complexity of the shift operations requires many low-level goals and the ordering of several time periods, hence the speedup is relatively good.

Number of tests : 58

Total CPU time (Barclay's) : 47448.3 seconds

Total CPU time (New) : 793.9 seconds

Speed-up :59.8

Additional tests found : 4

7.1.12 SHFTV

SHFTV is the same shift register as is modeled in SHFT; however the register is modeled as a vector rather than a collection of individual bits. The new method encountered a problem with the selection of test values for this circuit. Three of the STUCKDATA faults needed a test value with a 1 in one or more of the three left-most bits. This is because only the these three bits are used after the shifting operation is performed. This fact is not known to the routine which selects test values, so the new method tries all combinations of 0's and X's as test values before trying any 1's. These three tests were aborted, and test values containing 1's were inserted. The method behaved normally following this.

Number of tests	: 29
Total CPU time (Barclay's)	: 9917.2 seconds
Total CPU time (New)	: 527.4 seconds
Speed-up	: 18.8
Additional tests found	: 4

7.1.13 UARTO

UARTO models the transmit half of a simple uart. The complexity of the circuit operations lead to tests which involve a relatively high number of events, and several time periods. This allows the new method to demonstrate a reasonable speed improvement.

Number of tests	: 4
Total CPU time (Barclay's)	: 10970.4 seconds
Total CPU time (New)	: 191.4 seconds
Speed-up	: 57.3

Additional tests found : 8

7.2 Summary

The new test generation method demonstrates a speed improvement for all circuit models. The method develops the greatest speedup in those circuits which require a large number of low-level goals and a number of time periods in the final test. This is because the method eliminates most of the time spent handling the list of unsolved goals and finding a ordering of relatively scheduled events. For circuits that do not involve large numbers of subgoals or multiple time periods, the speedup is proportionally smaller. Therefore, it appears that the new algorithm demonstrates a larger speedup for more complex circuit models, which is an indication that algorithm could be efficiently applied in a test generation tool of practical size.

8.0 Suggestions

The new method demonstrates a definite speed improvement over Barclay's method in generating test vectors. As an algorithm for finding tests for individual faults, the method performs satisfactorily. However, when considered as a system for the generation of a complete set of tests for a chip, there are several areas which could be further developed. This chapter mentions some of these areas, and discusses possible directions for exploring them.

8.1.1 Expansion of the VHDL Subset and Fault Model

The speed performance of the improved algorithm suggests that it would be capable of handling larger circuit models. However, the limitations imposed by the restricted VHDL subset make it difficult to construct larger models. Most notable is the restriction to single process models. The modeling techniques typically used in building multi-process models use features of VHDL that are not included in the current fault model. The bus resolution functions that must be applied when multiple processes assign to the same signals typically employ some type of algorithmic description, using loop and variables. The current fault model does not consider looping constructs, and variables are considerably more difficult to handle than signals. (Variable assignments execute sequentially, which means that the value of a variable depends not only on time, but on the position of the statement relative to other variable assignments.)

If, however, multiple process models are created which accomplish multiplexing without the use of algorithmic bus resolution functions, the extension to the test generation method to handle them is relatively straightforward. In performing propagation and justification, all statements are considered as possible solutions, regardless of the process to which they belong. Upon executing a statement, however, an additional condition exists: the process containing the statement must be activated. This condition is satisfied by examining the sensitivity list for the process, and causing one of the signals in it to change. If the justification process involved in executing the statement does not change one of the signals, it may be necessary for the user to provide an intelligent selection of the signal to be changed. (For example, in a process which is started by raising a RUN signal, the algorithm should be directed to choose this signal.) If future work expands the fault model to include these constructs, it is likely that test conditions for the faults can be expressed in terms of justification, propagation, and execution, so that the current algorithm could be used to develop tests for the faults.

8.1.2 Analyzing Circuit Function

While the new algorithm tries to take advantage of a higher-level view in the goal solving process, no attempt is made to take a high-level view of the circuit model itself. An analysis of the function performed by each statement (or group of statements) could lead to better selection of test values, and also to better selection of statements used to accomplish justification and propagation. A higher level view of the model could also identify operations which need only occur once, such as the setting of direction flags in a counter. The new algorithm sets all flags to the necessary value

each time an operation (such as increment or shift) is performed, even though it is not necessary.

8.1.3 Retention of Past Work

In the course of generating tests for a list of faults for a circuit, justification, propagation, and execution operations are solved repeatedly. In the current method, there is no provision to save the result of an operation for use the next time a similar operation is performed. It is feasible that the results of an operation could be stored and retrieved as a single, complex event. When an operation is required which has been solved in the past, the complete result could be inserted in the event queue, and each sub-event could be assigned a time tag at that point. Over the course of generating tests for a set of circuit faults, the time savings from using such a technique could be substantial.

8.1.4 Avoiding Duplicate Tests

The set of tests for a list of faults in a circuit often contains a number of duplicate tests. In a gate level test generation method, a test which covers more than one fault is considered an advantage, as it reduces a large test set. However, in a chip-level test generation environment, the number of faults predicted by the fault model is typically much less than the number of stuck-faults in a corresponding gate-level model. Each chip-level test is responsible for a block of gate-level logic; therefore the removal of a single test vector from the test set may reduce the total fault coverage, and is considered a disadvantage.

For this reason, instead of removing a duplicate test from the test set, it is better to find a different test for each fault involved, if it is possible. This will increase the number of different chip-level test vectors in the test set, which should help to boost overall fault coverage.

Related to this idea is the selection of values for "don't care" bits in the test vector. The current method leaves don't care bits unassigned in the final test vector. If these bits were specified in such a way that a series of different values were assigned to the don't care bits, the test set would exercise the data paths in the circuit more completely. The selection of values for don't care bits generated during the test generation process is a topic of current research at Virginia Tech.

8.1.5 Proving Algorithm Validity

Before an algorithm can be widely accepted, some verification of the method is required. The successful application of the algorithm to a variety of circuit models would provide some degree of confidence, but a more formal validation is preferable. The test generation system presented in this thesis essentially attempts to provide rules describing the course of action to be taken under a set of circumstances typically encountered during test generation. It is difficult to demonstrate that such a system has defined enough rules to handle the general test generation problem. However, if the space of all possible circumstances under which justification and propagation might occur were defined, then it might be demonstrated that a certain set of rules covers the entire space. The difficulty lies in expressing time constraints, since a set of rules will need to be able to consider all possible orderings of the events that it generates in order to be certain that it can find a test if one exists.

9.0 Conclusions

The improved test generation algorithm demonstrates a speed-up of approximately one order of magnitude over the method developed by Barclay. A higher level of abstraction in the handling of the goal tree and the explicit handling of time during the goal solving process allow the new method to achieve this speed-up.

The performance of the method as an algorithm for generating test vectors is satisfactory; however, the algorithm must be incorporated into a more complete test generation system to achieve high fault coverage and truly efficient performance when generating a complete set of tests for a chip or circuit. Placed in a system which provides high level information extracted from the VHDL model and assists in the determination of appropriate test values, the algorithm should contribute to the performance of an efficient test generation tool.

BIBLIOGRAPHY

1. D. S. Barclay, "A Heuristic Chip-Level Test Generation Algorithm", *23rd Design Automation Conference*, pp. 257-262, June, 1986.

2. D. S. Barclay, "An Automatic Test Generation Method for Chip-Level Circuit Descriptions", *Master's Thesis*, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, January 1986.

3. Y. H. Levendel and P. R. Menon, "Test Generation Algorithms for Computer Hardware Description Languages", *IEEE Transactions on Computers*, vol C-29, no. 3, March 1980.

4. 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.

5. R. Khorram, "Functional Test Pattern Generation for Integrated Circuits" *Proceedings of the 1984 International Test Conference*, pp. 246-249.

6. K. Son and J. Y. O. Fong, "Automatic Behavioral Test Generation", *Proceedings of the 1982 International Test Conference*, pp. 161-165.

7. J. R. Armstrong, A. K. Gupta, and J. Stewart, "Functional Fault Modeling for VLSI Devices", Final Report for IBM Contract YD 190121, 1984.

8. A. K. Gupta, "Functional Fault Modeling and Test Vector Development for VLSI Systems", *Master's Thesis*, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, March, 1985.

9. F. E. Norrod, "A Graph-Oriented Test Generation Algorithm for Hardware Description Languages", *Unpublished manuscript.*

10. J. R. Armstrong and D. G. Burnette, "A Structured Approach to Chip-Level Modeling", *Unpublished manuscript.*

11. J. R. Armstrong, *Chip Level Modeling with VHDL*, to be published.

12. D. Han, "Optimal Constructs for Chip Level Modeling", *Master's Thesis*, Virginia Polytechnic Institute and State University, August, 1986.

13. C. Chao, "Micro-operation Perturbations in Chip Level Fault Modeling", *Unpublished manuscript.*

14. C. Cho, "A Chip Level Fault Coverage Experiment", Department of Electrical Engineering, Virginia Polytechnic Institute and State University, August, 1986.

15. M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, Rockville, Maryland, 1976.

16. P. K. Lala, *Fault Tolerant & Fault Testable Hardware Design*, Prentice-Hall, London, 1985.

17. A Miczo, *Digital Logic Testing and Simulation*, Harper & Row, Inc., New York, New York, 1986.

18. F. J. Hill and B. Huey, "SCIRTISS: A Search System for Sequential Circuit Test Sequences", *IEEE Transactions on Computers*, vol. C-26, no. 5. pp. 490-502.

19. M. J. Bending, "Hitest: A Knowledge-Based Test Generation System", *IEEE Design & Test*, vol. 2, no. 3, pp. 83-92.

20. P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits", *IEEE Transactions on Computers*, vol. C-30, no. 3. pp. 215-222.

21. J. A. Darringer, "The Application of Program Verification Techniques to Hardware Verification", *Poceedings of the 16th Design Automation Conference*, 1979, pp. 375-381.

22. T. Lin and S. Y. H. Su, "The S-Algorithm: A Promising Solution for Systematic Functional Test Generation", *IEEE Transactions on Computer-Aided Design*, vol. CAD-4, no. 3.

23. R. A. Marlett, "EBT: A Comprehensive Test Generation Technique for Highly Sequential Circuits", *Proceedings of the 15th Design Automation Conference*, 1978, pp. 335-339.

24. W. Clocksin and C. Mellish, *Programming in Prolog*.

25. *VHDL Language Reference Manual Version 7.2*, IR-MD-045-2, Intermetrics, Inc., August, 1985.

26. *VHDL User's Manual Volume II: User's Reference Guide*, IR-MD-065-1, Intermetrics, Inc., August, 1985.

Appendix A: User's Guide

9.1 Introduction

The new test generation algorithm is implemented in Prolog using the Portable Prolog interpreter developed at the University of York. The system is currently installed on a Data General MV-10000 computer.

