An Efficient Test Generation Algorithm

for Behavioral Descriptions of Digital Devices

by

Dhanendra Dinesh Jani

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

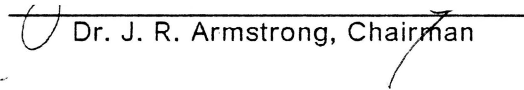
in partial fulfillment of the requirements for the degree of

Master of Science

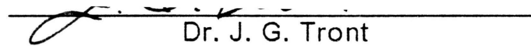
in

Electrical Engineering

APPROVED:



Dr. J. R. Armstrong, Chairman



Dr. J. G. Tront

Dr. F. G. Gray

December, 1988

Blacksburg, Virginia

An Efficient Test Generation Algorithm

for Behavioral Descriptions of Digital Devices

by

Dhanendra Dinesh Jani

Dr. J. R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

An efficient test generation algorithm for behavioral descriptions is discussed. It generates tests for behavioral dataflow descriptions of digital circuits written in VHDL. The algorithm accepts input descriptions containing multiple process statements and concurrent signal assignment statements. The fault model based on previous research includes micro-operation and control faults. The test generation algorithm uses artificial intelligence techniques of goal trees and rule databases and it can make use of human understanding of the device model to generate more efficient tests. An improved timing model helps detect conflicts more quickly and improves the speed performance of the algorithm. The test generation algorithm has been used to generate tests for complex circuits. Results of fault coverage experiments for some of these circuits is presented.

Acknowledgements

I would like to thank Dr. Armstrong for the challenging environment he has made available, and the support and guidance he lent this research. I would also like to thank Dr. Tront and Dr. Gray for serving as members of my committee. I wish to express my gratitude to my teachers and friends for their inspiration and constant support. I dedicate this work to my parents without whose love and care I would be nothing.

Table of Contents

Introduction **1**

1.1 Contents 4

Literature Survey **6**

2.1 Digital Circuit Faults and Fault Modeling 6

2.2 Test Generation Techniques 7

2.3 Gate Level Test Generation 7

2.4 Register Level Methods 9

2.5 Graph Methods 12

2.6 Binary Decision Diagrams 13

2.7 AI Methods 14

2.8 High Level HDLs 15

The Hardware Description Language **19**

3.1 Behavioral Data Flow Descriptions 19

3.2 VHDL Subset Used 20

3.3 Modeling Sequential Behavior 21

3.4 Tester Assumption 22

Behavioral Fault Model **24**

Previous Work **28**

New Method **34**

6.1 Improved Timing Model 34

6.2 Two phase tests 36

6.3 Extension to Multiple Process Statements 38

6.4 Solving VIE goals during preprocessing 39

6.5 Eliminating storage of justification rules 39

6.6 Increased Forward Implication 40

6.7 User Assists to the Test Generation algorithm 41

6.8 The Test Generation System 42

6.8.1 Preprocessor 43

6.8.1.1 VHDL to Prolog Conversion 43

6.8.1.2 Pre-processing the Prolog representation 43

6.8.2 Formation of the fault list 45

6.8.3 Test Generation Algorithm 46

6.8.3.1 Problem Solving Method 46

6.8.3.2 High Level Goals 47

6.8.3.3 Test requirements for different Fault Models 57

6.8.3.4 Test Generation Example 62

Results **65**

7.1 CKTA 67

7.2 REGISTER 67

7.3 COCN 68

7.4 I8212 69

7.5 UARTO 70

7.6 Speed Performance and Memory Requirement 71

Analysis and Suggestions 72

8.1 Forward Implication 72

8.2 Reconvergent Fanout 73

8.3 Traversal of the fault site during propagation 74

8.4 Extension to Board Level Testing 74

8.5 Retention of Past Work 75

8.6 Increasing Test Effectiveness 76

8.7 Expansion of the VHDL subset 76

Conclusions 77

Bibliography 78

User's Manual 81

A.1 Preprocessor 83

A.1.1 Translator 83

A.1.2 Intermediate Form Extractor 84

A.2 Fault List Extractor 87

A.3 Behavioral Test Generator 88

Circuit Models and Fault Lists 91

Vita 104

List of Illustrations

Figure 1. Structural and Behavioral descriptions of an example circuit 3

Figure 2. Tester Assumption 23

Figure 3. New Timing Model 35

Figure 4. Prolog representation of Figure 1 44

Figure 5. Flow Diagram of the Test Generation Algorithm 48

Figure 6. VHDL Model for an 8-bit Register 63

Figure 7. A Behavioral Test Generation System 82

Figure 8. Available micro-operations and their description 85

List of Tables

Table 1. Results of experiments with BTG 70

Table 2. Results of fault coverage experiment 70

Chapter I

Introduction

With the increasing complexity of VLSI circuits, test generation has become one of the most complicated and time-consuming problems in digital system design. Classical algorithms [1-3] which use gate level descriptions cannot generate tests for more complex circuits because of the possible exponential growth in run time. Also, they are largely limited to testing combinational circuits. Thus, it has become very important to develop an approach to modeling faults at a higher level of abstraction, and a method that can automatically generate tests to detect these faults. Several approaches [5-19] have been presented that attempt to reduce the test complexity of VLSI systems by using a higher level of abstraction.

Recently developed high level test generation methods make use of various Hardware Description Languages(HDL's) [20-23]. Hardware description languages such as VHDL, HELIX, and ISP′ provide a convenient tool for the functional specification and simulation of digital circuits, reducing the quantity and detail of the information which the designer must manage. They allow system designers to describe digital

circuits at varying levels of abstraction, i.e., the gate, register, or chip level. However, at each abstraction level a design can be defined in either of two domains [31]:

Structural Domain - a domain in which a component is described in terms of an interconnection of more primitive components.

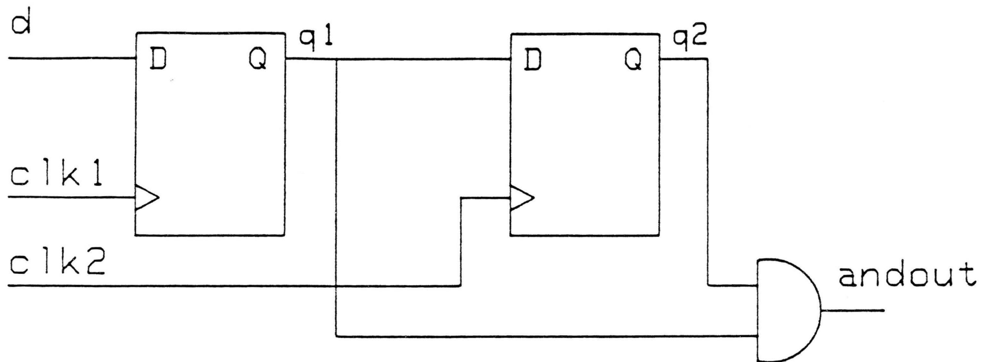
Behavioral Domain - a domain in which a component is described by defining its input/output response by means of a procedure.

Figure 1 on page 3 shows a circuit described both structurally and behaviorally.

While HDL's can be of considerable help to the designer, they can also be of help to the test engineer. The handling of reduced information (expressed using HDL constructs) could mean the generation of tests more easily. Traditionally, fault modeling and test generation have been carried out in the structural domain at the gate level. The recent test generation methods which use HDL's also work for descriptions in the structural domain. However, this thesis discusses a test generation algorithm which uses behavioral descriptions of digital devices for generating test vectors.

Recently Barclay and Armstrong [23] proposed an algorithm for generating tests from chip level device descriptions written in VHDL. A chip level fault model derived from HDL constructs is used to define faults and fault sensitization requirements. Artificial intelligence techniques of goal trees and goal solving are used to represent and manipulate sensitization, justification, and propagation requirements. In [27] O'Neill describes an improved algorithm which offers better speed performance.

Structural Description of Example circuit



Behavioral Description of Example circuit

```
    entity CKTA is
     (D, CLK1, CLK2 : in BIT;
      ANDOUT : out BIT)
    end CKTA;

    architecture BEHAVIOR of CKTA is

      signal Q1, Q2 : BIT;

s1: process(CLK1)
    begin
s2:   if CLK1 = '1' then
s3:      Q1 <= D;
      end if;
    end process;

s4: process(CLK2)
    begin
s5:   if CLK2 = '1' then
s6:      Q2 <= Q1;
      end if;
    end process;

s7: ANDOUT <= Q1 and Q2;

    end BEHAVIOR;
```

Figure 1. Structural and Behavioral descriptions of an example circuit

This thesis describes an efficient test generation algorithm derived from O'Neill's method, which incorporates the following

- An improved Timing Model
- A generalized two phase test technique
- An extended subset of VHDL
- Improved high level goal types ("justification", "execution", and "propagation")
- Reduced memory space requirement and comparable speed performance
- User assists to the test generation process

Besides this, the thesis also makes explicit the principles of behavioral test generation which may have been implicit in previous work.

1.1 Contents

Chapter II, "Literature Review", reviews previous approaches to test generation, from gate-level methods to functional methods.

Chapter III, "The Hardware Description Language", discusses the subset of VHDL used by the new algorithm.

Chapter IV, "Behavioral Fault Model", discusses the fault model used to define faults in VHDL behavioral descriptions.

Chapter V, "Previous Work", summarizes Barclay's and O'Neil's test generation algorithms.

Chapter VI, "New Method", discusses the approaches taken in the new algorithm. It also describes the new test generation system.

Chapter VII, ″Results″, demonstrates the improvement gained by applying the new algorithm to a set of complex circuit models. Also, results of fault coverage experiments for some of these circuits are presented.

Chapter VIII, ″Analysis and Suggestions″, discusses the limitations of the current algorithm and describes areas for future work.

Chapter IX, ″Conclusions″, draws conclusions concerning the algorithm and its implementation.

Appendix A, ″User's Manual″, describes how to use the new test generation system.

Appendix B, ″Circuit Models and Fault Lists″, shows the VHDL models used and gives the fault lists for each model.

Chapter II

Literature Survey

2.1 Digital Circuit Faults and Fault Modeling

A *fault* in a circuit is a physical defect which may cause the circuit to deviate from its specified behavior. Faults may occur due to errors caused during the layout or due to defects occurring during the fabrication of an integrated circuit. It is however impossible to list all defects that could possibly occur in a given circuit. A *fault model* thus selects a subset of the *all possible physical defects*. This subset represents the set of faults more likely to occur during the manufacture and/or the operation of the given circuit. Thus, a given fault model does not model all possible physical defects, but a reasonable subset of it so that if the circuit is free of the modeled faults it will probably work as desired.

The list of faults obtained from a given fault model is called the *fault list*. Test generation systems usually specify a particular fault model and attempt to generate tests for all faults in the fault list.

2.2 Test Generation Techniques

The test generation process consists of three main activities [4]:

1. Selecting a good descriptive model, at a suitable level, for the circuit under consideration. Such a model should reflect the exact behavior of the system in all its possible modes of operation.
2. Developing a fault model to define the types of faults that will be considered during the test generation. In selecting a fault model, the percentage of possible faults covered by the model should be maximized.
3. Generating tests to detect all the faults in the model. Designing a test sequence to detect a certain fault in a digital circuit involves sensitizing the fault and then propagating the fault syndrome to an observable output.

Test generation procedures have been presented for many levels of abstraction of design representations.

2.3 Gate Level Test Generation

The classical approach of modeling digital circuits as a group of connected gates and flip-flops has been used extensively. Using this level of abstraction, test designers have introduced many types of fault models, such as the classical stuck-at model. This model assumes that a fault in a logic gate results in one of its inputs or the output being fixed to either a logic 0 (stuck-at-0) or a logic 1 (stuck-at-1). The first algo-

rithmic method to generate a complete set of tests for gate level stuck at faults was the D-algorithm [1].

The D-algorithm uses the symbol 'D' to indicate a value of '1 in a good circuit and a '0' in the circuit containing the fault. The first step in the algorithm involves finding inputs that will force a faulty value to appear at the fault site if the fault exists. The next step is to propagate the effects of the fault (a D or $\overline{D}$) from the fault site to an observable output. Any conditions required on the propagation path are then satisfied by a consistency check which establishes input conditions to generate the test. When conflicts are discovered, backtracking is used to recover from bad choices.

PODEM (Path oriented decision making) [2], has been shown to be very efficient for circuits having reconvergent fanout and significantly more efficient than the D-algorithm over the general spectrum of combinational circuits. For an n-input circuit, the test generation problem is viewed as an n-dimensional 0-1 search space. Heuristics are used to achieve an efficient implicit search of the space of all possible primary input patterns until either a test is found or the space is exhausted.

In order to accelerate an algorithm for test generation, it is necessary to reduce the number of occurrences of backtracks in the algorithm and to shorten the the processing time between backtracks. A refinement of the PODEM, called FAN [3] accelerates the the PODEM algorithm by using techniques to reduce the number of backtracks and by performing special processing of fanout points. Their experimental results on large combinational circuits show that the FAN algorithm is is faster and more efficient than PODEM in computing time, the number of backtracks, and test coverage.

However for VLSI circuits having many thousands of gates, the gate level approach to test generation is not very feasible. It has been shown that the problem of generating tests to detect single stuck-at faults in a combinational circuit modeled at the gate level is an NP-complete problem. Moreover, if the circuit is sequential, the problem can become even more difficult. Thus, it is necessary to view the circuit under test at a level higher than the gate level to generate tests more efficiently.

2.4 Register Level Methods

The register connection level methods model the circuit under test at the register level, a level above the gate level. At this level, functional units such as registers, multiplexers, ALUs. ROMs, and RAMs are considered basic circuit primitives.

Shteingart et al. [5] presented RTG, a register level test generator. RTG is targeted for boards containing SSI, MSI, and small LSI components. It uses gate level models for combinational components, and register level model for sequential components. By lumping, isolating, and limiting the sequentiality of the model within the RL sequential components, a sequential circuit is, in effect, converted into a combinational circuit. Techniques similar to the 9-V algorithm are used to perform propagation and justification through combinational parts of the circuit in conjunction with certain primitive routines to handle sequential components. A disadvantage of the RTG is that it cannot generate tests for all types of circuits, but only for circuits designed using certain RTG design for testability' rules.

Marlett [6] presented EBT, a test generation technique for highly sequential components. It effectively handles the problems of initialization, fault steering, device mod-

eling, and races, the major test generation problems associated with sequential logic. EBT uses a truth table like representation for combinational devices and for sequential devices a table with pairs of rows is used to define the time dependence. The first row of each pair specifies data for the CTF (current time frame) and the second row specifies data for the PTF (previous time frame). The test generation software first selects a topological path (TP), from the fault site to an output pin. It then processes the TP in reverse order from the circuit output to the site of the fault, determining at each step the device input combination which produces the required interior signal in the given time frame. To reduce backtracking, decision numbers are assigned to each node to find the most recent related decision; backtracking skipps to the node having the highest decision number, thus bypassing all unrelated decisions.

Above the register connection level is the register transfer level in which the circuit is described using Register Transfer Languages (RTL's). RTL's are hardware description languages that describe a circuit in terms of transfers between registers and operations on data stored in registers. An RTL description does not necessarily reflect the structure of the actual circuit.

