AN AUTOMATIC TEST GENERATION METHOD FOR CHIP-LEVEL CIRCUIT DESCRIPTIONS

by

Daniel Scott Barclay

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:

James R. Armstrong, Chairman

Joseph G. Tront

Charles E. Nunnally

February 6, 1987

Blacksburg, Virginia

AN AUTOMATIC TEST GENERATION METHOD FOR CHIP-LEVEL CIRCUIT DESCRIPTIONS

by

Daniel Scott Barclay

James R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

An automatic method generates tests for circuits described in a hardware description language (HDL). The input description is in a non-procedural subset of VHDL, with a simplified period-oriented timing model. The fault model, based on previous research, includes micro-operation and control statement faults. The test method uses path-tracing, working directly from the circuit description, not a derived graph or table. Artificial intelligence problem-solving techniques of goals and goal solving are used to represent and manipulate sensitization, justification, and propagation requirements. Backtracking is used to recover from incorrect choices. The method is implemented in ProLog, an artificial intelligence language. Results of this experimental ProLog implementation are summarized and analyzed for strengths and weaknesses of the test method. Suggestions are included to counter the weaknesses. A user's manual is included for the experimental implementation.

Table of Contents

List of Illustrations

List of Tables

Chapter 1

Introduction

As circuit design and simulation has evolved from the gate level to the chip level to handle increasing circuit complexities, fault modeling and automatic test generation have lagged behind.

When digital circuits were described only at the gate level. Circuit faults were modeled with the classical stuck-at fault model, i.e. gate inputs or outputs became stuck at a logic one or a logic zero. Roth's D-algorithm [1] and direct enhancements [2,3,4] generated tests for stuck-at faults from a combinational gate-level description. The D-algorithm was also extended to test sequential gate-level circuits [5,6].

At the register connection level, the description consists of connected devices such as registers, counters, multiplexors, ALUs, etc., where these devices are description primitives for simulation efficiency and are not hierarchically constructed from gates. Typically, stuck-at models are used at this level. Test generation methods for this level are usually based on the D-algorithm [7,8,9].

With AHPL [10] and similar register transfer languages (RTLs), circuit descriptions no longer directly reflect circuit structure. This level required new fault models. Su [11, 12, 13, 14, 15, 16] developed fault models and a test generation method for RTL descriptions.

Recently, with the introduction of hardware description languages (RTLs) such as GSP [17, 18], GSP2 [19, 20], and VHDL [21, 22], circuit description has reached the chip level, where circuits are described as a whole, not as a hierarchy of low-level primitives, in a syntax similar to a computer program. Chip-level modeling also requires new fault models. Recent fault modeling research [22, 23, 24] gives a chip-level fault model derived from HDL constructs.

This chip-level fault model provides the basis for a chip-level test generation method. The fault model tells to how to sensitize chip-level faults, and the definition of the HDL in which a circuit is described tells how to complete tests given the sensitization requirements.

Contents

This thesis describes an approach to test generation for chip level models, including the HDL and fault model used, the test generation strategy, the current implementation and results, and suggestions for improvement.

Chapter II, "Literature Review", is a review of previous methods and techniques used in automatic test generation, from gate-level methods to other chip-level method.

Chapter III, "Test Method", describes the test method. It outlines the overall strategy, general test generation, special cases, and efficiency enhancements.

Chapter IV, "Results", describes the current implementation and reviews results of running test generation on some small models.

Chapter V, "Analysis and Suggestions", analyzes test generation results and suggests some improvements.

Chapter VI, "Conclusions", gives conclusions about the test method method.

Appendix A, "USER'S Manual", describes how to use the ProLog implementation.

Appendix B, "Circuit Models and Fault Lists", lists VHDL and internal-form descriptions of the circuit models used in this research, and gives faults lists for the circuit models, including brief comments on successes and failures.

Chapter 2

Literature Review

Gate Level Methods

Original automatic test generation work was done at the gate level.

Probably the most important development in automatic test generation was the Roth's D-algorithm [1]. The D-algorithm is based on D-calculus, in which D represents a good value of 1 and a faulty value of 0, $\overline{D}$ represents a good value of 0 and a faulty value of 1, and gate functions are redefined in terms of D or $\overline{D}$ (in addition to only 0 and 1). D-calculus provides for defining the effects of a stuck-at fault, and is used by many subsequent test methods. The D-algorithm uses the structure of a circuit to propagate faulty values through the circuit, and to justify values needed on internal lines. Justification may require choices, which in turn may lead to conflicts. The D-algorithm uses backtracking when conflicts are discovered, as with all subsequent methods.

The D-algorithm was adapted for sequential gate-level circuits. Kobu treated sequential sub-circuits (i.e. flip-flops) as primitive elements [6]. Roth used a heuristic loop-cut method to cut combinational feedback loops in gate-level circuits [5]. Both methods are based on changing the

sequential problem to a combinational one, by making multiple copies of the combinational portion of the circuit, cut at the state variables, and running combinational test generation on the resulting iterated circuit.

Goel [2] presents a method called PODEM, designed to handle reconvergent fanout, a problem on which the D-algorithm bogs down. Goel uses Roth's D-notation and D-propagation, but uses a more efficient control structure to implicitly search the space of all possible tests. He used the "branch and bound" method of searching to manipulate primary inputs to find a test, with heuristics to improve the efficiency of the search.

The 9-V (for nine logic values) algorithm [4] is another method addressing reconvergent fanout. For multiple path tracing (needed when there is reconvergent fanout), propagating a value through a gate sometimes needs a fixed 0 or 1 on inputs of the gate, and sometimes needs another D or $\overline{D}$. The D-algorithm tries each choice separately, backtracking when one fails. The 9-V algorithm uses additional logic values to represent each combination of choices, so less backtracking is required.

Register Connection Level Methods

Above the gate level is the register level, where circuit primitives are functional units such as registers, counters, multiplexors, ALUs, etc. (That is, these devices are not modeled hierarchically in terms of gates.)

Shteingart, Nagle, and Grason [7] present RTG, a register-level stuck-at test generator. RTG uses high-level modeling of sequential elements for simulation and test generation efficiency. Specifically, sequential components are modeled using primitive functions such as clear, load, etc., and are not constructed from gates. RTG uses the D-algorithm-based 9-V algorithm, extended to handle clock pulses, to work on the combinational portion of a sequential circuit. RTG has a global approach to sequential test generation. One key element is that it runs combinational test

generation on several time slices at once. (That is, it does not work progressively on adjacent time slices). When working with sequential circuits, a test generator must be careful about getting into endless loops by moving data values around in sequential feedback loops, or by returning to the same state. When loading a register, RTG checks that the register was not encountered before when justifying the current test.

Marlett [8, 9] presents a method called EBT. The main elements are two-period truth tables for sequential circuits and a unidirectional time flow for test vector generation.

The two-period truth tables relate the inputs, clocks, and outputs for sequential circuits. For a typical rising-edge-triggered device, the edge sensitivity is represented as a low value in the first period and a high in the second period. Required inputs are given in the first period, and outputs are given in the second period. Thus, the current state is related to previous period inputs and state.

EBT works backwards, generating the last test vector first, and the first one last. Instead of working bidirectionally forwards (for propagation) and backwards (for justification) from the time of the actual internal fault detection at the site of the fault, EBT works from detection at an output pin back to the first justification step. The purpose was to gather all constraints for a given time period at once, for better conflict detection. Marlett also uses decision numbers to improve backtracking. At a conflict, decision numbers associated with node values are used to find the most recent related decision; backtracking skips to that point, bypassing decisions known to be unrelated.

Register Transfer Level Methods

Above the register level is the register transfer level, in which the description consists of a sequence of instructions, and does not necessarily reflect the structure of the actual circuit.

Stephen Su has done work at register-transfer level [11, 12, 13, 14, 15, 16]. Lin and Su [11, 12] describe the S-algorithm. Their work is based on an AHPL-style [10] control structure. They use a fault model based on possible failure modes of the various elements of the RTL description.

They use symbolic execution to find expressions representing good and bad result values in terms of register and input values. This execution includes path constraints, values needed to select an execution path through particular statements. They then select input values such that the path constraints are satisfied and the results for good and bad execution differ. If such values are found, they constitute a test.

Graph Methods

Work has been done using various types of system graphs. However, most methods apply only to microprocessors, and are not useful for general circuits [26, 27, 28].

Abadir and Reghbati [29] propose a path-tracing test method for circuit modeled as connected modules, where the function of each module is described with binary decision trees. Binary decision trees are a concise description of a module, considerably denser that truth tables for typical circuits. The authors define current and next state variables, to treat sequential circuits as combinational ones. Module testing involves testing each branch of the binary decision tree describing each bit of the module. Faults are stuck lines and functional faults, stuck-at perturbations of the binary decision trees. Tests generation is based on the D-algorithm, modified for sequentiality. It runs on time frames, with checks to ensure trying only a finite number of internal states.

Hardware Description Language Methods

Most recently, circuits have begun to be described at the chip level [30], with hardware description languages (HDLs).

Levendel and Menon [31] generalize the D-algorithm to apply to "CHDL" descriptions of circuits. They derive D-propagation cubes for boolean switching expressions, and also for non-switching operations such as shifting and addition. They analyze IF and CASE control statements controlling transfers to switching expressions to derive D-propagation information. However, instead of these derived D-propation cubes, they use the structure of the CHDL description for test generation. They consider procedural (sequential) and non-procedural (concurrent) interpretations of an HDL description. For procedural interpretation, they use a method similar to Su's symbolic execution [11].

R. Khorram [32] worked on test generation from models described in a procedural HDL which executes once per clock cycle. Khorram breaks testing into several parts: testing a particular statement, justifying required local inputs, activating the HDL statement, and propagating statement result to an output. Testing the statement refers to selecting values to test the variables or operations in the statement. Justification, as in other test methods, involves finding primary input values to get the required values to the statement under test. Activating a statement refers to presetting any conditions necessary to execute the HDL statement (e.g. the control expression of an IF statement). Propagation, also as expected, involves determining how to get local test results to an output. Khorram's method does stuck-at tests, plus functional tests for operators.

Khorram's justification involves finding a statement to load a variable, if the variable is not a primary input. This can lead to other required values, if the variable is loaded from other variables. Propagation involves finding a statement to move the fault syndrome closer to an output, unless the fault syndrome is not at a primary output. Similarly, propagation can imply other required values which must be justified. Since conflicts may crop up, backtracking is used, and Khorram suggests using heuristics to select choices.

Chapter 3

Test Method

The method described here is a automatic test generation method for the chip level. The test method accepts an HDL description of a circuit, and uses a chip-level fault model to enumerate and sensitize faults. As opposed to more abstractly algorithmic methods, the test method directly examines the circuit description to justify and propagate values, as a test engineer might to generate tests manually. The method uses goals and goal solving, artificial intelligence problem-solving techniques, to represent and satisfy original, intermediate, and final test requirements.

Hardware Description Language Characteristics

The test generation method works from a data-flow description of a circuit in an asynchronous block-structured concurrent HDL. Block-structured means that conditional execution is described with IF and CASE statements, not with GOTO or other branch statements (as in a register transfer language). Concurrent (or non-procedural) means that statements execute in parallel, and that only

inputs and state variables are involved. (There are no temporary variables that can take on different values sequentially through the description.) Asynchronous means that the description "executes" continuously (or equivalently, each statement executes whenever any of its input variables changes). (A synchronous description would execute once for each system clock period and so cannot model asynchronous clocks.)

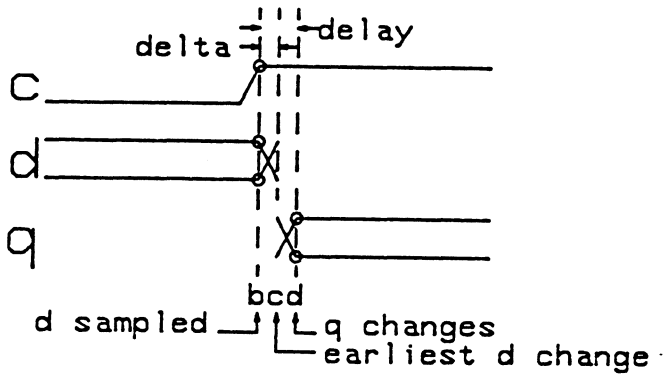
VHDL Subset

VHDL, the Department of Defense's VHSIC Hardware Description Language [21], was adopted as the input language to define HDL details. Since VHDL is such a powerful language, and since only concurrent data-flow descriptions are considered, only a small subset of VHDL is handled, and timing assumptions are made to fit our timing model.

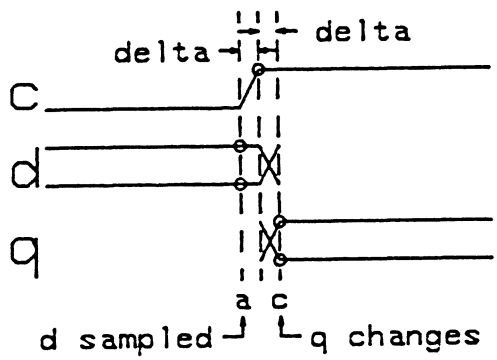
The method uses a simple period-based timing model, since it considers only functional tests and not timing tests. Primary inputs are set to values at the beginning of a time period and are held for the period. Clock inputs that require a rising edge require two time periods: one with a 0 immediately followed by one with a 1 (falling edges require a 1 followed by a 0). Primary outputs are sampled at the end of each period (before the next period's inputs are set). These periods are long enough for all signal propagation to complete and stabilize. This is the same style used in RTG [7], Marlett's EBT [8,9] and Khorram's work [32].

VHDL's delta timing is simplified to fit our time-period model. In VHDL, transitions are detected with the 'STABLE attribute [21]. The expression " x'STABLE " is false immediately after x changes, and true otherwise. 'STABLE is usually used as in

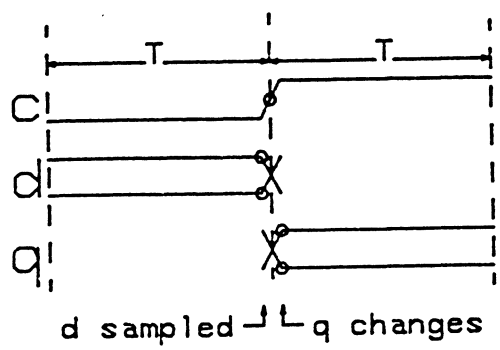
```
IF clk='1' AND NOT clk'STABLE THEN
   q <= d AFTER delay;
```



a) Original VHDL timing



b) True 'DELAYED timing diagram



c) New timing

Figure 1. VHDL Delta Timing to Time Period Mapping

to model a synchronous clock. In the example, a rising edge on clk would latch the value of d into q. Right after clk rises, the assignment statement executes. After the delay, q takes on its assigned value. (If no delay is specified, a infinitesimal delay of delta is used.) The timing is as in part a of Figure 1 on page 11. Note that as written the example takes the value of d right after the clock changes. Since with the propagation and hold times of real circuits, d will not change immediately if it is an output of another edge-clocked latch, an equivalent construction is:

```
IF clk='1' AND NOT clk'STABLE THEN
   q <= d'DELAYED AFTER delay;
```

which changes the timing to that of part b of Figure 1.

To fit the period timing model, the following is assumed:

- when a signal's value changes from one period to the next, 'STABLE is false for that signal
- for any object other than the clock, in the IF statement or in any statement in the THEN clause, the previous period's value is used
- any object assigned as the result of an edge clock (in the THEN clause) takes on its new value in the second time period

Thus, the assumed timing is as in part c of Figure 1. The statement is said to execute in the previous period, though the value is actually assigned in the second period (this is arbitrary; this particular choice reduces special cases in the implementation).

The VHDL subset includes only signal objects and not variable objects. (Signals have a constant value during one pass through the description, representing state variables. Variables can take on several values sequentially at different points in the description. Thus, the value of a variable is not uniquely specified by the time (or which simulation pass); it must be qualified by position in the code.)

Objects may be bits or bit vectors. Values are 0 and 1, plus X (don't-care) in test generation. (Z, high-impedance, is not implemented.) Many other types can be mapped into these two. Booleans can be translated to bits, and integers to bit vectors (especially in CASE statement selection expressions).

Statement may be IFs, CASEs, or simple assignments. (This excludes all loops, and any function or submodule declarations. Loops are sequential, and require variables, which are not implemented.)

The test method assumes that all assignments to a signal refer to the same driver for the signal. Single drivers are assumed to avoid bus resolution functions, which are commonly expressed algorithmically with a loop, and which typically use Z values which are not currently implemented. This assumption is equivalent to placing all assignments to a given object within one process statement, sensitive to everything.

The subset includes most basic operations on data: boolean operations AND, OR, XOR, NOT, and EQV on bits and bit vectors, and EQ and NOTEQ operations for bit vectors; unsigned arithmetic operations ADD, SUB, less than and less than or equal for bit vectors; concatenation of bit vectors in expressions; and slices (subvectors) and elements of objects and expressions, except for slices of destination objects or using a slice as an edge clock.

Because of the time period model, all sequential logic must be modeled explicitly, with using cross-connected gates or other combinational feedback. This is a minor restriction, since a chip-level model would not have such feedback anyway.

Appendix A, the User's Manual, contains other details of these restrictions and assumptions.

HDL Terminology

The following terms refer to various elements of the VHDL [21] description of a circuit:.

literal: a value appearing right in the code; in " x < = '1' ", " '1' " is a literal.

object: VHDL's term for simulation variable; VHDL has signals and variables as classes of objects; only signals are used here.

expression: object(s) and/or literal(s) combined with micro-operations; an expression may be arbitrarily complex, or may consist of a single object or single literal.

subexpression: an expression within an expression, as an argument to a micro-operation; in "IF a AND (b OR c) THEN...", subexpressions are " a ", "(b OR c)", " b ", and " c ".

source expression: the expression in an assignment statement, whose value is loaded into the destination object when the assignment statement executes.

destination object: (of an assignment statement): the object (to the left of the < =) set when the assignment statement executes.

control expression: the conditional expression of an IF statement or the selection expression of a CASE statement, which controls which THEN, ELSE, or WHEN clause executes.

conditional expression: the control statement of an IF statement; if its value is true, the statements under the THEN clause execute; if false, the ELSE executes.

selection expression: the control expression of a CASE statement; if its value matches a value in a WHEN phrase list, the statements under the WHEN clause execute.

clause: a set of statements under one branch of a control statement.

THEN clause: the statements under the THEN of an IF statement.

ELSE clause: the statements under the ELSE of an IF statement.

WHEN clause: the statements under a WHEN of a CASE statement.

WHEN phrase: the value(s) after the WHEN keyword which specify what value(s) is needed to execute the associated WHEN clause.

parent statement: the control statement controlling a statement.

under: controlled by; a statement is under a (control) statement if it is controlled by that statement.

under 'STABLE: under an IF statement with a 'STABLE; controlled by an edge clock.

top-level statement: a statement not under any other statement; top-level statements effectively execute all the time, and do not need any conditions set to execute.

Fault Models

Gate-level stuck-at fault are insufficient at the chip level. A correspondingly high-level fault was needed for chip-level test generation.

Original Fault Model

Chip-level fault models were first presented by J. R. Armstrong [23]. He suggested micro-operation and control faults. For a micro-operation fault, a micro-operation in the description fails to some other operation. As a heuristic, the dual of the operation was used, where defined. Though unusual, this choice has some hardware correspondence [24]. For a control fault, a branch functions incorrectly. For example, a conditional branch becomes inverted or unconditional, or an unconditional branch fails.

A. K. Gupta used these fault models for chip-level test generation [24]. He used GSP [17, 18], an assembly language-style HDL, for fault simulation. Using manual test generation for a 3700-gate signal processing chip, these chip-level faults gave 88.6% coverage of gate-level stuck-at faults [24, 34]. Other fault simulation with GSP yielded coverages near 90% [34, 35].

Modified Fault Model

Gupta's control faults were modified to fit the constructs of a block-structured HDL [35]. The resulting faults are:

STUCKTHEN and STUCKELSE: An IF statement becomes stuck one way or the other, as if the conditional expression became stuck true or false.

DEADCLAUSE: A WHEN clause in a CASE statement fails to execute when it is selected.

(A STUCKCLAUSE fault might seem more consistent, but DEADCLAUSE should give better coverage. To test for STUCKCLAUSE faults on n clauses ($n > 2$), only 2 tests are needed: execute clause 1 (to detect stuck at clause 2 through stuck at clause n), and execute clause 2 (to cover stuck at clause 1). A test for a dead clause fault tries to execute the faulty clause, so it takes n tests to test n clauses. Using n tests should yield better coverage then 2 tests).

ASSNCNTL: For an ASSNCNTL (assignment control) fault, an assignment statement fails to execute (i.e. the assigned object is not changed). This addition was to improve coverage; a pure transfer (with no micro-operations) would not be tested directly without this fault type.

MICRO-OP: A micro-operation fails to some other operation. Each micro-operation may have several fault modes.

Results of Modified Fault Model

Additional experiments were done to verify this modified fault model [25]. Models for 11 small/medium-size circuits (12 to 91 gates, 44 average; plus 0 to 16 flip-flops, 6 average) were written in GSP2, a block-structured language [19]. Using this fault model, tests were derived manually from the chip-level models using the chip-level fault models. These tests were run on gate-level models of the same circuits, yielding an average coverage of 92.4%. Thus, the modifications seem valid.

General Approach to Solving

The chip-level test method is a path-tracing method that works directly from the HDL description of a circuit. After sensitizing the fault, path-tracing test generation must propagate the fault syndrome to an output and justify any needed interal values back to inputs. The particular HDL constructs in a description are used directly to justify and propagate values.

In a gate-level model, the inputs and state variables determine the outputs, through internal node values. This holds for chip-level descriptions, except for what constitutes a node. In a gate-level mode, nodes (or nets) are gate inputs and outputs. In an HDL description, nodes correspond to objects, micro-operations, and whether statements execute (particularly, those statements under control statements dependent on expression values). Chip-level test generation works with these virtual nodes.

Justification: Justification involves working backwards to set inputs such that internal node values are satisfied. Khorram [32] separates justification of object and expression values from activation of statements. Our method considers these all justification.

Justifying expression values parallels combinational gate-level justification. An expression may consists of a micro-operation and its argument expressions. Justifying the expression's value requires selecting values for the arguments such that the micro-operation computes the desired value, much as gate inputs are selected to justify the gate output value.

To justify a value in an object, the circuit description is searched for an assignment statement to use to load the value into the object. The statement must be executed, and its source expression must be set to the value to be loaded. These requirements must be justified in turn. In general, the object could be loaded in a time period previous to when it is needed. The choice of when to load the value is separate from the choice of which statement to use.

To justify the execution of a statement, any IF or CASE statements controlling the statement must be executed, and any control expressions must be set appropriately, and justified in turn.

Propagation: Propagation involves using the fault syndrome at some place to affect another object, eventually moving it to an output.

Propagating an object value is similar to gate-level propagation of a state variable with fanout. Since the object can be used in expressions, an appearance in an expression is selected, and the fault syndrome is propagated through the expression. Also, since the object may hold its value over time, the time to observe the value must be chosen.

Propagating fault syndromes through micro-operations in expressions parallels propagation through a gate with a fanout of one. The value of the micro-operation's other argument is selected to pass a fault syndrome as the result of the micro-operation.

For an assignment statement source expression, the fault syndrome can be propagated to the assigned object by executing the statement. For a control expression, the faulty value affects execution (i.e. the good value executes one clause and the faulty value executes another clause). This execution difference is detected by its effects on an object assigned different values by assignment statements within the affected clauses. Expressions in these assignment statement may have to be set to ensure that the affected object gets different values in each case.

Sequentiality and Time Choices: Since HDL descriptions are in general sequential, justification and propagation involve choosing when to load objects. The time choice strategy is to assign each load event its own time designation, determine how to accomplish the load, and then decide exactly when to do the load relative to other load events. The time designation is called a base time. Constraints such that one load must precede another are expressed by saying that the former's base time be less than the latter's. At some point, the absolute offset between them is decided.

Goals and Data Structures

The test method is implemented with goals. Goals are an artificial intelligence problem-solving technique used to break a problem into small pieces. Goals are used to represent and manipulate the requirements of a problem. The test method uses goals to represent basic sensitization requirements, derived justification and propagation requirements, and final test vector specifications. There are eleven types of goals to represent these requirements:

1. Preload goals:
   a. VIO (Value in Object): Need **value** in **object** at **time.**
   b. VIE (Value in Expression): Need **expression** equal to **value** at **time.**
2. Execution goals
   a. EXEC (Execute): Execute **statement** at **time.**
   b. DNE (Do Not Execute): Don't execute **statement** at **time.**
   c. EXG: (Execute Given): Execute **statement** at **time** given that **set of statements** execute.
3. Observation goals:
   a. OBSOBJ (Observe Object): Observe value in **object** at **time** for **good value** or **bad value**
   b. OBSEXPR (Observe Expression): Observe **(sub)expression** at **time** for **good value** or **bad value**

c. OBSEXEC (Observe Execution): Observe execution of clauses of **statement** at **time**, expecting **good** or **bad** clauses to execute.

4. Preserve goals:

   a. DND2 (Do Not Disturb, Multiple): Preserve **value** in **object** loaded at **time1** until **time2**.

   b. DND1 (Do Not Disturb, Single): Preserve **value** in **object** at period **time**.

5. TR (Time Relation): Specify time relation **time1** $\leq$ **time2** or **time1** = **time2**.

VIO and VIE goals are used to load objects and set expression values. EXEC and EXG goals are used to activate statements. OBSOBJ and OBSEXPR specify where a fault syndrome currently is. OBSEXEC specifies a fault-induced difference of execution to be detected. TR goals represent constraints on when to load object or to propagate results. DND2, DND1, and DNE goals are ensure consistency by ensuring that loaded values are not overwritten.

Times are represented as a base time and an offset, in the form tn$\pm$m. Base times refer to arbitrary time periods. The offset refers to time periods relative to the base time. For example, t1 + 3 refers to the third period after period t1. Until decided and explicitly specified, there is no implied relation between different base times; they may be the same period or different periods.

The value of an object at a certain time is referred to by specifying the object name and the time. For example, x@t3-1 refers to the value of object x at time period t3-1.

There are two representations for expressions. One refers to expressions and subexpressions within the circuit description, for example, "the expression in statement 5," or " the left subexpression of the right subexpression of statement 7." (The latter example could refer to "x AND y" in "(a AND b) XOR ((x AND y) OR z).") Position codes are used, as in "5-" or "7RL", for example. The left/right nomenclature is based on the parse tree of an expression. The position code method can specify where to expect a fault syndrome, for OBSEXPR goals, or where a faulty micro-operation is. This is similar to numbering nodes in a gate-level circuit.

However, the position code method can only refer to expressions and subexpressions existing in the description. The method must be able to construct new expressions which might never occur in the description. For example, to make x and y different, one could require that "x $\neq$ y" be true,

but "x ≠ y" might never occur in the description. To represent such expressions, a parse tree representing the expression is constructed and manipulated. To be able to refer to different objects at different times, times are associated with individual objects instead of the whole expression, for example in "x@t0-1 ≠ y@t1." Associating times with objects in expressions is called *time-tagging* the expression. This second form of expressions is used in VIE expression fields, and in OBSOBJ and OBSEXPR value fields.

Actions

The test method uses the chip-level fault models to list faults and to select "basic tests" for faults.

Listing Faults

Faults are listed by traversing the circuit description and listing faults for each HDL item. IF statements are listed for STUCKTHEN and STUCKELSE tests. CASE statement clauses are listed for DEADCLAUSE tests. Assignment statements are listed for ASSNCNTL tests. Each expression in the description is traversed to find micro-operations to test. For each micro-operation found, its fault cases are looked up and listed for testing.

Selecting Basic Tests

Basic tests are the sensitization requirements for faults, and are represented with goals. Basic tests for each fault are derived from the fault model. Base time t0 is chosen to refer to the time the internal test occurs (as opposed to when results are propagated to a primary output).

STUCKTHEN/STUCKELSE Test

A STUCKTHEN fault is sensitized by trying to execute the ELSE clause, by executing the IF statement and setting the conditional expression false. An EXEC goal specifies to execute the IF statement. A VIE goal sets the expression false: the IF statement control expression is copied, is time-tagged with t0 (the time of the basic test), and is set false with the VIE goal.

The fault is detected by observing which clause executes. An OBSEXEC goal specifies to detect whether the statement's THEN clause (indicating faulty operation) or ELSE clause (indicating good operation) executes at t0.

