A Test Generation System For Behaviorally Modeled Digital Circuits

by

Wencheng Li

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering

APPROVED:

Dr. J. R. Armstrong, Chairman

Dr. O. Balci

Dr. F. G. Gray

Dr. W. R. Cyre

Dr. P. M. Athanas

June 1996

Blacksburg, Virginia

A Test Generation System For Behaviorally Modeled Digital Circuits

by

Wencheng Li

Dr. J. R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

This dissertation presents an approach to generating tests from a VHDL behavioral model. The tests can be used to thoroughly exercise the VHDL model and detect the faults in the equivalent gate level circuit of the model. The VHDL model is developed with the help of the Modeler's Assistant and represented as a Process Model Graph (PMG). A set of VHDL functions have been constructed to help develop VHDL models. Two algorithms are proposed to implement the test generation. **P-Algorithm** is used to generate tests at the process level. For each process a symbolic test set and the corresponding fixed valued test packages (FVTPs) are generated. Synthesis-related FVTP generation algorithms for the VHDL functions are derived to support the P-algorithm. **E-Algorithm** is employed to generate the entity level tests. The symbolic entity level tests are generated first and then the final fixed valued entity level tests are obtained by calculating the symbolic expressions. The Synopsys synthesis tools are used to get the equivalent gate level circuit of a VHDL model. The HILO fault grader is used to generate fault coverage. Several conversion programs have been developed to support the test evaluation.

Acknowledgments

I would like to express my greatest thanks to my advisor Dr. James. R. Armstrong for his constant support and guidance throughout this research, and the challenging environment he has made available. I would also like to thank Dr. F. G Gray, Dr. W. R. Cyre, Dr. P. M. Athanas, Dr. O. Balci, Dr. D. S. Ha, and Dr. E. A. Brown for serving as members of my committee presently or previously.

I wish to express my gratitude to my teachers and friends for their inspiration and constant support. It has been a great pleasure to work with so many hard-working and friendly people in the Virginia Tech Information Systems Center.

I would like to dedicate this work to my parents, my wife Jianping and my daughter Irene who have encouraged me at all times. My wife's and daughter's constant patience during the preparation of this dissertation has meant more to me than anything else. Their unreserved love and support are the most important reason for all my achievements.

Table of Contents

List of Illustrations

List of Tables

Chapter 1. Introduction

1.1 Motivation

With hardware description languages such as VHDL, designs can be described in either a top-down or bottom-up fashion through varying levels of abstraction. However, there is a corresponding need for an accompanying set of test generation techniques that can work throughout the abstraction hierarchy. Traditional gate level test generation methods [55, 56, 57] are time-consuming. Wealth of the rich behavioral information contained in a VHDL description cannot be used by such methods, but the behavioral information is potentially very helpful in easing the test generation problem and reducing the test generation time. Therefore, developing a behavioral level test generation method is an important research topic.

One approach to generating tests at the behavioral level is using behavioral fault models. Thatte and Abraham [11] developed functional fault models for a microprocessor based

on its instruction set and the functions performed. Various fault models are presented corresponding to different functions present in a microprocessor such as register decoding function, control function, data storage function and data manipulation function. Levendel and Menon [18] extended the D-algorithm for generating tests from computer hardware description languages (CHDLs). The fault models considered are function variables stuck at logic 0 or 1, control faults, and function faults with user-specified faulty behaviors. Cho and Armstrong [35] proposed the B-algorithm to generate tests directly from behavioral VHDL circuit descriptions. Through perturbing VHDL constructs, they defined three types of behavioral faults: behavioral stuck-at faults, behavioral stuck-open faults, and micro-operation faults. In general, behavioral level test generation methods with specific behavioral fault models do not use any gate level information. Therefore it is likely that the tests generated by such methods will have lower gate level fault coverage if the behavioral fault models cannot properly reflect the physical faults in a gate level circuit.

Another approach to developing behavioral level test generation methods is using both behavioral level information and gate level information. Behavioral information can be used to construct sensitization paths and resolve conflicts. Gate level information usually is combined into some kind of high level primitives. Abradir and Breuer [3] introduced the I-path (I for identity) which can transfer data from one place in a circuit to another place without modification. Freeman [1] weakened the conditions for the definition of an

I-path and defined the F-path (fault path) and S-path (S for stimulus) for propagating the fault effects to the primitive outputs and applying tests on the primitive inputs. Su and Kime [2] proposed the Hpath for multiple path sensitization in circuits with hierarchical descriptions. They also used a PODEM-like algorithm for local path sensitization of gate level hierarchical circuits. Lee and Patel [21, 22] incorporated gate level algorithms with high level approaches in ARTEST. Vishakantaiah, Abraham and Saab [8] introduced an approach which exploits a top-down design methodology and applies high level test knowledge to a low level test generator. Murray and Hayes [14] proposed a hierarchical test generation method in which circuits are constructed by smaller high level functional modules and the test data for such modules are stored as predifined stimulus/response packages. Kunda, Narain, Abraham and Rathi [9] introduced a methodology to speed up the test generation process for circuits with high-level primitives. Sarfert, Markgraf, Trischler and Schulz [30] also used high-level primitives in their test generation system.

Rao, Pan and Armstrong [36] introduced a system called Hierarchical Behavioral Test Generator (HBTG) to generate tests for VHDL behavioral models. Such models are represented as Process Model Graphs (PMGs) [44]. Each individual process of a PMG has a precomputed stimulus/response test set. During test generation, HBTG constructs sensitive paths first and then uses the test sets for each process to generate the final test sequence. The HBTG algorithm is based on I/O path sensitization similar to the second approach above where both behavioral and gate level information is used. It uses no fault

models but merely tries to exercise all input/output paths through the VHDL behavioral models. Kapoor and Armstrong [37] tried to generate an automatic test bench for a VHDL behavioral model. The Process Test Generator (PTG) is used to generate the stimulus/response test sets for the processes of a PMG. The test sequence generated by HBTG is converted into a test bench by the Test Bench Generator (TBG). The PTG algorithm starts by constructing a Control Flow Graph (CFG) and then assigns an event to each sensitive signal and decides both the related signals and their values to make the event controllable and observable. A VHDL simulator is needed to decide output values. Zhong [69] modified the PTG to remove the need for the VHDL simulator.

The methods introduced in [36, 37, 69] have some limitations. First, PTG [37] uses a VHDL simulator during test generation. This is time-consuming and the results can only reflect a partial data relation. The modified PTG [69] removes the need for the simulator but can only generate symbolic tests for logic operators. Next, HBTG [36] allows no symbolic substitution. The test selected from the primitive test set of a process must exactly match the symbolic value to be justified or propagated. This allows the HBTG to only generate simple symbolic tests. Finally, only symbolic tests are generated. They do not give a method for selecting specific values for the symbolic values. Therefore the tests cannot give high gate level fault coverage. The previous methods also need to be improved to generate tests for more complex models. However they do provide an

effective method to acquire test information contained in a VHDL behavioral model. Such information is used by the test generation method presented here.

1.2 Contributions

The principle objective of this dissertation is to develop a system that can generate tests directly from a VHDL behavioral description and at the same time determine the gate level fault coverage. Such tests can be used to thoroughly exercise the VHDL model (test bench) and can also be useful for detecting low level faults of the equivalent gate level circuit. The test generation approach presented here is hierarchical and behavioral, which follows the basic idea in the original HBTG and PTG [36, 37]. That is, it uses I/O path sensitization and the primitive test sets of processes. Two new algorithms are proposed in this dissertation to implement the test generation. **P-Algorithm** is used to generate tests at the process level. **E-Algorithm** is employed to generate the entity level tests. The test generation algorithms for a set of predefined functions were developed after analyzing the synthesized gate level circuits. These algorithms are synthesis-related so that they are limited to a particular synthesis tool. However, they can be developed for other synthesis tools. The test generation algorithms for these functions or other functions can also be provided by users corresponding to their synthesis tools. The following is the main features of the new algorithms. The relations to the previous methods are also indicated.

For the P-Algorithm,

1. A Control Data Flow Graph (CDFG) that incorporates the CFG [37] and DFG [69] for a process is employed to represent the data dependence and control sequence within a process.
2. A symbolic test set and fixed valued test packages are generated for each process. They can be put into a library for future use.
3. A mode is assigned to each symbolic test to reflect the properties of the test. Different kinds of tests with different modes are generated.
4. VHDL functions are defined to implement data path operators such as addition, comparison, incrementation, multiplication, and shifting. These functions act as general logic operators, which makes circuit modeling more flexible.
5. Various algorithms are developed to generate tests for the functions. Applying the tests to the synthesized gate level circuits corresponding to such functions can get high fault (stuck at faults) coverage.
6. Moreover, like high level primitives in [30], these functions contain some low level information. No gate level test generator is needed but the same effects are achieved.

For the E-Algorithm,

1. Both combinational circuits and some sequential circuits (flip-flops, counters, registers, et al.) can be handled.

2. A symbolic test is generated first based on data path construction and then many fixed value test patterns are obtained by calculation, which reduces the test generation time.
3. A backtracking mechanism is provided to handle conflicts.
4. The final test patterns give high gate level fault coverage.
5. Some methods are developed to increase the testability for a VHDL behavioral model.
6. The test patterns can be used to test both behavioral models and gate level circuits.

The P-Algorithm and E-Algorithm have been implemented on a SUN workstation using C. Some parts of PTG and HBTG [36, 37, 69] are used after modification. These include CFG and DFG construction in PTG, data structures of PTG and HBTG, and logic operator symbolic test generation in PTG, but the P-algorithm and E-algorithm programs are completely new in architecture and functionality.

The other contributions made by the dissertation include the following:

1. A GHDL sub-circuit library was created to assist HILO simulation tools.
2. A program called **tx** was developed to automatically convert the circuits from Synopsys EDIF to HILO GHDL.
3. The test sequence generated by E-Algorithm is converted to DWL format by a program called **fct** and the results of HISIM are parsed by a program called **rsd** to decide some output values.

4. A set of VHDL models were developed and the fault coverage evaluated to assess the effectiveness of the algorithms.

1.3 Contents

This dissertation is organized as follows. Chapter 2 discusses gate level, RTL level, and behavioral level test generation techniques. Chapter 3 explains the test generation system and fault coverage evaluation system. Some other supporting systems are briefly discussed. Chapter 4 describes the P-Algorithm, explains some basic concepts, and discusses test generation rules and function test generation algorithms. Chapter 5 describes the E-Algorithm in detail. Forward path and backward path generation, symbolic operation, and conflict resolution are discussed. Chapter 6 explains how to get gate level circuits from a VHDL description and convert them to HILO GHDL format, discusses the conversion from test sequence to HILO DWL format, and gives a fault coverage example. Chapter 7 discusses the fault coverage results for various models. Chapter 8 considers the limitations of P-Algorithm and E-Algorithm, proposes possible improvements, and suggests the future research areas. Chapter 9 gives conclusions regarding the algorithms, their implementations, and the test results.

There are three appendixes. Appendix A describes the various programs including **ptg**, **etg**, **fct**, **rsd**, **tx**, and **tg** and provides user's instructions. Appendix B presents the VHDL

codes for example models. Appendix C gives the equivalent gate level GHDL descriptions of the components generated by Synopsys synthesis tools.

Chapter 2. Literature Review

Test generation can be implemented at different abstraction levels of a digital circuit. In a top-down design environment, designers usually start with a behavioral description of a circuit (Behavioral Level) and then convert it into a structural description (Register-Transfer Level or RTL). If the conversion is done using a synthesis tool, such an RTL description usually acts as an intermediate format which will be further optimized into an implementable description (Gate Level or Logic Level). In the following sections, some gate level test generation methods are briefly reviewed first. Then we concentrate on the test generation techniques of register transfer level and behavioral level.

2.1 Gate Level Test Generation

When a circuit is modeled using logic gates, gate level test generation techniques can be effectively applied. These techniques can be random or deterministic. We will review some deterministic algorithms. The best-known algorithms for combinational circuits are

D-algorithm defined by Roth [55], PODEM (Path Oriented DEcision Making) proposed by Goel [56], and FAN (Fanout-Oriented Test Generation) proposed by Fujiwara and Shimono [57].

The D-algorithm is a systematic test generation algorithm for combinational circuits which will find a test for a fault if such a test exists. The test generation starts by creating a D (stuck-at 0) or $\bar{D}$ (stuck-at 1) at the fault site. Then the D or $\bar{D}$ is propagated to a primary output along the sensitized paths. The necessary internal signal values are assigned during propagation. If the propagation succeeds, the internal signal values are justified to the primary inputs. All the signal value assignments are made locally. Therefore a conflict occurs when the value assigned to a line is different from the value implied on the line by other assignments in the circuit. In order to keep the value assignments consistent, global consistency is checked from time to time. If an inconsistency is found, the algorithm backtracks to a previous decision point to search for an alternative path. Multiple path propagation may be needed if there exists fanout reconvergency. The concepts of propagation and justification used at the gate level are also employed by some higher level test generation algorithms.

The PODEM algorithm is an implicit enumeration algorithm in which all possible primary input patterns are examined as tests for a given fault based on the branch and bound algorithms. The examination of PI patterns is terminated as soon as a test is found,

i.e., a D or $\bar{D}$ has been implied on a primary output. Unlike the D-algorithm, PODEM only assigns values to PI and computes the values on all internal lines by forward implications. Therefore no conflict occurs, which eliminates the global consistency check. Backtracks are still needed when the assignments to PIs cannot make the error signal D or $\bar{D}$ propagate to POs, but, in comparison to the D-algorithm, the number of backtracks is much smaller. Experimental results presented in [56] shows that PODEM is more efficient than the D-algorithm.

The motivation of the FAN algorithm is to accelerate a test generation algorithm like PODEM by reducing the number of backtracks and shortening the processing time between backtracks. Several techniques are employed by FAN. First, in each step of the algorithm, FAN determines as many signal values as possible which can be uniquely implied. This is done by both forward and backward implications. Second, it only assigns a faulty signal value D or $\bar{D}$ which is uniquely determined or implied by the fault under consideration and leaves all the values not determined uniquely unspecified. Third, it applies a unique sensitization when the D-frontier consists of a single gate. Fourth, it stops the backtrace at a head line (a head line is a free line which is adjacent to some bound line; a line is bounded if it is reachable from some fanout point; a free line is a unbound line) and postpones the line justification for the head line to the last. Finally, multiple backtrace is used to concurrently trace more than one path. These techniques are

very useful in decreasing the number of backtracks. Thus FAN is faster and more efficient than PODEM.

Test generation is more complex for sequential circuits than for combinational circuits [59, 60] because of the presence of feedback loops and memory elements. Practical algorithms [6, 61, 62, 63] are usually obtained by modifying and extending the test generation algorithms for combinational circuits.

A synchronous sequential circuit can be modeled by a pseudo-combinational iterative array [6]. One cell of this array represents a time frame. In this transformation, a flip-flop is modeled as a combinational element having an additional input to represent its current state and an additional output to represent its next state, which becomes the current state in the next time frame. An input vector of the iterative combinational array represents an input sequence for the sequential circuit. So computations in consecutive time frames are mapped onto computations in consecutive cells of the array. Therefore a combinational test generation algorithm (e.g., D-algorithm, PODEM, FAN) can be used as long as it is modified to handle the multiple faults that arise in multiple time frames. Ma et al. [61] proposed a PODEM-based approach to sequential test generation. It uses the iterative array model and a partial State Transition Graph (STG) of the circuit for fault excitation, fault propagation and state justification. Both the fault propagation and state justification need a sequence of input vectors. The methods presented in [62, 63] are basically the

extension of D-algorithm. Finally, it should be mentioned that although there exist a number of sequential test generation algorithms, for large sequential circuits with complex cyclic structures practical solutions rely on design for testability (DFT) techniques.

2.2 Register Level Test Generation

With the increasing complexity of VLSI circuits, gate level test generation algorithms become impractical due to the enormously large search space. The alternative way is seeking test generation techniques at a higher level. RTL description of a circuit is an interconnection of higher-level components (primitives) such as registers, multiplexers, adders, multipliers, counters, etc. RTL test generation algorithms use functional knowledge contained in these components to make line justification and error propagation simple. Such algorithms can be fault-oriented or fault-independent.

Fault-independent test generation algorithms usually use certain special path construction techniques. Abradir and Breuer [3, 4] introduced the I-path to design testable VLSI chips and generate test plans. Freeman [1] weakened the conditions for the definition of an I-path and defined the F-path and S-path in his test generation method. An I-path is an identity transfer path which can transfer test data from one place in a circuit to another place without modification. Such a path is basically a subcircuit with n-bit-wide input and

output data ports together with a set of control signals such that data on the input port can be transferred unchanged to output port when the control signals are appropriately assigned. When I-paths are used for test generation, they transmit test stimuli and responses for different components and allow the components to be tested one by one without dependence on DFT hardware. I-paths have three properties: the data transformation is "one-to-one", "onto", and identity. This is quite restrictive. Removing the property "one-to-one" or "onto" , Freeman defined the F-path and S-path. An F-path is a path with the property "one-to-one" which can faithfully transmit fault effects on PI to PO. An S-path only possesses the property "onto" which guarantees any desired stimulus in an intermediate component can be provided by some appropriate pattern on PI. I-paths, F-paths, and S-paths are also useful in manual test generation for data-path logic.

Su and Kime [2] proposed the Hpath algorithm for multiple path sensitization in circuits with hierarchical descriptions. The sensitization is done by two components of Hpath, GPath and FPath. GPath is an implicit enumeration algorithm (PODEM-like algorithm) which utilizes symbolic manipulation to achieve multiple path sensitization for gate level circuits. FPath is a rule-based subsystem which sensitizes multiple paths for functional level descriptions through a sequence of rule application. Su and Kime classified a sensitized path as surjective path (same as S-path in [1]) or injective path (same as F-

path). The algorithm is applicable to circuits which can be described as semiregular and pseudocombinational.

Murray and Hayes [14] proposed a hierarchical test generation method in which circuits are constructed by smaller high level functional modules and the test data for such modules are stored as predifined stimulus/response packages (or symbolic test packages). Such packages are processed symbolically using techniques derived from artificial intelligence. The implementation of the method is based on the D-algorithm. However, test packages containing tests for module faults replace primitive D-cubes of failures. Test packages containing propagation information are used to relate module inputs and outputs during the signal propagation phase, thus playing the role of propagation D-cubes. In another paper [15], they introduced a theory of propagation for modules and circuits which can be used for hierarchical test generation and design for testability. Ambiguity sets are used to represent the propagation characteristics of a module. Algebraic operations are performed on ambiguity sets to determine the propagation characteristics of multi-module circuits.

Fault-oriented RTL test generation algorithms generally target gate level faults. Kunda, Narain, Abraham *et al* [9, 10] introduced a high level test generation algorithm. In their algorithm, a circuit is modeled as an interconnection of high level primitives. Test vector files for each module store a test vector for each fault in the module along with the faulty

module response. Such test can be generated by using a low-level test generator and a fault model of interest. Fault excitation, fault propagation, fault elimination, and line justification are used in their algorithm. A dependency directed backtracking method is implemented to speed up the algorithm.

Sarfert, Markgraf, Trischler and Schulz [30] proposed a hierarchical test generation algorithm for combinational circuits modeled with high-level primitives. Both higher level and lower level faults are targeted. They tried to model all faults at the highest possible level. A special fault transfer algorithm transfers faults from lower level to higher level. However in order to perform deterministic ATG for internal faults of high level primitives, the high level primitives are dynamically expanded to their gate level realization. High level implication procedure, unique sensitization procedure, and multiple backtrace procedure are introduced to improve the effectiveness of the algorithm.

Lee and Patel [21, 22] introduced an ATPG methodology working at an architectural level. The circuits to test are modeled as two parts: data path and control unit. Both high level and gate level algorithms are developed to handle different faults. For the data path portion, a PODEM-based branch and bound searching algorithm is used at a high level and the dependence on the gate level information is relieved. Several data types are defined for the manipulation of all possible fault effects. For the control faults, gate level

algorithms are incorporated with high level approaches to excite the fault and differentiate the fault effect to primary outputs. A functional equivalent model is used for sequential modules, which makes the methodology extendible beyond the register transfer level. Lee and Patel [24] presented a new architectural level test generation algorithm based on a nonlinear equation-solving methodology to solve conflicts. For each pattern to be justified at a high level, an instruction sequence and the under-determined system of nonlinear equations are derived. The solution of the system of equations is calculated by a signal-driven discrete relaxation algorithm. The test generation is performed by recursively assembling the instruction sequence and solving the system of equations. This approach is used in [26] to form tests for a variety of microprocessor-like circuits.

2.3 Behavioral Level Test Generation

A behavioral description of a digital circuit may be developed in a programming language like C, a hardware description language like VHDL, or a graphic language like flowcharts and dataflow diagrams. Unlike a gate level description or RTL description of a circuit, where the test information (like faults and relation between inputs and outputs) required by a test generation algorithm can be obtained directly from the structures of the primitives, the behavioral description of the circuit usually does not give such information clearly. The first step of test generation is to abstract test information from

the behavioral description. Test generation algorithms can also be fault-oriented or fault-independent.

Vishakantaiah, Abraham, and Abadir [7, 8] introduced a Automatic Test Knowledge Extraction Tool (ATKET) [7] which automatically generates pieces of test knowledge by using structural and behavioral information in the VHDL description of a design. The Module Operation Tree (MOT) is used to capture the behavior of modules in the design. The MOT is a binary tree in which the nodes correspond to VHDL conditional statements. Leaf nodes in the MOT contain a list of statements which correspond to the input-output relations defined for the module. Depending on the relation, edges in the MOT are categorized into four basic categories: Initialization Edge, Propagation Edge, Hold Edge, and State Edge. The final test knowledge generated by ATKET is also categorized into different modes (global modes) corresponding to the different edge categories. Vishakantaiah *et al* also introduced an approach [8] to generate tests by using ATKET and a gate level test generator CRIS. Two kinds of design descriptions are required. One is VHDL description for ATKET. The other is structural description for CRIS. ATKET generates global test knowledge which reflects the boundary constraints between the modules. CRIS generates gate level tests targeting the gate level faults inside of each module. Final test patterns for the design are obtained by combining the high level test knowledge and gate level tests. They gave the gate level fault coverage for some

citcuits which is below 90 percent. For axample, the fault covergae for AM2910 is 85.12%.