The test generation program expects the file structure shown in Figure 7. The main directory contains the Prolog files for the test generation algorithm. Subordinate to this directory are two sub-directories: the HDL directory and the WAVE directory. The HDL directory contains the VHDL source code files, the Prolog rule files, and the fault list files for each circuit model. Each time the test generation program produces a test, the test vector will be stored in an appropriate file in the WAVE directory.

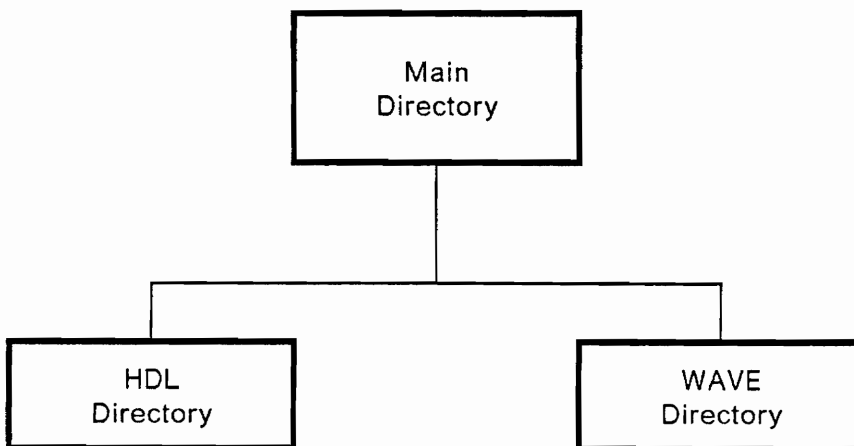


Figure 7. Directory structure

The Portable Prolog interpreter is written in Pascal and designed so that it can be easily ported between computer systems. The documentation pamphlet for the Portable Prolog interpreter describes the for installing the system. For a good tutorial on the Prolog language, and a description of the syntax used in Portable Prolog, see *Programming in Prolog* by Clocksin and Mellish.

9.2 VHDL-to-Prolog Translation

9.2.1 Writing the VHDL Model

The test generation method is designed to work with model descriptions written in VHDL. The subset of VHDL constructs which may be used to form circuit descriptions is described below.

- **Statements:** IF, CASE, and assignment statements may be used. Only simple assignment statements are allowed - 'AFTER phrases are not allowed.

- **Objects:** Objects may be of type bit or bit_vector. Only 0 and 1 bit types are included; Z values are not implemented. Boolean types and many user-defined types can be expressed in terms of bits or bit vectors.

- **Operations:** Figure 8 on page 60 lists the allowed operations, and the syntax used to express the operations in the Prolog environment. The 'STABLE attribute may only be applied to bits.

Bit Operations:

AND	BITAND	And two bits
OR	BITOR	Or two bits
NOT	BITNOT	Invert one bit
XOR	BITXOR	Exclusive or two bits
EQV	BITEQV	Exclusive nor two bits

Vector Operations:

AND	BVAND	Bit-by-bit and
OR	BVOR	Bit-by-bit or
NOT	BVNOT	Bit-by-bit inversion
XOR	BITXOR	Bit-by-bit exclusive or
EQV	BITEQV	Bit-by-bit exclusive nor
EQ	BVEQ	Vector equality
NEQ	BVNEQ	Vector inequality
ADD	BVADD	Unsigned addition
SUB	BVSUB	Unsigned subtraction
LE	BVLT	Unsigned less than
LT	BVLE	Unsigned less than or equal

Other Operations:

A & B	BVCAT	Vector concatenation
A[1 to 3]	(BVSUBV 1 3)	Subvector extraction
A[0]	(BVSUBV 0)	Bit-from-bit vector extraction
'STABLE	STABLE	Object 'STABLE attribute

Figure 8. Allowable operations: VHDL operation, Prolog representation, and description.

9.2.2 Initial Preprocessing

The VHDL source code is first translated to a Prolog representation using the C program VHDLTOP. Invoke this program by typing

```
VHDLTOP
```

You will be prompted for the name of the input file name (the VHDL source file). The VHDL source file typically will have the extension .VHD. In order for the test generation to find the file during the rest of the process, the VHDL file should be placed in the HDL directory as described above. The program will perform the translation operation and produce an output file with the same name as the input file plus an extension of .HDL.

9.2.3 Rule Extraction

After initial translating to Prolog, some information used by the test generation algorithm is extracted and stored in the form of Prolog rules. This rule extraction occurs in two steps. The first step generates basic rules concerning the number and type of statements in the model, the names and types of all objects, the various expressions used, etc. These facts are stored in a file with the extension .AUX, which is placed in the HDL directory. The second phase generates justification rules for each assignment statement and input pin in the model. These rules are stored in a file with the extension .PRO, which is also placed in the HDL directory.

To run either or both of the rule extraction steps, first move to the main code directory. Start the Prolog interpreter by typing:

```
prolog
```

Once in the interpreter, load the Prolog code for the preprocessing step by typing:

```
consult('preprocess.pro').
```

(Do not forget the period at the end of the command.) The short form for the consult command is ['preprocess.pro'].; this accomplishes the same thing.

Once the code has loaded, type

```
start.
```

to begin the program. You will be prompted for the model name; the model name is the name of your VHDL source file without the .VHD extension. The Prolog form of the VHDL model will now be loaded. When this has completed, you will have three options:

1. Type **go.** to run both rule extraction steps.

2. Type **step1.** to run the first stage.

3. Type **step2.** to run the second stage. The second stage of rule extraction assumes that the first stage has been run successfully.

After completing the preprocessing, type

```
end.
```

to exit the Prolog interpreter.

9.3 Generating a Fault List

After the preprocessing stages have been completed, a fault list for the model can be generated. To run this operation, start the Prolog interpreter as discussed above. Load the Prolog code for the fault list program by typing

```
consult('pickfaults.pro').
```

Type **start.**, answer the prompt for the model name, and then type **go.** to begin the fault list process. The fault list will be stored in a file with the extension .FAULTS, placed in the HDL directory.

All faults predicted by the fault model will be listed; however, some fault cases are not practical. Faults which cause a clocked statement to execute continuously (STUCKTHEN or STUCKELSE conditions, typically) correspond to a fault which causes clocking at an infinite frequency. Therefore, the fault list should be examined, and any clocking faults which are not realistic should not be included in the test generation process.

9.4 Generating Tests

To begin the test generation process, start the Prolog interpreter, and load the Prolog code by typing **consult('dotest.pro').**. Type **start.** to begin execution. Enter the model name in response to the prompt. When the Prolog rule files have been loaded, you will be asked whether the DONT_DISTURB rules should be enforced. In general, these rules should be enforced, as they force values on inputs such as clear and set operations to inactive states when they are not being used. However, models which contain statements such as those in Figure 9 on page 65 can fall into a loop condition trying to enforce the DONT_DISTURB requirement. Referring to the figure, of the object LOADFLAG is set to TRUE, a DONT_DISTURB rule will try to set the expression in statement s9 to prevent s11 from executing. Since the expression itself involves LOADFLAG, a new DONT_DISTURB is generated, and a loop condition results. In the models used to test the method, this situation occurs only in CCNT2 and UARTO.

```
   s9:  if (STRB='1' and not STRB'STABLE) and LOADFLAG
then
   s10:   LIM <= DATA;
   s1:    LOADFLAG <= FALSE;
```

Figure 9. VHDL fragment for DONT_DISTURB example

The test generation process may now begin. Answer the prompt for the fault number by selecting a fault from the fault list and entering the corresponding number. As the goal solving progresses, statements describing the current operation will be displayed on the screen. The information contained in these statements indicates the values, expressions, and times that are being used in the current operation.

When the solving process has completed, a waveform output is produced. Due to the nature of Prolog, the process of preparing the waveform can take 10 seconds for short tests up to a minute for longer tests. When the waveform has been developed, it is displayed on the screen and also stored in a file. The file is placed in the WAVE directory, and the filename is constructed by appending the fault number to the model name, and adding the extension .WAVE.

In normal operation, the test generation process runs to completion without stopping. For observation or debugging purposes, the spy point feature of the Prolog interpreter may be used. A spy point is a break point set on a Prolog rule. When a rule on which a spy point has been set is encountered, execution stops and the user has the opportunity to examine the state of the program. (See the Portable Prolog documentation or the book by Clocksin and Mellish for more information on setting and using spy points.) Typical rules for spy points are the justification, propagation, and execution rules.

9.5 Running in Batch Mode

Files have been created to run all of the above operations in batch mode. They are stored in the BATCH directory on the Data General system. The comments contained in the files explain their use.

Appendix B: Circuit Models and Fault Lists

This section contains the VHDL source listings and the fault lists for each model mentiooned in the results section. For each fault, there is an indication of the test generation result for that fault case. Times are given in seconds of CPU time. For faults which were excluded or for which no test could be generated, one of the following comments applies:

- *Excl* - Excluded clocking fault.
- *Fail* - Test generation failed. Typically this occurs because the presence of the fault prevents execution of statements required to prload, justify, or propagate values.
- *Over* - Test generation program caused the Prolog interpreter to overflow; or the process exceeded maximum limits on CPU time.
- *NAF* - Not a fault.
- *Wrong* - Test generated is invalid (due to BVADD/BVSUB not fully implemented, or other reasons as explained in the Results section.)
- - - Not attempted. STUCKDATA tests which were not generated under Barclay's method were not run under the new method.