A test for STUCKELSE is similar, with the expression set true, to try to execute the THEN clause, and with the OBSEXEC good and bad clauses reversed.

DEADCLAUSE test

A DEADCLAUSE fault is sensitized by trying to execute the clause. A WHEN clause is executed by executing the CASE statement and having the selection expression equal to one of the values given in the WHEN phrase (usually a single value). An EXEC goal specifies to execute the CASE statement at time t0. To set the conditional expression, an expression of the form "(e = v1) OR (e = v2) ... (e = vn)" is made, where e is a copy of the selection expression, and v1..vn are values in the WHEN phrase, any of which selects the clause. This expression is time tagged with t0, and

specified to be true using a VIE goal. Thus, the selection expression will be set to one of the WHEN phrase values.

The fault is detected by observing whether the clause actually executes. An OBSEXEC goal specifies the CASE statement, the time (t0), and specifies that good execution would be the tested clause and bad execution would be nothing. (That is, nothing under the CASE statement would execute.)

ASSNCNTL Test

An assignment control fault is sensitized by executing the statement with an EXEC goal, and detected with an OBSEXEC goal. The OBSEXEC goal specifies that the statement will execute for good operation, and will not for bad operation. (This OBSEXEC will observe execution by trying to load a value using the assignment statement.)

Micro-Operation Test

A micro-operation fault is sensitized by setting the arguments of the operation such that the result of the good micro-operation differs from the result of the bad micro-operation (i.e., $f_{good}(arguments) \neq f_{bad}(arguments)$). To simplify this selection of values, a condition is given in a lookup table for each micro-operation fault case. This condition is expressed as an expression of the micro-operation's arguments which must have a required value. For example, for " <arg1> AND <arg2> " failing to " <arg1> OR <arg2> ", the condition is <arg1> ≠ <arg2>. The lookup table rules construct an expression using the micro-operation's arguments. This expression is time-tagged with t0 (test time), and put in a VIE goal with the required value. When the VIE is solved, necessary conditions will be satisfied.

The micro-operation's result is detected with an OBSEXPR goal. The OBSEXPR specifies the position of the micro-operation, and specifies the good and bad values. The expected good and

bad results are simply the good function of the arguments and the bad function of them, respectively. The lookup table specifies this, or may specify a fixed value, if already constrained by the input condition.

Solving for a Test Vector

The test generation is done in the goal-solving phase. This solving phase takes the basic test goals and recursively solves them into goals representing the test vectors for a fault. Goals are solved by breaking them into subgoals which will accomplish the original goal, using rules for each goal type which take into account the circuit description.

The solving phase maintains an unsolved goal stack. Basic test goals are initially placed on this stack. As goals are picked off the stack, they examined. A goal may be primitive, inherently solved, solvable, or unsolvable.

Primitive Goals: A goal is primitive if it specifies a needed input pin value or an expected output pin value. Obviously, input pins can be directly controlled and output pins can be directly observed, so primitive goals are added to the test vector.

Inherently Solved Goals: A goal is inherently solved if nothing needs to be done. A goal requiring that x = x be true would be inherently solved; whatever x is, x = x is true. Nothing is done for inherently solved goals, since they are solved.

Solvable Goals: A goal is solvable if it can be accomplished by specifying several other goals. Most goals are solvable with subgoals. For example, a VIE goal requiring a AND b = 1 can be solved with VIE a = 1 and VIE b = 1. For solvable goals, the new subgoals are added to the unsolved goal stack, so they can be solved in turn. Frequently, there is a choice of how to solve a goal.

When one choice is made, the other options are noted so they can be tried later in case the original choice leads to a conflict.

Unsolvable Goals: A goal is unsolvable if there is no way to satisfy the goal. An example would be a VIO goal requiring one value in an object when the object has already been assigned a different value. This is also called a conflict. When an unsolvable goal is encountered, solving "backtracks" to the most recent choice point and selects a new option.

Backtracking

Backtracking is a technique to back up and try other options after encountering a conflict. At each point at which there is a choice of what to do, one option is chosen, and the others are saved. At a conflict, execution backs up to the most recent choice (restoring the state to what it was before that choice), and the next remaining untried option at that choice point is chosen. If no options remain, backtracking proceeds to the previous choice point.

In the test method, there are choices about how to solve goals. There is a choice of when to load an object or propagate a value. There may be multiple paths usable to load an object. There may be a choice of object values which will give the needed value for an expression.

Conflicts occur when a goal cannot be solved. Two goals may require conflicting (i.e. different) values in the same object at the same time. An execute goal may clash with a do not execute goal for the same statement at the same time. An unsolvable goal is considered a conflict.

Solving Goals

The various types of goals are solved as follows:

Solving VIO (Value in Object) goals: A VIO goal specifies that some object needs a certain value at some time. If the object is a primary input, the VIO is primitive, and becomes part of the test vector.

If the object is not a primary input, the object must be loaded, using an assignment statement, by the time the value is needed. A new base time is allocated to represent the time period in which the object is actually loaded, and is specified to be less than or equal to the time at which the value is needed. The statement must be executed, and its source expression must be set to the value given in the VIO goal. For an asynchronous statement (not under a 'STABLE), an EXEC goal specifies to execute the statement at the new base time, and a VIE goal specifies the needed value for the source expression at the new base time. For a synchronous statement (one controlled by a 'STABLE), the VIE applies in the previous time period (i.e. the new base time minus one period), before the clock edge. With the convention for when a synchronous statement executes, the EXEC goal also specifies this previous period. Finally, a DND2 goal specifies to preserve the value from the new base time (when it was loaded) to the time in the VIE goal (when it is needed).

Solving VIE (Value in Expression) Goals: A VIE goal requires a given expression of time-tagged objects, literals, and micro-operations to have a certain value. VIE goals are solved by selecting values for objects such that the expression has the needed value. There are several VIE cases.

If the expression is a just a literal value (i.e., no objects or operations) which is the same as the needed value, the VIE is inherently solved.

If the expression is a literal value different from the needed value, the VIE is unsolvable.

If the expression is a single object, the VIE is solved with a VIO requiring the value in that object, at the time with the time-tagged object. This case simply rewrites the VIE as a VIO.

The main case is when the expression consists of a micro-operation and its arguments. In this case, values are selected for the arguments to give the needed result. Since the arguments are in general expressions, a VIE goal specifies the selected value for each argument expression. There may be several combinations of values to give the needed result, so when one is chosen, the others are noted for backtracking.

Solving EXEC (Execute) Goals: An EXEC goal specifies to execute a statement at a particular time. Since EXEC goals are a special case of EXG goals, they are solved in terms of EXG goals (see "Solving EXG Goals (Execute statement given statement list)", below).

Solving EXG Goals (Execute statement given statement list): An EXG goal is used to execute a statement assuming that some higher-level statements already execute. EXGs are use for solving OBSEXEC goals, and for solving EXEC goals.

EXG goals are used mainly in solving OBSEXEC goals. An OBSEXEC goal specifies which clauses of a control statement execute for good and faulty operation. To detect execution, an assignment statement within the clause must execute to affect some object. The assignment may be nested under other IFs or CASEs, so values must be set to execute that assignment. For example, in

```
s1:  IF a AND b THEN
s2:     IF c OR d THEN
s3:        x <= '1';
        ELSE
s4:        y <= '0';
        END IF;
s5:     z <= '1';
     ELSE
       . . .
     END IF;
```

consider a test for statement s1 stuck at ELSE. Statement s1 would be executed with an EXEC goal, the control expression " a AND b" would be set true with a VIE goal to try to execute the THEN clause, and an OBSEXEC goal would specify good execution as the THEN clause and bad execution as the ELSE clause. If s5 is chosen to detect execution (by setting z), then s5 will execute whenever the THEN clause executes. However, if statement s3 is chosen to detect execution of the THEN clause (by changing the value in x), the expression "c OR d" must be set true.

EXG goals are used to set up such conditions needed to execute statements under a clause. In this example, an EXG goal would specify to execute s3 given that s1's THEN clause executes.

The THEN clause is specified as s2 and s5, since execution of either s2 or s5 indicates that the THEN clause executed. (An EXEC s3 goal would be wrong, because s3 is not to be executed all the time, just if s1's THEN executes.) Solving this EXG would set " c OR d " true. If s5 is used, the EXG goal would be used to execute s5 given that s2 and s5 execute. In this case, no further goals are necessary.

EXG goals are also used to solve EXEC goals. If an EXG goal's given list consists of top-level statements (which always execute), solving the EXG goal will ensure that the statement executes. Thus, EXEC is a special case of EXG. (That is, the EXG goal executes the statement if the top-level statements execute. The top-level statements execute under any conditions. Therefore, the statement will always be executed.) An EXEC is solved by making an EXG goal, specifying the top-level statements as the assumed list.

EXG goals are solved as follows:

If the needed statement is in the list of given statements, the EXG is inherently solved (e.g. if it says execute s2 assuming that s1, s2, and s3 execute, s2 can be assumed to execute, so it is inherently solved.) This is the termination case.

Otherwise, the parent of the needed statement must be executed, and the clause containing the needed statement must be sensitized. The parent is executed with an EXG goal, specifying the same given statements. The clause is sensitized with a VIE goal. If the statement is in the THEN clause of an IF statement, the VIE sets the conditional expression true. If in the ELSE clause, it sets it false. If in a WHEN clause of a CASE statement, The VIE sets the expression "(e = v1) AND (e = v2) ... (e = vn)" true, where e in the selection expression and v1..vn are the values in the WHEN phrase, so that the selection expression has one of the WHEN values. This activates the clause if the CASE executes.

Solving DNE Goals (Don't execute statement at time): A DNE goal specifies to ensure that a statement does not execute at some time (e.g., to avoid changing a value in an object).

A DNE for a top-level statement is unsolvable, since a top-level statement always executes.

A statement can be avoided by avoiding its parent statement, or by desensitizing its parent clause (the parent statements's clause which contains the statement to avoid). The parent is avoided with a DNE for the parent statement. If that fails, the parent clause can be desensitized by setting the control expression's value with a VIE goal. For a statement under a THEN, the conditional expression is set false; for ELSE, it is set true. For a statement in a WHEN clause, the selection expression for the CASE statement must be set so that its value is not one of the WHEN phrase values which would select the WHEN clause. To specify this requirement, the expression " (e≠v1) AND (e≠v2) ... (e≠vn) " is set true, where e is the time-tagged control expression and v1..vn are WHEN values which could select the clause. This expression will be true only if the selection expresion value is *not* one of the values for the WHEN clause. Thus, setting the expression true will ensure that if the control statement executes, the WHEN clause containing the statement will not execute.

Note that a DNE is the negation of an EXEC goal. An EXEC goal executes the parent and sensitizes the parent's clause; a DNE goal avoids the parent, or desensitizes the parent clause.

Solving OBSOBJ Goals (Observe object value at time1): An OBSOBJ goal specifies to observe the fault syndrome in some object at some time, with the the expected good and faulty values.

An OBSOBJ is primitive if the object is an output. Obviously, a fault syndrome at an output is directly observable. Such an OBSOBJ becomes part of the test vector.

If the object is not a primary output, the object's value must be observed indirectly by propagating the fault syndrome towards an output. The object value may be used in a source expression in an assignment that loads another object, or may be used in a control expression to affect execution. A new base time is selected as the time to use the value in the object and a TR goal specifies that this new time be equal to or after the time the fault syndrome appears in the object. An OBSEXPR is used to further propagate the fault syndrome within the expression using the object. The OBSEXPR refers to the position of the object within the expression, the time, and the good and faulty values. A DND2 goal specifies not to disturb the object's value until it is observed.

Solving OBSEXPR Goals (Observe expression value at time1): OBSEXPR specifies a particular (sub)expression whose value to observe. This is equivalent to knowing a gate has a D or $\overline{D}$ on it, and trying to propagate the value through combinational logic. The value in a subexpression is propagated up through expression operations, and then observed. There are several cases for OBSEXPR solving:

If the expression is the argument of a unary micro-operation, the propagated results are simply f(good) and f(bad), where f is the micro-operation. A new OBSEXPR is issued, referring now to the micro-operation, at the same time, and with the new good and bad values.

The binary case is a more complicated. The other argument of the micro-operation may need to be constrained. This constraint is expressed with a VIE goal, specifying a value for a symbolic expression constructed using the other argument. As in the unary case, the resultant values must be computed as a function of both the fault syndrome and the other argument. Finally, a new OBSEXPR is issued, pointing to the micro-operation, and specifying the newly-computed good and bad values.

If the expression to be observed is not an argument of a micro-operation, it is an assignment statement source expression or a control statement control expression.

For an assignment source expression, the fault syndrome can be loaded into some other object. An EXEC goal is used to execute the assignment statement, and an OBSOBJ goal specifies to observe the fault syndrome now in the object. (If the assignment statement is under a 'STABLE, the OBSOBJ specifies the following time period, since the value of the source expression before the clock edge will only show up in the destination object after the clock edge.)

For a control statement, the good vs. bad values affect which clauses execute. In an IF statement a good vs. bad value of 0 vs. 1 becomes ELSE vs.THEN, and 1 vs. 0 becomes THEN vs. ELSE. An OBSEXEC for the control statement specifies THEN vs. ELSE or ELSE vs. THEN as good vs. bad execution. For a CASE statement, the good value will select one clause, and the bad value will select another. An OBSEXEC specifies these clauses.

Solving OBSEXEC Goals (Observe execution of clauses of statement): OBSEXEC goals are used to observe the execution path, which cannot be observed directly, by observing its effects on some object. An OBSEXEC specifies a control statement and two of its clauses (one possibly empty) to check.

First, one of the objects loaded by statements under either of the two clauses is picked (alternates are saved for backtracking). Next, a statement assigning to that object is picked from each clause. These statements are used to load one value if the good clause executes and another if the bad clause executes. In a variation, only one statement from one of the clauses is picked, and no statements in the other clause are executed. Thus, the old value will be changed if that clause executes. The choice of statements is also subject to backtracking.

When two statements are selected, a VIE goal is used to specify that the source expression be different, by requiring e1 $\neq$ e2, where e1 is one statement's source expression and e2 is the other's. An EXG goal for each ensures that each executes if its respective clause executes (the statements may be nested under other control statements). An OBSOBJ goal checks the resulting value. (For one statement, the VIE sets the object's old value to be not equal to the value the statement would load.)

Solving DND2 Goals (Preserve value in object from time1 to time2): DND2 specifies to preserve a value in an object from a start period to an end period. Most DND2 goals are issued with two different base times; thus, they cannot be solved until the accompanying TR goal is solved and one base time is represented in terms of the other. A DND2 is solved with a DND1 for each period in the DND2 interval.

Solving DND1 Goals (Preserve value in object at time): DND1 specifies to preserve a value in an object for a single time period. A DND1 is solved by finding all statements assigning to the object, and using a DNE for each to avoid executing it. In case the DNE fails, a VIE is used to set the source expression of a statement to the value already in the object, so if the statement executes, it reloads the value already in the object, thus preserving that value.

The statement that initially loads the value in the object must not be disallowed. DND2 and DND1 goals also include this loading statement. When a DND2 goal for an interval is solved with DND1 goals for each period in the interval, the DND1 goal for the first period (when the object was loaded) has a first-period flag set. When a DND1 goal is solved with DNEs for each statement that could disturb the object, if the DND1 is for the first period, no DNE is issued for the loading statement.

Solving TR Goals (Timing Relations): Time relation goals specify constraints between different base times. There are two forms: (TR t1 ≤ t2) and (TR t1 = t2).

A (TR t1 ≤ t2) goal specifies that time t1 be less than or equal to time t2. Usually (but not necessarily), each time is a base time with zero offset. The TR goal is solved by picking an offset between the times that satisfies the constraint. For example, (TR t1 + 1 ≤ t2) could be solved with t1 + 1 = t2, t1 + 1 = t2-1, t1 + 1 = t2-2, etc. When an offset is selected, a TR = goal is issued to unify one base time to some offset from the other. The offset is arbitrarily limited to some maximum.

A TR = goal is a very special case. It specifies one base time in terms of the other. To "solve" a TR = goal, all occurrences of one base time in all solved and unsolved goals are substituted in terms of the other base time. This binds the base times together. If time substitution leads to any conflicts, the TR = is unsolvable, and leads to backtracking (usually to try a different offset from the TR ≤ goal).

Base Times and Substitution

Recall that base times are names for arbitrary time periods. The test base time is t0. As base times are created, the index is incremented (i.e. the next is t1, then t2, etc.) When two times are unified with a TR = goal, the substitution is in terms of the one with the lower index, (e.g. t2 and t5 -> t2 and t2±offset). Thus, all final times will be in terms of t0; the offsets may range from negative to positive, which may be unfamiliar, but time labels are arbitrary anyway.

Loop Cuts

Test generation for sequential circuits must deal with sequential feedback. Sequential feedback can lead to loops, in which an object is used to load itself, or the value in an object is propagated back to the object. Such loops waste effort, and can trap automatic test generation.

To sense and avoid loops, VIO and VIE goals, and OBSOBJ and OBSEXPR goals have an extra parameter. For a VIO or VIE goal, this parameter is a list of objects loaded from the current object or expression. The list specifies that the current VIO or VIE goal results from justifying values in these other objects. A register must not be loaded from itself, directly or indirectly. If the object to be loaded appears in the list, it means a loop was just completed. Under this condition, the VIO goal fails as unsolvable, to try another path. Solving a VIO goal adds the object to the path, and the VIE goal passes the path information on to the VIO goals it can lead to. OBSOBJ and OBSEXPR goals work similarly, checking that the fault syndrome is not loaded back into any place it has been.

Fault Awareness

Test generation must keep track of the fault to ensure a correct test. This makes goal solving more complex. The goal solving rules given above assume correct operation, and must be modified to handle goal-solving correctly in the presence of a fault. Most modifications avoid using the faulty item to preload before the test, or to propagate results afterwards. Without checks, using the faulty item to test itself could mask out the fault. However, these checks preventing use of the statement or micro-operation must be overridden to use the statement or micro-operation once to test it.

For a STUCKTHEN fault, statements in the ELSE clause cannot be executed, and no statement in the THEN clause can be avoided if the IF statement executes. STUCKELSE faults are similar. For a DEADCLAUSE fault, no statement in the clause can be executed. Unless overridden, EXEC solving checks that the statement is not directly in a clause which cannot execute.

DNE solving will only issue a DNE for the parent statement, and not try to desensitize the parent clause if the statement is in a STUCK clause.

For an ASSNCNTL fault, the statement cannot be used to preload or observe, since it may be faulty. Unless overridden, EXEC solving also checks that there is no ASSNCNTL fault on a statement.

Micro-operation faults usually are not a problem, but a faulty micro-operation must not be used to preload or propagate its own test. When an expression is copied from the description (i.e. to make a VIE goal), the faulty operation is marked. So marked, it cannot be used, so it is never reused where it should not be.

Two-Phase Tests

One part of the fault model that causes problems is the ASSNCNTL fault. The standard test is to preload one value v1 into the destination object, try to load a new value v2 ($\neq$ v1) by executing the assignment statement with the source expression set to v2, and observe the resulting value in the object. If the assignment is the only assignment to the object, there is no other assignment with which to load the first value into the object. If the faulty statement is used to preload the first value, one cannot be sure the object is loaded, since the statement may fail. For example, if an object x has a value of 1 (random initialization), and one tries to load a 0 and then load a 1, the original 1 will be in x, and one will think the statement works. .

The solution is to do a "two-phase" test. In a two-phase test, value v1 is loaded and the object is observed for v1; then v2 is loaded, and the object is observed. (Actually, because of a limitation, this becomes a three-phase test: load v1, check v1; load v1 again, load v2, and check v2. Value v1 is loaded again because observing v1 may destroy the v1 in the object, for example, observing a bit in a serial-output shift register.)

Two-phase tests are implemented in OBSEXEC solving. Normally, OBSEXEC goal uses a VIE goal to specify to preload one value, another VIE goal to set the source expression, an EXEC

goal to execute the statement, and an OBSOBJ goal to observe the value. If the ASSNCNTL statement is the only assignment to the object, the OBSEXEC solving issues an extra VIE goal to load the value and an extra OBSOBJ goal to verify that value. These two goals are issued for a new, totally independent base time. (I.e., the new base time will never be constrained with a TR goal, and will never be substituted in terms of t0. This will effectively yield two tests which can be done in any order.)

Control faults can require two-phase tests, when all statements that load an object are under a faulty control statement. If all assignments to an object are under a faulty IF statement, or in a dead WHEN clause, OBSEXEC also initiates a two-phase test.

Enhancements

Several enhancements added to basic solving strategy speed up execution.

Simple Controllability and Observability

Crude controllability and observability estimates are used to select justification and propagation paths. These estimates are simple input and output distance measures. Input distance of an object is the "distance" from an input, measured as the number of assignments needed to get to the object. Input pins have distance zero, objects loaded from inputs have distance 1, objects loaded from distance-1 objects have distance 2, etc. Similar measures are used for output distance. However, this definition does not specify how to count objects in control expressions. They obviously affect how easily objects can be loaded, but are harder to count since they are indirectly related. Unspecified distances are counted as infinity for now.

When selecting an assignment statement to use to load an object to solve a VIO goal, the assignment with the most controllable source expression is chosen first. When selecting a use of an

object (in some expression) when solving an OBSOBJ goal, the one loading the object with the best observability is selected first. When picking an object to use to detect execution when solving an OBSEXEC goal, the most observable object is picked first.

Conflict Checks

Additional conflict checks are done to catch conflicts more quickly. The basic check is for two VIO goals for the same object at the same time requiring different values. This is sufficient for correctness, but is inefficient. One additional check is for an EXEC goal and a DNE goal at the same time for the same statement. (This would eventually be caught when selecting corresponding object values to set control expressions, but this saves the work and time spent selecting those values.) Other new checks are for unsolvable goals, such as VIE goals with impossible combinations of expressions and values, or DNE goals for top-level statements.

Conflict checks are significant because of when they are done. Subgoals returned from solving a goal are checked immediately, to catch conflicts or impossible goals then, instead of considerably later when the subgoals are examined for solving. Compatibility is also checked immediately after time substitution, since many conflicts appear then.

Object Value Substitution

Related to conflict checks is the substitution of known object values in expressions. Substitution is done to simplify VIE expressions, when the value of any object in the expression is known, to determine that the VIE goal is unsolvable or inherently solved. Also, since OBSOBJ and OBSEXPR good vs. faulty values are represented as expressions of argument values, these expressions are substituted to resolve them to literal values by the time all goals are solved. Known object values are substituted in goals when the goals are returned as subgoals, and again when each

goal is solved (to substitute any values decided in between). This substitution corresponds to (partial) forward implication.

Solved and Decided Goal Lists

Instead of a single goal stack, there are actually three goal lists: the unsolved list previously described, the solved goal list, and the decided (but not yet solved) goal list. The unsolved list keeps tracks of goals which must be solved. The solved goal list consists of goals already solved (primitive, inherently solved, or solved with subgoals). The decided goal list consists of all goals (solved and unsolved). It is called the decided list because it lists all goals decided to be done, regardless of whether they are solved or unsolved.

The solved goal list is used to catch new goals that have been solved already. For example, if a VIO goal for some value in an object at some time was solved, and later for another reason another VIO goal requires the same value in the same object at the same time, the second VIO goal is considered inherently solved, since it was already solved. Immediately before being solved, a goal is checked against the solved list.

The decided goal list is used for enhanced conflict checks. Originally, a goal was checked for conflicts only against previously-solved goals. This was correct, but inefficient. Consider the case where a goal is solved with subgoals and one subgoal is incompatible with some solved goals. Backtracking should begin at this point, but would not begin until solving got to the the incompatible goal. Subgoals, being decided even before being solved, are put in the decided list (along with solved goals) so everything decided is checked for conflicts, not just what has been solved.

Another use for the decided goal list is for object value substitution. All values decided, not just those which have been justified, are substituted to catch conflicts or discover inherently solved goals.

DND2 VIO checks

Another enhancement is embedded in DND2 goal solving. Recall that a DND2 goal is used to preserve the value in some object over an interval of time periods. The DND2 goal becomes a DND1 goal for each period, and each DND1 goal usually becomes a DNE for each statement that could assign to the object. In case of a bad time choice, such that one value is loaded, then a second value is loaded, and then the first value is used, there will eventually be a clash when the EXEC goal loading the second value conflicts with a DNE (from the DND2 from loading the first value), preserving the first value. To catch such conflicts early, the DND2 creates a temporary VIO goal for each period in the interval, and checks these VIOs against any existing VIOs. Thus, conflicts can be detected when solving the DND2 (which is the first thing solved after a time choice), instead of later when the DND1s are solved with DNE's. (These VIO goals should be permanently added to the solved list, in case conflicting VIOs pop up later.)

Unsolved Goal List Sorting

A major enhancement is sorting the unsolved goal stack (it becomes the unsolved goal list). The aim is to catch conflicts more quickly by solving goals which lead to conflicts first, and to reduce work repeated near choices, especially involving deferred time choices.

When a decision is made, as much as possible should be done to find out if the decision is bad before making more decisions. (Recall that when solving an exponential-order problem, it is the choice points and conflicts that count. Most other improvements yield only linear improvement, which is insignificant.) Thus, if there are any goals to be solved which involve no choices, they should be done first. Next should be any goals related to the choice (since related goals are more likely to cause or reveal conflicts than non-related ones).

Solving non-choice goals first helps for another reason too. For example, if goal A has no choices, goal B has three choices, and conflicts can be detected only after solving both A and B,

doing A first is better. If B is done first, three options of B are tried, doing A for each one. If A is done first, and then B's three options are done, two solvings of A are avoided.

Thus, non-choice goals should be solved first, and then "related" goals. This is implemented by assigning a weight to each goal and then sorting by this weight. The intrinsic weight of a goal is computed as an estimate of the number of ways to solve it, with several special cases. Goals on the same base time are considered related, so goals are first grouped by base time, and then sorted by intrinsic weight. TR goals involve two base times; they are kept in their own group, the last to be solved.

Sorting is also used to defer some goals until they can be solved. DND2 goals are usually issued with two different base times, and cannot be solved until one base time is substituted in terms of the other; DND2s with one base time are sorted to near the top, and DND2s on different bases are sorted to the end. Some OBSEXPR goals cannot be solved until object values are decided and substituted. OBSEXPR goals with unsimplified expressions are sorted to be after VIE goals; when the appropriate VIEs are solved, values will be decided and substituted into the OBSEXPR expression. When the OBSEXPR is fully substituted, sorting will move it nearer to the top, so it can be solved.

Sorting is also used to gather constraints on object values. For a four-bit vector x, a VIE 10 in x[1 downto 0] leads to VIO XX10 in x, and VIE 0 in x[3] leads to VIO 0XXX. If the VIOs are solving separately, the work is done twice to find subgoals to load x. Also, these two VIOs must really occur at the same time (otherwise, if there are two separate loads, the second will undo the first). Test generation may try them at different times, failing until they happen to be at the same time. If the two VIO goals are combined to form VIO 0X10 in x, solving is done only once, and trying different time offsets is eliminated. (If they can not be combined it is a conflict.) To combine VIOs, the first one to occur must not be solved until the second one occurs (being returned as a subgoal solving a VIE goal). To satisfy this requirement, VIOs are sorted to be after VIE goals. Thus, a VIO is not solved until any other related VIOs exist; and when a VIO is returned as a subgoal, it is combined with any related VIOs.

Chapter 4

Results

Experimental Implementation of Method

The test method is implemented in about 7000 lines (including space and comments; about 280K bytes) of Virginia Tech ProLog [36] on a DEC VAX™ running VMS™. ProLog is a language designed for artificial intelligence work. As such, backtracking is built in. When a rule fails, ProLog automatically backtracks to the most recent choice, and continues solving.

Some Implementation Details

The circuit description must be translated into an internal form. The internal form consists of ProLog facts or rules representing things such as object declarations, the type of each statement, and the structure of expressions in statements. The translation is done by hand, since the research concentrated on testing and not on compiler issues such as input parsing and error checking. Only

two rules are not direct translations of the VHDL. One indicates state variables, for efficiency. The other indicates whether statements are controlled by a 'STABLE. Both of these are derivable from the VHDL source, so this hand translation does not invalidate the method. Appendix A describes this translation.

The internal form input is preprocessed to compute input and output distance measures and to pre-sort lists of object assignments. These sorted lists are used to select objects based on input and output distance. The internal form plus the preprocessor output is used by test generation.