Thatte and Abraham [11] abstracted a graph-theoretic model (S-graph) for microprocessor architecture based on the organization and instruction set of the microprocessor. In an S-graph, the nodes are registers and a directed edge exists between two nodes if data flow occurs from one node to another node during the execution of a instruction. Various functional level fault models are presented corresponding to different functions present in a microprocessor such as register decoding function, control function, data storage function, and data manipulation function. The test generated for a functional fault is an instruction or a sequence of instructions. An example for testing an 8-bit microprocessor is presented. Approximately 2200 single stuck-at faults were simulated and the fault coverage is about 96 percent.

Levendel and Menon [18] extended the D-algorithm for generating tests from the computer hardware description languages (CHDLs). The fault models considered are function variables stuck at 0 or 1, control faults, and general function faults. The test generation algorithm consists of inserting a fault effect, propagating the fault effect through CHDL constructs to an observable point, and justifying required decisions to the primary inputs. Both nonprocedural and procedural languages are treated. Norrod [33] introduced an approach to test generation from Hardware Description Language circuit

models. It is also the extension of the D-algorithm. The fault model considers two types of faults: defects in functional operation blocks, and stuck-at faults in data or control lines.

Cho and Armstrong [35] presented the B-algorithm to generate tests directly from behavioral VHDL circuit descriptions. The behavioral VHDL model is represented as a set of equivalent process statements and connections among them. The behavioral fault model consists of three types of behavioral faults: behavioral stuck-at faults, behavioral stuck-open faults, and micro-operation faults. The B-algorithm follows three basic test generation operations: activation, propagation, and justification. Rules for the test generation operations are defined using the concepts of two-phase activation and two-phase propagation. They also discussed the difference between simulation semantics and test generation semantics. Armstrong *et al* [39 ~ 42] extensively discussed functional fault model and behavioral fault model based test generation and fault simulation.

Several fault-independent test generation methods are reported. Rao, Pan and Armstrong [36] introduced a hierarchical test generation method (HBTG) for VHDL behavioral models. The test generation algorithm is based on I/O path sensitization. It does not use a fault model, but merely tries to exercise all input/output paths through the VHDL behavioral model. It uses a precomputed stimulus/response test set for each module within the model. Kapoor and Armstrong [37] present a process test generation (PTG)

method to generate such a test set. The PTG abstracts the detailed information contained in a VHDL process (module) and constructs a Control Flow Graph (CFG). Then it tries to execute every assignment node of the CFG at least once. Zhong [69] modified the PTG so that it can automatically generate stimulus/response tests for a certain kind of processes.

Anirudhan and Menon [19] presented a symbolic test generation algorithm which uses a hierarchical model of the data path and a finite state model of the control section for a system under test. The control section is assumed to have been tested using some existing test generation method. The data path is described by means of a structural graph model which has nodes corresponding to functional modules and edges represent signal flow between modules. Each node in the structural model has a behavioral path model which is a simplified behavioral description of the model used for fault effect propagation and justification through it. There are three modes of paths: 'p' (propagation) mode, 'j' (justification) mode, and 's' (status) mode. The algorithm involves several steps such as target path activation, backward and forward propagation, updating the global control sequence, transition validation, data line justification, and generation of sequential test paths.

Chapter 3. System Overview

This chapter gives an overview of the test generation system introduced in the dissertation. The VHDL subset used by the test generation system is also presented. The Modeler's Assistant, the Synopsys Synthesis Tools, the VTIP, and the System HILO are briefly introduced. The interaction between the test generation system and these supporting systems are indicated.

3.1 Test Generation System

Figure 3.1 shows the test generation system. The VHDL behavioral model for a digital circuit is developed by a CAD tool called the Modeler's Assistant [44]. It employs a graphical representation of a VHDL behavioral model called a Process Model Graph

(PMG). The PMG is used as the base for our test generation approach. Here we also call a VHDL behavioral model the MUT (model under test).

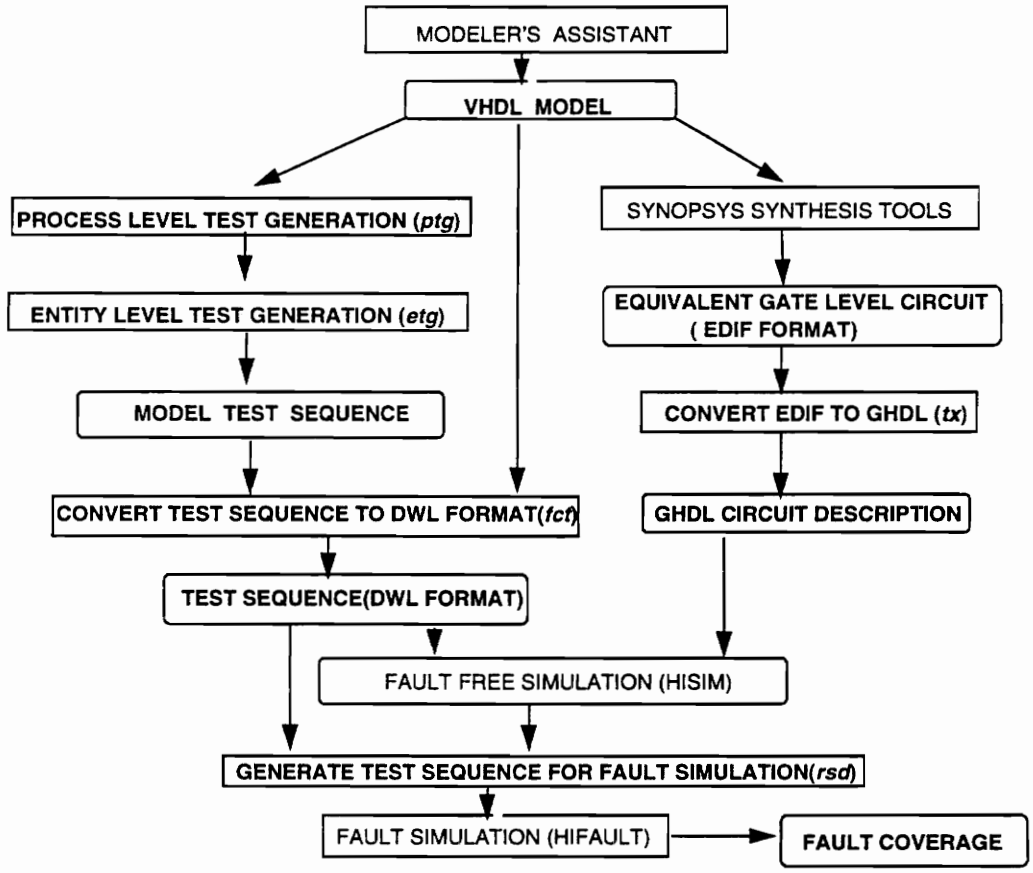


Figure 3.1. Test Generation System.

The VHDL model represented by a PMG contains a single entity consisting of multiple processes (or a single process). For each process, the process level test generation program (**ptg**), which employs the P-Algorithm, generates a *symbolic test set* (STS) that

is a stimulus/response test set using a symbolic notation for different kinds of values. A symbolic test in the symbolic test set may have a corresponding *fixed valued test package* (FVTP) that is a set of the specific values assigned to the symbolic test according to specific algorithms or rules. The FVTPs of all the symbolic tests of a process form a *fixed valued test set* (FVTS) for the process. Applying the FVTS to the equivalent gate level circuit of the process will generate high fault coverage.

Given the STSs and FVTSs of the processes of a MUT and the connection information among the processes, the entity level test generation program (**etg**), which employs E-Algorithm, tries to apply all the FVTSs by constructing forward propagation and backward justification paths using the STSs. The final test sequences generated are converted to HILO DWL format by program **fct** to test the synthesized gate level circuit corresponding to the MUT.

The Synopsys synthesis tools are used to generate the gate level circuits (EDIF format) from the VHDL description of a MUT. Such EDIF format circuits are passed to HILO TX_EDIF which gives a preliminary translation result. The program **tx** takes the preliminary result and generates the final GHDL format required by HILO. Because there exist some don't care values and other uncertain values in the test sequences, fault free simulation (HISIM) is performed, and its outputs are parsed by program **rsd** to decide these values. Finally fault coverage is obtained through fault simulation (HIFAULT).

In summary, the whole system can be divided into three parts: VHDL Behavioral Model Development completed by using the Modeler's Assistant, Test Generation composed of programs **ptg** and **etg**, and Test Evaluation supported by several commercial CAD systems and programs **fct**, **tx**, and **rsd**. Such systems and model development are discussed in the following sections. The test generation and test evaluation will be explained in detail in the separate chapters. The main contributions of this dissertation are the above programs and the corresponding algorithms.

3.2 VHDL Behavioral Model Development

3.2.1 The Modeler's Assistant

The Modeler's Assistant developed at Virginia Tech is a VHDL model development CAD tool. It employs a graphical representation of a VHDL behavioral model called a *Process Model Graph* (PMG). The nodes in a PMG are processes and the arcs are signals between the processes. A PMG is used to represent the division of functionality within a VHDL behavioral model, and gives an interconnection of different processes that partition the model into interrelated but distinct functional modules. Such information is used in the test generation algorithms.

Figure 3.2 shows a Modeler's Assistant screen showing the PMG for a five process model *incadd*. In the PMG, large circles represent processes. Small circles on the large circles represent ports. Filled small circles indicate that such ports are sensitive inputs. If there exists a global loop, the related ports are resented by double circles. The VHDL code for each process is specified by the user. Figure 3.3 shows the VHDL code for model incadd.

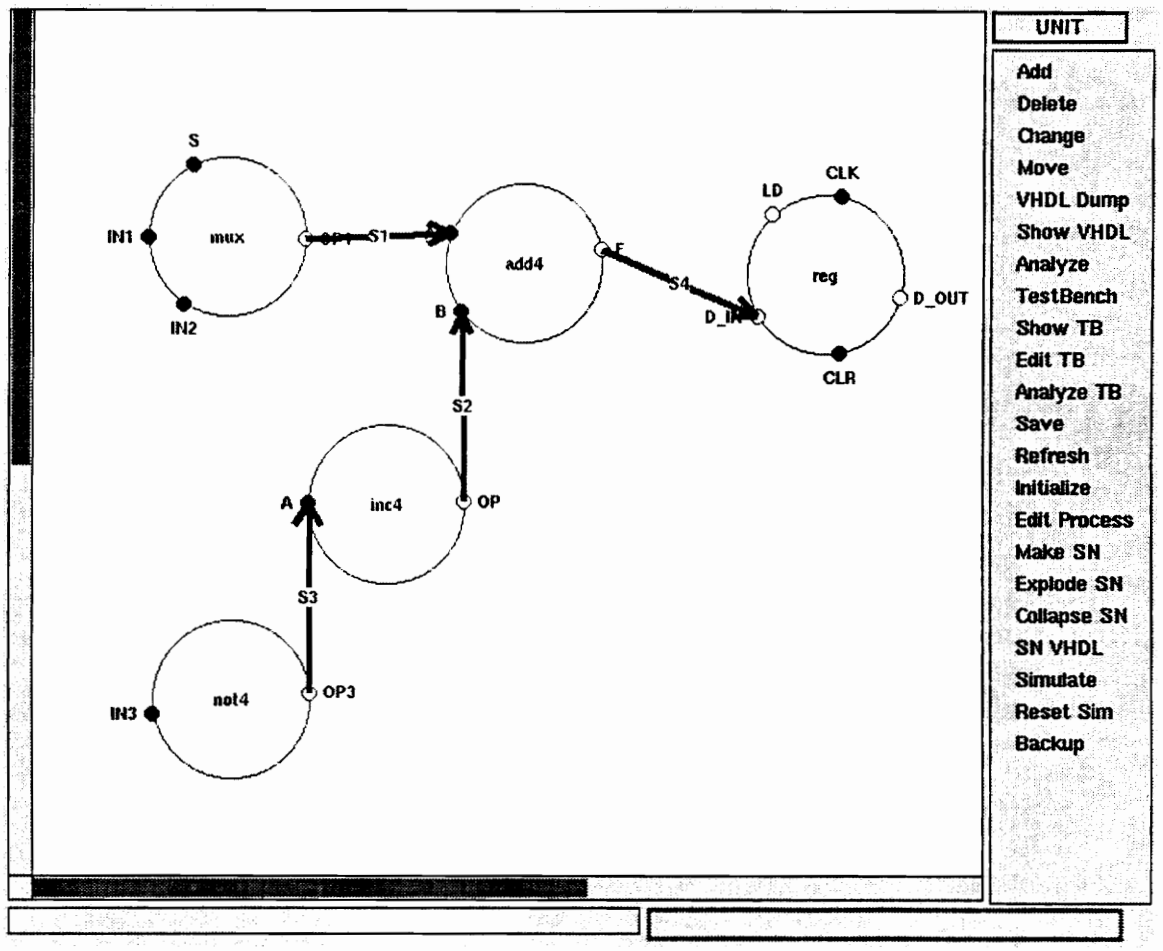


Figure 3.2 Modeler's Assistant (PMG of model incadd).

```
use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- ************************************************************
entity incadd is
  port (IN3: in BIT_VECTOR(3 downto 0);
    IN2: in BIT_VECTOR(3 downto 0);
    IN1: in BIT_VECTOR(3 downto 0);
    S: in BIT;
    LD: in BIT;
    D_OUT: out BIT_VECTOR(3 downto 0);
    CLK: in BIT;
    CLR: in BIT);
end incadd;
-- ************************************************************

architecture BEHAVIORAL of incadd is

  signal S3: BIT_VECTOR(3 downto 0);
  signal S2: BIT_VECTOR(3 downto 0);
  signal S1: BIT_VECTOR(3 downto 0);
  signal S4: BIT_VECTOR(3 downto 0);
begin

  -- ------------------------------------------------------
  -- Process Name: not4
  -- ------------------------------------------------------
  not4_4: process (IN3)
  begin
     S3 <= not IN3;
  end process not4_4;

  -- ------------------------------------------------------
  -- Process Name: inc4
  -- ------------------------------------------------------
  inc4_8: process (S3)
  begin
     S2 <= INC(S3);
  end process inc4_8;

  -- ------------------------------------------------------
  -- Process Name: mux
  -- ------------------------------------------------------
  mux_12: process (IN2,IN1,S)
  begin
     if S='0' then
          S1<= IN1;
     else
          S1<=IN2;
     end if;
  end process mux_12;
```

```
-- ------------------------------------------------------
-- Process Name: add4
-- ------------------------------------------------------
add4_18: process (S2,S1)
begin
   S4 <= ADD(S1, S2);
end process add4_18;

-- ------------------------------------------------------
-- Process Name: reg
-- ------------------------------------------------------
reg_23: process (CLK,CLR)
begin
   if CLR='1' then
        D_OUT<="0000";
   elsif CLK'EVENT and CLK='1' then
     if LD='1' then
        D_OUT<=S4;
     end if;
   end if;
end process reg_23;

end BEHAVIORAL;
```

Figure 3.3 VHDL Code of Model incadd.

The Modeler's Assistant is a powerful modeling tool. Only part of its function is used during test generation. Advanced functions like supernode can be used in the future to generate tests for more complicated models.

3.2.2 VHDL Subset

VHDL is an IEEE standard Hardware Description Language [53]. It has rich constructs to provide variety and flexibility for modeling digital circuits. Usually simulation tools can

support all the constructs in the language but synthesis tools only support a subset of VHDL. The VHDL subset chosen here is synthesizable and simple. This makes test generation easy without loosing generality.

Types

Predefined enumeration type BIT and array type BIT_VECTOR are allowed. Some other types defined in package VHDLCAD are also allowed. They are MVL, MVL_VECTOR, TSL, and TSL_VECTOR that can deal with 4-value logic and resolved values. No physical types are allowed. All the signal assignments have delta delays.

The reason for selecting these types is that the signals with such types will posses similar representations between the VHDL description and the equivalent gate level circuit. Other types like *IEEE std_logic_1164* can also be employed after making minor changes in the test generation programs.

Objects

The signal class is allowed. The permitted modes for ports are IN, OUT, and INOUT. Constant values are allowed. Indexed signals and variables can also be used in functions.

Attributes

The attribute 'EVENT is allowed in a process. The attributes 'RANGE, 'HIGH, 'LOW, and 'LENGTH are allowed in functions.

Operators

Logic operators AND, NAND, OR, NOR, XOR, NOT, and relational operator = are allowed. Arithmetic operations are implemented by functions.

Concurrent Statements

The only allowed concurrent statement is the process statement. But it can be shown that all concurrent statements map to processes [35].

Sequential Statements

A process statement can contain three kinds of sequential statements. They are *signal assignment statement*, *if statement*, and *case statement*. The predefined functions can use *variable assignment*, *loop statement*, *exit*, and *null*.

3.3 Supporting Systems

3.3.1 Synopsys Synthesis Tools [48, 49]

Synopsys synthesis tools can perform behavioral synthesis to convert a VHDL behavioral model into a gate level circuit. Such a conversion is completed by using the Synopsys VHDL Compiler [48] and Synopsys Design Compiler [49]. The VHDL Compiler translates and optimizes (high level) a VHDL description to an internal representation. The Design Compiler reads the design in internal format, then optimizes (logic level) and maps (gate level) the design's logical structure for a specific ASIC technology library.

VHDL Compiler

The VHDL Compiler reads a VHDL source code and maps various VHDL constructs inside the code to different function blocks corresponding to generic components. The circuit represented by such blocks usually is inefficient and may contain some redundant components. The VHDL Compiler performs high level optimization to get a more efficient internal representation. Typical high level optimization involves constant propagation and resource sharing. Constant propagation is the compile-time evaluation of expressions containing constants. Through constant propagation, the amount of hardware required to implement operators can be reduced. Sometimes it is possible that no hardware is constructed. For example, an AND gate with an input of logic 0 will be replaced by a wire connected to the ground. Resource sharing is the assignment of similar VHDL operations to a common netlist cell. Without resource sharing, each VHDL operation is built with separate circuitry. Therefore resource sharing reduces the amount

of hardware to implement VHDL operations. The VHDL Compiler performs the resource sharing for the build-in arithamtic operators under certain conditions.

Design Compiler

After a VHDL description is translated by the VHDL Compiler, the Design Compiler optimizes the design into a technology-specific circuit based on the attributes and constraints placed on the design. Optimizations can be performed at the logic level or gate level. At the logic level, combinational logic is represented by Boolean equations. Sequential and hierarchical portions of a design are linked to the equations with input and output variables. At the gate level, a design is represented as an interconnected network of circuit components such as inverters, logic gates, and flip-flops. *Flattening* and *structuring* are optimization steps operating at the logic level. *Mapping* is an optimization step that operates at the gate level. Flattening reduces a design's logic structure to a set of sum-of-products equations and therefore produces a faster design with larger area. After flattening, the original structure of its VHDL description is lost. Structuring regroups logic in a VHDL description by finding common factors to produce a slower design with smaller area. Mapping maps a gate-level description onto a technology library's components which are chosen on the basis of their function, their intrinsic gate delay, and their area and power requirements. The output of Design Compiler has several formats. The Electronic Design Interchange Format (EDIF) is chosen to facilitate the transfer of design information from Synopsys to HILO.

3.3.2 VTIP [52]

The VHDL Tool Integration Platform (VTIP) includes the Design Library System (DLS), the VHDL Analyzer, the VHDL Generator, example files, and command files. In our test generation system, the VHDL source code of a process is analyzed with the help of the VTIP VHDL Analyzer to obtain the various constructs of the process and put them into the DLS. Then such design information stored in the DLS is retrieved by the program **ptg** to construct the CFG and DFG of the process. The VHDL Generator translates design information stored in a design unit within the DLS to corresponding VHDL text. Therefore we can use it to automatically construct a new VHDL code after inserting some structures in the original VHDL source code to improve the testability in test synthesis.

VHDL Analyzer

The VHDL analyzer processes VHDL source code in two steps. In the first step, *Syntactic Analysis*, the syntactic correctness of the input code is verified, and DLS data structures are built to represent that input. In the second step, *Semantic Analysis*, the declarations and statements produced by the first step are examined to deduce their meaning and to verify their conformance to the rules of VHDL described in the Language Reference

Manual [53]. The final step puts the resulting DLS description of the source into the working design library.

Design Library System (DLS)

The DLS is a portable software platform that supports the integration of software tools used in the design of electronic systems. It consists of two fundamental components - one abstract, the other concrete. The abstract component is the DLS Data Definition which specifies the data elements, objects, and structures together with the abstract operations performed upon them. Corresponding to the specifications, there exist three layers: the Data Model, the Schema, and the Information Model.

The Data Model defines the basic data elements such as numbers, strings, lists, etc. Some of these elements are generic. They can be used to represent different kinds of data depending upon how they are interpreted. The Schema defines specific versions of such generic objects, and in doing so specifies their interpretation. The Information Model defines how the specific objects can be assembled into a complex structure. Associated with each layer in the DLS Data Definition is a collection of abstract operations that can be performed on the elements, objects, or structures defined by that layer.

The DLS models design data using an object-oriented approach. Each of the layers above defines a set of objects. The Data Model defines primitive objects (strings and numeric

values) as well as the generic objects *Node* and *List*. The Schema instantiates Nodes and Lists to create specific objects representing compound data elements (e.g., records). Finally, the Information Model defines structures consisting of these elements assembled in particular ways. Each such structure can be thought of as a composite object. Because of the object-oriented approach, design data can be modeled simply and compactly. The characteristics of object-oriented approach also make the modeling flexible.

The top-level organization of such design data consists of *libraries* that contain *library units*. Each library unit is a separate graph structure with unique root node. A single library may contain any number of library units. A library unit can refer to other library units, in the same library or in different libraries.

The concrete component of the DLS is the Software Procedural Interface (SPI) which is a software implementation of the DLS Data Definition that supports the development of an application based upon the DLS Data Definition. The SPI can be used to access and manipulate design data stored within the Design Library System. Using SPI facilities, a design tool can open a library unit, traverse its internal structure, examine data or objects within the structure, and even add or modify data. The SPI provides a standard interface for such tools, thus forcing a common interpretation of the design data managed by the DLS.