9.6 Adder VHDL 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;
```

9.7 ADDER Fault List

Fault	Barclay's Method	New Method
[1,assncntl,s1]	118.8	3.2
[2,microop,[s1,"-",""],bvadd,bvsub]	70.8	1.5
[3,microop,[s1,"-",""],bvadd,bvxor]	39.2	1.2
[4,stuckdata,[s1,"-",""],[bv,"0000"]]	19.8	1.5
[5,stuckdata,[s1,"-",""],[bv,"1111"]]	28.4	1.3
[6,stuckdata,[s1,"-","L"],[bv,"0000"]]	34.6	2.0
[7,stuckdata,[s1,"-","L"],[bv,"1111"]]	34.7	1.8
[8,stuckdata,[s1,"-","R"],[bv,"0000"]]	34.8	2.0
[9,stuckdata,[s1,"-","R"],[bv,"1111"]]	34.9	1.9

9.8 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;
```

9.9 ADDR2 Fault List

Fault	Barclay's Method	New Method
[1,assncntl,s1]	95.7	4.2
[2,microop,[s1,"-",""],bitxor,biteqv]	31.9	1.5
[3,microop,[s1,"-","R"],bitxor,biteqv]	27.0	2.0
[4,assncntl,s2]	185.8	12.3
[5,microop,[s2,"-",""],bitor,bitand]	61.3	4.8
[6,microop,[s2,"-","L"],bitand,bitor]	43.4	5.9
[7,microop,[s2,"-","R"],bitand,bitor]	45.6	6.0
[8,microop,[s2,"-","RR"],bitxor,biteqv]	56.6	10.7
[9,assncntl,s3]	200.8	13.0
[10,microop,[s3,"-",""],bitxor,biteqv]	58.2	5.2
[11,microop,[s3,"-","R"],bitxor,biteqv]	48.7	5.7
[12,assncntl,s4]	257.7	21.8
[13,microop,[s4,"-",""],bitor,bitand]	88.8	8.6
[14,microop,[s4,"-","L"],bitand,bitor]	73.6	9.8
[15,microop,[s4,"-","R"],bitand,bitor]	75.5	9.9
[16,microop,[s4,"-","RR"],bitxor,biteqv]	82.8	31.4
[17,assncntl,s5]	309.1	23.1
[18,microop,[s5,"-",""],bitxor,biteqv]	89.0	9.0
[19,microop,[s5,"-","R"],bitxor,biteqv]	50.8	9.5
[20,assncntl,s6]	401.4	36.4
[21,microop,[s6,"-",""],bitor,bitand]	86.0	14.5
[22,microop,[s6,"-","L"],bitand,bitor]	101.1	15.7
[23,microop,[s6,"-","R"],bitand,bitor]	105.4	16.0
[24,microop,[s6,"-","RR"],bitxor,biteqv]	81.1	91.0
[25,assncntl,s7]	442.7	35.1
[26,microop,[s7,"-",""],bitxor,biteqv]	125.0	13.1
[27,microop,[s7,"-","R"],bitxor,biteqv]	53.1	13.6
[28,assncntl,s8]	374.3	41.1
[29,microop,[s8,"-",""],bitor,bitand]	59.4	13.9
[30,microop,[s8,"-","L"],bitand,bitor]	108.9	15.1
[31,microop,[s8,"-","R"],bitand,bitor]	111.5	15.5
[32,microop,[s8,"-","RR"],bitxor,biteqv]	55.2	296.9
[33,stuckdata,[s1,"-",""],[bit,"0"]]	22.3	2.0
[34,stuckdata,[s1,"-",""],[bit,"1"]]	22.3	2.0
[35,stuckdata,[s1,"-","L"],[bit,"0"]]	30.9	2.4
[36,stuckdata,[s1,"-","L"],[bit,"1"]]	32.1	2.4
[37,stuckdata,[s1,"-","R"],[bit,"0"]]	27.0	2.3
[38,stuckdata,[s1,"-","R"],[bit,"1"]]	25.5	2.3
[39,stuckdata,[s1,"-","RL"],[bit,"0"]]	27.9	2.8
[40,stuckdata,[s1,"-","RL"],[bit,"1"]]	28.4	2.8
[41,stuckdata,[s1,"-","RR"],[bit,"0"]]	29.4	2.8
[42,stuckdata,[s1,"-","RR"],[bit,"1"]]	30.3	2.8
[43,stuckdata,[s2,"-",""],[bit,"0"]]	47.2	6.0
[44,stuckdata,[s2,"-",""],[bit,"1"]]	50.4	6.0
[45,stuckdata,[s2,"-","L"],[bit,"0"]]	52.8	6.2
[46,stuckdata,[s2,"-","L"],[bit,"1"]]	50.3	6.7

[47,stuckdata,[s2,"-","LL"],[bit,"0"]]	45.0	6.7
[48,stuckdata,[s2,"-","LL"],[bit,"1"]]	51.9	6.7
[49,stuckdata,[s2,"-","LR"],[bit,"0"]]	48.7	6.7
[50,stuckdata,[s2,"-","LR"],[bit,"1"]]	52.2	6.7
[51,stuckdata,[s2,"-","R"],[bit,"0"]]	53.5	6.6
[52,stuckdata,[s2,"-","R"],[bit,"1"]]	47.0	6.6
[53,stuckdata,[s2,"-","RL"],[bit,"0"]]	54.6	7.1
[54,stuckdata,[s2,"-","RL"],[bit,"1"]]	58.0	7.1
[55,stuckdata,[s2,"-","RR"],[bit,"0"]]	53.8	7.0
[56,stuckdata,[s2,"-","RR"],[bit,"1"]]	52.8	7.0
[57,stuckdata,[s2,"-","RRL"],[bit,"0"]]	62.2	12.3
[58,stuckdata,[s2,"-","RRL"],[bit,"1"]]	54.9	7.6
[59,stuckdata,[s2,"-","RRR"],[bit,"0"]]	65.1	12.0
[60,stuckdata,[s2,"-","RRR"],[bit,"1"]]	56.4	7.6
[61,stuckdata,[s3,"-",""],[bit,"0"]]	46.3	5.7
[62,stuckdata,[s3,"-",""],[bit,"1"]]	49.6	5.7
[63,stuckdata,[s3,"-","L"],[bit,"0"]]	61.5	6.1
[64,stuckdata,[s3,"-","L"],[bit,"1"]]	66.1	6.1
[65,stuckdata,[s3,"-","R"],[bit,"0"]]	54.0	6.0
[66,stuckdata,[s3,"-","R"],[bit,"1"]]	49.7	6.0
[67,stuckdata,[s3,"-","RL"],[bit,"0"]]	55.6	6.5
[68,stuckdata,[s3,"-","RL"],[bit,"1"]]	59.0	6.5
[69,stuckdata,[s3,"-","RR"],[bit,"0"]]	54.0	6.5
[70,stuckdata,[s3,"-","RR"],[bit,"1"]]	57.0	6.5
[71,stuckdata,[s4,"-",""],[bit,"0"]]	51.7	10.1
[72,stuckdata,[s4,"-",""],[bit,"1"]]	96.0	10.0
[73,stuckdata,[s4,"-","L"],[bit,"0"]]	91.8	10.1
[74,stuckdata,[s4,"-","L"],[bit,"1"]]	85.6	10.5
[75,stuckdata,[s4,"-","LL"],[bit,"0"]]	49.1	10.6
[76,stuckdata,[s4,"-","LL"],[bit,"1"]]	83.1	10.6
[77,stuckdata,[s4,"-","LR"],[bit,"0"]]	52.1	10.6
[78,stuckdata,[s4,"-","LR"],[bit,"1"]]	100.6	10.6
[79,stuckdata,[s4,"-","R"],[bit,"0"]]	91.9	10.6
[80,stuckdata,[s4,"-","R"],[bit,"1"]]	81.9	10.8
[81,stuckdata,[s4,"-","RL"],[bit,"0"]]	84.1	11.1
[82,stuckdata,[s4,"-","RL"],[bit,"1"]]	97.5	11.1
[83,stuckdata,[s4,"-","RR"],[bit,"0"]]	90.7	10.9
[84,stuckdata,[s4,"-","RR"],[bit,"1"]]	84.9	10.9
[85,stuckdata,[s4,"-","RRL"],[bit,"0"]]	93.4	33.4
[86,stuckdata,[s4,"-","RRL"],[bit,"1"]]	91.2	11.6
[87,stuckdata,[s4,"-","RRR"],[bit,"0"]]	93.9	33.1
[88,stuckdata,[s4,"-","RRR"],[bit,"1"]]	88.8	11.6
[89,stuckdata,[s5,"-",""],[bit,"0"]]	51.5	9.6
[90,stuckdata,[s5,"-",""],[bit,"1"]]	81.7	9.6
[91,stuckdata,[s5,"-","L"],[bit,"0"]]	102.1	10.1
[92,stuckdata,[s5,"-","L"],[bit,"1"]]	98.6	10.0
[93,stuckdata,[s5,"-","R"],[bit,"0"]]	89.4	9.9
[94,stuckdata,[s5,"-","R"],[bit,"1"]]	54.4	9.9
[95,stuckdata,[s5,"-","RL"],[bit,"0"]]	56.5	10.4
[96,stuckdata,[s5,"-","RL"],[bit,"1"]]	56.2	10.4
[97,stuckdata,[s5,"-","RR"],[bit,"0"]]	56.2	10.4

[98,stuckdata,[s5,"-","RR"],[bit,"1"]]	97.6	10.4
[99,stuckdata,[s6,"-",""],[bit,"0"]]	51.8	16.3
[100,stuckdata,[s6,"-",""],[bit,"1"]]	122.8	16.2
[101,stuckdata,[s6,"-","L"],[bit,"0"]]	126.6	16.2
[102,stuckdata,[s6,"-","L"],[bit,"1"]]	113.8	17.3
[103,stuckdata,[s6,"-","LL"],[bit,"0"]]	48.4	16.6
[104,stuckdata,[s6,"-","LL"],[bit,"1"]]	117.6	16.7
[105,stuckdata,[s6,"-","LR"],[bit,"0"]]	43.1	16.6
[106,stuckdata,[s6,"-","LR"],[bit,"1"]]	103.2	16.7
[107,stuckdata,[s6,"-","R"],[bit,"0"]]	72.7	16.9
[108,stuckdata,[s6,"-","R"],[bit,"1"]]	98.9	17.4
[109,stuckdata,[s6,"-","RL"],[bit,"0"]]	72.7	17.3
[110,stuckdata,[s6,"-","RL"],[bit,"1"]]	106.4	17.3
[111,stuckdata,[s6,"-","RR"],[bit,"0"]]	73.4	17.1
[112,stuckdata,[s6,"-","RR"],[bit,"1"]]	73.0	17.1
[113,stuckdata,[s6,"-","RRL"],[bit,"0"]]	80.3	94.5
[114,stuckdata,[s6,"-","RRL"],[bit,"1"]]	75.1	17.7
[115,stuckdata,[s6,"-","RRR"],[bit,"0"]]	80.7	94.0
[116,stuckdata,[s6,"-","RRR"],[bit,"1"]]	25.1	17.7
[117,stuckdata,[s7,"-",""],[bit,"0"]]	48.7	13.8
[118,stuckdata,[s7,"-",""],[bit,"1"]]	109.0	13.8
[119,stuckdata,[s7,"-","L"],[bit,"0"]]	117.3	14.4
[120,stuckdata,[s7,"-","L"],[bit,"1"]]	117.8	14.1
[121,stuckdata,[s7,"-","R"],[bit,"0"]]	112.4	14.1
[122,stuckdata,[s7,"-","R"],[bit,"1"]]	52.8	14.0
[123,stuckdata,[s7,"-","RL"],[bit,"0"]]	54.0	14.6
[124,stuckdata,[s7,"-","RL"],[bit,"1"]]	53.9	14.5
[125,stuckdata,[s7,"-","RR"],[bit,"0"]]	54.0	14.6
[126,stuckdata,[s7,"-","RR"],[bit,"1"]]	113.7	14.5
[127,stuckdata,[s8,"-",""],[bit,"0"]]	29.1	16.1
[128,stuckdata,[s8,"-",""],[bit,"1"]]	107.9	15.8
[129,stuckdata,[s8,"-","L"],[bit,"0"]]	111.9	15.5
[130 stuckdata,[s8,"-","L"],[bit,"1"]]	105.8	17.1
[131,stuckdata,[s8,"-","LL"],[bit,"0"]]	27.3	16.0
[132,stuckdata,[s8,"-","LL"],[bit,"1"]]	112.0	16.0
[133,stuckdata,[s8,"-","LR"],[bit,"0"]]	27.8	16.0
[134,stuckdata,[s8,"-","LR"],[bit,"1"]]	113.0	16.0
[135,stuckdata,[s8,"-","R"],[bit,"0"]]	52.7	16.6
[136,stuckdata,[s8,"-","R"],[bit,"1"]]	104.8	17.3
[137,stuckdata,[s8,"-","RL"],[bit,"0"]]	51.7	16.9
[138,stuckdata,[s8,"-","RL"],[bit,"1"]]	112.0	16.7
[139,stuckdata,[s8,"-","RR"],[bit,"0"]]	51.8	16.6
[140,stuckdata,[s8,"-","RR"],[bit,"1"]]	51.3	16.6
[141,stuckdata,[s8,"-","RRL"],[bit,"0"]]	58.3	301.8
[142,stuckdata,[s8,"-","RRL"],[bit,"1"]]	53.3	17.2
[143,stuckdata,[s8,"-","RRR"],[bit,"0"]]	59.6	302.4
[144,stuckdata,[s8,"-","RRR"],[bit,"1"]]	54.0	17.2

