Test Generation for Behavioral Models with Reconvergent Fanout and Feed-back

by

Fong-Shek Lam

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

APPROVED:

Dr. J. R. Armstrong, Chairman

Dr. S. F. Midkiff

[illegible] Dr. J. G. Tront

September, 1989

Blacksburg, Virginia

Test Generation for Behavioral Models with Reconvergent Fanout and Feed-back

by

Fong-Shek Lam

Dr. J. R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

In this thesis, new methods to handle reconvergent fanout and feed-back during behavioral level test generation are proposed. These methods have been implemented into a previously developed automatic test generator. The improved test generator was tested on five behavioral circuit models. For circuits with the reconvergent fanout situation, the improved test generator can generate tests completely automatically. For circuits with feed-back, user assistance in a circuit initialization step is required. Some suggestions for future development for the test generator are discussed. Examples on how to use the improved test generator are presented.

Acknowledgements

I would like to thank Dr. Armstrong for providing the challenging environment in which to pursue this project, and for providing guidance during this research. I would also like to thank Dr. S. F. Midkiff and Dr. J. G. Tront for serving as members of my committee.

I would like to dedicate this work to my wife and my family, whose love and support have made my life wonderful.

Table of Contents

List of Illustrations

List of Tables

Chapter 1. Introduction

Modern VLSI technology has enabled the production of integrated circuit chips that are composed of hundred-thousands to a million transistors. For instance, Intel's new i860 microprocessor consists of 1 million transistors [1]. Test generation for increasingly complex VLSI circuits is very difficult and time consuming. The traditional gate level test generation approaches become inefficient when applied to complex VLSI circuits. This is because test generation time increases exponentially with the number of gates in a circuit. More efficient test generation methods are urgently needed in order to ensure the reliability of the VLSI circuit products.

Several higher level approaches to test generation for VLSI circuits have been proposed [7-24]. Instead of describing a circuit under test with gate interconnections, it is described with module interconnections, or with its actual input/output response (behavioral description). A module in a circuit under test could be an adder, a subtracter, a multiplexer, or a register. Hence, the number of primitive components in a circuit under test is reduced compared to the gate level description.

A behavioral level test generator, organized with an artificial intelligence's goal tree technique and implemented with PROLOG, was developed at Virginia Tech in the past few years [18-22]. This test generator takes a VHDL behavioral circuit description as input, and generates a test sequence for each defined behavioral fault in the circuit. A detailed discussion of this previously developed test generator is in chapter 3.

This previously developed test generator used a single path propagation technique to propagate a fault in a circuit towards a primary output. Therefore, it could not handle most of the reconvergent fanout situations in a circuit under test. Furthermore, this test generator could not handle circuits with feed-back and/or looping.

The research described in this thesis continues the development of this test generator. A new method to handle reconvergent fanout situations in test generation is proposed. Also, a new scheme for testing circuits with feed-back and/or looping is proposed. These two new methods have been incorporated into the test generator.

1.1 Contents

Chapter 2, "Background and literature review", discusses some test generation issues and reviews several gate level and functional level test generation techniques.

Chapter 3, "Previous development", discusses the development of the behavioral level test generator in the past few years.

Chapter 4, ″New development″, discusses the problems in the previous test generator and proposes new solutions for the problems. The new development is focused on the reconvergent fanout, feed-back, and looping problems that cannot be handled by the previous test generator.

Chapter 5, ″The improved test generation algorithm″, describes the procedures and rules for the improved test generator. Also, two test generation examples are presented.

Chapter 6, ″Results″, discusses the test generation results of the improved test generator.

Chapter 7, ″Future development″, discusses the possible future improvements of the test generator.

Chapter 8, ″Conclusion″.

Appendix A, ″How to use the test generator″, shows several examples of using the test generator.

Appendix B, ″Circuit models and fault lists″, shows a VHDL behavioral description, a PROLOG representation of the VHDL description, and a fault list for each circuit model that was tested by the improved test generator.

Chapter 2. Background and literature review

2.1 Digital circuit fault and fault model relationships

Digital circuit faults in integrated circuits are usually caused by random defects during fabrication processes. These random defects cause some transistors inside the fabricated circuit to malfunction, and therefore the digital circuit may not function as it was designed. Since the integrated circuits are very complex, it is impossible to test each transistor inside the circuit. Therefore, testing of the VLSI circuit must be at the level which is higher than the transistor level, such as gate-level and chip-level. Fault models for integrated circuit testing are created according to the level of abstraction that is used in testing.

In gate-level test generation, the most commonly used fault models are the stuck-at fault and stuck-open fault. The stuck-at fault model assumes a logic line in a gate-level circuit is stuck at a fixed logic value. The stuck-open fault model assumes a logic line in a gate-level circuit is stuck open. Since some logic lines in gate-level circuits may not exist in transistor-level circuits, some faults that appear in a gate-level circuit may not actually appear in the corresponding transistor-level circuit [2].

The advancements in VLSI technology have enabled increasingly complex digital circuits to be integrated into a single IC chip. Due to the high complexity of the IC circuits, test generation for complex IC at gate-level is very difficult. Therefore, test generation must be moved to a higher level of abstraction. In chip-level test generation, fault models are created according to the functional behavior of circuits. They are independent of the circuit implementation details in gate-level and in transistor-level.

2.2 Static and dynamic controllability measures

Controllability is a measure that determines how easy a point in a digital circuit network can be assigned a particular value. Static controllability is determined before any logic value is assigned to the primary inputs of the circuit. Once a logic value is assigned to a primary input of the circuit, static controllability ceases to reflect the true controllability of each logic line in the circuit, especially for circuits with reconvergent fanout. In order to accurately measure the controllability of each logic line in the circuit during a test generation process, the controllability must be recomputed after a value is assigned to a logic line. This kind of controllability measure is called a dynamic controllability measure.

An automatic test generation algorithm that uses a dynamic controllability measure can generate test vectors more efficiently than an algorithm that does not use it. With a dynamic controllability measure, the number of back-tracking steps that is needed to generate certain test vectors in reconvergent fanout circuits can be significantly reduced. Also, more test vectors can be generated, and so the fault coverage for a circuit can be increased [3-4].

2.3 Gate-level test generation techniques

Gate-level test generation has been proven to be efficient for MSI and SSI digital integrated circuits. A simple stuck-at fault model is used to assume logic lines in gate-level circuits are stuck at logic 1 or logic 0. The logic gates, such as AND, OR, NOT, XOR, etc., are the primitive components in gate level circuits. The functions of these logic gates are simple and well defined. Therefore gate-level test generation in small digital circuits is relatively straight forward compared to functional level test generation.

The D-algorithm was the first algorithmic method for generating tests for non-redundant gate-level combinational circuits. It was proposed by Roth in 1966 [5]. The fault model of the D-algorithm is stuck-at fault model. It uses the symbol D to represent a good value of logic 1 and a stuck-at value of logic 0, and the symbol $\overline{D}$ to represent good value of 0 and stuck-at value of 1. Fault sensitization is done by justifying the symbol D or $\overline{D}$ at the fault site depending on the assumed stuck-at value. The D or $\overline{D}$ at the fault site is then propagated to an observable output by assigning appropriate values to primary inputs. In the case of reconvergent fanout circuits, the D or $\overline{D}$ is propagated through all the fanout paths towards the reconvergent point.

PODEM is a path oriented decision making gate-level test generation algorithm. It was developed by Goel [5]. Its principle is very similar to the D-algorithm. Its fault model is also the stuck-at fault. The symbol D or $\overline{D}$ is used for fault sensitization. The initial objective [5] of this algorithm is to set up D or $\overline{D}$ at the output of the gate under test by assigning a value at a primary input of a chosen path in the circuit; and then determine whether the assigned value is sufficient to sensitize the fault. If the fault is not yet sensitized, additional values are assigned to other primary inputs in the same manner. After the fault is sensitized with D or $\overline{D}$, the D or $\overline{D}$ is propagated to the next gate towards a primary output by assigning other values to primary inputs [5]. The logic values which have been assigned at primary inputs may or may not be the test for the fault. Therefore, the next step in determining whether the logic values at the primary inputs are a test for the fault, is to determine the implied logic value of internal objects as the result of the value assignment at primary inputs. Knowing the implied logic values at internal signals can avoid conflict in logic value assignment of a signal when continuing propagation of D or $\overline{D}$ towards a primary output. As the result of this, the number of back tracking steps that is needed for certain tests can be greatly reduced compared to the D-algorithm. However, to determine whether the logic values at primary inputs are a test for the fault may require simulation of the circuit under test; and simulation may consume substantial computer time. However, the PODEM algorithm has been found to be more efficient than the D-algorithm, especially for digital circuits with reconvergent fanout.

FAN is a fanout oriented gate-level test generation algorithm developed by Fujiwava and Shimono. The fault model of the algorithm is the stuck-at fault and D and $\overline{D}$ are used for fault sensitization. The FAN algorithm overcomes some deficiencies of the PODEM algorithm by using heuristic rules to determine uniquely implied signal values as early as possible, so that many conflicting signal assignments during test generation process can be avoided. Therefore, the number of back tracking steps can be reduced and the test generation time shortened [6]. Fujiwara and Shimono have shown that the FAN algorithm is more efficient and has higher fault coverage than the PODEM algorithm.

2.4 Functional-level test generation techniques

The complexity of VLSI digital circuits has made test generation at the gate-level increasingly difficult. In order to reduce the difficulty of the test generation process, the number of primitive components in the circuit under test must be reduced. Therefore, for functional level test generation, the primitive components are not simple logic gates, but blocks of logic; the logic lines in the circuit under test are no longer only a single line representing a bit, instead they may be a vector of lines representing a bit vector. The objectives of functional testing are to verify the functional operation according to the design specification of the circuit [11] and to develop tests which will find faults in real chips. In this section, a few functional-level test generation techniques are reviewed.

2.4.1 Register-level test generation

In register level test generation, the combinational components, such as multiplexers, decoders, adders, etc., could be modeled at the gate level [7] or treated as complex primitive components with some well defined functional tables to describe their functional operations [8]. Sequential circuits are also modeled at the register level using models of registers, latches, and counters. Fault models which may be used at this level of testing are stuck-at fault, control fault, micro-operation fault, and register-transfer fault.

The RTG algorithm [7] that was proposed by Shteingart, Nagle, and Greson was designed to detect all classical single "stuck-at" faults in digital circuits. It models combinational components at the gate-level and sequential components at the register level. The D-algorithm is used in the combinational part of the circuit and in overall control of the test generation. The sequential behavior of registers is modeled by some primitive routines. There are two types of clock configurations that the RTG cannot handle. First, the outputs of one sequential component cannot be directly or indirectly fed to the clock input of another sequential component. Second, the data input of a sequential component cannot be the clock signal of the same or another sequential component.

Su and Hsieh have proposed an algorithm to generate functional test vectors for digital systems which are modeled with a register transfer language [8]. Data faults, control faults, and operational faults are considered in this test generation algorithm. Tests for functional faults are generated by applying the D-algorithm on data graphs which are constructed from the register transfer language description. Each statement of the RTL description is represented by a node, and the relationships between statements are represented by an arc in the data graph. The tests for control faults are generated by symbolic simulation techniques. The symbolic simulation technique requires the simulation of the fault-injected circuit model and a fault-free circuit model. This algorithm was further developed by Tonysheng Lin and Stephen Y.H. Su. They used heuristic rules to improve test generation efficiency and fault coverage [9-10].

2.4.2 Functional test generation with binary decision diagrams

Binary Decision Diagrams (BDD) are constructed from the functional implementation of the circuit under test. There is one BDD for each output variable. Each node in the BDD is represented by an input variable; and every node has two branches. The left branch of a node denotes where the decision goes for the node value 0, and the right branch of a node denotes where the decision goes for the node value 1 [12-14]. There are two classes of faults considered in this test generation algorithm. One class is stuck-at faults which affect the module's input lines, output lines, or internal memory elements. Another class is functional faults which affect the execution of one of its experiments [14]. An experiment is to find a complete path on a BDD with an exit value of 0 or 1 [12]. This complete path is a test vector. A test vector set for a circuit is formed by combining all the test vectors found in all the BDD in the circuit under test. This test generation algorithm was found to be feasible for functional level test generation in a modular design environment [12]. However, for complex functional modules, the BDD would become complicated and therefore the test generation efficiency may be reduced.

2.4.3 Functional test generation with AI techniques

An artificial intelligence technique resembles a human's problem solving method that uses heuristic approaches in test vector generation. The heuristic rules are stored in a knowledge base. Test generation methods are organized in a form of goal tree. On the top of a goal tree is a high-level goal, such as generate a test for a fault. The high-level goal is then broken into sub-goals. The high-level goal is solved when all sub-goals are solved. Each sub-goal can be broken into the next level of sub-goals, until the sub-goals are primitive-goals. The primitive goals are to assign some appropriate values to the primary inputs of the circuit, and observe primary outputs such that the fault can be detected. The heuristic rules in the knowledge base are used to assist the test generation process in selecting the best sub-goals, so that the high-level goal on the top of the goal tree can be solved in the most efficient way. This kind of AI technique can be applied to gate-level test generation [15] as well as to functional test generation. In this section, a few of the AI techniques influencing functional test generation are reviewed.

Hitest was developed to generate tests for sequential circuits. Its knowledge base specifies the partition of the circuit into combinational and sequential parts, and the strategies for manipulation of the sequential part in test generation. The knowledge base is mainly entered manually. PODEM is used to generate tests for the combinational part of the circuit. The test generation process of Hitest is organized in the form of a goal tree. The heuristic rules in the knowledge base aid in selecting and solving the sub-goals in the goal tree. Some degree of human interaction is needed to generate tests which no algorithmic approach might be able to handle within a reasonable amount of time [16-17].

A test generation system, developed at MIT (referred as MIT-TG) [16], builds its test generation data base by recording the functional characteristic, propagation and justification requirements of the circuit under test during symbolic simulation. The MIT-TG can generate tests for a fault only when the fault-free behavior is discernible from the symbolic simulation step [16]. In the test generation process, appropriate test values are assigned to the inputs and outputs of the module under test. Justification and propagation of these test values to primary inputs and primary outputs of the circuit relies on the data that is recorded from the symbolic simulation.

A test generation system was developed at Virginia Tech using artificial intelligence techniques [18-22]. This test generation program is implemented with PROLOG. The goal tree structure is used in organizing the test generation process. The data base of the program is built by translating the VHDL behavioral description of the circuit into PROLOG form and extracting the observability and controllability of the signal objects in the circuit, as well as the control signal requirements for the control constructs in the circuit. The concepts of propagation and justification of the D-algorithm are used in the process of test generation. The process of test generation is guided with a set of heuristic rules built into the test generation program, so that a test for a fault can be generated in the most efficient way.

2.4.4 Test generation with hardware description languages

A hardware description language (HDL) is an excellent means of modeling and simulating digital circuits. These models contain all the functional characteristics of the circuits. Therefore HDL descriptions of digital circuits are excellent sources of information for test generation at the functional level. In this section, several of the hardware description language approaches to test generation using HDLs are reviewed.

Levendel and Menon proposed a test generation method for digital circuits described with an HDL [23]. They utilized the D-algorithm approach to generate D-equations and D-propagation cubes for each functional module and control construct in the HDL description. Both procedural (sequential) and non-procedural (concurrent) types of HDL description are considered in this test generation method. For a procedural type of HDL description, the path functions which represent successive HDL statements are used during D-propagation; and the D-algorithm is applied in the test generation process. For a non-procedural type of HDL description, the path function is not required, thus it is much simpler than those of the procedural type of description [23]. The stuck-at fault model is used to test pin faults, state variable faults and control faults. The derivation of D-equations, D-propagation cubes, and path functions for relatively complex functional modules is very complicated and therefore the test generation efficiency could be reduced.

Norrod from Virginia Tech has proposed the E-algorithm in [24, 26]. The E-algorithm, which is an extension of the D-algorithm, can be applied to automatic test generation on the data path and control constructs in HDL descriptions. The fault models in this algorithm include the micro-operation fault and the control fault. The HDL description of the circuit under test is first transformed into a graphical representation. This directed graph has nodes representing micro-operations and arcs representing data flow. The test generation process, that includes fault sensitization, propagation, and justification, operates on the graphical representation. The D-algorithm cannot handle fault propagation through control constructs. Therefore, a special method called E-propagation is used to propagate the fault effect through control constructs [24].

A behavioral automatic test generator for digital circuits modeled with VHDL was developed at Virginia Tech using artificial intelligence techniques [18-22]. This algorithm does not require the transformation of the VHDL description into a graphical representation or fault simulation. The test generation process, which includes fault sensitization, propagation, justification, and execution, operates directly on the VHDL description statements. This algorithm is implemented with the PROLOG computer language. An intelligent data base is maintained by extracting the behavioral and functional characteristic of the circuit from the VHDL behavioral description. The research in this thesis continues the development of this algorithm. The next chapter gives detailed discussion of the previous development of this algorithm.

Chapter 3. Previous development

This automatic test generation algorithm for VHDL behavioral description of digital circuits was initially developed by Barclay and Armstrong in 1986 [18]. The algorithm was implemented with PROLOG and used the artificial intelligence goal tree approach to organize the test generation process. It successfully generated tests for circuits with VHDL behavioral descriptions. However, the test generation process consumed much computer time as the result of using low-level goals during the test generation process [19-20]. In 1987, O'Neill and Armstrong continued the development of this algorithm. High-level goals, such as propagation, justification, and execution, were used to replace the low-level goals. This resulted in much speed improvement in the test generation process. In 1988, Jani and Armstrong further improved the algorithm [21] by developing an improved timing model. This timing model uses a symbol 'R' to represent a '0' to '1' transition. and a symbol 'F' to represent a '1' to '0' transition. The method of solving the high-level goals (propagation, justification, execution) was also improved in order to achieve higher test generation efficiency [21].