The SPI is a fundamental part of the DLS. The SPI consists of data types and callable routines that implement the data types and operations of the DLS. The SPI provides layers of data types and routines to support the layers of the DLS Data Definition. These layers include data types and generic data types of the DLS Data Model, and structures dependent on the DLS Schema and Information Model. The SPI provides additional types and routines for general tool support such as error management and host system interface.

3.3.3 System HILO [51]

System HILO is a set of software tools providing the designer of digital circuits with design verification and test validation capability. It consists of a set of modular programs, each of which executes from a menu-driven user interface called the Monitor. Monitor pages are provided which correspond to the main functions of the software suite.

Circuit Description and Compilation

The GenRad Hardware Description Language (GHDL) is used to define the structure and functionality of a circuit. Normally, a circuit is described as a number of GHDL primitive gates and sub-circuits which, when used together, describe the complete circuit. Circuit descriptions are compiled using the CIRCUIT monitor page. Running a compilation will check for source code errors and give a report. Circuit descriptions should be compiled

from the lowest level upwards. This means that the top level description is compiled last. The file CIRCUIT.IDX gives the index for all the circuits compiled up to date.

Waveform Preparation

Stimulus and response data for a circuit is specified in a waveform written in the Design Waveform Language (DWL). The WAVEWFORM page can be used to compile the source file and check for errors. However the compilation occurs automatically when running a simulation.

Fault-Free Simulation (HISIM)

The simulation can be performed once the circuit and waveform data have been prepared. The fault-free simulator inputs are specified on the HISIM page. These comprise the circuit description (*.cct), the waveform file (*.dwl), and other additional information. The simulator provides straightforward simulation of a circuit and waveform and monitoring activity on the circuit's wires and gates. The results of HISIM will give information about strobe failures which are used to decide some uncertain values in our test generation system.

During the initialization phase of the run, the circuit is loaded, the waveform is compiled and connected to the circuit, the simulator is initialized and finally the display sub-

systems are initialized. Then the simulation results are displayed during the execution of the simulator.

Fault Simulation (HIFAULT)

Fault simulation is executed from the HIFAULT page of the Monitor. All the data needed for the fault-free simulation are required. The fault simulator evaluates waveform test patterns by applying them into a circuit model containing faults. The waveform detects a fault when a circuit primary output value differs from the fault-free value. The fault simulator provides five fault classes:

- stuck — A short circuit between a wire and a power supply (stuck0 and stuck1).
- open — A break in the wire or solder.
- drive — An internal gate fault causing the output to drive a permanent 0 or 1 at its normal strength (drive0 and drive1).
- inhibit — A functional fault resembling a stuck wire which prevents an event firing.
- short — A generalized form of 'stuck' where any two wires may be soldered together.

When we use the Synopsis synthesis tool to get a gate level circuit, the circuit may contain some blocks such as flip-flop, latch, and multiplexer. Fault simulation can be done at the top-level or all-level. At the top-level, the faults within a block will not be activated. At the all-level, all the faults in a circuit will be activated. Faults to be simulated are specified through fault specification file or the FAULTS Monitor page. For

example, one may specify to simulate all top-level wires with stuck-at faults. HIFAULT generates a fault dictionary. Summaries of the statistics of the fault simulation are also available.

The System HILO assists our test generation system to complete fault coverage evaluation. The gate level circuits generated by using the Synopsys synthesis tools are converted into GHDL format. The test sequences generated by **ptg** and **etg** are translated into DWL format. The HISIM is employed to obtain good output values of the circuits by applying their test sequences. Finally, the HIFAULT gives the fault coverage (stuck faults). The fault dictionary generated by the HIFAULT are also helpful for finding hard-detected faults in the gate level circuits.

Chapter 4. Process Level Test Generation

In this chapter, some basic concepts are presented first. Then the test generation for the predefined functions is discussed, which includes the VHDL implementations of the functions, the symbolic test characteristics, and the FVTP generation algorithms. Before presenting the P-algorithm, some criteria and assumptions are explained. Finally, a few examples are showed.

4.1 Control Data Flow Graph

A *Control Flow Graph* (CFG) is a directed graph $G(N_c, E_c)$. It consists of a set of nodes N_c and a set of directed edges E_c such that $e_{ij} \in E_c$ denotes a directed edge from the head node $n_i \in N_c$ to the tail node $n_j \in N_c$.

A node n_i is defined as an element in CFG that represents a control statement or an assignment statement within a VHDL process. A directed edge e_{ij} is defined as a pair of

nodes (n_i, n_j) such that the node n_i proceeds the node n_j during the execution of the CFG and the edge has a property value (**T**rue or **F**alse) to indicate the condition of executing the following node. The control statements (also called *control nodes*) are *case statements*, *if statements* and *for loop statements*. An assignment statement (*assignment node*) is a *signal assignment statement* or a *variable assignment statement*.

A *Data Flow Graph* (DFG) is also a directed graph $G(N_d, E_d)$. Here, N_d is a set of nodes that represent the data (signals and variables) and the operations (shown as circles). E_d is a set of directed edges that indicate the direction of the data flow among the nodes.

A *Control Data Flow Graph* (CDFG) incorporates the CFG and DFGs of a process to represent the data flow and sequencing of the DFGs. The CDFG of a VHDL process can show the data dependencies and control sequences within the process.

In a CFG, each node except for first one (first statement), has one input edge. An assignment statement node has one output edge or no output edge. A control statement node has one or two output edges. An *end node* is an assignment node with no output edge or a control statement node with one empty branch. Along a path from the first node to an end node in a CFG, the control signal values can be specified according to the edge property values. The data relation among the input/output signals can be determined by analyzing the DFG corresponding to the path. Figure 4.1 presents a CDFG for process

EXMP. Two DFGs are generated from the CFG by selecting different values of the control signal N.

```
EXMP: process (A, B, N)
  variable XA, XB: BIT_VECTOR(A'LENGTH-1 downto 0)
begin
  XA:=A;                  --s1;
  XB:="0001";             --s2;
  if N='1' then           --s3;
    XB:=B;                --s4;
  end if;
  SUM<=ADD(XA,XB);  --s5;
end process EXMP;
```

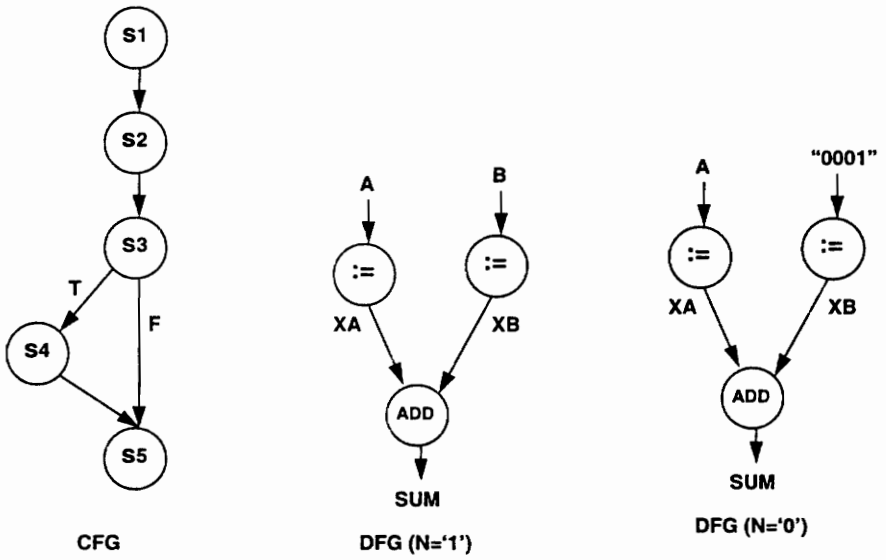


Figure 4.1 Illustration of CDFG.

Besides the CDFG, the process level test generation algorithm needs other information such as the declarations of ports, and the type and range of a signal. Thus two additional

nodes, the *declaration node* and the *signal/variable node*, are defined to complement the information stored in CDFG nodes, and thus give a complete *Directed Model Graph* (DMG) representing the information to be used for process level test generation [37].

4.2 Symbolic Test

4.2.1 Symbolic Notation

We use D_i *(i=1,2,3,...)* to represent data inputs and outputs, R (0->1) and F (1->0) to represent the events on control signals that usually are clock signals, P to indicate the previous value of a signal in a sequential process, *X* to indicate the don't care, and *Z* to represent the high-impedance state. DC_0 (all *0s*) and DC_1 (all *1s*) are used to represent the preassigned data inputs.

Table 4.1 gives the symbolic tests for logic operators AND, OR, XOR, NOT and functions INC, DEC, ADD, MULT, COMP, SHIFTL, SHIFTR, SHIFTA. The tests for NAND (NOR) can be obtained by complementing the output values of AND (OR). There are four unary operators: *n* (not), *i* (increment), *d* (decrement), and *t* (arithmetic right shift) and eight binary operators: *a* (and), *o* (or), *e* (xor), *s* (addition), *c* (comparison), *m* (multiply), *l* (left shift), and *r* (right shift). Symbol ' is used to represent inversion

operations. In Section 4.3 we will show how these tests are constructed and what kinds of properties they have.

Table 4.1 Symbolic Tests for Functions and Logic Operations.

	op	A	B	F	mode
F<= A op B;	AND	DC1	D1	D1	P
		D1	DC1	D1	P
		D1	D2	a(D1, D2)	A
	OR	DC0	D1	D1	P
		D1	DC0	D1	P
		D1	D2	o(D1, D2)	A
	XOR	DC0	D1	D1	P
		D1	DC0	D1	P
		D1	D2	e(D1, D2)	A
F<=op(A);	NOT	D1		nD1	P
	INC	D1		iD1	P
	DEC	D1		dD1	P
	SHIFTA	D1		tD1	A
F<= op(A,B);	SHIFTR	D1	D2	r(D1, D2)	A
	SHIFTL	D1	D2	l(D1, D2)	A
	ADD	DC0	D1	D1	P
		D1	DC0	D1	P
		D1	D2	s(D1, D2)	A
	MULT	D1	D2	m(D1, D2)	A
	COMP	D1	D2	c(D1, D2)	A

4.2.2 Symbolic Test Modes

The *mode* of *a symbolic test* in a symbolic test set (STS) reflects the specific type of relation between the input signals and an output signal of a process. The E-algorithm uses modes to select appropriate symbolic tests. There are four kinds of modes.

(1) *Initialization mode* (I-mode)

An I-mode test assigns the output a constant. Therefore it can be used to initialize a process representing a sequential circuit.

(2) *Hold mode* (H-mode)

An H-mode test makes the output of a process keep its previous value. In VHDL, this can be explicitly indicated (like *A*<=*A*;) or implied by a conditional structure. For example, an incompletely specified *if statement* in a combinational process will imply a latch in the synthesized gate level circuit. This will correspond to an H-mode test.

(3) *Propagation mode* (P-mode)

A P-mode test establishes a one-to-one and onto mapping between a data input and a data output of a process. It can be shown that the input and the output must have same bit width in order to maintain such a mapping. A P-mode test is constructed by keeping an input as a symbolic variable and assigning all the other inputs as fixed values. For example, a P-mode test for an adder can be obtained by assigning one input as D_1 and the other input as a constant. A P-mode test is suitable for both forward propagation and backward justification.

(4) *Activation mode* (A-mode)

An A-mode test is a symbolic test that does not belong to the above three modes. It usually has more than one input. No one-to-one and onto mapping exists. But there may exist a one-to-one or onto mapping. Such a test is used to activate a process and also can be a candidate for propagation and justification. Actually, the test represents the complete data relation under certain control conditions within a process.

In above discussion, we pay special attention to the properties of the mappings between the inputs and the outputs of a process. This is because such properties can be used to select an appropriate data propagation path or justification path during test generation. For example, in order to propagate the fault effects of preceding processes to the primary outputs through a process, different input values of the process should result in different output values. This means the mapping should be one-to-one. For justification, each output should have a corresponding input. That means the mapping should be onto. Therefore a one-to-one mapping can be used for propagation and an onto mapping used for justification.

From Table 4.1 we can know that some operations have both P-mode and A-mode tests. However some operations only have A-mode tests, so it may be difficult to propagate or justify a value through such operations.

4.3 Function Test Generation

The following six functions and the VHDL subset are the basic elements for constructing a behavioral model. However we can define more functions if necessary. All these functions are included in the package VHDLCAD. The VHDL code, symbolic test derivation, and FVTP generation algorithm for each function will be introduced next.

Algorithms 1~6 generate the FVTPs for adder, incrementer, decrementer, shifters, multiplier, and 3-function comparator. Suppose that all the data inputs are n-bit BIT_VECTORs. We use a pair $(0^m 1^{n-m}, 1^k 0^{n-k})$ to represent the values on inputs A and B. Here $A=0^m 1^{n-m}$ (*m 0s* followed by *n-m 1s*); $B=1^k 0^{n-k}$ (*k 1s* followed by *n-k 0s*). 1^0 or 0^0 means that no *1* or *0* exists.

4.3.1 Addition

The function ADD adds two operands A and B and returns the sum. It is implemented by the ripple carry method. For simplicity, we assume that no carry input and carry output exist.

```
function ADD(A,B: in BIT_VECTOR) return BIT_VECTOR is
  variable SUMV,AV,BV: BIT_VECTOR(A'LENGTH-1 downto 0);
  variable CARRY: BIT;
begin
  AV := A;
  BV := B;
  CARRY := '0';
  for I in 0 to SUMV'HIGH loop
    SUMV(I) := AV(I) xor BV(I) xor CARRY;
```

```
    CARRY := (AV(I) and BV(I)) or (AV(I) and CARRY) or (BV(I) and CARRY);
  end loop;
  return SUMV;
end ADD;
```

The values on one input can be propagated to the output by assigning any specific value to the other input. For simplicity, we choose DC_0 (all *0s*) as such a specific value and get two P-mode tests. An A-mode test is also constructed. It can be used to handle multiple path propagation and generate new P-mode tests by assigning the specific values to one input.

Compared to the symbolic test generation of a function, the corresponding FVTP generation is much more difficult. This is because the symbolic tests are decided by the functionality of the function but the FVTPs are dependent on the implementation of the function. We can always generate some kinds of symbolic tests for a function according to its functionality. But when we generate the FVTPs, we should know how the function is implemented. For example, the Synopsys synthesis tools can give three implementations for the "+" (addition) operator: ripple carry synthesis model, carry look-ahead synthesis model, and fast carry look-ahead synthesis model [50]. In our system, we implemented a function by providing the predefined VHDL code. The code can be converted to a gate level circuit by using certain kinds of synthesis tools. Different synthesis tools will generate different gate level circuits. The FVTP generation algorithm for a function is derived by analyzing the VHDL implementation and the gate level

circuit. Therefore, the algorithms derived are synthesis related. Because we use the Synopsys synthesis tools during deriving the algorithms, all the algorithms are meaningful only when the Synopsys synthesis tools are used. However, the algorithms can be developed for other synthesis tools.

Algorithm 1 generates fixed valued tests for ADD. They are composed of two parts. The first part is the tests for data paths which have the effect of exhaustive tests (explained in next section). Each test in the second part will generate a carry within the carry chain. The experimental results have shown that the tests generated by such an algorithm have high gate level fault coverage. The other five algorithms are derived based on the same idea. Figure 4.2 shows the vectors generated for 4-, 8-, and 16-bit adders. Such vectors have fixed form and length.

Algorithm 1 (test generation for ADD(A,B))

(1). test vector generation for data paths

put $(0^n,0^n)$, $(0^n,1^n)$, $(1^n,0^n)$, and $(1^n,1^n)$ into the test set.

(2). test vector generation for the carry chain

for $k=0$ to $n-1$, do{put $(0^{n-k-1}10^k,0^{n-k-1}10^k)$ and $(0^{n-k-1}10^k,1^n)$ into the test set}.

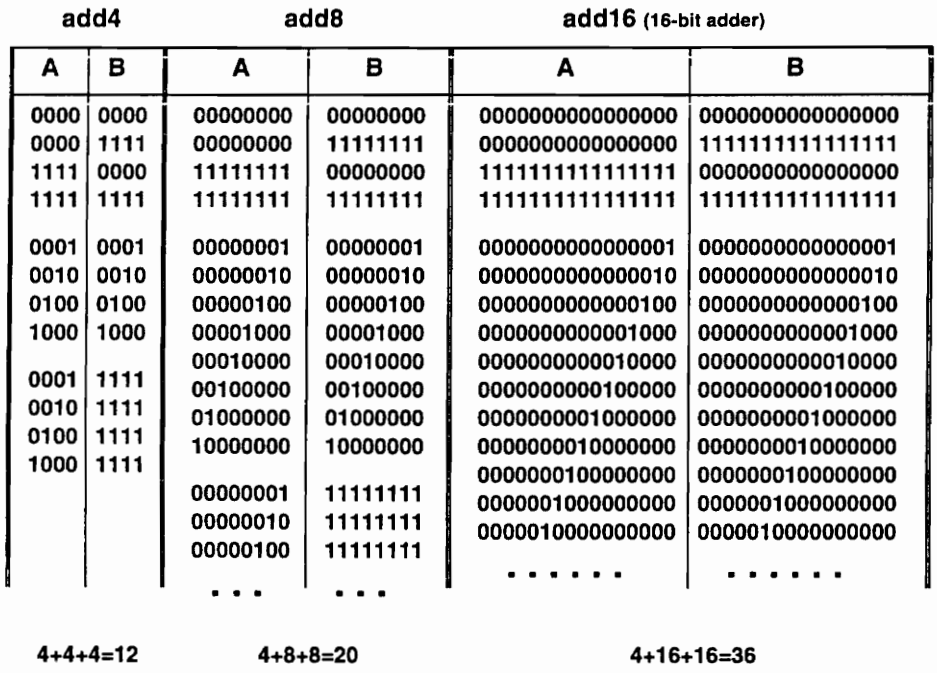


Figure 4.2 Test Vectors of 4-, 8-, 16-bit Adders.

4.3.2 Increment and Decrement

The algorithms for implementing function INC (increment) and DEC (decrement) come from [32].

```
function INC(X : BIT_VECTOR) return BIT_VECTOR is
  variable XV: BIT_VECTOR(X'LENGTH-1 downto 0);
 begin
  XV := X;
  for I in 0 to XV'HIGH loop
   if XV(I) = '0' then
```

```
    XV(I) := '1';
    exit;
   else XV(I) := '0';
   end if;
  end loop;
  return XV;
 end INC;


function DEC(X : BIT_VECTOR) return BIT_VECTOR is
  variable XV: BIT_VECTOR(X'LENGTH-1 downto 0);
 begin
  XV := X;
  for I in 0 to XV'HIGH loop
   if XV(I) = '1' then
    XV(I) := '0';
    exit;
   else XV(I) := '1';
   end if;
  end loop;
  return XV;
end DEC;
```

Following shows that the mapping between the input and the output of function INC is one-to-one and onto. Therefore the symbolic test is a P-mode test. We can use the same approach to show that the function DEC also creates a one-to-one and onto mapping.

Let $B^n=\{(b_1b_2 \ldots b_n) \mid b_i \in \{0,1\},\ 1 \le i \le n,\ n$: positive integer$\}$. That is B^n is the set of all n-bit binary numbers. Then $F = INC(A)$ defines a mapping on B^n . Here $A, F \in B^n$, and $F \equiv A+1 (mod\ 2^n)$. So, $F=A+1$ for $0 \le A < 2^n-1$ and $F=0$ for $A=2^n-1$. It is obvious that for any $A_1, A_2 \in B^n$ and $A_1 \neq A_2$, there exists $F_1, F_2 \in B^n$ such that $F_1 = INC(A_1)$,

F_2= *INC(A2),* and $F_1 \neq F_2$. So INC is a one-to-one mapping. On the other side, for any $F_i \in B^n$ we can find an $A_i \in B^n$ such that $F_i = INC(A_i)$. Therefore INC is also an onto mapping.

Algorithms 2 and 3 generate fixed valued tests for the functions. Figure 4.3 shows the vectors for 4-, 8-, and 16-bit incrementers and decrementers.

Algorithm 2 (test generation for INC(A))

(1). put 1^n, 0^n , and $1^{n-1}0$ into the test set.

(2). for $k=1$ to $n-1$, do {put $(0^{n-k}1^k)$ into the test set}.

INC(A)	INC(A)	INC(A) (16-bit incrementer)
A	A	A
1111	11111111	1111111111111111
0000	00000000	0000000000000000
1110	11111110	1111111111111110
0001	00000001	0000000000000001
0011	00000011	0000000000000011
0111	00000111	0000000000000111
	00001111	0000000000001111
	00011111	0000000000011111
	00111111	0000000000111111
	01111111	0000000001111111
		0000000011111111
		0000000111111111
		0000001111111111
		0000011111111111
		. . . . .
3+3=6	3+7=10	3+15=18

DEC(A)	DEC(A)	DEC(A) (16-bit decrementer)
A	A	A
1111	11111111	1111111111111111
0000	00000000	0000000000000000
0001	00000001	
0010	00000010	0000000000000001
0100	00000100	0000000000000010
1000	00001000	0000000000000100
	00010000	0000000000001000
	00100000	0000000000010000
	01000000	0000000000100000
	10000000	0000000001000000
		0000000010000000
		0000000100000000
		0000001000000000
		0000010000000000
		. . . . .
2+4=6	2+8=10	2+16=18

Figure 4.3 Test Vectors of Incrementers and Decrementer.

Algorithm 3 (test generation for DEC(A))

(1). put 1^n and 0^n into the test set.

(2). for $k=0$ to $n-1$, do {put $(0^{n-k-1}10^k)$ into the test set}.