Hill and Huey [7] developed SCIRTSS (Sequential Circuit Test Search System), an automatic fault test generator that worked with a RTL called AHPL (A Hardware Programming Language). A heuristic graph search routine uses the AHPL description of the machine to find a path from the present state to any desired state. A fault injection logic simulator verifies the results of the graph search. For a given fault, the D-algorithm is used to generate a set of D-vectors consisting of external inputs and flip-flop state specifications, which will sensitize a path from the fault site to a circuit

output or into a flip-flop. A disadvantage of this system is that it cannot generate tests for faults in control structures of the circuit.

Based on an RTL description, Su and others [8-9] give two approaches for functional testing. In the first method, the RTL description is converted into a data graph. The D-algorithm is then used to generate tests for functional faults. However, it cannot generate tests for control faults where the sites of the fault is not observable (which cannot be handled by other algorithms too). As a second method, symbolic execution technique is used to generate tests for detecting control faults.

Lin and Su [10], further proposed the S-algorithm, a symbolic execution approach for functional testing based upon an RTL description. An RTL fault model based on a well defined RTL is described. The S-algorithm generates tests by injecting RT level faults (jump faults, condition faults, data transfer faults, register decoding faults, and operator decoding faults) and then simulating symbolically both the good and faulty machines. The symbolic inequalities obtained are then solved; input values are selected such that the constraints are satisfied and the results for good and bad execution differ. This set of input values, if they are found, constitute a test. They report that the generated test sets give good functional level fault coverages.

Stamelos et al. [11] also used an RTL to generate test vectors. A fault extractor is used in mapping physical failures to the fault model at the RTL level. They report that high quality test sequences are generated by allowing the user to intervene during test pattern generation.

The RTL approaches have their inherent limits since the asynchronous behavior of digital devices cannot be described using RTL's.

2.5 Graph Methods

The graph methods model the circuit under test as directed graphs and use graph theoretic methods to simplify the test generation problem.

Robach and Saucier [12] proposed an approach in which each instruction of a microprocessor is represented by an *abstract execution graph*. In such a bipartite graph, each node represents a memory element or a micro-operation and each arc represents the data flow between the nodes. The instructions of a microprocessor are grouped into classes, where all instructions in the same class have the same shape of abstract execution graph. The instructions in a class are further grouped into subclasses according to the accessibility and observability of each instruction. The instructions in each subclass can also be organized into a well ordered sequence by the functional covering algorithm. Functional testing can be done by either verifying correct execution of instructions from the smallest subclass to the largest subclass (start-small test) or verifying correct execution from the largest to the smallest one (start-big test).

Thatte and Abraham [13] modeled a microprocessor as a directed graph (S-graph) where nodes are registers and edges represent data flows. A fault model is developed which is independent of the implementation details. It considers register decoding faults, instruction decoding faults, and data transfer faults. A disadvantage of this technique is that the S-graph does not contain the control flow information.

These graph approaches are well suites to microprocessor testing, but they lack the generality necessary for application to all types of digital systems.

2.6 Binary Decision Diagrams

Akers [14] first proposed the use of BDD's for functional test generation. A BDD is a graphical means of describing logical operations of digital functions. Given the values on inputs, a user can easily determine the resulting output by simply traversing the branches of the BDD till he exits it. An *experiment* for an output variable is defined as a path in the BDD from the output value to an exit value. Akers argues that functional testing is the process of verifying that a given device functions as it is supposed to. So, in his test generation process, all *experiments* derived from BDD's compose the test.

Abadir and Reghbati [15] propose a path tracing method as connected modules, where each module is described as a BDD. Sequential circuits are treated as combinational ones by defining current and next state variables and a modified form of the D-algorithm is employed. Faults are categorized into two classes - stuck-at faults and module's functional faults which adversely affect the execution of one of its experiments.

Chang et al. [16] also defined functional faults using BDD's. A logic module performing any other undesired function in addition to the desired function was another fault added to the fault model.

The BDD approaches are not suitable for VLSI circuits since they require knowledge of the the detailed internal circuit structure. Also the resultant BDD becomes very complex for circuits of this size.

2.7 AI Methods

The AI methods use AI techniques (e.g heuristics,expert system) to reduce the problem of combinatorial explosion. Heuristics help to guide the search space through the number of choices that grow exponentially. Expert systems tend to apply techniques based on the premise that the techniques have worked in the past, or that the techniques resemble those used by humans to solve the problem.

The HITEST [17] test generator is an example of an expert system. It attempts to capture the essential characteristics of experienced human test engineers and embodies them in a system software environment. In that sense it can be called an knowledge-based AI system. The system permits test engineering knowledge to be stored in usable form by the system and is also interactive so that a test engineer can take control and apply guidance and corrections if needed.

Shirly [18] proposed an approach which generates tests using a device's designed behavior, i. e., the operations that a device is intended to carry out. This approach attempts to reduce the search space by limiting the problem domain to a device's designed behavior without considering the device's unintended behavior. It also uses concepts of a hierarchical planner by partitioning the problem domain in two parts, one that focuses closely on a single component and another that involves the rest of the circuit. Although this approach reduces the search space by simplifying the problem domain and using the concepts of a planner, it has the following two drawbacks. First, it cannot generate tests which determine whether unintended functions are additionally performed or not. Second, it can generate tests for only

data path (not for control path) faults because it considers a digital circuit as a simple collection of components of one type.

Brahme and Abraham [19] proposed an approach which uses the hierarchical description of a circuit. They reduced the search space by using two strategies. First, components at each level in the hierarchy store the knowledge of propagating and backtracking signals through them. Second, the algorithm performs backtracking and propagation differently depending on the type of the component (data or control) and type of the interconnection. This approach has two drawbacks. First, it must store a large list of faults because the fault model is very general (a fault in a device is represented by a list of input values and the corresponding faulty output values). Second, significant computing overhead will be required for identifying control components and data components and applying different heuristics to different types of components or interconnections.

2.8 High Level HDLs

The high level HDL methods model the circuit under test using the constructs available in the high level language.

Levendal and Menon [20] proposed an approach using hardware description language constructs and functional operations to describe the behavior of general logic networks. They consider both procedural (sequential) and non-procedural (concurrent) interpretations of an HDL description. Their fault model includes data stuck-at faults, control faults, and function faults with user specified faulty behavior. Faults are inserted either in an HDL statement or at an HDL block boundary, and D-

propgation cubes are used in combination with the structure of the circuit to propagate the effect to an observable output. This technique is not very efficient for large circuits because it requires very complex algebraic expressions to derive D-cubes.

Khorram [21] worked on test generation from models described in a procedural HDL. He decomposes the test generation problem into four subproblems.

1. Picking up a statement in a model and describing how to test it.
2. Finding a way to manipulate the primary inputs such that the desired stimulus pattern appears where we really want it. This problem is called *backward propagation*.
3. Activating the desired statement i.e if the execution of a statement is dependent on some conditions, then to *activate* that statement those conditions will have to be satisfied.
4. Making the response on an internal node observable at an external node. This problem is called *forward propagation*.

Backtracking is used to take care of conflicts which may arise while selecting values for variables during *forward propagation* and *backward propagation*. To increase efficiency, Khorram suggests implementing heuristics which guide the search such that the easy alternatives are always tried out first.

Son and Fong [22] described a technique in which functional tables are created from an HDL model. A function 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 s-a- faults, control faults, operation faults and functional faults. Justification and propagation are accomplished with the

aid of the functional tables, which essentially define the present and previous machine states in a form that is easily traversed. No experimental results were reported on the efficiency of this technique.

Gupta and Armstrong [29] proposed a functional fault modeling scheme based on the hardware description language GSP (General Simulation Program). The HDL description is viewed as a sequence of micro-operations and control structures. A fault model based on model perturbation is used. Two types of faults are defined - micro-operation faults and control faults. Micro-operation faults perturb individual micro-operations. Control faults perturb the control points that switch between micro-operation sequences.

Barclay and Armstrong [23] have incorporated the above fault model into a test generation system based on the hardware description language VHDL. A non-procedural subset of VHDL is used, and based on the fault model discussed above four types of faults are defined - stuckthen/stuckelse faults on IF statements, deadclause faults on each clause of the CASE statement, ASSCNTL fault on each assignment statement and MICROOP faults on each micro-operation in the model.

O'Neill [27] has further enhanced Barclay's system by providing a new method for selecting times when goals must be solved and using a depth-first approach to problem solving. This has resulted in an average speed improvement of an order of magnitude over Barclay's algorithm.

Norrod [28], proposed the E-algorithm, a test generation algorithm that also works with chip-level device descriptions written in VHDL. The VHDL description is transformed into a directed graph with the nodes representing data items and data oper-

ations, and arcs indicating dataflow. Given this representation, a modified D-algorithm is employed to generate tests. The algorithm uses the symbols {0,1,X,D, $\overline{D}$,E,$\overline{E}$}. The addition of the symbol E is required to propagate the effect of a fault through control gates. The fault model uses the same microop faults employed by Barclay's algorithm. However, control faults are modeled as stuck-ats on the lines of the graph.

Chapter III

The Hardware Description Language

3.1 Behavioral Data Flow Descriptions

In our approach faults are modeled by perturbing a behavioral description of a digital device. We express this behavior with the constructs of a hardware description language. We have adopted the VHDL (VHSIC Hardware Description Language) [24] as the base language for our system since it contains a very powerful set of constructs for modeling and it is an emerging standard for hardware description languages.

Behavioral descriptions have two basic forms: *algorithmic* and *data flow*. Algorithmic descriptions consist of i) an algorithmic part of sequential code which uses an algorithm to compute values of some variables based on input conditions, and ii) an output statement part of concurrent signal assignment statements which assigns value(s) to output signal(s) based on the value(s) of the variables computed in i). Data flow descriptions contain only concurrent assignment statements. Data flow behavioral descriptions more closely model real circuit behavior than algorithmic de-

scriptions. Therefore we use a form of dataflow models as input descriptions for the test generation algorithm. However,this modified form of dataflow models is different from pure dataflow models in that constructs such as process, if-then-else and case are also supported; also signal assignment statements must be either under a process or within a single block statement.

3.2 VHDL Subset Used

The test method uses a subset of VHDL for model description. The subset includes:

1. signal objects but not variable objects. This is because a signal has a fixed value at a given time whereas the value of a variable depends on the position in the code.
2. signal types BIT and BIT_VECTOR. Other types can be mapped to these. For example, BOOLEAN can be translated to BIT, and INTEGER to BIT_VECTOR.
3. basic operations: logical operators AND, OR, XOR, NOT, and EQV on bits and bit vectors; unsigned arithmetic operators ADD and SUB (implemented as VHDL functions); less than and less than or equal to; concatenation and slices of bit vectors.
4. multiple process statements. For example, for the description shown in Figure 1 on page 3, there are two process statements s1 & s4.
5. concurrent signal assignment statements. For example, for the description shown in Figure 1 on page 3, s7 is a concurrent signal assignment statement.
6. IF and CASE statements.

7. delta delay timing . Since we are considering fixed faults, assignment statements have no "after" clauses or they are ignored.

Note that such high level modeling constructs such as FOR loops and WAIT statements have been excluded. The chosen subset results in behavioral models which allow the fault sensitization and signal propagation requirements of the algorithm to be met more easily than with higher level descriptions. An example of a circuit description using the above subset is shown in Figure 1 on page 3.

3.3 Modeling Sequential Behavior

With the delta delay timing used, sequential behavior can be modeled with the use of either the PROCESS statement or the 'STABLE attribute.

1. A PROCESS statement executes when any one signal in its sensitivity list changes. For example, the statement

   ```
   process(sig1,sig2)
   ```

   executes when there is a change in the value of either sig1 or sig2.
2. The expression "obj'STABLE" is false whenever the value of obj has just changed. For example, the expression

   ```
   CLOCK = '1' and not CLOCK'STABLE
   ```

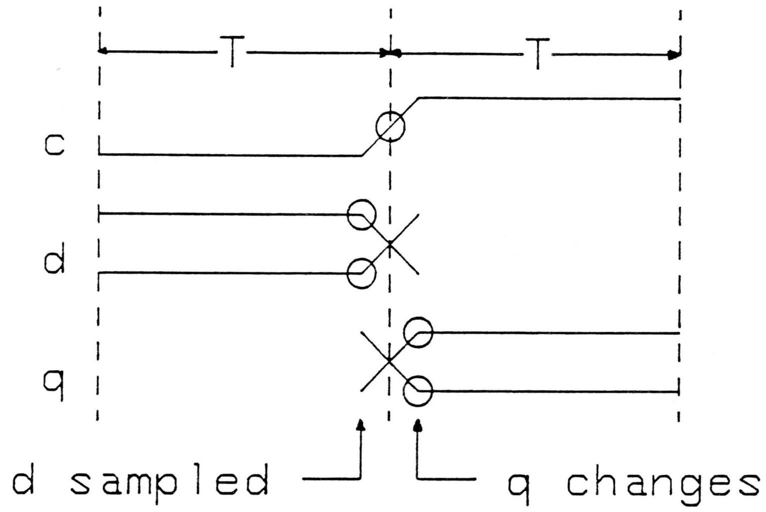
   is true whenever the signal CLOCK makes a 0 to 1 transition.

In both these cases signal values in two adjacent time periods is implied which constitutes sequential behavior.

3.4 Tester Assumption

O'Neill's algorithm assumed a a simple period based tester. Primary inputs are set to values at the beginning of a time period and primary outputs sampled at the end of each period. Clock inputs that require a rising edge require two time periods - one with a 0 immediately followed by a 1. The algorithm ensures that these two time periods are never split, i.e. no events are inserted in the time queue between these two events.

The tester assumption has been modified from that stated above for clock signals. Under the new approach, only one time slice is used. Primary inputs are set to values at the beginning of the time slice and held at that value for that time slice. Primary outputs are sampled at the end of the every time slice. Clock inputs that require a rising edge are assigned a value "R" for that time slice ("F" for falling edges). This tester assumption has been made to facilitate the use of multiple process statements in the input description and to accommodate the improved timing model used by the algorithm. Also, this tester assumption reduces the processing overhead associated with clock signals and hence reduces the runtime of the algorithm.
As an example, Figure 2 on page 23 shows some waveforms for a D flip-flop. The tester assumption under the old and new approaches is shown by the tabular representation below.



d	c	q
—	—	—
1	0	—
—	1	1

Old Timing

d	c	q
—	—	—
1	R	1

New Timing

Figure 2. Tester Assumption

Chapter IV

Behavioral Fault Model

As mentioned in Chapter II, the purpose of testing is to ensure that a fabricated IC is free of *faults*. As it would be impossible to check for each and every fault at the silicon or transistor level, the IC is viewed at a higher level of abstraction. Traditionally this has been the gate level. However, looking at a circuit of this level of abstraction forces some loss of information and consequently not all physical defects in silicon can be mapped as gate level faults. However, generating stimulus to verify device functionality is much simpler at the gate level than at the transistor level. Thus, as we move up the hierarchy of design representation, the number of physical defects that get modeled decrease but the ease with which tests can be generated increases.

Classical fault modeling has been tied to physical defects. Consequently it has been dependent on the type of technology used. For example, in TTL circuits, common failure modes can be modeled as stuck-at faults and so the stuck-at fault model has been effective for TTL circuits. However, in MOS circuits, some failure modes cannot be modeled as stuck-at faults. Thus, a different model, the stuck-open fault model has

been proposed for MOS circuits. Although, these models have been proved effective for systems built from MSI devices, they have not been very effective for systems built from VLSI devices.