3.1 Test generation approach

The test generation program is written with PROLOG. It uses the goal tree structure to organize its test generation process. The main goal of the test generation process is to generate a test for a particular fault. In order to solve this main goal, the process has to solve a number of sub-goals. These sub-goals include fault sensitization and fault propagation. To solve the fault sensitization sub-goal, a number of next level sub-goals, such as justification and execution, have to be solved. Similarly, the fault propagation sub-goal is solved by solving an appropriate set of next level sub-goals, which include justification and execution. The sub-goals continue to break down to the next level of sub-goals, until the sub-goals are primitive goals. Primitive goals are to assign some appropriate values to some primary inputs and observe a primary output. The selection of the sub-goal set is guided by heuristic rules which are built into the test generation program. The solving of each sub-goals is further guided by the information stored in the test generation data base. The controllability and observability of each signal object as well as the control signal requirement for each control construct is extracted from the VHDL description of the circuit. This information is stored in the data base before test generation begins.

The test generation algorithm uses the concepts of propagation and justification from the D-algorithm to implement its propagation and justification rules. However, instead of using D or $\overline{D}$ in fault sensitization as in the D-algorithm, a good/bad value pair is used in fault sensitization. A good-value is the predicted value in a fault free circuit; and a bad-value is the predicted value in a faulty circuit. The selection of good/bad value could be made by user or automatically by the program. The good-value must not be equal to bad-value. There are sets of heuristic rules which are defined for the propagation of the good/bad value pair through control constructs in the circuit. If the control construct is faulty, the effect of the fault is propagated towards a primary output through an assignment statement under the control construct.

The propagation path for the good/bad value pair is selected automatically by the program based on the observability measure; it can also be selected by user. In the case of more than one possible propagation path, the shortest path to a primary output is chosen first. If the first path fails to propagate the good/bad value pair, the second shortest path is chosen, and so on. Regardless of the type of circuit, only single path propagation is considered. This causes the program to generate wrong tests or makes it unable to generate tests for most reconvergent fanout circuits. Also, this program cannot generate tests when the circuit has feed-back and/or looping.

Two-phase testing is one of the special features of this test generation program. The test of a fault requires two phases; the first phase is preloading, the second phase is checking data transfer [21]. The purpose of the preloading phase is to preload a known value into the object. The second phase forces a different value into that object so that the data transfer can be observed. Traversal of the fault site during the preloading phase requires the propagation of the result of preloading to an observable output in order to confirm the success of the preloading operation.

Special fault sensitization and propagation schemes for VHDL control constructs, such as IF-THEN-ELSE and CASE, have been developed. Unlike the signals in data paths, control signals in digital circuits do not have direct path to observable outputs. Therefore, the value of the control signals can only be observed by observing the effect of the control signal. This is done by observing a selected data path controlled by the control signal.

3.2 VHDL sub-set

This automatic test generation algorithm takes the VHDL behavioral description of a digital circuit as input, and generates a test vector sequence for each defined fault in the circuit. The VHDL behavioral description of a digital circuit for this test generation algorithm is restricted to the use of a sub-set of VHDL. This sub-set includes the following [21].

Signal object:	A signal object contains a fixed value at a given time, just like a signal line in digital circuit.
Signal type:	Signal type includes BIT, and BIT-VECTOR. Other type of signal such as BOOLEAN and INTEGER can be represented in the form of BIT and BIT-VECTOR respectively.
Micro-operations:	The micro-operations include: AND, OR, XOR, NOT, equivalence, ADD, and SUBTRACT. The ADD and SUBTRACT operations are implemented in VHDL functions.
Process statements:	Multiple processes in a single VHDL model are allowed.

Assignment statement:	Assignment statements are allowed to be inside a process or outside a process.
Control constructs:	IF-THEN-ELSE and CASE.
Delta time delay:	Each micro-operation executes in delta time delay. This means when new inputs are assigned to a micro-operation, the output of the micro-operation is up-dated after delta time delay.

The FOR loop and WAIT statements of VHDL are not included in this VHDL sub-set, because a sensitized fault in the VHDL description is very difficult to propagate through these statements. An example of a VHDL behavioral description for this test generator is shown in Figure 1 on page 20.

3.3 Fault models

In high-level test generation, fault models must be independent from the implementation detail of the circuit under test. There are a number of fault models which have been developed for this behavioral test generation algorithm. These fault models are directly related to the VHDL sub-set used.

For the process statement, there is a **dead-process** fault. The dead-process fault model is used to check if the process is activated when a signal in its sensitivity list changes. Note that the dead-process fault is removed from the fault list in the new

```
      ENTITY I8212 IS
       (DI : IN BIT_VECTOR(1 TO 8);
        NDS1, DS2, MD, STB, NCLR : IN BIT;
        DO : OUT BIT_VECTOR(1 TO 8));
        NINT : OUT BIT;
      END I8212;

      ARCHITECTURE BEHAVIOR OF I8212 IS

       SIGNAL S0, S1, S2, S3, SRQ : BIT;
       SIGNAL Q : BIT_VECTOR(1 TO 8);

      BEGIN
  1:  PROCESS(NCLR,S1)
      BEGIN
  2:    IF (NCLR ='1') THEN
  3:      Q <= '00000000';
  4:    ELSE IF (S1 = '1') THEN
  5:      Q <= DI;
        ENDIF;
      END PROCESS;

  6:  PROCESS(S3)
      BEGIN
  7:    IF (S3 ='1') THEN
  8:      DO <= Q;
        ELSE
  9:      DO <= '11111111';
        ENDIF;
      END PROCESS;

 10:  PROCESS(S2,STB)
      BEGIN
 11:    IF (S2 = '0') THEN
 12:      SRQ <= '1';
 13:    ELSE IF (STB = '1' AND NOT STB'STABLE) THEN
 14:      SRQ <= '0';
        ENDIF;
      END PROCESS;

 15:  S0 <= NOT NDS1 AND DS2 ;
 16:  S1 <= S0 AND MD OR STB AND NOT MD;
 17:  S2 <= S0 NOR NOT NCLR;
 18:  S3 <= S0 OR MD;
 19:  NINT <= NOT SRQ NOR S0;

      END BEHAVIOR;
```

Figure 1. VHDL behavioral description of Intel 8212 chip

implementation of this algorithm. It is because the dead-process fault can be tested by testing an assignment control fault on one of the assignment statements within the process.

For the IF-THEN-ELSE statement, there are the **stuck-THEN** fault and the **stuck-ELSE** fault. The stuck-THEN fault causes the statements under THEN control to be always executed regardless of the state of the condition tested by the IF statement. For the stuck-ELSE fault, the statements under ELSE control always execute.

For example:

```
IF C='1' THEN
   A <= D;
ELSE
   B <= D;
ENDIF;
```

In case of a stuck-THEN fault, the statement 'A < = D' is always executed regardless of the value of C (the control). In case of a stuck-ELSE fault, the statement 'B < = D' is always executed regardless of the value of C (the control).

For a CASE statement, there is the **dead-cause** fault. The dead-cause fault occurs when an assignment statement under CASE control is selected, but this assignment statement fails to execute.

For example:

```
CASE   E    is
   when 1 => A <= X;
   when 0 => B <= X;
END CASE;
```

In the case of a dead-cause fault, when E='1', the assignment statement 'A <= X' does not execute; when E='0', the assignment statement 'B <= X' does not execute.

For assignment statements, there is the **assignment control** fault. With an assignment control fault, the destination object of an assignment statement does not receive the result from the source expression.

For example: C <= A and B;

C is the destination object and ' A and B ' is the source expression of the assignment statement. In case of the assignment control fault, the destination object C does not receive the result from the source expression ' A and B '.

The **micro-operation** fault consists of perturbing one micro-operation to another micro-operation. The micro-operation perturbations are shown in Table 1 on page 23.

For data lines there is the **stuck-data** fault. The stuck-data fault causes the data line to be stuck-at '0' or stuck-at '1'.

Table 1. Micro-operation perturbations

Good Micro-operation	Fail to Micro-operation
AND	OR
OR	AND
XOR	EQV
NOT	BUF
EQV	XOR
ADD	SUB XOR
SUB	ADD

3.4 Timing considerations

For sequential circuits, a test for a fault is usually composed of a sequence of test vectors. The order in the sequence of test vectors is important. The order of the test vectors is kept by a time queue which is generated during test generation. Before test generation begins, a reference time is created. This reference time is denoted as t1. No testing activity happens at t1 or after t1. All testing activities happens before t1. The t1 is used as reference to create the latest time period. For example, if the latest time period is t2, then the time queue is ' t2, t1 '. During test generation, additional time periods can be created between t2 and t1, or before t2.

Each time period is assumed to be long enough for the signals in the source expression of an assignment statement to propagate to its destination object. Figure 2 on page 24, shows a VHDL behavioral description of a simple D flip-flop and its timing diagram. The timing diagram shows that when the CLK rises in the beginning of t2, the input signal D at the beginning of t2 latches to output Q at the end of t2.

```
If (CLK = '1' and not'stable) then
        Q <= D;
endif;
```

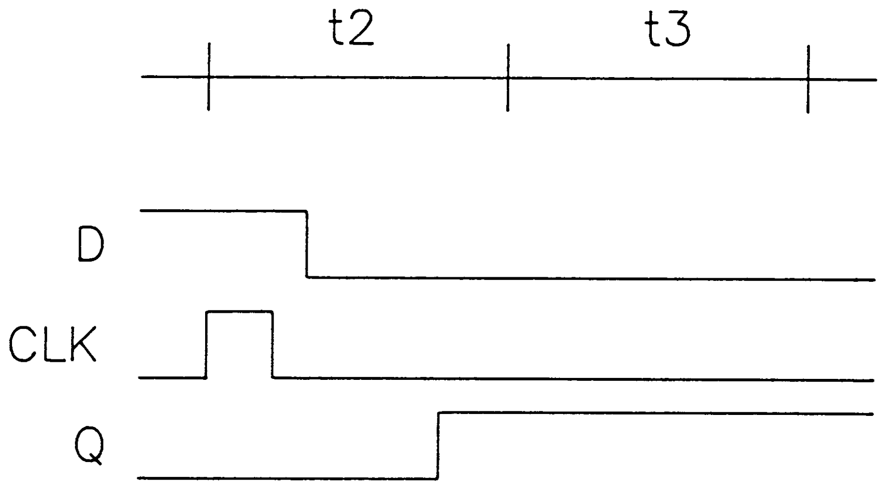


Figure 2. A VHDL behavioral description of a D flip-flop and its timing diagram

Chapter 4. New development

In this thesis, we focus on developing techniques to overcome the problems in generating test vector sets for digital circuits with reconvergent fanout and feed-back. The previous algorithm uses single path propagation of faults. Therefore, it cannot generate tests for most of the reconvergent fanout cases, except where the reconvergent fanout does not affect the single path propagation of fault. Also, the previous algorithm cannot handle sequential circuits with feed-back. The new algorithm includes new schemes to handle reconvergent fanout and an new technique to initialize the feed-back of sequential circuits. This chapter is devoted to the discussion of the new development.

4.1 Reconvergent fanout detection

Reconvergent fanout in digital circuits often creates conflict in signal value assignment during test generation. To overcome this problem, the test generation program must be aware of the reconvergent fanout situation and use rules to avoid conflicting signal value assignment. The issue here is how to find out the existence of reconvergent fanout. The fanout of a signal can be noticed by checking if the signal appears in more than one VHDL statements. However, to determine whether the fanout signal has reconverged is not very easy. All the fanout paths of the signal must be traced towards primary outputs, and one must check whether some of the paths have reconverged. This process is very time consuming, and it is frequently not feasible to perform this process for every signal in the circuit.

The reconvergent fanout detection method presented here does not use the approach mentioned above. This method only detects the reconvergent fanout situation that is on the fault propagation path and affects the fault propagation. This approach can efficiently handle the reconvergent fanout cases that are significant to the test generation.

The new reconvergent fanout detection scheme requires recording the objects which are on the fault propagation path. During justification of an object value for fault propagation, if one of the objects that needs to be justified is on the fault propagation path, then the reconvergent fanout situation on the fault propagation path is detected. For example, consider Figure 3 on page 27, where [b, c, d] is the fault propagation path. When propagating the fault from d to e, the f needs to be justified. Then, g and b need to be justified; and b is member of the fault propagation path [b, c, d]. Thus, reconvergent fanout on the fault propagation path is detected.

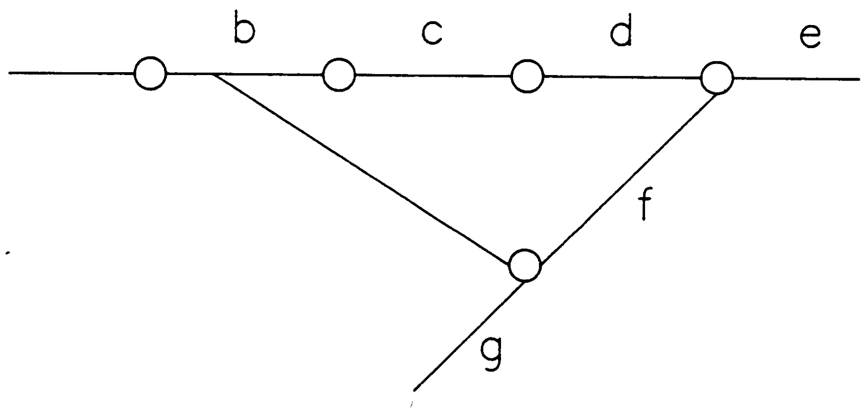


Figure 3. Reconvergent fanout on a fault propagation path

4.2 *Fault propagation through reconvergent fanout*

This section is devoted to the discussion of how the fault should be propagated through reconvergent fanout. Should it be propagated through one of the paths or propagated through all the fanout paths? The previous version of the test generation program propagates faults through one path regardless of the reconvergent fanout situation. Jani[21] suggested that a fault should be simultaneously propagated through all the fanout paths. The implementation of a program to accomplish the simultaneous propagation of fault through all fanout paths is very difficult; and fault simulation may have to be used. Fault simulation is very time consuming, and the test generation process should avoid using fault simulation. A better approach needs to be developed to handle the fault propagation through reconvergent fanout problem.

The scheme proposed here takes advantage of the simplicity of single path propagation whenever possible, and propagates the fault through the fanout paths towards the reconvergent point whenever necessary. To demonstrate this idea, consider the circuit in Figure 3 on page 27. Assume that a good/bad value pair from a fault site has propagated through the path [b, c, d]. When attempting to propagate the fault from d to e, the desired value for f needs to be justified. Then, the g needs to be justified to give the desired value to f. Note that, the b already contains a good/bad value pair, and it should not be justified to any other value unless the object b is not required to maintain the good/bad value pair. If the justification of g cannot give the desired value to f and b cannot be justified to the other value, then the good/bad value pair of b should be propagated through the path [b, f]. Therefore, the fault is propagated through two paths towards the reconvergent point e.

The advantage of this approach is that the reconvergent fanout situation in the circuit does not have to be detected before fault propagation, and the fault is not required to propagate through more than one path when it is not necessary.

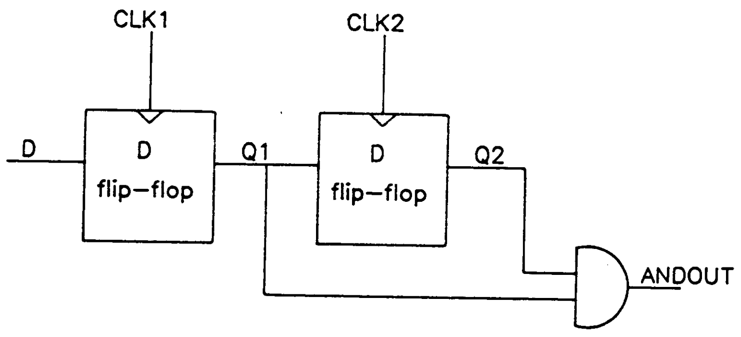


Figure 4. Simple circuit

4.3 *Improved fault sensitization scheme*

Fault sensitization includes selecting a fault site, choosing a good-value/bad-value pair that will sensitize the fault, and justifying some values to some objects in the circuit resulting in the good-value/bad-value pair being placed at the fault site.

A fault site is selected by choosing a fault in the fault list. The fault list is prepared based upon the predefined fault models and the VHDL circuit description.

The good-value/bad-value pair should be chosen such that it can be justified at the fault site. If the assignment statement (such as 'Q < = D') that is used to justify the good-value or bad-value does not have a literal source expression, the good-value or bad-value can be any value that sensitizes the fault. However, if the assignment statement that is used to justify the good-value or bad-value has a literal source expression, then the good-value or the bad-value has to equal to the literal source expression in order to have the justification be a success. For example, if the assignment statement, ' Q < = "00" ' , is used to justify a good-value or a bad-value to Q, this good-value or bad-value must be "00". It is not possible to justify a "11" to Q by using the assignment statement ' Q < = "00" '. The previous scheme for choosing good-value/bad-value pair did not take into consideration of this situation.