4.3.3 Shifting

There are three shifting functions: SHIFTL, SHIFTR, and SHIFTA.

```
function SHIFTL(A: in BIT_VECTOR; SI: in BIT) return BIT_VECTOR is
  variable DATA_OUT: BIT_VECTOR(A'LENGTH-1 downto 0);
begin
```

```
    DATA_OUT := A(A'LENGTH-2 downto 0) & SI;
    return DATA_OUT;
end SHIFTL;

function SHIFTR(A: in BIT_VECTOR; SI: in BIT) return BIT_VECTOR is
  variable DATA_OUT: BIT_VECTOR(A'LENGTH-1 downto 0);
begin
    DATA_OUT := SI & A(A'LENGTH-1 downto 1);
    return DATA_OUT;
end SHIFTR;

function SHIFTA(A: in BIT_VECTOR) return BIT_VECTOR is
  variable DATA_OUT: BIT_VECTOR(A'LENGTH-1 downto 0);
begin
    DATA_OUT := A(A'LENGTH-1) & A(A'LENGTH-1 downto 1);
    return DATA_OUT;
end SHIFTA;
```

Assume *"F <= SHIFTL(A, B);"*. Here, the type of A is BIT_VECTOR and the type of B is BIT. *F= A(A'LENGTH-2 downto 0)&B*. The mapping from the inputs A and B to the output F is onto but not one-to-one because for any value of F one can find the values of A and B to make *F=SHIFTL(A,B)* but two values of A that are same in all the bits except the leftmost will map to the same output value after setting B a fixed value. It is obvious that the mapping from the input B to the output F after assigning the input A a fixed value is one-to-one but not onto. However the mapping between the input A and the output F after assigning the input B a fixed value is neither one-to-one nor onto. This is because two input values that are same in all the bits except the leftmost will map to the same output value and no input value exits corresponding to an output with the rightmost bit that equals to the complement of B. Therefore only A-mode symbolic tests exist. One can get the same conclusions for the other two shift operators SHIFTR and SHIFTA.

Algorithm 4 will produce the fixed valued tests for the shifting functions. It is obtained assuming all three functions are used in a process.

Algorithm 4 (test generation for shifting functions)

(1) test vector generation for left shift

put $(0^n,0)$ and $(1^n,0)$ into the test set.

(2) test vector generation for right shift

put $(0^n,1)$ and $(1^n,1)$ into the test set.

(3) test vector generation for arithmetic right shift

put $(1^n,1)$ into the test set.

4.3.4 Multiplication

Let $X = (X_{n-1}, ..., X_0)$, $Y = (Y_{n-1}, ..., Y_0)$, and $P = (P_{2n-1}, ..., P_0)$ be the multiplicand, multiplier and product. The multiplication can be implemented as follows [58].

```
function MULT(A,B: in BIT_VECTOR) return BIT_VECTOR is
  variable X,Y: BIT_VECTOR(A'LENGTH-1 downto 0);
  variable P,C: BIT_VECTOR(2*(A'LENGTH)-1 downto 0);
begin
  X := A;
  Y := B;
  for I in 0 to P'HIGH loop
        P(I):='0';
  end loop;
  for I in 0 to X'HIGH loop
    if Y(I)/='0' then
        C(I):='0';
```

```
        for K in 0 to X'HIGH loop
          C(I+K+1):=(P(I+K) and X(K)) or((P(I+K) xor X(K)) and C(I+K));
          P(I+K):=P(I+K) xor X(K) xor C(I+K);
        end loop;
        P(I+X'HIGH+1):=C(I+X'HIGH+1);
    end if;
  end loop;
  return P;
end MULT;
```

No one-to-one and onto mapping exists between the inputs and output of the function MULT . However a one-to-one mapping between one input and the output can be constructed by assigning the other input a non-zero value. Therefore data propagation can be done. Because the bit width of the output is $2n$ and that of the inputs is n, it is impossible to construct an onto mapping between one input and the output. In addition, because not every value can be represented as the product of other two values, there exists no onto mapping between the two inputs and output. So the justification may fail for some values of the output.

Algorithm 5 (test generation for MULT(A, B))

(1) for $k=0$ to $n-1$, do {put $(0^{n-k-1}10^k, 1^n)$ into the test set}.

(2) for $k=0$ to $n-2$, do {put $(0^{n-k-2}110^k, 1^n)$ into the test set}.

(3) put $(10^{n-2}1,1^n)$, $(0101...,1^n)$, $(1010...,1^n)$, and $(0^{n-1}1,0^n)$ into the test set.

(4) put all reverse vectors(A<->B) from (1) to (3) into the test set.

(5) put *(1010...,11001100...)* and *(1010...,11001100...01)* into the test set.

4.3.5 Comparison

The function COMP is a two input comparator. It compares two BIT_VECTORs (A and B) and returns an output condition: equal *("00")* or greater-than *("01")* or less-than *("10")*.

```
function COMP(A,B: in BIT_VECTOR) return BIT_VECTOR is
variable XV: BIT_VECTOR(1 downto 0);
 begin
    XV := "00";
    for I in A'HIGH downto 0 loop
        if (A(I) = '1' and B(I) = '0') then
          XV:="01";
          return XV;
        elsif (A(I) = '0' and B(I) = '1') then
          XV := "10";
          return XV;
        else
          null;
        end if;
    end loop;
    return XV;
 end COMP;
```

The fault effects on one input can be propagated to the output by assigning the two inputs the same symbolic values. When no fault effect exists on any input, the output F will be *"00"*. When faults make one input change, the output will be *"01"* or *"10"*. Therefore fault effects can be propagated.

The function COMP creates a mapping from $B^n \times B^n$ to B^2. Here $B^2 = \{00,01,10,11\}$. Because there exist at least two elements in $B^n \times B^n$ that are mapped to *"00"*, thus the mapping is not one-to-one. The range R of the mapping is $\{00,01,10\}$. It is obvious that $R \subset B^2$. So the mapping is not onto. However if $F \in R$, the justification still can be done. If $F \in B^2$ but if $F \notin R$, the justification fails.

Algorithm 6 (test generation for COMP(A, B))

(1) test vector generation for $A > B$

for $k=0$ to $n-2$, do {put $(0^k10^{n-k-1}, 0^{k+1}10^{n-k-2})$ into the test set };

put $(0^{n-1}1, 0^n)$ into the test set.

(2) test vector generation for $A < B$

for $k=0$ to $n-2$, do put{$(0^{k+1}10^{n-k-2}, 0^k10^{n-k-1})$ into the test set};

put $(0^n, 0^{n-1}1)$ into the test set.

(3) test vector generation for $A = B$: put $(0^n, 0^n)$ and $(1^n, 1^n)$ into the test set.

4.4 Test Generation Rules

Rule 1 (test generation for the processes like registers, flip-flops, multiplexers, latches, and logic operations)

All these processes have P-mode tests. Each of these tests has only one D_i on a data input and other data inputs, if exist, are X or DC_i. Moreover the control signals are set to fixed

values. Choose a P-mode test from the symbolic test set of such a process and assign all 0s and all 1s as the values of D_i in the test. Then combine the data values and control values to form the FVTP corresponding to the test. Only the FVTPs of the P-mode tests are needed for testing such processes.

Rule 1 is suitable for the bitwise symmetric structures. Each bit of the signal has the same properties. So applying all *1s* and all *0s* to the signal is equivalent to applying both 1 and 0 to each bit of the signal. Therefore the effect of an exhaustive test is obtained. We can also apply some other constant value pairs such as *"101010..."* and *"010101..."* to replace all *1s* and *0s*.

Rule 2 (test generation for the process with functions)

Based on the FVTP(s) of a special function, assign other signal values.

If a process contains such a function, then the test vectors for the process are generated to cover all the FVTP(s) of this function. For example, the 4-bit up-counter in Figure 4.4 contains the function INC, so the FVTP for the counter should cover the vectors *"1111 0000 1110 0001 0011 0111"* that are in the FVTP of 4-bit incrementer.

```
upcnt4: process (CLR, CLK)
begin
        if CLR = '1' then
          CNT <= "0000";
        elsif CLK'EVENT and CLK='1' then
```

```
            if LD='1' then
                  CNT<=DATA_IN;
            elsif CT='1' then
                  CNT<=INC(CNT);
            end if;
          end if;
end process;
```

Figure 4.4 VHDL Code for 4-bit Up-Counter.

Rule 3 (test generation for the process with loops like A<=INC(A))

If there exists such a loop, the output signal will also appear in the source expression. Assign "P" as the signal value at the source expression and treat it like other input signals. Then use the rule 1 and 2 to construct FVTP for the process.

4.5 Criteria and Assumptions

The main criterion used by previous PTG [37] was the test sets generated should be such that every assignment node of the CFG is passed (or executed) at least once. This is not sufficient for generating a test set with high gate level fault coverage. The new algorithm uses different criteria.

The criteria for symbolic test generation are **exhaustive control path construction (CFG)** and **selective multiple data path construction**. Because the processes in a MUT are relatively simple, it is feasible to exhaustively construct all single execution paths

from the inputs to the outputs of a process. In certain cases, we also need to construct multiple data paths to activate some functions. For example, when a process contains the function MULT(A,B), we need to construct two backward data paths starting from A and B to the primary inputs. The multiple propagation paths may also be generated if fanout exists. All the assignment statements and the decision branches of the control assignments can be executed at least once by using such criteria.

The equivalent gate level implementations of different assignment statements can be quite different. They may contain a few gates or thousands of gates. Therefore, the symbolic tests corresponding to such statements should have the FVTPs with different sizes. We have many choices when the input signals are of the type BIT_VECTOR. For an n-bit vector, there are 2^n possible values. It is impossible to contain all the values in a FVTP when n is large. Thus the criterion for fixed valued test generation should be such that **the size of each FVTP is as small as possible without decreasing the gate level fault coverage**. In order to implement such a criterion, we develop a set of rules and algorithms for the symbolic tests of basic operations and functions. The FVTPs of other symbolic tests for a process can be determined by applying the related basic FVTPs.

The criteria for symbolic test generation are based on software testing principles that try to exercise all the required functions. The criterion for fixed valued test generation is based on gate level test generation principles. This is because there exists a corresponding

part in the synthesized gate level circuit for an assignment statement with a complex function. Therefore we can apply the FVTP for such a function so as to increase the fault coverage for whole circuit. On the other hand, we test the control conditions of a process by exercising all possible paths.

Assumptions

To implement a behavioral function, we can construct a VHDL model with a few complex processes or a model with a lot of simple processes. So we need to do some trade-off between the number of processes in a MUT and the complexity of its processes. We give four assumptions for the processes in a MUT. For the number of the processes, we do not give any restriction. These assumptions are sufficient to create a behavioral model with moderate complexity.

(1) Use the VHDL subset.

(2) No fanout reconvergency exists in a DFG.

(3) A DFG can contain one function and the input(s) and output of the function can be only related to no more than one signal respectively. Otherwise the FVTP for the process will be too complex.

(4) The control structure can be complex like embedded *if* or *case statements*.

Under the assumption (2), a DFG will become a directed tree with the output signal as its root or a group of such trees with each output signal as a root. However these trees may have cross parts when there exists fanout. Assumptions (2) and (3) make the process level test generation easy. If a process can not meet such assumptions, we try to split the process into several processes. Entity level test generation algorithm has the backtrack mechanism so it can handle the complex cases like fanout reconvergency.

4.6 P-Algorithm

Based on the basic symbolic test set and the basic FVTP generation algorithms and rules, a process level test generation algorithm, called **P-algorithm**, is developed. It contains following five parts:

1. Construct a control flow graph (CFG) for the process
2. Activate each CFG path by selecting values for control signals
3. Generate a data flow graph (DFG) for each CFG path
4. Generate symbolic test set (STS) for the DFG
5. Generate fixed value test packages (FVTPs) from the symbolic tests

While generating symbolic tests for a DFG, the algorithm does symbolic value propagation and justification. The symbolic tests in Table 4.1 establish the base for such

operations. The final symbolic tests for a process are obtained by combining the results from the step 2 and 4. Each symbolic test is assigned a test mode. Figure 4.5 shows the detailed pseudocode of the algorithm that is explained in following sections.

```
P-algorithm(analyzed VHDL process model):
begin
  construct  CFG; /* modified from Kapoor's program[37] */
  while there are paths in the CFG unexercised do
    select a path from root to an end node;/* select_path();*/
    while there exists node in the path unprocessed do
      if  condition node then
        decide control signal value;
      else if assignment node then
        construct a subtree; /* modified from Zhong's program[69] */
      end if;
    end while;
    if DFG contains no subtree then /* no assignment statement exists in the CFG path*/
      generate an H-mode symbolic test;
    else if DFG contains more than one subtree then /*multiple assignment statements*/
      for from last subtree to first subtree do
        if target signal is unassigned then
          if the subtree contains only a single leaf with constant value then
            generate an I-mode symbolic test;
          else if the subtree contains only a single signal leaf  then
            if there exists a loop then
              generate an H-mode symbolic test;
            else
              assign symbolic value to the leaf signal;
              generate a P-mode test;
            end if;
          else if the subtree contains multiple leaves then
            assign symbolic values to leaves;
            generate an A-mode symbolic test;
          end if;
        end if;
        resolve test mode;
      end for;
```

```
    else if DFG contains single subtree then /*only one assignment statement exists.*/
      if DFG contains only a single leaf with constant value then
        generate an I-mode symbolic test;
      else if DFG contains only a single signal leaf then
        if there exists a loop then
          generate an H-mode symbolic test;
        else
          assign Di to the leaf signal;
          generate a P-mode test;
        end if;
      else if DFG contains multiple leaves then
        if there exists a loop then
          assign "P" to the leaf within the loop and Di to other leaves;
          generate an A-mode symbolic test;
        else if there exists a leaf with constant value then
          assign Di to other leaves;
          generate an A-mode symbolic test;
        else
          if there exists no function like multiplication, comparison and shifting then
            generate a P-mode test for each leaf by constructing data path;
          end if;
          if there exists such a function or other bi-operator then
            generate an A-mode symbolic test;
          end if;
        end if;
      end if;
    end if;
  end while;
  generate FVTPs;
end P-algorithm;
```

Figure 4.5 P-algorithm.

4.6.1 CDFG Construction

The P-algorithm starts by constructing a CFG for a process. The VHDL code for the process is obtained from the PMG of an entity which contains the process. Such code is expanded to become a new entity that is analyzed with the VTIP VHDL analyzer. The algorithm reads the information stored in VTIP's DLS Design Libraries and parses the VHDL structures to construct the CFG. All the other information required by the P-algorithm is also obtained at the same time.

Using the depth-first path search method, the P-algorithm selects control paths from the CFG one by one. A control path starts from *root node* which is the first statement in the process to an *end node* which is an assignment node with no output edge or a control statement node with one empty branch. Figure 4.6 is used to illustrate how to construct control paths. Here the root node is node 1. The end nodes are nodes 3, 4, 7, 8, and 9. A stack "Pstack" is used to store branch nodes. The path selection algorithm chooses node 1 as first node. Node 2 is selected as second node and node 6 is pushed into Pstack. Then node 3 is selected as third node and node 5 is pushed into the stack. Because node 3 is a end node, a control path is completely constructed. Node 4 is pushed into Pstack for next path construction. Now we get the first control path P1={1(T), 2(T), 3(F)}. The stack is Pstack={6, 5, 4}. Then each following path is constructed based on the previous one. For example, second path P2 is constructed through getting a node (node 4) from the stack and finding its parent node in P1. Because node 4 is an end node, we get a second path P2={1(T), 2(T), 3(T), 4} with Pstack={6,5}. The third path P3 is constructed by finding

the parent node of node 5 in P2 and searching following nodes until the end node 9 is reached. Then P3={1(T), 2(T), 5(T), 9} and Pstack={6}. The other paths can be constructed in the same way.

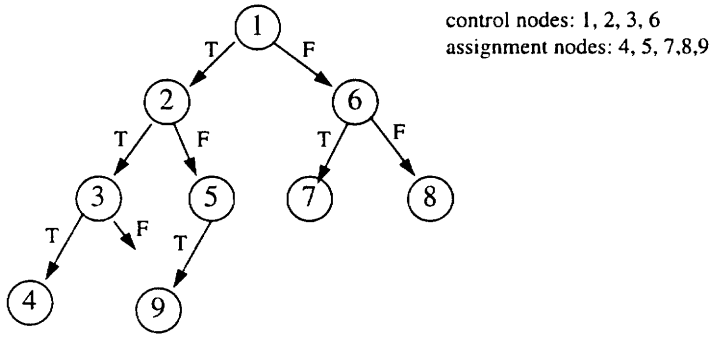


Figure 4.6 A CFG to Illustrate Path Selection.

Along a path from the first node to an end node in a CFG, the control signal values can be specified according to the edge property values (True or False). The number of assignment nodes in a path varies from 0 to more than one. A subtree will be constructed corresponding to each assignment node. The DFG of the path consists of all the subtrees. The data relation among the input and output signals can be decided after analyzing the DFG.

4.6.2 Symbolic Test Generation

Because the number of subtrees is variable, three cases will be considered based on the number. The P-algorithm uses different methods to handle the three cases while generating symbolic tests for the DFG.

Case 1: no subtree

If there exists no assignment statement along a control path, all the output signals will keep their previous values under the given control values. The P-algorithm assigns a symbolic value "P" to each output signal and generates an H-mode symbolic test.

Case 2: one subtree

This is a typical case. The processes like flip-flops, multiplexers, registers, logic gates usually contain a single assignment statement along a control path. Such an assignment statement can be represented as $F <= f(X_1, X_2, \ldots, X_n);$. A subtree is constructed to reflect the data relation among the input and output signals. Four kinds of tests can be generated according to the data relation.

- $f(X_1, X_2, \ldots, X_n) = C$ (constant value): An I-mode symbolic test is generated.
- $f(X_1, X_2, \ldots, X_n) = X_i$ (single signal): If F and X_i are same signal, an H-mode symbolic test is generated. Otherwise, an P-mode test is generated.

- $f(X_1, X_2, \ldots, X_n)$ ($n > 1$): If F and $X_{i|i \in \{1, 2, \ldots, n\}}$ are same signal, an A-mode symbolic test is constructed by assigning $X_{j|j=i}$ as "P" and $X_{j|j \neq i}$ as D_j. Otherwise, P-mode tests or A-mode tests will be generated. If $f(X_1, X_2, \ldots, X_n)$ contains a constant value, e.g. $X_{i|i \in \{1, 2, \ldots, n\}} = C$, an A-mode test will be generated by assigning $X_{j|j \neq i}$ as D_j. If $f(X_1, X_2, \ldots, X_n)$ does not contain constant values and also does not contain functions MULT, COMP, SHIFTL, SHIFTR, and SHIFTA, a P-mode test will be constructed for each X_i by assigning $X_{j|j=i}$ as D_i and $X_{j|j \neq i}$ as some fixed value. However, when $f(X_1, X_2, \ldots, X_n)$ contains such functions, only A-mode tests are generated.

Case 3: multiple subtrees

First we will discuss the general case concerning multiple subtrees. Then we explain what the P-algorithm can do. Assume that a process has K input signals $(I_1, I_2, \ldots, I_K)$, M output signals $(O_1, O_2, \ldots, O_M)$ and N inout signals $(IO_1, IO_2, \ldots, IO_N)$, and that a control path contains three assignment statements (S_1, S_2, S_3).

S_1: $F_a <= f_a(X_{a1}, X_{a2}, \ldots, X_{ak});$

S_2: $F_b <= f_b(X_{b1}, X_{b2}, \ldots, X_{bm});$

S_3: $F_c <= f_c(X_{c1}, X_{c2}, \ldots, X_{cn});$

Here, $X_{q \mid q = a1, \ldots, ak, b1, \ldots, bm, c1, \ldots, cn} \in \{I_{h \mid h = 1, 2, \ldots, K}\} \cup \{IO_{j \mid j = 1, 2, \ldots, N}\}$

$F_{p \mid p = a, b, c} \in \{O_{i \mid i = 1, 2, \ldots, M}\} \cup \{IO_{j \mid j = 1, 2, \ldots, N}\}$

f_a, f_b, and f_c specify the logic or arithmetic functions.

Each statement will correspond to a subtree. The data relation within a subtree is the same as Case 2. We will define the data dependence between two assignment statements by using the same notation as what is used in computer architecture when handling a data harzard. Consider two assignment statements S_i and S_j, with S_i occurring before S_j (e.g. S_1 and S_2). There are four possible types of data dependence[73]:

- WAW (write after write) — S_i and S_j assign values to a same signal F $(F \equiv F_a \equiv F_b)$.
- RAR (read after read) — S_i and S_j contain a same signal X in their source expressions $(X_{ai} \equiv X_{bj} \equiv X)$.
- RAW (read after write) — S_i assigns a value to a signal FX and S_j contains this signal in its source expression $(F_a \equiv X_{bj} \equiv FX)$.
- WAR (write after read) — S_i contains a signal XF in its source expression and S_j assigns a value to this signal $(X_{ai} \equiv F_b \equiv XF)$.

It is obvious that $FX, XF \in \{IO_j \mid _{j = 1, 2, \ldots, N}\}$. For RAR, signal X acts as a fanout stem. If both WAW and RAR involving certain signals exist, fanout reconvergency occurs. If both WAR and RAW involving certain signals exist, a loop will appear.

The VHDL compiler (concerning synthesis) reference manual (pp. 6-9 in [48]) states that "If several values are assigned to a given signal in one process, only the *last* assignment is

effective. Even if a signal in a process is assigned, then read, then assigned again, the value read (either inside or outside the process) is the *last* assignment value." and Synopsys synthesis tools will generate a circuit which matches this statement. For example, in following process both OUT1 and OUT2 are '0'.

```
process(S)
begin
   S <= '1';
   OUT1 <= S;
   S <= '0';
   OUT2 <= S;
end process;
```

Thus, for WAW, signal F will be decided by statement S_j. For WAR, signal XF in S_i will be decide by S_j. That means latter statement has effects on the former one. The P-algorithm deals with this case by generating tests for each subtree from the last one to first one. Only when the target signal of a assignment statement is unassigned, the subtree corresponding to the assignment is processed. Otherwise the subtree is abandoned. Also when an input signal is assigned a value, it will remain unchanged. This helps handling the RAR. As for RAW, it means that two assignments can be merged into one. We can move the first statement into a new process if the statement is too complicated.

When generating a test from a subtree, the P-algorithm will assign a test mode to the test. The tests from several such subtrees (corresponding to multiple outputs) will be combined to become one test. These modes are resolved to become a new composite

mode. If they are same, the final mode also is same. Otherwise, the final mode will be A-mode.

4.6.3 FVTP Generation

The algorithm will generate fixed valued test packages for some symbolic tests. If a symbolic test contains an operator corresponding to a function, then the FVTP of the symbolic test will cover the FVTP of this function. That means assigning specific values to the symbolic values according to the fixed valued tests of the function.