Control faults and their tests are hard-coded. However, micro-operation faults are listed in a lookup table. Test parameters are given by rules for each fault case. (Stuck-at faults are implemented for comparative work and for program development because stuck-at faults can be easier to test. Vector-wide stuck-at faults are implemented (e.g. stuck at 000 and 111, not 0XX, X0X, XX0, 1XX, X1X, and XX1).)

A path-tracing method must be able to evaluate micro-operation results, and to backtrace and propagate through micro-operations. The program has "evaluate," "backwards," and "propagate" rules for each micro-operation. The backwards rule is most complicated; it must pick micro-operation argument values to give the required output value. (The backwards rules were not completely implemented; addition and subtraction results with Xs cannot be justified currently.)

The 'STABLE attribute is a special case. All micro-operations have one or two explicit arguments; 'STABLE has the given object and implicitly has the value of the object in the adjacent period. When 'STABLE of an object must be true or false at some time, as in object'STABLE = TRUE at some time or (object@time)'STABLE=TRUE, the 'STABLE is translated to not equals, as in (object@time) ≠ (object@previoustime). (Because of a time convention, it is actually the following time period.) Currently, propagation through a 'STABLE is not implemented.

The unsolved goal list is sorted explicitly for easy modification. Lookup rules based on goal type and parameters give the intrinsic weight numerically; the numbers are easily changed for experimentation. Grouping by base times is superimposed by adding offset based on the base time used in each goal. Goals are sorted on these final weights. The base time grouping is also easily modified.

To avoid reusing faulty micro-operations (for justification or propagation), the bad micro-operation is marked so that it cannot be reused. Any other micro-operations, literals, or objects in the expression can be used, as long as the faulty micro-operation is not. (This does not apply for the actual test, just for justification or propagation of the test.)

Some faults near 'STABLEs were ill-defined or did not make sense in the time-period framework and were excluded from test generation. For example, in

```
      . . .
s12    IF x='1' AND NOT x'STABLE THEN
s13       count <= ADD(count,"001")
      . . .
```

a STUCKTHEN for s12 does not make sense. In a strict VHDL interpretation, a STUCKTHEN means that s13 executes all the time, i.e. at every delta time, or at infinite frequency. Such a fault is obviously unrealistic. This fault cannot be tested by the test method, so such faults are excluded. The excluded faults include STUCKTHENs for IF statements containing 'STABLEs, most micro-operation faults on such IF statements' expressions, and some stuck-at faults in those expressions (the stuck-active faults; stuck-inactive faults are not excluded).

The basic tests for some stuck faults were invalid. A literal "001" in an expression would be listed for stuck-at-000 and stuck-at-111 faults. The test for a stuck-at-000 fault is to set the item to 111. However, a literal "001" cannot be set to 111, so such basic tests are invalid.

Circuit Models Used

Thirteen small models were used for test generation. Table 1 on page 43 lists these models. The VHDL and actual internal form for these models is given in Appendix B, "Circuit Models and Fault Lists". There were four combinational models (ADDER, ADDR2, FNTST, PRTY), two

Table 1. Sample models used

Code	Description
ADDER	4-bit adder, vector implementation
ADDR2	4-bit adder, bit implementation
CCNT2	2-bit controlled counter
CKTA	2 serially connected D F/Fs and AND gate
CKTCV	2-bit, 4-register multiplexor
CNTR	3-bit counter, bit implementation
CNTRV	3-bit counter, vector implementation
DFF	D flip/flop with set and clear
FNTST	reconvergent fanout test
PRTY	8-bit parity generator
SHFT	4-bit bidir. par. in shift reg., bit implementation
SHFTV	4-bit bidir. par. in shift reg., vector implementation
UARTO	2-bit UART transmit half, vectors

very small sequential models (DFF, CKTA), four small sequential models (CNTR, CNTRV, SHFT, SHFTV), and three medium models (CKTCV, CCNT2, UARTO).

ADDER

The ADDER model describes a simple four-bit adder at the vector level. "ADDER Description" on page 85 gives the VHDL and internal form description, and "ADDER Fault List" on page 86 gives the fault list.

ADDR2

The ADDR2 model describes a simple four-bit adder at the bit level, to exercise test generation for micro-operations. "ADDR2 Description" on page 87 gives the VHDL and internal form description, and "ADDR2 Fault List" on page 90 gives the fault list.

CCNT2

The CCNT2 model describes a controlled up/down counter with limit register. CCNT2 was used to exercise test generation for interdependent statements and signals. "CCNT2 Description" on page 93 gives the VHDL and internal form description, Figure 2 on page 96 shows the circuit, and "CCNT2 Fault List" on page 97 gives the fault list.

CKTA

The CKTA model consists of two serially-connected independently-clocked flip-flops and an AND gate; see Figure 3 on page 100. CKTA was used to exercise selection of loading times. "CKTA

Description" on page 98 gives the VHDL and internal form description, Figure 3 on page 100 shows the circuit, and "CKTA Fault List" on page 101 gives the fault list.

CKTCV

The CKTCV model describes a multiplexor with input, output, and control registers. CKTCV was designed to exercise the preloading of multiple registers through one input path, as might be needed to test a microprocessor by loading various registers through the input data bus. "CKTCV Description" on page 102 gives the VHDL and internal form description, Figure 4 on page 105 shows the circuit, and "CKTCV Fault List" on page 106 gives the fault list.

CNTR

The CNTR model describes a three-bit clearable counter at the bit level. "CNTR Description" on page 107 gives the VHDL and internal form description, and "CNTR Fault List" on page 109 gives the fault list.

CNTRV

The CNTRV model describes a three-bit clearable counter at the vector level. "CNTRV Description" on page 111 gives the VHDL and internal form description, and "CNTRV Fault List" on page 113 gives the fault list.

DFF

The DFF model describes a simple D flip-flop with clear and set. DFF was used to validate tests for asynchronous and synchronous clocks. "DFF Description" on page 114 gives the VHDL and internal form description, and "DFF Fault List" on page 117 gives the fault list.

FNTST

The FNTST model is used to show how reconvergent fanout causes failure. "FNTST Description" on page 119 gives the VHDL and internal form description, and "FNTST Fault List" on page 121 gives the fault list.

PRTY

The PRTY model describes an eight-bit parity generator, to exercise micro-operation faults. "PRTY Description" on page 122 gives the VHDL and internal form description, and "PRTY Fault List" on page 123 gives the fault list.

SHFT

The SHFT model describes a 4-bit parallel-in, serial-out, bidirectional shift register at the bit level. SHFT has serial output to exercise observation goals. "SHFT Description" on page 125 gives the VHDL and internal form description, and "SHFT Fault List" on page 130 gives the fault list.

SHFTV

The SHFTV model describes the same register as SHFT, but at the vector level. "SHFTV Description" on page 132 gives the VHDL and internal form description, and "SHFTV Fault List" on page 135 gives the fault list.

UARTO

The UARTO model describes the transmit half of a very simple 2-bit UART. UARTO was used to test preloading and propagation for interdependent signals. "UARTO Description" on page 137 gives the VHDL and internal form description, and "UARTO Fault List" on page 140 gives the fault list.

Test Generation Results

Table 2 on page 48 summarizes the results of test generation for sample models. The first row for each model is for chip-level faults (ASSNCNTL, STUCKTHEN, STUCKELSE, and MICROOP); the second row is for the vector-wide stuck-at faults done for comparison (STUCKDATA). The "Faults" column is number of faults listed by the program. The "NAF," or "Not a Fault" column is the number of listed faults which either were not faults (e.g. a literal '1' stuck at 1) or were undetectable (e.g. a dead clause which contained no statements, so failure makes no difference). NAF also includes invalid basic tests and redundant faults. The "Excl." or "Excluded" column is the number of unrealistic edge-clock timing faults which were excluded. "Tried" is the number of tests attempted, or "Faults" minus "NAF" minus "Excluded." "Done" is the

Table 2. Test generation results

Code	Description	Faults	NAF	Excl.	Tried	Done	Failed	Over.
ADDER	vector adder	3	-	-	3	3	-	-
		6	-	-	6	6	-	-
ADDR2	bit adder	32	-	-	32	32	-	-
		92	-	-	92	92	-	-
CCNT2	contr. counter	41	-	14	27	1	2	24
CKTA	F/Fs and AND	14	-	8	6	3	3	-
		34	2	12	20	8	12	-
CKTCV	latched mux.	17		4	13	12	1	-
		30	2	6	22	15	7	-
CNTR	bit counter	18	-	4	14	4	9	1
		44	5	6	33	15	18	-
CNTRV	vector counter	12	-	4	8	2	5	1
		26	5	6	15	2	13	-
DFF	D flip/flop	18	-	4	14	14	-	-
		38	7	6	25	25	-	-
FNTST	fanout test	5	3	-	2	1	1	-
		12	4	-	8	6	2	-
PRTY	parity gen.	9	-	-	9	9	-	-
		45	-	-	45	45	-	-
SHFT	bit shift reg.	30	1	4	25	21	2	2
		56	5	6	45	37	8	-
SHFTV	vector shift reg.	18	1	4	13	9	2	2
		52	3	6	43	36	7	-
UARTO	UART	27	-	9	18	4	6	8

Note: The first line of each model is for chip-level faults. The second line is for vector-wide stuck-at faults done for comparison.
Faults: Number of faults listed.
NAF (Not a fault): Faults which are not really faults (e.g. "1" stuck at 1, or empty dead clause)
Excl.: Excluded faults which disturb edge clocking.
Tried: Faults tried (Total - NAF - Excluded = Tried)
Done: Tests generated successfully
Failed: Generation failed because of fault model interpretation, test method, or implementation.
Over.: Generation failed because of ProLog overflow.

number of successful, correct tests. "Failed" is the number of failed or incorrect tests. "Over." is the number of test attempts which overflowed or crashed the ProLog interpreter.

Success was not limited to extremely simple circuits, though controllability of internal registers appears important. The combinational circuits (ADDER, ADDR2, PRTY) did well, with 100% success. Most tests for the shift registers (SHFT, SHFTV) succeeded, as did many for the latched multiplexor (CKTCV). The significant sequentiality of these circuits did not prevent test generation.

The main causes of failure were the fault interfering with justification and propagation, incomplete or incorrect implementation, inefficient implementation, and ProLog interpreter limitations.

The fault can interfere with the execution or use of statements needed to justify values. The test method keeps track of the fault to ensure that the fault does not inadvertently mask itself out. The faulty HDL item is generally avoided when justifying or propagating its own test.

Incomplete implementation prevented some tests. Justification of addition or subtraction results with Xs (don't-cares) was not implemented. Such values would be used but would not be justified. This made many CNTRV tests incorrect.

Some tests overflowed the ProLog interpreter. Size overflows do not directly reflect on the test method; a larger ProLog or new implementation is needed to retry these failures.

Sample Model Results

Highlights of test results for each model are given here. See Appendix B, "Circuit Models and Fault Lists" for fault lists with comments for individual faults.

ADDER

All chip-level and stuck faults for ADDER were correctly tested.

An idiosyncrasy of the fault model and test method appears in ADDER tests. The adder is combinational, but the test vector for an ASSNCNTL fault for statement s1 was:

```
Simple adder, vectors
(1 ASSNCNTL s1)
t\obj a         b         c
t0-1  0000      0000      -
t0+0  0000      1111      1111/0000
t3+0  0000      0000      0000/1111
```

This is an example of a two-phase test. The test is to see if statement s1 can load c. Signal c is loaded with 0000 (at t0-1), and then with 1111 (at t0), to see if the load works. Since 0000 was loaded with the potentially faulty statement, 0000 is loaded separately and checked (at t3). Such two-phase tests are similar to two stuck-at tests, but seem strange for a combinational circuit.

ADDR2

All tests were successful. Again, two-phase tests were used, though ADDR2 was combinational.

CCNT2

Very few CCNT2 tests succeeded. Most attempts overflowed the ProLog interpreter, or exceed the batch job time limit. Some tests failed because CCNT2 had the poorest observability. No internal signals were directly observable; each had to be observed by its effect on COUNT. The size of CCNT2 emphasized inefficiencies which were minor problems for other circuit models. Many tests overflowed or took too long and thus did not complete, because of inefficient justifica-

tion of object values. Recall that a new base time is allocated for each load event. Thus, each load is treated as a separate event, including loads which can only occur at the same time (i.e. their statements are related, such that if one executes, the other will execute also). Also, the loading of an object may be treated as two or more separate events if the value in the object is needed to solve two or more separate goals. The time for each load is chosen separately, so only combinations of time choices where the loads occur simultaneously will succeed. However, time choices are made one at a time, not together, so a bad time choice for one object may not be discovered for a while. Meanwhile, other choices may be made. Making any new choices before discovering an existing bad choice severely reduces efficiency as the test method tries alternatives which cannot succeed.

For CCNT2, the EN and DIR flags were typically involved. The flag would be set to solve one goal, but some other goals might require the same value and issue more VIO goals setting the flag. For example, EN might be true to increment the counter, but to preload the counter by clearing and incrementing, EN must also be true. In a final test, EN would be set true once, and not cleared until after all incrementing was done. However, multiple VIOs were issued to set EN true, and solved separately. Solving them separately, when ultimately they must map to the same time, causes conflict detection and backtracking to be very inefficient.

CKTA

Three of six CKTA non-excluded chip-level faults were generated. One failed because of fault avoidance; a STUCKELSE prevent preloading for its test. The other two faults backtracked too much because of poor goal solving orders. Two-phase tests create a new base time; goals on this second base time are independent of other testing. Because of the sorting order, these goals are solving before some of the regular goals, delaying the catching of conflicts.

CKTCV

CKTCV was successful, with 12 successful tests out of 13 attempted chip-level faults. Multi-period test vectors were generated to load the various registers needed for tests. For example, to test for statement s6 stuck at THEN (i.e. C loads from A even when it should be from B), the test vector was:

```
                Program Output                    Notes

Circuit V - Latched Multiplexor with Vectors
(13 STUCKTHEN s6)
t\obj   clock in   cmd   a   b   p   c
t0-6    0     00   01    -   -   -   -       -load 00 in A
t0-5    1     -    -     -   -   -   -
t0-4    0     11   10    -   -   -   -       -load 11 in B
t0-3    1     -    -     -   -   -   -
t0-2    0     11   00    -   -   -   -       -load 11 in P (for B->C)
t0-1    1     -    -     -   -   -   -
t0+0    0     -    11    -   -   -   -       -expect 11 from B if good,
t0+1    1     -    -     -   -   -   11/00           00 from A if bad
```

For an ASSNCNTL fault for statement s7 (i.e., C cannot be loaded from A), the test was:

```
                Program Output                              Notes

Circuit V - Latched Multiplexor with Vectors
(16 ASSNCNTL s7)
t\obj   clock in   cmd   a   b   p   c
t0-10   0     11   00    -   -   -   -       -load 11 in P (for B->C)
t0-9    1     -    -     -   -   -   -
t0-8    0     00   10    -   -   -   -       -load 00 in B
t0-7    1     -    -     -   -   -   -
t0-6    0     -    11    -   -   -   -       -load 00 in C from B
t0-5    1     -    -     -   -   -   -
t0-4    0     00   00    -   -   -   -       -load 00 in P (for A->C)
t0-3    1     -    -     -   -   -   -
t0-2    0     11   01    -   -   -   -       -load 11 in A
t0-1    1     -    -     -   -   -   -
t0+0    0     -    11    -   -   -   -       -try to load 11 from A to C
t0+1    1     -    -     -   -   -   11/00         expect 00 if it fails
```

The one failure was for a STUCKELSE fault. All assignment statements were under the THEN clause; so none could be execute to preload or propagate the test. Most stuck faults were successful. The failures were due to fault avoidance.

CNTR

Several CNTR tests failed because of loop checks. Loop checks are used to avoid getting caught in loops loading an object from itself. The problem is that loop checks are local to an object, not taking the global state of the circuit into account. For CNTR, if 011 is needed in q2, q1, and q0, loop checks would prevent loading those values. Consider q1. To count to 011, 010 must be the previous state. Thus, q1 must have a 1. But this 1 in q1 for 010 is used to load the 1 in q1 for 011. Local loop checks see that q1's value 1 is used to load q1 with a 1, even though the global state of the circuit changes from 010 to 011. Such local loop checks may be too restrictive. CNTRV did not have loop check problems, because the count was modeled as a vector; loop checks saw the whole state at once, not just individual bits.

CNTRV

Many CNTRV tests failed because of the incomplete implementation of BVADD. Vector values with Xs were not justified correctly (they were effectively forgotten), so tests using such values were invalid.

DFF

All DFF tests were successfully generated. However, in some cases, two-phase tests were used when not necessary (though they were valid). For faults 16 and 17 (ASSNCNTL for s9 and 10),

the faulty statement was reused to preload for the test. Two-phase tests were used, so the tests were valid. It would seem better to use the set or clear functions to load one value, and then test s9 or s10 by loading another value. The two-phase tests were used because two-phase testing can be initiated before all other assignment statements are tried.

FNTST

As expected, reconvergent fanout prevented some FNTST tests. If IN is stuck 1, IN is set 0 to sensitize the fault, and the fault syndrome of 0 vs. 1 is propagated to a, then to x (y is symmetric), and then through the AND. Signal y is set to 1 to allow propagation, and y is loaded from a, but a cannot be loaded with 1 since IN is set to 0, so test generation fails. However, since y will end up also holding a 0 vs. 1 fault syndrome, OUT will have the fault syndrome, and the fault could be tested. Because of the model structure, many other faults were redundant, and could not be tested either.

PRTY

All PRTY faults were successfully tested. As with ADDER and ADDR2, there was a two-phase test, even though PRTY is combinational.

SHFT

Testing for SHFT was quite successful. Of 25 chip-level faults tried, 21 succeeded. Two overflowed the ProLog interpreter. The remaining two failed because of avoiding the faulty item. Of 45 stuck faults tried, 37 were successful. The remaining 8 failed because of avoiding the faulty item.

SHFTV

SHFTV was also successful: 12 of 14 chip-level faults succeeded, and 37 of 43 stuck faults succeeded. Two chip-level fault overflowed, and 2 chip-level and 7 stuck faults failed because of fault avoidance.

UARTO

Only a few tests for the UART succeeded. Those that did were mostly simple tests, where the results of the fault appeared on the TXBUSY output line. There were some failures because of fault avoidance, as in other models. As with CCNT2, many tests overflowed or took too long because of the multiple load problem.

Summary

The implementation had trouble generating tests for functional and stuck faults for micro-operations in control statements, because the micro-operations were needed to execute statements to preload values, but fault avoidance prevented that reuse. Similarly, some STUCKTHEN and STUCKELSE faults prevented preloading of observation. For the larger sequential models, excessive backtracking was a problem. Tracing the execution showsthat the goal-solving order is not ideal. Bad choices are not discovered for a while, during which time other choices are made. These late choices backtrack a lot, until backtracking returns to the originally bad choice.

Execution Speed

No claim is made about the speed of this ProLog implementation. Small tests take a minute or two of CPU time; others can take 15 minutes to hours. These speeds are mostly due to the ProLog interpreter and the implementation within ProLog. Virginia Tech ProLog is slow; unrelated experience shows that even Apple II P-code Pascal is at about 10 times faster that Virginia Tech ProLog on a VAX. A ProLog compiler from the Virginia Tech Computer Science department is expected to be 100 times faster than the current interpreter. Implementing the method directly in, say, Pascal could increase the speed another 2 or 3 orders of magnitude. These are rough guesses; the point is that the speed of the test method cannot be evaluated currently.

However, the time difference between fast and slow tests shows that too much backtracking occurs. This usually occurs because the method tries many options to try to generate a test, but conflicts are not caught in time, and much work is done before the conflict is detected. The method as designed to be flexible, to try several alternatives when a first choice fails. In cases were no choices can succeed, all are tried before backtracking backs up the the decision which caused the problem. Imperfect conflict detection may let such bad choices by, catching conflicts only when they become more apparent. A nonoptimal order of solving goals may contribute to causing conflicts which are not detected quickly. This is a backtracking method; what will count is the exponential term of the speed from backtracking, not linear (and worse) terms in the current ProLog implementation.

Chapter 5

Analysis and Suggestions

As the results chapter shows, the test generation method has certain deficiencies. This chapter reviews weaknesses of the method and suggests possible solutions. Analysis is based on execution logs, showing how goal solving progressed, and on further interactive exploration of execution.

Fault Awareness and Reuse Avoidance

The test method keeps track of the fault to ensure that the fault is not used to justify or propagate its own test, so it cannot mask itself out. The faulty HDL item is generally avoided when justifying or propagating its own test.

For a faulty micro-operation, the micro-operation is not used to preload or observe values in its own test; it is just used once for the test. A micro-operation in a control statement can cause problems. The statements under the control statement are used to detect execution, and usually are also needed to preload for the test. Since the faulty micro-operation cannot be reused, the

control expression cannot be set to execute the statements, so preloading cannot be done. If no other statements can be used to preload for the test, no test can be generated.

For example, in the the shift registers (SHFT, SHFTV; "SHFT Description" on page 125 and "SHFTV Description" on page 132), a faulty AND in the sixth statement of each causes this problem. The fault is detected by observing which clause executes, by replacing a preloaded value in an object with a new value (by loading or shifting). However, the object cannot be preloaded, because preloading requires executing the first statement and setting its control expression true to execute any statements used to preload. The AND result cannot be set true, because the AND may be faulty, so the control expression cannot be set true. Thus, no test can be generated.

A STUCK fault can cause similar problems. For a STUCKTHEN, no statements under the ELSE can execute. If these are the only statements that can be used to preload, no test can be generated. Additionally, statements in the STUCK clause cannot be avoided if the IF statement executes. This can also eliminate tests.

Stuck-At Versus Assignment Control Faults

In some cases, stuck-at faults are easier to test than the current assignment control (ASSNCNTL) faults. Testing for a stuck-at fault only requires loading an object with one value, the opposite of the suspected stuck value. An ASSNCNTL fault needs two values, one to preload an object and then a different value to detect whether the faulted statement executes and loads the object. If the faulty assignment is the only assignment statement for the object, the object cannot be preloaded normally, and a two-phase test is used. For a bit object, this is equivalent to the two stuck-at tests for the bit. For a vector, it is usually equivalent, since the first choice for selecting different vector values is 00...0 versus 11...1. Thus, testing a stuck-at fault can be only half as complicated as testing an assignment control fault.

Reconvergent Fanout

Reconvergent fanout has long been an efficiency concern of test generation; however, one serious deficiency in the current method is it cannot handle reconvergent fanout. The deficiency is due to the representation of object and expression values.

In OBSOBJ and OBSEXPR goals, the good and bad values are represented explicitly, with 1s and 0s (and possibly Xs for extra bits in vectors); this is equivalent to a D or $\overline{D}$. Thus on the output path D and $\overline{D}$ values can be represented, the same as in the D-algorithm.

VIO and VIE goals can only specify a single value for the expression (i.e. a micro-operation result) or object. Thus, a VIO or VIE goal can only be satisfied by a definite (1 or 0, not D or $\overline{D}$) value loaded into an object or expression.

The problem with reconvergent fanout is that to propagate a value the test method might first try a 1 or 0 (or a vector of 0s and 1s) to propagate through a micro-operation. This 1 or 0 would be in a VIE goal. For reconvergent fanout, it would be unable to load the 1 or 0 because the expression is in the output path of the fault. To handle reconvergent fanout, the method would allow the expression value to be 1/0 (D) or 0/1 ($\overline{D}$), as appropriate. However, the equivalent of D or $\overline{D}$ cannot be represented in a VIO or VIE goal value field since they use single values.

The root of this problem is that originally only single-path propagation was addressed. A solution would be to allow good and bad value pairs in VIO and VIE goals and try various combinations as in the D-algorithm [1], or allow X's in one value, equivalent to the 9-V algorithm's approach [4].

Forward Implication

An efficiency concern is the lack of complete forward implication, used in many other test methods. (Substituting decided object values in expressions is partial forward implication, espe-

cially since cases such and '0' AND 'X' -> '0' are included. However, forward implication is not done to determine whether statements execute or whether objects are assigned.) Forward implication would help find conflicts more quickly, would help find already-solved goals, and could help find unintentionally-propagated fault results.

Time Choice Strategy

The strategy for justifying values in objects was first to decide which statement to use to load the object, and then to decide when to actually load it. That is, conditions necessary for execution are solved before the time is chosen. The intent was to save work by finding conflicts in how to load an object before deciding when to load it. If the time was decided and then the object was loaded but a conflict occured, another time would be picked and all the goal solving to execute the loading statement would be repeated. If the loading is solved first, conflicts are found before the time is chosen. This strategy can work well when there is no choice of how to load an object.

The current implementation saves all time choices for last, instead of making each time choice after its related loading goals are solved. This worked for some circuits, but may defer conflict detection too far and may cause wasteful backtracking. It is not clear which approach is better.

One problem with this strategy is that when the same value is needed in an object for different reasons, the loading of the value is considered multiple times. The loads are considered separate events, even if they must map to the same time period. If time choices were made sooner, when one value was loaded, the next goal requiring the value would find that the value has already been loaded, and would be inherently solved.

A less abstract time choice strategy is probably necessary to handle the multiple-load problem. Solutions might be typical adjacent-period methods, or Marlett's unidirectional time flow approach [8, 9].

Two-Phase Tests and Time Choices

Another problem with time choice order appears with two-phase tests. Consider a VIO goal on base time t1, with t2 being the highest-numbered base time currently allocated. If the VIO goal can be solved normally, the subgoals are based on t3, with t3 ≤ t1. However, if the VIO goal uses an ASSNCNTL-faulted statement to load the object, extra goals to do the extra load and observe phase are issued. These goals are based on t4. Since goals are solving in increasing order of their base times' subscripts, the regular goals on t3 will be solved (possibly yielding t5, t6, etc.), and then the second-phase goals on t4 will be solved. Next t5 goals (from the t3 goals) will be solved. If a conflict occurs, backtracking will try other options in t4 goals. Since t4 goals are independent of t5 goals, alternate choices will keep failing until until a choice of a t3 goal is changed. Thus, many choices of t4 goals will be tried when none affects the conflict. The problem is that the second-phase t4 goals are solved before conflicts implied from solving t3 goals are found.

Inequality Solving and Dummy Variables

The need to solve inequalities arises from the high-level fault model. (Su's high-level method also uses inequality solving [11].) For a stuck-at model, the value needed for a test is simply the opposite of the stuck value (though choices equivalent to inequality solving may appear elsewhere). For the chip-level fault models, exact values are flexible, but must satisfy a constraint, usually that two values are unequal. Inequality solving is an inherent part of testing for faults: ASSNCNTL tests directly require two different values to detect whether the statement executes, and IF and CASE tests need to detect which clause executes.

The problem with solving inequalities is that there can be many solutions, especially when the unequal items are wide vectors. The current strategy is to try, for example for a 3-bit vector, the values 000/111, 111/000, 0XX/1XX, 1XX/0XX, X0X/X1X, X1X/X0X, XX0/XX1, and

XX1/XX0. All zeroes and all ones are tried for maximum propagation chance (this is effectively propagating several Ds or $\overline{D}$s at once). Then single bit Ds or $\overline{D}$s are tried in case all zeroes or all ones is not possible (for example, the difference after one shift of a register). This retry assumes that failure to load was because of the data values; if the failure results from the inability to execute a statement to load values, time is wasted trying to find values, when no choice will work anyway.

One possible solution is to gather more constraints before choosing values using "dummy variables." When solving an inequality, one expression or object would be set equal to D1 and the other to D2, with the constraint that D1 $\neq$ D2. The symbolic values D1 and D2 would be loaded into the expressions or objects, further constraining the values for D1 and D2. For x $\neq$ y, VIOs would set x to D1 and y to D2, and specify that "D1 $\neq$ D2". If x is loaded with, say, "0", then D1 becomes "0", and "D1 $\neq$ D2" becomes "0 $\neq$ D2", which reduces to "D2 = 1". This is similar to subscripted D's in the subscripted D-algorithm [37].

New Framework

The Need for a New Framework

The test method as currently implemented obviously needs more work. It is ironic that while this automated test method defined on a fault model was intended to eliminate ad hoc test generation done by hand, the method itself is too ad hoc.