Thus on the basis of the preceding paragraphs, it can be concluded that for efficiently generating tests for VLSI devices :

1. The circuit under test must be viewed at a higher level of abstraction.
2. A fault model independent of the technology and implementation must be used for this higher level of abstraction.

With the above goals in mind, a new approach to fault modeling based on model perturbation is adopted. Here a systematic way is developed to perturb the device model. The list of such perturbations becomes the fault list. A test is found for each fault in the list. The stuck-at fault model can be considered to be a form of model perturbation. As an example, given a 2-input AND gate we could define six perturbed models with stuck-ats on inputs and outputs. An important issue for model perturbation is to define a reasonable subset of of the perturbations so that test vectors for these perturbations thoroughly exercise the logic of the circuit. The test generation algorithm discussed in this thesis uses a form of model perturbation for behavioral models.

In behavioral descriptions, a model of a digital device is constructed as a single logical entity [31]. Its basic structure is that of a sequence of micro-operations expressed in a hardware description language. Micro-operations are statements that perform some transformation on data. Control structures govern the flow of execution between sequences of micro-operations depending upon the value in the control ex-

pression. Based on this view, two types of faults are defined : micro-operation faults and control faults. *Micro-operation faults* perturb individual micro-operations. *Control faults* perturb the control points that switch control between micro-operation sequences. Such an HDL-based fault model was first proposed in [29].

We discuss control faults first. Control is achieved by means of VHDL constructs such as PROCESS, IF-THEN-ELSE or CASE. The IF construct is faulted by assuming it to be stuck such that branching always occurs in one direction independent of control expression values. Thus we have two fault cases corresponding to an IF statement - *stuck-then* and *stuck-else* faults. CASE statements are faulted by assuming that a clause selected by the CASE statement does not execute. Thus for each clause under a CASE statement a DEADCLAUSE fault is defined. A third type of control fault is an *assignment* fault, i.e. it models the effect of a single register assignment not taking place. A fourth type is the *dead process* fault, i.e., it models the effect of a VHDL process being insensitive to the inputs that normally trigger it.

A micro-operation fault is achieved as a micro-operation such as AND, OR, XOR, ADD, and SUB fails to some other micro-operation. A difficult question to answer is: To which micro-operation should we perturb? In [23] and [29], we faulted a micro-operation by replacing it with its "logical dual" (e.g., AND by OR, OR by AND, and XOR by equivalence). Chao and Gray [30] have shown that perturbing a logic micro-operation to its "logical dual" gives consistently high equivalent gate level fault coverage. For arithmetic operations, we used a similar heuristic, e.g. replacing ADD by SUB.

To prove the effectiveness of the fault model, a fault coverage experiment on eleven logic circuits representing a cross-section of generic types of logic was carried out at Virginia Tech. For each circuit

1. A behavioral model for the circuit was developed.
2. A complete set of behavioral model faults was determined manually.
3. A test set that detected the complete behavioral model fault set was developed.
4. This test set was fault graded with HILO [32] using a gate-level model of the circuit to determine what percentage of stuck-at faults were covered.
5. For the same gate-level fault coverage the lengths of the behavioral model test set and pseudo-randomly generated test set were compared.

92.4% (average) fault coverage was obtained for the circuit models using the behavioral test generation method. However, the behavioral test generator discussed in this thesis was not used for this purpose. Instead, the tests were generated manually. The result of the experiments with the behavioral test generator are discussed in the Results section.

Chapter V

Previous Work

As stated in the introduction, a chip-level test generation algorithm was recently developed by Barclay [23]. The concept of goal trees is used to organize the algorithm structure. With this approach, each node in the tree represents a step or a goal in the development of a test vector. The process of developing and simplifying the goal tree involves breaking goals down into subgoals. The process is complete when all goals are broken down into *primitive goals* such as specifying circuit input conditions or observing a circuit output.

The following *low-level goal types* are defined and used during the process of solving for test vectors:

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 operations of value justification and fault effect propagation are solved using these goal types. The steps involved in the algorithm are now described.

First, fault cases are identified based on the VHDL description and the fault model discussed above. A set of basic goals for each fault case is specified. The solving process then works to find a set of input values that realize these initial goals. The algorithm maintains a list of all subgoals remaining to be solved. The list is sorted according to a set of heuristics, which were determined by observing which types of goals are best solved immediately, and which are best left until others have been solved. The order in which goals are solved is critical to minimize the number of conflicts generated during the solving process. When conflicts do occur during solving, backtracking to find alternate solutions takes place. Some attempt is made to order the alternative solutions for each goal type. This ordering is determined by controllability and observability measures, as well as other heuristic guidance.

When the set of initial goals for a test is solved in terms of primitive goals, the test vector may be extracted from the goal tree. Note that during the solving process, a set of constraints on time periods is established through TR goals. The TR goals

specify relative constraints, not absolute time references. A compatible set of solved goals must be extracted, where compatibility implies that all time relations are satisfied, and no conflicting assignments are attempted. Incompatibilities are resolved by backtracking.

While Barclay's algorithm successfully found tests for faults in circuits of various types, the speed performance of the algorithm was not satisfactory. O'Neill [27] identified two main factors which affect the speed performance of the method: the level at which the goal tree is handled, and the manner in which time periods are assigned.

To overcome the first problem, O'Neill defines the following *high-level goal types*:

JUSTIFICATION :- Given an expression, 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 that expression. An initial justification goal can require solving several justification and execution subgoals. These subgoals are satisfied in a depth-first fashion, and the order in which they are attacked is completely defined by the justification rules formed during a preprocessing stage.

PROPAGATION :- Given a good/bad value pair, 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. Justification routines are called to justify values required in other objects in the expressions of the path such that the good/bad result will be propagated.

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 and justification routines are called to place the required value in that expression.

The basic goals for each fault case are specified in terms of these operations, rather than expressing them as a set of lower level goals. Consequently, the goal tree for this approach is much less complex than that with the use of low level goals it is ordered in a depth-first fashion rather than a best-first fashion.

To overcome the second problem, O'Neill handles time explicitly. As each operation is performed, it is assigned an actual time period in which it will occur. To facilitate the resolution of conflicts which may arise during this process, operations have the ability to split two time periods apart and create new time periods between them. The combined effect of using a depth-first approach with high level goal types and handling time in absolute terms gives O'Neill's algorithm a speed advantage of 10 to 100 (depending upon the type of circuit) over the low level approach.

Problems

O'Neill's performed well over Barclay's algorithm but it suffered from the following problems which resulted in wrong and/or inefficient tests in some cases.

The first factor is the way in which time is handled. As discussed before, time is handled in absolute terms during the goal solving process. An event queue is maintained, and events are inserted into the time queue as soon as they arise. Although the algorithm takes care of conflicting assignments, the order in which events should

occur is not always correct. This results from inserting events at the wrong points in the time queue.

The second problem is with STUCK faults. For STUCKTHEN/STUCKELSE faults with no matching destination objects under the good and bad clauses, an assignment statement under the good clause (or bad clause) is selected and good/bad values are propagated through its destination object. The algorithm yields wrong results if there are no other statements assigning values to that object.

The third problem is with the execution goal. Execution of a statement requires i)determining a list of values required in the various control expressions governing the execution of that statement, and ii)justifying values in the respective control expressions. It is important to note that the list of values is fixed for a given statement. O'Neill's algorithm determines this list of values every time a statement is to be executed which for deeply imbedded statements results in considerable time penalty.

The fourth problem is with the extraction of justification rules from the VHDL model during a preprocessing step. For each assignment statement in the model, a rule is extracted which details the conditions under which that statement can be used to perform value justification. During the goal solving process, whenever a value is to be justified in an object, the first justification rule for that object is selected and justification attempted through that statement. If it fails, the next justification rule (corresponding to another statement assigning values to that object) is selected and justification attempted through it. Actually, a set of three justification rules have to be stored for each assignment statement for the following three possible values which may have to be justified in the destination object of that assignment statement - i)rising edge, ii)falling edge, and iii)some fixed value. Thus for a model having n

assignment statements, 3n justification rules need to be stored in the database. This may not be desirable for big models due to the large number of justification rules that will have to be stored in the database.

The final factor concerns the lack of forward implication. O'Neill's algorithm does not use forward implication at all resulting in inefficient tests (and incorrect tests) in some cases.

Chapter VI

New Method

This chapter discusses approaches used to overcome the problems mentioned in the previous section, and discusses the test generation algorithm which makes use of these new approaches.

6.1 Improved Timing Model

An improved timing model, similar to the one presented in RTG [5] is used. This timing model assumes a tester described in Chapter III.

The timing model is described with the aid of an example. Consider the circuit shown in Figure 1 on page 3 redrawn in Figure 3 on page 35. Suppose we want to generate a test for terminal q1 stuck-at-0. The algorithm starts the test generation at time slice t1, and requires q1="1" at t1. Since q1 is the output terminal of a sequential device, a new time slice t2 is created before time slice t1, and the algorithm tries to justify

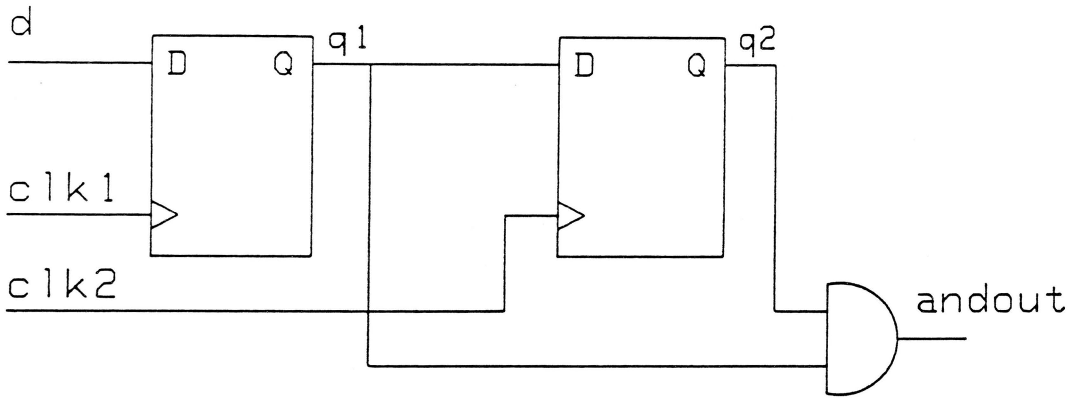


Figure 3. New Timing Model

q1="1" during time slice t2. Using justification and execution operations it generates the requirements d="1" and clk1="R" at time t2. Since these are primary inputs no further justification is required. Having sensitized the fault, the next step is to propagate the effect of the fault to an observable output. The value on q1 can be propagated to an output pin along two paths. Using simple observability measures the algorithm selects the direct path through the AND gate. Since under the new timing, value was latched into q1 at the end of time slice t2, propagation can continue in a time slice after t2. Propagation of the fault syndrome through the AND gate requires q2="1" in a time slice after t2. Since q2 is the output of a sequential device, a new time slice t3 is created after t2, and an attempt made to justify q2="1" during t3. q1 being set to "1" at t2, justification and execution operations generate the requirement clk2="R" at time t3.

The plot at the bottom of Figure 3 on page 35 identifies the order in which the algorithm generates test vectors and the order in which these test vectors must be applied to detect the fault. It may be noted that time slice t1 is just a reference time and that the tester does not input values or observe outputs during this time slice. The tester inputs and observes values during time slices t2 and t3 only; time slice t1 being a reference time used by the algorithm.

6.2 Two phase tests

A generalized two-phase approach to testing is adopted under the new approach. As discussed in Chapter IV, the fault model defines two types of faults - micro-operation faults and control faults. A test for a micro-operation fault is found by ex-

amining the assignment statement for data-transfer; a test for a control fault is found by picking up an assignment statement under that control structure and observing it for data-transfer. Thus, in either case an assignment statement is used to check for data-transfer.

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

If the fault site is traversed during preloading, the transfer of data will not occur if the fault is present and so the transfer must be verified by propagating the "bad_val"/dummy (dummy is any value other than "bad_val") pair to an observable output. Thus, if a value other than the "bad_val" is observed by the tester at this point, it can be concluded that the transfer of data has not taken place and we don't need to proceed any further. However, even if the "bad_val" is observed, transfer of data cannot be assumed since the "bad_val" observed may be due to the initial charge on that object. Thus, it is necessary to place another value "good_val" into that object

and propagate the "good_val"/"bad_val" to an observable output to make sure that the transfer of data actually takes place. Thus, testing for a fault involves two steps - preloading and then checking for data transfer and so it is called a *two phase test.*

6.3 Extension to Multiple Process Statements

The algorithm has been extended to handle multiple process statements and concurrent signal assignment statements.

A process executes when one of the signals in its sensitivity list changes. To execute a process statement at a given time, the algorithm checks if the value of any signal in its sensitivity list has been changed by the justification operations involved in executing the assignment statement under that process. If it has the search is stopped, otherwise the algorithm forces a change in the value of one of the signals (in the sensitivity list). This requires loading the selected signal in the previous time period with a value different from the desired value. A concurrent signal assignment statement in a block is treated as a process with a single assignment statement. A concurrent signal assignment statement gets triggered by a change in the value of any one of the signals that make up the source expression of that signal assignment statement. Thus, to achieve this, the objects involved in the source expression of such a statement are determined during a preprocessing step. This set of objects becomes the sensitivity list for the implied process statement.

6.4 Solving VIE goals during preprocessing

To overcome the problem of solving VIE goals during goal solving (mentioned in the previous section), the new algorithm determines the list of values required in control expressions governing the execution of every statement during a preprocessing step and stores them in the database. Thus, this list has not to be determined every time a statement is executed as in the previous method. Each time a statement is to be executed, the list corresponding to that statement is retrieved from the database and values justified in the respective control expressions. For each statement in the model just one Prolog fact is stored and so the storage requirement is not too high. Thus the new approach trades off a slight increase in storage for significantly reduced execution time. Since the execution goal is used very often by the algorithm, storing this information results in a considerable speedup.

6.5 Eliminating storage of justification rules

To overcome the problem of storing justification rules (mentioned in the previous section), justification rules for assignment statements are not stored in the database. Instead only three justification rules, corresponding to justifying i) rising edge, ii) falling edge, and iii) stable value are defined and stored in the database. During the goal solving process, whenever a value is to be justified in an object, the desired value, object name and time are passed to one of the justification rules. This justification rule picks up statements assigning values to that object, one at a time, till the justification is successful. This process of selecting statements is guided by the controllability of the statements and it is left to the Prolog backtracking mechanism

to select alternative choices when one fails. Here the new approach trades off a slight increase in execution time for considerable savings in storage.

6.6 Increased Forward Implication

One of the factors that can greatly affect the efficiency of a test generation algorithm is forward implication. By forward implication we mean determining the effects of an assignment of a value to a signal. Forward implication helps detection of conflicts more easily, reducing backtracking and hence reducing the test generation time.