The new method for picking the good-value/bad-value pair takes into consideration the assignment statements which are used to justify the good-value/bad-value pair. A good-value is justified by using a **good statement**; and a bad-value is justified by using a **bad statement** or a statement that is not related to the fault site. A **good statement** is a statement that will be executed if the circuit is fault free. A **bad statement** is a statement that will be executed if the circuit is faulty. In the new method, if the good statement has a literal source expression, the value of the literal source expression is chosen to be a good-value. Similarly, if the statement that is used to justify the bad-value has a literal source expression, the value of the literal source expression is chosen to be a bad-value.

Fault sensitization for a micro-operation fault often requires justifying two object values. Sometimes, justification of the first object value makes the second object value impossible to be justified without erasing the value of the first object. If the second object value is justified first, the first object value can also be justified. Consider the circuit in Figure 4 on page 29, To test the AND fails to OR micro-operation fault requires a '0' and a '1' at the input of the AND gate; and the output of the AND gate will have a good-value/bad-value of 0/1. In this case, Q1 and Q2 are the inputs of the AND gate, either Q1='0', Q2='1' or Q1='1', Q2='0' can be chosen. Note that justification of Q1 first makes Q2 impossible to be justified without erasing the value in Q1. However, if Q2 is justified first, then Q1 can also be justified in this case. A fault sensitization procedure for micro-operation faults which can handle these situations efficiently is described in chapter 5 section 5.2.3.

4.4 Improved justification scheme with dynamic controllability

During the justification process, a reconvergent fanout circuit may create a conflict in object value assignment. To avoid the conflict, if the object in the circuit has already been assigned a value and a different value should not be assigned to that object, then this situation must be noted. And, the test generation program should not assign new values to these objects. The ability of an object to accept new value changes during a test generation process is called dynamic controllability.

The dynamic controllability of an object is redetermined after an assignment statement is executed. By observation, when an assignment statement is controlled by some control statement, the destination object of the assignment statement will not be affected by the changes of the source expression unless the control condition of the control statement is satisfied. For example, the assignment statement S1 in Figure 5 on page 32 will not be executed unless the signal CLK goes from '0' to '1'. After

```
        If (CLK = '1' and not CLK'stable) then
S1:         Q <= D;
        endif;


        If C = '1' then
S2:         Z <= A and B;
        endif;


S3:         Y <= A and B;
```

Figure 5. VHDL assignment statements

the statement S1 has executed, the destination object Q will contain a new value, and the object Q should not be assigned a different value until the new value in Q is transferred to another object and the change of Q will not affect the value of that object. The source expression D is able to accept the assignment of any value, because any value in D will not affect the value in Q unless the signal CLK goes from '0' to '1' again. The assignment statement S2 in Figure 5 will not be executed unless the object C is set to logic '1'. After assignment S2 is executed, the object C will still have a value of '1' unless it is reset to '0'. As long as the object C has a value of '1', the change of object values in the source expression ' A and B ' will affect the value of the destination object Z. Therefore, the objects in the source expression and destination object of S2 should not be assigned new values after the statement S2 is executed. For the assignment statement S3 in Figure 5, there is no control statement governing its execution. The change of object value in the source expression ' A and B ' will affect the value of the destination object Y. Therefore, after the assignment statement S3 is executed, its objects in the source expression and destination object should not be assigned new values.

The ability of each object in the assignment statement to accept new value is recorded after the assignment statement is executed. During the justification process, if the object is marked not to accept a new value, the new value will not be assigned to that object and an alternative way would be chosen to satisfy the justification process if possible.

This new justification with dynamic controllability scheme is especially effective in solving the conflicting object value assignment problem in reconvergent fanout circuits.

4.5 Improved execution scheme

The execution process tries to set up the control condition of an assignment statement, such that the assignment statement will be executed. The previous execution scheme assumed that an assignment statement, which has no control statement governing its execution, would not be executed unless an object value in the source expression had changed. Therefore, in order to have those assignment statements executed, a value that is different from the desired value had to be assigned to an object before the desired value was assigned to that object. This assumption is not realistic relative to actual logic. For example, for the output of an AND gate be '1', both the A and the B inputs of the AND gate must be set to '1'. However, the previous value of A or B is not required to be '0'. Therefore, assigning 0 to A or to B before assigning '1' to both the A and the B is not required.

The new execution scheme has separated the assignment statements not governed by control statements from the statements which are governed by some control statements. For the assignment statement not governed by a control statement, the statement is always able to be executed without setting up a change in object value. For the statements which are governed by some control statement, the control condition is set up to allow the statement to be executed.

This improved execution scheme is designed for the implementation of justification with dynamic controllability.

4.6 Improved two-phase testing scheme

Two-phase testing includes a preloading phase and a data transfer phase. If the fault site is traversed during preloading phase, the result of the preloading operation is propagated to an observable output. In the process of propagating this result to the observable output, many values in objects along the propagation path must be assigned in order to accomplish the propagation. In the data transfer phase, new values are assigned to inputs during the test generation process. The change of input object values implies the change of some internal object values. However, the previous test program does not check every internal object value after a new value is assigned to an input. Therefore, some internal object values which are assigned in the preloading phase are out-dated.

The solution for this problem is to remove all the internal object values after the result of the preloading operation is propagated to an observable output. Therefore, in the data transfer phase, the internal objects will not contain any values which were assigned during the preloading operation.

4.7 New scheme to handle feed-back and looping

Feed-back and looping in digital sequential circuits are difficult problems for automatic test vector generation. Algorithmic automatic test generation approaches, such as the D-algorithm, PODEM, and FAN algorithm, do not have the capability to handle feed-back and looping in sequential circuits. Consider the circuit in Figure 6 on page 36, which shows a feed-back situation in digital circuit. The circuit inside the logic black box operates depending upon the control signal and the input signal IN; and the control signal depends upon the output signal OUT and the signal Q. Before the logic black box operates, the output signal OUT is unknown, and therefore, the control signal is unknown. The logic black box cannot operate when the control signal is unknown. It would be impossible to propagate a sensitized fault inside the logic block

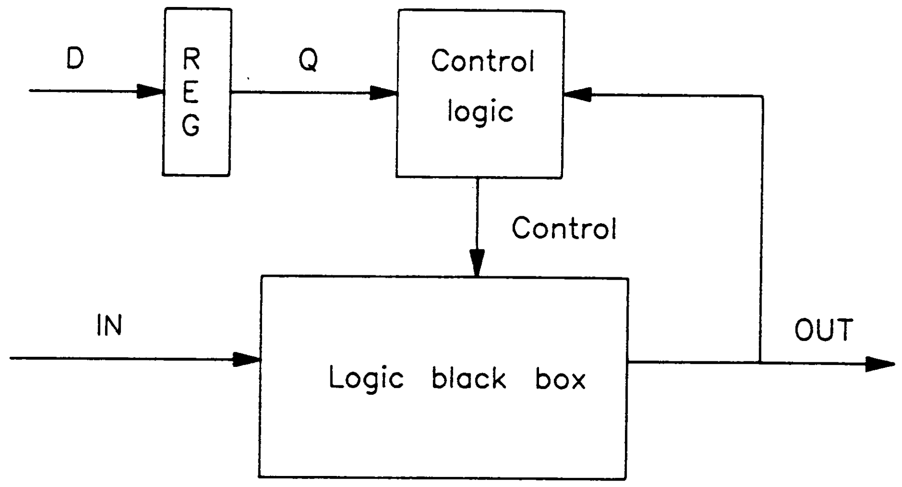


Figure 6. Feed-back in digital circuits

box to the observable output OUT in this situation. In Figure 7 on page 37, a looping situation is shown. It is a state diagram that shows four states, "00", "01", "10", "11". If the test generation process requests that the circuit go to a particular state, for example state "01", the test generation program would find out that the previous state of "01" is "00", and check whether the current state of the circuit is "00". If the current state of the circuit is unknown, the test generation program cannot make this determination, and thus will be unable to drive the circuit to state "01".

The problems discussed above are caused by the initialization problem in sequential circuit testing. The design of a sequential circuit always includes logic to initialize the circuit into its initial state. For example, a counter circuit includes the logic to clear

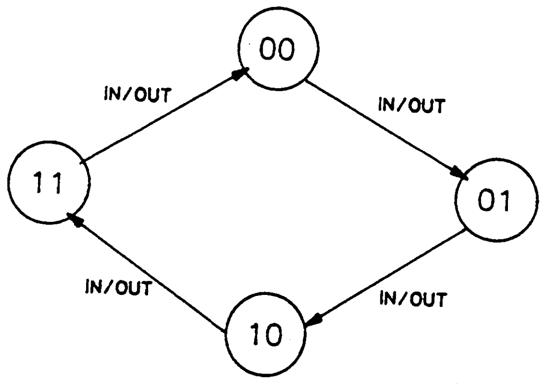


Figure 7. Looping situation

the counter (set counter output to a vector of "0"s) without activate the counting function.

In the testing of sequential circuits, the unknown initial state of the sequential circuit often creates unknown values in some logic lines. This causes the circuits with feed-back not to be testable due to the unknown feed-back values. To overcome this problem, the feed-back value must be initialized before a test generation process begins. The initialization logic could be affected by the fault that exists in the circuit. However, if the initialization logic is at fault, it should be detected by observing that the circuit cannot be initialized properly. For the problem in Figure 6 on page 36, the output signal OUT and the signal Q must be initialized before test generation begins. For the problem in Figure 7, the current state must be known and the circuit must be initialized to its initial state before test generation begins.

The previous implementation of our test generation program does not have the capability to drive the circuit into its initial state. The new improved implementation has taken advantage of the VHDL behavioral description. From the VHDL description, one can easily find out which VHDL behavioral description statement can be used to initialize the circuit or to set the circuit to a particular state. The user inputs a VHDL statement number and the state, and then the new test program will drive the circuit to that state by setting up the appropriate inputs of the circuit.

The new test generation scheme for sequential circuits with feed-back and/or looping includes two steps. The first step is circuit initialization. The second step is test vector generation. In the first step, a circuit should be initialized to an initial state and then driven to a particular state if necessary. Also, some internal object(s) in the circuit may have to be initialized before the circuit can begin its normal operation. For example, consider the circuit in Figure 6 on page 36, the object Q must also be initialized before the circuit begins its normal operation. The test vector generation step is the normal test generation procedure, which includes fault sensitization, fault propagation, justification, and execution.

Chapter 5. The improved test generation algorithm

The new improved test generation algorithm is designed to handle test vector generation for combinational and sequential digital circuits with reconvergent fanout, feed-back and looping. The digital circuits under test should be modeled with a VHDL behavioral description by using the VHDL sub-set that is described in chapter 3. A fault list is generated in the preprocessing step from the VHDL behavioral description and the defined fault models. For sequential circuits with feed-back and/or looping, the initialization step should be used to initialize the circuit. The test generation process starts with fault sensitization. After that, the sensitized fault is propagated to an observable output with the guidance of heuristic rules. The output of the test generation program is a sequence of test vectors that will test a fault.

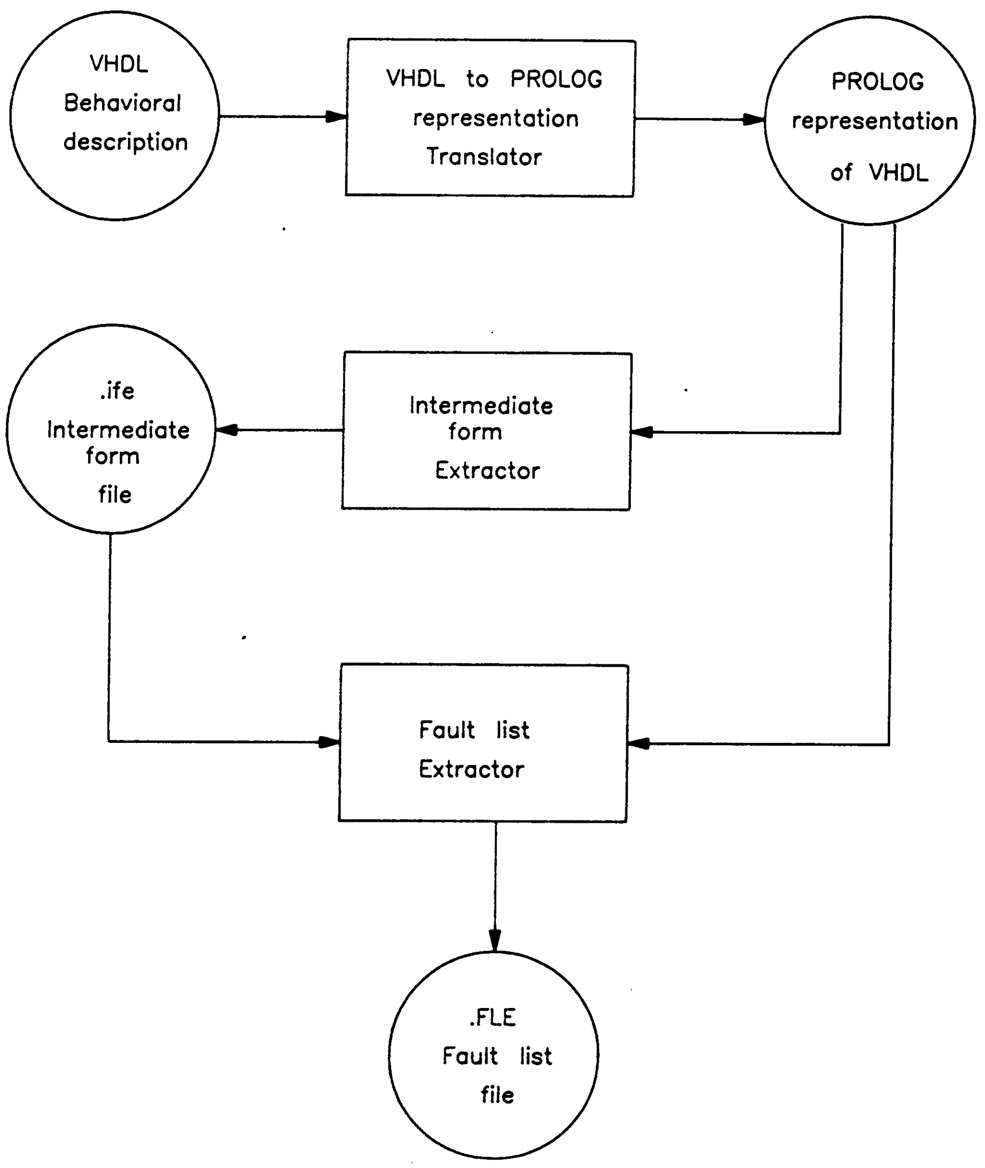

VHDL Behavioral description
VHDL to PROLOG representation Translator
PROLOG representation of VHDL
.ife Intermediate form file
Intermediate form Extractor
Fault list Extractor
.FLE Fault list file

Figure 8. Test generation preprocessor

```
File CKA.HDL

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

    architecture BEHAVIOR of CKTA is

      signal Q1, Q2 : BIT;

    process(CLK1)
    begin
1:    if (CLK1='1' and not(CLK1'stable)) then
2:       Q1 <= D;
    end process;

    process(CLK2)
    begin
3:    if (CLK2='1' and not(CLK1'stable)) then
4:       Q2 <= Q1;
    end process;

5:    ANDOUT <= Q1 AND Q2;

    end BEHAVIOR;
```

Figure 9. VHDL behavioral description

5.1 Test generation preprocessor

The test generation preprocessor is composed of a VHDL to PROLOG representation translator, an intermediate form extractor, and a fault list extractor. A block diagram of the preprocessor is shown in Figure 8 on page 40. The research in this thesis has not altered the implementation of this test generation preprocessor.

```
fileprefix("cka").
modelname("CKA").

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

statementtype(s1,if).
controlexpression(s1,[biteqv,[obj,clk1],[lit,[bit,"R"]]]).
subordinaterange(s1,then,[s2]).
subordinaterange(s1,else,[]).

statementtype(s2,assignment).
sourceexpression(s2,[obj,d]).
destinationobject(s2,q1).
edge_response(s2,true).

statementtype(s3,if).
controlexpression(s3,[biteqv,[obj,clk2],[lit,[bit,"R"]]]).
subordinaterange(s3,then,[s4]).
subordinaterange(s3,else,[]).

statementtype(s4,assignment).
sourceexpression(s4,[obj,q1]).
destinationobject(s4,q2).
edge_response(s4,true).

statementtype(s5,assignment).
sourceexpression(s5,[bitand,[obj,q1],[obj,q2]]).
destinationobject(s5,andout).

?- end.
```

Figure 10. PROLOG representation of a VHDL behavioral description

The VHDL to PROLOG translator was implemented with the C language in the very early stages of the development of this test generation algorithm. Due to the improvement and the changes of the test generation algorithm, this translator is no longer up to date. Therefore, the VHDL to PROLOG representation is presently

```
statements([s1, s2, s3, s4, s5]).
objects([d, clk1, clk2, q1, q2, andout]).
clauses(s1, [else, then]).
clauses(s3, [else, then]).
assignments(d, []).
assignments(clk1, []).
assignments(clk2, []).
assignments(q1, [s2]).
assignments(q2, [s4]).
assignments(andout, [s5]).
uses(d, [[s2, [45], []]]).
uses(clk1, [[s1, [45], [76]]]).
uses(clk2, [[s3, [45], [76]]]).
uses(q1, [[s5, [45], [76]], [s4, [45], []]]).
uses(q2, [[s5, [45], [82]]]).
uses(andout, []).
parentstatement(s2, s1).
parentstatement(s4, s3).
parentstatement(s1, []).
parentstatement(s3, []).
parentstatement(s5, []).
subordinatelist(s3, else, []).
subordinatelist(s3, then, [s4]).
subordinatelist(s1, else, []).
subordinatelist(s1, then, [s2]).
made_vies(s1, [s1],
[[[bit, [49]], [biteqv, [obj, clk1], [lit, [bit, [82]]]], s1]]) :- !.
made_vies(s3, [s3],
[]) :- !.
made_vies(s4, [s3, s4],
[[[bit, [49]], [biteqv, [obj, clk2], [lit, [bit, [82]]]], s3]]) :- !.
made_vies(s5, [s5],
[[[bit, [49]], [bitor, [bitnot, [stable, [obj, q1]]], [bitnot, [stable, [obj, q2]]]], s5]]) :- !.
evaluated_pos(s1, [76], [], biteqv, [76], [obj, clk1], [lit, [bit, [82]]]) :- !.
evaluated_pos(s1, [82], [], biteqv, [82], [lit, [bit, [82]]], [obj, clk1]) :- !.
evaluated_pos(s3, [76], [], biteqv, [76], [obj, clk2], [lit, [bit, [82]]]) :- !.
evaluated_pos(s3, [82], [], biteqv, [82], [lit, [bit, [82]]], [obj, clk2]) :- !.
evaluated_pos(s5, [76], [], bitand, [76], [obj, q1], [obj, q2]) :- !.
evaluated_pos(s5, [82], [], bitand, [82], [obj, q2], [obj, q1]) :- !.
evaluated_pos(_2393, [], [], [], [], [], []) :- !.
```

Figure 11. Intermediate form file

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

Figure 12. Fault list file

translated manually. The PROLOG representation is stored in an .hdl file. An example of a VHDL behavioral description is shown in Figure 9 on page 41 and the PROLOG representation of this VHDL is shown in Figure 10.