If only one D_i exists in a P-mode symbolic test which contains no function operator, the FVTP can be obtained by assigning DC_0 and DC_1 as the values of the D_i. This is because there exist bitwise symmetric structures in the process with such a P-mode test. Each bit of the signal with D_i has the same properties. So applying all *1s* and all *0s* to the signal is equivalent to applying both *1* and *0* to each bit of the signal. Therefore the effect of exhaustive testing is obtained.

If there exists a loop in a process, some symbolic test will contain the symbol P (previous value). When generating the FVTP for such a symbolic test, the algorithm assigns specific values to P. At entity level, E-algorithm will choose a suitable symbolic test to generate such specific values before applying the symbolic test.

4.7 Examples

Figure 4.7 shows the symbolic test sets (left columns) and fixed valued test packages (right columns) for the five processes in Figure 3.3. For example, there are three symbolic tests in "add4" : *(D_6 D_6 DC_0), (D_7 DC_0 D_7),* and *($s(D_8, D_9)$ D_8 D_9).* The first two are P-mode. The third one is A-mode. The time frame for each test is 1. Each FVTP of first two symbolic tests contains 2 fixed valued tests, but the FVTP of the third symbolic test contains 12 fixed valued tests. Figure 4.8 gives the tests of 4-bit up counter in the Figure 4.4

---

portorder: F B A	F	B	A
1 ----(number of frames)	1010	1010	0000
P ----(mode)	0101	0101	0000
2 ----(number of FVTP)	1010	0000	1010
D6 D6 DC0 ----(symbolic test)	0101	0000	0101
1	0010	0001	0001
P	0100	0010	0010
2	1000	0100	0100
D7 DC0 D7	0000	1000	1000
1	0000	0001	1111
A	0001	0010	1111
12	0011	0100	1111
s(D8,D9) D8 D9	0111	1000	1111
	0000	0000	0000
	1111	1111	0000
	1111	0000	1111
	1110	1111	1111

(a) add4

portorder: OP A	OP	A
1	0001	0000
P	0010	0001
6	0100	0011
iD4 D4	1000	0111
	0000	1111
	1111	1110

(b) inc4

```
portorder: OP1 IN2 IN1 S        OP1   IN2   IN1   S
1                               1111  1111  X     1
P                               0000  0000  X     1
2                               1111  X     1111  0
D1 D1 X 1                       0000  X     0000  0
1
P
2
D2 X  D2 0
```

(c) mux

```
portorder: OP3 IN3              OP3   IN3
1                               0000  1111
P                               1111  0000
2
nD3 D3
```

(d) not4

```
portorder: LD D_IN D_OUT CLK CLR    LD  D_IN  D_OUT  CLK  CLR
1                                   1   1111  1111   R    0
P                                   1   0000  0000   R    0
2                                   0   X     P      R    0
1 D5 D5 R 0                         0   X     0000   0    R
1
H
1
0 X P R 0
1
I
1
0 X D0 0 R
```

(e) reg

Figure 4.7 STs and FVTPs of Five Processes for the MUT incadd.

```
portorder: CLR CNT DATA_IN CT LD CLK
1
I
1
1 0000 X X X X
1
```

```
P
2
0 D1 D1 X 1 R
1
H
1
0 P X 0 0 R
1
A
6
0 iP X 1 0 R
```

(a) Symbolic tests

CLR	CNT	DATA_IN	CT	LD	CLK
1	0000	X	X	X	X
0	1010	1010	X	1	R
0	0101	0101	X	1	R
0	P	X	0	0	R
0	0010	X	1	0	R
0	0100	X	1	0	R
0	1000	X	1	0	R
0	0000	X	1	0	R
0	1111	X	1	0	R

(b) Fixed valued tests

Figure 4.8 7 STs and FVTPs of 4-Bit Up-Counter.

Chapter 5. Entity Level Test Generation

This chapter starts from introducing the criterion used by the E-algorithm. Then it discusses the general test methodology. After presenting the forward path generation, backward path generation, and the final test pattern generation, it gives the E-algorithm. Some specific problems such as fanout reconcergency, feedback, and testability enhancement are also discussed. Finally, an example shows the test format generated by the E-algorithm.

5.1 Criterion

The **criterion** used by the entity level test generation algorithm (E-algorithm) is that final test patterns generated for a MUT should contain all the FVTPs for each process of the MUT. This means that as to a process, its inputs and outputs will have the specific values of its FVTPs if all the final test patterns are applied.

Formally, use $<I_i, O_i, S_i, V_i>$ to represent a process P_i in the MUT. Here,

I_i : the input signal set of a process P_i

O_i : the output signal set of a process P_i

S_i : the symbolic test set (STS) of a process P_i

V_i : the set of all the FVTPs of a process P_i

Given $<I_i, O_i, S_i, V_i>$ for each process in a MUT and the connectivity among the processes, the test generation for the MUT involves finding the mappings from the primary inputs to the primary outputs by using all the S_i. Such mappings will transfer required values from inputs to outputs for a process and generate the FVTPs of the process.

5.2 General Methodology

In this section we will show the general methodology for path based test generation and give the rules for selecting the symbolic tests of a process during test generation. What can be done by the E-algorithm will be discussed in other sections of this chapter.

Let us consider a special case in which each process is converted to an equivalent gate level circuit after synthesis and the whole circuit of a MUT is the collection of these sub-

circuits. This is equivalent to the synthesis results of a structural model with the *dont_touch* attribute. If the mappings can generate the FVTs of the V_i for a given process in the inputs and outputs of the sub-circuit, then the faults in the sub-circuit can be detected. For the general case, such a sub-circuit may not be clearly separated but the function of the process can be performed by applying the same FVTs.

Suppose that a MUT contains a process set $\{P_i \mid i=1, 2, \ldots, n\}$ and the corresponding gate level circuit of each P_i has the fault number k_i and the V_i of the process P_i has fault coverage α_i. After connecting all these circuits (assuming no new faults introduced) and applying all the V_i, the detected faults of P_i are at least $k_i\alpha_i$. This is because the originally undetected faults in a process may be detected by the tests for exercising other processes. The fault coverage α of the whole circuit is : $\alpha \geq \sum k_i\alpha_i / \sum k_i \geq min\{\alpha_i\}$. Therefore it is reasonable to generate the tests for a MUT by using the criterion stated in 5.1.

Let P_i be a process in a MUT. Without losing generality, suppose that $O_i = \{y_i\}$ and $I_i = \{c_i\} \cup \{x_i\}$. Here signal c_i is a control input of P_i and signal x_i is a data input of P_i. Signal y_i is the data output of P_i. Our goal is controlling the values of c_i and x_i to exercise V_i and observing the value of y_i at the same time. If c_i and x_i are primary inputs (PIs), then any values can be applied. Therefore there is no problem to exercise V_i. If y_i is a primary output (PO), we can observe the effects of any tests on this signal. For the general case,

we assume that c_i and x_i are not PIs and y_i is not a PO. Figure 5.1 shows the process connections in the MUT.

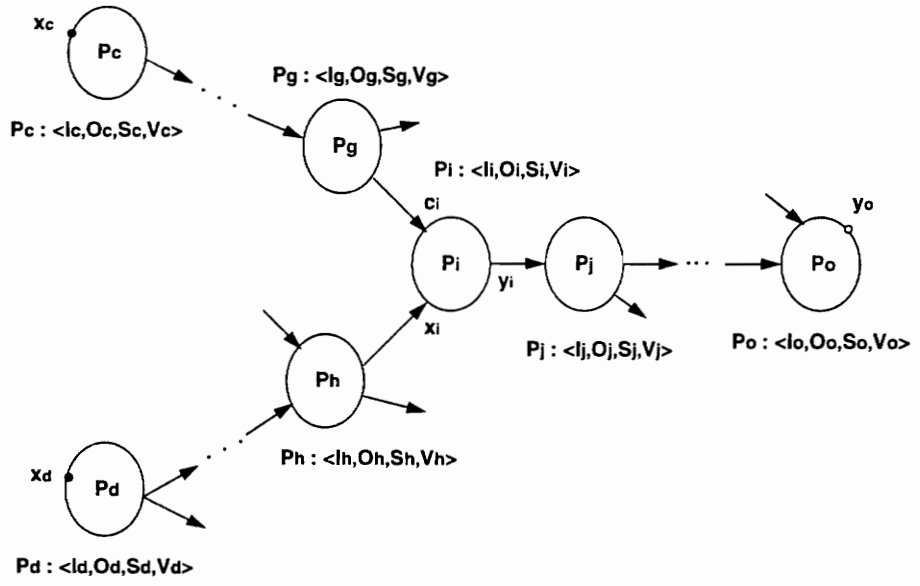


Figure 5.1 Process Connection Illustration.

Because the symbolic test set of a process contains all possible data paths (known from DFGs) and control paths (known from CFG) within the process, we can construct a control path from P_C to P_i to control the state of c_i and a data path from P_d to P_i to set the values of x_i. The way to choose the specific values of c_i and x_i is decided by V_i. Also we can construct a data path from P_j to P_O to observe signal y_i by propagation. Next we show how to construct such paths.

The signal y_i is also an input of process P_j. So there are $y_i \in I_j$, $I_j = \{y_i\} \cup I_j'$, and $\{y_i\} \cap I_j' = \Phi$. We can select a symbolic test from S_j that will create a path from y_i to y_j ($y_j \in O_j$). All the values of y_i can be propagated to y_j. Starting from y_j, we can get a data path to propagate y_j to the output of the succeeding process. Upon reaching the process P_O, a data path from y_i to y_o is completed. So y_i can be observed on the primary output y_O ($y_O \in O_O$).

A data path from a primary data input to x_i can also be constructed by a similar way through justification. The input signal x_i of process P_i is an output of process P_h. By creating a data path in P_h with x_i as one end and $x_h \in I_h$ as the other end, we can control x_i by assigning x_h to a specific value. This procedure continues until a process like P_d with a primary input x_d is reached. We get a series of paths through which x_i can be set to the desired values.

The control signal c_i can be set to the desired states by event justification (such as clock signal in a flip-flop) or constant justification (such as selection signal in a multiplexer).

Now let's discuss the properties of the data paths in the processes that are along a path from a primary input to a primary output. The data path of a process creates a mapping from its inputs to outputs. For propagating the fault effects of preceding processes, the different inputs of the process should have different outputs. This means the mapping

should be one-to-one. For justification, each output should have a corresponding input. That means the mapping should be onto. Combining the above results, we can conclude that these mappings should be one-to-one and onto, which means that P-mode tests are needed.

However a mapping without such properties may also propagate the fault effects and implement justification. For example, no one-to-one mapping exists in a comparator. It still can propagate fault effects by carefully selecting the input values. If one input is in a data path we can set the other input as the same value. Then if a fault changes the value along the path, the fault effect can be observed at the output of the comparator. Also for multiplier, the propagation can be done from one input to the output by setting the other input to a non zero value. But the justification may be impossible for some values.

There are two possible approaches to handling the cases in which a process does not have a one-to-one and onto mapping (P-mode test). The first approach is improving the process during modeling to make it obtain such a mapping. Another approach is using A-mode tests to do propagation and justification. We will discuss the first approach later by introducing testability enhancement. The second method will be discussed in Chapter 8.

There may exist path conflicts if a MUT has fanout reconvergency. There are two cases that will create fanout branches. One is that a signal from a process is connected to more

than one process. The other is that there are two or more outputs in a process and there exists a fanout point inside the process that will control at least two outputs. Such branches may meet at a process, which will result in fanout reconvergency and can cause data path or control path conflicts. When such conflicts occur, a mechanism is needed to resolve them. We will discuss how to resolve fanout reconvergency later.

In following sections, we will discuss how the E-algorithm generates the forward data path from the process under test to an primary output and backward signal paths from primary inputs to the process. How to dealt with conflicts and global loops are also presented. After giving final test pattern generation method, we will show the E-algorithm completely. We will conclude this chapter by showing a few examples. But first, we will explain some concepts concerning E-algorithm.

Test patterns for a unit are composed of *time frames*. Each time frame contains all the signal values at a given time. To say that signal values occurred in two consecutive time frames of a test pattern means that the signal values in first time frame will be applied first and the signal values in second time frame will be applied next when the test patterns are used.

A symbolic test pattern of a unit is a time frame sequence which contains required time frames while constructing forward and backward paths for a given symbolic test of a

process. *Final test patterns* corresponding to a symbolic test pattern are the time frame sequences obtained after replacing the symbolic values in the symbolic test pattern by specific values. Each symbolic value may have several specific values, so a symbolic test patterns may repeat several times during such replacement.

5.3 Forward Path Generation

Given a symbolic test (ST) of a process under test, the forward path generation procedure tries to construct a signal path that will make an output value in the symbolic test observable at a primary output. Figure 5.2 shows the procedure. It consists of two steps. First, the ST is assigned to a time frame. Then an output value of the ST is propagated to a primary output along a path. If conflicts exist during propagation, backtracking is performed to select another available path. If backtracking fails, the forward path generation fails. Whenever a symbolic test is selected and assigned to a time frame, it will be assigned an unique label called *level* which represents the assigning sequence of a symbolic test. A smaller level value implies the symbolic test with this level is assigned earlier * and a larger level value means later. The level values are used to help remove some symbolic tests from time frames while backtracking.

5.3.1 Applying

* In the test generation process.

Depending on the properties of the ST, the procedure decides how to apply the ST. When the ST contains *iP*, implying that a statement like *"A<=INC(A);"* exists in the process, the procedure searches the symbolic test set of the process for a P-mode test which can provide required signal values. In Figure 4.4 upcnt4 (also referring Figure 4.8 for the tests of upcnt4), while applying *(0 iP X 1 0 R)*, the procedure selects *(0 D_1 D_1 X 1 R)* to provide previous values and generates two time frames with the following format (only showing related signal values):

CLR	*CNT*	*DATA_IN*	*CT*	*LD*	*CLK*
0	*D_1*	*D_1*	*X*	*1*	*R*
0	*iD_1*	*X*	*1*	*0*	*R*

The symbol *P* in second time frame is replaced by the corresponding symbol D_1 in first frame. Any value required by *iP* can be assigned by setting appropriate D_1. Sometimes another H-mode test may exist in the symbolic test set. In above example, such an H-mode test is *(0 P X 0 0 R)*. The procedure will generate the third time frame to contain the P-mode test, i.e., *(0 iD_1 X 0 0 R)* will be the third time frame. By recording the P-mode and H-mode tests as used, these two symbolic tests will not be applied later.

When the ST contains *dP (A <= DEC(A);)*, a similar procedure is performed. When neither *iP* nor *dP* is included in the ST, the ST will be assigned to one time frame. After applying the ST, the forward path generation procedure continues to the next step.

5.3.2 Propagation

A *forward backtracking stack* is maintained to store the information concerning the connection paths between two processes during propagation. When a symbolic test of a process is selected and assigned to a time frame, all the output ports of the process with valid symbolic values are stored in the stack together with the destination port and process. Each record in the stack represents a possible propagation path between two processes.

Given the input port (or propagation port) and the value to be propagated, the propagation procedure selects a symbolic test from the symbolic test set of the process containing the input port so that the value can be propagated to the output of the process. There are two cases encountered while selecting such a symbolic test. One is that the value of the input port in the symbolic test matches the value to be propagated. The other case is that the symbolic test is P-mode and the value in the port is D_i.

For first case, if other values in the symbolic test are consistent with the corresponding values in the given time frame, the symbolic test can be chosen as required one. For the second case, the procedure performs a forward symbolic substitution to replace the D_i by the symbolic expression to be propagated. While performing such substitutions, each

symbolic value containing D_i is replaced. The symbolic test obtained through performing symbolic substitutions is also checked for consistency.

After getting such a symbolic test and passing the consistency check, the procedure assigns the symbolic test to a time frame. Depending on the sensitivity of the input port, a new time frame may be created. If the input port is in the sensitivity list of the process, the assignment will take place in the current time frame. However if the port is non sensitive, a new time frame is created and the assignment is performed in this new time frame. This reflects the semantic meaning of VHDL with regard to process sensitivity lists.

The above procedure will repeat until a primary output port is reached or no symbolic test can be chosen for further propagation.

5.3.3 Backtracking

Backtracking is performed when there exists inconsistency between the signal values to be assigned to a time frame and the values already assigned. Such inconsistency usually occurs at fanout branches. Another case which causes backtracking is that there exists no suitable symbolic test in the symbolic test set of a process so that the signal value of the propagation port can not be propagated to an output port of the process. For example,

there may not exist a symbolic test which contains a port value or is a P-mode test with D_i as the port value.

Backtracking starts from finding a backtracking point in the forward backtracking stack. Such a backtracking point can be located at current process or a previous one. If it is located at current process, a new symbolic test is selected and propagation continues. If the backtracking point is located at previous process, all the signal values with higher levels will be removed from the time frames. The symbolic test corresponding to this point is assigned to a time frame and the propagation proceeds from this point.

Through backtracking, different propagation paths are selected in order to reach a primary output. Here the procedure just considers single path propagation. If all the single paths are tried but it is still impossible to reach a primary output, the propagation procedure fails. Therefore the symbolic test to be applied can not result in a symbolic test pattern for the unit. However, successful forward path generation will let the procedure go to the next procedure, backward path generation.

Activate (ST of the process under test)
begin
*/*applying the ST*/*
  if ST contains iP or dP **then**
    search for a P-mode test to supply required values of P;
  end if;
  if such a P-mode test exists **then**

```
      apply the P-mode test to first time frame using Assign_valuep;
      apply the ST to second time frame;
      if there exists H-mode test then
        apply the H-mode test to third time frame;
      end if;
    end if;
    if no such P-mode test exists or the ST contains no iP or dP then
      only apply the ST to current time frame;
    end if;
/*propagation to an primary output*/
    while do not reach PO do
      select next process and port for propagation;
      put other propagation processes and ports into forward backtracking stack;
      select an ST from STS of next process by Select_tst_set_f;/*including backtracking*/
      if such an ST exists then
        apply the ST;
      else
        abort the propagation.
      end if;
    end while;
end Activate;


Assign_valuep (process, propagation port, ST)
begin
  for each port in the process do
    if the port signal needs to justify then
      set justification stack corresponding to the signal;
    end if;
    if no conflict exists then
      assign port value to the signal in current time frame;
    else
      set conflict flag;
    end if;
  end for;
end Assign_valuep;


Select_tst_set_f (process, propagation port and value to be propagated)
begin
  while there exists ST untried in the process do
    get the ST;
    if the value of propagation port in the ST matches the value to be propagated then
```

```
        check for consistence;
        if consistent then
          select next process and port for propagation;
          put other propagation processes and ports into forward backtracking stack;
          return the ST as required ST;
        end if;
      else if  the ST is P-mode and the value corresponding to propagation port is Di then
        perform forward symbolic operation; /* SymbolicOperation_f() */
        check for consistence;
        if consistent then
          select next process and port for propagation;
          put other propagation processes and ports into forward backtracking stack;
          return the ST as required ST;
        end if;
      end if;
    end while;
    if  forward backtracking stack non empty then/*backtracking*/
      while there exists backtracking point at same level do
        get the values corresponding to this propagation point;
        recursively call Select_tst_set_f to select ST;
        if successful then
          return selected ST;
        end if;
      end while;
      while there exists backtracking point at lower level do
        remove signal values with higher level from the time frames within same group;
        remove the empty time frames;
        get the values corresponding to this propagation point;
        recursively call Select_tst_set_f to select ST;
        if successful then
          return selected ST;
        end if;
      end while;
      backtracking failed;
    end if;
    selecting ST failed;
end Select_tst_set_f;


SymbolicOperation_f (ST to be modified, port and value to be propagated)
begin
  get the symbolic value DI in the ST corresponding to the port;
```

```
  if DI ≠ the value to be propagated then
    for each port do
      if the symbolic test contains DI then
        replace the DI by the value to be propagated;
      end if;
    end for;
  end if;
  return modified ST;
end SymbolicOperation_f;
```

Figure 5.2 Forward Path Generation Procedure.

5.4 Backward Path Generation

5.4.1 Justification

Each signal in a time frame has a *justification flag*. When the flag is set during assigning values in the propagation procedure or the justification procedure itself, the signal will be justified later. Like forward path generation, the backward path generation procedure also maintains a *backward backtracking stack* which stores the complete information that can be used to form an alternative justification path. Such information includes source port and destination port of the signal to justify and a valid symbolic test which can be used to construct the justification path.

The justification procedure justifies the signal values in the same group of time frames from the last frame to first one. Some signals in a frame may be justified more than once depending on other justification results. If a signal needs to be justified, the source port (justification port) of the signal and the process containing the port will be found first. Then a suitable symbolic test is selected from the symbolic test set of the process to create a path from an input port of the process to the source port.

Symbolic Test Selection

While selecting a symbolic test for justification, the procedure checks all the symbolic tests of the process. A symbolic test is suitable for justification if it meets one of the following conditions:

- the value of justification port equals the value to be justified;
- it is P-mode test and the symbolic value of justification port is D_i;
- the symbolic value of justification port is *iP* or *dP* and the value to be justified is constant.

For the first case, if other values in the symbolic test are consistent with the values in the time frame which the symbolic test is assigned to, the symbolic test is put into the backward backtracking stack. For second case, the backward symbolic operation is performed to make the justification port have the value to be justified. After such

operation, the modified symbolic test is checked for consistency and if consistent, the modified symbolic test is put into the stack. As for last case, because a constant value can be obtained by performing increment or decrement from some initial value, the test can also be used for justification if no inconsistency exists. After checking all the symbolic tests, any symbolic test which can be used for justification is put into the backward backtracking stack.

Symbolic Test Assignment

When assigning a selected symbolic test to a time frame, the procedure needs to decide if any other symbolic tests are necessary to help complete such an assignment. For above first two cases, the procedure only assigns a symbolic test to a time frame. If the justification port is connected to a sensitive port, the time frame is current time frame. Otherwise if the justification port is connected to a non sensitive port, the time frame will the previous one within same group. If no such a frame exists, a new frame will be created to accept the symbolic test.