9.10 CCNT2 VHDL 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) ;
          LOADFLAGBOOLEAN

        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;
```

9.11 CCNT2 Fault List

Fault	Barclay's Method	New Method
[1,stuckthen,s1]	Excl	Excl
[2,stuckelse,s1]	Fail	27.7
[3,microop,[s1,"-",""],bitand,bitor]	Excl	Excl
[4,microop,[s1,"-","L"],biteqv,bitxor]	Excl	Excl
[5,microop,[s1,"-","R"],bitnot,bitbuf]	Excl	Excl
[6,deadclause,s2,[[bv,"11"]]]	Over	257.3
[7,deadclause,s2,[[bv,"10"]]]	Over	235.2
[8,deadclause,s2,[[bv,"01"]]]	Fail	Fail
[9,deadclause,s2,[[bv,"00"]]]	Over	29.2
[10,assncntl,s3]	Over	22.6
[11,assncntl,s4]	Over	Fail
[12,assncntl,s5]	Over	407.0
[13,assncntl,s6]	Over	155.6
[14,assncntl,s7]	Over	241.3
[15,assncntl,s8]	Over	152.7
[16,stuckthen,s9]	Excl	Excl
[17,stuckelse,s9]	Over	Fail
[18,microop,[s9,"-",""],bitand,bitor]	Excl	Excl
[19,microop,[s9,"-","L"],bitand,bitor]	Excl	Excl
[20,microop,[s9,"-","LL"],biteqv,bitxor]	Excl	Excl
[21,microop,[s9,"-","LR",bitnot,bitbuf]	Excl	Excl
[22,assncntl,s10]	Over	Fail
[23,assncntl,s11]	Fail	Fail
[24,stuckthen,s12]	Excl	Excl
[25,stuckelse,s12]	6004.3	108.0
[26,microop,[s12,"-",""],bitand,bitor]	Excl	Excl
[27,microop,[s12,"-","L"],bitand,bitor]	Excl	Excl
[28,microop,[s12,"-","LL"],biteqv,bitxor]	Excl	Excl
[29,microop,[s12,"-","LR",bitnot,bitbuf]	Excl	Excl
[30,stuckthen,s13]	Over	51.9
[31,stuckelse,s13]	Over	129.3
[32,microop,[s13,"-",""],biteqv,bitxor]	Over	152.3
[33,assncntl,s14]	Over	106.5
[34,microop,[s14,"-",""],bvadd,bvsub]	Over	33.4
[35,microop,[s14,"-",""],bvadd,bvxor]	Over	33.1
[36,assncntl,s15]	Over	59.9
[37,microop,[s15,"-",""],bvsub,bvadd]	Over	34.1
[38,stuckthen,s16]	Over	239.6
[39,stuckelse,s16]	Over	Fail
[40,microop,[s16,"-",""],bveq,bvneq]	Over	Fail
[41,assncntl,s17]	Over	230.9

9.12 CKTA VHDL 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;
```

9.13 CKTA Fault List

Fault	Barclay's Method	New Method
[1,stuckthen,s1]	Excl	Excl
[2,stuckelse,s1]	Fail	17.2
[3,microop,[s1,"-",""],bitand,bitor]	Excl	Excl
[4,microop,[s1,"-","L"],biteqv,bitxor]	Excl	Excl
[5,microop,[s1,"-","R"],bitnot,bitbuf]	Excl	Excl
[6,assncntl,s2]	Over	21.4
[7,stuckthen,s3]	Excl	Excl
[8,stuckelse,s3]	1113.0	20.7
[9,microop,[s3,"-",""],bitand,bitor]	Excl	Excl
[10,microop,[s3,"-","L"],biteqv,bitxor]	Excl	Excl
[11,microop,[s3,"-","R"],bitnot,bitbuf]	Excl	Excl
[12,assncntl,s4]	1611.3	20.0
[13,assncntl,s5]	Over	16.4
[14,microop,[s5,"-",""],bitand,bitor]	547.4	9.3
[15,stuckdata,[s1,"-",""],[bit,"0"]]	Fail	-
[16,stuckdata,[s1,"-",""],[bit,"1"]]	Excl	Excl
[17,stuckdata,[s1,"-","L"],[bit,"0"]]	Fail	-
[18,stuckdata,[s1,"-","L"],[bit,"1"]]	Excl	Excl
[19,stuckdata,[s1,"-","LL"],[bit,"0"]]	Fail	-
[20,stuckdata,[s1,"-","LL"],[bit,"1"]]	Excl	Excl
[21,stuckdata,[s1,"-","LR"],[bit,"0"]]	Excl	Excl
[22,stuckdata,[s1,"-","LR"],[bit,"1"]]	NAF	NAF
[23,stuckdata,[s1,"-","R"],[bit,"0"]]	Fail	-
[24,stuckdata,[s1,"-","R"],[bit,"1"]]	Excl	Excl
[25,stuckdata,[s1,"-","RL"],[bit,"0"]]	Excl	Excl
[26,stuckdata,[s1,"-","RL"],[bit,"1"]]	Fail	-
[27,stuckdata,[s2,"-",""],[bit,"0"]]	Fail	-
[28,stuckdata,[s2,"-",""],[bit,"1"]]	Fail	-
[29,stuckdata,[s3,"-",""],[bit,"0"]]	Fail	-
[30,stuckdata,[s3,"-",""],[bit,"1"]]	Excl	Excl
[31,stuckdata,[s3,"-","L"],[bit,"0"]]	Fail	-
[32,stuckdata,[s3,"-","L"],[bit,"1"]]	Excl	Excl
[33,stuckdata,[s3,"-","LL"],[bit,"0"]]	Fail	-
[34,stuckdata,[s3,"-","LL"],[bit,"1"]]	Excl	Excl
[35,stuckdata,[s3,"-","LR"],[bit,"0"]]	Excl	Excl
[36,stuckdata,[s3,"-","LR"],[bit,"1"]]	NAF	NAF
[37,stuckdata,[s3,"-","R"],[bit,"0"]]	Fail	-
[38,stuckdata,[s3,"-","R"],[bit,"1"]]	Excl	Excl
[39,stuckdata,[s3,"-","RL"],[bit,"0"]]	Excl	Excl
[40,stuckdata,[s3,"-","RL"],[bit,"1"]]	Fail	-
[41,stuckdata,[s4,"-",""],[bit,"0"]]	208.5	8.9
[42,stuckdata,[s4,"-",""],[bit,"1"]]	208.141	8.9
[43,stuckdata,[s5,"-",""],[bit,"0"]]	244.4	9.7
[44,stuckdata,[s5,"-",""],[bit,"1"]]	48.8	3.8
[45,stuckdata,[s5,"-","L"],[bit,"0"]]	246.1	9.6
[46,stuckdata,[s5,"-","L"],[bit,"1"]]	246.9	9.6

[47,stuckdata,[s5,"-","R"],[bit,"0"]]	220.6	9.6
[48,stuckdata,[s5,"-","R"],[bit,"1"]]	530.7	9.6

9.14 CKTCV VHDL 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;
```

9.15 CKTCV Fault List

Fault	Barclay's Method	New Method
[1,stuckthen,s1]	Excl	Excl
[2,stuckelse,s1]	Fail	21.7
[3,microop,[s1,"-",""],bitand,bitor]	Excl	Excl
[4,microop,[s1,"-","L"],biteqv,bitxor]	Excl	Excl
[5,microop,[s1,"-","R"],bitnot,bitbuf]	Excl	Excl
[6,deadclause,s2,[[bv,"11"]]]	3908.3	31.5
[7,deadclause,s2,[[bv,"10"]]]	2059.2	17.3
[8,deadclause,s2,[[bv,"01"]]]	2117.0	17.2
[9,deadclause,s2,[[bv,"00"]]]	14359.5	25.2
[10,assncntl,s3]	11176.2	41.7
[11,assncntl,s4]	2258.5	28.5
[12,assncntl,s5]	5051.1	28.6
[13,stuckthen,s6]	912.9	15.9
[14,stuckelse,s6]	964.7	15.9
[15,microop,[s6,"-",""],bveq,bvneq]	967.2	21.0
[16,assncntl,s7]	5073.4	25.2
[17,assncntl,s8]	13367.7	24.8
[18,stuckdata,[s1,"-",""],[bit,"0"]]	Fail	-
[19,stuckdata,[s1,"-",""],[bit,"1"]]	Excl	Excl
[20,stuckdata,[s1,"-","L"],[bit,"0"]]	Fail	-
[21,stuckdata,[s1,"-","L"],[bit,"1"]]	Excl	Excl
[22,stuckdata,[s1,"-","LL"],[bit,"0"]]	Fail	-
[23,stuckdata,[s1,"-","LL"],[bit,"1"]]	Excl	Excl
[24,stuckdata,[s1,"-","LR"],[bit,"0"]]	Excl	Excl
[25,stuckdata,[s1,"-","LR"],[bit,"1"]]	NAF	NAF
[26,stuckdata,[s1,"-","R"],[bit,"0"]]	Fail	-
[27,stuckdata,[s1,"-","R"],[bit,"1"]]	Excl	Excl
[28,stuckdata,[s1,"-","RL"],[bit,"0"]]	Excl	Excl
[29,stuckdata,[s1,"-","RL"],[bit,"1"]]	Fail	-
[30,stuckdata,[s2,"-",""],[bv,"00"]]	Fail	-
[31,stuckdata,[s2,"-",""],[bv,"11"]]	Fail	-
[32,stuckdata,[s3,"-",""],[bv,"00"]]	1133.4	20.5
[33,stuckdata,[s3,"-",""],[bv,"11"]]	1148.9	20.5
[34,stuckdata,[s4,"-",""],[bv,"00"]]	556.9	13.0
[35,stuckdata,[s4,"-",""],[bv,"11"]]	564.4	12.9
[36,stuckdata,[s5,"-",""],[bv,"00"]]	563.9	13.1
[37,stuckdata,[s5,"-",""],[bv,"11"]]	564.7	13.0
[38,stuckdata,[s6,"-",""],[bit,"0"]]	1002.9	16.7
[39,stuckdata,[s6,"-",""],[bit,"1"]]	1042.5	16.7
[40,stuckdata,[s6,"-","L"],[bv,"00"]]	1125.6	17.2
[41,stuckdata,[s6,"-","L"],[bv,"11"]]	1087.7	17.2
[42,stuckdata,[s6,"-","R"],[bv,"00"]]	NAF	NAF
[43,stuckdata,[s6,"-","R"],[bv,"11"]]	811.3	17.0
[44,stuckdata,[s7,"-",""],[bv,"00"]]	424.6	11.4
[45,stuckdata,[s7,"-",""],[bv,"11"]]	426.8	11.3
[46,stuckdata,[s8,"-",""],[bv,"00"]]	436.1	11.5

[47,stuckdata,[s8,"-",""],[bv,"11"]]	421.4	11.4

9.16 CNTR VHDL 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;
```