The intermediate form extractor is implemented with PROLOG. It takes the PROLOG representation of VHDL behavioral description as input, and produces an .ife file as output. This .ife file contains the PROLOG facts that represent the knowledge of the circuit and information on selecting paths for propagation, justification, and execution [21]. The .ife file from the PROLOG representation of VHDL in Figure 10 on page 42 is shown in Figure 11 on page 43.

The fault list extractor, which is implemented with PROLOG, takes the PROLOG representation of VHDL file and the intermediate form file as input, and produces a .fle file as output. This output file contains a fault list which represents all the modeled fault for the circuit. An example of fault list is shown in Figure 12 on page 44.

5.2 Test generation process

The test generation process begins after the preprocessing step is completed. The test generation program takes the three files (.hdl, .ife, .fle) as input and produce a test vector file as output. A block diagram of the input/output files of the test generation program is shown in Figure 13 on page 46.

In this test generation program, faults are sensitized in the VHDL description and propagated through VHDL statements. No fault simulation is required. The test generation process is guided with heuristic rules and is organized in the form of a goal tree. A high level goal tree of the test generation program is shown in Figure 14 on page 47. The main goal of the goal tree is to generate a test for a selected fault. This

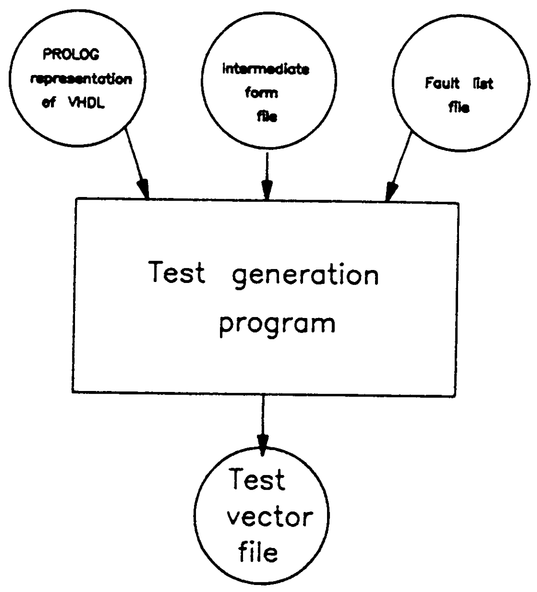


Figure 13. Test generation program input and output files

main goal is then broken down into three sub-goals, which are circuit initialization, fault sensitization, and fault propagation. If all these three sub-goals are solved, the main goal is solved. The circuit initialization sub-goal is optional, the user can choose to initialize the circuit or not to initialize the circuit. As the discussion in chapter 4 showed, sequential circuits with feed-back and/or looping need to be initialized. The fault sensitization sub-goal is solved according to the type of fault for which a test is being developed, i.e., there are different types of schemes to sensitize different types of faults. For the fault propagation sub-goal, there are different heuristic rules to guide the fault to propagate through different types of VHDL statements and to handle special circuit configurations, e.g., reconvergent fanout. The next level of sub-goals are justification and execution. These two sub-goals are used to

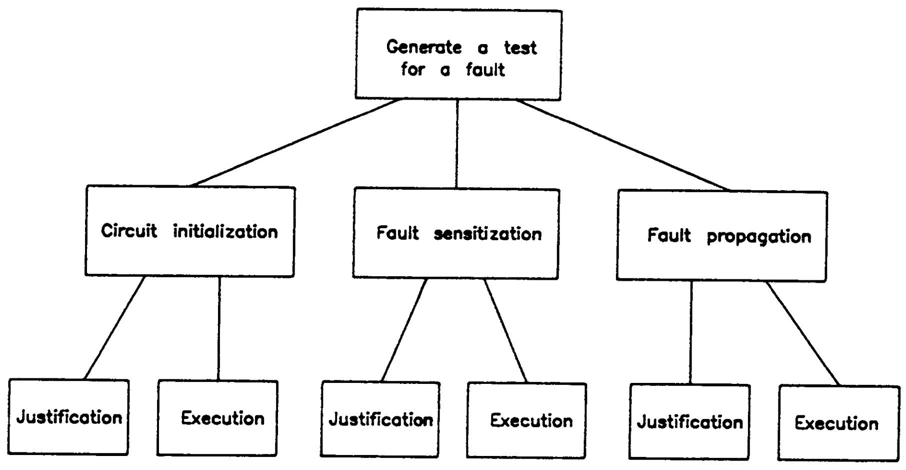


Figure 14. Test generation high level goal tree

assign values to the objects in the circuit. They are used in solving circuit initialization, fault sensitization, and fault propagation sub-goals.

5.2.1 Circuit initialization

As discussed in chapter 4, sequential circuits with feed-back and/or looping require initialization in order to have the test generation process succeed. The initialization step is a built-in option in the test generation program. A user can choose to initialize the destination object of any assignment statement in the VHDL description. It can be done by providing the VHDL statement number and the value that is desired to be placed at the destination object of the assignment statement. Based upon this provided information, the test program will generate all the input requirements to place the desired value in that destination object. Note that, for circuits without feed-back and/or looping circuit initialization is not needed. For these circuits, the test program can generate the test completely automatically.

5.2.2 Test generation procedures

Definitions:

Good statement - a good statement is a statement that will be executed if a circuit is fault free.

Bad statement - a bad statement is a statement that will be executed if a circuit is faulty.

Assignment control fault:

Given a statement number:

[1]. Record that this statement is a faulty statement.

[2]. Get the destination object of this statement.

[3]. Get the source expression of this statement.

[4]. Pass the statement number, source expression, and destination object to the fault sensitization schemes to preform fault sensitization.

[5]. Propagate the sensitized good/bad value pair from this statement towards an observable output.

Stuck-then-fault:

Given a statement number:

[1]. Get the list of statements under THEN, and mark this list of statements as bad statements.

[2]. Record that this statement is a faulty statement.

[3]. Use the fault sensitization schemes to propagate the good/bad value pair of 0/1 through this statement to an assignment statement under this control construct.

[4]. Propagate the good/bad value pair from the assignment statement towards an observable output.

Stuck-else-fault:

Given a statement number:

[1]. Get the list of statements under ELSE, and mark this list of statements as bad statements.

[2]. Record that this statement is a faulty statement.

[3]. Use the fault sensitization schemes to propagate the good/bad value pair of 1/0 through this statement to an assignment statement under this control construct.

[4]. Propagate the good/bad value pair from this assignment statement towards an observable output.

Micro-operation fault:

Given a statement number, the position of the micro-operation, the good micro-operation, and the bad micro-operation:

[1]. If the micro-operation is in a control statement, record that all the assignment statements under the control statement are bad statements.

[2]. Get the operands of the micro-operation.

[3]. Use the fault sensitization schemes to sensitized the fault.

[4]. Propagate the good/bad value pair from this statement towards an observable output.

Dead-clause fault:

Given a statement number and a dead-clause:

[1]. Get an assignment statement which is under the control of the clause.

[2]. Perform an assignment control fault test on this assignment statement.

Stuck-data fault:

Given a statement number, the position, and the stuck value:

[1]. If this statement is a control statement, get the assignment statements which are under the control statement. Then record that these assignment statements are bad statements.

[2]. Record that this statement is a faulty statement.

[3]. Get the object which is stuck.

[4]. Use fault sensitization schemes to sensitize the fault.

[5]. Propagate the good/bad value pair towards an observable output.

5.2.3 Fault sensitization

The purpose of fault sensitization is to assign values to the objects in the circuit under test, such that the faulty circuit will have different outputs from the fault free circuit. The fault sensitization techniques are different for each behavioral fault model.

Assignment control fault:

Existence of an assignment control fault on an assignment statement will prevent the destination object of the assignment from accepting the result from its source expression. To sensitize the assignment control fault, first, a known value (bad value) must be preloaded into the destination object, and then, some appropriate values justified in the source expression such that a value (good value) which is different from the preloaded value will transfer to the destination object if the circuit is fault free. In selecting a good/bad value pair, if the source expression of the statement which is used to preload or to transfer data has a literal source expression, the good or bad value should be chosen according to the source expression. A detail discussion of the rationale for selecting the good/bad value pair is in section 4.3.

Fault sensitization procedure:

Given a statement number, source expression, and destination object:

[1]. Use good/bad value selection rules to select a good value and a bad value.

[2]. Check if the destination object already contains a value which is equal to the bad value. If no, assign (preload) the bad value to the destination object, and if

this statement (faulty statement) is used, propagate the result of the preloading to an observable output.

[3]. Justify the source expression of this statement, such that the good value will be transferred to the destination object of this statement if the circuit is fault free.

[4]. Use~~d~~ execution schemes to execute this statement.

Rules for selecting good/bad values:

Rule 1: If the statement (faulty statement) has a literal source expression, then take the literal value as good value. If there is another assignment statement in the VHDL description with a same destination object and it also has a literal source expression, then take the literal value as bad value. If the bad value is equal to the good value or the bad value is not yet picked, choose a bad value which is different from the good value.

Rule 2: If this statement (faulty statement) does not have a literal source expression, check if there is another assignment with the same destination object having a literal source expression. If yes, then take the literal value as bad value. Choose a good value which is different from the bad value.

Rule 3. If this statement (faulty statement) does not have a literal source expression, and no other assignment statement with a same destination object has literal source expression, then pick a good value and choose a bad value which is different from the good value.

Stuck-then-fault:

In a IF-THEN-ELSE control construct, the existence of a stuck-then-fault will prevent the statements under ELSE from being executed. The objective of the fault sensitization is to set up some values in some objects of the circuit under test, such that execution of a statement under ELSE can be observed if the circuit is fault free. If there is no statement under ELSE, a statement under THEN should be observed.

Case I. Matching destination object assignment statements exist under THEN and under ELSE.

For Example:

```
IF   condition   THEN
     Q <=  ... ;
ELSE
     Q <=  ... ;
ENDIF;
```

Fault Sensitization procedure:

[1]. Find an assignment statement under ELSE.

[2]. Find an assignment statement under THEN, where this assignment statement has the same destination object as the selected statement under ELSE.

[3]. Choose a good value and a bad value.

[4]. Justify the bad value into the destination object of the assignment statement under THEN.

[5]. Justify the good value into the destination object of the assignment statement under ELSE.

[6]. Record the destination object as a member of the fault propagation path.

[7]. Set up conditions such that the assignment statement under ELSE will be executed if the circuit is fault free.

Case II. No assignment statements exist under ELSE, and no matching destination object assignment statement exists outside the IF-THEN construct.

For Example:

```
IF    condition    THEN
      Q < = ... ;
ENDIF;
```

Fault sensitization procedure:

[1]. Find an assignment statement under THEN.

[2]. Choose a good value and a bad value.

[3]. Justify the good value into the destination object of the assignment statement under THEN.

[4]. Set up conditions such that the assignment statement under THEN will be executed if the circuit is fault free.

[5]. Justify the bad value into the destination object of the assignment statement under THEN.

[6]. Record the destination object as a member of the fault propagation path.

[7]. Set up conditions such that the assignment statement under THEN will not be executed if the circuit is fault free; and the assignment statement under THEN will be executed if the circuit is faulty.

Case III. No assignment statements exist under ELSE, but a matching destination object assignment statement exists outside the IF-THEN construct.

For Example:

```
          Q <= ... ;
      IF     condition    THEN
          Q <= ... ;
      ENDIF;
```

Fault sensitization procedure:

[1]. Find an assignment statement under THEN.

[2]. Find an assignment statement outside the IF-THEN control construct where this assignment statement has the same destination object as the selected assignment statement inside the control construct.

[3]. Choose a good value and a bad value.

[4]. Justify the good value into the destination object of the assignment statement outside the IF-THEN control construct.

[5]. Justify the bad value into the destination object of the assignment statement under the THEN clause.

[6]. Record the destination object as a member of the fault propagation path.

[7]. Set up conditions such that the assignment statement under THEN will not be executed if the circuit is fault free.

Case IV. No assignment statement exists that has a destination object that matches a destination object of an assignment statement under ELSE.

For Example:

```
IF    condition    THEN
      A < = ... ;
ELSE
      B < = ... ;
ENDIF;
```

Fault sensitization procedure:

[1]. Find an assignment statement under ELSE.

[2]. Choose a good value and a bad value.

[3]. Justify the bad value into the destination object of the assignment statement under ELSE.

[4]. Set up conditions such that the assignment statement under ELSE will be executed if the circuit is fault free.

[5]. Propagate the bad-value/dummy pair to an observable output. ("dummy" is a value that is not any particular value but is different than the bad value).

[6]. Justify the good value into the destination object of the assignment statement under ELSE.

[7]. Record the destination object as a member of the fault propagation path.

[8]. Set up conditions such that the assignment statement under ELSE will be executed if the circuit if fault free.

Case V. There is an assignment statement outside the IF-THEN-ELSE construct that has a destination object that matches a destination object of an assignment statement under ELSE; but no assignment statement under THEN has the same destination object.

For Example:

```
          B <= ... ;
IF    condition    THEN
          A <= ... ;
ELSE
          B <= ... ;
ENDIF;
```

Fault sensitization procedure:

[1]. Find an assignment statement under ELSE.

[2]. Find an assignment statement outside the IF-THEN-ELSE control construct where this assignment statement has the same destination object as the selected assignment statement inside the IF-THEN-ELSE control construct.

[3]. Choose a good value and a bad value.

[4]. Justify the bad value into the destination object of the assignment statement outside the IF-THEN-ELSE control construct.

[5]. Justify the good value into the destination object of the assignment statement under ELSE.

[6]. Record the destination object as a member of the fault propagation path.

[7]. Set up conditions such that the assignment statement under ELSE will be executed if the circuit is fault free.

Stuck-else-fault:

In a IF-THEN-ELSE control construct, the existence of a stuck-else-fault will prevent the statements under THEN from being executed. The objective of the fault sensitization is to set up values in some objects of the circuit under test, such that execution of a statement under THEN can be observed if the circuit is fault free.

Case I. Matching destination object assignment statements exist under THEN and under ELSE.

For Example:

```
IF   condition   THEN
     Q <=  ... ;
ELSE
     Q <=  ... ;
ENDIF;
```

Fault Sensitization procedure:

[1]. Find an assignment statement under THEN.

[2]. Find an assignment statement under ELSE, where this assignment statement has a same destination object as the selected statement under THEN.

[3]. Choose a good value and a bad value.

[4]. Justify the bad value into the destination object of the assignment statement under ELSE.

[5]. Justify the good value into the destination object of the assignment statement under THEN.

[6]. Record the destination object as a member of the fault propagation path.

[7]. Set up conditions such that the assignment statement under THEN will be executed if the circuit is fault free.

Case II. No assignment statements exist under ELSE, and no matching destination object assignment statement exists outside the IF-THEN construct.

For Example:

```
IF    condition    THEN
      Q <= ... ;
ENDIF
```

Fault sensitization procedure:

[1]. Find an assignment statement under THEN.

[2]. Choose a good value and a bad value.

[3]. Justify the bad value into the destination object of the assignment statement under THEN.

[4]. Set up conditions such that the assignment statement under THEN will be executed if the circuit is fault free.

[5]. Propagate the bad_value/dummy to an observable output.

[6]. Justify the good value into the destination object of the assignment statement under THEN.

[7]. Record the destination object as a member of the fault propagation path.

[8]. Set up conditions such that the assignment statement under THEN will be executed if the circuit is fault free; and the assignment statement under THEN will not be executed if the circuit is faulty.

Case III. No assignment statements exist under ELSE, but a matching destination object assignment statement exists outside the IF-THEN construct.

For Example:

```
      Q < = ... ;
IF    condition   THEN
      Q < = ... ;
ENDIF;
```

Fault sensitization procedure:

[1]. Find an assignment statement under THEN.