If the symbolic test to be assigned belongs to the third case, the procedure tries to find an I-mode test in the same symbolic test set. Assume that each sequential process contains a reset signal so such an I-mode exists. If the symbolic value at the justification port is *iP*, the procedure assigns the I-mode test first to create an initial value and then repeatedly

assign the symbolic test containing *iP* until the value to be justified is reached. The number of the time frames containing the symbolic test is decided by the difference between the initial value and the value to be justified. For the cases where the symbolic test contains *dP* as the value at the justification port, if there exists a suitable I-mode test, the procedure performs in the similar way as the *iP*. Otherwise, the procedure will search for some tests in the same test set which can be combined with the symbolic test to make the signal value at the justification port become the value to be justified. Multiple time frames are needed to apply these tests.

Such selection and assignment processes are repeated until all the signals to justify reach primary inputs. However there exist some cases in which conflicts occur. If such conflicts can be resolved through backtracking, the justification procedure continues. Otherwise the justification fails.

5.4.2 Backtracking

When selecting symbolic test for justification fails, the procedure performs backtracking. A backtracking point is obtained from the backward backtracking stack. Such a backtracking point gives a valid symbolic test ST, the justification port, and a level value related to previously assigned symbolic test during justification. The procedure removes all the signal values with higher or same level values from the time frames within the

same group. Such signal values are assigned in previous justification assignments. After removing the signal values, the procedure assigns the symbolic test ST to an appropriate time frame. Then justification procedure continues as before. The justification procedure may fail if the backtracking stack becomes empty but some conflicts remain unresolved.

Successful justification means that the backward path generation succeeds. Therefore a final symbolic test pattern is generated. If justification fails after backtracking, there still exists chance to generate a test pattern. This happens when the forward backtracking stack is not empty. The procedure selects another forward path and if successful performs justification. This process will be discussed later.

```
Justify()
begin
  for each time frame within current group do
    set justify_flag to 1;
    while justify_flag equals 1 do
      clear justfy_flag;
      for each signal do
        if the signal needs to be justified then
          find source port of the signal and the process containing the port;
          if the signal is connected to non sensitive port then
            create new time frame for setting justified values;
          end if;
          select ST from the STS of the process for justification through
          using Select_tst_set_b;
          if don't assign any value during selecting ST  then
            assign signal values to the time frame using Assign_valuej;
          else
            set justify_flag to 1;
            exit current loop and restart justification from first signal;
```

```
            end if;
          end if;
        end for;
      end while;
    end for;
end Justify;


Select_tst_set_b()
begin
  while there exists ST untried in the process do
/*put all STs which can be used for justification into backward backtracking stack*/
      get the ST;
      if the value of justification port in the ST matches the value to be justified  then
        check for consistence;
        if consistent then
          put the ST and related information into backward backtracking stack;
/*information: level, port, et al */
        end if;
      else if  the ST is P-mode and the value corresponding to justification port is Di then
        perform backward symbolic operation using SymbolicOperation_b;
        check for consistence;
        if consistent then
          put the ST and related information into backward backtracking stack;
        end if;
      else if  the value corresponding to justification port in the ST is iP or dP then
        if the value to be justified is constant then
          check for consistence;
          if consistent then
            put the ST and related information into backward backtracking stack;
          end if;
        end if;
      end if;
  end while; /*end of searching ST*/
  if put any ST into backward backtracking stack in above loop then
    return the ST from the top of the stack as selected ST for justification;
  else if  backward backtracking stack nonempty then
    get an ST at some level from the stack;
    remove signal values with higher or same level from time frames within same group;
    assign signal values to time frame using Assign_valuej;
  else
    selecting ST failed;
```

```
  end if;
end Select_tst_set_b;


Assign_valuej (justification process, justification port, ST)
begin
  if symbolic value at justification port is "iP" then
     find an I-mode test in STs of the justification process;
     create multiple time frames;
     assign initial value to the justification port using the I-mode test;
     assign other values to the justification port using the ST until reaching
     the value to be justified;
     return successfully.
  end if;
  if symbolic value at justification port is "dP" then
     find a symbolic test with "iP" at the justification port;
     find an I-mode test;
     create multiple time frames;
     assign initial value to the justification port using the I-mode test;
     assign other values to the justification port using the test with "iP"
     until reaching the value to be justified;
     assign value to justification port using the ST;
     return successfully.
  end if;
  for each port in the process do/*ST contains neither iP nor dP*/
     if the port signal needs to justify then
        set justification stack corresponding to the signal;
        set justify_flag to 1 so that justify will repeat;
     end if;
     if no conflict exists then
        assign port value to the signal in current time frame;
     end if;
  end for;
end Assign_valuej;


SymbolicOperation_b (ST to be modified, port and value to be justified)
begin
  get the symbolic value DI in the ST corresponding to the port;
  if DI ≠ the value to be propagated then
     if the DI is Di then
        for each port do
```

```
        if the symbolic test contains the DI then
          replace the DI by the value to be justified;
        end if;
      end for;
    else if the DI is nDi, iDi, or dDi then
      for each port do
        if the symbolic test is DI then
          replace the DI by the value to be justified;
        else if the symbolic test contains only Di within DI then
          replace the symbolic test by the value to be justified and inverse operator;
        end if;
      end for;
    end if;
  end if;
  return modified ST;
end SymbolicOperation_b;
```

Figure 5.3 Backward Path Generation Procedure.

5.5 Handling Fanout Reconvergency and Feedback

Fanout Reconvergency

Fanout reconvergency can be resolved through three ways. First, eliminate some branch paths if possible. Multiplexers and registers added when modeling can help to do so by providing different paths or keeping previous values. Second, backtrack to select another path. Third, do multiple path propagation. Figure 5.4 illustrates these cases.

```
P3: process(I, H)
begin
  J<= ADD(I, H);
end process P3;
```

```
P2: process(D, E,G)
begin
  if D='1' then
     F<=E;
  else
     F<=G;
  end if;
end process P2;

Symbolic test set(P2):
D  E   G   F
1  D1  X   D1
0  X   D2  D2

P2_0: process(D, E,G)
begin
  if D='1' then
     F<=G;
  else
     F<=INC(G);
  end if;
end process P2_0;
```

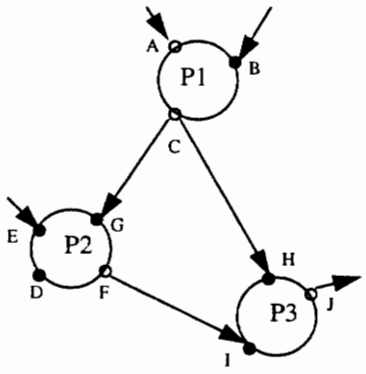


Figure 5.4 Fanout Reconvergency Example.

Take *P1* as the process to test. Assume a symbolic test of *P1* makes $C = D_1$. We try to propagate the value D_1 to a PO. According to our algorithm, propagation is done along a single path first. First we select path *(C, G, F, I, J).* We should set *(D, G)* as *(0, D_1)* and

H as DC_0 according the test sets of $P2$ and $P3$. But the value of H is D_1 so conflict occurs. Next we choose path (C, H, J) and set H as D_1 and I as DC_0. To justify DC_0 of signal I, we need to assign symbolic tests to the ports of $P2$. A successful path is obtained by assigning (D, E) as $(1, DC_0)$. Now let us consider another case in which the process $P2$ is replaced by process $P2\_0$. Then single path propagation is impossible. The only choice is multiple path propagation by selecting A-mode test $(H, I, J) = (D_1, D_2, s(D_1, D_2))$. Here, $D_2=D_1$ or $D_2=iD_1$. At present the multiple path propagation is not considered by E-Algorithm.

Feedback

The PMG of a MUT gives a special notation (external port) to a primary output port which will feedback to an internal port. The E-algorithm uses the notation to detect a global feedback loop.

It is not necessary to consider feedback loops while constructing forward path, because the E-algorithm will finish the path generation when such a primary output port is reached. However when generating a backward path through justification, such feedback loops need to be detected. The E-algorithm checks the status of a signal and decides whether the signal is external. If the source port (a primary output) of an external signal has been assigned the value which equals the value to be justified, justification succeeds.

Otherwise if the two values are not equal, the justification fails. When the source has not been assigned a value, justification will continue from the source port.

5.6 Final Test Pattern Generation

After successfully constructing forward and backward paths for a given symbolic test of a process, the E-algorithm gets a symbolic test pattern for the MUT. Each test pattern may include one or more time frames in which signal values are represented by symbolic expressions. The final test patterns are obtained through evaluating such expressions for various fixed values related to the symbolic test of the process.

Here are the basic symbols used in our test generation system:

- unary operator: *i* (increment), *i'* (inversion of increment), *d* (decrement), *d'* (inversion of decrement), *n* (not), *t* (arithmetic right shift); There exists that $i' = d$ and $d' = i$.
- binary operator: *o* (or), *a* (and), *e* (xor), *s* (addition), *m* (multiplication), *l* (left shift), *r* (right shift), *c* (comparison);
- bit constant value: *0, 1, R, F, X, Z*
- vector constant value: DC_1 (all *1s*), DC_0 (all *0s*), *X, Z*, combinations of *1s* and *0s*
- variable value: $D_{i\ (i=1,\ 2,\ 3,\ \ldots),}\ P$
- parenthesis: *( )*

A symbolic expression is composed of such basic symbols. It may contain a single symbol or multiple symbols. Followings are some examples.

- single symbol expressions: *1, 0, F, X, DC0, D5, P, 0000, 010, 11110, ...*
- multiple symbol expressions: iD_3, $d'D_1$, $i'ndD_2$, $s(D_1, D_2)$, $m(D_1, s(iD_2, D_3))$, ...

Figure 5.5 shows the final test pattern generation procedure. The symbolic test ST applied at the beginning of the symbolic test generation and the final symbolic test pattern generated are taken as the inputs of the procedure. The procedure gets a fixed valued test from the fixed valued test package of the symbolic test. A specific value for each symbol in the symbolic test ST is derived from the fixed valued test. Starting from the first time frame to the last one of the symbolic test pattern, the procedure calculates the symbolic expressions of each signal value to get a final fixed valued test pattern. Each fixed valued test will result in such a final test. Therefore a sequence of final test patterns are generated corresponding to the fixed valued test package of the symbolic test ST.

There are two steps to get the value from a symbolic expression: analysis and calculation. Different kinds of symbols contained in the expression are separated by analysis. Based on the operators obtained from the expression, different calculations are performed. Both steps are performed recursively. Now let us take expression $s(iD_1, D_2)$ as an example.

Assume that *"0101"* and *"0001"* are derived for D_1 and D_2 respectively from a fixed valued test. By parsing the expression, we get an operator 's' (addition). Then addition is performed. Because *'s'* requires two operands, iD_1 is taken as first operand and D_2 as second one. Here iD_1 still contains other operator. After recursive calculation, *"0110"* is returned as the value of iD_1. Now both operands of *'s'* have constant values. The final result *"0111"* is obtained after performing addition.

```
PatternGen(symbolic test applied)
  while FVTP of the symbolic test is not empty do
    get an FVT from the FVTP;
    for each time frame within same group do
      for each signal in the time frame do
        get test pattern by computing symbolic expression of the signal value;
       /*Get_exp()*/
      end for;
      assign value to X;
      append the test patterns  to test file;
    end for;
  end while;
```

Figure 5.5 Final Test Pattern Generation Procedure.

5.7 E-Algorithm

The objective of the E-algorithm is trying to generate final test patterns for the MUT that will contain all the fixed value tests for each process. The E-algorithm includes following steps and the details are shown in Figure 5.6.

1. Initialize processes
2. Select a process to test
3. Get an FVTP and the corresponding symbolic test
4. Propagate and justify the symbolic test to generate a symbolic entity test
5. Apply the FVTP to the symbolic entity test to form fixed valued entity test patterns
6. Repeat until all FVTPs of the processes in the MUT are applied

Most of the steps above have been discussed in the previous sections. Now we just explain first two steps. At the beginning the sequential processes need to be initialized. The E-algorithm selects an I-mode test for each sequential process to do initialization. If the input signals of the process are not PIs, then the algorithm does justification.

Some processes such as buffers are not necessary to test because synthesis tools will map them to wires. The tests for a complex process may cover the tests for other processes along the same data path. So the algorithm only selects the processes needed to test.

E-algorithm(analyzed MUT):

```
begin
  load PMG;
  load STS and FVTPs of each process and set P-queue;
/*A-mode tests or P-mode tests containing function are put into P-queue during loading*/
  initialize sequential processes; /* Initialize_Modules();*/
  while there are processes to test do
    select a process from P-queue;
    while there exists FVTP to apply do
      get the A-mode (or P-mode) ST corresponding to the FVTP;
      set flags; /*applied; level; group id*/
      clear forward backtracking stack and backward backtracking stack;
      create forward path; /*Activate()*/
      if fail to create forward path then
        break the test generation process for this ST;
      end if;
      construct backward path;/*justify()*/
      if fail to construct backward path then
        while forward backtracking stack non empty do /*backtracking*/
          backtrack to find another forward path; /*backtrack()*/
          if fail to create forward path then
            break the test generation process for this ST;
          end if;
          construct backward path again; /*justify()*/
          if success to construct backward path then
            exit this loop and stop backtracking;
          end if;
        end while;
      end if;
      if data path generation successful then
/* both forward and backward path complete */
        update flags;
        construct a symbolic test pattern by arranging the symbolic values;
        generate final fixed valued test patterns; /*PatternGen()*/
      end if;
    end while;
  end while;
  while exist unapplied FVTPs in some process do /*clear up*/
    generate tests for the FVTPs using the same procedure;
  end while;
end E-algorithm;
```

Figure 5.6 E-Algorithm.

```
backtrack()
begin
  get backtracking point from top of forward backtracking stack;
  remove signal values with higher level from the time frames within same group;
  get value to be propagated and port corresponding to this point;
  select an ST from STS of this process; /*Select_tst_set_f()*/
  apply the ST;
  while do not reach PO do
    get next process and port for propagation;
    select an ST from STS of next process; /*Select_tst_set_f()*/
    if such an ST exists then
      apply the ST;
    else
      abort the propagation.
    end if;
  end while;
end backtrack;
```

Figure 5.7 Backtracking Procedure.

5.8 Example

Figure 5.8 gives the symbolic entity tests for the MUT *incadd* in Figure 3.3. The frames indicate the test sequence for each test. Here frame 0 is for initialization. Frames 1 and 2 are obtained after propagating and justifying the symbolic test $(iD_4 \;\; D_4)$ of process *inc4*. Blank positions will be filled as previous values. Frames 3 and 4 are for testing process *add4*. Frames 5 and 6 give the tests for applying the test $(D_2 \;\; X \;\; D_2 \;\; 0)$ of process *mux*. Each test is assigned an *"id"* number. The other symbolic test of the process *mux* and the

symbolic tests of process *not4* and *reg* are not necessary to apply because they have be covered by the symbolic entity tests for other processes. Here operator *i'* means inverse operation of operator *i*. That is *i'* implements decrement operation.

Unit name is incadd
No. of signals = 12
A complete test sequence for model incadd:

frame	id	OP3	IN3	OP	OP1	IN2	IN1	S	F	LD	D_OUT	CLK	CLR
0	0								X	0	D0	0	1
1	1	D4	nD4	iD4	DC0	DC0	X	1	iD4				
2	1								iD4	1	iD4	R	0
3	2	i'D8	ni'D8	D8	D9	D9	X	1	s(D8,D9)				
4	2								s(D8,D9)	1	s(D8,D9)	R	0
5	3	i'DC0	ni'DC0	DC0	D2	X	D2	0	D2				
6	3								D2	1	D2	R	0

Figure 5.8 Symbolic Tests for the Model incadd.

Each symbolic entity test for the MUT contains Di belonging to the symbolic test of a given process. By applying the corresponding FVTP, several final test patterns can be obtained. For example, a total of 12 final test patterns are obtained through one symbolic test (frame 3 and 4). Now let's see how to apply the first fixed valued test *(0010 0001 0001)* in Figure 4.8 (a). The E-algorithm calculates the value of IN3: $ni'D_8 = ni'(0001) = n(0000) = 1111$. Other PIs can be assigned directly.

Unit name is incadd
A complete test sequence for model incadd:

frame	id	OP3	IN3	OP	OP1	IN2	IN1	S	F	LD	D_OUT	CLK	CLR
0	0								X	0	0000	0	1
1	1	0000	1111	0001	0000	0000	X	1	0001				
2	1								0001	1	0001	R	0
1	1	0001	1110	0010	0000	0000	X	1	0010				
2	1								0010	1	0010	R	0
1	1	0011	1100	0100	0000	0000	X	1	0100				
2	1								0100	1	0100	R	0
1	1	0111	1000	1000	0000	0000	X	1	1000				
2	1								1000	1	1000	R	0
1	1	1111	0000	0000	0000	0000	X	1	0000				
2	1								0000	1	0000	R	0
1	1	1110	0001	1111	0000	0000	X	1	1111				
2	1								1111	1	1111	R	0
3	2	0000	1111	0001	0001	0001	X	1	0010				
4	2								0010	1	0010	R	0
3	2	0001	1110	0010	0010	0010	X	1	0100				
4	2								0100	1	0100	R	0
3	2	0011	1100	0100	0100	0100	X	1	1000				
4	2								1000	1	1000	R	0
3	2	0111	1000	1000	1000	1000	X	1	0000				
4	2								0000	1	0000	R	0
3	2	0000	1111	0001	1111	1111	X	1	0000				
4	2								0000	1	0000	R	0
3	2	0001	1110	0010	1111	1111	X	1	0001				
4	2								0001	1	0001	R	0
3	2	0011	1100	0100	1111	1111	X	1	0011				
4	2								0011	1	0011	R	0
3	2	0111	1000	1000	1111	1111	X	1	0111				
4	2								0111	1	0111	R	0
3	2	1111	0000	0000	0000	0000	X	1	0000				
4	2								0000	1	0000	R	0
3	2	1110	0001	1111	0000	0000	X	1	1111				
4	2								1111	1	1111	R	0
3	2	1111	0000	0000	1111	1111	X	1	1111				
4	2								1111	1	1111	R	0
3	2	1110	0001	1111	1111	1111	X	1	1110				
4	2								1110	1	1110	R	0
5	3	1111	0000	0000	1111	X	1111	0	1111				
6	3								1111	1	1111	R	0
5	3	1111	0000	0000	0000	X	0000	0	0000				
6	3								0000	1	0000	R	0

Figure 5.9 Final Test Patterns of the Model incadd.

5.9 Testability Enhancement

In order to propagate or justify a signal, we hope to get at least one P-mode test for each process. For some processes, as such a test may not exist. Sometimes it is possible to get a P-mode test by inserting additional statements. For example, the function MULT does not have P-mode test. We can modify a multiplier process as follows.

```
mux_mult: process(SEL,B,A)
begin
if SEL='1' then
D_OUT<=A&B;
else
D_OUT<=MULT(A,B);
end if;
end process mux_mult;
```

The operation & merges two signals to a single one. We can get a P-mode test *(OUT, A, B, SEL)* = $(D_1\&D_2, D_1, D_2, 1)$ *from* the operation. At entity level, we can also insert a process like multiplexer to increase the testability of whole model. Another method is creating a P-mode test while constructing a function if possible.

Chapter 6. Test Evaluation System

A general introduction to the CAD systems used in our test generation system was presented in Chapter 3. This chapter shows how to implement the test evaluation by using these CAD systems.

6.1 Gate Level Circuit Generation

Using Synopsys synthesis tools, one can get several equivalent circuits under the different optimization constraints for the same VHDL model. Figure 6.1 shows a design space of possible circuits. Two kinds of synthesized circuits are used here to assess the effectiveness of the test patterns generated by the P-algorithm and E-algorithm. One is the smallest circuit (smallest area). The other is the fastest one (smallest propagation delay). Such circuits can be obtained by setting optimization constraints *max_area* and *max_delay*. However we will evaluate most of models using a single optimized circuit.

Figure 6.2 gives a script file used in the Synopsys Design Compiler to get a gate level circuit for the model upcnt4. Two equivalent gate level circuits of another VHDL model are presented in Figure 6.3.