9.17 CNTR Fault List

Fault	Barclay's Method	New Method
[1,stuckthen,s1]	Fail	8.7
[2,stuckelse,s1]	Fail	7.1
[3,microop,[s1,"-",""],biteqv,bitxor]	Fail	8.6
[4,assncntl,s2]	Fail	10.9
[5,assncntl,s3]	1434.7	22.5
[6,assncntl,s4]	Fail	41.2
[7,stuckthen,s5]	Excl	Excl
[8,stuckelse,s5]	105.5	7.6
[9,microop,[s5,"-",""],bitand,bitor]	Excl	Excl
[10,microop,[s5,"-","L"],biteqv,bitxor]	Excl	Excl
[11,microop,[s5,"-","R"],bitnot,bitbuf]	Excl	Excl
[12,assncntl,s6]	2115.3	7.5
[13,microop,[s6,"-",""],bitnot,bitbuf]	121.3	11.0
[14,assncntl,s7]	Fail	17.8
[15,microop,[s7,"-",""],bitxor,biteqv]	Over	31.6
[16,assncntl,s8]	Fail	38.8
[17,microop,[s8,"-",""],bitxor,biteqv]	Fail	78.0
[18,microop,[s8,"-","R"],bitand,bitor]	Fail	17.9
[19,stuckdata,[s1,"-",""],[bit,"0"]]	Fail	-
[20,stuckdata,[s1,"-",""],[bit,"1"]]	Fail	-
[21,stuckdata,[s1,"-","L"],[bit,"0"]]	Fail	-
[22,stuckdata,[s1,"-","L"],[bit,"1"]]	Fail	-
[23,stuckdata,[s1,"-","R"],[bit,"0"]]	NAF	NAF
[24,stuckdata,[s1,"-","R"],[bit,"1"]]	Fail	-
[25,stuckdata,[s2,"-",""],[bit,"0"]]	NAF	NAF
[26,stuckdata,[s2,"-",""],[bit,"1"]]	13.0	1.5
[27,stuckdata,[s3,"-",""],[bit,"0"]]	NAF	NAF
[28,stuckdata,[s3,"-",""],[bit,"1"]]	12.2	1.5
[29,stuckdata,[s4,"-",""],[bit,"0"]]	NAF	NAF
[30,stuckdata,[s4,"-",""],[bit,"1"]]	12.3	1.5
[31,stuckdata,[s5,"-",""],[bit,"0"]]	109.8	8.2
[32,stuckdata,[s5,"-",""],[bit,"1"]]	Excl	Excl
[33,stuckdata,[s5,"-","L"],[bit,"0"]]	117.6	9.4
[34,stuckdata,[s5,"-","L"],[bit,"1"]]	Excl	Excl
[35,stuckdata,[s5,"-","LL"],[bit,"0"]]	121.1	10.3
[36,stuckdata,[s5,"-","LL"],[bit,"1"]]	Excl	Excl
[37,stuckdata,[s5,"-","LR"],[bit,"0"]]	Excl	Excl
[38,stuckdata,[s5,"-","LR"],[bit,"1"]]	NAF	NAF
[39,stuckdata,[s5,"-","R"],[bit,"0"]]	122.6	9.5
[40,stuckdata,[s5,"-","R"],[bit,"1"]]	Excl	Excl
[41,stuckdata,[s5,"-","RL"],[bit,"0"]]	Excl	Excl
[42,stuckdata,[s5,"-","RL"],[bit,"1"]]	120.4	9.9
[43,stuckdata,[s6,"-",""],[bit,"0"]]	97.9	5.6
[44,stuckdata,[s6,"-",""],[bit,"1"]]	Fail	-
[45,stuckdata,[s6,"-","L"],[bit,"0"]]	Fail	-
[46,stuckdata,[s6,"-","L"],[bit,"1"]]	99.2	5.8

[47,stuckdata,[s7,"-",""],[bit,"0"]]	Fail	-
[48,stuckdata,[s7,"-",""],[bit,"1"]]	380.9	7.6
[49,stuckdata,[s7,"-","L"],[bit,"0"]]	Fail	-
[50,stuckdata,[s7,"-","L"],[bit,"1"]]	Fail	-
[51,stuckdata,[s7,"-","R"],[bit,"0"]]	Fail	-
[52,stuckdata,[s7,"-","R"],[bit,"1"]]	731.5	8.0
[53,stuckdata,[s8,"-",""],[bit,"0"]]	Fail	-
[54,stuckdata,[s8,"-",""],[bit,"1"]]	600.5	8.7
[55,stuckdata,[s8,"-","L"],[bit,"0"]]	Fail	-
[56,stuckdata,[s8,"-","L"],[bit,"1"]]	Fail	9.4
[57,stuckdata,[s8,"-","R"],[bit,"0"]]	Fail	-
[58,stuckdata,[s8,"-","R"],[bit,"1"]]	1539.4	9.2
[59,stuckdata,[s8,"-","RL"],[bit,"0"]]	Fail	-
[60,stuckdata,[s8,"-","RL"],[bit,"1"]]	Fail	-
[61,stuckdata,[s8,"-","RR"],[bit,"0"]]	Fail	-
[62,stuckdata,[s8,"-","RR"],[bit,"1"]]	Fail	-

9.18 CNTRV VHDL 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;
```

9.19 CNTRV Fault List

Fault	Barclay's Method	New Method
[1,stuckthen,s1]	Fail	92.6
[2,stuckelse,s1]	Wrong	8.7
[3,microop,[s1,"-",""],biteqv,bitxor]	Fail	10.2
[4,assncntl,s2]	2855.9	13.3
[5,stuckthen,s3]	Excl	Excl
[6,stuckelse,s3]	Wrong	78.7
[7,microop,[s3,"-",""],bitand,bitor]	Excl	Excl
[8,microop,[s3,"-","L"],biteqv,bitxor]	Excl	Excl
[9,microop,[s3,"-","R"],bitnot,bitbuf]	Excl	Excl
[10,assncntl,s4]	Over	79.3
[11,microop,[s4,"-",""],bvadd,bvsub]	155.3	12.8
[12,microop,[s4,"-",""],bvadd,bvxor]	Over	18.6
[13,stuckdata,[s1,"-",""],[bit,"0"]]	Over	-
[14,stuckdata,[s1,"-",""],[bit,"1"]]	Over	-
[15,stuckdata,[s1,"-","L"],[bit,"0"]]	Over	-
[16,stuckdata,[s1,"-","L"],[bit,"1"]]	Over	-
[17,stuckdata,[s1,"-","R"],[bit,"0"]]	NAF	NAF
[18,stuckdata,[s1,"-","R"],[bit,"1"]]	Fail	-
[19,stuckdata,[s2,"-",""],[bv,"000"]]	NAF	NAF
[20,stuckdata,[s2,"-",""],[bv,"111"]]	14.2	1.8
[21,stuckdata,[s3,"-",""],[bit,"0"]]	Wrong	-
[22,stuckdata,[s3,"-",""],[bit,"1"]]	Excl	Excl
[23,stuckdata,[s3,"-","L"],[bit,"0"]]	Wrong	-
[24,stuckdata,[s3,"-","L"],[bit,"1"]]	Excl	Excl
[25,stuckdata,[s3,"-","LL"],[bit,"0"]]	Wrong	-
[26,stuckdata,[s3,"-","LL"],[bit,"1"]]	Excl	Excl
[27,stuckdata,[s3,"-","LR"],[bit,"0"]]	Excl	Excl
[28,stuckdata,[s3,"-","LR"],[bit,"1"]]	NAF	NAF
[29,stuckdata,[s3,"-","R"],[bit,"0"]]	Wrong	-
[30,stuckdata,[s3,"-","R"],[bit,"1"]]	Excl	Excl
[31,stuckdata,[s3,"-","RL"],[bit,"0"]]	Excl	Excl
[32,stuckdata,[s3,"-","RL"],[bit,"1"]]	Wrong	-
[33,stuckdata,[s4,"-",""],[bv,"000"]]	Fail	-
[34,stuckdata,[s4,"-",""],[bv,"111"]]	Fail	-
[35,stuckdata,[s4,"-","L"],[bv,"000"]]	Fail	-
[36,stuckdata,[s4,"-","L"],[bv,"111"]]	115.3	7.427
[37,stuckdata,[s4,"-","R"],[bv,"000"]]	Wrong	-
[38,stuckdata,[s4,"-","R"],[bv,"111"]]	Wrong	-

9.20 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
s2:     if CLRBAR = 0 then
s3:       Q <= 0;
s4:       QBAR <= 1;
        else
s5:       if SETBAR = 0 then
s6:         Q <= 1;
s7:         QBAR <= 0;
          else
s8:         if (not CLOCK'STABLE) and (CLOCK = 1) then
s9:           Q <= DATA;
s10:          QBAR <= not DATA;
            end if
          end if
        end if
        end process;
        end arch;
```