[2]. Find an assignment statement outside the IF-THEN control construct where this assignment statement has the same destination object as the selected assignment statement inside the control construct.

[3]. Choose a good value and a bad value.

[4]. Justify the bad value into the destination object of the assignment statement outside the IF-THEN control construct.

[5]. Justify the good value into the destination object of the assignment statement under THEN.

[6]. Record the destination object as a member of the fault propagation path.

[7]. Set up conditions such that the assignment statement under THEN will be executed if the circuit is fault free.

Case IV. No assignment statement exists that has a destination object that matches a destination object of an assignment statement under THEN.

For Example:

```
IF    condition    THEN
      A < = ... ;
ELSE
      B < = ... ;
ENDIF
```

Fault sensitization procedure:

[1]. Find an assignment statement under THEN.

[2]. Choose a good value and a bad value.

[3]. Justify the bad value into the destination object of the assignment statement under THEN.

[4]. Set up conditions such that the assignment statement under THEN will be executed if the circuit is fault free.

[5]. Propagate the bad-value/dummy to an observable output. ("dummy" is a value that is not any particular value but is different than the bad value).

[6]. Justify the good value into the destination object of the assignment statement under THEN.

[7]. Record the destination object as a member of the fault propagation path.

[8]. Set up conditions such that the assignment statement under THEN will be executed if the circuit is fault free.

Case V. There is an assignment statement outside the IF-THEN-ELSE construct that has a destination object that matches a destination object of an assignment statement under THEN; but no assignment statement under ELSE has the same destination object.

For Example:

```
          A <= ... ;
IF     condition     THEN
          A <= ... ;
ELSE
          B <= ... ;
ENDIF
```

Fault sensitization procedure:

[1]. Find an assignment statement under THEN.

[2]. Find an assignment statement outside the IF-THEN-ELSE control construct, where this assignment statement has the same destination object as the selected assignment statement inside the IF-THEN-ELSE control construct.

[3]. Choose a good value and a bad value.

[4]. Justify the bad value into the destination object of the assignment statement outside the IF-THEN-ELSE control construct.

[5]. Justify the good value into the destination object of the assignment statement under THEN.

[6]. Record the destination object as a member of the fault propagation path.

[7]. Set up conditions such that the assignment statement under THEN will be executed if the circuit is fault free.

Micro-operation fault:

Testing of a micro-operation fault assumes that a good micro-operation fails to a bad micro-operation. The objective of the sensitization is to set up the appropriate values in the operands of the micro-operation, such that the result of the good micro-operation is different from the result of the bad micro-operation.

Fault sensitization procedure:

Given a good micro-operation, a bad micro-operation,
the operands of the micro-operation, and a statement number:

[1].Use the look-up table in the data base to determine the value of the left operand, the value of the right operand, and the resulting good/bad value pair of the micro-operation.

[2]. Justify the left value to the left operand and record which objects in the circuit cannot be assigned a new value, because these objects are holding some values for this justification.

[3]. Justify the right value to the right operand. If this right value cannot be justified due to some objects in the circuit already holding some values for the justification of the left operand and a new value cannot be assigned to these objects,

then remove all the values previously assigned to objects and try to justify the right operand first, and then try to justify the left operand second.

[4]. Execute the assignment statement.

Stuck-data fault:

The test for a stuck-data fault checks if an object is stuck at a fixed value. Therefore, a test value that is the complement of the stuck value must be justified to the faulty object. If the circuit is fault free, the object will accept the test value. Otherwise, the object stuck at a fixed value.

Fault sensitization procedure:

Given a statement number, a stuck value, and the object that is stuck:

[1]. Choose a test value which is the complement of the stuck value.

[2]. Justify the test value to the object that is stuck.

[3]. Record the object as a member of the fault propagation path.

5.2.4 Fault propagation

The propagation goal tries to propagate the good/bad value pair in a VHDL statement towards an observable output. Given a good/bad value pair and its location in the VHDL description, the propagation goal will set up all the input requirements to propagate the good/bad value pair to an observable output.

Rule for choosing good/bad values:

These rules are for propagation of a good/bad value pair through a control construct. Since the good/bad value pair arriving at the control construct does not have direct data path to an observable output, the effect of the good/bad value pair at the control construct must be observed through an assignment statement under the control construct. Therefore, a good/bad value pair must be chosen for the assignment statement. The good/bad value pair will be propagated towards an observable output through this assignment statement. These rules are defined based upon the existence of a "good" statement and a "bad" statement. A good statement is the statement that will be executed if the circuit is fault free. A bad statement is the statement that will be executed if the circuit is faulty.

Rule 1: There is a good statement but no bad statement. If the good statement has a literal source expression, take the literal value as the good value. Check if there is another assignment statement with the same destination object that has a literal source expression. If yes, then take the literal value as the bad value. If both the good value and the bad value are chosen, then stop. If either the good value or the bad value is chosen, then choose the bad value or the good value such that the good value is not equal to the bad value. If neither the good value nor the bad value is chosen, then pick a good value and a bad value such that the good value is not equal to the bad value.

Rule 2: There is a bad statement but no good statement. If the bad statement has a literal source expression, take the literal value as the bad value. Check if there is another assignment statement with the same destination object that has a literal source expression. If yes, then take the literal value as the good value. If both the good value and the bad value are chosen, then stop. If either the good value or the bad value is chosen, then choose the bad value or the good value such that the good value is not equal to the bad value. If neither the good value nor the bad value is chosen, then pick the good value and the bad value such that the good value is not equal to the bad value.

Rule 3. There is a good statement and a bad statement. If the good statement has a literal source expression, then take the literal value as the good value, else pick a good value. If the bad statement has a literal source expression and the literal value is not equal to the good value, then take the literal value as the bad value, else choose the bad value such that the bad value is not equal to the good value.

Propagation through data path:

Case 1. Data path through an assignment statement.

For example: Z <= A or B;

If 'A' contains a good/bad value pair, then 'B' must be set to '0' in order to have the good/bad value pair propagate to the destination object 'Z'.

Propagation procedure:

Given a good/bad value, its location, and a statement number.

[1]. Get the micro-operation and the objects in the source expression.

[2]. Use the look-up table in the data base to determine the required object value for the other object(s) in the source expression in order to have the good/bad value pair propagate to the destination object.

[3]. Justify the required object value(s) in the source expression.

[4]. Use execution goals to execute this assignment statement.

[5]. Record the destination object as a member of the fault propagation path.

Case 2. Data path through a control statement.

For example: If (A or B) then .... ;

If 'A' contains a good/bad value pair, then 'B' must be set to '0' in order to have the good/bad value pair propagate to the control expression. The good/bad value pair is then propagated through the control expression.

Propagation procedure:

Given a good/bad value, its location, and a statement number:

[1]. Use the procedure in case 1 to propagate the good/bad value pair to the control expression.

[2]. Propagate the good/bad value pair through the control expression.

Propagation through the control construct:

In a control expression, logic value '1' represents the case when the control expression is true, logic value '0' represents the case when the control expression is false. The **good clause** is the clause that will be executed if the circuit is fault free. The statements in the good clause are "good" statements. The **bad clause** is the clause that will executed if the circuit is faulty. The statements in the bad clause are "bad" statements.

For example:

```
        If   condition   then
S1:          Z <= A or B;
        else
S2:          Z <= A and B;
        endif;
```

If the condition is 1/0 (the condition is true in the fault free circuit, and is false in the faulty circuit), the statement S1 will be executed in a fault free circuit, the statement S2 will be executed in a faulty circuit. Therefore, statement S1 is the good clause and a good statement; statement S2 is the bad clause and a bad statement.

Case 1. There are matching destination object assignment statements existing under both the good clause and the bad clause.

Propagation procedure:

Given a good/bad value pair in a control expression:

[1]. Choose a good statement in the good clause and a bad statement in a bad clause.

[2]. Choose a good/bad value pair by using the good/bad value selection rules.

[3]. Check if the destination object of the bad statement already contains a value which is equal to the bad value. If yes, go to [6].

[4]. Justify the source expression of the bad statement such that the destination object of the bad statement will have the bad value if the bad statement is executed.

[5]. Do not alter the values in the control expression that received the good/bad value pair; set up other input requirements such that the bad statement will be executed if the the circuit is faulty.

[6]. Justify the source expression of the good statement such that the destination object of the good statement will have the good value if the good statement is executed.

[7]. Do not alter the values in the control expression that received the good/bad value pair; set up other input requirements such that the good statement will be executed if the circuit is fault free.

[8]. Record the destination object as a member of the fault propagation path.

[9]. If the destination object is not an output pin, continue the propagation of the good/bad value pair.

Case 2. There is an assignment statement existing under the good clause, but no matching destination object assignment statement exists under the bad clause.

Propagation procedure:

Given a good/bad value pair in a control expression:

[1]. Choose a good statement in the good clause.

[2]. Choose a good/bad value pair by using the good/bad value selection rules.

[3]. Check if the destination object of the good statement already contains a value which is equal to the bad value. If yes, go to [5].

[4]. Justify the bad value to the destination object of the good statement. Execute the statement which is used in this justification. If the good statement is used for this justification, propagate the bad value to an observable output. (This is done because if the circuit is faulty, the good statement will not be executed, so, whether the bad value is successfully justified to the destination object is unknown).

[5]. Justify the source expression of the good statement such that the destination object of the good statement will have the good value if the good statement is executed.

[6]. Do not alter the values in the control expression that received the good/bad value pair; set up other input requirements such that the good statement will be executed if the circuit is fault free.

[7]. Record the destination object as a member of the fault propagation path.

[8]. If the destination object is not an output pin, continue the propagation of the good/bad value pair.

Case 3. There exists an assignment statement under the bad clause, but no assignment statement exists under the good clause.

Propagation procedure:

Given a good/bad value pair in a control expression:

[1]. Choose a bad statement in the bad clause.

[2]. Choose a good/bad value pair by using the good/bad value selection rules.

[3]. Check if the destination object of the bad statement already contains a value which is equal to the good value. If yes, go to [5].

[4]. Justify the good value to the destination object of the bad statement by using an assignment statement other than the bad statement. (If no other matching destination object assignment statement exists, the fault cannot be propagated.) Execute the statement which is used in this justification.

[5]. Justify values in the source expression of the bad statement such that the destination object of the bad statement will have the bad value if the bad statement is executed.

[6]. Do not alter the values in the control expression that received the good/bad value pair; set up other input requirements such that the bad statement will be executed if the circuit is faulty.

[7]. Record the destination object as a member of the fault propagation path.

[8]. If the destination object is not an output pin, continue the propagation of the good/bad value pair.

5.2.5 Justification

Given a source expression, a statement number and a desired value at a given time, the justification goal will justify the appropriate values in the objects of the source expression, such that the desired value can be obtained. If the objects in the source expression already contain some values and these values are appropriate for this justification, then new values will not be assigned to those objects, otherwise, new values will be assigned to those objects.

For example:

```
S1:  A < = obj1 and obj2;
S2:  B < = obj3 and obj4;
S3:  C < = A and B;
```

Using the justification goal to assign a logic '1' to 'C', the source expression of the statement S3 and the desired value of logic "1" must be given. The justification goal will first check the values in objects 'A' and 'B'. If 'A' and 'B' contain some values, the source expression 'A and B' will be evaluated and the result compared with '1'. If the result is equal to '1', the justification goal is a success. Otherwise, the test program will find out the appropriate values for the 'A' and the 'B', then justify the appropriate value to 'A' through assignment statement S1, and the 'B' through assignment statement S2.

Justification procedure:

[1]. Evaluate the source expression with the existing values in the objects of the source expression.

[2]. Compare the result of the evaluation and the desired value. If the desired value is equal to the result of the evaluation, the justification is a success; else go to [3].

[3]. If the source expression is a single object, assign the desired value to the object. If the justification is for fault propagation purposes, record the assignment statement which is used to assign this desired value (forming a justification path).

[4]. If the source expression is an unary or binary micro-operation, find out the appropriate values for each object in the source expression such that the desired value can be justified. Use [3] to justify the appropriate value to each object.

During fault propagation, a fault propagation path is recorded. During justification for fault propagation, the assignment statements which are used for the justification are recorded. In the case where a 2nd path propagation is required, this 2nd path must be the justification path.

Procedure for assigning values to non-input objects:

Given an object and a value that is desired for the object at given time.

[1]. Get an assignment statement which has this object as its destination object.

[2]. Get the objects of the source expression of this assignment statement.

[3]. Partition the objects of this source expression into to two sets. Set-A contains the objects which are members of the fault propagation path. Set-B contains the objects which are not the members of the fault propagation path.

[4]. Justify the objects in Set-B and attempt to get the desired value. If the desired value is obtained, justification is a success; else go to [5].

[5]. Record the object at the fanout point, use the 2nd path propagation rule to propagate the fault through the 2nd path.

Procedure for 2nd path propagation:

[1]. Get the assignment statement that was most recently used for the justification.

[2]. Get the good/bad value pair of the object at the fanout point.

[3]. Propagate the good/bad value pair through the assignment statement.

[4]. Continue to propagate the good/bad value pair through the path which was used for the justification towards the reconvergent point.

5.2.6 Execution

Given a statement, and a given time, the execution goal will set up all the conditions in the control expression (if any), so that the statement can be executed. Also, it will record the other statements (if any) that will be executed as the result of setting up the control expression [21]. If the statement being executed is an assignment statement, the controllability of the objects in this assignment statement is determined. The controllability of an object is the ability of the object to accept new a value. If an object is marked as not being able to accept a new value, a new value will not be assigned to that object during a test generation process. The following rules are used to determine if an object should be assigned a new value.

Rules for determining the controllability of the objects:

Rule 1. If the control expression of the assignment statement is edge-responsive (ex. clk = 'R' or clk = 'F'), the objects in the source expression can be reassigned new values after the statement is executed; the destination object of the statement cannot be reassigned new values.

Rule 2. If the control expression of the assignment statement is not edge-responsive or no control expression governs the assignment statement, all the objects in the assignment statement cannot be reassigned new values after the statement is executed.

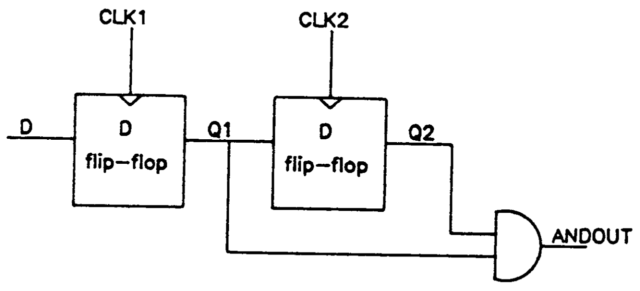
Execution procedure:

[1]. Check if there is a control statement governing the statement being executed. If yes, go to [2]; else go to [3].

[2]. Obtain the control expression requirements from the data base. Set up the control requirements and record the other statements (if any) that will also be executed as the result of setting up the control requirements.

[3]. If the statement being executed is an assignment statement, determine the controllability of the objects in the assignment statement.