One main problem is the treatment of the fault during preloading and propagation. For preloading, the fault is avoided, or if it cannot be, preloading is done anyway but the preloaded value is checked. Part of this results from the fault model (directly from ASSNCNTL and indirectly from other faults). A sequential D/$\overline{D}$ algorithm can handle passing through the fault site again; our method is not yet built to handle this case.

A more unified framework is needed, which handles special cases more consistently, and handles things such as reconvergent fanout.

State Vector Style

One representation of most goals that would enable easier forward implication and would organize goals would be to use a state-vector style representation, where instead of the actual state of the circuit at some time (inputs and state variables), the test method keeps the state of all possible goals describing the circuit at each period. There would be entries for each object, each expression node (objects, literals, and micro-operation results), each statement and each clause. Most current goals would map to these entries. A VIO for an object would be represented by having the VIO value at the object's position in the state vector for the time given in the VIO goal. Most VIEs (those for copies of actual expressions, but not for constructed expressions) would map to a value in a state vector for the expression node the VIE refers to. Others would be kept separately as constraints, similar to the suggested dummy variables. An EXEC would map to a 1 for a statement; a DNE for the statement would map to a 0. In place of EXGs, there would be EXECCLAUSE goals, which would map to a place in the state vector.

State vector representation would simplify forward implication because the relations between different goals are represented as relations between nodes. State vector style could also save overhead by making goal representation more consistent.

An extension of state vector style leads to a circuit isomorphism. One can construct a logic circuit whose nodal relationships is isomorphic to the relationships between goals (this circuit is not the same as the original circuit). In addition, "gates" of this circuit will perform HDL-level micro-operations such as vector-wide and/or arithmetic operations. With this isomorphism, it might be possible to use a gate-style technique extended for vector-wide operations on the isomorphic circuit.

PODEM-Style Searching

Instead of path-tracing, the style of PODEM [2, 3] could be used to trying input combinations to find a test. Instead of solving goals from the site of the fault outwards (as the D-algorithm [1] does), inputs and state variable values could be manipulated (as PODEM manipulates inputs to control internal values) to sensitize and propagate tests. Of course, this is complicated by sequentiality. The manipulation could apply to inputs only for an arbitrary maximum number of time periods, or to inputs and state variables in a period, justifying one period's state variables in preceding periods.

At a higher level, test generation could keep track of basic circuit operations, choosing among them to construct tests.

Decision Numbers

An optimization applicable to a backtracking algorithm is keeping track of decision numbers to assist in backtracking [9]. Each choice is assigned a decision number greater than that of the previous choice. Goals from a choice are assigned the choice's decision number. Derived subgoals are assigned the highest decision number involved in deriving them. (E.g., substituting a VIO value into a VIE expression gives a new VIE goal; the new goal is assigned the higher of the numbers from the VIO or original VIE goal.) At a conflict, backtracking backs up to the most recent choice involved in causing the conflict. This eliminates retrying more recent choices which would have no effect on the conflict, and which would waste time.

Chapter 6

Conclusions

The successes of the experimental test method implementation show that the chip-level fault model can be used in automatic test generation, and indicate that the test method can generate tests. Goals can indeed represent elements of the testing problem, specifically sensitization, justification, and propagation. The goal solving used can transform basic test requirements into complete tests, working directly from the HDL circuit description.

The failures and weaknesses indicate that much work is still needed to develop a useful test generation method. Within the current framework of goal solving, improvements are needed in the time choice strategy and in the order of solving goals, to catch conflicts more quickly and reduce the number of new choices made before previous choices are validated. A new framework could organize goal solving more, to enable more consistent and efficient test generation.

References

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

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

[3] Prabhakar Goel and Barry C. Rosales, "PODEM-X: An Automatic Test Generation System for VLSI Logic Structures", *18th Design Automation Conference*, pp. 260-268, 1981.

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

[5] Gianfranco R. Putzolu and J. Paul Roth, "A Heuristic Algorithm for the Testing of Asynchronous Circuits", *IEEE Transactions on Computers*, vol C-20, pp. 639-647, June, 1971.

[6] H. Kubo, "A Procedure for Generating Tests Sequences to Detect Sequential Circuit Failures," *NEC Res. Develop.*, No. 12, October, 1968.

[7] Semyon Shteingart, Andrew W. Nagle, John Grason, "RTG: Automatic Register Level Test Generator", *22nd Design Automation Conference*, pp. 803-807, 1985.

[8] Ralph A. Marlett, "EBT: A Comprehensive Test Generation Technique for Highly Sequential Circuits", *15th Design Automation Conference*, pp. 335-339, 1978.

[9] Ralph A. Marlett, "An Effective Test Generation System for Sequential Circuits", *23rd Design Automation Conference*, pp. 250-256 June, 1986.

[10] Frederick J. Hill and Gerald R. Peterson, *Digital Systems: Hardware Organization and Design*, 1973.

[11] Tonysheng Lin and Stephen Y. H. Su, "The *S*-Algorithm: A Promising Solution for Systematic Functional Test Generation", *IEEE Transactions of Computer-Aided Design*, vol CAD-4, pp. 250-263, July, 1985.

[12] Tonysheng Lin and Stephen Y. H. Su, "VLSI Functional Test Pattern Generation - A Design and Implementation", *1985 International Test Conference*, pp. 922-929, 1985.

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

[14] Yinghua Min and Stephen Y. H. Su, "Testing Functional Faults in VLSI", *19th Design Automation Conference*, pp. 384-392, 1982.

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

[16] Chi-Chang Liaw, Stephen Y. H. Su, and Yashwant K. Malaiya, "State Diagram Approach for Functional Testing of Control Section", *1981 IEEE Test Conference*, pp. 433-446, 1981.

[17] *GSP User's Manual*, E. E. Dept., Virginia Polytechnic Institute and State University, December, 1982.

[18] James R. Armstrong and D. E. Devlin, "GSP: A Simulator for LSI," *18th Design Automation Conference*, pp 518-524, 1981.

[19] James M. Kerr and Chang H. Cho, "GSP2 Hardware Description and Modeling Language; Language Reference Manual", E. E. Dept, Virginia Polytechnic Institute and State University. February, 1986.

[20] James M. Kerr, "Compiled Code, Chip-Level Simulation Using a High Level Hardware Definition Language", Master's Thesis, Virginia Polytechnic Institute and State University, June, 1984.

[21] *VHDL Language Reference Manual, Version 7.2*, IR-MD-045-2, August 1985.

[22] *VHDL Language Reference Manual, IEEE Preliminary Version*, October, 1986.

[23] James R. Armstrong, Anil K. Gupta, and James Stewart, "Functional Fault Modeling for VLSI Devices", Final Report for IBM contract YD 190121, 1984.

[24] Anil Gupta, "Functional Fault Modeling and Test Vector Development for VLSI Systems", Master's Thesis, Virginia Polytechnic Institute and State University, March, 1985.

[25] Chang Cho, "A Chip Level Fault Coverage Experiment", E. E. Dept., Virginia Polytechnic Institute and State University, August, 1986.

[26] Satish M. Thatte and Jacob A. Abraham, "Test Generation for Microprocessors", *IEEE Transactions on Computers*, vol C-29, pp. 429-441 June, 1980.

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

[28] C. Clary, R. K. Joobbani, F. M. Smith, *Development of a Methodology for Verifying Military Computer Family Built-In-Test Performance Specifications*, Research and Development Technical Report, CORADCOM-80-0780-F, May, 1980.

[29] M. S. Abadir and H. K. Reghbati, "Functional Test Generation for LSI Circuits Described by Binary Decision Diagrams", *1985 International Test Conference*, pp. 483-492, 1985.

[30] James R. Armstrong, "Chip Level Modeling with Hardware Description Languages", *Design and Test*, to be published.

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

[32] Ramin Khorram, "Functional Test Pattern Generation for Integrated Circuits", *1984 International Test Conference*, pp. 246-249, 1984.

[33] S. K. Sathe, "Functional Level Simulation in LSI Devices", Master's Thesis, E. E. Dept, Virginia Polytechnic Institute and State University, June, 1982.

[34] Anil K. Gupta and James R. Armstrong, "Functional Fault Modeling and Simulation for VLSI Devices", *22nd Design Automation Conference*, pp. 720-726, 1985.

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

[36] John Roach and Glenn Fowler, *Virginia Tech ProLog/Lisp*, Virginia Tech Computer Science Department.

[37] C. Benmehrez and J. F. MacDonald, "Measured Performance of a Programmed Implementation of the Subscripted D-Algorithm", *20th Design Automation Conference*, pp. 308-315, 1983.

Appendix A

USER'S Manual

Introduction to User's Manual

This is the user's manual for the experimental ProLog implementation of the chip-level automatic test generation method.

VHDL Subset Allowed

The test method uses a restricted, modified subset of VHDL for model description. Concurrent IF, CASE, and simple assignment statements are allowed. Objects can be bit and bit vector signals (1 and 0, no Z). Standard logical and unsigned arithmetic operations are provided. Operations also include concatenation and restricted slices and elements of vector expressions (slices and

vector elements are not allowed for assignment statement destination objects or for an edge clock (with a 'STABLE)).

It is assumed that all assignments to a signal refer to the same driver. This is equivalent to putting all assignments to an object in one PROCESS statement, so there is only one driver. To maintain correct operation, the PROCESS statement would be sensitive to all signals used with it (this ensures that concurrent behavior is matched).

Assignments are of the form "< signal > < = < expression >." Any AFTER phrase is ignored, and multiple values are not allowed.

The use of 'STABLE is restricted. 'STABLE may be used only in an IF statement control expression. It must be used on an unsliced, unscripted signal. There may be only one 'STABLE in the expression, and the expression must not be of the form (... clk'STABLE) OR (...). IF statements with 'STABLEs may not be nested. (These formations are not common, anyway.)

In an IF expression with a 'STABLE, and in any source or control expressions under the IF's THEN clause, all objects are assumed to have a 'DELAYED, except the object with the 'STABLE, which is assumed not to. (This simply means that if an edge clock and a data input changes at the same time, the old data value is used.)

There may be no combinational feedback; all sequential behavior must be explicitly modeled (a chip-level model would not have combinational feedback anyway).

Using Other VHDL Constructs

Though not included, some other VHDL constructs can be rewritten in terms of what is implemented.

Translating into Internal Form

The input VHDL description must be translated into an internal form.

The internal form consists of ProLog facts or rules representing things such as object declarations, the type of each statement, and the structure of expressions in statements. The format of these rules is based on Virginia Tech ProLog's data formats. A knowledge of ProLog data structures and commands is recommended. See the Prolog user's manual [36].

Note: ProLog is case sensitive; upper and lower case is different. Use boldface words below exactly as given, and be consistent with the case of user-defined tokens such as signal names and statement IDs. Also, note the double parenthesis [((,))] used in these ProLog rules.

Declarations

The name of the model is declared as ((**modelname** " < name of model > ")). This is required, though used only for labeling output. There must also be a ((**fileprefix** " < filename > ")) rule, where < filename > is the name field of the VAX/VMS filename to be used for output files.

Objects are declared with (datatype), (statevar), (inputpin), and (outputpin) rules.

The data type of an object is declared as ((**datatype** < objectname > < type >)), where < objectname > is the name of the object, and < type > is **BIT** for bits and **BV** for bit vectors. For a BV declaration, the length of the vector is specified with ((**bvlength** < objectname > < length >)), where < length > is the number of bits. (Starting and ending bit indices are not kept; it is assumed that the rightmost bit is bit 0 and the leftmost of n bits is bit n-1.)

The only declaration rule which is not a direct translation of VHDL is the (statevar) rule. This rule specifies whether an object is a state variable. A state variable is any object whose value cannot be determined solely from the current values of other objects, or an object whose value depends

on its previous values. The form of the rule is **((statevar** <objectname> **t))** for state variables, and **((statevar** <objectname> **nil))** for non-state variables.

"In" and "out" ports are labeled with **((inputpin** <objectname>**))** and **((outputpin** <objectname>**))**, respectively. (Bidirectional pins are not implemented, so there are no inout ports). An internal object has no (inputpin) or (outputpin) rule.

Expressions

After declarations, the structure of the code must be represented. The code contains expressions of literal values, objects, and micro-operations. These must be represented.

Values

Bit values are represented as **(BIT "0")** or **(BIT "1")**. Bit vectors are represented as **(BV "**<string>**")**, where <string> is consists of one or more ones and zeroes.

Expression Elements

Expressions consist of literals, objects, and operations. Literals are represented as **(LIT** <value>**)**, where <value> is the representation of the value, as given above. Objects are represented as **(OBJ** <objectname>**)**. (This appears redundant, but these distinctions are used in the program.)

Expressions are represented as their parse trees, similar to prefix notation. An expression consisting of a binary operation and two arguments is represented as (<operation> <arg1> <arg2>), where <operation> is the operation (see below), and <arg1> and <arg2> are the internal form representing the arguments of the operation. A unary operation has the form

(< operation > < argument >). For example, the expression "a AND (b OR c)" for bits becomes **(BITAND (OBJ a) (BITOR (OBJ b) (OBJ c)))**.

Table 3 on page 74 gives the micro-operations available.

The BIT— operations must be used on bit arguments; BV— operations must be used on vectors.

BVCAT concatenates two bit vectors. For vectors a and b, VHDL "a & b" becomes **(BVCAT (OBJ a) (OBJ b))**. BITBV converts a bit to a one-bit bit vector, usually used for concatenating a bit to a bit vector. For example, if a is a bit, the **(BITBV (OBJ a))** is a vector of length one.

BVSUBV (Bit Vector SUBVector) is used to take a slice or element of an object or expression. BVSUBV has a special representation. The form **(BVSUBV** < left bit no. > < right bit no. > **)** takes a slice of a vector; e.g., " a[3 downto 2] " becomes **((BVSUBV 3 2) (OBJ a))**. (Note that the < operation > field is (BVSUBV 3 2), not just BVSUBV). The expression **((BVSUBV 3 3)** means " a[3 downto 3] ". The form **(BVSUBV** < bit number > **)** takes a single bit of a vector, as in **((BVSUBV 3) (OBJ a))**. This form returns a bit; the previous form returns a vector, even if only 1 bit wide. The distinction comes from VHDL.

The 'STABLE attribute on a object is represented as **(STABLE (OBJ** < objectname > **))**. Thus, " c'STABLE " becomes **(STABLE (OBJ c))**.

These representations for elements of expressions are used to describe expressions in statements.

Statements

In rules describing statements, < stmtid > refers to the statement. This is an arbitrary token, so it does not actually matter what is used, as long as the same token is used for all rules for one statement, and each statement has a different token. Numbering the statements sequentially and using s1, s2, s3, etc. is suggested.

The type of each statement is given as **((statementtype** < stmtid > < type > **))**, where < stmtid > is the id. of the statement, and < type > is **IF, CASE,** or **ASSIGNMENT.**

Table 3. Micro-Operations Available.

BIT operations:

BITAND	- AND
BITOR	- OR
BITNOT	- NOT
BITXOR	- XOR (exclusive or)
BITEQV	- XNOR (exclusive nor)

Vector operations:

BVAND	- bit-by-bit AND
BVOR	- bit-by-bit OR
BVNOT	- bit-by-bit NOT
BVXOR	- bit-by-bit XOR
BVEQV	- bit-by-bit XNOR
BVEQ	- vector-wide equality
BVNEQ	- vector-wide inequality
BVADD	- unsigned addition
BVSUB	- unsigned subtraction
BVLE	- unsigned less than
BVLT	- unsigned less than or equal

Special operations

BVCAT	- concatenation of two bit vectors
(BVSUBV <bit number>)	- single bit of an object or expression
(BVSUBV <left bit> <right bit>)	- slice of an object or expression
BITBV	- convert bit to bit vector
STABLE	- 'STABLE attribute

Note: BVADD and BVSUB are not fully implemented. Partially-constrained values (vectors with one or more X bits) are not correctly justified. If test generation tries to used such a partially-constrained value, the test will be invalid.

Assignment statements have a destination object and a source expression. The destination object is specified as ((**destinationobject** <stmtid> <objectname>)) and the source expression as ((**sourcexpression** <stmtid> <expr>)), where <expr> is the internal form of the expression.

IF and CASE statements have control expressions, and have clauses containing other statements. A control expression is specified with ((**controlexpression** <stmtid> <expr>)). The statement controlled by the IF or CASE are given with (subordinaterange) rules. Each clause has a rule of the form ((**subordinaterange** <stmtid> <clauseid> <range>)). The field <clauseid> is **THEN** or **ELSE** for IF statement clauses. For a CASE statement, the <clauseid> is a list of the values in the clauses's WHEN phrase, enclosed by parentheses. Usually, CASEs use integers, but since integers are not implemented, they are translated to bit vectors. For example, "WHEN 1|2|4 = >" becomes, after translation to vectors, " ((BV "001") (BV "010") (BV "100")) ". For a WHEN phrase value of "others," the values must be explicitly listed, unless the WHEN clause is empty. The IntVal used in such a case statement is not needed. The <range> field lists statements immediately under the clause. For example, in

```
s1: IF x = '0' THEN
s2:    y <= '1';
s3:    IF z THEN
s4:       w <= "01";
       ELSE
s5:       w <= "10";
       END IF
s6:    z <= u;
    ELSE
      . . .
```

the subordinate range for statement s1's THEN clause is s2, s3 and s6, but not s4 and s5 (which are children of statement s3). The rule would be ((**subordinaterange s1 THEN (s2 s3 s6)**)).

One additional rule specifies whether a statement is controlled by a 'STABLE. If an IF statement contains a 'STABLE on some object, the form is ((**understable2** <stmtid> <objectname>)), where <objectname> is the name of the object with the 'STABLE. For any statement under that IF, there is also an (understable2) rule, for that statement. (Note that if there

are two (or more) edge clocks, some statements will have (understable2) rules for one object, and some will have for the other object.) Any other statement must have **((understable2** <stmtid> **nil))**, specifying that it is not 'STABLE-controlled.

Table 4 on page 77 summarizes these rules.

These internal-form rules are entered into a standard VMS text file. The program assumes a default filetype of ".HDL" if not specified, so this filetype is recommended. The file must begin with "(assert", and end with ")" with the rules in between. (In ProLog terms, the rules representing the description are asserted.) The rules are listed in any order. Comments may be entered to the right of a semicolon (;) on any line.

Because of the nature of the work, there is little error checking for the input description. The usual symptom is that test generation will fail. Errors can be found by tracing execution, watching for failures when a goal should succeed.

Running the Programs

The test generation program runs in Virginia Tech Prolog on a DEC VAX under VMS. General knowledge of VMS is assumed, as is some familiarity with Virginia Tech ProLog command formats and interrupt breaks.

The DCL command file ATVG.COM runs the test generation programs. At the DCL "$" prompt, enter "@[<directory>]**ATVG SETUP**," where <directory> is the name of the directory containing the source files, to set up logical names and command symbols. ATVG.COM assumes the ProLog code resides in the same directory. Table 5 on page 78 lists the ProLog source files.

Enter **SETHDL** <filename> to point to an HDL file. If not specified, and the default directory and a filetype of .HDL are assumed. Whatever VMS directory holds the .HDL file will be used for the xxx.AUX, xxx.FAULTS, and xxx.TESTS files, below.

Table 4. Summary of Translation Rules

Declarations:

(modelname " < model name > ")

(fileprefix " < filename > ")
-filename part of HDL, TESTS, and FAULTS files

(inputpin < object name >) - input ports only

(outputpin < object name >) - output ports only

(datatype < object name > < type >) - all objects
< type > is BIT or BV

(bvlength < object name > < vector length >) - bit vectors only

(statevar < object name > < value >) - all objects
< value > is **t** if object is state variable, **nil** otherwise

Statement structure:

(statementtype < stmtid > < type >) - all statements
< type > is IF, CASE, or ASSIGNMENT

(understable2 < stmtid > < stableobj >) - all statements
< stableobj > is **nil** if statement is not under 'STABLE

Control statements only:

(controlexpression < stmtid > < expr >)
< expr > is control expression or IF or CASE statement

(subordinaterange < stmtid > < clauseid > < range >)
- for each clause of control statement
< clauseid > is **THEN** or **ELSE** for IF statement,
or list of WHEN values (**nil** if none) for CASE statement
< range > is statements directly under clause

Assignment statements only:

(sourceexpression < stmtid > < expr >)
< expr > is source expression of assignment statement

(destinationobject < stmtid > < object >)
< object > is object assigned by statement

Table 5. ProLog source files

- ATVG.COM - command file to control ProLog files

- BACKSOLVE.HC - goal-solving rules
- BACKTRACK.HC - goal-solving supervisory routines
- BACKUTIL.HC - unsolved goal list routines
- BASICTEST.HC - basic test selection
- BITOPS.HC - bit operation rules
- BVOPS.HC - bit vector operation rules
- COMPATIBLE.HC - conflict checks
- DSBLIB.HC - library of assorted rules
- EXPRRULES.HC - expression rules
- HDLRULES.HC - HDL element extraction rules
- INTOPS.HC - integer operation rules (not fully implemented)
- LOOKUP.HC - micro-operation fault case and basic test lookup rules
- OPRULES.HC - loads bitops, bvops
- PICKFAULTS.HC - fault listing rules
- PRESCAN.HC - preprocessing rules
- TIMERULES.HC - time substitution rules
- TRACE.HC - tracing rules
- WAVEFORM.HC - waveform formatting rules

Enter **SCAN** to preprocess the file, to create an xxx.AUX file. (The xxx is the value given for **((fileprefix "xxx"))**, and generally matches the filename of the .HDL file.)

To list the faults, enter **FAULTS**. This generates a file xxx.FAULTS, containing a list of faults for the HDL file.

Enter **TESTS** to select basic tests. This will generate a file xxx.TESTS, containing the basic test goals for each fault.

To run the test generation program, enter **SOLVE** This will load the main program which runs interactively, and give the user the ProLog command prompt (":-").

ProLog Command Entry: ProLog commands may be entered at command prompts, interrupt prompts, or user break prompts. The command prompt is ":-". This prompt means no execution is pending. The interrupt prompt is "<0>", "<1>", etc. Entering control-C at any time in ProLog interrupts execution and gives an interrupt prompt. At any "cr:" prompt from the program (used to acknowledge a message), entering any character before the <CR> (carriage return) gives a user break prompt ("ack>"). At an interrupt prompt, **t** continues execution and **nil** aborts *all* pending execution. At a break prompt, **t** continues execution (**nil** forces a local failure and usually is not used.)

Commands are:

- **(dotest)** - run tests
- **(showfaults)** - list faults in the .FAULTS file
- **(mytrace ...)** - set tracing options
- **(setbreak ...)** - set breakpoints
- **(clearbreak ...)** - clear breakpoints
- **(startlog ...)** - start log of terminal output
- **(quit)** - exit ProLog interpreter

Generating Test Vectors

To generate a test, enter **(dotest)** < CR > (< CR > means carriage return) at the command prompt (":-"). The program will ask for the fault number. At that point, enter the number of the fault to test, and press < CR >. The program will display the fault, and begin solving.

If the program succeeds in finding a test, it writes the test vector goals in the current default directory to xxxnn.VCTR, and the formatted waveform to xxxnn.WAVE, where xxx is the filename given in the **((fileprefix** xxx**))** rules, and nn is the fault number. If test generation fails, no files are written.

Displaying the Fault List

To display the fault list (to see fault numbers) from within the program, enter **(showfaults)**.

Tracing Goal Solving

Commands are provided for tracing various parts of execution of the program.

Tracing is set with **(mytrace)** or **(mytrace** < type > < level > **)** commands. Types are:

trying - display each goal when selected to be solved

solve - display each goal immediately before solving it

subgoals - display subgoals returned from solving a goal

backtrack - display message when trying alternate options
(not incompletely implemented)

already - display message for goals found to be already solved

unsolvable - display message for goals found to be unsolvable

(not incompletely implemented)

incompat - display message showing conflict

subs - display goals before and after object value substitution

The types trying and subgoals are of the most interest to the user.

There are three levels: 0, 1, and 2. Zero means do nothing, 1 means display a message, and 2 means to prompt the user with "cr:" to acknowledge the message by hitting <CR>.

Breakpoints

Breakpoints may be set as follows:

- **(setbreak level** <level>) - break right before solving reaches a depth <level>
- **(setbreak goal** <goaltype>) - break before solving any goal of that type
- **(setbreak timesubs)** - break before substituting times
- **(setbreak rule** <rulename>) - break before executing a certain rule

Breaks for **level** and **goal** are usually used for running test generation up to a point, then stopping to change tracing to observe execution step by step. **Timesubs** and **rule** breakpoints are mostly used for debugging. To break on several items, enter a (setbreak ...) command for each.

Breakpoints are cleared with the **(clearbreak ...)** command. The arguments are the same are for (setbreak) commands. To clear multiple breakpoints of one type, multiple (clearbreak) command must be used.

Keeping a Log

The **(startlog "<filename>")** command is used to start a log of whatever ProLog displays on the terminal. The command **(closelog log)** closes the log file.

Exiting Prolog

The **(quit)** command exits ProLog, returning the user to DCL.

Modifying the Fault Model

Fault modes for each micro-operation are defined in LOOKUP.HC, with the (lookupfaultmodes) rules. The form is **((lookupfaultmodes** <micro-op name> <faultmodes list>)), where <micro-op name> is the name of the operation (e.g. BITAND, BVADD) and <faultmodes list> is a parenthesized list ("nil" if empty) of failure modes for the micro-operation. Each mode is just name to match in a (lookuptest) rule. Usually, this name is the name of a existing micro-operation to which the intended operation fails; but since it only matches a corresponding name in a (lookuptest) rules, it could be anything.

Micro-operation sensitization conditions are given in (lookuptest) rules. The format is given in the ProLog file LOOKUP.HC. These rules give the sensitization condition by specifying a value to which a given expression must evaluate. The expression may be constructed from the micro-operation expressions. These rules also give the good and bad result, as symbolic functions of the argument values.

Appendix B

Circuit Models and Fault Lists

This appendix lists the .HDL files and annotated fault lists for the circuit models used in the research. The .HDL files are ProLog input files; the VHDL description is given in each file as ProLog comments. The ProLog rules describing the VHDL description follow the VHDL. Circuit diagrams are included for CCNT2, CKTA, and CKTCV (Figure 2 on page 96, Figure 3 on page 100, and Figure 4 on page 105). The fault list is the output file from the FAULTS command. Brief comments indicate exclusion, success, or failure for each fault attempted.