O'Neill's algorithm used partial forward implication. Substituting decided object values in objects is partial forward implication. Under the new approach, forward implication has been extended to determine what other statements get executed as a result of one statement getting executed. Every time a control statement is executed its implications are determined. The statements are not evaluated but simply time-tagged. During the justification operation involved in justifying a value in an object, a check is made to see if the object has received an *implied* value at a time previous to the desired time. If so, the time at which it was implied is retrieved and the statement evaluated at that time. If the value is same as the desired value no further justification is necessary.

6.7 User Assists to the Test Generation algorithm

The test generation algorithm views the test generation problem as a search problem over the HDL description of the circuit under test. At each node of this search space a choice needs to be made from the several available alternatives. The algorithm uses some built-in heuristics to perform this search. However, in some cases these prove to be ineffective and result in no test being generated where one does exist. This results from the algorithm having limited knowledge of the high level operation of the device, e.g the algorithm does not know how to easily drive a circuit from one state to another. To overcome this problem, the algorithm has a user interface which can help to guide the test generation algorithm to make decision at critical nodes in the search space. Thus, with user assistance more effective tests can be generated and in some cases they can be generated more easily. User assistance can prove useful in the following areas:

1. All tests require a known "good_val" and a known "bad_val". For objects that represent bit vectors, the test generator left to its own would select a value pair for "good_val"/"bad_val". However, these values may not be best suited for that circuit. For example, if the value pair 0000/1111 is selected for an object in a 4 - bit counter circuit, the counter may be forced to go through all its states resulting in a long test vector sequence. This may not be desirable for complex sequential circuits having a large number of states.
2. In some behavioral descriptions, "good_val"/"bad_val" may stand for state names. Again the BTG left to itself would take values which may correspond to invalid states and the algorithm having no knowledge of this would unsuccessfully attempt to generate tests using these values.

3. For micro-operations that represent complex circuits, there could be a large number of input combinations that result in a desired value from that micro-operation. As an example, consider a comparator with two n-bit inputs A and B. To justify a value of 1(0) at the output, there are a large number of combinations for values of A and B. The algorithm could require considerable backtracking before the correct set of values are found for A and B.
4. The algorithm uses simple controllability/observability measures to choose paths for propagation and justification. However, these may not be always the best suited for that circuit. This could again lead to considerable backtracking before the correct path for propagation (or justification) is found. In such cases, the user could guide the algorithm in selecting paths at crucial nodes in the circuit and thus get tests for which the test generator wouldn't.

Thus, in the above cases user assistance can result in more effective tests.

6.8 The Test Generation System

The test generation algorithm which uses the above discussed approaches is now described. Although the VHDL to Prolog Translator is not part of my thesis, it is discussed here to make the description complete.

6.8.1 Preprocessor

6.8.1.1 VHDL to Prolog Conversion

Since the test generation algorithm is written in Prolog, the information contained in the VHDL model is first translated into a set of Prolog facts. This translation is accomplished by a program written in C.

For example, consider the circuit and its VHDL behavioral description shown in Figure 1 on page 3. the Prolog representation obtained with the preprocessor for this description is shown in Figure 4 on page 44.

6.8.1.2 Pre-processing the Prolog representation

Given the Prolog representation of the circuit, a Prolog program then extracts information which is used later by the fault list extractor and the test generator. To begin with, the Prolog representation is scanned for basic information such as statement types, object names, clauses under each control statement and assignment statements under them, etc. This information is used by the pre-processor itself to form Prolog facts that represent the knowledge of the circuit. Specifically, this includes information on selecting paths for justification, propagation and execution.

The controllability[1] and observability[2] of each signal in the model is measured. These measures are used to select paths for justification and propagation. The observability of a signal is measured by counting the number of statements which must be exe-

---

1 the ability to set a signal in a circuit to a particular value via the primary inputs of the circuit.

2 the ability to observe the response of a signal in a circuit at a primary output.

```
datatype(d,bit). inputpin(d).
datatype(clk1,bit). inputpin(clk1).
datatype(clk2,bit). inputpin(clk2).
datatype(q1,bit).
datatype(q2,bit).
datatype(andout,bit). outputpin(andout).

statementtype(s2,if).
controlexpression(s2,[biteqv,[obj,clk1],[lit,[bit,"1"]]]).
subordinaterange(s2,then,[s3]).
subordinaterange(s2,else,[]).

statementtype(s3,assignment).
sourceexpression(s3,[obj,d]).
destinationobject(s3,q1).

statementtype(s4,process).
controlexpression(s4,[bitnot,[stable,[obj,clk2]]]).
subordinaterange(s4,then,[s5]).
subordinaterange(s4,else,[]).

statementtype(s5,if).
controlexpression(s5,[biteqv,[obj,clk2],[lit,[bit,"1"]]]).
subordinaterange(s5,then,[s6]).
subordinaterange(s5,else,[]).

statementtype(s6,assignment).
sourceexpression(s6,[obj,q1]).
destinationobject(s6,q2).

statementtype(s7,assignment).
sourceexpression(s7,[bitand,[obj,q1],[obj,q2]]).
destinationobject(s7,andout).
```

Figure 4. Prolog representation of Figure 1

cuted in order to observe the signal. Each statement which must be executed is assigned a weight based on the estimated difficulty of actually executing the statement. The controllability of a signal is determined by the position of its source statement in the model. Before starting the test for a fault the user can change the controllability/observability sequence of any signal from the default sequence.

Next, for each statement in the model, the sequence of control statements (if any) which govern the execution of that statement is determined. Also, the values required in their control expressions is determined. Referring to the model in Figure 1 on page 3, execution of s3 requires a change in the value of CLK1; also the value in CLK1 at that time must be "1". The Prolog fact that would be stored in the database is

```
made_vies(s3,[s1,s2],[[bit,"1"],[biteqv,[obj,clk1],[lit,[bit,"1"]]]],
                     [[bit,"1"],[bitnot,[stable,[obj,clk1]]]]).
```

6.8.2 Formation of the fault list

Based on the fault model discussed in the previous section, the following fault types are employed. Each type may have a number of cases.

- MICRO-OP : A micro-operation fails to some other operation.
- ASSCNTL : An assignment statement fails to execute.
- IFCNTL : An IF statement fails to a stuck-at-THEN (STUCKTHEN) or stuck-at-ELSE (STUCKELSE).
- CASECNTL : A WHEN clause in a CASE statement fails to execute when it is selected (DEADCLAUSE).
- PROCCNTL : A process fails to execute when it is selected (DEADPROCESS).

- STUCKDATA : Any signal stuck-at-0 or stuck-at-1. Although, not a true behavioral fault, it is included to improve fault coverage along datapaths.

As an example, consider the model in Figure 1 on page 3. The fault list that would be generated in this case is given below.

```
     TYPES                                        CASES

 PROCCNTL  s1                                 deadprocess, s1
 IFCNTL    s2                                 stuckthen,s2
                                              stuckelse,s2
 MICROOP   s2 - (=)
 ASSNCNTL  s3
 PROCCNTL  s4                                 deadprocess, s4
 IFCNTL    s5                                 stuckthen,s5
                                              stuckelse,s5
 MICROOP   s5 - (=)
 ASSNCNTL  s6
 ASSNCNTL  s7
 MICROOP   s7 - (and)
```

6.8.3 Test Generation Algorithm

6.8.3.1 Problem Solving Method

The approach adopted is an equivalent of the D-algorithm [1] for behavioral descriptions. Given a faulty micro-operation or control operation, a test is developed by (1) activating the faulted operation (fault sensitization) and (2) propagating the effect of the fault through other statements to a statement that will transfer the effect to an output (fault propagation). With both these operations, justifications to primary inputs is required. The algorithm can perform this process for descriptions that represent sequential as well as combinational circuits.

The concept of goal trees is used to organize the algorithm structure. With this approach, each node in the tree represents a step or a goal in the development of a test vector. The process of developing and simplifying the goal tree involves breaking goals down into subgoals. The process is complete when all goals are broken down into "primitive goals" such as specifying circuit input conditions or observing a circuit output. The following three goals - justification, execution and propagation are used to simplify the goal tree in a depth-first fashion.

A flow diagram for the test generation algorithm is shown in Figure 5 on page 48. Generating a test for a fault requires - fault sensitization and fault propagation. The fault is sensitized by forcing the faulted operation to be executed. This may require values to be justified in certain objects. Also certain statements will have to be executed; execution of statements will further require values to be justified in certain objects. Thus, fault sensitization uses the justification and execution goals recursively till values are justified to the primary inputs. Having sensitized the fault, the next step is to propagate the fault to an observable output. This requires the use of the propagation goal. To propagate the fault, values will have to be justified in certain objects and some statements will have to be executed. Thus, fault propagation uses the propagation, justification and execution goals recursively till the fault is propagated to a primary output.

6.8.3.2 High Level Goals

JUSTIFICATION: Given an expression, and a desired value at a given time, the justification operation works backwards to determine the set of external input values which will result in placing the value in that expression.

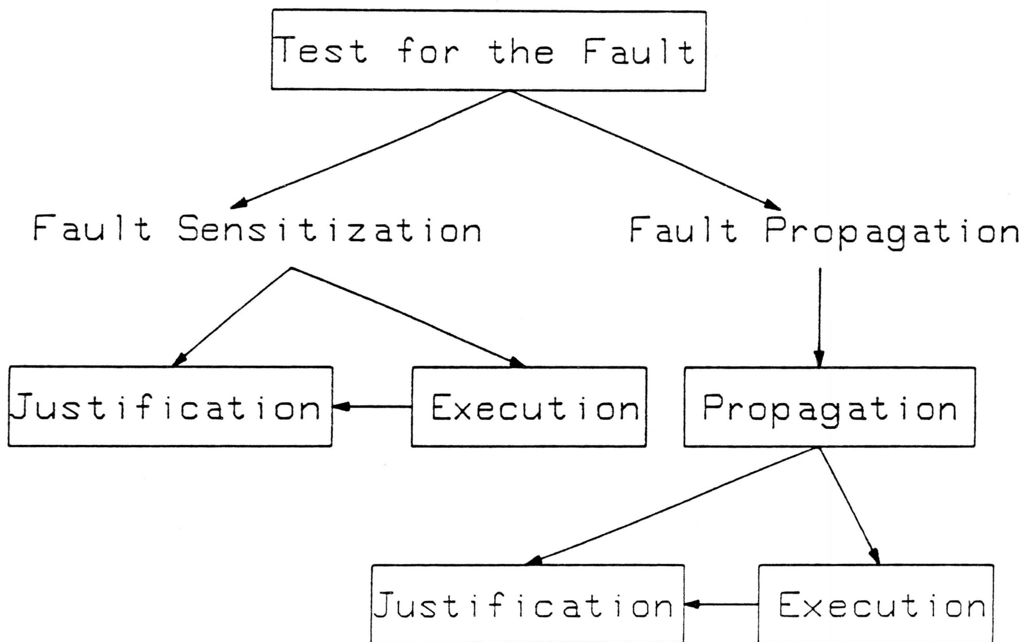


Figure 5. Flow Diagram of the Test Generation Algorithm

The justification operation first evaluates the expression at the given time using the latest values stored in the objects (including any 'implied_vios' as discussed under EXECUTION) of that expression. If this value is same as the desired value no further justification is necessary; otherwise the justification operation determines a set of values for the objects involved in the expression such that the desired value results 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.

For single objects, if a value has been loaded into an object at the given time, a check is made to see whether the existing value and desired values are combinable (e.g. "xxx1" and "xx0x" are combinable). If they are, the justification operation for that object is complete; otherwise the *justification goal fails*.

If a value has not been loaded into that object at the given time, the latest value stored in that object is determined. Again, if the stored value and desired value are combinable, the justification operation is complete, otherwise the *justification rule* is used to accomplish the justification. There are two sets of *justification rules* stored in the database - one for non-inputs and the other for inputs. The rule for non-inputs is as follows:

```
do_noninputs(Object,Value,Time) :-
   destinationobject(Stmt,Object),
   sourceexpression(Stmt,Exp),
   justify(Value,Exp,Time),
   execute(Stmt,Time),
   dont_disturb(Object,Time).
```

This rule picks up an assignment statement whose destination object is the object in question; justifies the desired value in its source expression; executes all control statements which govern the execution of that assignment statement and ensures that the value in that object does not get disturbed at that time. Again, there will be more than one statement assigning values to that object; the alternate choices are considered when backtracking occurs and the order in which these choices are considered is determined by the controllability of the assignment statements.

For inputs, the justification rule is trivial; if there are no conflicts at that time, the requirements simply become a part of the test vector.

For justifications requiring changes in values of objects (as for objects in the sensitivity list of a VHDL process statement), the object name, desired value and the given time are passed to a different justification rule. This rule first checks the value of the object in the previous time period. Different values in the two time periods indicate a change and the justification is complete; otherwise a change of value is forced by preloading a value (different from current value) in that object in the previous time period.

When a justification goal has been solved completely, the result is a set of input values (possibly in different time periods) which, when applied to the device, will cause the given value to be placed in the desired expression at the given time.

```
Example
           :
s3   : OBJ_1 <= SOURCE_EXP_1;
           :
s6   : OBJ_2 <= SOURCE_EXP_2;
           :
s10  : DEST <= OBJ_1 xor OBJ_2;
           :
```

Shown above are fragments of a VHDL code where OBJ_1, OBJ_2 and DEST are signals. A typical justification goal which would be encountered by the algorithm is

```
?- justify(s10, [bit,"1"], [xor, [obj,OBJ_1], [obj,OBJ_2]], t1).
```

Assuming no values loaded into objects OBJ_1 and OBJ_2, this would lead to the following justifications

```
        justify(s3, [bit,"0"], SOURCE_EXP_1, t1).
and     justify(s6, [bit,"1"], SOURCE_EXP_2, t1).
```

The other set of values {[bit,"1"], [bit,"0"]} for {OBJ_1,OBJ_2} would be considered by the algorithm on backtracking. This process of justifying values is continued till values have been justified to the primary inputs of the model.

EXECUTION: Given a statement and a time, the execution operation places the required values into the control statements (if any) which govern the execution of the given statement.

As discussed before, during the preprocessing step, for each statement in the model, the sequence of control statements and the values required in their control expressions are determined and stored in the form of Prolog facts.

During the runtime of the algorithm, whenever a statement has to be executed, the Prolog fact for that statement is retrieved from the database and the desired values justified in each control expression of that fact. Thus, when all values for control expressions have been justified to the external inputs, the result is a sequence of input values (possibly in different time periods) which will cause the given statement to execute.

Example

```
s1 : process (OBJ)
s2 : if (OBJ = '1') then
s3 :    DEST_1 <= SOURCE_EXP_1;
s4 :    DEST_2 <= SOURCE_EXP_2;
     else
s5 :    DEST_3 <= SOURCE_EXP_3;
s6 :    DEST_4 <= SOURCE_EXP_4;
     end if;
     end process;
```

Shown above is a short VHDL fragment. A typical execution goal that would be encountered by the algorithm is

```
?- execute(s5, t1).
```

As can be seen from the VHDL model, execution of s5 requires a change in the value of object OBJ at time t1; also the value in OBJ at the end of time slice t1 must be "0". The Prolog fact that would be retrieved from the database would be

```
made_vies(s5, [bit,"0"], [biteqv, [obj,OBJ], [lit, [bit,"1"]]],
              [bit,"1"], [bitnot, [stable, [obj,OBJ]]]).
```

The execution operation would then perform the following justifications

```
     justify(s2, [bit,"0"], [biteqv, [obj,OBJ], [lit,[bit,"1"]]], t1).
and  justify(s1, [bit,"1"], [bitnot, [stable, [obj,OBJ]]], t1).
```

After a value is justified in each control expression, its implications are determined. By implications we mean the assignment statements which get executed as a result of justifying a value in a control expression. With reference to the above example, statement s6 always executes when statement s5 executes (and vice-versa). Thus a note of this 'implied_vio' (implied value in object) is made in the database. The next time when a value is to be justified in object DEST_4, SOURCE_EXP_4 is evaluated at time t1; if the result of evaluation is the desired value no further justification is

necessary, otherwise the justification goal is called to justify the desired value in object DEST_4.