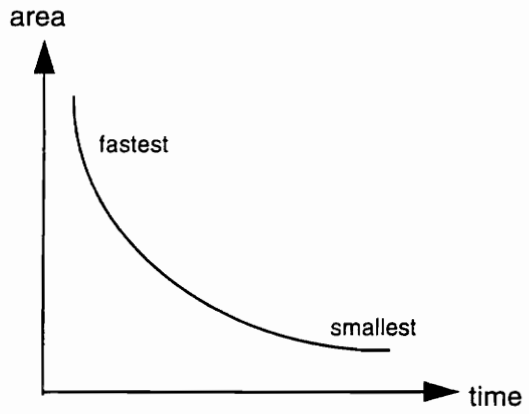
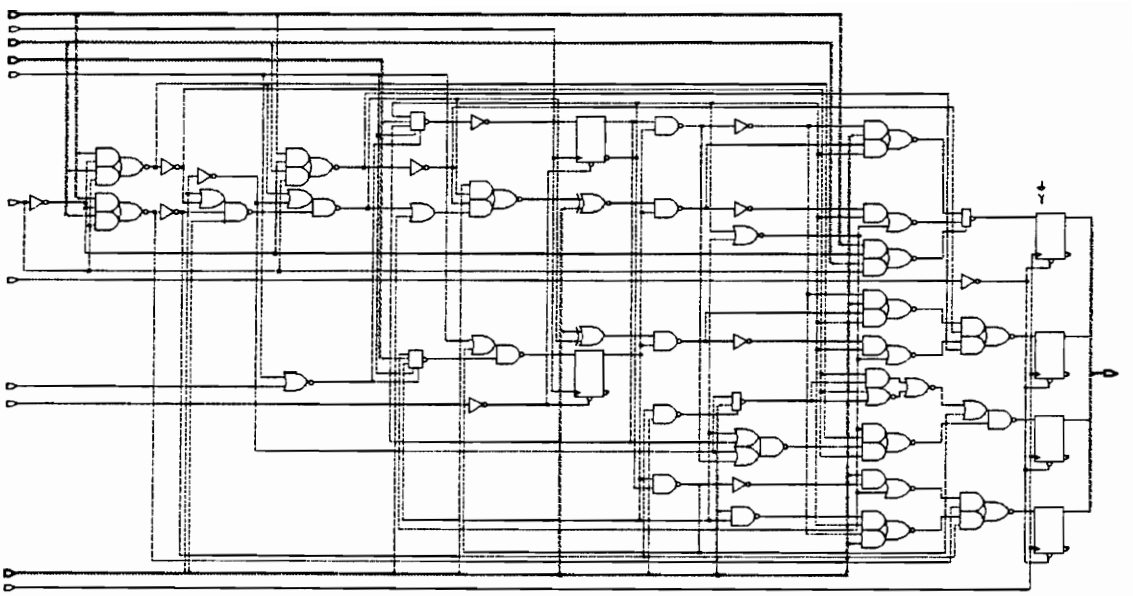


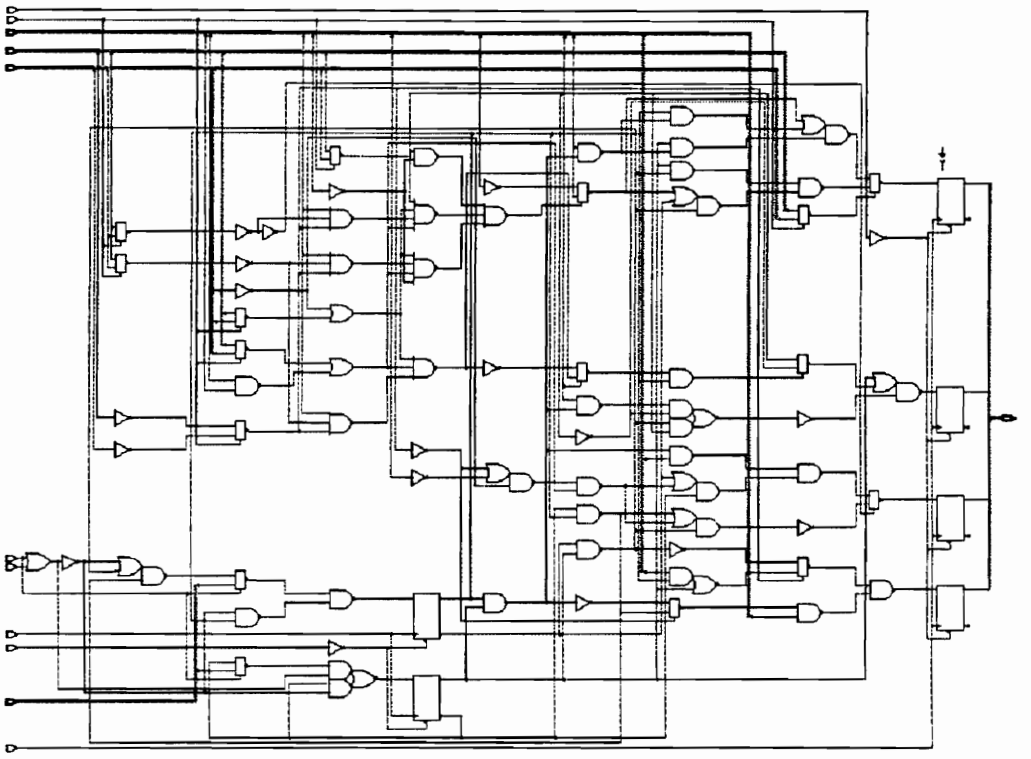
Figure 6.1 Design Space.

```
des= upcnt4
set_variable_edifout_netlist_only="true"
/* These two packages only are analyzed once.*/
/*analyze -format vhdl  VHDLCAD.vhd
analyze -format vhdl  USER_TYPES.vhd.vhd*/
analyze -format vhdl  upcnt4.vhd
elaborate des
remove_constraint -all
create_clock -name CLK
set_default_flip_flop_type FD2
current_design =  upcnt4
compile -map_effort medium -boundary_optimization -ungroup_all
report_timing > "upcnt4.timing"
write -format edif -hierarchy -output "upcnt4.edif"
exit
```

Figure 6.2 A Script File (file of dc_shell commands).



(a) Smallest design of model mpu1(132 primitive gates).



(b) Fastest design of model mpu1(170 primitive gates).

Figure 6.3 Two Equivalent Gate Level Circuits of Same VHDL Model.

6.2 EDIF to GHDL Conversion

The EDIF netlist of a circuit is converted using the HILO EDIF → GHDL converter (TX_EDIF) which generates a nonstandard GHDL format. Figure 6.4 shows a circuit with this format (Figure 4.4 4-bit up counter). In order to get a GHDL description used by HILO, two kinds of translations are needed. One is primitive gate translation. The other is sub-circuit translation.

The primitive gate representation generated by Synopsys synthesis tools differs from the format of GHDL. For example, the OR gates with two inputs and four inputs are represented originally as (inputs: A, B, C, D; output: O):

OR2 U1(A, B, O)

OR4 U2(A, B, C, D, O).

But GHDL represents them as

or U1(O, A, B)

or U2(O, A, B, C, D).

Other primitive gates (AND, NAND, NOR, XOR, NXOR, NOT) have similar relations. The other components (non primitive gates) used in the synthesized circuits are specified by the sub-circuits in file cells.cct with GHDL format [Appendix C]. The file is compiled

once before any other circuits in order to make the sub-circuits available to the other circuits.

The program **tx** reads the circuit representation with non standard GHDL format and performs translations. A table is created to map the primitive gates in the circuit from non standard GHDL format to standard GHDL format. Other components are mapped to sub-circuits with same names. The circuit generated from **tx** then is compiled before simulation. Figure 6.5 shows the circuit after translation.

```
** GHDL file generated from EDIF by GENRAD Translator Version 0.8
**=========================================================
**
** Translated on :                Tue May  7 01:35:55 1996
**
** EDIF Source filename :         SYNOPSYS_EDIF
** EDIF Version :                 2 0 0
**
****************************************************************
** Circuit 'UPCNT4' translated from edif cell 'UPCNT4'
CIRCUIT BIN VERSION=NETLIST UPCNT4 (CLR, CT, LD, CLK, CNT[3], CNT[2],
CNT[1], CNT[0], DATA_IN[3], DATA_IN[2], DATA_IN[1], DATA_IN[0])

FD2 CNT_REG_0_ (N185, CLK, N181, CNT_0_, );
FD2 CNT_REG_1_ (N184, CLK, N181, CNT_1_, );
EO U60 (CNT_2_, N169, N173);
IV U70 (N178, N183);
EO1 U61 (N170, N171, N170, N171, N174);
IV U71 (N179, N184);
EO U62 (CNT_0_, CT, N175);
FD2 CNT_REG_2_ (N183, CLK, N181, CNT_2_, );
IV U72 (N180, N185);
AO2 U63 (N172, N177, DATA_IN_3_, LD, N176);
AO2 U64 (N173, N177, DATA_IN_2_, LD, N178);
IVA U55 (CLR, N181);
AO2 U65 (N174, N177, DATA_IN_1_, LD, N179);
ND2 U56 (N169, CNT_2_, N168);
FD2 CNT_REG_3_ (N182, CLK, N181, CNT_3_, );
AO2 U66 (N175, N177, DATA_IN_0_, LD, N180);
```

```
ND2 U57 (CNT_0_, CT, N170);
IVP U67 (CNT_1_, N171);
NR2 U58 (N171, N170, N169);
IVA U68 (LD, N177);
EN U59 (CNT_3_, N168, N172);
IV U69 (N176, N182);

INPUT CLR;
INPUT CT;
INPUT LD;
INPUT CLK;
BIDIR CNT[3] IS CNT_3_;
BIDIR CNT[2] IS CNT_2_;
BIDIR CNT[1] IS CNT_1_;
BIDIR CNT[0] IS CNT_0_;
INPUT DATA_IN[3] IS DATA_IN_3_;
INPUT DATA_IN[2] IS DATA_IN_2_;
INPUT DATA_IN[1] IS DATA_IN_1_;
INPUT DATA_IN[0] IS DATA_IN_0_;
.
```

Figure 6.4 Non Standard GHDL (upcnt4.cct).

```
** GHDL file generated from EDIF by GENRAD Translator Version 0.8
**==================================================
**
** Translated on :              Tue May  7 01:35:55 1996
**
** EDIF Source filename :       SYNOPSYS_EDIF
** EDIF Version :               2 0 0
**
****************************************************************
** Circuit 'UPCNT4' translated from edif cell 'UPCNT4'
CIRCUIT BIN VERSION=NETLIST UPCNT4 (CLR, CT, LD, CLK, CNT[3], CNT[2],CNT[1], CNT[0],
DATA_IN[3], DATA_IN[2], DATA_IN[1], DATA_IN[0])

FD2 CNT_REG_0_ (N185, CLK, N181, CNT_0_, );
FD2 CNT_REG_1_ (N184, CLK, N181, CNT_1_, );
xor(1,1) U60 (N173, CNT_2_, N169);
not(1,1) U70 (N183, N178);
EO1 U61 (N170, N171, N170, N171, N174);
not(1,1) U71 (N184, N179);
xor(1,1) U62 (N175, CNT_0_, CT);
FD2 CNT_REG_2_ (N183, CLK, N181, CNT_2_, );
not(1,1) U72 (N185, N180);
AO2 U63 (N172, N177, DATA_IN_3_, LD, N176);
```

```
AO2 U64 (N173, N177, DATA_IN_2_, LD, N178);
not(1,1) U55 (N181, CLR);
AO2 U65 (N174, N177, DATA_IN_1_, LD, N179);
nand(1,1) U56 (N168, N169, CNT_2_);
FD2 CNT_REG_3_ (N182, CLK, N181, CNT_3_, );
AO2 U66 (N175, N177, DATA_IN_0_, LD, N180);
nand(1,1) U57 (N170, CNT_0_, CT);
not(1,1) U67 (N171, CNT_1_);
nor(1,1) U58 (N169, N171, N170);
not(1,1) U68 (N177, LD);
nxor(1,1) U59 (N172, CNT_3_, N168);
not(1,1) U69 (N182, N176);

INPUT CLR;
INPUT CT;
INPUT LD;
INPUT CLK;
OUTPUT CNT[3] IS CNT_3_;
OUTPUT CNT[2] IS CNT_2_;
OUTPUT CNT[1] IS CNT_1_;
OUTPUT CNT[0] IS CNT_0_;
INPUT DATA_IN[3] IS DATA_IN_3_;
INPUT DATA_IN[2] IS DATA_IN_2_;
INPUT DATA_IN[1] IS DATA_IN_1_;
INPUT DATA_IN[0] IS DATA_IN_0_;

ENDCIRCUIT
```

Figure 6.5 GHDL Format(upcnt4_n.cct).

6.3 Test Sequence to DWL Conversion

The final test patterns generated from **etg** for a VHDL model is converted into HILO DWL format by the program **fct**. Because the symbolic value F (R) requires two values 1 and 0 (0 and 1), each time frame of a test pattern will be converted into two strobes as shown in Figure 6.6. Here clock signal CLK is rising-edge available.

```
waveform upcnt4;
```

```
base bin;

input
        CLR:=X
        DATA_IN[3:0]:=X
        CT:=X
        LD:=X
        CLK:=X;
output
        CNT[3:0]=X;

begin
.....

at 2400
        CLR:=0
        DATA_IN[3:0]:=1111
        CT:=0
        LD:=1
        CLK:=0;
        CNT[3:0]=X;
at 2450
        strobe();
at 2500
        CLR:=0
        DATA_IN[3:0]:=1111
        CT:=0
        LD:=1
        CLK:=1;
        CNT[3:0]=X;
at 2550
        strobe();
at 2600
        CLR:=0
        DATA_IN[3:0]:=1111
        CT:=1
        LD:=0
        CLK:=0;
        CNT[3:0]=X;
at 2650
        strobe();
at 2700
        CLR:=0
        DATA_IN[3:0]:=1111
        CT:=1
        LD:=0
        CLK:=1;
        CNT[3:0]=X;
at 2750
        strobe();
...
```

```
end
endwaveform upcnt4
```

Figure 6.6 Parts of upcnt4.dwl.

HISIM performs fault free simulation and generates the good output values of a circuit. The output of the HISIM gives the information about strobe failures if the expected output signal values are different from the actual(good) ones. Program **rsd** reads the output and the original waveform file generated by **fct** to decide the uncertain output values in the waveform. Finally a waveform file with complete input and output specifications is produced and used to drive the GenRad HILO fault grader. Figure 6.7 shows such a waveform file.

```
waveform upcnt4;
base bin;

input
        CLR:=X
        DATA_IN[3:0]:=X
        CT:=X
        LD:=X
        CLK:=X;
output
        CNT[3:0]=X;

begin

...

at 2400
        CLR:=0
        DATA_IN[3:0]:=1111
        CT:=0
        LD:=1
        CLK:=0;
```

```
        CNT[3:0]=1000;
at 2450
        strobe();
at 2500
        CLR:=0
        DATA_IN[3:0]:=1111
        CT:=0
        LD:=1
        CLK:=1;
        CNT[3:0]=1111;
at 2550
        strobe();
at 2600
        CLR:=0
        DATA_IN[3:0]:=1111
        CT:=1
        LD:=0
        CLK:=0;
        CNT[3:0]=1111;
at 2650
        strobe();
at 2700
        CLR:=0
        DATA_IN[3:0]:=1111
        CT:=1
        LD:=0
        CLK:=1;
        CNT[3:0]=0000;
at 2750
        strobe();

...

end
endwaveform upcnt4
```

Figure 6.7 Parts of upcnt4n.dwl.

6.4 Fault Grading

Once the GHDL description of a circuit (*.cct) and the DWL waveform of the test patterns (*n.dwl) are obtained, we can set them as the input of HIFAULT. HIFAULT performs fault simulation and gives the fault coverage. Depending on the fault insertion level, HIFAULT gives the top-level or all-level fault coverage. Figure 6.8 gives an example with the fault coverage results for top-level stuck faults (without including the faults within a sub-circuit) and Figure 6.9 shows the fault coverage results for all-level stuck faults. The status of faults in these figures is explained as follows.

At each strobe in the waveform the fault simulator will make a comparison between the fault free value at each primary output of the circuit, and the corresponding faulty values. If different, the faulty values will be marked as detected or potential detected. For a 3-valued (0, 1, X) logic case, if a fault free value is 1 (0) and the faulty value is 0 (1), the faulty value is marked as **detected** (det'd). Otherwise if a fault free value is 1 (0) and the faulty value is X, the faulty value is marked as **potential detected** (pdetd). A fault is **dropped** (dropd) if it is detected or potentially detected a number of times. In the latter case, faults are normally dropped after six potential detections.

A fault is classified as **catastrophic fault** (catas) if

— The fault has caused ambiguous simulation values at initialization but the corresponding fault-free value is unambiguous;

— The fault has caused a zero delay oscillation to occur and has been automatically suppressed.

```
System HILO DWL Version 4.0.16
Log file started on  6-JUN-96 at time 14:33:05

Running the HIFAULT simulator
Loading the default version of circuit 'UPCNT4'  time: 0.41 secs
Circuit Loading Complete  took: 0.44 secs  time: 0.85 secs
Circuit UPCNT4 has nanosecond scaling
Delay set used is TYPICAL
Number of subcircuit elements loaded =   9
Number of primitive  gates   loaded =  52
0 declaration expressions; 0 event statements
Memory used for symbols and  objects  by loader =    6868 bytes
Memory used for simulator  structures by loader =   15620 bytes
DWL Compiler
Waveform UPCNT4 compiled successfully
DWL to Circuit Linker
Initialising simulator:   time: 1.35 secs
Default set of faults contains:
   60 top level stuck faults

Fault dictionary statistics:
  60 entries after sampling and pre_analysis
Loading of faults completed :   took 0.06 secs  (total 1.41 secs)
DWL Execution
60 faults to be simulated
Simulator initialised:   took 0.04 secs  (total 1.45 secs)
Start simulation:   took 0.06 secs  (total 1.51 secs)
SIMULATOR START, number of active faults 60
At time 150 number of active faults is 44
At time 250 number of active faults is 40
At time 350 number of active faults is 24
At time 550 number of active faults is 23
At time 750 number of active faults is 22
At time 950 number of active faults is 16
At time 1150 number of active faults is 15
At time 1350 number of active faults is 14
At time 1550 number of active faults is 9
At time 1950 number of active faults is 8
At time 2150 number of active faults is 3
At time 2550 number of active faults is 2
At time 3650 number of active faults is 0
Finish at 3650
Finish simulation:   took 0.22 secs  (total 1.73 secs)
```

Simulator finished: took 0.00 secs (total 1.73 secs)
Written Fault Dictionary: took 0.03 secs (total 1.76 secs)

Fault Dictionary Summary:

	all_faults					simulated_faults				
fault	total	dropd	det'd	pdetd	catas	total	dropd	det'd	pdetd	catas
Stuck0	30	30	28	2	0	30	30	28	2	0
Stuck1	30	30	28	2	0	30	30	28	2	0
All	60	60	56	4	0	60	60	56	4	0
%'s		100.0	93.3	6.7	0.0		100.0	93.3	6.7	0.0

%'s are of detectable faults only.

Figure 6.8 Fault Simulation Summary (Top-Level).

Fault Dictionary Summary:

	all_faults					simulated_faults				
fault	total	dropd	det'd	pdetd	catas	total	dropd	det'd	pdetd	catas
Stuck0	60	60	58	2	0	60	60	58	2	0
Stuck1	60	60	54	6	0	60	60	54	6	0
All	120	120	112	8	0	120	120	112	8	0
%'s		100.0	93.3	6.7	0.0		100.0	93.3	6.7	0.0

%'s are of detectable faults only.

Figure 6.9 Fault Simulation Summary (All-Level).

Chapter 7. Results

The effectiveness of the algorithms presented in the previous chapters has been evaluated by using various VHDL behavioral models. This chapter will present the test results for the models with a single process and the models with multiple processes. Appendix B contains the VHDL code for such models.

7.1 Models with a Single Process

One type of single process models we used contain only a single VHDL function shown in Section 4.3. The test patterns generated for such a model were applied to the different equivalent circuits of the same model. The equivalent circuits of the models with a same function but different bit width (4-bit, 8-bit, 16-bit inputs) were also tested. The fault coverages were found to be about in the range of 90% to 100%.

Table 7.1 shows the test results after applying the test patterns to the different bit width adders. Table 7.2 and 7.3 are the results for incrementers and decrementers. From the tables we can see that the fault coverages for smallest implementations are high but the fault coverages for fastest implementations are decreased. After analyzing the gate level circuits, we found the following possible reasons. The smallest implementation of a model usually has a circuit structure which well matches its VHDL model. For example, we can see the different stages in the smallest implementation of a ripple adder. But the fastest implementation usually has less layers and more components, so it is necessary to have more test vectors in order to get higher fault coverages. This shows that the function FVTP generation algorithms are synthesis related.

Table 7.1 Test Results for Adders.

(Note: S--smallest implementation F--fastest implementation)

Model	Bit Width	Type	Gate Number	Total Faults (Top-Level)	Fault Coverages(%)
add4	4	S	18	46	100.0
		F	25	56	100.0
add8	8	S	49	98	100.0
		F	73	146	97.3
add16	16	S	109	204	100.0
		F	157	332	95.8

Table 7.2 Test Results for Incrementers.

Model	Bit Width	Type	Gate Number	Total Faults (Top-Level)	Fault Coverages(%)
inc4	4	S	9	20	100.0
		F	10	22	100.0
inc8	8	S	29	50	100.0
		F	40	72	98.6
inc16	16	S	89	176	99.4
		F	101	180	91.7

Table 7.3 Test Results for Decrementers.

Model	Bit Width	Type	Gate Number	Total Faults (Top-Level)	Fault Coverages(%)
dec4	4	S	13	24	100.0
		F	17	36	97.2
dec8	8	S	46	92	100.0
		F	53	112	92.0
dec16	16	S	98	196	99.5
		F	153	296	91.6

Tables 7.4, 7.5, and 7.6 give some other test results for the processes with a single VHDL function. Only one gate level circuit for each model was evaluated. But the fault simulation was performed at both top-level and all-level. At top-level, the faults within a sub-circuit such as flip-flop, latch, and multiplexer will not be activated. At all-level, all the faults in a circuit will be activated. The logic structure of such sub-circuits are described in Appendix C.

All the different multiplier models were constructed by using the function MULT and assigning different bit width (4-bit, 8-bit, 16-bit, 32-bit) to the input signals. The comparators were constructed by using the same way. As for the shifters, they were constructed by combining the three shift functions: SHIFTR, SHIFTL, and SHIFTA into a 4-function shifter. When generating the gate level circuits, no synthesis constrains were assigned.

Table 7.4 Test Results for Multipliers.

Model	Bit Width	Level	Comt. Number	Total Faults	Fault Cov.(%)	
					dropd	det'd
mult4	4	Top	22	152	100.0	100.0
		All	105	226	98.7	98.7
mult8	8	Top	109	636	99.8	99.4
		All	510	1052	98.7	97.9
mult16	16	Top	491	2662	100.0	100.0
		All	2267	4598	99.7	99.7
mult32	32	Top	2004	10950	100.0	99.2
		All	9388	18904	100.0	99.3

Table 7.5 Test Results for Comparators.

Model	Bit Width	Level	Comt. Number	Total Faults	Fault Cov.(%)	
					dropd	det'd
comp4	4	Top	6	58	100.0	100.0
		All	28	72	100.0	100.0
comp8	8	Top	8	108	100.0	100.0
		All	47	126	100.0	100.0
comp16	16	Top	19	210	100.0	100.0
		All	95	254	100.0	100.0
comp32	32	Top	31	428	99.8	99.8
		All	194	516	99.8	99.8

Table 7.6 Test Results for Shifters.

Model	Bit Width	Level	Comt. Number	Total Faults	Fault Cov.(%)	
					dropd	det'd
shift4	4	Top	9	46	100.0	100.0
		All	31	76	98.7	98.7
shift8	8	Top	16	86	98.8	98.8
		All	55	132	99.2	99.2
shift16	16	Top	32	150	100.0	100.0
		All	104	246	100.0	100.0
shift32	32	Top	64	278	100.0	100.0
		All	200	470	100.0	100.0

Table 7.7 shows the test results for more general single process models. Among them, **upcnt4** is shown in the Figure 4.4. **downcnt8** is an 8-bit down counter. **COUNTER** is a controlled counter which can count up or down. **alu32** is an 8-function 32-bit ALU. **SHIFTREG** is a 4-bit shift register. **mux4_1** is a 32-bit 4-to-1 multiplexer. Finally, **demux1_4** is an 8-bit 1-to-4 demultiplexer. Both dropped faults and detected faults are evaluated.