Exclusion reasons are as follows:

- Excluded - inverted clock: would clock on opposite edge
- Excluded - clocks on any edge: would clock on rise or fall, not just intended edge
- Excluded - continuous clocking: would clock at infinite frequency
- Excluded - clocks on high: would clock infinitely when high
- Excluded - clocks on high or edge: would clock when high, or on any edge
- Excluded - clocks on high unless edge: would clock when high, except at edge

Failure reasons are as follows:

- Failed; can't DNE under STUCK: can't avoid statements under STUCK clause
- Failed; faulty micro-op: MICROOP or STUCKDATA fault prevents execution of statements to justify or propagate test
- Failed; STUCK prevents preload: statements needed for preloading are inaccessible because of STUCK fault

ADDER Description

```
;    ENTITY Adder(
;          a,
;          b
;           : IN BIT_VECTOR(3 DOWNTO 0);
;          c: OUT BIT_VECTOR(3 DOWNTO 0)
;          ) IS
;    END Adder;
;
;    ARCHITECTURE Arch OF Adder is
;
;    PROCESS(a,b)
;    BEGIN
; s1:    c < = ADD(a,b)
;    END BLOCK;
;
;    END Adder;
(assert

((fileprefix "adder"))
((modelname "Simple adder, vectors"))

((datatype a BV)) ((bvlength a 4)) ((statevar a t))
((inputpin a))

((datatype b BV)) ((bvlength b 4)) ((statevar b t))
((inputpin b))

((datatype c BV)) ((bvlength c 4)) ((statevar c nil))
((outputpin c))


((statement s1))
((statementtype s1 ASSIGNMENT))
((sourceexpression s1 (BVADD (OBJ a) (OBJ b))))
((destinationobject s1 c))
((understable2 s1 nil))

) ; end assert
```

ADDER Fault List

"Simple adder, vectors"

(1 ASSNCNTL s1)	Successful, but 2-phase
(2 MICROOP (s1 - "") BVADD BVSUB)	Successful
(3 MICROOP (s1 - "") BVADD BVXOR)	Successful
(4 STUCKDATA (s1 - "") (BV "0000"))	Successful
(5 STUCKDATA (s1 - "") (BV "1111"))	Successful
(6 STUCKDATA (s1 - L) (BV "0000"))	Successful
(7 STUCKDATA (s1 - L) (BV "1111"))	Successful
(8 STUCKDATA (s1 - R) (BV "0000"))	Successful
(9 STUCKDATA (s1 - R) (BV "1111"))	Successful

ADDR2 Description

```
;    ENTITY Addr2
;     (a1,a2,a3,a4,
;      b1,b2,b3,b4,
;      c0: IN BIT;
;      s1,s2,s3,s4,
;      c4: OUT BIT) IS
;    END Addr2;
;
;    ARCHITECTURE Arch OF addr2 is
;
;    PROCESS (a1,a2,a3,a4,b1,b2,b3,b4,c0,c1,c2,c3,s1,s2,s3)
;      SIGNAL c1,
;             c2,
;             c3 : BIT;
;    BEGIN
; s1:   s1 = a1 XOR (b1 XOR c0);
; s2:   c1 = (a1 AND b1) OR (c0 AND (a1 XOR b1) ) ;
; s3:   s2 = a2 XOR (b2 XOR c1);
; s4:   c2 = (a2 AND b2) OR (c1 AND (a2 XOR b2) ) ;
; s5:   s3 = a3 XOR (b3 XOR c2);
; s6:   c3 = (a3 AND b3) OR (c2 AND (a3 XOR b3) ) ;
; s7:   s4 = a4 XOR (b4 XOR c3);
; s8:   c4 = (a4 AND b4) OR (c3 AND (a4 XOR b4) ) ;
;    END BLOCK;
;
;    END Arch;
(assert

((modelname "Gate-Level Adder"))
((fileprefix "addr2"))

((datatype a1 BIT)) ((statevar a1 t)) ((inputpin a1))

((datatype a2 BIT)) ((statevar a2 t)) ((inputpin a2))

((datatype a3 BIT)) ((statevar a3 t)) ((inputpin a3))

((datatype a4 BIT)) ((statevar a4 t)) ((inputpin a4))

((datatype b1 BIT)) ((statevar b1 t)) ((inputpin b1))

((datatype b2 BIT)) ((statevar b2 t)) ((inputpin b2))

((datatype b3 BIT)) ((statevar b3 t)) ((inputpin b3))

((datatype b4 BIT)) ((statevar b4 t)) ((inputpin b4))

((datatype c0 BIT)) ((statevar c0 t)) ((inputpin c0))

((datatype s1 BIT)) ((statevar s1 nil)) ((outputpin s1))

((datatype s2 BIT)) ((statevar s2 nil)) ((outputpin s2))
```

```
((datatype s3 BIT)) ((statevar s3 nil)) ((outputpin s3))

((datatype s4 BIT)) ((statevar s4 nil)) ((outputpin s4))

((datatype c1 BIT)) ((statevar c1 nil))

((datatype c2 BIT)) ((statevar c2 nil))

((datatype c3 BIT)) ((statevar c3 nil))

((datatype c4 BIT)) ((statevar c4 nil)) ((outputpin c4))

)(assert

((statement s1))
((statementtype s1 ASSIGNMENT))
((destinationobject s1 s1))
((sourceexpression s1 (BITXOR (OBJ a1)
                              (BITXOR (OBJ b1) (OBJ c0)))))
((understable2 s1 nil))

((statement s2))
((statementtype s2 ASSIGNMENT))
((destinationobject s2 c1))
((sourceexpression s2 (BITOR (BITAND (OBJ a1) (OBJ b1))
                             (BITAND (OBJ c0)
                                     (BITXOR (OBJ a1)
                                             (OBJ b1))))))
((understable2 s2 nil))

((statement s3))
((statementtype s3 ASSIGNMENT))
((destinationobject s3 s2))
((sourceexpression s3 (BITXOR (OBJ a2)
                              (BITXOR (OBJ b2) (OBJ c1)))))
((understable2 s3 nil))

((statement s4))
((statementtype s4 ASSIGNMENT))
((destinationobject s4 c2))
((sourceexpression s4 (BITOR (BITAND (OBJ a2) (OBJ b2))
                             (BITAND (OBJ c1)
                                     (BITXOR (OBJ a2)
                                             (OBJ b2))))))
((understable2 s4 nil))

((statement s5))
((statementtype s5 ASSIGNMENT))
((destinationobject s5 s3))
((sourceexpression s5 (BITXOR (OBJ a3)
                              (BITXOR (OBJ b3) (OBJ c2)))))
((understable2 s5 nil))

((statement s6))
((statementtype s6 ASSIGNMENT))
```

```
((destinationobject s6 c3))
((sourceexpression s6 (BITOR (BITAND (OBJ a3) (OBJ b3))
                             (BITAND (OBJ c2)
                                  (BITXOR (OBJ a3)
                                       (OBJ b3))))))
((understable2 s6 nil))

((statement s7))
((statementtype s7 ASSIGNMENT))
((destinationobject s7 s4))
((sourceexpression s7 (BITXOR (OBJ a4)
                              (BITXOR (OBJ b4) (OBJ c3)))))
((understable2 s7 nil))

((statement s8))
((statementtype s8 ASSIGNMENT))
((destinationobject s8 c4))
((sourceexpression s8 (BITOR (BITAND (OBJ a4) (OBJ b4))
                             (BITAND (OBJ c3)
                                  (BITXOR (OBJ a4)
                                       (OBJ b4))))))
((understable2 s8 nil))

)
```

ADDR2 Fault List

"Gate-Level Adder"

(1 ASSNCNTL s1)	Successful, but 2-phase
(2 MICROOP (s1 - "") BITXOR BITEQV)	Successful
(3 MICROOP (s1 - R) BITXOR BITEQV)	Successful
(4 ASSNCNTL s2)	Successful, but 2-phase
(5 MICROOP (s2 - "") BITOR BITAND)	Successful
(6 MICROOP (s2 - L) BITAND BITOR)	Successful
(7 MICROOP (s2 - R) BITAND BITOR)	Successful
(8 MICROOP (s2 - RR) BITXOR BITEQV)	Successful
(9 ASSNCNTL s3)	Successful, but 2-phase
(10 MICROOP (s3 - "") BITXOR BITEQV)	Successful
(11 MICROOP (s3 - R) BITXOR BITEQV)	Successful
(12 ASSNCNTL s4)	Successful
(13 MICROOP (s4 - "") BITOR BITAND)	Successful
(14 MICROOP (s4 - L) BITAND BITOR)	Successful
(15 MICROOP (s4 - R) BITAND BITOR)	Successful
(16 MICROOP (s4 - RR) BITXOR BITEQV)	Successful
(17 ASSNCNTL s5)	Successful
(18 MICROOP (s5 - "") BITXOR BITEQV)	Successful
(19 MICROOP (s5 - R) BITXOR BITEQV)	Successful
(20 ASSNCNTL s6)	Successful
(21 MICROOP (s6 - "") BITOR BITAND)	Successful
(22 MICROOP (s6 - L) BITAND BITOR)	Successful
(23 MICROOP (s6 - R) BITAND BITOR)	Successful
(24 MICROOP (s6 - RR) BITXOR BITEQV)	Successful
(25 ASSNCNTL s7)	Successful
(26 MICROOP (s7 - "") BITXOR BITEQV)	Successful
(27 MICROOP (s7 - R) BITXOR BITEQV)	Successful
(28 ASSNCNTL s8)	Successful
(29 MICROOP (s8 - "") BITOR BITAND)	Successful
(30 MICROOP (s8 - L) BITAND BITOR)	Successful
(31 MICROOP (s8 - R) BITAND BITOR)	Successful
(32 MICROOP (s8 - RR) BITXOR BITEQV)	Successful
(33 STUCKDATA (s1 - "") (BIT "0"))	Successful
(34 STUCKDATA (s1 - "") (BIT "1"))	Successful
(35 STUCKDATA (s1 - L) (BIT "0"))	Successful
(36 STUCKDATA (s1 - L) (BIT "1"))	Successful
(37 STUCKDATA (s1 - R) (BIT "0"))	Successful
(38 STUCKDATA (s1 - R) (BIT "1"))	Successful
(39 STUCKDATA (s1 - RL) (BIT "0"))	Successful
(40 STUCKDATA (s1 - RL) (BIT "1"))	Successful
(41 STUCKDATA (s1 - RR) (BIT "0"))	Successful
(42 STUCKDATA (s1 - RR) (BIT "1"))	Successful
(43 STUCKDATA (s2 - "") (BIT "0"))	Successful
(44 STUCKDATA (s2 - "") (BIT "1"))	Successful
(45 STUCKDATA (s2 - L) (BIT "0"))	Successful
(46 STUCKDATA (s2 - L) (BIT "1"))	Successful
(47 STUCKDATA (s2 - LL) (BIT "0"))	Successful
(48 STUCKDATA (s2 - LL) (BIT "1"))	Successful
(49 STUCKDATA (s2 - LR) (BIT "0"))	Successful
(50 STUCKDATA (s2 - LR) (BIT "1"))	Successful
(51 STUCKDATA (s2 - R) (BIT "0"))	Successful
(52 STUCKDATA (s2 - R) (BIT "1"))	Successful

(53 STUCKDATA (s2 - RL) (BIT "0")) Successful
(54 STUCKDATA (s2 - RL) (BIT "1")) Successful
(55 STUCKDATA (s2 - RR) (BIT "0")) Successful
(56 STUCKDATA (s2 - RR) (BIT "1")) Successful
(57 STUCKDATA (s2 - RRL) (BIT "0")) Successful
(58 STUCKDATA (s2 - RRL) (BIT "1")) Successful
(59 STUCKDATA (s2 - RRR) (BIT "0")) Successful
(60 STUCKDATA (s2 - RRR) (BIT "1")) Successful
(61 STUCKDATA (s3 - "") (BIT "0")) Successful
(62 STUCKDATA (s3 - "") (BIT "1")) Successful
(63 STUCKDATA (s3 - L) (BIT "0")) Successful
(64 STUCKDATA (s3 - L) (BIT "1")) Successful
(65 STUCKDATA (s3 - R) (BIT "0")) Successful
(66 STUCKDATA (s3 - R) (BIT "1")) Successful
(67 STUCKDATA (s3 - RL) (BIT "0")) Successful
(68 STUCKDATA (s3 - RL) (BIT "1")) Successful
(69 STUCKDATA (s3 - RR) (BIT "0")) Successful
(70 STUCKDATA (s3 - RR) (BIT "1")) Successful
(71 STUCKDATA (s4 - "") (BIT "0")) Successful
(72 STUCKDATA (s4 - "") (BIT "1")) Successful
(73 STUCKDATA (s4 - L) (BIT "0")) Successful
(74 STUCKDATA (s4 - L) (BIT "1")) Successful
(75 STUCKDATA (s4 - LL) (BIT "0")) Successful
(76 STUCKDATA (s4 - LL) (BIT "1")) Successful
(77 STUCKDATA (s4 - LR) (BIT "0")) Successful
(78 STUCKDATA (s4 - LR) (BIT "1")) Successful
(79 STUCKDATA (s4 - R) (BIT "0")) Successful
(80 STUCKDATA (s4 - R) (BIT "1")) Successful
(81 STUCKDATA (s4 - RL) (BIT "0")) Successful
(82 STUCKDATA (s4 - RL) (BIT "1")) Successful
(83 STUCKDATA (s4 - RR) (BIT "0")) Successful
(84 STUCKDATA (s4 - RR) (BIT "1")) Successful
(85 STUCKDATA (s4 - RRL) (BIT "0")) Successful
(86 STUCKDATA (s4 - RRL) (BIT "1")) Successful
(87 STUCKDATA (s4 - RRR) (BIT "0")) Successful
(88 STUCKDATA (s4 - RRR) (BIT "1")) Successful
(89 STUCKDATA (s5 - "") (BIT "0")) Successful
(90 STUCKDATA (s5 - "") (BIT "1")) Successful
(91 STUCKDATA (s5 - L) (BIT "0")) Successful
(92 STUCKDATA (s5 - L) (BIT "1")) Successful
(93 STUCKDATA (s5 - R) (BIT "0")) Successful
(94 STUCKDATA (s5 - R) (BIT "1")) Successful
(95 STUCKDATA (s5 - RL) (BIT "0")) Successful
(96 STUCKDATA (s5 - RL) (BIT "1")) Successful
(97 STUCKDATA (s5 - RR) (BIT "0")) Successful
(98 STUCKDATA (s5 - RR) (BIT "1")) Successful
(99 STUCKDATA (s6 - "") (BIT "0")) Successful
(100 STUCKDATA (s6 - "") (BIT "1")) Successful
(101 STUCKDATA (s6 - L) (BIT "0")) Successful
(102 STUCKDATA (s6 - L) (BIT "1")) Successful
(103 STUCKDATA (s6 - LL) (BIT "0")) Successful
(104 STUCKDATA (s6 - LL) (BIT "1")) Successful
(105 STUCKDATA (s6 - LR) (BIT "0")) Successful
(106 STUCKDATA (s6 - LR) (BIT "1")) Successful
(107 STUCKDATA (s6 - R) (BIT "0")) Successful
(108 STUCKDATA (s6 - R) (BIT "1")) Successful

(109 STUCKDATA (s6 - RL) (BIT "0")) Successful
(110 STUCKDATA (s6 - RL) (BIT "1")) Successful
(111 STUCKDATA (s6 - RR) (BIT "0")) Successful
(112 STUCKDATA (s6 - RR) (BIT "1")) Successful
(113 STUCKDATA (s6 - RRL) (BIT "0")) Successful
(114 STUCKDATA (s6 - RRL) (BIT "1")) Successful
(115 STUCKDATA (s6 - RRR) (BIT "0")) Successful
(116 STUCKDATA (s6 - RRR) (BIT "1")) Successful
(117 STUCKDATA (s7 - "") (BIT "0")) Successful
(118 STUCKDATA (s7 - "") (BIT "1")) Successful
(119 STUCKDATA (s7 - L) (BIT "0")) Successful
(120 STUCKDATA (s7 - L) (BIT "1")) Successful
(121 STUCKDATA (s7 - R) (BIT "0")) Successful
(122 STUCKDATA (s7 - R) (BIT "1")) Successful
(123 STUCKDATA (s7 - RL) (BIT "0")) Successful
(124 STUCKDATA (s7 - RL) (BIT "1")) Successful
(125 STUCKDATA (s7 - RR) (BIT "0")) Successful
(126 STUCKDATA (s7 - RR) (BIT "1")) Successful
(127 STUCKDATA (s8 - "") (BIT "0")) Successful
(128 STUCKDATA (s8 - "") (BIT "1")) Successful
(129 STUCKDATA (s8 - L) (BIT "0")) Successful
(130 STUCKDATA (s8 - L) (BIT "1")) Successful
(131 STUCKDATA (s8 - LL) (BIT "0")) Successful
(132 STUCKDATA (s8 - LL) (BIT "1")) Successful
(133 STUCKDATA (s8 - LR) (BIT "0")) Successful
(134 STUCKDATA (s8 - LR) (BIT "1")) Successful
(135 STUCKDATA (s8 - R) (BIT "0")) Successful
(136 STUCKDATA (s8 - R) (BIT "1")) Successful
(137 STUCKDATA (s8 - RL) (BIT "0")) Successful
(138 STUCKDATA (s8 - RL) (BIT "1")) Successful
(139 STUCKDATA (s8 - RR) (BIT "0")) Successful
(140 STUCKDATA (s8 - RR) (BIT "1")) Successful
(141 STUCKDATA (s8 - RRL) (BIT "0")) Successful
(142 STUCKDATA (s8 - RRL) (BIT "1")) Successful
(143 STUCKDATA (s8 - RRR) (BIT "0")) Successful
(144 STUCKDATA (s8 - RRR) (BIT "1")) Successful

CCNT2 Description

(See diagram, Figure 2 on page 96.)

```
;;  ENTITY controlled_ctr(
;;        clk,
;;        strb : IN BIT;
;;        con : IN BIT_VECTOR(1 DOWNTO 0);
;;        data : IN BIT_VECTOR(1 DOWNTO 0);
;;        count : OUT BIT_VECTOR(1 DOWNTO 0)) is
;;
;;  END controlled_ctr;
;;
;;
;;  ARCHITECTURE Arch OF controlled_ctr IS
;;
;;    PROCESS (clk,strb,con,data,consig,count,lim);
;;    SIGNAL
;;      en : BOOLEAN; ??
;;      dir : ?? (up,down)
;;      lim
;;        : BITVECTOR(1 DOWNTO 0) ;
;;      loadflag : BOOLEAN
;;    BEGIN
;; s01:   IF strb='1' AND NOT strb'STABLE THEN
;; s02:     CASE IntVal(con) IS
;;            WHEN 0 = >
;; s03:         count < = "00";
;;            WHEN 1 = >
;; s04:         loadflag < = TRUE;
;;            WHEN 2 = >
;; s05:         en < = TRUE;
;; s06:         dir < = up;
;;            WHEN 3 = >
;; s07:         en < = TRUE;
;; s08:         dir < = down;
;;          END CASE;
;;      END IF;
;;
;; s09:   IF (strb='0' AND NOT strb'STABLE) AND loadflag THEN
;; s10:     lim < = data;
;; s11:     loadflag < = FALSE;
;;      END IF;
;;
;; s12:   IF (clk='1' AND NOT clk'STABLE) AND en THEN
;; s13:     IF dir=up THEN
;; s14:       count < = ADD(count,"01");
;;          ELSE
;; s15:       count < = SUB(count,"01");
;;          END IF;
;;      END IF;
;;
;; s16:   IF count = lim THEN
;; s17:     en < = FALSE;
;;      END IF;
```

```
;;
;;  END PROCESS;
;; END Arch;
(assert

((modelname "Controlled Counter (2)"))
((fileprefix "CCNT2"))

((datatype clk BIT)) ((statevar clk t)) ((inputpin clk))

((datatype strb BIT)) ((statevar strb t)) ((inputpin strb))

((datatype con BV)) ((bvlength con 2)) ((statevar con t)) ((inputpin con))

((datatype data BV)) ((bvlength data 2)) ((statevar data t)) ((inputpin data))

((datatype count BV)) ((bvlength count 2)) ((statevar count t))
((outputpin count))

((datatype en BIT)) ((statevar en t))

((datatype dir BIT)) ((statevar dir t))

((datatype lim BV)) ((bvlength lim 2)) ((statevar lim t))

((datatype loadflag BIT)) ((statevar loadflag t))


((statementtype s1 IF)) ((understable2 s1 strb))
((controlexpression s1 (BITAND (BITEQV (OBJ strb) (LIT (BIT "1")))
                               (BITNOT (STABLE (OBJ strb))))))
((subordinaterange s1 THEN (s2)))
((subordinaterange s1 ELSE nil))

((statementtype s2 CASE)) ((understable2 s2 strb))
((controlexpression s2 (OBJ con)))
((subordinaterange s2 ((BV "00")) (s3)))
((subordinaterange s2 ((BV "01")) (s4)))
((subordinaterange s2 ((BV "10")) (s5 s6)))
((subordinaterange s2 ((BV "11")) (s7 s8)))

((statementtype s3 ASSIGNMENT)) ((understable2 s3 strb))
((destinationobject s3 count)) ((sourceexpression s3 (LIT (BV "00"))))

((statementtype s4 ASSIGNMENT)) ((understable2 s4 strb))
((destinationobject s4 loadflag)) ((sourceexpression s4 (LIT (BIT "1"))))

((statementtype s5 ASSIGNMENT)) ((understable2 s5 strb))
((destinationobject s5 en)) ((sourceexpression s5 (LIT (BIT "1"))))

((statementtype s6 ASSIGNMENT)) ((understable2 s6 strb))
((destinationobject s6 dir)) ((sourceexpression s6 (LIT (BIT "0"))))

((statementtype s7 ASSIGNMENT)) ((understable2 s7 strb))
((destinationobject s7 en)) ((sourceexpression s7 (LIT (BIT "1"))))
```

```
((statementtype s8 ASSIGNMENT)) ((understable2 s8 strb))
((destinationobject s8 dir)) ((sourceexpression s8 (LIT (BIT "1"))))

((statementtype s9 IF)) ((understable2 s9 strb))
((controlexpression s9 (BITAND (BITAND (BITEQV (OBJ strb) (LIT (BIT "0")))
                                (BITNOT (STABLE (OBJ strb))) )
                               (OBJ loadflag) )))
((subordinaterange s9 THEN (s10 s11)))
((subordinaterange s9 ELSE nil))

((statementtype s10 ASSIGNMENT)) ((understable2 s10 strb))
((destinationobject s10 lim)) ((sourceexpression s10 (OBJ data)))

((statementtype s11 ASSIGNMENT)) ((understable2 s11 strb))
((destinationobject s11 loadflag)) ((sourceexpression s11 (LIT (BIT "0"))))

((statementtype s12 IF)) ((understable2 s12 clk))
((controlexpression s12 (BITAND (BITAND (BITEQV (OBJ clk) (LIT (BIT "1")))
                                (BITNOT (STABLE (OBJ clk))))
                               (OBJ en) ) ))
((subordinaterange s12 THEN (s13)))
((subordinaterange s12 ELSE nil))

((statementtype s13 IF)) ((understable2 s13 clk))
((controlexpression s13 (BITEQV (OBJ dir) (LIT (BIT "0")))))

((subordinaterange s13 THEN (s14)))
((subordinaterange s13 ELSE (s15)))

((statementtype s14 ASSIGNMENT)) ((understable2 s14 clk))
((destinationobject s14 count))
((sourceexpression s14 (BVADD (OBJ count)
                              (LIT (BV "01")))))

((statementtype s15 ASSIGNMENT)) ((understable2 s15 clk))
((destinationobject s15 count))
((sourceexpression s15 (BVSUB (OBJ count)
                              (LIT (BV "01")))))

((statementtype s16 IF)) ((understable2 s16 nil))
((controlexpression s16 (BVEQ (OBJ count) (OBJ lim))))
((subordinaterange s16 THEN (s17)))
((subordinaterange s16 ELSE nil))

((statementtype s17 ASSIGNMENT)) ((understable2 s17 nil))
((destinationobject s17 en)) ((sourceexpression s17 (LIT (BIT "0"))))
) ; end assert
```

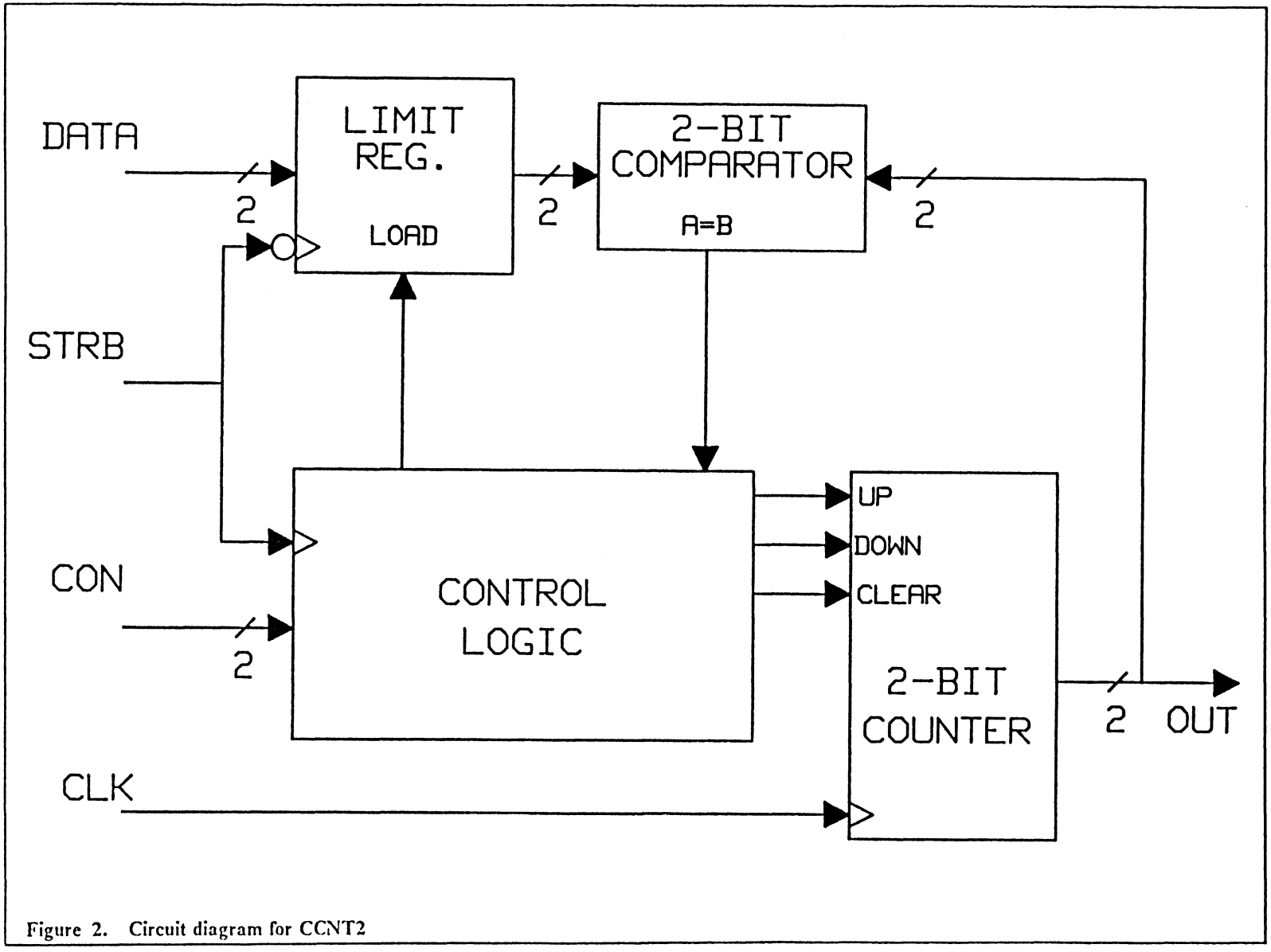


Figure 2. Circuit diagram for CCNT2

CCNT2 Fault List

"Controlled Counter (2)"	
(1 STUCKTHEN s1)	Excluded - continuous clocking
(2 STUCKELSE s1)	Failed; STUCK prevents preload
(3 MICROOP (s1 - "") BITAND BITOR)	Excluded - clocks on high or any edge
(4 MICROOP (s1 - L) BITEQV BITXOR)	Excluded - inverted clock
(5 MICROOP (s1 - R) BITNOT BITBUF)	Excluded - clocks on high unless edge
(6 DEADCLAUSE s2 ((BV "11")))	Overflow; can't load EN flag
(7 DEADCLAUSE s2 ((BV "10")))	Overflow;
(8 DEADCLAUSE s2 ((BV "01")))	Failed; DEAD prevents preload
(9 DEADCLAUSE s2 ((BV "00")))	Overflow; multiple loads
(10 ASSNCNTL s3)	Overflow; can't preload
(11 ASSNCNTL s4)	Overflow; fault interferes with preload
(12 ASSNCNTL s5)	Overflow; bad initial choices for count and limit
(13 ASSNCNTL s6)	Overflow; also, BVADD/BVSUB not fully implemented
(14 ASSNCNTL s7)	Overflow; multiple loads; bad initial choices
(15 ASSNCNTL s8)	Overflow; multiple loads; possible bad choices
(16 STUCKTHEN s9)	Excluded - continuous clocking
(17 STUCKELSE s9)	Overflow; endless loop (OBSEXEC needs loop cuts)
(18 MICROOP (s9 - "") BITAND BITOR)	Excluded - clocks on high or any edge
(19 MICROOP (s9 - L) BITAND BITOR)	Excluded - clocks on high or any edge
(20 MICROOP (s9 - LL) BITEQV BITXOR)	Excluded - inverted clock
(21 MICROOP (s9 - LR) BITNOT BITBUF)	Excluded - clocks on high unless edge
(22 ASSNCNTL s10)	Overflow; two-phase delays conflict detection
(23 ASSNCNTL s11)	Failed; endless loop (OBSEXEC needs loop checks)
(24 STUCKTHEN s12)	Excluded - continuous clocking
(25 STUCKELSE s12)	Successful
(26 MICROOP (s12 - "") BITAND BITOR)	Excluded - clocks on high or any edge
(27 MICROOP (s12 - L) BITAND BITOR)	Excluded - clocks on high or any edge
(28 MICROOP (s12 - LL) BITEQV BITXOR)	Excluded - inverted clock
(29 MICROOP (s12 - LR) BITNOT BITBUF)	Excluded - clocks on high unless edge
(30 STUCKTHEN s13)	Overflow; bad inequality choices; ADD/SUB incorrect
(31 STUCKELSE s13)	Overflow; bad inequality choices; ADD/SUB incorrect
(32 MICROOP (s13 - "") BITEQV BITXOR)	Overflow; bad inequality choices
(33 ASSNCNTL s14)	Overflow; bad inequality choices; delayed conflict detection
(34 MICROOP (s14 - "") BVADD BVSUB)	Overflow; bad inequality choice
(35 MICROOP (s14 - "") BVADD BVXOR)	Overflow; bad inequality choice
(36 ASSNCNTL s15)	Overflow; nonoptimal first choice; delayed fault detection
(37 MICROOP (s15 - "") BVSUB BVADD)	Overflow; bad inequality choices
(38 STUCKTHEN s16)	Overflow; multiple loads; bad inequality choice
(39 STUCKELSE s16)	Overflow; multiple loads; bad inequality choice
(40 MICROOP (s16 - "") BVEQ BVNEQ) ce	Overflow; multiple loads; bad inequality choi
(41 ASSNCNTL s17)	Overflow; multiple loads