9.21 DFF Fault List

Fault	Barclay's Method	New Method
[1,stuckthen,s2]	26.9	3.8
[2,stuckelse,s2]	18.1	3.1
[3,microop,[s2,"-",""],biteqv,bitxor]	22.6	4.2
[4,assncntl,s3]	195.3	5.7
[5,assncntl,s4]	151.9	5.7
[6,stuckthen,s5]	74.7	5.5
[7,stuckelse,s5]	158.5	5.0
[8,microop,[s5,"-",""],biteqv,bitxor]	82.6	6.1
[9,assncntl,s6]	166.0	5.5
[10,assncntl,s7]	168.1	5.4
[11,stuckthen,s8]	Excl	Excl
[12,stuckelse,s8]	184.5	7.6
[13,microop,[s8,"-",""],bitand,bitor]	Excl	Excl
[14,microop,[s8,"-","L"],bitnot,bitbuf]	Excl	Excl
[15,microop,[s8,"-","R"],biteqv,bitxor]	Excl	Excl
[16,assncntl,s9]	954.8	7.6
[17,assncntl,s10]	761.5	9.2
[18,microop,[s10,"-",""],bitnot,bitbuf]	88.3	4.8
[19,stuckdata,[s2,"-",""],[bit,"0"]]	20.9	3.5
[20,stuckdata,[s2,"-",""],[bit,"1"]]	20.3	3.5
[21,stuckdata,[s2,"-","L"],[bit,"0"]]	20.2	4.3
[22,stuckdata,[s2,"-","L"],[bit,"1"]]	20.5	4.3
[23,stuckdata,[s2,"-","R"],[bit,"0"]]	NAF	NAF
[24,stuckdata,[s2,"-","R"],[bit,"1"]]	36.4	4.3
[25,stuckdata,[s3,"-",""],[bit,"0"]]	NAF	NAF
[26,stuckdata,[s3,"-",""],[bit,"1"]]	17.2	1.5
[27,stuckdata,[s4,"-",""],[bit,"0"]]	17.2	1.5
[28,stuckdata,[s4,"-",""],[bit,"1"]]	NAF	NAF
[29,stuckdata,[s5,"-",""],[bit,"0"]]	100.0	5.4
[30,stuckdata,[s5,"-",""],[bit,"1"]]	78.4	5.8
[31,stuckdata,[s5,"-","L"],[bit,"0"]]	82.3	6.6
[32,stuckdata,[s5,"-","L"],[bit,"1"]]	81.5	6.2
[33,stuckdata,[s5,"-","R"],[bit,"0"]]	NAF	NAF
[34,stuckdata,[s5,"-","R"],[bit,"1"]]	97.8	6.3
[35,stuckdata,[s6,"-",""],[bit,"0"]]	22.8	2.7
[36,stuckdata,[s6,"-",""],[bit,"1"]]	NAF	NAF
[37,stuckdata,[s7,"-",""],[bit,"0"]]	NAF	NAF
[38,stuckdata,[s7,"-",""],[bit,"1"]]	22.9	2.7
[39,stuckdata,[s8,"-",""],[bit,"0"]]	258.4	8.2
[40,stuckdata,[s8,"-",""],[bit,"1"]]	Excl	Excl
[41,stuckdata,[s8,"-","L"],[bit,"0"]]	199.7	9.5
[42,stuckdata,[s8,"-","L"],[bit,"1"]]	Excl	Excl
[43,stuckdata,[s8,"-","LL"],[bit,"0"]]	Excl	Excl
[44,stuckdata,[s8,"-","LL"],[bit,"1"]]	200.5	9.8
[45,stuckdata,[s8,"-","R"],[bit,"0"]]	249.9	9.4
[46,stuckdata,[s8,"-","R"],[bit,"1"]]	Excl	Excl

[47,stuckdata,[s8,"-","RL"],[bit,"0"]]	254.6	10.3
[48,stuckdata,[s8,"-","RL"],[bit,"1"]]	Excl	Excl
[49,stuckdata,[s8,"-","RR"],[bit,"0"]]	Excl	Excl
[50,stuckdata,[s8,"-","RR"],[bit,"1"]]	NAF	NAF
[51,stuckdata,[s9,"-",""],[bit,"0"]]	88.6	4.8
[52,stuckdata,[s9,"-",""],[bit,"1"]]	86.9	4.8
[53,stuckdata,[s10,"-",""],[bit,"0"]]	87.1	5.0
[54,stuckdata,[s10,"-",""],[bit,"1"]]	89.5	5.0
[55,stuckdata,[s10,"-","L"],[bit,"0"]]	88.6	5.2
[56,stuckdata,[s10,"-","L"],[bit,"1"]]	111.4	5.2

9.22 FNTST VHDL 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;
```

9.23 FNTST Fault List

Fault	Barclay's Method	New Method
[1,assncntl,s1]	Fail	Fail
[2,assncntl,s2]	Fail	Fail
[3,assncntl,s3]	Fail	Fail
[4,assncntl,s4]	157.6	10.0
[5,microop,[s4,"-",""],bitand,bitor]	Fail	Fail
[6,stuckdata,[s1,"-",""],[bit,"0"]]	Fail	-
[7,stuckdata,[s1,"-",""],[bit,"1"]]	Fail	-
[8,stuckdata,[s2,"-",""],[bit,"0"]]	33.4	2.8
[9,stuckdata,[s2,"-",""],[bit,"1"]]	Fail	-
[10,stuckdata,[s3,"-",""],[bit,"0"]]	32.6	2.8
[11,stuckdata,[s3,"-",""],[bit,"1"]]	Fail	-
[12,stuckdata,[s4,"-",""],[bit,"0"]]	33.9	3.5
[13,stuckdata,[s4,"-",""],[bit,"1"]]	23.9	2.2
[14,stuckdata,[s4,"-","L"],[bit,"0"]]	33.1	3.4
[15,stuckdata,[s4,"-","L"],[bit,"1"]]	Fail	-
[16,stuckdata,[s4,"-","R"],[bit,"0"]]	33.5	3.4
[17,stuckdata,[s4,"-","R"],[bit,"1"]]	Fail	-

9.24 PRTY VHDL 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
```

9.25 PRTY Fault List

Fault	Barclay's Method	New Method
[1,assncntl,s1]	508.6	47.7
[2,microop,[s1,"-",""],bitxor,biteqv]	205.9	14.6
[3,microop,[s1,"-","L"],bitxor,biteqv]	169.7	16.8
[4,microop,[s1,"-","LL"],bitxor,biteqv]	161.2	18.4
[5,microop,[s1,"-","LR"],bitxor,biteqv]	161.6	18.4
[6,microop,[s1,"-","R"],bitxor,biteqv]	173.8	16.9
[7,microop,[s1,"-","RL"],bitxor,biteqv]	165.6	18.5
[8,microop,[s1,"-","RR"],bitxor,biteqv]	165.1	18.5
[9,stuckdata,[s1,"-",""],[bit,"0"]]	143.1	19.1
[10,stuckdata,[s1,"-",""],[bit,"1"]]	143.7	19.1
[11,stuckdata,[s1,"-","L"],[bit,"0"]]	166.7	18.8
[12,stuckdata,[s1,"-","L"],[bit,"1"]]	166.9	18.8
[13,stuckdata,[s1,"-","LL"],[bit,"0"]]	157.4	19.1
[14,stuckdata,[s1,"-","LL"],[bit,"1"]]	158.1	19.1
[15,stuckdata,[s1,"-","LLL"],[bit,"0"]]	157.0	19.8
[16,stuckdata,[s1,"-","LLL"],[bit,"1"]]	160.0	19.8
[17,stuckdata,[s1,"-","LLLL"],[bv,"00000000"]]	27.6	23.3
[18,stuckdata,[s1,"-","LLLL"],[bv,"11111111"]]	27.4	20.5
[19,stuckdata,[s1,"-","LLR"],[bit,"0"]]	158.9	19.9
[20,stuckdata,[s1,"-","LLR"],[bit,"1"]]	159.5	19.8
[21,stuckdata,[s1,"-","LLRL"],[bv,"00000000"]]	28.2	22.9
[22,stuckdata,[s1,"-","LLRL"],[bv,"11111111"]]	28.1	20.5
[23,stuckdata,[s1,"-","LR"],[bit,"0"]]	161.0	19.2
[24,stuckdata,[s1,"-","LR"],[bit,"1"]]	160.8	19.2
[25,stuckdata,[s1,"-","LRL"],[bit,"0"]]	159.2	19.9
[26,stuckdata,[s1,"-","LRL"],[bit,"1"]]	159.4	19.9
[27,stuckdata,[s1,"-","LRLL"],[bv,"00000000"]]	28.5	28.7
[28,stuckdata,[s1,"-","LRLL"],[bv,"11111111"]]	28.9	20.6
[29,stuckdata,[s1,"-","LRR"],[bit,"0"]]	162.3	20.0
[30,stuckdata,[s1,"-","LRR"],[bit,"1"]]	160.1	19.9
[31,stuckdata,[s1,"-","LRRL"],[bv,"00000000"]]	29.1	28.1
[32,stuckdata,[s1,"-","LRRL"],[bv,"11111111"]]	30.0	20.7
[33,stuckdata,[s1,"-","R"],[bit,"0"]]	171.3	18.9
[34,stuckdata,[s1,"-","R"],[bit,"1"]]	171.5	18.9
[35,stuckdata,[s1,"-","RL"],[bit,"0"]]	163.8	19.3
[36,stuckdata,[s1,"-","RL"],[bit,"1"]]	163.8	19.3
[37,stuckdata,[s1,"-","RLL"],[bit,"0"]]	163.1	20.0
[38,stuckdata,[s1,"-","RLL"],[bit,"1"]]	163.7	20.0
[39,stuckdata,[s1,"-","RLLL"],[bv,"00000000"]]	30.2	33.0
[40,stuckdata,[s1,"-","RLLL"],[bv,"11111111"]]	30.6	22.6
[41,stuckdata,[s1,"-","RLR"],[bit,"0"]]	168.7	22.0
[42,stuckdata,[s1,"-","RLR"],[bit,"1"]]	166.2	21.9
[43,stuckdata,[s1,"-","RLRL"],[bv,"00000000"]]	30.8	36.5
[44,stuckdata,[s1,"-","RLRL"],[bv,"11111111"]]	30.4	22.6
[45,stuckdata,[s1,"-","RR"],[bit,"0"]]	167.8	20.9
[46,stuckdata,[s1,"-","RR"],[bit,"1"]]	168.5	21.3

[47,stuckdata,[s1,"-","RRL"],[bit,"0"]]	166.9	22.0
[48,stuckdata,[s1,"-","RRL"],[bit,"1"]]	165.7	21.9
[49,stuckdata,[s1,"-","RRLL"],[bv,"00000000"]]	31.8	40.0
[50,stuckdata,[s1,"-","RRLL"],[bv,"11111111"]]	31.8	22.6
[51,stuckdata,[s1,"-","RRR"],[bit,"0"]]	173.5	21.9
[52,stuckdata,[s1,"-","RRR"],[bit,"1"]]	171.7	20.1
[53,stuckdata,[s1,"-","RRRL"],[bv,"00000000"]]	32.3	38.5
[54,stuckdata,[s1,"-","RRRL"],[bv,"11111111"]]	33.5	20.9