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

     architecture BEHAVIOR of CKTA is

        signal Q1, Q2 : BIT;

     process(CLK1)
     begin
1:      if (CLK1='1' and not(CLK1'stable)) then
2:         Q1 < = D;
     end process;

     process(CLK2)
     begin
3:      if (CLK2='1' and not(CLK1'stable)) then
4:         Q2 < = Q1;
     end process;

5:      ANDOUT < = Q1 AND Q2;

     end BEHAVIOR;
```



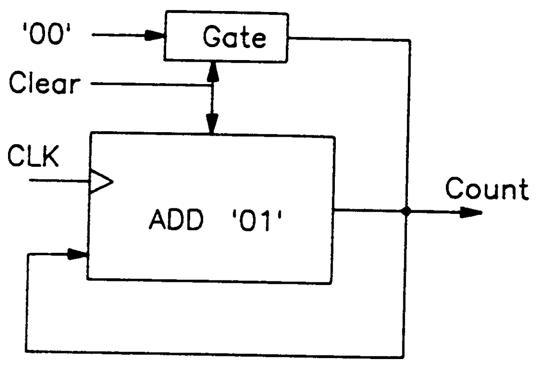
Test vector set for stuck-then fault on statement s1.

[1, stuckthen, s1]

t\obj	d	clk1	clk2	andout
t0+0	0	R	x	x
t0+1	1	F	x	x
t0+2	1	0	R	0/1

Figure 15. Example of test generation on a reconvergent fanout circuit

```
  entity COUNTER(
       CLK,CLEAR : in BIT;
       COUNT : out BIT2_VECTOR) is
  end COUNTER;


  architecture ARCH of COUNTER is

s0:  process (CLEAR)
    begin
s1:    if CLEAR = '1' then
s2:       COUNT < = "00";
      endif;

    end process CLEAR_CTR;

s3:  COUNT_UP:
    process (CLK)
    begin
s4:    if (CLK = '1' AND NOT CLK'STABLE AND CLEAR = '0') then
s5:           COUNT < = ADD(COUNT,"01");
      end if;
    end process COUNT_UP;

end ARCH;
```



Test vector set for stuck-else fault on statement S1.

[2, stuckelse, s1]

t\obj	clk	clear	count
t0+0	x	1	00/XX
t0+1	R	0	x
t0+2	1	1	00/01

Figure 16. Example of test generation on a circuit with looping

5.3 Test generation examples

In this section, two test generation examples are presented to explain the test generation processes for a reconvergent fanout circuit and for a looping circuit.

The Figure 15 on page 75 shows the VHDL behavioral description, the circuit diagram, and a test vector sequence for the testing of a stuck-then fault on statement S1 in a reconvergent fanout circuit. To test the stuck-then fault, the test program will first record a list of good statements, a list of bad statements, and the faulty statement. In this case, there is no good statement, statement S2 is the bad statement, and statement S1 is the faulty statement. Next the fault sensitization procedure for a stuck-then fault (case II) is used to sensitize the fault. The test program chooses the bad statement S2 and the good/bad value pair of 0/1. It justifies the good value of '0' to Q1 by assigning '0' to D and setting up the control condition of "clk1='R'". It justifies the bad value of '1' to Q1 by assigning a '1' to D and setting up the control condition of "clk1='F'" such that if the circuit is faulty, the '1' will go to Q1. After the fault is sensitized, the good/bad value pair of 0/1 on Q1 is propagated towards an output. In order to propagate the Q1 through the AND gate, Q2 should be '1'. The test program first tries to justify a '1' on Q2. To do this, it has to justify a '1' on Q1. However, Q1 contains a good/bad value pair of 0/1, hence, Q1 cannot be justified to '1'. Since Q1 is on the fault propagation path and contains a good/bad value pair of 0/1, the test program initiates the 2nd path propagation, and then propagates the fault from Q1 to Q2 by setting up a control condition of "clk2 = 'R'". Therefore, both inputs of the AND gate have the good/bad value pair of 0/1, and the good/bad value pair can be propagated to the output ANDOUT. This test generation process is fully automatic after providing a name of the circuit model and a fault number from the fault list.

The Figure 16 on page 76 shows a VHDL behavioral description, a circuit diagram, and a test vector sequence for the testing of a stuck-else fault on statement S1. This circuit is a simple counter, which includes a clear function and a count function. As the circuit diagram shows, there is a looping situation in the count function. The testing of faults in this circuit requires initialization of the circuit. In this case, a user has to provide the statement number of S2, and the desired initial state of "00", then, the test program will automatically generate the first test vector in the Figure 16 on page 76. After the circuit is initialized, the test program will automatically start the fault sensitization process by using the fault sensitization procedure for stuck-else fault (case III). First, the assignment statement S2 under the THEN and the assignment statement S5 outside the THEN are found. A good/bad value pair of "00/01" is chosen automatically by the program. Then the assignment statement S5 is used to justify the bad value of "01" and the control condition of " clk='R', clear='0' " is set up such that the "01" goes to the destination object COUNT. Next, the assignment statement S2 is used to justify the good value of "00" and the control condition of "clear='1'" is set up such that the "00" goes to the destination object COUNT if the circuit is fault free. Since COUNT is an output pin, the good/bad value pair of "00/01" does not require to propagate further.

Chapter 6. Results

The improved test generation algorithm was tested on five circuit models. All of these circuits were sequential circuits, some with reconvergent fanout, feed-back and/or looping. As previously stated, this test generation algorithm can handle reconvergent fanout automatically. For circuits with feed-back and/or looping, user assistance is required to identify a VHDL statement, which is used for the circuit initialization, and a desired state. The test program will then automatically determine the appropriate input vectors to drive the circuit to the desired state. A user can also choose a good/bad value pair and choose a fault propagation path during test generation. This user intervention is not necessary but improves test generation speed.

In the improved test generator, dead-process faults are removed from fault lists. This is because a dead-process fault can be tested by testing an assignment control fault on one of the assignment statements within the process. The fault lists for the improved test generator are shorter.

Due to the configuration of some circuits, some faults in circuits cannot be tested. The following are the two situations for which a test cannot be generated.

[1]. Faults that could not find a path to propagate towards an observable output.

[2]. During fault propagation, traversal of a fault site cannot be avoided and the fault sensitization is destroyed due to the traversal of the fault site.

The next several sections contain discussion of the test generation results for the circuits. For the VHDL behavioral descriptions and the fault lists, please refer to appendix B.

Table 2. Test generation result for "CKA"

	Improved method	Jani's method
Number of behavioral faults	10	12
Number of test generated automatically	10	12
Number of collapsed test sequences	6	6
Number of correct test sequences	6	3
Number of successful tests	10	6
Total test generation CPU time	209.47 sec	525.42 sec

6.1 "CKA"

The circuit model "CKA" is a sequential circuit consisting of two D flip-flops with separate directly controllable clocks, connected in series. The output of the first D flip-flop is fanned out to the input of the second D flip-flop and an input of an AND gate. The other input of the AND gate is the output of the second D flip-flop. Hence, a fanout has reconverged at the AND gate. The improved algorithm generated tests for all the behavioral faults in this circuit. The test generation results are summarized in Table 2.

Table 3. Test generation result for the 8-bit register

	Improved method	Jani's method
Number of behavioral faults	12	15
Number of test generated automatically	12	15
Number of collapsed test sequences	10	10
Number of correct test sequences	10	10
Number of successful tests	12	15
Total test generation CPU time	229.31 sec	288.93 sec

6.2 8-bit register

This 8-bit register has a STRB control which strobes an input into the register. An output buffer is controlled by an ENABLE signal. When the ENABLE signal is '1', the data in the register goes to an output DO; when the ENABLE signal is '0', the output DO is "11111111". The ENABLE signal is '1' only when the input signal DS1 is '1' and NDS2 is '0'. There is no reconvergent fanout, feed-back, or looping in this circuit. The improved algorithm generated tests for all the behavioral faults in the circuit, and a test sequence for each fault is shorter compared to a test sequence that was generated by the previous algorithm (Jani's method). The test generation results are shown in Table 3.

Table 4. Test generation result for the Intel's 8212 buffered latch

	Improved method	Jani's method
Number of behavioral faults	37	44
Number of test generated automatically	37	39
Number of collapsed test sequences	35	34
Number of correct test sequences	35	21
Number of successful tests	37	22
Total test generation CPU time	1078.31 sec	1939.18 sec

6.3 *Intel's 8212 buffered latch*

This circuit is an 8-bit buffered latch that has five control inputs (NDS1, DS2, MD, STB, NCLR) controlling the device selection, data latching, data output, and service request functions. The device is selected when the NDS1 is '0' and the DS2 is '1'. This buffered latch is in the output mode when the MD is '1', and is in the input mode when the MD is '0'. The service request flip-flop is set when the device is selected or the NCLR is '1'. The 8-bit input data is latched in when the device is selected and the MD is '1' (output mode), or when the STB is '1' and the MD is '0' (input mode). The output buffer is enabled when MD is '1', or MD is '0' and the device is selected. The improved algorithm generated tests for all 37 behavioral faults in this circuit, and the average length of a test sequence is 3.6 test vectors which is much shorter than the average length of a test sequence of 6.8 test vectors from the previous algorithm (Jani's method). The results of the test generation are shown in Table 4.

Table 5. Test generation result for the simple counter

Number of behavioral faults	12
Number of collapsed test sequences	9
Number of correct test sequences	9
Number of successful tests	11
Total test generation CPU time	140.25 sec

6.4 Simple counter

This circuit is a 2-bit counter with an asynchronous clear function. The counter counts up 1 when the CLOCK rise and the CLEAR signal is '0'. When the CLEAR signal is '1', the counter output is "00". In the testing of this circuit, the present state of the counter must be known, otherwise the test program will not be able to drive the counter to a particular state. The state of the counter must be initialized by using the asynchronous clear function during the initialization step. For this circuit, a user has to input an assignment statement number (S2), and a desired state of "00" during the initialization step. The test program will automatically find the appropriate input vector that will drive the circuit to the state of "00". The test generation process begins after the initialization step has completed. This test program has generated tests for 11 out of 12 of the behavioral faults in this circuit. The one behavioral fault that could not be tested was because traversal of fault site during fault propagation destroyed the fault sensitization. The test generation results are summarized in Table 5.

6.5 Controlled counter

This is a two-bit up/down counter with load limit and asynchronous clear functions. The control section of the counter has a 2 bit register. The STRB signal loads the register with the 2-bit CON input. The output of the register is decoded into a 4-bit signal "CONSIG". The truth table of this decoder is shown in Table 6 on page 88. The functional operations of the counter are summarized in Table 7 on page 88.

The output signal COUNT is fed back to compare with the loaded limit in the limit register, and the result of the comparison becomes a control signal for the count-up and the count-down functions. This situation is similar to the one that was discussed in section 4.7 of chapter 4.

For the count-up and count-down functions, there is a looping situation. This looping situation will put the test generation program into an infinite loop during test generation process if the current state of the circuit is unknown. This is because when the program tries to drive the circuit to a particular state, it will first find a previous state of that particular state, then compare the previous state with the current state. If the current state is equal to the previous state, the program will set up input conditions to drive to circuit from the current state to that particular state. However, if a current state of the circuit is unknown, the current state will never be equal to a previous state that is found by the program, and therefore, the program will never stop finding a previous state. This situation is discussed in section 4.7 of chapter 4.

In the testing of this circuit, user assistance in circuit initialization is required. The limit register must be loaded, and the signal COUNT must be initialized to a known state. To load the limit register, a user should input an assignment statement number s10 and the "limit" that he wants to load, such as "11". To initialize the COUNT, an user should input an assignment statement number s13 and an initial state of "00". With this information, the test program will automatically determine appropriate input vectors to accomplish the initialization. Furthermore, if a user wants to have the COUNT in a state other than the state "00", assignment statements S16 and S19 can be used to drive the circuit to other states. This can be done by inputting the statement number S16 or S19, and a desired state to the test program. Then the program will automatically find the required input vectors. After completion of the initialization step, test generation begins.

During test generation, sometimes, choosing a good/bad pair for an object manually can shorten test generation time and shorten a test vector sequence for a fault. The example in Figure 17 on page 87 shows the difference between choosing a good/bad value pair for the object COUNT automatically and manually.

For the first test sequence, the good/bad value pair of "11/00" for the object COUNT is chosen manually during test generation process. The good/bad value pair of "01/00" for the object COUNT in the second test sequence is chosen automatically by the test program. In this case, choosing a good/bad value pair manually can shorten the test generation time and the test sequence. In the first test sequence, the circuit was initialized to the state of "00" by the 1st test vector and the limit register was loaded with "01" by the 2nd and the 3rd test vectors; the 4th and the 5th test vectors are used to drive the object COUNT from the state of "00" to "11". Since the destination object COUNT changed from "00" to "11", the assignment control fault on statement S19 is tested. In the second test sequence, the first three test vectors are used

```
Controlled Counter (2)
[41, assncntl, s19]

t\obj clk strb con  data  count
t0+0  x   R    00   x     00/XX
t0+1  x   R    01   x     x
t0+2  x   F    01   01    x
t0+3  x   R    11   01    x
t0+4  R   1    11   01    11/00

Test generation CPU time: 34.17 sec.



Controlled Counter (2)
[41, assncntl, s19]

t\obj clk strb con  data  count
t0+0  x   R    00   x     00/XX
t0+1  x   R    01   x     x
t0+2  x   F    01   01    x
t0+3  x   R    11   01    x
t0+4  R   1    11   01    x
t0+5  R   1    11   01    x
t0+6  R   1    11   01    01/00

Test generation CPU time: 42.6 sec.
```

Figure 17. Test vector sequences for the assignment control fault on S19

to initialize the circuit. The 4th and 5th test vectors drive the COUNT from "00" to "11". Since the chosen good value is "01", the 6th test vector drives the COUNT to "10", and the 7th test vector drives the COUNT to "01" (the chosen good-value). In this example, the counter was performing a subtract function; if the counter was performing an add function, the good/bad value pair of "01/00", which is chosen automatically by the test program, is better.

The improved algorithm has generated tests for 37 out of 42 of the behavioral faults in the circuit. The five behavioral faults that cannot be tested are the faults in the limit register. There is no propagation path that can be used to propagate these faults to an observable output. Table 8 on page 88 summarizes the test generation results.

Table 6. Truth table of the counter control decoder

CON	STRB	CONSIG
00	R	1000
01	R	0100
10	R	0010
11	R	0001

Table 7. Function table of the controlled counter

CONSIG	STRB	COUNT = LIM	FUNCTIONS
1000	X	X	clear
0100	F	X	load limit
0010	R	count /= lim	count up
0001	R	count /= lim	count down

Table 8. Test generation result for the controlled counter

Number of behavioral faults	42
Number of collapsed test sequences	22
Number of correct test sequences	22
Number of successful tests	37
Total test generation CPU time	1556.88 sec

Chapter 7. Future development

7.1 Fully automated test generation

Currently, test generation for circuits with feed-back and/or looping requires user assistance in the initialization step. This initialization step could be made automatic in a future development. This could be done by marking the object(s) that required initialization during the preprocessing step. The required initial state can be recorded in a preprocessing step as well. Then the test generation program could initialize the object(s) that required initialization before doing fault sensitization.

7.2 Intelligent selection of good/bad value pair automatically

As the example shown in Figure 17 on page 87 in chapter 6 illustrates, sometimes selecting a good/bad value pair manually can increase test generation efficiency. Currently, the test program does not consider what good/bad value pair would test the fault most efficiently. This situation can be improved by storing the state sequence(s) of a sequential circuit during a preprocessing step, so that the test program will have the knowledge of what is the previous state and what is the next state of the circuit. Then, when choosing a good/bad value pair automatically during test generation, the test program will consider the "state" information and pick the best good/bad value pair that can test the fault.

7.3 Identification of equivalent faults

Currently, a test sequence is generated for each fault listed in a fault list. Among these test sequences, many of them could be the same test sequence. This means the same test sequence can test two or more faults in a fault list. Equivalent faults could be identified during a preprocessing step. Some special cases are currently known to be equivalent faults, others will have to be identified.

7.4 Test for microprocessors

A microprocessor can be modeled with a VHDL behavioral description by using the VHDL sub-set that is described in chapter 3. The test generator should be applied to this case and other large sequential circuits.

7.5 User adjustable VHDL to PROLOG representation translator

Currently, the VHDL descriptions have to be translated to a PROLOG representations manually. It is feasible to build a translator by using PROLOG. However, in the on going development of this test generator, the PROLOG representation of VHDL may have to be adjusted. Thus, the translator should have a capability to allow users to alter the PROLOG representation of VHDL when it is necessary.

An adjustable translator can be implemented by using the idea of matching. In a translation process, one VHDL statement or one VHDL control construct will match to one PROLOG representation of the VHDL. This PROLOG representation of VHDL could be stored in a data base. If one makes this PROLOG representation variable and allows a user to request a change in this PROLOG representation, then this translator would be adjustable.

Chapter 8. Conclusion

The previous test generator has been improved to handle reconvergent fanout, feedback and looping. The test generation results also showed that the total test generation time for all the faults in a fault list was shortened. More test sequences were generated for every behavioral model with a reconvergent fanout situation. Also, in the improved algorithm, a change in value in a source expression object is not required for the execution of a concurrent assignment statement. As the result of this, more realistic and shorter test sequences were generated.

Some suggestions for future development for this test generator were discussed in chapter 7. With the continued development, this test generator should become a powerful behavioral test generation tool.

Bibliography

1. Tekla S. Perry, "Intel's Secret Is Out", IEEE Spectrum, pp. 22-28, April, 1989.
2. Wojciech Maly, "Realistic Fault Modeling for VLSI Testing", 24th ACM/IEEE Design Automation Conference, pp. 173-180, 1987.
3. Andre Ivanov, Vinod K. Agarwal, "Testability Measure - What Do They Do for ATPG?", IEEE International Test Conference, pp. 129-138, 1986.
4. Andre Ivanov, Vinod K. Agarwal, "Dynamic Testability Measure for ATPG?", IEEE Transactions on Computer Aided Design", pp. 598-608, May, 1988.
5. Parag K. Lala, "Fault Tolerant and Fault Testable Hardware Design", Prentice-Hall International Inc., London, pp. 25-68, 1985.
6. H. Fujiwara, T. Shimono, "On The Acceleration of Test Generation Algorithms", IEEE Transaction on Computers, pp. 1137-1144, December, 1983.
7. Shteingart, Nagle, Grason, "RTG: Automatic Register Level Test Generator", IEEE 22nd Design Automation Conference, pp.803-807, 1985.
8. S. Y. H. Su, Y. Hsieh, "Testing Functional Faults in Digital Systems Described by Register Transfer Language", IEEE Test Conference, pp. 447-457, 1981
9. T. S. Lin, S. Y. H. Su, "VLSI Function Test Pattern Generation - A Design and Implementation", IEEE International Test Conference, pp. 922-929, 1985.
10. T. S. Lin, S. Y. H. Su, "The S-Algorithm: A Promising Solution For Systematic Functional Test Generation", IEEE Transaction on Computer-Aided Design, pp. 250-263, July, 1985.
11. S. Y. H. Su, T. S. Lin "Functional Testing Techniques for Digital LSI/VLSI System", IEEE 21st Design Automation Conference, pp. 517-528, 1984.
12. H. P. Chang, W. A. Rogers and J. A. Abraham, "Structured Functional Level Test Generation Using Binary Decision Diagrams", IEEE International Test Conference, pp. 97-104, 1986.
13. S. B. Akers, "Binary Decision Diagrams", IEEE Transaction on Computer, pp. 509-516, June, 1978.

14. M. S. Abadir, H. K. Reghbati, "Functional Test Generation for LSI Circuits Described by Binary Decision Diagrams", IEEE International Test Conference, pp. 483-492, 1985.

15. C. W. Yan "Concurrent Test Generation Using AI Techniques", IEEE International Test Conference, pp. 722-731, 1986.

16. D. Bhattacharya, B. T. Murray, J. P. Hayes, "High-Level Test Generation For VLSI", Computer, pp. 16-24, April, 1989.

17. G. D. Robinson, "Hitest - Intelligent Test Generation", IEEE International Test Conference, pp. 311-323, 1983.

18. D. S. Barclay, J. R. Armstrong, "A heuristic Chip-Level Test Generation Algorithm", IEEE 23rd Design Automation Conference, pp. 257-261, 1986.

19. D. S. Barclay, "An Automatic Test Generation Method for Chip-Level Circuit Descriptions", Master's thesis, VPI & SU, February, 1987.

20. M. D. O'Neill, "An Improved Chip-Level Test Generation Algorithm", Master's Thesis, VPI & SU, January, 1988.

21. D. D. Jani, "A Efficient Test Generation Algorithm For Behavioral Descriptions of Digital Devices, Master's Thesis, VPI & SU, December, 1988.

22. M. D. O'Neill, D. D. Jani,C. H. CHO, J. R. Armstrong, "BTG: A Behavioral Test Generator", Computer Hardware Description Language Conference, June, 1989.

23. Y. H. Levendel, P. R. Menon, "Test Generation Algorithm for Computer Hardware Description Languages", IEEE Transactions on Computer, PP. 577-588, June, 1982.

24. F. E. Norrod, "The E-algorithm: An Automatic Test Generation Algorithm For Hardware Description Languages", Master Thesis, VPI & SU, February, 1988.

25. J. R. Armstrong, "Chip Level Modeling with VHDL", Prentice Hall Inc., August, 1988.

26. F. E. Norrod, "An Automatic Test Generation Algorithm for Hardware Description Language" 26th ACM/IEEE Design Automation Conference, pp. 427-434, June 1989.

Appendix A. How to use the test generator

A.1 Test generation preprocessor

The test generation preprocessor consist of an intermediate form extractor and an fault list extractor. The next two examples show how to run the intermediate form extractor and the fault list extractor.

Example A1. Intermediate form extractor

```
$ run prolog

Portable Prolog Release 2.1.