CKTA Description

```
; (See diagram, Figure 3 on page 100.)
; File CIRCUITA.HDL -

;    ENTITY CircuitA is
;     (datain,
;      clock1,
;      clock2
;       : IN BIT;
;      andout : OUT BIT)
;    END CircuitA;
;
;    ARCHITECTURE Arch OF CircuitA is
;
;    PROCESS(clock1,clock2,q1,q2)
;      SIGNAL
;        q1,
;        q2
;         : BIT;
;    BEGIN
; 1:    IF (clock1='1' AND NOT clock1'STABLE) THEN
; 2:      q1 < = datain;
; 3:    IF (clock2='1' AND NOT clock2'STABLE) THEN
; 4:      q2 < = q1;
; 5:    andout < = q1 AND q2;
;    END PROCESS;
;
;    END Arch;
(assert

((fileprefix "ckta"))
((modelname "Circuit A"))

((datatype datain BIT)) ((statevar datain t)) ((inputpin datain))

((datatype clock1 BIT)) ((statevar clock1 t)) ((inputpin clock1))

((datatype clock2 BIT)) ((statevar clock2 t)) ((inputpin clock2))

((datatype q1 BIT)) ((statevar q1 t))

((datatype q2 BIT)) ((statevar q2 t))

((datatype andout BIT)) ((statevar andout nil)) ((outputpin andout))


((statement s1))
((statementtype s1 IF))
  ((controlexpression s1 (BITAND (BITEQV (OBJ clock1) (LIT (BIT "1")))
                         (BITNOT (STABLE (OBJ clock1))))))
  ((subordinaterange s1 THEN (s2) ))
  ((subordinaterange s1 ELSE nil ))
((understable2 s1 clock1))
```

```
((statement s2))
((statementtype s2 ASSIGNMENT))
  ((sourceexpression s2 (OBJ datain) ))
  ((destinationobject s2 q1))
((understable s2))
((understable2 s2 clock1))

((statement s3))
((statementtype s3 IF))
   ((controlexpression s3 (BITAND (BITEQV (OBJ clock2) (LIT (BIT "1")))
                          (BITNOT (STABLE (OBJ clock2)))) ))
  ((subordinaterange s3 THEN (s4) ))
  ((subordinaterange s3 ELSE nil ))
((understable2 s3 clock2))

((statement s4))
((statementtype s4 ASSIGNMENT))
  ((sourceexpression s4 (OBJ q1) ))
  ((destinationobject s4 q2))
((understable s4))
((understable2 s4 clock2))

((statement s5))
((statementtype s5 ASSIGNMENT))
  ((sourceexpression s5 (BITAND (OBJ q1) (OBJ q2)) ))
  ((destinationobject s5 andout))
((understable2 s5 nil))

) ; end assert
```

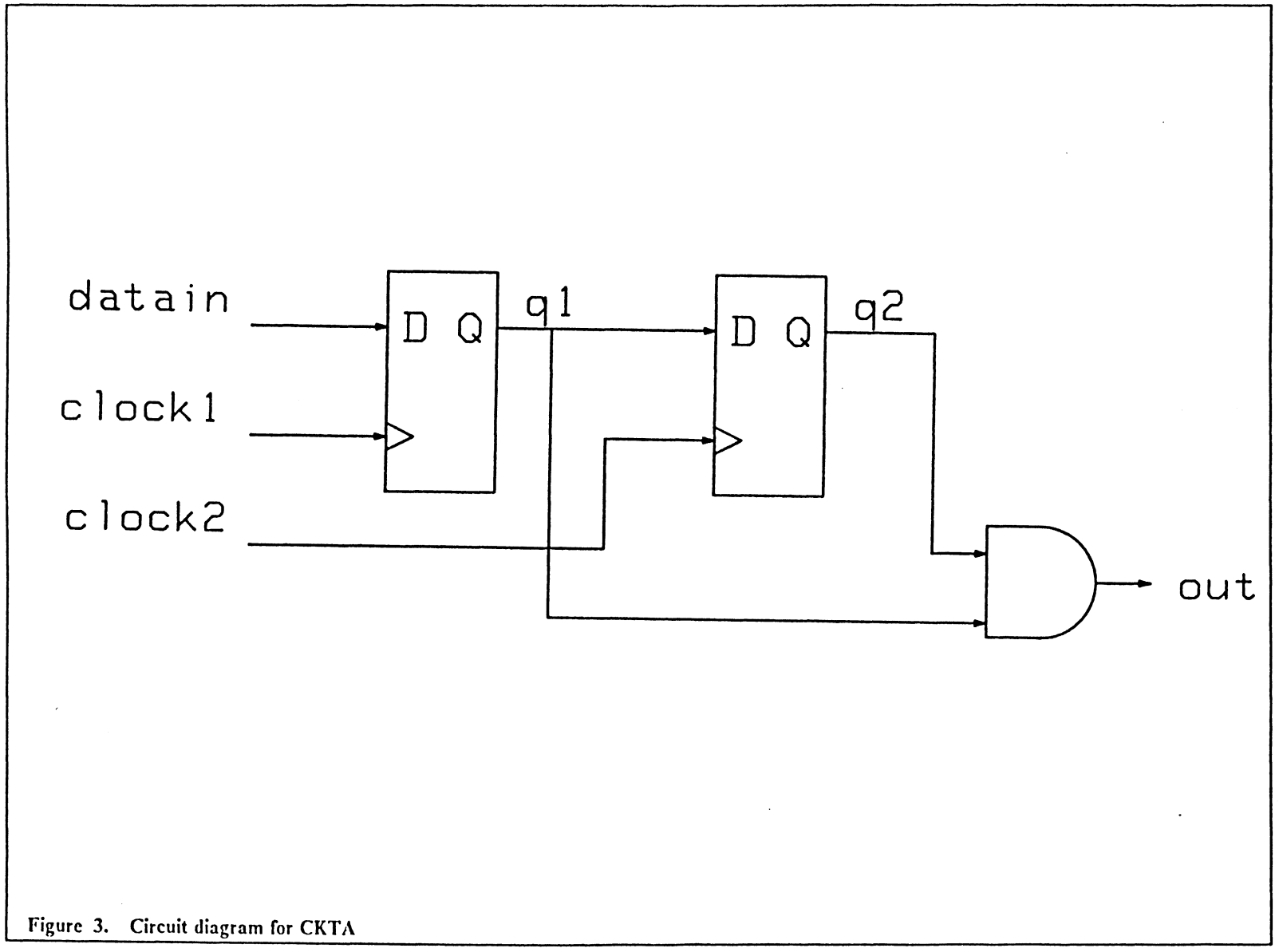


Figure 3. Circuit diagram for CKTA

CKTA Fault List

"Circuit A"	
(1 STUCKTHEN s1)	Excluded - continuous clocking (but successful)
(2 STUCKELSE s1)	Failed; STUCK prevents preload
(3 MICROOP (s1 - "") BITAND BITOR)	Excluded - high or edge
(4 MICROOP (s1 - L) BITEQV BITXOR)	Excluded - inverted clock
(5 MICROOP (s1 - R) BITNOT BITBUF)	Excluded - clocks on high unless edge
(6 ASSNCNTL s2)	Overflow; many 2-phase tries
(7 STUCKTHEN s3)	Excluded - continuous clocking
(8 STUCKELSE s3)	Successful, but 2-phase
(9 MICROOP (s3 - "") BITAND BITOR)	Excluded - high or edge
(10 MICROOP (s3 - L) BITEQV BITXOR)	Excluded - inverted clock
(11 MICROOP (s3 - R) BITNOT BITBUF)	Excluded - clocks on high unless edge
(12 ASSNCNTL s4)	Successful
(13 ASSNCNTL s5)	Overflow; bad time choices; two-phase slows conflict det
(14 MICROOP (s5 - "") BITAND BITOR)	Successful
(15 STUCKDATA (s1 - "") (BIT "0"))	Failed; faulty micro-op
(16 STUCKDATA (s1 - "") (BIT "1"))	Excluded - continuous clocking
(17 STUCKDATA (s1 - L) (BIT "0"))	Failed; faulty micro-op
(18 STUCKDATA (s1 - L) (BIT "1"))	Excluded - clocks on any edge
(19 STUCKDATA (s1 - LL) (BIT "0"))	Failed; faulty micro-op
(20 STUCKDATA (s1 - LL) (BIT "1"))	Excluded - clocks on any edge
(21 STUCKDATA (s1 - LR) (BIT "0"))	Excluded - inverted clock
(22 STUCKDATA (s1 - LR) (BIT "1"))	Not a fault
(23 STUCKDATA (s1 - R) (BIT "0"))	Failed; faulty micro-op
(24 STUCKDATA (s1 - R) (BIT "1"))	Excluded - clocks on high
(25 STUCKDATA (s1 - RL) (BIT "0"))	Excluded - clocks on high
(26 STUCKDATA (s1 - RL) (BIT "1"))	Failed; faulty micro-op
(27 STUCKDATA (s2 - "") (BIT "0"))	Failed; faulty micro-op
(28 STUCKDATA (s2 - "") (BIT "1"))	Failed; faulty micro-op
(29 STUCKDATA (s3 - "") (BIT "0"))	Failed; faulty micro-op
(30 STUCKDATA (s3 - "") (BIT "1"))	Excluded - continuous clocking
(31 STUCKDATA (s3 - L) (BIT "0"))	Failed; faulty micro-op
(32 STUCKDATA (s3 - L) (BIT "1"))	Excluded - clocks on any edge
(33 STUCKDATA (s3 - LL) (BIT "0"))	Failed; faulty micro-op
(34 STUCKDATA (s3 - LL) (BIT "1"))	Excluded - clocks on any edge
(35 STUCKDATA (s3 - LR) (BIT "0"))	Excluded - inverted clock
(36 STUCKDATA (s3 - LR) (BIT "1"))	Not a fault
(37 STUCKDATA (s3 - R) (BIT "0"))	Failed; faulty micro-op
(38 STUCKDATA (s3 - R) (BIT "1"))	Excluded - clocks on high
(39 STUCKDATA (s3 - RL) (BIT "0"))	Excluded - clocks on high
(40 STUCKDATA (s3 - RL) (BIT "1"))	Failed; faulty micro-op
(41 STUCKDATA (s4 - "") (BIT "0"))	Successful
(42 STUCKDATA (s4 - "") (BIT "1"))	Successful
(43 STUCKDATA (s5 - "") (BIT "0"))	Successful
(44 STUCKDATA (s5 - "") (BIT "1"))	Successful
(45 STUCKDATA (s5 - L) (BIT "0"))	Successful
(46 STUCKDATA (s5 - L) (BIT "1"))	Successful
(47 STUCKDATA (s5 - R) (BIT "0"))	Successful
(48 STUCKDATA (s5 - R) (BIT "1"))	Successful

CKTCV Description

```
; (See diagram, Figure 4 on page 105.)

; File CKTCV.HDL

;    ENTITY RegMux(
;         clock : IN BIT;
;         cmd,
;         inp
;          : IN BIT_VECTOR(1 DOWNTO 0);
;         c : OUT VIT_VECTOR(1 DOWNTO 0)
;         ) IS
;    END RegMux;

;    ARCHITECTURE Arch OF RegMux IS
;
;    PROCESS(clock,cmd,inp,a,b)
;    BEGIN
; s1:   IF clock='1' AND NOT clock'STABLE THEN
; s2:      CASE cmd IS
; s3:         WHEN "00" = > p < = inp;   -- load control register p
; s4:         WHEN "01" = > a < = inp;   -- load a register
; s5:         WHEN "10" = > b < = inp;   -- load b register
;             WHEN "11" = >              -- load c according to p
; s6:            IF p="00" THEN
; s7:               c < = a;
;               ELSE
; s8:               c < = b;
;               END IF;
;         END CASE;
;       END IF;
;    END PROCESS;
;    END Arch;
(assert

((fileprefix "cktcv"))
((modelname "Circuit V - Latched Multiplexer with Vectors"))

((datatype clock BIT)) ((statevar clock t)) ((inputpin clock))

((datatype inp BV)) ((bvlength inp 2)) ((statevar inp t))
((inputpin inp))

((datatype cmd BV)) ((bvlength cmd 2)) ((statevar cmd t))
((inputpin cmd))

((datatype a BV)) ((bvlength a 2)) ((statevar a t))

((datatype b BV)) ((bvlength b 2)) ((statevar b t))

((datatype p BV)) ((bvlength p 2)) ((statevar p t))

((datatype c BV)) ((bvlength c 2)) ((statevar c t))
((outputpin c))
```

```
((statement s1))
((statementtype s1 IF))
((controlexpression s1 (BITAND (BITEQV (OBJ clock) (LIT (BIT "1")))
                        (BITNOT (STABLE (OBJ clock))) ) ))
((subordinaterange s1 THEN (s2) ))
((subordinaterange s1 ELSE nil ))
((understable2 s1 clock))

((statement s2))
((statementtype s2 CASE))
((controlexpression s2 (OBJ cmd)))
((subordinaterange s2 ((BV "00")) (s3) ))
((subordinaterange s2 ((BV "01")) (s4) ))
((subordinaterange s2 ((BV "10")) (s5) ))
((subordinaterange s2 ((BV "11")) (s6) ))
((understable s2))
((understable2 s2 clock))

((statement s3))
((statementtype s3 ASSIGNMENT))
((sourceexpression s3 (OBJ inp)))
((destinationobject s3 p))
((understable s3))
((understable2 s3 clock))

((statement s4))
((statementtype s4 ASSIGNMENT))
((sourceexpression s4 (OBJ inp)))
((destinationobject s4 a))
((understable s4))
((understable2 s4 clock))

((statement s5))
((statementtype s5 ASSIGNMENT))
((sourceexpression s5 (OBJ inp)))
((destinationobject s5 b))
((understable s5))
((understable2 s5 clock))

((statement s6))
((statementtype s6 IF))
((controlexpression s6 (BVEQ (OBJ p) (LIT (BV "00"))) ))
((subordinaterange s6 THEN (s7) ))
((subordinaterange s6 ELSE (s8) ))
((understable s6))
((understable2 s6 clock))

((statement s7))
((statementtype s7 ASSIGNMENT))
((sourceexpression s7 (OBJ a)))
((destinationobject s7 c))
((understable s7))
((understable2 s7 clock))
```

```
((statement s8))
((statementtype s8 ASSIGNMENT))
((sourceexpression s8 (OBJ b)))
((destinationobject s8 c))
((understable s8))
((understable2 s8 clock))

) ; end assert
```

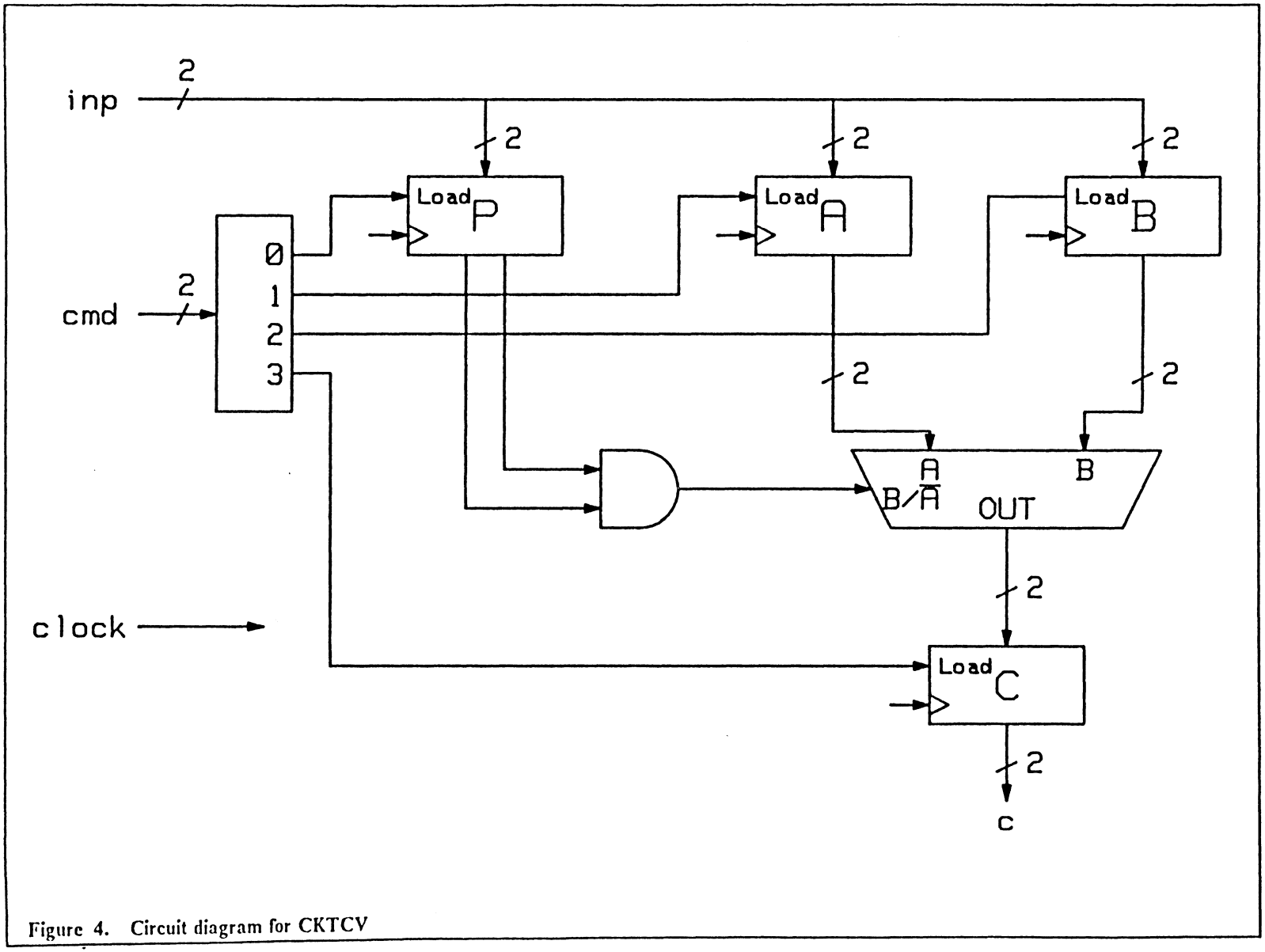


Figure 4. Circuit diagram for CKTCV

CKTCV Fault List

"Circuit V - Latched Multiplexor with Vectors"

(1 STUCKTHEN s1)	Excluded - continuous clocking
(2 STUCKELSE s1)	Failed; STUCK prevents EXEC for preload
(3 MICROOP (s1 - "") BITAND BITOR)	Excluded - clocks on high or edge
(4 MICROOP (s1 - L) BITEQV BITXOR)	Excluded - inverted clock
(5 MICROOP (s1 - R) BITNOT BITBUF)	Excluded - clocks on high unless edge
(6 DEADCLAUSE s2 ((BV "11")))	Successful (contorted, but works)
(7 DEADCLAUSE s2 ((BV "10")))	Successful
(8 DEADCLAUSE s2 ((BV "01")))	Successful
(9 DEADCLAUSE s2 ((BV "00")))	Successful
(10 ASSNCNTL s3)	Successful
(11 ASSNCNTL s4)	Successful
(12 ASSNCNTL s5)	Successful
(13 STUCKTHEN s6)	Successful
(14 STUCKELSE s6)	Successful
(15 MICROOP (s6 - "") BVEQ BVNEQ)	Successful
(16 ASSNCNTL s7)	Successful
(17 ASSNCNTL s8)	Successful, but 2-phase when not necessary
(18 STUCKDATA (s1 - "") (BIT "0"))	Failed; faulty micro-op
(19 STUCKDATA (s1 - "") (BIT "1"))	Excluded - continuous clocking
(20 STUCKDATA (s1 - L) (BIT "0"))	Failed; faulty micro-op
(21 STUCKDATA (s1 - L) (BIT "1"))	Excluded - clocks on any edge
(22 STUCKDATA (s1 - LL) (BIT "0"))	Failed; faulty micro-op
(23 STUCKDATA (s1 - LL) (BIT "1"))	Excluded - clocks on any edge
(24 STUCKDATA (s1 - LR) (BIT "0"))	Excluded - inverted clock
(25 STUCKDATA (s1 - LR) (BIT "1"))	Not a fault
(26 STUCKDATA (s1 - R) (BIT "0"))	Failed; faulty micro-op
(27 STUCKDATA (s1 - R) (BIT "1"))	Excluded - clocks on high
(28 STUCKDATA (s1 - RL) (BIT "0"))	Excluded - clocks on high
(29 STUCKDATA (s1 - RL) (BIT "1"))	Failed; faulty micro-op
(30 STUCKDATA (s2 - "") (BV "00"))	Failed; faulty micro-op
(31 STUCKDATA (s2 - "") (BV "11"))	Failed; faulty micro-op
(32 STUCKDATA (s3 - "") (BV "00"))	Successful
(33 STUCKDATA (s3 - "") (BV "11"))	Successful
(34 STUCKDATA (s4 - "") (BV "00"))	Successful
(35 STUCKDATA (s4 - "") (BV "11"))	Successful
(36 STUCKDATA (s5 - "") (BV "00"))	Successful
(37 STUCKDATA (s5 - "") (BV "11"))	Successful
(38 STUCKDATA (s6 - "") (BIT "0"))	Successful
(39 STUCKDATA (s6 - "") (BIT "1"))	Successful
(40 STUCKDATA (s6 - L) (BV "00"))	Successful
(41 STUCKDATA (s6 - L) (BV "11"))	Successful
(42 STUCKDATA (s6 - R) (BV "00"))	Not a fault
(43 STUCKDATA (s6 - R) (BV "11"))	Successful
(44 STUCKDATA (s7 - "") (BV "00"))	Successful
(45 STUCKDATA (s7 - "") (BV "11"))	Successful
(46 STUCKDATA (s8 - "") (BV "00"))	Successful
(47 STUCKDATA (s8 - "") (BV "11"))	Successful

CNTR Description

```
;    ENTITY Counter(
;         clrbar,
;         clock : IN BIT;
;         q1,
;         q2,
;         q3 : OUT BIT) IS
;    END Counter;
;
;    ARCHITECTURE Arch OF Counter IS
;
;    PROCESS(clrbar,clock)
;    BEGIN
; s1:   IF clrbar='0' THEN
; s2:     q1 <= '0';
; s3:     q2 <= '0';
; s4:     q3 <= '0';
;       ELSE
; s5:     IF clock='1' AND NOT clock'STABLE THEN
; s6:       q1 <= NOT q1;
; s7:       q3 <= q2 XOR q1;
; s8:       q3 <= q3 XOR (q1 AND q2);
;         END IF
;       END IF
;
;    END PROCESS;
;
;    END Arch;
(assert

((fileprefix "cntr"))
((modelname "Three-Bit Counter 1"))

((datatype clrbar BIT)) ((statevar clrbar t)) ((inputpin clrbar))

((datatype clock BIT)) ((statevar clock t)) ((inputpin clock))

((datatype q1 BIT)) ((statevar q1 t)) ((outputpin q1))

((datatype q2 BIT)) ((statevar q2 t)) ((outputpin q2))

((datatype q3 BIT)) ((statevar q3 t)) ((outputpin q3))

((statement s1))
((statementtype s1 IF))
((controlexpression s1 (BITEQV (OBJ clrbar) (LIT (BIT "0"))) ))
((subordinaterange s1 THEN (s2 s3 s4) ))
((subordinaterange s1 ELSE (s5) ))
((understable2 s1 nil))

((statement s2))
((statementtype s2 ASSIGNMENT))
((sourceexpression s2 (LIT (BIT "0")) ))
```

```
((destinationobject s2 q1))
((understable2 s2 nil))

((statement s3))
((statementtype s3 ASSIGNMENT))
((sourceexpression s3 (LIT (BIT "0")) ))
((destinationobject s3 q2))
((understable2 s3 nil))

((statement s4))
((statementtype s4 ASSIGNMENT))
((sourceexpression s4 (LIT (BIT "0")) ))
((destinationobject s4 q3))
((understable2 s4 nil))

((statement s5))
((statementtype s5 IF))
((controlexpression s5 (BITAND (BITEQV (OBJ clock) (LIT (BIT "1")))
                               (BITNOT (STABLE (OBJ clock))) ) ))
((subordinaterange s5 THEN (s6 s7 s8) ))
((subordinaterange s5 ELSE nil ))
((understable2 s5 clock))

((statement s6))
((statementtype s6 ASSIGNMENT))
((sourceexpression s6 (BITNOT (OBJ q1)) ))
((destinationobject s6 q1))
((understable s6))
((understable2 s6 clock))

((statement s7))
((statementtype s7 ASSIGNMENT))
((sourceexpression s7 (BITXOR (OBJ q2) (OBJ q1)) ))
((destinationobject s7 q2))
((understable s7))
((understable2 s7 clock))

((statement s8))
((statementtype s8 ASSIGNMENT))
((sourceexpression s8 (BITXOR (OBJ q3) (BITAND (OBJ q1) (OBJ q2))) ))
((destinationobject s8 q3))
((understable s8))
((understable2 s8 clock))

) ; end assert
```

CNTR Fault List

"Three-BIT Counter 1"	
(1 STUCKTHEN s1)	Failed; can't DNE under STUCK
(2 STUCKELSE s1)	Failed; loop cuts
(3 MICROOP (s1 - "") BITEQV BITXOR)	Failed; faulty micro-op
(4 ASSNCNTL s2)	Failed; loop cuts; many tries
(5 ASSNCNTL s3)	Successful
(6 ASSNCNTL s4)	Failed; loop cuts
(7 STUCKTHEN s5)	Excluded - continuous clocking
(8 STUCKELSE s5)	Successful
(9 MICROOP (s5 - "") BITAND BITOR)	Excluded - clocks on any edge or high
(10 MICROOP (s5 - L) BITEQV BITXOR)	Excluded - inverted clock
(11 MICROOP (s5 - R) BITNOT BITBUF)	Excluded - when high (unless edge)
(12 ASSNCNTL s6)	Successful
(13 MICROOP (s6 - "") BITNOT BITBUF)	Successful
(14 ASSNCNTL s7)	Failed; loop cuts
(15 MICROOP (s7 - "") BITXOR BITEQV)	Overflow; bad inequality choice
(16 ASSNCNTL s8)	Failed; loop cuts
(17 MICROOP (s8 - "") BITXOR BITEQV)	Failed; loop cuts
(18 MICROOP (s8 - R) BITAND BITOR)	Failed; loop cuts
(19 STUCKDATA (s1 - "") (BIT "0"))	Failed; faulty micro-operation
(20 STUCKDATA (s1 - "") (BIT "1"))	Failed; faulty micro-operation
(21 STUCKDATA (s1 - L) (BIT "0"))	Failed; faulty micro-operation
(22 STUCKDATA (s1 - L) (BIT "1"))	Failed; faulty micro-operation
(23 STUCKDATA (s1 - R) (BIT "0"))	Not a fault
(24 STUCKDATA (s1 - R) (BIT "1"))	Failed; faulty micro-operation
(25 STUCKDATA (s2 - "") (BIT "0"))	Not a fault
(26 STUCKDATA (s2 - "") (BIT "1"))	Successful
(27 STUCKDATA (s3 - "") (BIT "0"))	Not a fault
(28 STUCKDATA (s3 - "") (BIT "1"))	Successful
(29 STUCKDATA (s4 - "") (BIT "0"))	Not a fault
(30 STUCKDATA (s4 - "") (BIT "1"))	Successful
(31 STUCKDATA (s5 - "") (BIT "0"))	Successful
(32 STUCKDATA (s5 - "") (BIT "1"))	Excluded - continuous clocking
(33 STUCKDATA (s5 - L) (BIT "0"))	Successful
(34 STUCKDATA (s5 - L) (BIT "1"))	Excluded - clocks on any edge
(35 STUCKDATA (s5 - LL) (BIT "0"))	Successful
(36 STUCKDATA (s5 - LL) (BIT "1"))	Excluded - clocks on any edge
(37 STUCKDATA (s5 - LR) (BIT "0"))	Excluded - inverted clock
(38 STUCKDATA (s5 - LR) (BIT "1"))	Not a fault
(39 STUCKDATA (s5 - R) (BIT "0"))	Successful
(40 STUCKDATA (s5 - R) (BIT "1"))	Excluded - clocks on high
(41 STUCKDATA (s5 - RL) (BIT "0"))	Excluded - clocks on high
(42 STUCKDATA (s5 - RL) (BIT "1"))	Successful
(43 STUCKDATA (s6 - "") (BIT "0"))	Successful
(44 STUCKDATA (s6 - "") (BIT "1"))	Failed; faulty micro-op
(45 STUCKDATA (s6 - L) (BIT "0"))	Failed; faulty micro-op
(46 STUCKDATA (s6 - L) (BIT "1"))	Successful
(47 STUCKDATA (s7 - "") (BIT "0"))	Failed; faulty micro-op
(48 STUCKDATA (s7 - "") (BIT "1"))	Successful
(49 STUCKDATA (s7 - L) (BIT "0"))	Failed; faulty micro-op
(50 STUCKDATA (s7 - L) (BIT "1"))	Failed; faulty micro-op
(51 STUCKDATA (s7 - R) (BIT "0"))	Failed; faulty micro-operation
(52 STUCKDATA (s7 - R) (BIT "1"))	Successful