9.26 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
```

9.27 SHFT Fault List

Fault	Barclay's Method	New Method
[1,stuckthen,s1]	Fail	28.6
[2,stuckelse,s1]	443.6	6.1
[3,microop,[s1,"-",""],biteqv,bitxor]	Fail	5.4
[4,assncntl,s2]	160.0	7.2
[5,assncntl,s3]	870.0	17.7
[6,assncntl,s4]	541.7	23.2
[7,assncntl,s5]	150.8	24.6
[8,stuckthen,s6]	Excl	Excl
[9,stuckelse,s6]	276.1	23.5
[10,microop,[s6,"-",""],bitand,bitor]	Excl	Excl
[11,microop,[s6,"-","L"],bitnot,bitbuf]	Excl	Excl
[12,microop,[s6,"-","R"],biteqv,bitxor]	Excl	Excl
[13 deadclause s7,[[bv,"11"]]]	414.2	5.9
[14 deadclause s7,[[bv,"10"]]]	383.8	5.9
[15 deadclause s7,[[bv,"01"]]]	771.7	23.2
[16 deadclause s7,[[bv,"00"]]]	NAF	NAF
[17,assncntl,s8]	1024.0	23.2
[18,assncntl,s9]	3057.4	24.7
[19,assncntl,s10]	2429.7	18.6
[20,assncntl,s11]	393.9	5.9
[21,assncntl,s12]	396.5	5.9
[22,assncntl,s13]	2570.3	38.1
[23,assncntl,s14]	2482.7	31.5
[24,assncntl,s15]	805.4	17.3
[25,assncntl,s16]	758.9	6.0
[26,assncntl,s17]	7701.1	12.7
[27,assncntl,s18]	5022.3	12.8
[28,assncntl,s19]	752.5	6.0
[29,assncntl,s20]	Over	26.7
[30,assncntl,s21]	Over	8.7
[31,stuckdata,[s1,"-",""],[bit,"0"]]	Fail	-
[32,stuckdata,[s1,"-",""],[bit,"1"]]	Fail	-
[33,stuckdata,[s1,"-","L"],[bit,"0"]]	Fail	-
[34,stuckdata,[s1,"-","L"],[bit,"1"]]	Fail	-
[35,stuckdata,[s1,"-","R"],[bit,"0"]]	Fail	-
[36,stuckdata,[s1,"-","R"],[bit,"1"]]	Fail	-
[37,stuckdata,[s2,"-",""],[bit,"0"]]	NAF	NAF
[38,stuckdata,[s2,"-",""],[bit,"1"]]	27.8	1.7
139,stuckdata,[s3,"-",""],[bit,"0"]]	NAF	NAF
[40,stuckdata,[s3,"-",""],[bit,"1"]]	246.6	6.2
[41,stuckdata,[s4,"-",""],[bit,"0"]]	NAF	NAF
[42,stuckdata,[s4,"-",""],[bit,"1"]]	250.6	6.1
[43,stuckdata,[s5,"-",""],[bit,"0"]]	NAF	NAF
[44,stuckdata,[s5,"-",""],[bit,"1"]]	26.3	1.7
[45,stuckdata,[s6,"-",""],[bit,"0"]]	282.9	24.4
[46,stuckdata,[s6,"-",""],[bit,"1"]]	Excl	Excl

[47,stuckdata,[s6,"-","L"],[bit,"0"]]	303.7	24.9
[48,stuckdata,[s6,"-","L"],[bit,"1"]]	Excl	Excl
[49,stuckdata,[s6,"-","LL"],[bit,"0"]]	Excl	Excl
[50,stuckdata,[s6,"-","LL"],[bit,"1"]]	298.0	25.1
[51,stuckdata,[s6,"-","R"],[bit,"0"]]	388.7	24.6
[52,stuckdata,[s6,"-","R"],[bit,"1"]]	Excl	Excl
[53,stuckdata,[s6,"-","RL"],[bit,"0"]]	339.0	25.1
[54,stuckdata,[s6,"-","RL"],[bit,"1"]]	Excl	Excl
[55,stuckdata,[s6,"-","RR"],[bit,"0"]]	Excl	Excl
[56,stuckdata,[s6,"-","RR"],[bit,"1"]]	NAF	NAF
[57,stuckdata,[s7,"-",""],[bv,"00"]]	Fail	-
[58,stuckdata,[s7,"-",""],[bv,"11"]]	Fail	-
[59,stuckdata,[s8,"-",""],[bit,"0"]]	545.2	21.2
[60,stuckdata,[s8,"-",""],[bit,"1"]]	539.9	5.8
[61,stuckdata,[s9,"-",""],[bit,"0"]]	1165.9	22.9
[62,stuckdata,[s9,"-",""],[bit,"1"]]	1165.0	12.5
[63,stuckdata,[s10,"-",""],[bit,"0"]]	943.7	16.7
[64,stuckdata,[s10,"-",""],[bit,"1"]]	945.4	12.6
[65,stuckdata,[s11,"-",""],[bit,"0"]]	98.7	4.2
[66,stuckdata,[s11,"-",""],[bit,"1"]]	97.2	4.2
[67,stuckdata,[s12,"-",""],[bit,"0"]]	99.3	4.2
[68,stuckdata,[s12,"-",""],[bit,"1"]]	101.2	4.2
[69,stuckdata,[s13,"-",""],[bit,"0"]]	1054.4	35.8
[70,stuckdata,[s13,"-",""],[bit,"1"]]	1021.2	12.6
[71,stuckdata,[s14,"-",""],[bit,"0"]]	1223.8	29.2
[72,stuckdata,[s14,"-",""],[bit,"1"]]	1203.6	12.6
[73,stuckdata,[s15,"-",""],[bit,"0"]]	537.9	15.2
[74,stuckdata,[s15,"-",""],[bit,"1"]]	542.5	5.8
[75,stuckdata,[s16,"-",""],[bit,"0"]]	99.2	4.2
[76,stuckdata,[s16,"-",""],[bit,"1"]]	96.5	4.2
[77,stuckdata,[s17,"-",""],[bit,"0"]]	434.8	11.0
[78,stuckdata,[s17,"-",""],[bit,"1"]]	431.2	11.0
[79,stuckdata,[s18,"-",""],[bit,"0"]]	427.2	11.0
[80,stuckdata,[s18,"-",""],[bit,"1"]]	430.1	11.0
[81,stuckdata,[s19,"-",""],[bit,"0"]]	99.1	4.2
[82,stuckdata,[s19,"-",""],[bit,"1"]]	100.6	4.2
[83,stuckdata,[s20,"-",""],[bit,"0"]]	108.0	23.5
[84,stuckdata,[s20,"-",""],[bit,"1"]]	111.6	2.1
[85,stuckdata,[s21,"-",""],[bit,"0"]]	103.9	5.9
[86,stuckdata,[s21,"-",""],[bit,"1"]]	101.0	2.1

9.28 SHFTV VHDL Model

```
        entity SHFTV
          (CLOCK,
          CLEAR,
          LEFTIN,
          RIGHTIN,
          CONTROL : in BIT;
          D : in BIT_VECTOR(3 downto 0);
          RIGHTOUT,
          LEFTOUT : out BIT) is
        end SHFTV;

        architecture arch of SHFTV is

        process(CLEAR,CLOCK,q)
        signal q:BIT_VECTOR(3 downto 0);
        begin
s1:     if CLEAR = '0' then
s2:       q < = "0000";
        else
s6:       if not CLOCK'STABLE and CLOCK = '1' then
s7:         case CONTROL is
              when "00" = >
                null;          -- no op.
              when "01" = >
s8:             Q < = Q(2 downto 0) & (LEFTIN);   -- shift left
                what is vhdl for BIT-BV ?
              when "10" = >
s12:            Q < = (RIGHTIN) & Q(3 downto 1); -- shift right
                what is vhdl for BIT-BV ?
              when "11" = >
s16:            Q < = D;     -- parallel load
            end case;
          end if;
        end if;
s20:    LEFTOUT < = q(3);
s21:    RIGHTOUT < = q(0);
        end process;
        end arch;
```

9.29 SHFTV Fault List

Fault	Barclay's Method	New Method
[1,stuckthen,s1]	Fail	31.1
[2,stuckelse,s1]	483.9	12.5
[3,microop,[s1,"-",""],biteqv,bitxor]	Fail	11.5
[4,assncntl,s2]	168.0	13.1
[5,stuckthen,s6]	Excl	Excl
[6,stuckelse,s6]	253.5	18.7
[7,microop,[s6,"-",""],bitand,bitor]	Excl	Excl
[8,microop,[s6,"-","L"],bitnot,bitbuf]	Excl	Excl
[9,microop,[s6,"-","R"],biteqv,bitxor]	Excl	Excl
[10,deadclause,s7,[[bv,"11"]]]	249.2	14.7
[11,deadclause,s7,[[bv,"10"]]]	707.4	18.3
[12,deadclause,s7,[[bv,"01"]]]	656.7	18.3
[13,deadclause,s7,[[bv,"00"]]]	NAF	NAF
[14,assncntl,s8]	677.3	19.3
[15,assncntl,s12]	732.6	18.3
[16,assncntl,s16]	1202.0	14.8
[17,assncntl,s20]	Over	41.5
[18,assncntl,s21]	Over	13.8
[19,stuckdata,[s1,"-",""],[bit,"0"]]	Fail	-
[20,stuckdata,[s1,"-",""],[bit,"1"]]	Fail	-
[21,stuckdata,[s1,"-","L"],[bit,"0"]]	Fail	-
[22,stuckdata,[s1,"-","L"],[bit,"1"]]	Fail	-
[23,stuckdata,[s1,"-","R"],[bit,"0"]]	NAF	NAF
[24,stuckdata,[s1,"-","R"],[bit,"1"]]	Fail	-
[25,stuckdata,[s2,"-",""],[bv,"0000"]]	NAF	NAF
[26,stuckdata,[s2,"-",""],[bv,"1111"]]	30.4	2.2
[27,stuckdata,[s6,"-",""],[bit,"0"]]	255.2	19.7
[28,stuckdata,[s6,"-",""],[bit,"1"]]	Excl	Excl
[29,stuckdata,[s6,"-","L"],[bit,"0"]]	264.1	19.9
[30,stuckdata,[s6,"-","L"],[bit,"1"]]	Excl	Excl
[31,stuckdata,[s6,"-","LL"],[bit,"0"]]	Excl	Excl
[32,stuckdata,[s6,"-","LL"],[bit,"1"]]	263.2	20.1
[33,stuckdata,[s6,"-","R"],[bit,"0"]]	289.1	19.7
[34,stuckdata,[s6,"-","R"],[bit,"1"]]	Excl	Excl
[35,stuckdata,[s6,"-","RL"],[bit,"0"]]	288.7	20.1
[36,stuckdata,[s6,"-","RL"],[bit,"1"]]	Excl	Excl
[37,stuckdata,[s6,"-","RR"],[bit,"0"]]	Excl	Excl
[38,stuckdata,[s6,"-","RR"],[bit,"1"]]	NAF	NAF
[39,stuckdata,[s7,"-",""],[bv,"00"]]	Fail	-
[40,stuckdata,[s7,"-",""],[bv,"11"]]	Fail	-
[41,stuckdata,[s8,"-",""],[bv,"0000"]]	559.3	8.6
[42,stuckdata,[s8,"-",""],[bv,"1111"]]	559.7	8.5
[43,stuckdata,[s8,"-","L"],[bv,"000"]]	Fail	25.2
[44,stuckdata,[s8,"-","L"],[bv,"111"]]	Fail	8.1
[45,stuckdata,[s8,"-","LL"],[bv,"0000"]]	Fail	25.7
[46,stuckdata,[s8,"-","LL"],[bv,"1111"]]	Fail	8.3

[47,stuckdata,[s8,"-","R"],[bv,"0"]]	Fail	8.0
[48,stuckdata,[s8,"-","R"],[bv,"1"]]	Fail	7.9
[49,stuckdata,[s8,"-","RL"],[bit,"0"]]	Fail	7.9
[50,stuckdata,[s8,"-","RL"],[bit,"1"]]	Fail	7.9
[51,stuckdata,[s12,"-",""],[bv,"0000"]]	574.9	24.2
[52,stuckdata,[s12,"-",""],[bv,"1111"]]	578.1	8.2
[53,stuckdata,[s12,"-","L"],[bv,"0"]]	Fail	8.3
[54,stuckdata,[s12,"-","L"],[bv,"1"]]	Fail	8.2
[55,stuckdata,[s12,"-","LL"],[bit,"0"]]	Fail	8.3
[56,stuckdata,[s12,"-","LL"],[bit,"1"]]	Fail	8.2
[57,stuckdata,[s12,"-","R"],[bv,"000"]]	Fail	23.9
[58,stuckdata,[s12,"-","R"],[bv,"111"]]	Fail	7.8
[59,stuckdata,[s12,"-","RL"],[bv,"0000"]]	Fail	52.3*
[60,stuckdata,[s12,"-","RL"],[bv,"1111"]]	Fail	8.0
[61,stuckdata,[s16,"-",""],[bv,"0000"]]	99.4	4.9
[62,stuckdata,[s16,"-",""],[bv,"1111"]]	102.4	4.7
[63,stuckdata,[s20,"-",""],[bit,"0"]]	115.6	32.6
[64,stuckdata,[s20,"-",""],[bit,"1"]]	120.0	2.8
[65,stuckdata,[s20,"-","L"],[bv,"0000"]]	113.7	34.0*
[66,stuckdata,[s20,"-","L"],[bv,"1111"]]	109.9	2.9
[67,stuckdata,[s21,"-",""],[bit,"0"]]	112.1	8.6
[68,stuckdata,[s21,"-",""],[bit,"1"]]	115.9	2.8
[69,stuckdata,[s21,"-","L"],[bv,"0000"]]	118.6	34.1*
[70,stuckdata,[s21,"-","L"],[bv,"1111"]]	116.6	2.9

9.30 UARTO VHDL Model