PROPAGATION: Given a good/bad value pair, and the 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 following rules are used to propagate the fault syndrome from its initial location to an external output :

PROPAGATION THROUGH CONTROL CONSTRUCTS The following cases may arise when propagating fault syndromes through control constructs :

1. Matching destination objects exist under good[3] and bad clauses[4]

   The fault syndrome is transferred from the faulty control construct to an assignment statement under the good clause. Preloading is achieved through the assignment statement under the bad clause having the same destination object. As an example, consider the VHDL fragments shown below. For a 0/1[5] value pair being propagated by the if statement s10, "bad_val" is preloaded through s15 and "good_val" is loaded through s18.

```
                :
s10 : if (EXPR) then          For a 0/1 value pair propagated by s10
                :                "bad_val" loaded through s15
s15 :      DEST <= ...           "good_val" loaded through s18
                :             "bad_val"/"good_val" propagated further
         else
                :
s18 :      DEST <= ...
                :
```

---

[3] good clause = clause that gets executed in the fault free circuit.

[4] bad clause = clause that gets executed in the faulty circuit.

[5] 0 = value in the good circuit, 1 = value in the faulty circuit

2. Assignment statements exist under the good clause but no assignment statement exists under the bad clause having the same destination object :
   The fault syndrome is transferred from the faulty control construct to this assignment statement existing under the good clause. The destination object of the selected assignment statement is preloaded with a "bad_val" in one of the following three ways:

   a. The destination object has been previously loaded with some value. In this case, this value is taken as "bad_val".

   b. The destination object is preloaded with a "bad_val" using an assignment statement under the faulty control construct. In this case, the effect of preloading must be propagated to an external output.

```
Example
              :
s10 : if (EXPR) then        For a 1/0 value pair propagated by s10
              :               "bad_val" preloaded through s15
s15 :     DEST <= ...         "bad_val"/dummy propagated to external output
              :                       and then
      end if;                 "good_val" loaded through s15
              :               "good_val"/"bad_val" propagated further
```

   c. The destination object is preloaded with a "bad_val" using an assignment statement which is not under the faulty control construct. In this case no propagation to an external output is required.

```
Example
              :
s5  : DEST <= ...
              :
s10 : if (EXPR) then            For a 1/0 value pair propagated by s10
              :                 "bad_val" preloaded through s5
s15 :     DEST <= ...           "good_val" loaded through s15
              :                 "good_val"/"bad_val" propagated further
      end if;
              :
```

Having preloaded the destination object one way or the other, the next step is to place the "good_val" (different from "bad_val") into the destination object of the selected assignment statement and execute the assignment statement.

3. Assignment statements exist under the bad clause but no assignment statement exists under the good clause having the same destination object :
   The fault syndrome is transferred from the faulty control construct to this assignment statement existing under the bad clause. The destination object of the selected assignment statement is preloaded with a "good_val" in one of the following three ways:

   a. The destination object has been previously loaded with some value. In this case, this value is taken as "good_val".
   b. The destination object is preloaded with a "good_val" using an assignment statement under the faulty control construct. In this case, the effect of preloading must be propagated to an external output.

```
Example
              :
s10 : if (EXPR) then      For a 0/1 value pair propagated by s10
              :             "good_val" preloaded through s15
s15 :       DEST <= ...    dummy/"good_val" propagated to external output
              :                      and then
        end if;             "bad_val" loaded through s15
              :             "good_val"/"bad_val" propagated further
```

   c. The destination object is preloaded with a "good_val" using an assignment statement which is not under the faulty control construct. In this case no propagation to an external output is required.

```
Example
            :
s5   : DEST <= ...
            :
s10 : if (EXPR) then            For a 0/1 value pair propagated by s10
            :                   "good_val" preloaded through s5
s15 :       DEST <= ...         "bad_val" loaded through s15
            :                   "good_val"/"bad_val" propagated further
        end if;
            :
```

Having preloaded the destination object one way or the other, the a "bad_val" is justified in the source expression of the selected assignment statement and input conditions set up such that the statement does not get executed. Thus "good_val" will be observed in the destination object in the good circuit and "bad_val" in the faulty circuit.

Having set up "good_val"/"bad_val" value pair in the destination object, further propagation of the fault syndrome may be necessary if the destination object is not an output pin.

PROPAGATION THROUGH DATA PATHS : Propagation of fault syndromes through data paths is achieved as in the the D-algorithm.

1. DATA PATHS IN ASSIGNMENT STATEMENTS :

Given a good/bad value pair in an object in an assignment statement, values are justified in other objects of that statement and a good/bad value pair formed for the destination object of the assignment statement. A statement (which uses that destination object) is then chosen to propagate the fault syndrome further. Often there will be more than one statements through which this can be done; the al-

ternate choices are considered when backtracking occurs and the order in which they are used is determined by the observability of those statements.

Example

```
        :                        0/1 value pair on OBJ_1 transferred
DEST <= OBJ_1 xor OBJ_2;         as 1/0 value pair on OUT by
        :                        justifying OBJ_2 = '1' and OBJ_3 = '0'
OUT <=  DEST or OBJ_3;
        :
```

2. DATA PATHS IN CONTROL STATEMENTS :

Given a good/bad value pair in an object of a control statement, values are justified in other objects of that statement and a good/bad value pair formed for the control expression of the control statement. This fault syndrome is further propagated through the control expression using the rules listed above.

Example

```
    :
if (OBJ_1 and OBJ_2) then        0/1 value pair on OBJ_1 transferred
    :                            as 0/1 value pair on the control
    :                            expression by justifying OBJ_2 = '1'
```

6.8.3.3 Test requirements for different Fault Models

This section discusses the strategies adopted in generating test vectors for the different fault models described above.

ASSCNTL: In order to determine if an assignment statement accomplishes its data transfer when executed, a "good_val" must be justified in the source expression of the statement and the statement must then be forced to execute. The destination object of the assignment statement may then be observed to see if the transfer of data occurred. If the transfer is successful, the value justified in the source ex-

pression will be observed. However, if the transfer of data did not take place, the value of the object will be undefined, as we do not know the "old" value of the object. Thus, to allow detection of the ASSCNTL fault, a "bad_val" (different from the "good_val") must be *preloaded* in the destination object of the faulty assignment statement. Then, should the transfer fail to occur, the tester will know what value to look for as an indication of the faulty condition.

Thus the requirements of an ASSCNTL test are :-

1. Preload the destination object with a known "bad_val". The faulty assignment statement is avoided to accomplish the preload. This is achieved by placing the justification rule (formed during preprocessing) for the faulty assignment statement at the bottom of the database, so that it is used for preloading only if it is impossible to preload through any other assignment statement in the model. However, if the faulty assignment statement must be used for preloading, the transfer of data will not occur if the fault is present and so the transfer must be verified by propagating the "bad_val"/dummy value pair from the assignment statement to an observable output.
2. Place the "good_val" in the source expression of the faulty assignment statement and justify it.
3. Execute the faulty assignment statement.
4. Propagate the "good_val"/"bad_val" value pair to an observable output.

For assignment statements implying feedback within them

e.g. COUNT < = f(COUNT)

justifying the "good_val" in object COUNT might require justifying values in COUNT more than once (as opposed to only once in a simple register transfer). This may destroy the "bad_val" preloaded in object COUNT and defeat the purpose of preloading. To overcome this problem, the justification rule for the faulty assignment statement is relocated at the top of the database after the preload operation so that it is forced to be always used for justifying values in object COUNT.

STUCKTHEN: A test for a STUCKTHEN fault attempts to execute the ELSE clause of the IF statement, and then finds a way to observe whether the ELSE clause is executed (in the good case) or whether the THEN clause executes (in the faulty case). The THEN clause of an IF statement executes when the control expression of that IF statement is true (ELSE clause executes when the control expression is false). The algorithm uses boolean values 1/0 instead of true/false and so a STUCKTHEN fault condition actually behaves as if the control expression of the IF statement were stuck at a value of 1. Thus, in order to to accomplish a test, all that is required is to propagate the effects of 0/1 value pair from that IF statement to an observable output. The approaches mentioned in the previous subsection are used to acheive propagation of the fault through the control construct.

Thus the requirements for a STUCKTHEN fault are :-

1. Form a list of faulty assignment statements. For a STUCKTHEN fault, statements under the THEN clause of the IF statement execute irrespective of the value in the control expression. Thus, if assignment statements under the ELSE clause are used for justifying values in objects during the preload operation, the desired "bad_val" may not get preloaded and so it is necessary to propagate the effects of the preload to an observable output. During the preload operation, as each

assignment statement is executed, a check is made to see if that statement is a member of "faulties"; if so, a flag is set. At the end of the preload operation, this flag is checked and effects of preload propagated to an observable output if necessary.

2. Propagate the 0/1 value pair from the IF statement to an observable output.

STUCKELSE: A test for STUCKELSE fault condition follows the same procedure as that for the STUCKTHEN fault. The goal is to attempt execution of the THEN clause, and then to observe which clause is actually executed.

Thus the requirements of a STUCKELSE fault are :-

1. Set up the list of faulty assignments ("faulties") which in this case are the assignment statements under the THEN clause.
2. Propagate the 1/0 value pair from the IF statement to an observable output.

DEADPROCESS: A test for DEADPROCESS fault condition selects an assignment statement under that process and then observes whether that statement actually executes with other conditions set up as desired.

Thus the requirements of a DEADPROCESS fault are :-

1. Set up the list of faulty assignments ("faulties") which in this case are the assignment statements under that process statement.
2. Propagate the 1/0 value pair from the process statement to an observable output.

DEADCLAUSE: A test for a DEADCLAUSE fault condition attempts to execute the faulty clause of the CASE statement. A data object under that clause is then observed to

determine whether or not the clause actually executes. The dead clause is examined, and an assignment statement under it is chosen. In order to enable the tester to look for known "good_val"/"bad_val" the destination object of that assignment statement is preloaded with a "bad_val" and then the faulty clause is executed to execute that assignment statement. In essence, this is a procedure for testing an assignment statement under the dead clause for an ASSCNTL fault.

Thus the requirements for a DEADCLAUSE fault are :-

1. Set up the list of faulty assignments ("faulties") which in this case are the assignment statements under the dead clause.
2. Choose an assignment statement under the dead clause of the CASE statement.
3. Perform an ASSCNTL test on this assignment statement.

MICROOP: A test for a faulty micro-operation involves setting values to arguments such that the result of the good micro-operation differs from the result of the bad micro-operation. The fault model dictates the faulty behavior of the good micro-operation. A lookup table is maintained which stores the bad micro-operation(s) for each micro-operation, the inputs which sensitize the faulty operation, and the resulting good/bad output values.

Thus the requirements for a MICROOP test are :-

1. Set up the list of faulty assignments ("faulties"). For micro-operations in assignment statements it is simply that assignment statement whereas for micro-operations in control expressions it is the list of all assignment statements under that control construct.

2. Get the fault sensitization and fault propagation requirements for the faulty micro-operation from the lookup table.
3. Justify the values required for the micro-operation arguments to external inputs.
4. Propagate the good/bad results of the micro-operation to an observable output.

STUCKDATA: A STUCKDATA fault can be viewed as the degenerate case of a MICROOP fault. Any value different from the stuck value will sensitize the fault.

Thus the basic requirements for a STUCKDATA fault are :-

1. Set up the list of faulty assignments ("faulties"). For stuck objects in assignment statements it is simply that assignment statement whereas for stuck objects in control expressions it is the list of all assignment statements under that control construct.
2. Select a test-value at the fault site which is different from the stuck-value.
3. Justify the test-value at the fault site to external inputs.
4. Propagate the test-value/stuck-value value pair to an observable output.

6.8.3.4 Test Generation Example

The test generation process is described with the aid of an example. Figure 6 on page 63 shows the VHDL model for an 8-bit register from [31]. Also shown is the test vector sequence for micro-operation "and" in statement s5 failing to "or". The first test vector sensitizes the fault by setting DS1 and NDS2 to a 1. Note that NDS2 is set to a logic 1, while DS1 is set to R, which denotes a 0 to 1 transition. This transition is used to activate the process ENABLE. The above conditions sensitize the fault since

```
    entity REGISTER
     (DI: in BIT_VECTOR(1 to 8);
      STRB,
      DS1,
      NDS2: in BIT;
      DO: out BIT_VECTOR(1 to 8)) is
    end REGISTER;

    architecture ARCH of REGISTER is

     signal REG: BIT_VECTOR(1 to 8);
     signal ENBLD: BIT;

    begin
s1    STROBE : process(STRB)
       begin
s2       if (STRB = '1') then
s3         REG <= DI;
        end if;
       end process;

s4    ENABLE : process(DS1,NDS2)
       begin
s5       ENBLD <= DS1 and not NDS2;
       end process;

s6    OUTPUT : process(REG,ENBLD)
       begin
s7       if (ENBLD = '1') then
s8         DO <= REG;
        else
s9         DO <= '11111111';
        end if;
       end process;

    end ARCH;
```

Register
[14, microop, [s5, [45], []], bitand, bitor]

t\obj	di	strb	ds1	nds2	do
t0+0	-	-	R	1	-
t0+1	00000001	R	1	1	-
t0+2	00000000	R	1	1	11111111/00000000

Figure 6. VHDL Model for an 8-bit Register

the signal ENBLD will receive the value 1 in the presence of the fault (OR operation), and a value 0 for no fault (AND operation). The next two test vectors cause a 00000001 to 00000000 change to occur on the signal REG. This triggers the process OUTPUT. Process OUTPUT tests ENBLD with an if statement. Thus, DO will be 00000000 in the presence of the fault and 11111111 for the no fault case.

Chapter VII

Results

In this chapter, the results obtained by applying the new algorithm to five circuit models are presented. A brief description is given for each circuit model. Following this, for each model, the number of tests (behavioral) successfully generated and the total amount of CPU time required is listed. Also, for some of these circuits, the result of fault coverage experiment is presented. Appendix B contains listings of the VHDL source code and fault list for each circuit model.

There are certain faults for which the test generation algorithm does not generate tests. These faults fall into one of the following categories:

- Excluded faults: Some faults in control expressions involving the use of the 'STABLE attribute are excluded. For example, in

```
s1: if (CLK = '1'  and not CLK'stable) then
s2:     Q <= D;
```

a STUCKTHEN fault on s1 would mean that s1 executes every delta time i.e. at infinite frequency. This fault has no physical correspondence and the test gener-

ation algorithm will not generate a test for this fault. Thus, STUCKTHEN faults on IF statements containing the 'STABLE attribute, micro-operation and stuck-at faults on the expressions of such statements are excluded.