(53 STUCKDATA (s8 - "") (BIT "0"))	Failed; faulty micro-operation
(54 STUCKDATA (s8 - "") (BIT "1"))	Successful
(55 STUCKDATA (s8 - L) (BIT "0"))	Failed; faulty micro-operation
(56 STUCKDATA (s8 - L) (BIT "1"))	Succeeded (with help)
(57 STUCKDATA (s8 - R) (BIT "0"))	Failed; faulty micro-operation
(58 STUCKDATA (s8 - R) (BIT "1"))	Successful
(59 STUCKDATA (s8 - RL) (BIT "0"))	Failed; faulty micro-operation
(60 STUCKDATA (s8 - RL) (BIT "1"))	Failed; faulty micro-operation
(61 STUCKDATA (s8 - RR) (BIT "0"))	Failed; faulty micro-operation
(62 STUCKDATA (s8 - RR) (BIT "1"))	Failed; faulty micro-operation

CNTRV Description

```
;    ENTITY CounterV(
;          clrbar,
;          clock
;           : IN BIT;
;          count : OUT BIT_VECTOR(2 DOWNTO 0)
;     ) IS
;    END CounterV;
;
;    ARCHITECTURE Arch OF Counterv IS
;
;    PROCESS(clrbar,clock)
;
;    BEGIN
; s1:   IF clrbar='0' THEN
; s2:      count <= "000";
;       ELSE
; s3:      IF clock='1' AND NOT clock'STABLE THEN
; s4:         count <= BVADD(count,"001")
;          END IF
;       END IF
;    END BLOCK;
;
;    END Arch;
(assert

((fileprefix "cntrv"))
((modelname "Three-Bit Vector Counter"))

((datatype clrbar BIT)) ((statevar clrbar t)) ((inputpin clrbar))

((datatype clock BIT)) ((statevar clock t)) ((inputpin clock))

((datatype count BV)) ((bvlength count 3)) ((statevar count t))
((outputpin count))


((statement s1))
((statementtype s1 IF))
((controlexpression s1 (BITEQV (OBJ clrbar) (LIT (BIT "0"))) ))
((subordinaterange s1 THEN (s2) ))
((subordinaterange s1 ELSE (s3) ))
((understable2 s1 nil))

((statement s2))
((statementtype s2 ASSIGNMENT))
((sourceexpression s2 (LIT (BV "000")) ))
((destinationobject s2 count))
((understable2 s2 nil))

((statement s3))
((statementtype s3 IF))
((controlexpression s3 (BITAND (BITEQV (OBJ clock) (LIT (BIT "1")))
```

```
                    (BITNOT (STABLE (OBJ clock))) ) ))
((subordinaterange s3 THEN (s4) ))
((subordinaterange s3 ELSE nil ))
((understable2 s3 clock))

((statement s4))
((statementtype s4 ASSIGNMENT))
((sourceexpression s4 (BVADD (OBJ count) (LIT (BV "001"))) ))
((destinationobject s4 count))
((understable s4))
((understable2 s4 clock))

) ; end assert
```

CNTRV Fault List

"Three-BIT Vector Counter"

(1 STUCKTHEN s1)	Failed; can't DNE under STUCK
(2 STUCKELSE s1)	Wrong; BVADD/BVSUB not fully implemented
(3 MICROOP (s1 - "") BITEQV BITXOR)	Failed; faulty micro-op; tries many
(4 ASSNCNTL s2)	Successful
(5 STUCKTHEN s3)	Excluded - continuous clocking
(6 STUCKELSE s3)	Wrong; BVADD/BVSUB not fully implemented
(7 MICROOP (s3 - "") BITAND BITOR)	Excluded - clocks on high or edge
(8 MICROOP (s3 - L) BITEQV BITXOR)	Excluded - inverted clock
(9 MICROOP (s3 - R) BITNOT BITBUF)	Excluded - clocks on high unless edge
(10 ASSNCNTL s4)	Overflowed interpreter
(11 MICROOP (s4 - "") BVADD BVSUB)	Successful
(12 MICROOP (s4 - "") BVADD BVXOR)	Failed; faulty micro-op
(13 STUCKDATA (s1 - "") (BIT "0"))	Failed; faulty micro-op
(14 STUCKDATA (s1 - "") (BIT "1"))	Failed; faulty micro-op
(15 STUCKDATA (s1 - L) (BIT "0"))	Failed; faulty micro-op
(16 STUCKDATA (s1 - L) (BIT "1"))	Failed; faulty micro-op
(17 STUCKDATA (s1 - R) (BIT "0"))	Not a fault
(18 STUCKDATA (s1 - R) (BIT "1"))	Failed; faulty micro-op
(19 STUCKDATA (s2 - "") (BV "000"))	Not a fault
(20 STUCKDATA (s2 - "") (BV "111"))	Successful
(21 STUCKDATA (s3 - "") (BIT "0"))	Wrong; BVADD/BVSUB not fully implemented
(22 STUCKDATA (s3 - "") (BIT "1"))	Excluded - continuous clocking
(23 STUCKDATA (s3 - L) (BIT "0"))	Wrong; BVADD/BVSUB not fully implemented
(24 STUCKDATA (s3 - L) (BIT "1"))	Excluded - clocks on any edge
(25 STUCKDATA (s3 - LL) (BIT "0"))	Wrong; BVADD/BVSUB not fully implemented
(26 STUCKDATA (s3 - LL) (BIT "1"))	Excluded - clocks on any edge
(27 STUCKDATA (s3 - LR) (BIT "0"))	Excluded - inverted clock
(28 STUCKDATA (s3 - LR) (BIT "1"))	Not a fault
(29 STUCKDATA (s3 - R) (BIT "0"))	Wrong; BVADD/BVSUB not fully implemented
(30 STUCKDATA (s3 - R) (BIT "1"))	Excluded - clocks on high
(31 STUCKDATA (s3 - RL) (BIT "0"))	Excluded - clocks on high
(32 STUCKDATA (s3 - RL) (BIT "1"))	Wrong; BVADD/BVSUB not fully implemented
(33 STUCKDATA (s4 - "") (BV "000"))	Failed; faulty micro-op
(34 STUCKDATA (s4 - "") (BV "111"))	Failed; faulty micro-op
(35 STUCKDATA (s4 - L) (BV "000"))	Failed; faulty micro-op
(36 STUCKDATA (s4 - L) (BV "111"))	Successful
(37 STUCKDATA (s4 - R) (BV "000"))	Basic test invalid
(38 STUCKDATA (s4 - R) (BV "111"))	Basic test invalid

DFF Description

```
; File dff.HDL - internal form for D flip-flop  description

; D Flip-flop VHDL Description
;    ENTITY DFF(
;          clrbar,
;          setbar,
;          data,
;          clock: IN BIT;
;          q,
;          qbar: OUT BIT
;          ) IS
;    END DFF;
;
;    ARCHITECTURE Arch OF DFF IS
;
;       PROCESS(clrbar,setbar,data,clock)
;       BEGIN
; s2:       IF clrbar = 0 THEN
; s3:          q < = 0;
; s4:          qbar < = 1;
;          ELSE
; s5:          IF setbar = 0 THEN
; s6:             q < = 1;
; s7:             qbar < = 0;
;             ELSE
; s8:             IF (NOT clock'STABLE) AND (clock = 1) THEN
; s9:                q < = data;
; s10:               qbar < = NOT data;
;                END IF
;             END IF
;          END IF
;       END PROCESS;
;    END Arch;
(assert

((fileprefix "dff"))
((modelname "D Flip-Flop with Set and Clear"))

((datatype clrbar BIT)) ((statevar clrbar t)) ((inputpin clrbar))

((datatype setbar BIT)) ((statevar setbar t)) ((inputpin setbar))

((datatype clock BIT)) ((statevar clock t)) ((inputpin clock))

((datatype data BIT)) ((statevar data t)) ((inputpin data))

((datatype q BIT)) ((statevar q t)) ((outputpin q))

((datatype qbar BIT)) ((statevar qbar t)) ((outputpin qbar))

((statement s2))
((statementtype s2 IF))
((controlexpression s2 (BITEQV (OBJ clrbar)
```

```
                    (LIT (BIT "0"))
                ) ))
((subordinaterange s2 THEN (s3 s4) ))
((subordinaterange s2 ELSE (s5) ))
((understable2 s2 nil))

((statement s3))
((statementtype s3 ASSIGNMENT))
  ((sourceexpression s3 (LIT (BIT "0")) ))
  ((destinationobject s3 q))
((understable2 s3 nil))

((statement s4))
((statementtype s4 ASSIGNMENT))
  ((sourceexpression s4 (LIT (BIT "1")) ))
  ((destinationobject s4 qbar))
((understable2 s4 nil))

((statement s5))
((statementtype s5 IF))
((controlexpression s5 (BITEQV (OBJ setbar)
                    (LIT (BIT "0"))
                ) ))
  ((subordinaterange s5 THEN (s6 s7) ))
  ((subordinaterange s5 ELSE (s8) ))
((understable2 s5 nil))

((statement s6))
((statementtype s6 ASSIGNMENT))
  ((sourceexpression s6 (LIT (BIT "1")) ))
  ((destinationobject s6 q))
((understable2 s6 nil))

((statement s7))
((statementtype s7 ASSIGNMENT))
  ((sourceexpression s7 (LIT (BIT "0")) ))
  ((destinationobject s7 qbar))
((understable2 s7 nil))

((statement s8))
((statementtype s8 IF))
((controlexpression s8 (BITAND (BITNOT (STABLE (OBJ clock)))
                    (BITEQV (OBJ clock)
                        (LIT (BIT "1")) )
                ) ))
((subordinaterange s8 THEN (s9 s10) ))
((subordinaterange s8 ELSE () ))
((understable2 s8 clock))

((statement s9))
((statementtype s9 ASSIGNMENT))
((sourceexpression s9 (OBJ data) ))
((destinationobject s9 q))
((understable s9))
((understable2 s9 clock))
```

```
((statement s10))
((statementtype s10 ASSIGNMENT))
((sourceexpression s10 (BITNOT (OBJ data)) ))
((destinationobject s10 qbar))
((understable s10))
((understable2 s10 clock))

) ; end assert
```

DFF Fault List

"D Flip-Flop with Set and Clear"

(1 STUCKTHEN s2)	Successful
(2 STUCKELSE s2)	Successful
(3 MICROOP (s2 - "") BITEQV BITXOR)	Successful
(4 ASSNCNTL s3)	Successful
(5 ASSNCNTL s4)	Successful
(6 STUCKTHEN s5)	Successful
(7 STUCKELSE s5)	Successful
(8 MICROOP (s5 - "") BITEQV BITXOR)	Successful
(9 ASSNCNTL s6)	Successful
(10 ASSNCNTL s7)	Successful
(11 STUCKTHEN s8)	Excluded - continuous clocking
(12 STUCKELSE s8)	Successful
(13 MICROOP (s8 - "") BITAND BITOR)	Excluded - clocks on high or edge
(14 MICROOP (s8 - L) BITNOT BITBUF)	Excluded - clocks on high unless edge
(15 MICROOP (s8 - R) BITEQV BITXOR)	Excluded - inverted clock
(16 ASSNCNTL s9)	Successful, but 2-phase before trying others
(17 ASSNCNTL s10)	Successful, but 2-phase before trying others
(18 MICROOP (s10 - "") BITNOT BITBUF)	Successful
(19 STUCKDATA (s2 - "") (BIT "0"))	Successful
(20 STUCKDATA (s2 - "") (BIT "1"))	Successful
(21 STUCKDATA (s2 - L) (BIT "0"))	Successful
(22 STUCKDATA (s2 - L) (BIT "1"))	Successful
(23 STUCKDATA (s2 - R) (BIT "0"))	Not a fault
(24 STUCKDATA (s2 - R) (BIT "1"))	Successful
(25 STUCKDATA (s3 - "") (BIT "0"))	Not a fault
(26 STUCKDATA (s3 - "") (BIT "1"))	Successful
(27 STUCKDATA (s4 - "") (BIT "0"))	Successful
(28 STUCKDATA (s4 - "") (BIT "1"))	Not a fault
(29 STUCKDATA (s5 - "") (BIT "0"))	Successful
(30 STUCKDATA (s5 - "") (BIT "1"))	Successful
(31 STUCKDATA (s5 - L) (BIT "0"))	Successful
(32 STUCKDATA (s5 - L) (BIT "1"))	Successful
(33 STUCKDATA (s5 - R) (BIT "0"))	Not a fault
(34 STUCKDATA (s5 - R) (BIT "1"))	Successful
(35 STUCKDATA (s6 - "") (BIT "0"))	Successful
(36 STUCKDATA (s6 - "") (BIT "1"))	Not a fault
(37 STUCKDATA (s7 - "") (BIT "0"))	Not a fault
(38 STUCKDATA (s7 - "") (BIT "1"))	Successful
(39 STUCKDATA (s8 - "") (BIT "0"))	Successful
(40 STUCKDATA (s8 - "") (BIT "1"))	Excluded - continuous clocking
(41 STUCKDATA (s8 - L) (BIT "0"))	Successful
(42 STUCKDATA (s8 - L) (BIT "1"))	Excluded - clocks on high
(43 STUCKDATA (s8 - LL) (BIT "0"))	Excluded - clocks on high
(44 STUCKDATA (s8 - LL) (BIT "1"))	Successful
(45 STUCKDATA (s8 - R) (BIT "0"))	Successful
(46 STUCKDATA (s8 - R) (BIT "1"))	Excluded - clocks on any edge
(47 STUCKDATA (s8 - RL) (BIT "0"))	Successful
(48 STUCKDATA (s8 - RL) (BIT "1"))	Excluded - clocks on any edge
(49 STUCKDATA (s8 - RR) (BIT "0"))	Excluded - inverted clock
(50 STUCKDATA (s8 - RR) (BIT "1"))	Not a fault
(51 STUCKDATA (s9 - "") (BIT "0"))	Successful
(52 STUCKDATA (s9 - "") (BIT "1"))	Successful

(53 STUCKDATA (s10 - "") (BIT "0"))	Successful
(54 STUCKDATA (s10 - "") (BIT "1"))	Successful
(55 STUCKDATA (s10 - L) (BIT "0"))	Successful
(56 STUCKDATA (s10 - L) (BIT "1"))	Successful

FNTST Description

```
;    ENTITY FnTst
;     (
;      in : IN BIT;
;      out : OUT BIT) IS
;    END FnTst;
;
;    ARCHITECTURE Arch OF FnTst IS
;
;    PROCESS(in,a,x,y)
;      SIGNAL a,x,y : BIT;
;    BEGIN
; s1:   a < = in;
; s2:   x < = a;
; s3:   y < = a;
; s4:   out < = x AND y'
;    END BLOCK;
;
;    END Arch;
(assert

((fileprefix "fntst"))
((modelname "Test model for reconvergent fanout"))

((datatype in BIT)) ((statevar in t)) ((inputpin in))

((datatype a BIT)) ((statevar a nil))

((datatype x BIT)) ((statevar x nil))

((datatype y BIT)) ((statevar y nil))

((datatype out BIT)) ((statevar out nil)) ((outputpin out))

((statement s1))
((statementtype s1 ASSIGNMENT))
((sourceexpression s1 (OBJ in) ))
((destinationobject s1 a))
((understable2 s1 nil))

((statement s2))
((statementtype s2 ASSIGNMENT))
((sourceexpression s2 (OBJ a) ))
((destinationobject s2 x))
((understable2 s2 nil))

((statement s3))
((statementtype s3 ASSIGNMENT))
((sourceexpression s3 (OBJ a) ))
((destinationobject s3 y))
((understable2 s3 nil))

((statement s4))
((statementtype s4 ASSIGNMENT))
```

```
((sourceexpression s4 (BITAND (OBJ x) (OBJ y)) ))
((destinationobject s4 out))
((understable2 s4 nil))

) ; end assert
```

FNTST Fault List

"Test model for reconvergent fanout"	
(1 ASSNCNTL s1)	Failed; reconvergent fanout
(2 ASSNCNTL s2)	Failed; redundant fault
(3 ASSNCNTL s3)	Failed; redundant fault
(4 ASSNCNTL s4)	Successful
(5 MICROOP (s4 - "") BITAND BITOR)	Failed; redundant fault
(6 STUCKDATA (s1 - "") (BIT "0"))	Failed; reconvergent fanout
(7 STUCKDATA (s1 - "") (BIT "1"))	Failed; reconvergent fanout
(8 STUCKDATA (s2 - "") (BIT "0"))	Successful
(9 STUCKDATA (s2 - "") (BIT "1"))	Failed; redundant fault
(10 STUCKDATA (s3 - "") (BIT "0"))	Successful
(11 STUCKDATA (s3 - "") (BIT "1"))	Failed; redundant fault
(12 STUCKDATA (s4 - "") (BIT "0"))	Successful
(13 STUCKDATA (s4 - "") (BIT "1"))	Successful
(14 STUCKDATA (s4 - L) (BIT "0"))	Successful
(15 STUCKDATA (s4 - L) (BIT "1"))	Failed; redundant fault
(16 STUCKDATA (s4 - R) (BIT "0"))	Successful
(17 STUCKDATA (s4 - R) (BIT "1"))	Failed; redundant fault

PRTY Description

```
; File PRTY.HDL - sample parity generator
;    ENTITY Parity(
;          a : IN BIT_VECTOR(7 DOWNTO 0);
;          p : OUT BIT) IS
;    END Parity;
;
;    ARCHITECTURE Arch OF parity IS
;
;    PROCESS(a)
;    BEGIN
; s1:    out < = ((a(0) xor a(1)) xor (a(2) xor a(3))) xor
;              ((a(4) xor a(5)) xor (a(6) xor a(7)));
;    END PROCESS;
;
;    END Arch;
(assert

((fileprefix "PRTY"))
((modelname "Parity generator"))

((datatype in BV)) ((bvlength in 8)) ((statevar in t))
((inputpin in))

((datatype out BIT)) ((statevar out nil)) ((outputpin out))

((statement s1))
((statementtype s1 ASSIGNMENT))
((destinationobject s1 out))
((sourceexpression s1 (BITXOR (BITXOR (BITXOR ((BVSUBV 0) (OBJ in))
                                              ((BVSUBV 1) (OBJ in)))
                                      (BITXOR ((BVSUBV 2) (OBJ in))
                                              ((BVSUBV 3) (OBJ in))))
                              (BITXOR (BITXOR ((BVSUBV 4) (OBJ in))
                                              ((BVSUBV 5) (OBJ in)))
                                      (BITXOR ((BVSUBV 6) (OBJ in))
                                              ((BVSUBV 7) (OBJ in)))))
))
((understable2 s1 nil))

) ; end assert
```

PRTY Fault List

"Parity generator"	
(1 ASSNCNTL s1)	Successful
(2 MICROOP (s1 - "") BITXOR BITEQV)	Successful, but 2-phase
(3 MICROOP (s1 - L) BITXOR BITEQV)	Successful
(4 MICROOP (s1 - LL) BITXOR BITEQV)	Successful
(5 MICROOP (s1 - LR) BITXOR BITEQV)	Successful
(6 MICROOP (s1 - R) BITXOR BITEQV)	Successful
(7 MICROOP (s1 - RL) BITXOR BITEQV)	Successful
(8 MICROOP (s1 - RR) BITXOR BITEQV)	Successful
(9 STUCKDATA (s1 - "") (BIT "0"))	Successful
(10 STUCKDATA (s1 - "") (BIT "1"))	Successful
(11 STUCKDATA (s1 - L) (BIT "0"))	Successful
(12 STUCKDATA (s1 - L) (BIT "1"))	Successful
(13 STUCKDATA (s1 - LL) (BIT "0"))	Successful
(14 STUCKDATA (s1 - LL) (BIT "1"))	Successful
(15 STUCKDATA (s1 - LLL) (BIT "0"))	Successful
(16 STUCKDATA (s1 - LLL) (BIT "1"))	Successful
(17 STUCKDATA (s1 - LLLL) (BV "00000000"))	Successful
(18 STUCKDATA (s1 - LLLL) (BV "11111111"))	Successful
(19 STUCKDATA (s1 - LLR) (BIT "0"))	Successful
(20 STUCKDATA (s1 - LLR) (BIT "1"))	Successful
(21 STUCKDATA (s1 - LLRL) (BV "00000000"))	Successful
(22 STUCKDATA (s1 - LLRL) (BV "11111111"))	Successful
(23 STUCKDATA (s1 - LR) (BIT "0"))	Successful
(24 STUCKDATA (s1 - LR) (BIT "1"))	Successful
(25 STUCKDATA (s1 - LRL) (BIT "0"))	Successful
(26 STUCKDATA (s1 - LRL) (BIT "1"))	Successful
(27 STUCKDATA (s1 - LRLL) (BV "00000000"))	Successful
(28 STUCKDATA (s1 - LRLL) (BV "11111111"))	Successful
(29 STUCKDATA (s1 - LRR) (BIT "0"))	Successful
(30 STUCKDATA (s1 - LRR) (BIT "1"))	Successful
(31 STUCKDATA (s1 - LRRL) (BV "00000000"))	Successful
(32 STUCKDATA (s1 - LRRL) (BV "11111111"))	Successful
(33 STUCKDATA (s1 - R) (BIT "0"))	Successful
(34 STUCKDATA (s1 - R) (BIT "1"))	Successful
(35 STUCKDATA (s1 - RL) (BIT "0"))	Successful
(36 STUCKDATA (s1 - RL) (BIT "1"))	Successful
(37 STUCKDATA (s1 - RLL) (BIT "0"))	Successful
(38 STUCKDATA (s1 - RLL) (BIT "1"))	Successful
(39 STUCKDATA (s1 - RLLL) (BV "00000000"))	Successful
(40 STUCKDATA (s1 - RLLL) (BV "11111111"))	Successful
(41 STUCKDATA (s1 - RLR) (BIT "0"))	Successful
(42 STUCKDATA (s1 - RLR) (BIT "1"))	Successful
(43 STUCKDATA (s1 - RLRL) (BV "00000000"))	Successful
(44 STUCKDATA (s1 - RLRL) (BV "11111111"))	Successful
(45 STUCKDATA (s1 - RR) (BIT "0"))	Successful
(46 STUCKDATA (s1 - RR) (BIT "1"))	Successful
(47 STUCKDATA (s1 - RRL) (BIT "0"))	Successful
(48 STUCKDATA (s1 - RRL) (BIT "1"))	Successful
(49 STUCKDATA (s1 - RRLL) (BV "00000000"))	Successful
(50 STUCKDATA (s1 - RRLL) (BV "11111111"))	Successful
(51 STUCKDATA (s1 - RRR) (BIT "0"))	Successful
(52 STUCKDATA (s1 - RRR) (BIT "1"))	Successful

(53 STUCKDATA (s1 - RRRL) (BV "00000000")) Successful
(54 STUCKDATA (s1 - RRRL) (BV "11111111")) Successful