```
        entity UART(
          RESET,
          WRSTRB,
          CLOCK : in BIT;
          DATABUS: in BIT_VECTOR(1 downto 0);
          DATAOUT : out BIT
          TXBUSY : out BIT
          ) is
        end UART;

        architecture ARCH of UART is

        process(...)
        signal
          TXCNT : BIT_VECTOR(2 downto 0);       -- transmit BIT couner
          TXREG : BIT_VECTOR(3 downto 0);

        begin
   s01: if RESET='0' then        -- RESET transmit side to wait for byte
   s02:   TXBUSY <= '0';          -- clear transmit busy flag
        else
   s03:   if WRSTRB='1' and not WRSTRB'STABLE then -- write strobe
   s04:     TXCNT <= "11";         -- set count for: start + data + stop
   s05:     TXREG <= "1" & DATABUS & "0"; -- (r to l) start, data, stop
   s06:     TXBUSY <= '1';                -- transmitting
          end if;

   s07:   if (CLOCK='1' and not CLOCK'STABLE) and TXBUSY='1' then
   s08:     DATAOUT <= TXREG(0);           -- output low BIT
   s09:     TXREG <= "1" & TXREG(3 downto 1);   -- shift BITs over
   s10:     TXCNT <= bvsub(TXCNT,"01");    -- count BITs out
   s11:     if TXCNT = "00" then           -- note delta delay!
   s12:       TXBUSY <= '0';                 -- done transmitting
            end if;
          end if;
        end if
        end process;
        end arch;
```

9.31 UARTO Fault List

Fault	Barclay's Method	New Method
[1,stuckthen,s1]	Fail	
[2,stuckelse,s1]	273.6	5.1
[3,microop,[s1,"-",""],biteqv,bitxor]	Fail	
[4,assncntl,s2]	127.5	5.7
[5,stuckthen,s3]	Excl	Excl
[6,stuckelse,s3]	1140.6	5.5
[7,microop,[s3,"-",""],bitand,bitor]	Excl	Excl
[8,microop,[s3,"-","L"],biteqv,bitxor]	Excl	Excl
[9,microop,[s3,"-","R"],bitnot,bitbuf]	Excl	Excl
[10,assncntl,s4]	Fail	
[11,assncntl,s5]	Fail	
[12,assncntl,s6]	Over	
[13,stuckthen,s7]	Excl	Excl
[14,stuckelse,s7]	Fail	
[15,microop,[s7,"-",""],bitand,bitor]	Excl	Excl
[16,microop,[s7,"-","L"],bitand,bitor]	Excl	Excl
[17,microop,[s7,"-","LL"],biteqv,bitxor]	Excl	Excl
[18,microop,[s7,"-","LR"],bitnot,bitbuf]	Excl	Excl
[19,microop,[s7,"-","R"],biteqv,bitxor]	Fail	
[20,assncntl,s8]	Over	
[21,assncntl,s9]	Over	
[22,assncntl,s10]	Over	
[23,microop,[s10,"-",""],bvsub,bvadd]	Over	
[24,stuckthen,s11]	Over	
[25,stuckelse,s11]	Over	
[26,microop,[s11,"-",""],bveq,bvneq]	9428.7	175.1
[27,assncntl,s12]	Over	

Appendix C: Rule Definitions

9.32 Introduction

This appendix defines the rules governing the three basic operations that drive the test generation algorithm: justification, propagation, and execution. The definitions are presented as a set of all possible situations to which the rules may be applied. For each case, the action performed is given. The rules are designed to be applied recursively until a terminal case is reached.

9.33 Justification

A justification operation takes an expression and a desired value as its starting point. (This assumes that the routine calling the justification rules has already determined the source expression in the VHDL statement.) Each case is defined by the type of expression it handles.

- The expression is a literal value, and the desired value is equal to this literal. This is a terminal case, and the rule succeeds.
- The expression is a literal value, but the desired value is not equal to this literal. This is a terminal case, but the rule fails.
- The expression is a "don't care" (X) value. This is a terminal case in which the rule succeeds.
- The expression is a single object which is not an input pin. A new goal is created, which will attempt to justify the desired value into the object.
- The expression is a single object which is an input pin. This is a terminal case in which the rule succeeds. The input pin is assigned the desired value.
- The expression is a unary operation on some other (simpler) expression. The value which the simpler expression must have is determined, based on the desired value and the unary operation. A new justification goal is created, which will attempt to justify the new value into the new expression.
- The expression is a binary operation on two other (simpler) expressions. The value which the two new expressions must have are determined, based on the desired value and the binary operation. Two new justification goals are created, which attempt to justify the new values into the new expressions.

9.34 Propagation

A propagation operation takes a good/bad value pair and a statement through which to propagate it. The specification of the statement indicates the expression through which the values are being propagated. The cases are defined based on the type of statement and the condition of the current expression within the statement. For control statements, the cases are defined based on the statements under the good and

bad clauses. (The "good" clause is the clause executed if the control expression is evaluated with the good value substituted. The "bad" clause is similarly defined.) A statement under one of the clauses is selected, and an appropriate new propagation goal is created.

- The statement is an assignment statement, and the source expression is a single object which is an output pin. This is a terminal case in which the propagation succeeds.

- The statement is an assignment statement, and the source expression is a single object which is not an input pin. A new statement which uses the object in its expression is selected, and a new propagation goal is spawned, based on the new statement.

- The statement is an IF statement. Two assignment statements assigning to the same object exist; one under the good clause and one under the bad clause. The statements are used to propagate the good/bad values by justifying the good value into the source expression of the statement under the good clause, and justifying the bad value into the source expression under the bad clause. Propagation continues with the good statement.

- The statement is an IF statement. An assignment statement exists under the good clause, but none exist under the bad clause. The destination object of the assignment statement under the good clause is loaded with the bad value. The good value is then justified into the source expression of the assignment statement. When the IF statement executes, the destination object will take the good value if the fault is not present, or retain the bad value if it is.

- The statement is an IF statement. An assignment statement exists under the bad clause, but none exists under the good clause. The destination object of the assignment statement under the bad clause is loaded with the good value. The bad value is then justified into the source expression of the assignment statement. When the IF statement executes, the good destination object will take the bad value if the fault is present, or retain the good value if it is not.

- The statement is an IF statement. No assignment statements exist under either clause. This is a terminal case in which the propagation fails. This case can only result when an if statement does not control any data operations, which should never occur in normal modeling.

- The statement is a CASE statement. Two assignment statements exist, one under the good clause and one under the bad clause. This case is identical to the corresponding IF statement case.

- The statement is a CASE statement. An assignment statement exists under the good clause, but none exists under the bad clause. This case is identical to the corresponding IF statement case.

- The statement is a CASE statement. An assignment statement exists under the bad clause, but none exists under the good clause. This case is identical to the corresponding IF statement case.

- The statement is a CASE statement. No assignment statements exist under any clause. This is a terminal case in which propagation fails. This is an abnormal case that should not occur in normal modeling.
- The statement is of any type; the expression is part of the whole expression within the statement. A new good/bad value pair is determined based on the operation involved in the expression. A new expression is created, incorporating the original piece of the expression. Propagation continues with the new expression and value pair.

9.35 Execution

The execution operation is defined in terms of a justification goal. For a given statement, all parent statements are determined. The values to which each control expression in each parent statement must be set is selected. This list of desired values and control expressions makes up a list of justification goals. When each justification operation has been completed, the execution operation has been solved. (Note that the execution of a top-level statement is a special case: since a top-level statement always executes, the goal is inherently solved.)

Vita

Michael D. O'Neill was born on June 9, 1964. After graduating from Taconic Hills Central High School in 1982, he enrolled at Virginia Polytechnic Institute and State University in September 1982. After receiving the Bachelor of Science degree in Electrical Engineering, graduating *Cum Laude*, in 1986, he entered the graduate program at Virginia Tech in September of 1986. He completed requirements for the Master of Science degree in Electrical Engineering in December of 1987.

Michael has been employed with IBM Corporation in Poughkeepsie, New York since December of 1987.

Michael is a member of the IEEE and the IEEE Computer Society. He is a member of ΦΗΣ, ΤΒΠ and HKN honor societies.