- Invalid Faults: The Fault List Extractor lists vector expressions to be stuck-at all 0's and stuck-at all 1's. Thus a literal "0001" as an expression would be listed for stuck-at-0000 and stuck-at-1111 faults. A "0000" or a "1111" cannot be justified in the literal "0001" and so such tests are listed as invalid.
- Masked Faults: The algorithm does not generate tests for faults which reuse the fault site during propagation of the fault syndrome. Traversal of fault site during propagation could disturb the value pair set up at the fault site. This is discussed in more detail in the next section.

For all other faults in the model, the test generation algorithm should be able to generate tests successfully. However, faults for which tests are generated successfully again fall into one of the following two categories:

- User Assisted Tests: The tests for certain faults require human intervention to run the tests to completion without overflowing the Prolog interpreter. Also, some tests may require intervention to generate successful tests (e.g. to circumvent invalid states in a sequential circuit). In other cases, intervention may be necessary to produce efficient tests (e.g. short test vector sequences).
- Faults for which tests run to completion without any human intervention.

7.1 CKTA

CKTA is a simple circuit consisting of two D flip-flops, with directly controllable clocks. The outputs of the two flip-flops are ANDed to produce the only circuit output ANDOUT. Figure 1 on page 3 shows the circuit diagram and VHDL description for this circuit. An inspection of the VHDL model shows that it consists of two process statements along with a concurrent signal assignment statement.

Results of Fault Coverage Experiment: The test sequences generated by the algorithm were collapsed to eight test sequences. When the combine test sequences were applied to a gate-level model of the register, 33 out of the 33 stuck-at faults were detected, yielding a fault coverage of 100%.

7.2 REGISTER

REGISTER is a simple 8-bit register [31] with a STRB input which strobes the register, and enable inputs DS1 and NDS2 which enable the output buffers. The circuit is described using three processes. The STROBE process loads the register with a rising edge on the STRB input. The ENABLE process asserts the ENBLD signal when the AND of DS1 and $\overline{NDS2}$ is true. The OUTPUT process assigns a value to the 8-bit output DO which is either the contents of the register or "11111111" depending upon whether ENBLD is "1" or "0".

Results of Fault Coverage Experiment: The test sequences generated by the algorithm were collapsed to nine test sequences. When the combine test sequences (the

total number of test vectors was 33) were applied to a gate-level model of the register, 110 out of the 110 stuck-at faults were detected, yielding a fault coverage of 100%.

7.3 COCN

COCN is a 4-bit up/down controlled counter [31]. The counter consists of four logic blocks: i) a 2-bit control register and a 2-to-4 decoder (CONSIG), ii) a 4-bit limit register (LIM), iii) a 4-bit comparator, and iv) a 4-bit counter. The counter received a 2-bit control input (CON), which is decoded to perform the following four commands:

1. (CON = 00) - Clear the counter.
2. (CON = 01) - Load the LIM (limit) register with the value on the DATA input lines.
3. (CON = 10) - Count up until the COUNT equals the value in the LIM register.
4. (CON = 11) - Count down until COUNT equals the value in the LIM register.

The counter, when commanded to count (CON =10 or 11) and enabled (EN = 1), counts positive clock transitions, in an up or down mode, until the limit is reached. When a counting command is decoded, the ENIT signal is sent to the comparator by the decoder logic. The comparator uses the rising edge of ENIT to set EN = 1 initially. After that, EN is controlled by the comparator (if COUNT is equal to LIM, EN is set to 0; otherwise, EN remains at 1). CON is loaded and decoded on the rise of STRB, and DATA is loaded on the fall of STRB with the "load" (CON = 01) command. The VHDL description for this circuit is described using five processes which describe the behavior discussed above.

Results of fault coverage experiment: The test sequences generated by the algorithm were collapsed to eight test sequences. When the combined test sequences (the total number of test vectors was 132) were applied to the gate-level model of the counter, 157 out of the 171 stuck-at faults were detected, yielding a fault coverage of 91.8%. To evaluate the length of the test set, 10 sets of pseudo-random test vectors of length 255 were generated. Since the number of inputs to the counter is 8, the maximum number of vectors generated by a LFSR is $2^8 - 1 = 255$. The average gate-level fault coverage for the 10 test sets was 81.3%. Thus the test set generated by the algorithm gives a better coverage and shorter length compared to pseudo-random tests.

7.4 I8212

I8212 is Intel's Buffered Latch [31]. It has control inputs DS1, DS2, MD, and STB. These inputs are used to control device selection, data latching, output buffer state and service request flip-flop. When DS1 is low and DS2 is high the device is selected. When MD is high (output mode), the output buffers are enabled and the source of clock to the data latch is from the device selection logic. When MD = 0 (input mode), the STB input is used as a clock to the data latch, and to synchronously reset the service request flip-flop (SRQ). SRQ is negative edge-triggered. SRQ is set by NCLR being low or the device being selected. The circuit is described in VHDL using concurrent signal assignment statements for the control section. The latch sections are implemented using process statements.

7.5 UARTO

UARTO is the transmit part of a simple UART. An active RESET signal resets the UART and clears a transmit busy flag. A rising edge on a WRITE signal loads a count register with "11" and the transmit register with the data to be transmitted. Also, the transmit busy signal is asserted. Following this, every rising edge on the CLOCK signal transmits one bit on the output pin till the count register contains "00", whence the transmit busy flag is deasserted. This behavior is implemented in VHDL using three processes as shown in Appendix B.

Table 1. Results of experiments with BTG

Circuit Name	Number of Behavioral Faults	Number of Successful Tests	Total CPU Time
CKTA	12	12	284.35s
Register	15	15	151.98s
COCN	56	23	921.50s
I8212	44	36	552.21s
UARTO	28	23	199.15s

Table 2. Results of fault coverage experiment

Circuit Name	Number of Gate level Faults	Number of Faults Covered	Fault Coverage
CKTA	33	33	100%
Register	110	110	100%
COCN	171	157	91.8%
I8212	-	-	-
UARTO	-	-	-

7.6 Speed Performance and Memory Requirement

O'Neill's algorithm and the new method require different input descriptions for generating test vectors. However, to compare the speed performance a single process description of the circuit shown in Figure 1 on page 3 was given to both algorithms. O'Neill's algorithm required 105s of CPU time to generate 6 out of the 14 behavioral tests. The new method required 40.2s for the same tests. For a mutli-process description of the same circuit, the new method required 284.35s to generate all the 12 behavioral tests. This increased time required is due to the overhead involved in executing process statements (execution of a process statement requires different values in adjacent time periods in one of its objects, which have to be justified to primary inputs). Thus, with the expanded VHDL subset supported by the new method, the speed performance of the of the two algorithms is comparable.

O'Neill's algorithm required 192Kbytes of memory; the new algorithm requires 151Kbytes of memory space. Also, the number of alternatives for each rule has been reduced to the minimum that is required. The combined effect of this is that there are fewer stack overflows than before. Besides this, the step involving storage of justification rules has been eliminated. Although this does not result in great space saving for small circuits, it could be substantial for large models.

Chapter VIII

Analysis and Suggestions

This chapter discusses the limitation of the test generation algorithm and suggests areas for future work.

8.1 Forward Implication

One serious limitation of the test generation algorithm is the lack of complete forward implication. Complete forward implication in test generation from HDL descriptions would require determining the effects of each and every justification operation. After a value has been justified in an object, all its implications need to be determined. The object could appear in source expressions of other assignment statements or in the control expressions of control statements. In the first case the destination objects of those assignment statements would have *implied* values which could lead to further implications. In the second case, the destination objects of all statements that get executed will have *implied* values. Again, this could lead to further implications. Besides delaying the detection of conflicts, the lack of complete forward implication can

result in the generation of wrong test vectors. The algorithm during the course of generating tests would assume certain states for the circuit. However, some justification operation could have changed the states of the circuit and the algorithm having no knowledge of this would continue the process of generating a test resulting in a wrong test. Complete forward implication can be achieved only by using a simulator in parallel with the test generation algorithm. Every time a justification operation is performed the implications of that assignment could be determined to determine the new states of the circuit. In some cases this limitation can be overcome by making use of the user interface to intelligently provide test values.

8.2 Reconvergent Fanout

Another limitation of the algorithm is that it cannot handle reconvergent fanout. This limitation is due to,

- Representation of object values: The test generation algorithm uses only fixed values for objects i.e 0 or 1 and vectors of 0's and 1's. A D or $\overline{D}$ is not available for representing object values as in the D-algorithm. Also, rules defining the behavior of micro-operations also assume single fixed values.
- Single path propagation: The test generation algorithm uses single path propagation, with the fault syndrome being propagated along a single selected path and values being justified in other objects along that path. For circuits having reconvergent fanout, the fault syndrome needs to be propagated simultaneously along all paths from a fanout point. Since this is not done, no tests are generated in some cases for circuits having reconvergent fanout.

8.3 Traversal of the fault site during propagation

Imagine a fault at the site marked X. The test for this fault is determined by i) setting a good/bad value pair at the fault site X, and ii) propagating the value pair to an observable output.

Setting the good/bad value pair requires a)preloading the fault site with a bad value, and then b) loading the good val. If the fault site is traversed during preloading, one cannot be sure that the object will get loaded with the desired value. Barclay's algorithm marked the fault site to make sure that it was never used to preload or propagate its own test. Thus, no test was generated if the fault site was used during justification or propagation. However, the current implementation of the algorithm permits the fault site to be traversed during preloading. Also, this approach has been generalized to include fault sites in both assignment statements and control structures. At the end of the preload operation, a check is made to determine if the fault site has been traversed. If so, the effect of preloading the fault site is propagated to an observable output. However, the algorithm still does not permit traversal of the fault site during propagation of the fault syndrome. Traversal of the fault site during propagation could result in masking out the good/bad value pair set up at the fault site. This limitation causes many tests to fail in sequential circuits.

8.4 Extension to Board Level Testing

The current algorithm has been used to generate tests for behavioral descriptions of digital devices, i.e descriptions having a single entity body. A possible extension

could be to test a board consisting of devices whose behavioral dataflow descriptions are available, i.e descriptions with multiple entities. It should be relatively straightforward to handle this. What one needs to define is the mapping between the actual circuit connectivity and the the device descriptions. However, since we fault HDL constructs, it will be necessary to fully expand the device descriptions to obtain a single description with the various inter-device signals assigned names during the mapping. If future work expands in this direction, it is likely that test conditions for the faults can be expressed in terms of justification, execution and propagation, so that the current algorithm could be used to develop tests for these faults.

8.5 Retention of Past Work

In the course of generating tests for a set of faults, the justification, execution, and propagation goals are solved repeatedly. It would be a simple matter to store the results of these operations for use the next time that operation is performed. It could be retrieved as a single complex event and inserted in the event queue, with each subevent assigned a time tag at that point. It may seem that the time saving from using such a technique would be substantial. However, this technique was not considered under the new approach for the following simple reason. To justify a given value in a signal in a given circuit, there could be large number of possible assignments for input and signal values. The set of values determined by the justification operation at a given time may not be necessarily suitable at another time. Thus, in most cases it would result in an increase in the runtime rather than a saving. Also, saving all possible sets of values for a justification operation may require an inordinately large database.

8.6 Increasing Test Effectiveness

The current method leaves don't care bits unassigned in the test vector sequence. If these values are intelligently specified in such a way that a series of different values get assigned to the don't care bits, the test set would exercise the data paths in the circuit more completely. However, care needs to be taken while assigning values to don't cares for clock signals. Assigning different values at adjacent times may cause unintended clocking of the circuit and disturb conditions assumed by the test generation algorithm. Another, area where test vector effectiveness could be increased is for tests for micro-operations representing large blocks of logic. The test generation algorithm generates only one vector for faults in any micro-operation. However, all gate level faults in the big micro-operation cannot be detected by a single vector. The selection of a reasonable subset of the all possible inputs needed to sensitize the micro-operation is a topic of current research at Virginia Tech. Routines that select values for don't care bits and values for big micro-operations can be easily interfaced with the test generation algorithm.

8.7 Expansion of the VHDL subset

The test generation algorithm can be used to generate tests for large circuit models. However, it currently uses a restricted VHDL subset which makes large models complex. The subset can be expanded to include other logical and arithmetic micro-operations so that tests for larger models can be generated more efficiently.

Chapter IX

Conclusions

An efficient test generation algorithm for behavioral descriptions of digital devices has been presented. It accepts input descriptions in an extended subset of VHDL consisting of multiple process statements and concurrent signal assignment statements. The fault model supports more fault types and the algorithm can generate more tests for a given model compared to previous approaches. It can make use of user assists to generate more efficient tests. It requires less memory space and has comparable speed performance compared to the method developed by O'Neill. The new approaches discussed in this thesis allow the new method to achieve this. The limitations of the algorithm have been discussed and areas for future work identified. The test generation algorithm provided with the information extracted from the VHDL model and assists in the determination of test values should result in an efficient test generation system.

Bibliography

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

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

3. H. Fujiwara and T. Simono, "On the Acceleration of Test Generation Algorithms", IEEE Trans. on Computers, vol. C-30, pp. 1137-1144, Dec. 1983.

4. M. S. Abadir and H. K. Reghbati, "LSI Testing Techniques", IEEE Micro, pp. 34-50, Feb. 1983.

5. Seynyon Shteingart, Andrew W. Nagle, John Grason, "RTG: Automatic Register Level Test Generator", 22nd Design Automation Conference, pp. 803-807, 1985.

6. R. A. Marlett, "EBT: A Comprehensive Test Generation Technique for Highly Sequential Circuits", Proc. of the 15th Design Automation Conf., pp. 335-339, 1978.

7. F. J. Hill and B. Huey, "SCIRTISS: A Search System For Sequential Circuit Test Sequences", IEEE Trans. on Computers, Vol. C-26, no. 5, pp. 490-502, May 1977.

8. S. Y. H. Su and Y. Hsieh, "Testing Functional Faults in Digital Systems Described by Register Transfer Language", IEEE Inter. Test Conf., pp. 447-457, Oct. 1981.

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

10. T. Lin and S. Y. Su, "The S-Algorithm: A Promising Solution for Systematic Functional Test Generation", IEEE Trans. on Computer-Aided Design, Vol. CAD-4, pp. 250-263, July 1985.

11. I. Stamelos, M. Melgara, M. Paolini, S. Morpurgo, and C. Segre, " A Multi-level Test Pattern Generation and Validation Environment", IEEE International Test Conf., pp. 90-96, Sep. 1986.

12. C. Robach and G. Saucier, "Microprocessor Functional testing", IEEE International Test Conf., 1980.

13. S. M. Thatte and J. A. Abraham, "Test Generation for Microprocessors" IEEE Trans. on Computers, Vol. C-29, pp.429-441, June 1980.

14. S. B. Akers, "Functional Testing with Binary Decesion Diagrams", Proc. of the 8th International Symposium on Fault Tolerant Computing, pp. 82-92, June 1978.

15. M. S. Abadir and H. K. Reghbati, "Functional Test generation for LSI circuits described by Binary Decrsion Diagrams", IEEE Inter. Test Conf., pp. 483-492, Nov. 1985.