SHFT Description

```
; File SHIFT.HDL - try a shift register

;    ENTITY Shift(
;          clock,
;          clear,
;          leftin,
;          rightin,
;          control,
;          d0,
;          d1,
;          d2,
;          d3
;           : IN BIT;
;          rightout,
;          leftout
;           : OUT BIT) IS
;    END Shift;
;
;    ARCHITECTURE Arch OF Shft IS
;
;    PROCESS(clear,clock,q0,q3)
;      SIGNAL
;         q0,
;         q1,
;         q2,
;         q3 : BIT
;    BEGIN
; s1:  IF clear = '0' THEN
; s2:     q0 < = '0';
; s3:     q1 < = '0';
; s4:     q2 < = '0';
; s5:     q3 < = '0';
;      ELSE
; s6:    IF NOT clock'STABLE AND clock = '1' THEN
; s7:       CASE control IS
;             WHEN "00" = >  -- hold
;               null;
;             WHEN "01" = >  -- shift left
; s8:           q3 < = q2;
; s9:           q2 < = q1;
; s10:          q1 < = q0;
; s11:          q0 < = leftin;
;             WHEN "10" = >  -- shift right
; s12:          q3 < = rightin;
; s13:          q2 < = q3;
; s14:          q1 < = q2;
; s15:          q0 < = q1;
;             WHEN "11" = >  -- parallel load
; s16:          q3 < = d3;
; s17:          q2 < = d2;
; s18:          q1 < = d1;
; s19:          q0 < = d0;
;          END CASE;
```

```
;        END IF;
;      END IF;
; s20: leftout < = q3;          -- explicit output
; s21: rightout < = q0;         -- explicit output
;     END PROCESS;
;     END Arch;
(assert

((fileprefix "shft"))
((modelname "Bidirectional Parallel/Serial In Serial Out Shift Register"))

((datatype clock BIT))   ((statevar clock t)) ((inputpin clock))

((datatype clear BIT))   ((statevar clear t)) ((inputpin clear))

((datatype leftin BIT))   ((statevar leftin t))
((inputpin leftin))

((datatype rightin BIT))   ((statevar rightin t)) ((inputpin rightin))

((datatype control BV))   ((bvlength control 2)) ((statevar control t))
((inputpin control))

((datatype d0 BIT))   ((statevar d0 t)) ((inputpin d0))

((datatype d1 BIT))   ((statevar d1 t)) ((inputpin d1))

((datatype d2 BIT))   ((statevar d2 t)) ((inputpin d2))

((datatype d3 BIT))   ((statevar d3 t)) ((inputpin d3))

((datatype q0 BIT))   ((statevar q0 t))

((datatype q1 BIT))   ((statevar q1 t))

((datatype q2 BIT))   ((statevar q2 t))

((datatype q3 BIT))   ((statevar q3 t))

((datatype leftout BIT))   ((statevar leftout nil)) ((outputpin leftout))

((datatype rightout BIT))   ((statevar rightout nil)) ((outputpin rightout))

((statement s1))
((statementtype s1 IF))
((controlexpression s1 (BITEQV (OBJ clear) (LIT (BIT "0"))) ))
((subordinaterange s1 THEN (s2 s3 s4 s5) ))
((subordinaterange s1 ELSE (s6) ))
  ((understable2 s1 nil))

((statement s2))
((statementtype s2 ASSIGNMENT))
  ((destinationobject s2 q0))
  ((sourceexpression s2 (LIT (BIT "0")) ))
    ((understable2 s2 nil))
```

```
((statement s3))
((statementtype s3 ASSIGNMENT))
  ((destinationobject s3 q1))
  ((sourceexpression s3 (LIT (BIT "0")) ))
    ((understable2 s3 nil))

((statement s4))
((statementtype s4 ASSIGNMENT))
  ((destinationobject s4 q2))
  ((sourceexpression s4 (LIT (BIT "0")) ))
    ((understable2 s4 nil))

((statement s5))
((statementtype s5 ASSIGNMENT))
  ((destinationobject s5 q3))
  ((sourceexpression s5 (LIT (BIT "0")) ))
    ((understable2 s5 nil))

((statement s6))
((statementtype s6 IF))
((controlexpression s6 (BITAND (BITNOT (STABLE (OBJ clock)))
                    (BITEQV (OBJ clock) (LIT (BIT "1")))) ))
((subordinaterange s6 THEN (s7) ))
((subordinaterange s6 ELSE nil ))
  ((understable2 s6 clock))

((statement s7))
  ((understable s7))
((statementtype s7 CASE))
((controlexpression s7 (OBJ control) ))
((subordinaterange s7 ((BV "00")) nil ))
((subordinaterange s7 ((BV "01")) (s8 s9 s10 s11) ))
((subordinaterange s7 ((BV "10")) (s12 s13 s14 s15) ))
((subordinaterange s7 ((BV "11")) (s16 s17 s18 s19) ))
  ((understable2 s7 clock))

((statement  s8))
  ((understable s8))
((statementtype s8 ASSIGNMENT))
  ((destinationobject s8 q3))
  ((sourceexpression s8 (OBJ q2) ))
  ((understable2 s8 clock))

((statement  s9))
  ((understable s9))
((statementtype s9 ASSIGNMENT))
  ((destinationobject s9 q2))
  ((sourceexpression s9 (OBJ q1) ))
  ((understable2 s9 clock))

((statement s10))
  ((understable s10))
((statementtype s10 ASSIGNMENT))
  ((destinationobject s10 q1))
  ((sourceexpression s10 (OBJ q0) ))
  ((understable2 s10 clock))
```

```
((statement s11))
  ((understable s11))
((statementtype s11 ASSIGNMENT))
  ((destinationobject s11 q0))
  ((sourceexpression s11 (OBJ leftin) ))
  ((understable2 s11 clock))

((statement s12))
  ((understable s12))
((statementtype s12 ASSIGNMENT))
  ((destinationobject s12 q3))
  ((sourceexpression s12 (OBJ rightin) ))
  ((understable2 s12 clock))

((statement s13))
  ((understable s13))
((statementtype s13 ASSIGNMENT))
  ((destinationobject s13 q2))
  ((sourceexpression s13 (OBJ q3) ))
  ((understable2 s13 clock))

((statement s14))
  ((understable s14))
((statementtype s14 ASSIGNMENT))
  ((destinationobject s14 q1))
  ((sourceexpression s14 (OBJ q2) ))
  ((understable2 s14 clock))

((statement s15))
  ((understable s15))
((statementtype s15 ASSIGNMENT))
  ((destinationobject s15 q0))
  ((sourceexpression s15 (OBJ q1) ))
  ((understable2 s15 clock))

((statement s16))
  ((understable s16))
((statementtype s16 ASSIGNMENT))
  ((destinationobject s16 q3))
  ((sourceexpression s16 (OBJ d3) ))
  ((understable2 s16 clock))

((statement s17))
  ((understable s17))
((statementtype s17 ASSIGNMENT))
  ((destinationobject s17 q2))
  ((sourceexpression s17 (OBJ d2) ))
  ((understable2 s17 clock))

((statement s18))
  ((understable s18))
((statementtype s18 ASSIGNMENT))
  ((destinationobject s18 q1))
  ((sourceexpression s18 (OBJ d1) ))
  ((understable2 s18 clock))
```

```
((statement s19))
   ((understable s19))
((statementtype s19 ASSIGNMENT))
   ((destinationobject s19 q0))
   ((sourceexpression s19 (OBJ d0) ))
   ((understable2 s19 clock))

((statement s20))
((statementtype s20 ASSIGNMENT))
   ((destinationobject s20 leftout))
   ((sourceexpression s20 (OBJ q3) ))
   ((understable2 s20 nil))

((statement s21))
((statementtype s21 ASSIGNMENT))
   ((destinationobject s21 rightout))
   ((sourceexpression s21 (OBJ q0) ))
      ((understable2 s21 nil))

) ;end assert
```

SHFT Fault List

"Bidirectional Parallel/Serial In Serial Out Shift Register"

(1 STUCKTHEN s1)	Failed; can't DNE under STUCK
(2 STUCKELSE s1)	Successful, but 2-phase
(3 MICROOP (s1 - "") BITEQV BITXOR)	Failed; faulty micro-op
(4 ASSNCNTL s2)	Successful
(5 ASSNCNTL s3)	Successful
(6 ASSNCNTL s4)	Successful
(7 ASSNCNTL s5)	Successful
(8 STUCKTHEN s6)	Excluded - continuous clocking
(9 STUCKELSE s6)	Successful
(10 MICROOP (s6 - "") BITAND BITOR)	Excluded - clocks on high or edeg
(11 MICROOP (s6 - L) BITNOT BITBUF)	Excluded - clocks on high unless edge
(12 MICROOP (s6 - R) BITEQV BITXOR)	Excluded - inverted clock
(13 DEADCLAUSE s7 ((BV "11")))	Successful
(14 DEADCLAUSE s7 ((BV "10")))	Successful
(15 DEADCLAUSE s7 ((BV "01")))	Successful
(16 DEADCLAUSE s7 ((BV "00")))	Not a fault
(17 ASSNCNTL s8)	Successful
(18 ASSNCNTL s9)	Successful
(19 ASSNCNTL s10)	Successful
(20 ASSNCNTL s11)	Successful
(21 ASSNCNTL s12)	Successful
(22 ASSNCNTL s13)	Successful
(23 ASSNCNTL s14)	Successful
(24 ASSNCNTL s15)	Successful
(25 ASSNCNTL s16)	Successful, but 2-phase before trying others
(26 ASSNCNTL s17)	Successful, but 2-phase before trying others
(27 ASSNCNTL s18)	Successful, but 2-phase before trying others
(28 ASSNCNTL s19)	Successful, but 2-phase before trying others
(29 ASSNCNTL s20)	Overflowed interpreter
(30 ASSNCNTL s21)	Overflowed interpreter
(31 STUCKDATA (s1 - "") (BIT "0"))	Failed; faulty micro-op
(32 STUCKDATA (s1 - "") (BIT "1"))	Failed; faulty micro-op
(33 STUCKDATA (s1 - L) (BIT "0"))	Failed; faulty micro-op
(34 STUCKDATA (s1 - L) (BIT "1"))	Failed; faulty micro-op
(35 STUCKDATA (s1 - R) (BIT "0"))	Failed; faulty micro-op
(36 STUCKDATA (s1 - R) (BIT "1"))	Failed; faulty micro-op
(37 STUCKDATA (s2 - "") (BIT "0"))	Not a fault
(38 STUCKDATA (s2 - "") (BIT "1"))	Successful
(39 STUCKDATA (s3 - "") (BIT "0"))	Not a fault
(40 STUCKDATA (s3 - "") (BIT "1"))	Successful
(41 STUCKDATA (s4 - "") (BIT "0"))	Not a fault
(42 STUCKDATA (s4 - "") (BIT "1"))	Successful
(43 STUCKDATA (s5 - "") (BIT "0"))	Not a fault
(44 STUCKDATA (s5 - "") (BIT "1"))	Successful
(45 STUCKDATA (s6 - "") (BIT "0"))	Successful
(46 STUCKDATA (s6 - "") (BIT "1"))	Excluded - continuous clocking
(47 STUCKDATA (s6 - L) (BIT "0"))	Successful
(48 STUCKDATA (s6 - L) (BIT "1"))	Excluded -
(49 STUCKDATA (s6 - LL) (BIT "0"))	Excluded -
(50 STUCKDATA (s6 - LL) (BIT "1"))	Successful
(51 STUCKDATA (s6 - R) (BIT "0"))	Successful
(52 STUCKDATA (s6 - R) (BIT "1"))	Exclued - clocks on any edge

(53 STUCKDATA (s6 - RL) (BIT "0"))	Successful
(54 STUCKDATA (s6 - RL) (BIT "1"))	Excluded - clocks on any edge
(55 STUCKDATA (s6 - RR) (BIT "0"))	Excluded - inverted clock
(56 STUCKDATA (s6 - RR) (BIT "1"))	Not a fault
(57 STUCKDATA (s7 - "") (BV "00"))	Failed; faulty micro-op
(58 STUCKDATA (s7 - "") (BV "11"))	Failed; faulty micro-op
(59 STUCKDATA (s8 - "") (BIT "0"))	Successful
(60 STUCKDATA (s8 - "") (BIT "1"))	Successful
(61 STUCKDATA (s9 - "") (BIT "0"))	Successful
(62 STUCKDATA (s9 - "") (BIT "1"))	Successful
(63 STUCKDATA (s10 - "") (BIT "0"))	Successful
(64 STUCKDATA (s10 - "") (BIT "1"))	Successful
(65 STUCKDATA (s11 - "") (BIT "0"))	Successful
(66 STUCKDATA (s11 - "") (BIT "1"))	Successful
(67 STUCKDATA (s12 - "") (BIT "0"))	Successful
(68 STUCKDATA (s12 - "") (BIT "1"))	Successful
(69 STUCKDATA (s13 - "") (BIT "0"))	Successful
(70 STUCKDATA (s13 - "") (BIT "1"))	Successful
(71 STUCKDATA (s14 - "") (BIT "0"))	Successful
(72 STUCKDATA (s14 - "") (BIT "1"))	Successful
(73 STUCKDATA (s15 - "") (BIT "0"))	Successful
(74 STUCKDATA (s15 - "") (BIT "1"))	Successful
(75 STUCKDATA (s16 - "") (BIT "0"))	Successful
(76 STUCKDATA (s16 - "") (BIT "1"))	Successful
(77 STUCKDATA (s17 - "") (BIT "0"))	Successful
(78 STUCKDATA (s17 - "") (BIT "1"))	Successful
(79 STUCKDATA (s18 - "") (BIT "0"))	Successful
(80 STUCKDATA (s18 - "") (BIT "1"))	Successful
(81 STUCKDATA (s19 - "") (BIT "0"))	Successful
(82 STUCKDATA (s19 - "") (BIT "1"))	Successful
(83 STUCKDATA (s20 - "") (BIT "0"))	Successful
(84 STUCKDATA (s20 - "") (BIT "1"))	Successful
(85 STUCKDATA (s21 - "") (BIT "0"))	Successful
(86 STUCKDATA (s21 - "") (BIT "1"))	Successful

SHFTV Description

```
; File SHIFTV.HDL - try a shift register

;    ENTITY ShiftV
;     (clock,
;      clear,
;      leftin,
;      rightin,
;      control
;       : IN BIT;
;      d : IN BIT_VECTOR(3 DOWNTO 0);
;      rightout,
;      leftout
;       : OUT BIT) IS
;    END ShiftV;
;
;    ARCHITECTURE Arch OF Shiftv IS
;
;    PROCESS(clear,clock,q)
;      SIGNAL q:BIT_VECTOR(3 DOWNTO 0);
;    BEGIN
; s1:   IF clear='0' THEN
; s2:      q < = "0000";
;       ELSE
; s6:      IF NOT clock'STABLE AND clock='1' THEN
; s7:        CASE control IS
;              WHEN "00" = > -- no op.
;                null;
;              WHEN "01" = > -- shift left
; s8:            q < = q(2 downto 0) & (leftin);
                                ; what is VHDL for BIT-BV
;              WHEN "10" = > -- shift right
; s12:           q < = (rightin) & q(3 downto 1);
                                ; what is VHDL for BIT-BV
;              WHEN "11" = > -- parallel load
; s16:           q < = d;
;            END CASE;
;         END IF;
;       END IF;
; s20:   leftout < = q(3);
; s21:   rightout < = q(0);
;    END PROCESS;
;
;    END Arch;
(assert

((fileprefix "shftv"))
((modelname "Bidirectional Parallel/Serial In Serial Out Shift Register"))

((datatype clock BIT))  ((statevar clock t)) ((inputpin clock))

((datatype clear BIT))  ((statevar clear t))
((inputpin clear))
```

```
((datatype leftin BIT))  ((statevar leftin t)) ((inputpin leftin))

((datatype rightin BIT))  ((statevar rightin t)) ((inputpin rightin))

((datatype control BV))  ((bvlength control 2)) ((statevar control t))
((inputpin control))

((datatype d BV))  ((bvlength d 4))  ((statevar d t))
((inputpin d))

((datatype q BV))  ((bvlength q 4)) ((statevar q t))

((datatype leftout BIT))  ((statevar leftout nil)) ((outputpin leftout))

((datatype rightout BIT))  ((statevar rightout nil)) ((outputpin rightout))


((statement s1))
((statementtype s1 IF))
((controlexpression s1 (BITEQV (OBJ clear) (LIT (BIT "0"))) ))
((subordinaterange s1 THEN (s2) ))
((subordinaterange s1 ELSE (s6) ))
((understable2 s1 nil))

((statement s2))
((statementtype s2 ASSIGNMENT))
  ((destinationobject s2 q))
  ((sourceexpression s2 (LIT (BV "0000")) ))
((understable2 s2 nil))

((statement s6))
((statementtype s6 IF))
((controlexpression s6 (BITAND (BITNOT (STABLE (OBJ clock)))
                  (BITEQV (OBJ clock) (LIT (BIT "1")))) ))
((subordinaterange s6 THEN (s7) ))
((subordinaterange s6 ELSE nil ))
((understable2 s6 clock))

((statement s7))
((understable s7))
((statementtype s7 CASE))
((controlexpression s7 (OBJ control) ))
((subordinaterange s7 ((BV "00")) nil ))
((subordinaterange s7 ((BV "01")) (s8 ) ))
((subordinaterange s7 ((BV "10")) (s12 ) ))
((subordinaterange s7 ((BV "11")) (s16 ) ))
((understable2 s7 clock))

((statement  s8))
((understable s8))
((statementtype s8 ASSIGNMENT))
  ((destinationobject s8 q))
  ((sourceexpression s8 (BVCAT ((BVSUBV 2 0) (OBJ q)) (BITBV (OBJ leftin))) ))
((understable2 s8 clock))
```

```
((statement s12))
((understable s12))
((statementtype s12 ASSIGNMENT))
  ((destinationobject s12 q))
  ((sourceexpression s12 (BVCAT (BITBV (OBJ rightin))
                          ((BVSUBV 3 1) (OBJ q) ))  ))
((understable2 s12 clock))

((statement s16))
((understable s16))
((statementtype s16 ASSIGNMENT))
  ((destinationobject s16 q))
  ((sourceexpression s16 (OBJ d) ))
((understable2 s16 clock))

((statement s20))
((statementtype s20 ASSIGNMENT))
  ((destinationobject s20 leftout))
  ((sourceexpression s20 ((BVSUBV 3) (OBJ q)) ))
((understable2 s20 nil))

((statement s21))
((statementtype s21 ASSIGNMENT))
  ((destinationobject s21 rightout))
  ((sourceexpression s21 ((BVSUBV 0) (OBJ q)) ))
((understable2 s21 nil))

) ;end assert
```

SHFTV Fault List

"Bidirectional Parallel/Serial In Serial Out Shift Register"

(1 STUCKTHEN s1)	Failed; can't DNE under STUCK
(2 STUCKELSE s1)	Successful, but 2-phase
(3 MICROOP (s1 - "") BITEQV BITXOR)	Failed; faulty micro-op
(4 ASSNCNTL s2)	Successful
(5 STUCKTHEN s6)	Excluded - continuous clocking
(6 STUCKELSE s6)	Successful
(7 MICROOP (s6 - "") BITAND BITOR)	Excluded - clocks on high or edge
(8 MICROOP (s6 - L) BITNOT BITBUF)	Excluded - clocks on high unless edge
(9 MICROOP (s6 - R) BITEQV BITXOR)	Excluded - inverted clock
(10 DEADCLAUSE s7 ((BV "11")))	Successful
(11 DEADCLAUSE s7 ((BV "10")))	Successful
(12 DEADCLAUSE s7 ((BV "01")))	Successful
(13 DEADCLAUSE s7 ((BV "00")))	Not a fault
(14 ASSNCNTL s8)	Successful
(15 ASSNCNTL s12)	Successful
(16 ASSNCNTL s16)	Successful, but 2-phase
(17 ASSNCNTL s20)	Overflowed; bad choices and much backtracking
(18 ASSNCNTL s21)	Overflowed; bad choices and much backtracking
(19 STUCKDATA (s1 - "") (BIT "0"))	Failed; faulty micro-op
(20 STUCKDATA (s1 - "") (BIT "1"))	Failed; faulty micro-op
(21 STUCKDATA (s1 - L) (BIT "0"))	Failed; faulty micro-op
(22 STUCKDATA (s1 - L) (BIT "1"))	Failed; faulty micro-op
(23 STUCKDATA (s1 - R) (BIT "0"))	Not a fault
(24 STUCKDATA (s1 - R) (BIT "1"))	Failed; faulty micro-op
(25 STUCKDATA (s2 - "") (BV "0000"))	Not a fault
(26 STUCKDATA (s2 - "") (BV "1111"))	Successful
(27 STUCKDATA (s6 - "") (BIT "0"))	Successful
(28 STUCKDATA (s6 - "") (BIT "1"))	Excluded - continuous clocking
(29 STUCKDATA (s6 - L) (BIT "0"))	Successful
(30 STUCKDATA (s6 - L) (BIT "1"))	Excluded - clocks on high
(31 STUCKDATA (s6 - LL) (BIT "0"))	Excluded - clocks on high
(32 STUCKDATA (s6 - LL) (BIT "1"))	Successful
(33 STUCKDATA (s6 - R) (BIT "0"))	Successful
(34 STUCKDATA (s6 - R) (BIT "1"))	Excluded - clocks on any edge
(35 STUCKDATA (s6 - RL) (BIT "0"))	Successful
(36 STUCKDATA (s6 - RL) (BIT "1"))	Excluded - clocks on any edge
(37 STUCKDATA (s6 - RR) (BIT "0"))	Excluded - inverted clock
(38 STUCKDATA (s6 - RR) (BIT "1"))	Not a fault
(39 STUCKDATA (s7 - "") (BV "00"))	Failed; faulty micro-op
(40 STUCKDATA (s7 - "") (BV "11"))	Failed; faulty micro-op
(41 STUCKDATA (s8 - "") (BV "0000"))	Successful
(42 STUCKDATA (s8 - "") (BV "1111"))	Successful
(43 STUCKDATA (s8 - L) (BV "000"))	Successful
(44 STUCKDATA (s8 - L) (BV "111"))	Successful
(45 STUCKDATA (s8 - LL) (BV "0000"))	Successful
(46 STUCKDATA (s8 - LL) (BV "1111"))	Successful
(47 STUCKDATA (s8 - R) (BV "0"))	Successful
(48 STUCKDATA (s8 - R) (BV "1"))	Successful
(49 STUCKDATA (s8 - RL) (BIT "0"))	Successful
(50 STUCKDATA (s8 - RL) (BIT "1"))	Successful
(51 STUCKDATA (s12 - "") (BV "0000"))	Successful
(52 STUCKDATA (s12 - "") (BV "1111"))	Successful

(53 STUCKDATA (s12 - L) (BV "0"))	Successful
(54 STUCKDATA (s12 - L) (BV "1"))	Successful
(55 STUCKDATA (s12 - LL) (BIT "0"))	Successful
(56 STUCKDATA (s12 - LL) (BIT "1"))	Successful
(57 STUCKDATA (s12 - R) (BV "000"))	Successful
(58 STUCKDATA (s12 - R) (BV "111"))	Successful
(59 STUCKDATA (s12 - RL) (BV "0000"))	Successful
(60 STUCKDATA (s12 - RL) (BV "1111"))	Successful
(61 STUCKDATA (s16 - "") (BV "0000"))	Successful
(62 STUCKDATA (s16 - "") (BV "1111"))	Successful
(63 STUCKDATA (s20 - "") (BIT "0"))	Successful
(64 STUCKDATA (s20 - "") (BIT "1"))	Successful
(65 STUCKDATA (s20 - L) (BV "0000"))	Successful
(66 STUCKDATA (s20 - L) (BV "1111"))	Successful
(67 STUCKDATA (s21 - "") (BIT "0"))	Successful
(68 STUCKDATA (s21 - "") (BIT "1"))	Successful
(69 STUCKDATA (s21 - L) (BV "0000"))	Successful
(70 STUCKDATA (s21 - L) (BV "1111"))	Successful

UARTO Description

```
; File UARTO.HDL - Transmit Half of a Simple UART

;     ENTITY Uart(
;          reset,
;          wrstrb,                    -- write strobe
;          clock
;           : IN BIT;
;          databus: IN BIT_VECTOR(1 DOWNTO 0);
;          dataout : OUT BIT
;          txbusy : OUT BIT           -- transmit busy/enable flag
;          ) IS
;     END Uart;
;
;    ARCHITECTURE Arch OF Uart IS
;
;    PROCESS(...)
;    SIGNAL
;       txcnt : BIT_VECTOR(2 DOWNTO 0);      -- transmit bit couner
;       txreg : BIT_VECTOR(3 DOWNTO 0);
;    BEGIN
; s01:    IF reset = '0' THEN
;                          -- reset transmit side to wait for byte
; s02:       txbusy < = '0';         -- clear transmit busy flag
;         ELSE
; s03:       IF wrstrb = '1' AND NOT wrstrb'STABLE THEN   -- write strobe
; s04:          txcnt < = "11";    -- set count for:  start + data + stop
; s05:          txreg < = "1" & databus & "0";-- (R to L) start, data, stop
; s06:          txbusy < = '1';             -- transmitting
;            END IF;
;
; s07:       IF (clock = '1' AND NOT clock'STABLE) AND txbusy = '1' THEN
; s08:          dataout < = txreg(0);              -- output low bit
; s09:          txreg < = "1" & txreg(3 downto 1);       -- shift bits over
; s10:          txcnt < = BVSUB(txcnt,"01");     -- count bits out
; s11:          IF txcnt = "00" THEN          -- note delta delay!
; s12:             txbusy < = '0';              -- done transmitting
;             END IF;
;          END IF;
;         END IF
;    END PROCESS;
;
;    END ARch;
(assert

((fileprefix "UARTO"))
((modelname "Simple UART: Transmit Half"))


((datatype reset BIT)) ((statevar reset t)) ((inputpin reset))

((datatype wrstrb BIT)) ((statevar wrstrb t)) ((inputpin wrstrb))

((datatype clock BIT)) ((statevar clock t)) ((inputpin clock))
```

```
((datatype databus BV)) ((bvlength databus 2)) ((statevar databus t))
((inputpin databus))

((datatype dataout BIT)) ((statevar dataout t)) ((outputpin dataout))

((datatype txbusy BIT)) ((statevar txbusy t)) ((outputpin txbusy))

((datatype txcnt BV)) ((bvlength txcnt 2)) ((statevar txcnt t))

((datatype txreg BV)) ((bvlength txreg 4)) ((statevar txreg t))

((statement s1))
((statementtype s1 IF))
((controlexpression s1 (BITEQV (OBJ reset) (LIT (BIT "0"))) ))
((subordinaterange s1 THEN (s2) ))
((subordinaterange s1 ELSE (s3 s7) ))
((understable2 s1 nil))

((statement s2))
((statementtype s2 ASSIGNMENT))
((destinationobject s2 txbusy))
((sourceexpression s2 (LIT (BIT "0")) ))
((understable2 s2 nil))

((statement s3))
((statementtype s3 IF))
((controlexpression s3 (BITAND (BITEQV (OBJ wrstrb)
                                       (LIT (BIT "1")))
                               (BITNOT (STABLE (OBJ wrstrb)))) ))
((subordinaterange s3 THEN (s4 s5 s6) ))
((subordinaterange s3 ELSE () ))
((understable2 s3 wrstrb))

((statement s4))
((statementtype s4 ASSIGNMENT))
((destinationobject s4 txcnt))
((sourceexpression s4 (LIT (BV "11")) ))
((understable s4))
((understable2 s4 wrstrb))

((statement s5))
((statementtype s5 ASSIGNMENT))
((destinationobject s5 txreg))
((sourceexpression s5 (BVCAT  (LIT (BV "1"))
                              (BVCAT (OBJ databus)
                                     (LIT (BV "0"))))) ))
((understable s5))
((understable2 s5 wrstrb))

((statement s6))
((statementtype s6 ASSIGNMENT))
((destinationobject s6 txbusy))
((sourceexpression s6 (LIT (BIT "1"))))
((understable s6))
((understable2 s6 wrstrb))
```

```
((statement s7))
((statementtype s7 IF))
((controlexpression s7 (BITAND
                        (BITAND (BITEQV (OBJ clock) (LIT (BIT "1")))
                                (BITNOT (STABLE (OBJ clock))))
                        (BITEQV (OBJ txbusy) (LIT (BIT "1")))) ))
((subordinaterange s7 THEN (s8 s9 s10 s11) ))
((subordinaterange s7 ELSE () ))
((understable2 s7 clock))

((statement s8))
((statementtype s8 ASSIGNMENT))
((destinationobject s8 dataout))
((sourceexpression s8 ((BVSUBV 0) (OBJ txreg))))
((understable s8))
((understable2 s8 clock))

((statement s9))
((statementtype s9 ASSIGNMENT))
((destinationobject s9 txreg))
((sourceexpression s9 (BVCAT (LIT (BV "1"))
                             ((BVSUBV 3 1) (OBJ txreg)) ) ))
((understable s9))
((understable2 s9 clock))

((statement s10))
((statementtype s10 ASSIGNMENT))
((destinationobject s10 txcnt))
((sourceexpression s10 (BVSUB (OBJ txcnt) (LIT (BV "01")))))
((understable s10))
((understable2 s10 clock))

((statement s11))
((statementtype s11 IF))
((understable s11))
((controlexpression s11 (BVEQ (OBJ txcnt) (LIT (BV "00"))) ))
((subordinaterange s11 THEN (s12) ))
((subordinaterange s11 ELSE () ))
((understable2 s11 clock))

((statement s12))
((statementtype s12 ASSIGNMENT))
((destinationobject s12 txbusy))
((sourceexpression s12 (LIT (BIT "0"))))
((understable s12))
((understable2 s12 clock))


) ; end assert
```

UARTO Fault List

"Simple UART: Transmit Half"	
(1 STUCKTHEN s1)	Failed; can't DNE under STUCK
(2 STUCKELSE s1)	Successful, but 2-phase
(3 MICROOP (s1 - "") BITEQV BITXOR)	Failed; faulty micro-op
(4 ASSNCNTL s2)	Successful
(5 STUCKTHEN s3)	Excluded - continuous clocking
(6 STUCKELSE s3)	Successful
(7 MICROOP (s3 - "") BITAND BITOR)	Excluded - clocks on high or edge
(8 MICROOP (s3 - L) BITEQV BITXOR)	Excluded - clocks on high unless edge
(9 MICROOP (s3 - R) BITNOT BITBUF)	Excluded - inverted clock
(10 ASSNCNTL s4)	Failed; can't preload
(11 ASSNCNTL s5)	Failed; can't preload
(12 ASSNCNTL s6)	Overflow; can't preload
(13 STUCKTHEN s7)	Excluded - continous clocking
(14 STUCKELSE s7)	Failed; load under opposite from STUCK
(15 MICROOP (s7 - "") BITAND BITOR)	Excluded - clocks on high or edge
(16 MICROOP (s7 - L) BITAND BITOR)	Excluded - clocks on high or edge
(17 MICROOP (s7 - LL) BITEQV BITXOR)	Excluded - inverted clock
(18 MICROOP (s7 - LR) BITNOT BITBUF)	Excluded - clocks on high unless edge
(19 MICROOP (s7 - R) BITEQV BITXOR)	Failed; faulty micro-op
(20 ASSNCNTL s8)	Overflow; multiple loads
(21 ASSNCNTL s9)	Overflow (crashed)
(22 ASSNCNTL s10)	Overflow; bad inequality choices
(23 MICROOP (s10 - "") BVSUB BVADD)	Overflow; multiple loads; bad choices
(24 STUCKTHEN s11)	Overflow; multiple loads; needs better C/O
(25 STUCKELSE s11)	Overflow; multiple loads
(26 MICROOP (s11 - "") BVEQ BVNEQ)	Successful
(27 ASSNCNTL s12)	Overflow; multiple loads

The vita has been removed from the scanned document