?- consult('ife.pro').

'mdolib.pro' consulted.
'execute.pro' consulted.
'ife.pro' consulted.
yes
```

```
?- start.

Enter model name, followed by a period: cka.

'[lam.btg.hdldir]cka.hdl' consulted.
yes

?- go.

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

Writing to file...
yes

?- end. [Leaving Prolog]
```

Example A2. Fault list extractor

```
$ run prolog

Portable Prolog Release 2.1.

?- consult('fle.pro').

'mdolib.pro' consulted.
'lookup.pro' consulted.
'bitops.pro' consulted.
'bvops.pro' consulted.
'fle.pro' consulted.
yes

?- start.

Enter model name, followed by a period: cka.

'[lam.btg.hdldir]cka.hdl' consulted.
'[lam.btg.hdldir]cka.ife' consulted.
yes
```

```
?- go.

Listing faults...
Regrouping...
Numbering faults...
Writing fault list...
Done
yes

?- end. [Leaving Prolog]
```

A.2 Automatic test generation

For circuits without feed-back and/or looping, this test generator can handle them automatically. The next two examples show how to use the test generator to generate a test automatically.

Example A3. Automatic test generation

```
$ run prolog

Portable Prolog Release 2.1.

?- consult('btg1.pro').

'mdolib.pro' consulted.
'lib.pro' consulted.
'propagate.pro' consulted.
'justify.pro' consulted.
'execute.pro' consulted.
'bitops.pro' consulted.
'bvops.pro' consulted.
'lookup.pro' consulted.
'waveform.pro' consulted.
'time.pro' consulted.
'justrules.pro' consulted.
'propagate_2.pro' consulted.
'btg1.pro' consulted.
yes
```

```
?- start.

Enter model name, followed by a period: cka.

'[lam.btg.hdldir]cka.hdl' consulted.
'[lam.btg.hdldir]cka.ife' consulted.

Model Description: CKA


Human interaction required ? n.


Would you like to turn-on the trace command? n.

yes

?- go.

Enter fault number: 1.

Fault ID: [1, stuckthen, s1]

PROPAGATE: Stmt - s1
Good val: [bit, [48]]
Bad val:  [bit, [49]]
POSITION CODE: []t1

No objects exist under the good clause
No objects exist under the good clause
Attempting propagation thru Stmt s2
JUSTIFY:
[bit, [48]]
[obj, d]
s2 t2

Value asserted - d, [bit, [48]], t2
JUSTIFY:
[bit, [49]]
[biteqv, [obj, clk1],[lit, [bit, R]]]
s1 t2

Value asserted - clk1, [bit, [82]], t2
PROPAGATE: Stmt - s2
Good val: [bit, [48]]
Bad val:  [bit, [49]]
POSITION CODE: []t1

JUSTIFY:
[bit, [49]]
```

```
[obj, d]
s2 t4

Value asserted - d, [bit, [49]], t4
JUSTIFY:
[bit, [48]]
[biteqv, [obj, clk1],[lit, [bit, R]]]
s1 t4

Value asserted - clk1, [bit, [70]], t4

asserted good_bad_val(q1[bit, [48]][bit, [49]]t4)
Attempting propagation thru Stmt [s5, [45], [76]]
PROPAGATE: Stmt - s5
Good val: [bit, [48]]
Bad val:  [bit, [49]]
POSITION CODE: [76]t1

JUSTIFY:
[bit, [49]]
[obj, q2]
s5 t5


The desired value cannot be justified due to
reconvergent fanout. The fault propagates
through a second path from the fanout
point to the convergent point.

JUSTIFY:
[bit, [49]]
[biteqv, [obj, clk2],[lit, [bit, R]]]
s3 t6

Value asserted - clk2, [bit, [82]], t6

Second path propagation ended at object q2

PROPAGATE: Stmt - s5
Good val: [bit, [48]]
Bad val:  [bit, [49]]
POSITION CODE: []t1


asserted good_bad_val(andout[bit, [48]][bit, [49]]t7)
Elapsed CPU Time:      12720    milliseconds.

Preparing waveform...

Model name: CKA
Fault id: [1, stuckthen, s1]
```

```
t\obj d   clk1 clk2 andout
t0+0  0   R    x    x
t0+1  1   F    x    x
t0+2  1   0    R    0/1


Writing .WAVE file...
Done.

Another test? n.

yes

?- end.

[Leaving Prolog]
```

A.3 User assisted test generation

For circuits with feed-back and/or looping, human assistance in circuit initialization step is required. The next two test generation examples demonstrate how to use human assistance in test generation.

Example A4. User assisted test generation

```
$ run prolog

Portable Prolog Release 2.1.

?- consult('btg1.pro').

'mdolib.pro' consulted.
'lib.pro' consulted.
'propagate.pro' consulted.
'justify.pro' consulted.
'execute.pro' consulted.
'bitops.pro' consulted.
'bvops.pro' consulted.
'lookup.pro' consulted.
'waveform.pro' consulted.
```

```
'time.pro' consulted.
'justrules.pro' consulted.
'propagate_2.pro' consulted.
'btg1.pro' consulted.
yes

?- start.

Enter model name, followed by a period: coctr1.

'[lam.btg.hdldir]coctr1.hdl' consulted.
'[lam.btg.hdldir]coctr1.ife' consulted.

Model Description: Controlled Counter (2)


Human interaction required ? y.


Would you like to turn-on the trace command? n.

yes

?- go.

Enter fault number: 1.

Fault ID: [1, stuckthen, s2]


Would you like to set the circuit to its initial state? y.

Please input the STATEMENT NUMBER of the
statement that needs to be executed for initialization. s13.

Please input the STATE that you wish to initialized to. [bv,"00"].

JUSTIFY:
[bit, [49]]
[bveq, [[bvsubv, 3], [obj, consig]], [lit, [bv, [49]]]]
s12 t2

Wish to provide values for this justification? n.

JUSTIFY:
[bv, [48, 48]]
[obj, con]
s3 t3

Value asserted - con, [bv, [48, 48]], t3
JUSTIFY:
```

```
[bit, [49]]
[biteqv, [obj, strb],[lit, [bit, R]]]
s2 t3

Value asserted - strb, [bit, [82]], t3
Would you like to initialize another signal? y.


Please input the STATEMENT NUMBER of the
statement that needs to be executed for initialization. s10.

Please input the STATE that you wish to initialized to. [bv,"11"].

JUSTIFY:
[bv, [49, 49]]
[obj, data]
s10 t5

Value asserted - data, [bv, [49, 49]], t5
JUSTIFY:
[bit, [49]]
[bitand, [biteqv, [obj, strb], [lit, [bit, [70]]]], [bveq, [[bvsubv, 2], [obj, consig]], [lit, [bv, [49]]]]]
s9 t5

Value asserted - strb, [bit, [70]], t5
Wish to provide values for this justification? n.

JUSTIFY:
[bv, [48, 49]]
[obj, con]
s3 t6

Value asserted - con, [bv, [48, 49]], t6
JUSTIFY:
[bit, [49]]
[biteqv, [obj, strb],[lit, [bit, R]]]
s2 t6

Value asserted - strb, [bit, [82]], t6
Would you like to initialize another signal? y.


Please input the STATEMENT NUMBER of the
statement that needs to be executed for initialization. s16.

Please input the STATE that you wish to initialized to. [bv,"01"].

JUSTIFY:
[bv, [48, 49]]
[bvadd, [obj, count], [lit, [bv, [48, 49]]]]
s16 t7
```

```
JUSTIFY:
[bit, [49]]
[bitand, [biteqv, [obj, clk], [lit, [bit, [82]]]],
        [bitand, [bveq, [[bvsubv, 1], [obj, consig]], [lit, [bv, [49]]]],
                [bitnot, [bveq, [obj, count], [obj, lim]]]]]
s15 t7

Value asserted - clk, [bit, [82]], t7
Wish to provide values for this justification? n.

JUSTIFY:
[bv, [49, 48]]
[obj, con]
s3 t8

Value asserted - con, [bv, [49, 48]], t8
JUSTIFY:
[bit, [49]]
[biteqv, [obj, strb],[lit, [bit, R]]]
s2 t8

Value asserted - strb, [bit, [82]], t8
Would you like to initialize another signal? n.

State initialization finished.

PROPAGATE: Stmt - s2
Good val: [bit, [48]]
Bad val:  [bit, [49]]
POSITION CODE: []t1

No objects exist under the good clause
No objects exist under the good clause
Attempting propagation thru Stmt s4
Wish to enter good_val/bad_val for consig? n.

PROPAGATE: Stmt - s4
Good val: [bv, [48, 48, 49, 48]]
Bad val:  [bv, [49, 48, 48, 48]]
POSITION CODE: []t1

JUSTIFY:
[bv, [48, 48]]
[obj, con]
s3 t12

Value asserted - con, [bv, [48, 48]], t12
JUSTIFY:
[bit, [48]]
```

```
[biteqv, [obj, strb],[lit, [bit, R]]]
s2 t12

Value asserted - strb, [bit, [70]], t12

asserted good_bad_val(consig[bv, [48, 48, 49, 48]][bv, [49, 48, 48, 48]]t12)
Wish to change observability sequence of consig? y.

Default observability sequence is [s18, s15, s12, s9]
New sequence is [s12].

Attempting propagation thru Stmt [s12, [45], [76]]
PROPAGATE: Stmt - s12
Good val: [bv, [48, 48, 49, 48]]
Bad val:  [bv, [49, 48, 48, 48]]
POSITION CODE: [76]t1

Wish to provide values for this propagation? n.

PROPAGATE: Stmt - s12
Good val: [bit, [48]]
Bad val:  [bit, [49]]
POSITION CODE: []t1

No objects exist under the good clause
No objects exist under the good clause
Attempting propagation thru Stmt s13
Wish to enter good_val/bad_val for count? n.

PROPAGATE: Stmt - s13
Good val: [bv, [48, 49]]
Bad val:  [bv, [48, 48]]
POSITION CODE: []t1


asserted good_bad_val(count[bv, [48, 49]][bv, [48, 48]]t17)
Elapsed CPU Time:     49960    milliseconds.

Preparing waveform...

Model name: Controlled Counter (2)
Fault id: [1, stuckthen, s2]

t\obj clk strb con  data  count
t0+0  x   R    00   x     00/XX
t0+1  x   R    01   x     x
t0+2  x   F    01   11    x
t0+3  x   R    10   11    x
t0+4  R   1    10   11    01/XX
t0+5  1   F    00   11    01/00
```

```
Writing .WAVE file...
Done.

Another test? y.

'[lam.btg.hdldir]coctr1.hdl' reconsulted.
yes

?- go.

Enter fault number: 31.

Fault ID: [31, microop, [s16, [45], []], bvadd, bvsub]


Would you like to set the circuit to its initial state? y.

Please input the STATEMENT NUMBER of the
statement that needs to be executed for initialization. s13.

Please input the STATE that you wish to initialized to. [bv,"00"].

JUSTIFY:
[bit, [49]]
[bveq, [[bvsubv, 3], [obj, consig]], [lit, [bv, [49]]]]
s12 t2

Wish to provide values for this justification? n.

JUSTIFY:
[bv, [48, 48]]
[obj, con]
s3 t3

Value asserted - con, [bv, [48, 48]], t3
JUSTIFY:
[bit, [49]]
[biteqv, [obj, strb],[lit, [bit, R]]]
s2 t3

Value asserted - strb, [bit, [82]], t3
Would you like to initialize another signal? y.


Please input the STATEMENT NUMBER of the
statement that needs to be executed for initialization. s10.

Please input the STATE that you wish to initialized to. [bv,"11"].
```

```
JUSTIFY:
[bv, [49, 49]]
[obj, data]
s10 t5

Value asserted - data, [bv, [49, 49]], t5
JUSTIFY:
[bit, [49]]
[bitand, [biteqv, [obj, strb], [lit, [bit, [70]]]], [bveq, [[bvsubv, 2], [obj, consig]], [lit, [bv, [49]]]]]
s9 t5

Value asserted - strb, [bit, [70]], t5
Wish to provide values for this justification? n.

JUSTIFY:
[bv, [48, 49]]
[obj, con]
s3 t6

Value asserted - con, [bv, [48, 49]], t6
JUSTIFY:
[bit, [49]]
[biteqv, [obj, strb],[lit, [bit, R]]]
s2 t6

Value asserted - strb, [bit, [82]], t6
Would you like to initialize another signal? n.

State initialization finished.

JUSTIFY:
[bv, [48, 48]]
[obj, count]
s16 t7

JUSTIFY:
[bit, [49]]
[bitand, [biteqv, [obj, clk], [lit, [bit, [82]]]],
        [bitand, [bveq, [[bvsubv, 1], [obj, consig]], [lit, [bv, [49]]]],
                [bitnot, [bveq, [obj, count], [obj, lim]]]]]
s15 t7

Value asserted - clk, [bit, [82]], t7
Wish to provide values for this justification? n.

JUSTIFY:
[bv, [49, 48]]
[obj, con]
s3 t8
```

```
Value asserted - con, [bv, [49, 48]], t8
JUSTIFY:
[bit, [49]]
[biteqv, [obj, strb],[lit, [bit, R]]]
s2 t8

Value asserted - strb, [bit, [82]], t8
PROPAGATE: Stmt - s16
Good val: [bv, [48, 49]]
Bad val:  [bv, [49, 49]]
POSITION CODE: []t1


asserted good_bad_val(count[bv, [48, 49]][bv, [49, 49]]t7)
Elapsed CPU Time:     31420    milliseconds.

Preparing waveform...

Model name: Controlled Counter (2)
Fault id: [31, microop, [s16, [45], []], bvadd, bvsub]

t\obj clk strb con  data  count
t0+0  x   R    00   x     00/XX
t0+1  x   R    01   x     x
t0+2  x   F    01   11    x
t0+3  x   R    10   11    x
t0+4  R   1    10   11    01/11


Writing .WAVE file...
Done.

Another test? n.

yes

?- end.

[Leaving Prolog]
```

Appendix B. Circuit models and fault lists

VHDL behavioral description of CKA

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

    architecture BEHAVIOR of CKTA is

       signal Q1, Q2 : BIT;

    process(CLK1)
    begin
1:     if (CLK1='1' and not(CLK1'stable)) then
2:        Q1 <= D;
    end process;

    process(CLK2)
    begin
3:     if (CLK2='1' and not(CLK1'stable)) then
4:        Q2 <= Q1;
    end process;

5:     ANDOUT <= Q1 AND Q2;

    end BEHAVIOR;
```

PROLOG representation of CKA

```
fileprefix("cka").
modelname("CKA").

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

statementtype(s1,if).
controlexpression(s1,[biteqv,[obj,clk1],[lit,[bit,"R"]]]).
subordinaterange(s1,then,[s2]).
subordinaterange(s1,else,[]).

statementtype(s2,assignment).
sourceexpression(s2,[obj,d]).
destinationobject(s2,q1).
edge_response(s2,true).

statementtype(s3,if).
controlexpression(s3,[biteqv,[obj,clk2],[lit,[bit,"R"]]]).
subordinaterange(s3,then,[s4]).
subordinaterange(s3,else,[]).

statementtype(s4,assignment).
sourceexpression(s4,[obj,q1]).
destinationobject(s4,q2).
edge_response(s4,true).

statementtype(s5,assignment).
sourceexpression(s5,[bitand,[obj,q1],[obj,q2]]).
destinationobject(s5,andout).

?- end.
```

Fault list of CKA

```
"CKA".
[1, stuckthen, s1].
[2, stuckelse, s1].
[3, microop, [s1, [45], []], biteqv, bitxor].
[4, assncntl, s2].
[5, stuckthen, s3].
[6, stuckelse, s3].
[7, microop, [s3, [45], []], biteqv, bitxor].
[8, assncntl, s4].
[9, assncntl, s5].
[10, microop, [s5, [45], []], bitand, bitor].
?- end.
```

VHDL behavioral description of an 8-bit register

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

     architecture ARCH of REGISTER is

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

     begin
1:   process(STRB)
     begin
2:      if (STRB ='1'and not(STRB'stable)) then
3:         DID < = DI;
     end process;

5:      ENBLD < = DS1 and not NDS2;

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

     end ARCH;
```

PROLOG representation of the 8-bit register

```
fileprefix("Rg8").
modelname("Register").

datatype(di,bv). bvlength(di,8). inputpin(di).
datatype(strb,bit).              inputpin(strb).
datatype(ds1,bit).               inputpin(ds1).
datatype(nds2,bit).              inputpin(nds2).
datatype(did,bv). bvlength(did,8).
datatype(enbld,bit).
datatype(do,bv). bvlength(do,8). outputpin(do).