16. H. P. Chang, W. A. Rogers, and J. A. Abraham, "Structured Functional Level Test Generation Using Binary Decesion Diagrams", IEEE International Test Conf., pp. 97-104, Sep. 1986.

17. M. J. Bending, "Hitest: A Knowledge-based Test Generation System", IEEE Design and Test, vol 2, no 3, pp. 83-92.

18. M. H. Shirley, "Generating Tests by Exploiting Designed Behavior", Proc. of AAAI-86, pp. 884-890, August 1986.

19. D. S. Brahme and J. A. Abraham, "Knowledge Based Test Generation For VLSI Circuits", Proc. of Inter. Conf. on Computer-Aided Design, pp. 292-295, 1987.

20. Y. H. Levendel and P. R. Menon, "Test Generation Algorithms for Computer Hardware Description Languages", IEEE Trans. on Computers, Vol. C-31, pp. 577-588, July 1982.

21. R. Khorram, "Functional Test Pattern Generation for Integrated Circuits", IEEE International Test Conf., pp. 246-249, Oct. 1984.

22. K. Son and J. Y. Fong, "Automatic Behavioral Test Generation", IEEE International Test Conf., pp. 161-165, Nov. 1982.

23. D. S. Barclay and J. R. Armstrong, "A Heuristic Chip-level Test Generation Algorithm", 23rd Design Automation Conf., pp. 257-262, June 1986.

24. IEEE Standard VHDL Language Reference Manual, IEEE, Inc., NY, Mar. 1988.

25. J. R. Armstrong, "Chip Level Modeling of LSI devices", IEEE Trans. on Computer-Aided Design, Vol. CAD-3, pp. 288-297, Oct. 1984.

26. J. R. Armstrong, "Chip Level Modeling with HDLs", IEEE Design and Test of Computers, Vol. 5, No. 1, pp. 8-18, Feb. 1988.

27. M. D. O'Neill, An Improved Chip-level Test Generation Algorithm, Master's Thesis, E. E. Dept., Virginia Polytechnic Institute and State University, Dec. 1987.

28. F. E. Norrod, " The E-Algorithm: An automatic Test Generation Algorithm for Hardware Description Languages", Master's Thesis, VPI & SU, Feb. 1988.

29. A. K. Gupta and J. R. Armstrong, "Functional Fault Modeling and Simulation for VLSI Devices", 22nd Design Automation Conf., pp. 720-726, June 1985.

30. C. Chao and F. G. Gray, "Micro-operation Perturbations in Chip Level Fault Modeling", 25th Design Automation Conf., pp. 579-582, June 1988.

31. J. R. Armstrong, Chip Level Modeling with VHDL, Prentice Hall Inc., Aug. 1988.

32. HILO-3 User Manual, Doc. #2522-0100, Genrad Inc., June 1985.

33. W. Clocksin and C. Mellish, Programming in Prolog.

Appendix A

User's Manual

The new test generation algorithm is implemented in Prolog using the Portable Prolog interpreter and is currently installed on a Data General MV/10000 machine. The Portable Prolog interpreter is written in Pascal and the syntax used is discussed in [33]. The function of each module in the system is explained as follows:

1. **Preprocessor**
   a. **Translator** - VHDL behavioral descriptions are converted into Prolog predicates. This routine is written in C.
   b. **Intermediate Form Extractor (IFE)** - The raw Prolog predicates generated by the translator are converted into an intermediate form (Prolog facts) that represents the knowledge of the circuit. This routine is written in Prolog.
   c. **Fault List Extractor (FLE)** - A list of behavioral faults is constructed. This routine is written in Prolog.
2. **Behavioral Test Generator (BTG)** - This module implements the test generation algorithm and thus keeps the operation rules for the three high level goal types (justification, propagation, and execution). It generates tests for each fault in the

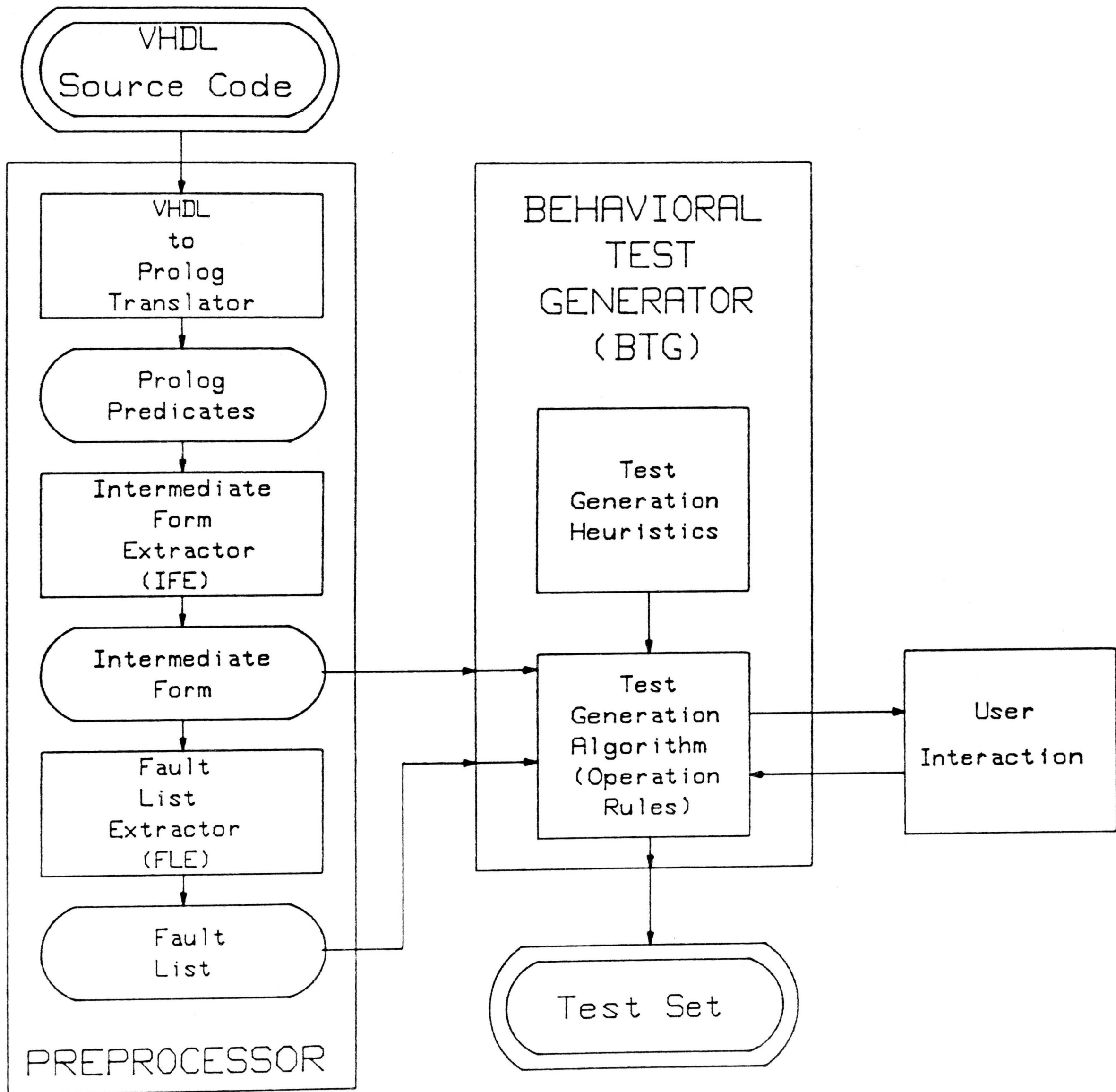


Figure 7. A Behavioral Test Generation System

fault list using the intermediate form generated by the IFE and the operation rules. The BTG has the option to make use of user assists to the test generation process.

At the current time, the IFE, FLE, and BTG are fully implemented. The translator was originally designed for an earlier version of the algorithm and is being updated. Currently, we translate VHDL into Prolog predicates manually.

All files used by the test generation algorithm reside in the NEW directory. This directory has two subdirectories: HDLDIR and WAVEDIR. The input and output files for the Translator, Intermediate Form Extractor, and Fault List Extractor are stored in HDLDIR. The output of the Behavioral Test Generator, i.e the test vectors are stored in the WAVEDIR.

A.1 Preprocessor

A.1.1 Translator

The input to the Translator is a behavioral description written in a subset of VHDL. The output of the translator is the Prolog representation of the behavioral description. The subset of VHDL that can used in the input description is discussed below.

1. signal objects but not variable objects. This is because a signal has a fixed value at a given time whereas the value of a variable depends on the position in the code.

2. signal types BIT and BIT_VECTOR. Other types can be mapped to these. For example, BOOLEAN can be mapped to BIT, and INTEGER to BIT_VECTOR.
3. concurrent signal assignment statements.
4. multiple process statements.
5. IF and CASE statements.
6. delta delay timing. Since we are considering fixed faults, assignment statements have no "after" clauses or they are ignored.
7. operations discussed in Figure 8 on page 85.

Note that such high level modeling constructs such as FOR loops and WAIT statements have been excluded. The chosen subset results in behavioral models which allow the fault sensitization and signal propagation requirements of the algorithm to be met more easily than with higher level descriptions. The translator is invoked by typing

```
VHDLTOP
```

The program prompts for the input file name which is assumed to have an extension .VHD. The output file will have the same name as the input file with an extension .HDL.

A.1.2 Intermediate Form Extractor

The input to the Intermediate Form Extractor is the Prolog representation of the VHDL behavioral description and the output is a set of Prolog facts that represent the knowledge of the circuit under test. To run the IFE, first move to the NEW directory and start the Prolog interpreter by typing

BIT operations:

Operation	Description
BITAND	-AND
BITOR	-OR
BITNOT	-NOT
BITXOR	-XOR
BITEQV	-XNOR

Vector Operations:

Operation	Description
BVAND	-bitwise AND
BVOR	-bitwise OR
BVNOT	-bitwise NOT
BVXOR	-bitwise XOR
BVEQV	-bitwise XNOR
BVEQ	-vector equality
BVNEQ	-vector inequality
BVADD	-unsigned vector addition
BVSUB	-unsigned vector difference
BVLE	-unsigned less than
BVLT	-unsigned less than or equal to

Other operations:

Operation	Description
BVCAT	-concatenation of vectors
BVSUBV <bit position>	-a bit of a vector
BVSUBV <left position> <right position>	-slice of a vector
BITBV	-convert bit to bitvector
STABLE	-'STABLE attribute

Figure 8. Available micro-operations and their description

```
)prolog
```

Load the IFE by typing

```
?-consult('ife.pro').
```

at the Prolog prompt.

To run the IFE, type

```
?-start.
```

It will prompt you for the model name (the model name is the name of your VHDL source file without the extension). The .HDL for that model will get loaded in the database. Next, type

```
?-go.
```

The IFE will extract information from the Prolog representation and store it in a file which has the same name as the VHDL source file but a IFE extension.

A typical session with the IFE is shown below (Bold-faced words denote user input).

```
)prolog
Portable Prolog Release 2.1.

?-consult('ife.pro').
'mdolib.pro' consulted.
'execute.pro' consulted.
'ife.pro' consulted.

?-start.
Enter model name, followed by a period: ckta.
'hdldir: ckta.hdl' consulted.
yes

?-go.
Model Name: Register
Scanning begins...
Picking statements...
Picking objects...
Picking expressions...
Picking clauses...
Picking parents...
Picking subordinates...
Picking assignments...
Finding outdistances...
Finding uses...
Sorting subordinates...
Making vies ...

Writing to file...
yes

?-end.
[Leaving Prolog]
```

A.2 Fault List Extractor

The input to the Fault list extractor is the Prolog representation of the circuit and the output is a set of behavioral faults. To run the FLE, first start the Prolog interpreter as discussed above. Then load the FLE, by typing **consult('fle.pro').**; type **start.**,enter model name, and then type **go.**. The fault list will be stored in a file with the name same as that of the VHDL source file but with a .FLE extension.

A typical session with the FLE is shown below.

```
)prolog
Portable Prolog Release 2.1.

?-consult('fle.pro').
'mdolib.pro' consulted.
'lookup.pro' consulted.
'bitops.pro' consulted.
'bvops.pro' consulted.
'fle.pro' consulted.

?-start.
Enter model name, followed by a period: ckta.
'hdldir: ckta.hdl' consulted.
'hdldir: ckta.ife' consulted.
yes

?-go.
Listing faults...
Regrouping...
Numbering faults...
Writing fault list...
Done
yes

?-end.
[Leaving Prolog]
```

A.3 Behavioral Test Generator

The BTG attempts to generate a test vector sequence for each and every fault in the the fault list generated by the IFE. To run the BTG follow the same steps as shown above.

A typical session with the BTG is shown below.

```
)prolog
Portable Prolog Release 2.1.

?-consult('btg.pro').
'mdolib.pro' consulted.
'lib.pro' consulted.
'propagate.pro' consulted.
'justify.pro' consulted.
'execute.pro' consulted.
'bitops.pro' consulted.
'bvops.pro' consulted.
'lookup.pro' consulted.
'waveform.pro' consulted.
'time.pro' consulted.
'justrules.pro' consulted.
'btg.pro' consulted.

?-start.
Enter model name, followed by a period: ckta.
'hdldir: ckta.hdl' consulted.
'hdldir: ckta.ife' consulted.

Model Description: CKTA

Human interaction required? y.

Should dont_disturb rules be enforced? n.

Change observability sequence of any signal? y.
Signal name is q1.
Default observability sequence is [s7,s6].
New sequence is [s6,s7].

Change observability of any other signal? n.

Change controllability of any signal? n.
yes

?-go.
Enter fault number: 2.
-------
-------
?-end.
[Leaving Prolog]
```

If human interaction is not requested the BTG will try to generate a test vector sequence using the rules in the database. If user interaction is requested, the BTG will

prompt the user in selecting good/bad values for testing statements, paths for propagation, and in the selection of vector-wide values. However, at each prompt the user will be first asked if he wishes to assist the BTG. If he does, the input from the terminal will be used, otherwise the BTG will continue to generate the test using its own rules.

Appendix B

Circuit Models and Fault Lists

```
    entity CKTA is
     (D, CLK1, CLK2 : in BIT;
      ANDOUT : out BIT)
    end CKTA;

    architecture BEHAVIOR of CKTA is

       signal Q1, Q2 : BIT;

1:  process(CLK1)
    begin
2:     if (CLOCK1='1') then
3:        Q1 <= D;
    end process;

    process(CLK2)
    begin
4:     if (CLOCK2='1') then
5:        Q2 <= Q1;
    end process;

6:     ANDOUT <= Q1 AND Q2;

    end BEHAVIOR;
```

```
"CKTA".
[1, deadprocess, s1].
[2, stuckthen, s2].
[3, stuckelse, s2].
[4, microop, [s2, [45], []], biteqv, bitxor].
[5, assncntl, s3].
[6, deadprocess, s4].
[7, stuckthen, s5].
[8, stuckelse, s5].
[9, microop, [s5, [45], []], biteqv, bitxor].
[10, assncntl, s6].
[11, assncntl, s7].
[12, microop, [s7, [45], []], bitand, bitor].
[13, stuckdata, [s2, [45], []], [bit, [48]]].
[14, stuckdata, [s2, [45], []], [bit, [49]]].
[15, stuckdata, [s2, [45], [76]], [bit, [48]]].
[16, stuckdata, [s2, [45], [76]], [bit, [49]]].
[17, stuckdata, [s2, [45], [82]], [bit, [48]]].
[18, stuckdata, [s2, [45], [82]], [bit, [49]]].
[19, stuckdata, [s3, [45], []], [bit, [48]]].
[20, stuckdata, [s3, [45], []], [bit, [49]]].
[21, stuckdata, [s5, [45], []], [bit, [48]]].
[22, stuckdata, [s5, [45], []], [bit, [49]]].
[23, stuckdata, [s5, [45], [76]], [bit, [48]]].
[24, stuckdata, [s5, [45], [76]], [bit, [49]]].
[25, stuckdata, [s5, [45], [82]], [bit, [48]]].
[26, stuckdata, [s5, [45], [82]], [bit, [49]]].
[27, stuckdata, [s6, [45], []], [bit, [48]]].
[28, stuckdata, [s6, [45], []], [bit, [49]]].
[29, stuckdata, [s7, [45], []], [bit, [48]]].
[30, stuckdata, [s7, [45], []], [bit, [49]]].
[31, stuckdata, [s7, [45], [76]], [bit, [48]]].
[32, stuckdata, [s7, [45], [76]], [bit, [49]]].
[33, stuckdata, [s7, [45], [82]], [bit, [48]]].
[34, stuckdata, [s7, [45], [82]], [bit, [49]]].
```

```
    entity REGISTER is
     (DI : in BIT_VECTOR(1 to 8);
      STRB,
      DS1,
      NDS2 : in BIT;
      DO : out BIT_VECTOR(1 to 8));
    end REGISTER;

    architecture ARCH of REGISTER is

     signal DID: BIT_VECTOR(1 to 8);
     signal ENBLD: BIT;CTOR(1 to 8);

    begin
1:  process(STRB)
    begin
2:    if (STRB ='1') then
3:       DID <= DI;
    end process;

4:  process(DS1,NDS2)
    begin
5:    ENBLD <= DS1 and not NDS2;
    end process;

6:  process(DID,ENBLD)
    begin
7:    if (ENBLD ='1') then
8:       DO <= DID;
      else
9:       DO <= '11111111';
      endif;
    end process;

    end ARCH;
```

```
"Register".
[1, deadprocess, s1].
[2, stuckthen, s2].
[3, stuckelse, s2].
[4, microop, [s2, [45], []], biteqv, bitxor].
[5, assncntl, s3].
[6, deadprocess, s4].
[7, assncntl, s5].
[8, microop, [s5, [45], []], bitand, bitor].
[9, microop, [s5, [45], [82]], bitnot, bitbuf].
[10, deadprocess, s6].
[11, stuckthen, s7].
[12, stuckelse, s7].
[13, microop, [s7, [45], []], biteqv, bitxor].
[14, assncntl, s8].
[15, assncntl, s9].
[16, stuckdata, [s2, [45], []], [bit, [48]]].
[17, stuckdata, [s2, [45], []], [bit, [49]]].
[18, stuckdata, [s2, [45], [76]], [bit, [48]]].
[19, stuckdata, [s2, [45], [76]], [bit, [49]]].
[20, stuckdata, [s2, [45], [82]], [bit, [48]]].
[21, stuckdata, [s2, [45], [82]], [bit, [49]]].
[22, stuckdata, [s3, [45], []], [bv, [48, 48, 48, 48, 48, 48, 48, 48]]].
[23, stuckdata, [s3, [45], []], [bv, [49, 49, 49, 49, 49, 49, 49, 49]]].
[24, stuckdata, [s5, [45], []], [bit, [48]]].
[25, stuckdata, [s5, [45], []], [bit, [49]]].
[26, stuckdata, [s5, [45], [76]], [bit, [48]]].
[27, stuckdata, [s5, [45], [76]], [bit, [49]]].
[28, stuckdata, [s5, [45], [82]], [bit, [48]]].
[29, stuckdata, [s5, [45], [82]], [bit, [49]]].
[30, stuckdata, [s5, [45], [82, 76]], [bit, [48]]].
[31, stuckdata, [s5, [45], [82, 76]], [bit, [49]]].
[32, stuckdata, [s7, [45], []], [bit, [48]]].
[33, stuckdata, [s7, [45], []], [bit, [49]]].
[34, stuckdata, [s7, [45], [76]], [bit, [48]]].
[35, stuckdata, [s7, [45], [76]], [bit, [49]]].
[36, stuckdata, [s7, [45], [82]], [bit, [48]]].
[37, stuckdata, [s7, [45], [82]], [bit, [49]]].
[38, stuckdata, [s8, [45], []], [bv, [48, 48, 48, 48, 48, 48, 48, 48]]].
[39, stuckdata, [s8, [45], []], [bv, [49, 49, 49, 49, 49, 49, 49, 49]]].
[40, stuckdata, [s9, [45], []], [bv, [48, 48, 48, 48, 48, 48, 48, 48]]].
[41, stuckdata, [s9, [45], []], [bv, [49, 49, 49, 49, 49, 49, 49, 49]]].
```

```
entity CONTROLLED_CTR(
       CLK,STRB : in BIT;
       CON : in BIT2_VECTOR;
       DATA : in BIT2_VECTOR;
       COUNT : out BIT2_VECTOR) is
end CONTROLLED_CTR;


architecture ARCH of CONTROLLED_CTR is

   signal
     EN ,ENIT: BIT;
     LIM : BIT2_VECTOR;
     CONSIG : BIT4_VECTOR;

s01: DECODE:
     process (STRB)
     begin
s02:    if STRB='1'  then
s03:       case INTVAL(CON) is
            when 0 =>
s04:           CONSIG <= "1000";
s05:           ENIT   <= '0';
            when 1 =>
s06:           CONSIG <= "0100";
s07:           ENIT   <= '0';
            when 2 =>
s08:           CONSIG <= "0010";
s09:           ENIT   <= '1';
            when 3 =>
s10:           CONSIG <= "0001";
s11:           ENIT   <= '1';
          end case;
        end if;
     end process DECODE;

s12: LOAD_LIMIT:
     process (STRB)
     begin
s13:    if (STRB='0' and CONSIG(2)='1') then
s14:       LIM <= DATA;
        end if;
     end process LOAD_LIMIT;

s15: CLEAR_CTR:
     process (CONSIG(3))
     begin
s16:    if CONSIG(3)='1' then
s17:       COUNT <= "00";
        end if;
     end process CLEAR_CTR;
```

```
s18: CNT_UP_OR_DN:
    process (CLK)
    begin
s19:    if (CLK='1' and EN='1') then
s20:      if CONSIG(1)='1' then
s21:           COUNT <= ADD(COUNT,"01");
s22:      else if CONSIG(0)='1' then
s23:           COUNT <= SUB(COUNT,"01");
          end if;
        end if;
    end process CNT_UP_OR_DN;

s24: COMP:
    process (COUNT,LIM,ENIT)
    begin
s25:    if (COUNT = LIM) then
s26:      EN <= '0';
s27:    else if (ENIT='1' and not ENIT'stable) then
s28:           EN <= '1';
        end if;
    end process COMP;

end ARCH;
```

"Controlled Counter ".
[1, deadprocess, s1].
[2, stuckthen, s2].
[3, stuckelse, s2].
[4, microop, [s2, [45], []], biteqv, bitxor].
[5, deadclause, s3, [[bv, [48, 48]]]].
[6, deadclause, s3, [[bv, [48, 49]]]].
[7, deadclause, s3, [[bv, [49, 48]]]].
[8, deadclause, s3, [[bv, [49, 49]]]].
[9, assncntl, s4].
[10, assncntl, s5].
[11, assncntl, s6].
[12, assncntl, s7].
[13, assncntl, s8].
[14, assncntl, s9].
[15, assncntl, s10].
[16, assncntl, s11].
[17, deadprocess, s12].
[18, stuckthen, s13].
[19, stuckelse, s13].
[20, microop, [s13, [45], []], bitand, bitor].
[21, microop, [s13, [45], [76]], biteqv, bitxor].
[22, microop, [s13, [45], [82]], bveq, bvneq].
[23, assncntl, s14].
[24, deadprocess, s15].
[25, stuckthen, s16].
[26, stuckelse, s16].
[27, microop, [s16, [45], []], bveq, bvneq].
[28, assncntl, s17].
[29, deadprocess, s18].
[30, stuckthen, s19].
[31, stuckelse, s19].
[32, microop, [s19, [45], []], bitand, bitor].
[33, microop, [s19, [45], [76]], biteqv, bitxor].
[34, microop, [s19, [45], [82]], biteqv, bitxor].
[35, stuckthen, s20].
[36, stuckelse, s20].
[37, microop, [s20, [45], []], bveq, bvneq].
[38, assncntl, s21].
[39, microop, [s21, [45], []], bvadd, bvsub].
[40, microop, [s21, [45], []], bvadd, bvxor].
[41, stuckthen, s22].
[42, stuckelse, s22].
[43, microop, [s22, [45], []], bveq, bvneq].
[44, assncntl, s23].
[45, microop, [s23, [45], []], bvsub, bvadd].
[46, deadprocess, s24].
[47, stuckthen, s25].
[48, stuckelse, s25].
[49, microop, [s25, [45], []], bveq, bvneq].
[50, assncntl, s26].

```
[51, stuckthen, s27].
[52, stuckelse, s27].
[53, microop, [s27, [45], []], bitand, bitor].
[54, microop, [s27, [45], [76]], biteqv, bitxor].
[55, microop, [s27, [45], [82]], bitnot, bitbuf].
[56, assncntl, s28].
```

```
    entity I8212 is
     (DI : in BIT_VECTOR(1 to 8);
      NDS1, DS2, MD, STB, NCLR : in BIT;
      DO : out BIT_VECTOR(1 to 8));
      NINT : out BIT;
    end I8212;

    architecture BEHAVIOR of I8212 is

     signal S0, S1, S2, S3, SRQ : BIT;
     signal Q : BIT_VECTOR(1 to 8);

    begin
1:  process(NCLR,S1)
    begin
2:     if (NCLR ='1') then
3:        Q <= '00000000';
4:     else if (S1 = '1' and not S1'stable) then
5:             Q <= DI;
       endif;
    end process;

6:  process(S3)
    begin
7:     if (S3 ='1') then
8:        DO <= Q;
       else
9:        DO <= '11111111';
       endif;
    end process;

10: process(S2,STB)
    begin
11:    if (S2 = '0') then
12:       SRQ <= '1';
13:    else if (STB = '1' and not STB'stable) then
14:            SRQ <= '0';
       endif;
    end process;

15: S0 <= not NDS1 and DS2 ;
16: S1 <= S0 and MD or STB and not MD;
17: S2 <= S0 nor not NCLR;
18: S3 <= S0 or MD;
19: NINT <= not SRQ nor S0;

    end BEHAVIOR;
```

```
"I8212".
[1, deadprocess, s1].
[2, stuckthen, s2].
[3, stuckelse, s2].
[4, microop, [s2, [45], []], biteqv, bitxor].
[5, assncntl, s3].
[6, stuckthen, s4].
[7, stuckelse, s4].
[8, microop, [s4, [45], []], bitand, bitor].
[9, microop, [s4, [45], [76]], biteqv, bitxor].
[10, microop, [s4, [45], [82]], bitnot, bitbuf].
[11, assncntl, s5].
[12, deadprocess, s6].
[13, stuckthen, s7].
[14, stuckelse, s7].
[15, microop, [s7, [45], []], biteqv, bitxor].
[16, assncntl, s8].
[17, assncntl, s9].
[18, deadprocess, s10].
[19, stuckthen, s11].
[20, stuckelse, s11].
[21, microop, [s11, [45], []], biteqv, bitxor].
[22, assncntl, s12].
[23, stuckthen, s13].
[24, stuckelse, s13].
[25, microop, [s13, [45], []], bitand, bitor].
[26, microop, [s13, [45], [76]], biteqv, bitxor].
[27, microop, [s13, [45], [82]], bitnot, bitbuf].
[28, assncntl, s14].
[29, assncntl, s15].
[30, microop, [s15, [45], []], bitand, bitor].
[31, microop, [s15, [45], [76]], bitnot, bitbuf].
[32, assncntl, s16].
[33, microop, [s16, [45], []], bitor, bitand].
[34, microop, [s16, [45], [76]], bitand, bitor].
[35, microop, [s16, [45], [82]], bitand, bitor].
[36, microop, [s16, [45], [82, 82]], bitnot, bitbuf].
[37, assncntl, s17].
[38, microop, [s17, [45], []], bitand, bitor].
[39, microop, [s17, [45], [76]], bitnot, bitbuf].
[40, assncntl, s18].
[41, microop, [s18, [45], []], bitor, bitand].
[42, assncntl, s19].
[43, microop, [s19, [45], []], bitand, bitor].
[44, microop, [s19, [45], [76]], bitnot, bitbuf].
```

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

    architecture BEHAVIOR of UARTO is

     signal TXCNT : BIT_VECTOR(2 downto 0);
     signal TXREG : BIT_VECTOR(3 downto 0);

    begin
 1: process(RESET, WRSTRB)
    begin
 2:   if RESET ='0' then
 3:     TXBUSY <= '0';
 4:   else if (WRSTRB = '1' and not WRSTRB'stable) then
 5:          TXBUSY <= '1';
 6:          TXREG <= "1" & DATABUS & "0";
 7:          TXCNT <= "11";
      endif;
    end process;

 8: process(CLK)
    begin
 9:   if (CLOCK = '1' and TXBUSY = '1') then
10:      DATAOUT <= TXREG(0);
11:      TXREG <= "1" & TXREG(3 downto 1);
12:      TXCNT <= bvsub(TXCNT,"01");
      endif;
    end process;

13: process(TXCNT)
    begin
14:   if (TXCNT = "00") then
15:     TXBUSY <= '0';
      endif;
    end process;

    end BEHAVIOR;
```

```
"UARTO".
[1, deadprocess, s1].
[2, stuckthen, s2].
[3, stuckelse, s2].
[4, microop, [s2, [45], []], biteqv, bitxor].
[5, assncntl, s3].
[6, stuckthen, s4].
[7, stuckelse, s4].
[8, microop, [s4, [45], []], bitand, bitor].
[9, microop, [s4, [45], [76]], biteqv, bitxor].
[10, microop, [s4, [45], [82]], bitnot, bitbuf].
[11, assncntl, s5].
[12, assncntl, s6].
[13, assncntl, s7].
[14, deadprocess, s8].
[15, stuckthen, s9].
[16, stuckelse, s9].
[17, microop, [s9, [45], []], bitand, bitor].
[18, microop, [s9, [45], [76]], biteqv, bitxor].
[19, microop, [s9, [45], [82]], bitnot, bitbuf].
[20, assncntl, s10].
[21, assncntl, s11].
[22, assncntl, s12].
[23, microop, [s12, [45], []], bvsub, bvadd].
[24, deadprocess, s13].
[25, stuckthen, s14].
[26, stuckelse, s14].
[27, microop, [s14, [45], []], biteq, bitneq].
[28, assncntl, s15].
```

The vita has been removed from the scanned document