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

statementtype(s3,assignment).
  sourceexpression(s3,[obj,di]).
  destinationobject(s3,did).

statementtype(s5,assignment).
  sourceexpression(s5,[bitand,[obj,ds1],[bitnot,[obj,nds2]]]).
  destinationobject(s5,enbld).

statementtype(s7,if).
  controlexpression(s7,[biteqv,[obj,enbld],[lit,[bit,"1"]]]).
  subordinaterange(s7,then,[s8]).
  subordinaterange(s7,else,[s9]).

statementtype(s8,assignment).
  sourceexpression(s8,[obj,did]).
  destinationobject(s8,do).

statementtype(s9,assignment).
  sourceexpression(s9,[lit,[bv,"11111111"]]).
  destinationobject(s9,do).

?- end.
```

Fault list of the 8-bit register

```
"Register".
[1, stuckthen, s2].
[2, stuckelse, s2].
[3, microop, [s2, [45], []], biteqv, bitxor].
[4, assncntl, s3].
[5, assncntl, s5].
[6, microop, [s5, [45], []], bitand, bitor].
[7, microop, [s5, [45], [82]], bitnot, bitbuf].
[8, stuckthen, s7].
[9, stuckelse, s7].
[10, microop, [s7, [45], []], biteqv, bitxor].
[11, assncntl, s8].
[12, assncntl, s9].
?- end.
```

VHDL behavioral description of an Intel's buffered latch

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

        architecture BEHAVIOR of I8212 is

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

        begin
    1:  process(NCLR,S1)
        begin
    2:     if (NCLR ='1') then
    3:        Q <= '00000000';
    4:     else if (S1 = '1') then
    5:            Q <= DI;
           endif;
        end process;

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

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

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

        end BEHAVIOR;
```

PROLOG representation of an Intel's buffered latch

```
fileprefix("I8212NR").
modelname("I8212NR").

datatype(di,bv). bvlength(di,8). inputpin(di).
datatype(nds1,bit).              inputpin(nds1).
datatype(ds2,bit).               inputpin(ds2).
datatype(md,bit).                inputpin(md).
datatype(stb,bit).               inputpin(stb).
datatype(nclr,bit).              inputpin(nclr).
datatype(q,bv). bvlength(q,8).
datatype(s0,bit).
datatype(s1,bit).
datatype(s2,bit).
datatype(s3,bit).
datatype(srq,bit).
datatype(do,bv). bvlength(do,8). outputpin(do).
datatype(nint,bit).              outputpin(nint).

statementtype(s2,if).
controlexpression(s2,[biteqv,[obj,nclr],[lit,[bit,"1"]]]).
subordinaterange(s2,then,[s3]).
subordinaterange(s2,else,[s4]).

statementtype(s3,assignment).
sourceexpression(s3,[lit, [bv, "00000000"]]).
destinationobject(s3,q).

statementtype(s4,if).
controlexpression(s4,[biteqv,[obj,s1],[lit,[bit,"1"]]]).
subordinaterange(s4,then,[s5]).
subordinaterange(s4,else,[]).

statementtype(s5,assignment).
sourceexpression(s5,[obj,di]).
destinationobject(s5,q).

statementtype(s7,if).
controlexpression(s7,[biteqv,[obj,s3],[lit,[bit,"1"]]]).
subordinaterange(s7,then,[s8]).
subordinaterange(s7,else,[s9]).

statementtype(s8,assignment).
sourceexpression(s8,[obj,q]).
destinationobject(s8,do).

statementtype(s9,assignment).
sourceexpression(s9,[lit, [bv, "11111111"]]).
destinationobject(s9,do).
```

```
statementtype(s11,if).
controlexpression(s11,[biteqv,[obj,s2],[lit,[bit,"0"]]]).
subordinaterange(s11,then,[s12]).
subordinaterange(s11,else,[s13]).

statementtype(s12,assignment).
sourceexpression(s12,[lit, [bit, "1"]]).
destinationobject(s12,srq).

statementtype(s13,if).
controlexpression(s13,[biteqv,[obj,stb],[lit,[bit,"R"]]]).
subordinaterange(s13,then,[s14]).
subordinaterange(s13,else,[]).

statementtype(s14,assignment).
sourceexpression(s14,[lit, [bit,"0"]]).
destinationobject(s14,srq).

statementtype(s15,assignment).
sourceexpression(s15,[bitand,[bitnot,[obj,nds1]],[obj,ds2]]).
destinationobject(s15,s0).

statementtype(s16,assignment).
sourceexpression(s16,[bitor,[bitand,[obj,s0],[obj,md]],
                     [bitand,[obj,stb],[bitnot,[obj,md]]]]).
destinationobject(s16,s1).

statementtype(s17,assignment).
sourceexpression(s17,[bitand,[bitnot,[obj,s0]],[obj,nclr]]).
destinationobject(s17,s2).

statementtype(s18,assignment).
sourceexpression(s18,[bitor,[obj,s0],[obj,md]]).
destinationobject(s18,s3).

statementtype(s19,assignment).
sourceexpression(s19,[bitand,[bitnot,[obj,s0]],[obj,srq]]).
destinationobject(s19,nint).

?- end.
```

Fault list of the Intel's buffered latch

```
"I8212NR".
[1, stuckthen, s2].
[2, stuckelse, s2].
[3, microop, [s2, [45], []], biteqv, bitxor].
[4, assncntl, s3].
[5, stuckthen, s4].
[6, stuckelse, s4].
[7, microop, [s4, [45], []], biteqv, bitxor].
[8, assncntl, s5].
[9, stuckthen, s7].
[10, stuckelse, s7].
[11, microop, [s7, [45], []], biteqv, bitxor].
[12, assncntl, s8].
[13, assncntl, s9].
[14, stuckthen, s11].
[15, stuckelse, s11].
[16, microop, [s11, [45], []], biteqv, bitxor].
[17, assncntl, s12].
[18, stuckthen, s13].
[19, stuckelse, s13].
[20, microop, [s13, [45], []], biteqv, bitxor].
[21, assncntl, s14].
[22, assncntl, s15].
[23, microop, [s15, [45], []], bitand, bitor].
[24, microop, [s15, [45], [76]], bitnot, bitbuf].
[25, assncntl, s16].
[26, microop, [s16, [45], []], bitor, bitand].
[27, microop, [s16, [45], [76]], bitand, bitor].
[28, microop, [s16, [45], [82]], bitand, bitor].
[29, microop, [s16, [45], [82, 82]], bitnot, bitbuf].
[30, assncntl, s17].
[31, microop, [s17, [45], []], bitand, bitor].
[32, microop, [s17, [45], [76]], bitnot, bitbuf].
[33, assncntl, s18].
[34, microop, [s18, [45], []], bitor, bitand].
[35, assncntl, s19].
[36, microop, [s19, [45], []], bitand, bitor].
[37, microop, [s19, [45], [76]], bitnot, bitbuf].
?- end.
```

VHDL behavioral description of a simple counter

```
  entity CONUTER(
         CLK,CLEAR : in BIT;
         COUNT : out BIT2_VECTOR) is
  end COUNTER;


  architecture ARCH of CONUTER is

s0:  process (CLEAR)
     begin
s1:     if CLEAR = '1' then
s2:        COUNT < = "00";
       endif;

     end process CLEAR_CTR;

s3:  COUNT_UP:
     process (CLK)
     begin
s4:     if (CLK = '1' AND NOT CLK'STABLE AND CLEAR = '0') then
s5:              COUNT < = ADD(COUNT,"01");
       end if;
     end process COUNT_UP;

end ARCH;
```

PROLOG representation of a simple counter

```
modelname("Counter (2)").
fileprefix("CTR").

datatype(clk,bit).                inputpin(clk).
datatype(clear,bit).              inputpin(clear).
datatype(count,bv). bvlength(count,2). outputpin(count).

statementtype(s1,if).
controlexpression(s1,[biteqv,[obj,clear],[lit,[bit,"1"]]]).
subordinaterange(s1,then,[s2]).
subordinaterange(s1,else,[]).

statementtype(s2,assignment).
destinationobject(s2,count).
sourceexpression(s2,[lit,[bv,"00"]]).

statementtype(s4,if).
controlexpression(s4,[bitand,[biteqv,[obj,clk],[lit,[bit,"R"]]],
                        [biteqv,[obj,clear],[lit,[bit,"0"]]]]).
subordinaterange(s4,then,[s5]).
subordinaterange(s4,else,[]).

statementtype(s5,assignment).
destinationobject(s5,count).
sourceexpression(s5,[bvadd,[obj,count],[lit,[bv,"01"]]]).

?- end.
```

Fault list of the simple counter

```
"Counter (2)".
[1, stuckthen, s1].
[2, stuckelse, s1].
[3, microop, [s1, [45], []], biteqv, bitxor].
[4, assncntl, s2].
[5, stuckthen, s4].
[6, stuckelse, s4].
[7, microop, [s4, [45], []], bitand, bitor].
[8, microop, [s4, [45], [76]], biteqv, bitxor].
[9, microop, [s4, [45], [82]], biteqv, bitxor].
[10, assncntl, s5].
[11, microop, [s5, [45], []], bvadd, bvsub].
[12, microop, [s5, [45], []], bvadd, bvxor].
?- end.
```

VHDL behavioral description of a controlled counter

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


   architecture ARCH of CONTROLLED_CTR is

     signal
       LIM : BIT2_VECTOR;
       CONSIG : BIT4_VECTOR;

 s01: DECODE:
     process (STRB)
     begin
 s02:    if STRB='1' and not STRB'stable  then
 s03:       case INTVAL(CON) is
             when 0 =>
 s04:            CONSIG <= "1000";
             when 1 =>
 s05:            CONSIG <= "0100";
             when 2 =>
 s06:            CONSIG <= "0010";
             when 3 =>
 s07:            CONSIG <= "0001";
           end case;
        end if;
     end process DECODE;

 s08: LOAD_LIMIT:
     process (STRB)
     begin
 s09:    if (STRB='0' and not STRB'stable and CONSIG(2)='1') then
 s10:       LIM <= DATA;
        end if;
     end process LOAD_LIMIT;

 s11: CLEAR_CTR:
     process (CONSIG(3))
     begin
 s12:    if CONSIG(3)='1' then
 s13:       COUNT <= "00";
        end if;
     end process CLEAR_CTR;

 s14: CNT_UP:
     process (CLK)
```

```
    begin
s15:   if (CLK='1' and not CLK'stable and CONSIG(1)='1'
          and  not(count = lim) then
s16:              COUNT <= ADD(COUNT,"01");
        endif;
    end process CNT_UP;

s17: CNT_DOWN:
    process (CLK)
    begin
s18:   if (CLK='1' and not CLK'stable and CONSIG(0)='1'
          and not(count = lim) then
s19:              COUNT <= SUB(COUNT,"01");
        endif;
    end process CNT_DN;

end ARCH;
```

PROLOG representation of a controlled counter

```
modelname("Controlled Counter (2)").
fileprefix("COCTR1").

datatype(clk,bit).                    inputpin(clk).
datatype(strb,bit).                   inputpin(strb).
datatype(con,bv). bvlength(con,2).    inputpin(con).
datatype(data,bv). bvlength(data,2).  inputpin(data).
datatype(count,bv). bvlength(count,2). outputpin(count).
datatype(lim,bv). bvlength(lim,2).
datatype(consig,bv). bvlength(consig,4).

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

statementtype(s3,case).
controlexpression(s3,[obj,con]).
subordinaterange(s3,[[bv,"00"]],[s4]).
subordinaterange(s3,[[bv,"01"]],[s5]).
subordinaterange(s3,[[bv,"10"]],[s6]).
subordinaterange(s3,[[bv,"11"]],[s7]).

statementtype(s4,assignment).
destinationobject(s4,consig).
sourceexpression(s4,[lit,[bv,"1000"]]).

statementtype(s5,assignment).
destinationobject(s5,consig).
sourceexpression(s5,[lit,[bv,"0100"]]).

statementtype(s6,assignment).
destinationobject(s6,consig).
sourceexpression(s6,[lit,[bv,"0010"]]).

statementtype(s7,assignment).
destinationobject(s7,consig).
sourceexpression(s7,[lit,[bv,"0001"]]).

statementtype(s9,if).
controlexpression(s9,[bitand,[biteqv,[obj,strb],[lit,[bit,"F"]]],
                  [bveq,[[bvsubv,2],[obj,consig]],[lit,[bv,"1"]]]]).
subordinaterange(s9,then,[s10]).
subordinaterange(s9,else,[]).

statementtype(s10,assignment).
destinationobject(s10,lim).
sourceexpression(s10,[obj,data]).
```

```
statementtype(s12,if).
controlexpression(s12,[bveq,[[bvsubv,3],[obj,consig]],[lit,[bv,"1"]]]).
subordinaterange(s12,then,[s13]).
subordinaterange(s12,else,[]).

statementtype(s13,assignment).
destinationobject(s13,count).
sourceexpression(s13,[lit,[bv,"00"]]).

statementtype(s15,if).
controlexpression(s15,
          [bitand,[biteqv,[obj,clk],[lit,[bit,"R"]]],
                  [bitand,[bveq,[[bvsubv,1],[obj,consig]],[lit,[bv,"1"]]],
                          [bitnot,[bveq,[obj,count],[obj,lim]]]]]]).
subordinaterange(s15,then,[s16]).
subordinaterange(s15,else,[]).

statementtype(s16,assignment).
destinationobject(s16,count).
sourceexpression(s16,[bvadd,[obj,count],[lit,[bv,"01"]]]).

statementtype(s18,if).
controlexpression(s18,
          [bitand,[biteqv,[obj,clk],[lit,[bit,"R"]]],
                  [bitand,[bveq,[[bvsubv,0],[obj,consig]],[lit,[bv,"1"]]],
                          [bitnot,[bveq,[obj,count],[obj,lim]]]]]]).
subordinaterange(s18,then,[s19]).
subordinaterange(s18,else,[]).

statementtype(s19,assignment).
destinationobject(s19,count).
sourceexpression(s19,[bvsub,[obj,count],[lit,[bv,"01"]]]).

?- end.
```

Fault list of the controlled counter

```
"Controlled Counter (2)".
[1, stuckthen, s2].
[2, stuckelse, s2].
[3, microop, [s2, [45], []], biteqv, bitxor].
[4, deadclause, s3, [[bv, [48, 48]]]].
[5, deadclause, s3, [[bv, [48, 49]]]].
[6, deadclause, s3, [[bv, [49, 48]]]].
[7, deadclause, s3, [[bv, [49, 49]]]].
[8, assncntl, s4].
[9, assncntl, s5].
[10, assncntl, s6].
[11, assncntl, s7].
[12, stuckthen, s9].
[13, stuckelse, s9].
[14, microop, [s9, [45], []], bitand, bitor].
[15, microop, [s9, [45], [76]], biteqv, bitxor].
[16, microop, [s9, [45], [82]], bveq, bvneq].
[17, assncntl, s10].
[18, stuckthen, s12].
[19, stuckelse, s12].
[20, microop, [s12, [45], []], bveq, bvneq].
[21, assncntl, s13].
[22, stuckthen, s15].
[23, stuckelse, s15].
[24, microop, [s15, [45], []], bitand, bitor].
[25, microop, [s15, [45], [76]], biteqv, bitxor].
[26, microop, [s15, [45], [82]], bitand, bitor].
[27, microop, [s15, [45], [82, 76]], bveq, bvneq].
[28, microop, [s15, [45], [82, 82]], bitnot, bitbuf].
[29, microop, [s15, [45], [82, 82, 76]], bveq, bvneq].
[30, assncntl, s16].
[31, microop, [s16, [45], []], bvadd, bvsub].
[32, microop, [s16, [45], []], bvadd, bvxor].
[33, stuckthen, s18].
[34, stuckelse, s18].
[35, microop, [s18, [45], []], bitand, bitor].
[36, microop, [s18, [45], [76]], biteqv, bitxor].
[37, microop, [s18, [45], [82]], bitand, bitor].
[38, microop, [s18, [45], [82, 76]], bveq, bvneq].
[39, microop, [s18, [45], [82, 82]], bitnot, bitbuf].
[40, microop, [s18, [45], [82, 82, 76]], bveq, bvneq].
[41, assncntl, s19].
[42, microop, [s19, [45], []], bvsub, bvadd].

?- end.
```