Table 7.7 Test Results for Single Process Models.

Model	Subcircuit Number	Gate Number	Total Faults (Top Level)	Fault Coverages(%)	
				dropd	det'd
upcnt4	9	52	60	100.0	93.3
downcnt8	23	108	122	100.0	98.4
COUNTER	14	83	108	98.1	83.3
alu32	314	1211	1616	99.6	99.6
SHIFTREG	15	68	90	96.7	86.7
mux4_1	64	230	464	98.7	98.7
demux1_4	0	46	112	100.0	100.0

From the above test results, we can know that the P-algorithm is effective on moderately complicated processes.

7.2 Models with Multiple Processes

The test results for six multiple process models are shown in Table 7.8. Their PMGs are presented in Figure 7.1 ~ 7.5. Among these models, **amd2910c** is a modified version of amd2910 micro-controller [54]. The amd2910c contains an instruction decoder, a stack and its pointers, a multiplexer, a controled up-counter, and a controlled down-counter. Depending on the instructions supplied and the internal state signal values, the instruction decoder generates various control signals to control the actions of other components.

mpu2 is a small CPU which uses a counter to select the different functions of its ALU. **CKA** is a model that can be used to illustrate path conflict resolution. **csi** is an example containing a 4-function shifter process. **muxadd** performs additions from multiple sources. Both combinational and sequential circuits exist among these models. Only one implementation is used for each model. The fault simulations were performed at top-level and all-level. The high fault coverage for these models shows that the E-algorithm is effective for multi-process circuits.

Table 7.8 Test Results for Multiple Process Models.

Model	Proc. No.	Level	Comt. Number	Total Faults	Fault Cov.(%)	
					dropd	det'd
amd2910c	10	Top	373	1386	95.2	95.1
		All	1732	3508	93.0	92.9
mpu2	6	Top	145	618	98.2	97.6
		All	565	1244	97.5	97.2
CKA	3	Top	16	88	98.9	96.6
		All	89	200	93.5	92.5
csi	4	Top	13	70	98.6	95.7
		All	60	138	99.3	97.8
incadd	5	Top	19	110	99.1	97.3
		All	87	206	98.5	97.6
muxadd	7	Top	646	3732	99.1	99.1
		All	2699	6430	98.0	98.0

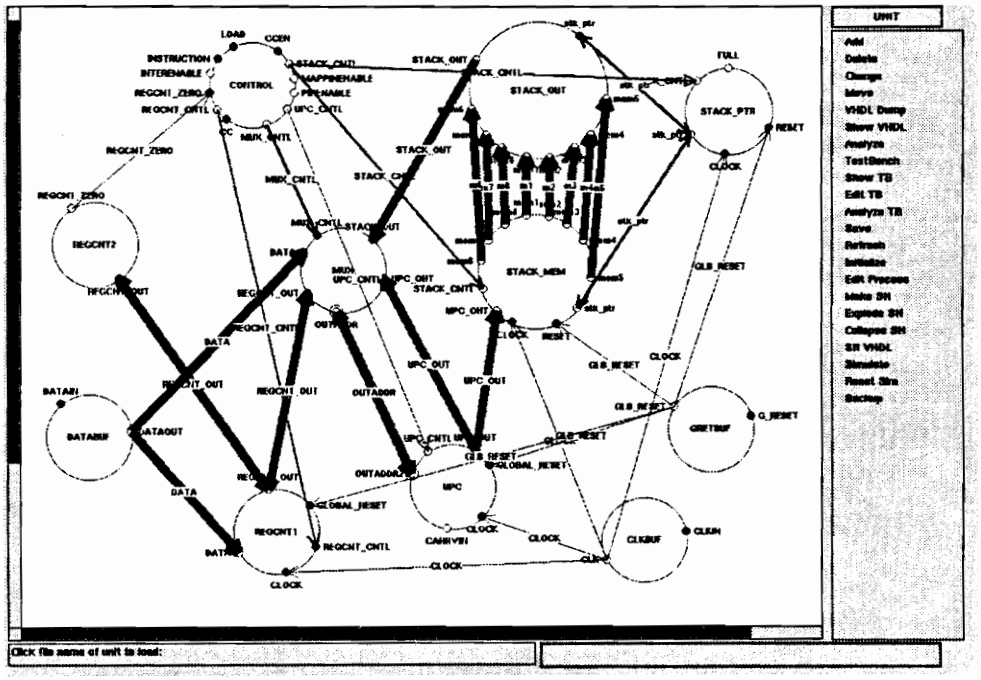


Figure 7.1 PMG of **amd2910c**.

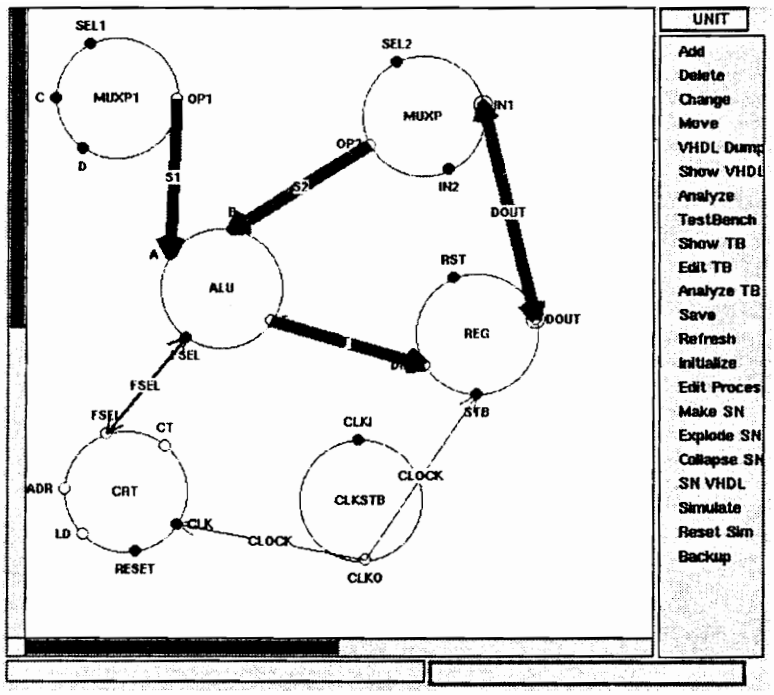


Figure 7.2 PMG of **mpu2**.

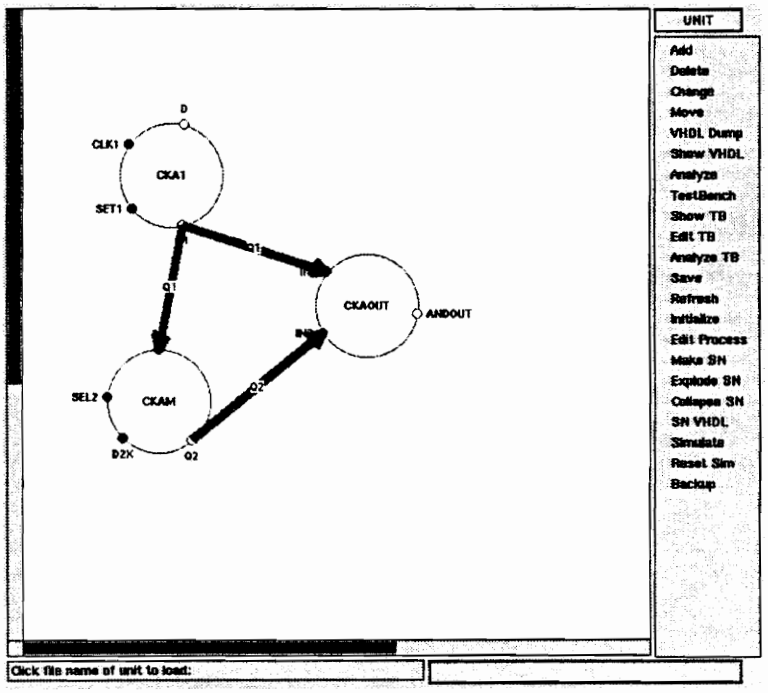


Figure 7.3 PMG of **CKA**.

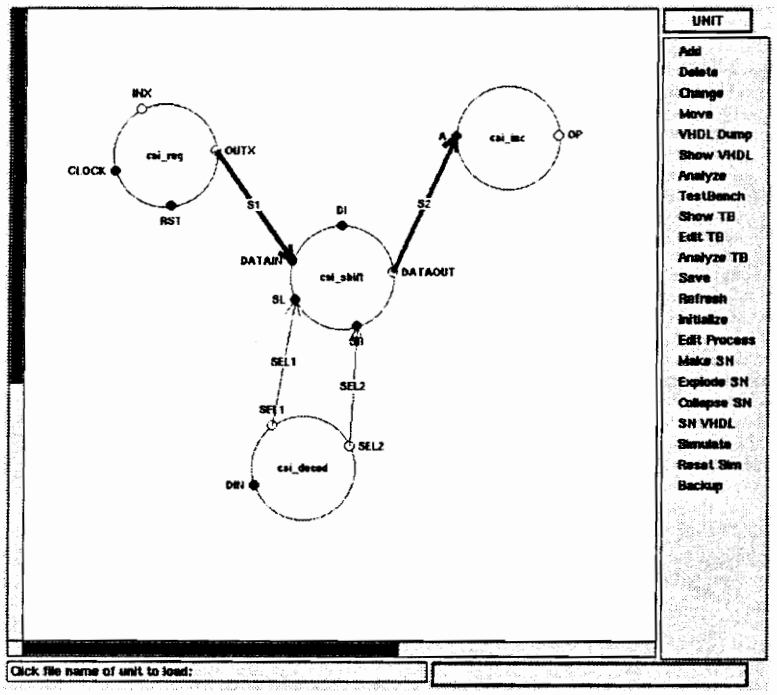


Figure 7.4 PMG of **csi**.

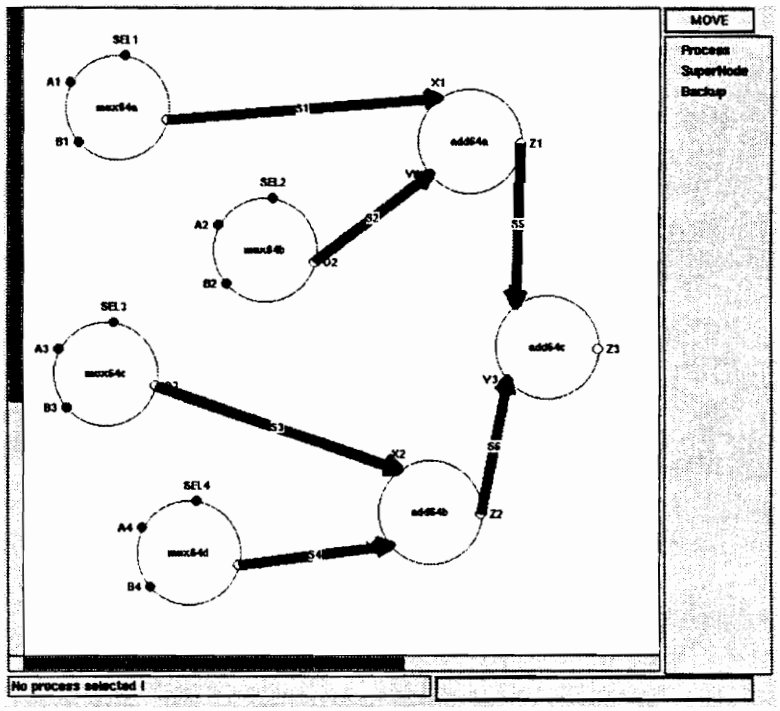


Figure 7.5 PMG of **muxadd**.

Chapter 8. Proposed Future Work

We have completely presented our test generation system. It is an effective and powerful behavioral level test generation tool. However there exist some limitations in its current implementation. This chapter discusses such limitations, proposes the possible improvement, and suggests future research areas.

8.1 Data Resolution

In the present implementation of the E-algorithm, we assume that the final test patterns can be obtained by calculating the final symbolic tests. But this is not always true when an A-mode test is used to perform propagation and justification. Figure 8.1 gives such an example.

Here we try to test the process ADD. At the beginning, the A-mode test $(B, C, D) = (D_1, D_2, s(D_1, D_2))$ is selected. We can use D_3 to represent $s(D_1, D_2)$. Then D_1 and D_2 are

justified to PIs and D_3 is propagated to POs. We can obtain that $O1 = i(D_3) = is(D_1, D_2)$ and $D_1 = m(I_1, I_2)$. The values of D_1 and D_2 will be decided by the FVTP of the process ADD. The output $O1$ can be obtained by calculating $is(D_1, D_2)$. The inputs $I3$ and $I4$ can be decided by P-mode test of the process AND. But we cannot guarantee to find suitable values of inputs $I1$ and $I2$ for a given D_1.

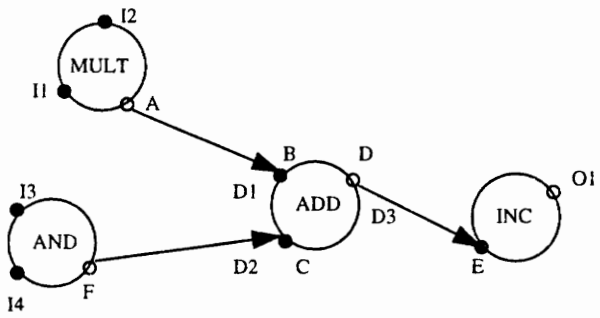


Figure 8.1 Illustration of Data Resolution.

Generally we can get several equations:

$D_1 = f_1(I_1, I_2, ..., I_n)$

$D_2 = f_2(I_1, I_2, ..., I_n)$.

...

$D_m = f_m(I_1, I_2, ..., I_n)$

Here I_j $(j=1,2,..,n)$ is a primary input and D_i $(i=1,2, ..., m)$ is the input for a given process. The PI values can be decided by resolving the equations given the values of each D_i. If for

a given value of D_i the corresponding PI values can not be decided, we can try to use another two values. One value is greater than the given value. The other is less than the given value. But both are closest to the given value. Such data resolution problems are inherent and inevitable in high level or behavioral level test generation. The reason is that there exists a large data space because the bus or vector values are used. For a n-bit vector, there are 2^n possible values. But only parts of the values are chosen to test a high level sub-unit (primitive or function). Some of these values may not be propagated to POs or PIs if no one-to-one and onto mappings exist in some sub-units.

8.2 Control Logic Test Generation

Most digital functions can be divided into the following categories [74]:

- data path operators
- control structures
- memory elements
- I/O cells

The models used in this dissertation consist of first two kinds of structures as shown in Figure 8.2. The memory elements and I/O cells usually are tested by using quite different approaches. For example, self-testing techniques [6] may be used to test the memories.

Data path logic unit can be constructed by using the VHDL functions. Control logic unit (shadow part) can be an FSM, a PLA, a controlled ROM, *et al.* Our test generation algorithms works well for data path logic unit and certain kinds of control logic. For general control logic, the algorithms need to be improved.

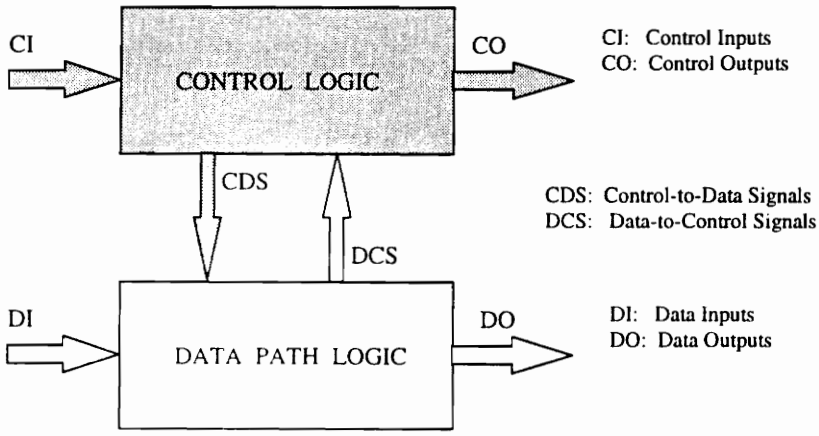


Figure 8.2 General Model for Test Generation.

8.3 Synthesis Related Issues

We have indicated that the FVTP generation algorithm for a function is synthesis related, i.e., when deriving the algorithm, we referred to the structures of the synthesized gate level circuits which were generated by certain synthesis algorithms employed by the Synopsys synthesis tools. If we know how the synthesis algorithms work, we can deduce what kinds of circuit structures will be generated by the algorithms for a given VHDL function. Therefore we may obtain a more accurate FVTP generation algorithm.

We suppose that there exists a corresponding sub-circuit for a process in the equivalent gate level circuit of the VHDL model containing the process. This assumption is true for the intermediate synthesis results. After high level optimization (resource sharing), logic-level optimization (flattening and structuring), and gate-level optimization (mapping), there should still exist a sub-circuit which will perform the functionality of the process. However such a sub-circuit may belong in another sub-circuit which will perform the functionality of another process. Therefore, the final test patterns for the whole circuit will detect such sub-circuits but may contain some redundant parts.

Another issue concerning synthesis is test-set preserving [16]. As a circuit is transformed during logic synthesis to improve area or performance, its testing properties also change in some ways. If a transformed logic circuit can be tested by a set of tests designed for the original circuit, the transformation is said "test-set preserving".

We have indicated that several gate level implementations for a same VHDL model can be obtained by selecting different optimization constraints. The test patterns generated by our algorithms were evaluated in at least one implementation and had high fault coverage. The fault coverage in other implementations should be high if such implementations are test-set preserving to a certain extent. For a function FVTP generation algorithm, we can improve it to make it effective for a series of implementations. The P-algorithm which

uses the improved algorithm should generate more effective test sets for a process containing such a function.

Now let's think about how to apply our test generation algorithms to a synthesis system. One way is guiding the synthesis process. Our algorithms can be used to evaluate the testability of a VHDL model before performing synthesis. If the FVTP for the process to be tested can not be applied because of symbolic test generation failures or final test pattern calculation failures, this implies that the model may have poor testability. The synthesis system can insert additional circuits to improve the testability. For example, inserting a multiplexer can provide P-mode symbolic tests which may allow the FVTP to be applied. Because no scan logic is used, the cost of such additional circuits may be relatively lower. The synthesis system can also inform the user to improve his VHDL model to increase the testability.

Another way is using the test set generated by our algorithms as the preliminary tests for a general ATPG system embedded in a synthesis system. First, the test set is evaluated with the help of the fault simulator in the ATPG system and the detected faults are dropped. Then, if there exist undetected faults, the ATPG system is invoked to try generating tests for such hard detected faults. Our algorithms can also call the ATPG system to generate fixed valued tests for some processes or functions if it is difficult to derive the FVTP generation algorithms for these processes and functions.

In summary, research on behavioral level test generation is closely related to synthesis because some implementation details are needed. Therefore, combining them together is a promising approach.

8.4 Test Pattern Compaction

No fault simulation is performed during test generation. Therefore there may exist some test patterns which are redundant and detect no more undetected faults. Such test patterns need to be removed from final test set. An approach to remove such redundant tests is using the fault simulation results as shown in the Figure 6.8. By parsing the results, the test patterns which detect no faults can be removed.

8.5 Developing Procedures and More Functions

At present, eight VHDL functions are provided to develop VHDL models. We can construct more functions to implement new operations. For same functions, we can also construct other implementations. For example, the function ADD can be implemented as two functions: ADDR (ripple adder) and ADDL (carry look-ahead), which can be used to construct models with specific requirements. We also need to develop new algorithms for these functions to give high gate level fault coverage.

A function can only return one value. In order to return two or more values, we need some specific method. For example, the function COMP returns a 2-bit vector to represent three cases: equal *("00")*, greater-than *("01")*, and less-than *("10")*. However a more convenient way is employing VHDL procedure because a procedure can have any number of input and output parameters. Following is a procedure to implement the ripple-carry adder with two outputs: sum and carry. The experiment results show that the test set generated by using the FVTP generation algorithm for the function ADD will also give high fault coverage when the tests are applied to the equivalent gate level circuit of the procedure.

```
procedure ADC(A,B: in BIT_VECTOR; SUM: out BIT_VECTOR; C: out BIT) is
  variable SUMV,AV,BV: BIT_VECTOR(A'LENGTH-1 downto 0);
  variable CARRY: BIT;
begin
  AV := A;
  BV := B;
  CARRY := '0';
  for I in 0 to SUMV'HIGH loop
    SUMV(I) := AV(I) xor BV(I) xor CARRY;
    CARRY := (AV(I) and BV(I)) or (AV(I) and CARRY) or (BV(I) and CARRY);
  end loop;
  SUM:=SUMV;
  C:=CARRY;
end ADC;
```

Chapter 9. Conclusions

A behavioral level test generation system for VHDL models was presented in this dissertation. The test generation algorithms were discussed extensively. These include the function test generation algorithms, the P-algorithm, and the E-algorithm. Some supporting tools were also introduced. The system has been implemented using C on Sun Workstations.

The test patterns generated for a VHDL model give the high fault coverage for the equivalent gate level circuit. Such a circuit can be a combinational circuit or clocked circuit which random test generation can not handle. There exist the slight differences in the fault coverage when the test patterns were applied to the different equivalent gate level circuits of the same model. But most of them reach a desired high level of coverage. All the circuits were obtained by using the Synopsys synthesis tools. The function test algorithms were derived after analyzing such circuits. Therefore they are currently limited to the Synopsys synthesis tools but can be developed for other specific synthesis tools.

More experiments are needed to help improve the algorithms and programs. More VHDL functions can be developed and the corresponding algorithms need to be derived. Such algorithms can be obtained by using the same approach introduced in this dissertation or provided by users of specific synthesis algorithms or tools. If the test generation algorithms can be combined with synthesis algorithms or tools, more perspective and practical results can be obtained.

Bibliography

[1] S. Freeman, "Test generation for data-path logic: the *F*-path method," *IEEE J. Solid -State Circuit,* Vol.. 23, Apr. 1988, pp.421-427.

[2] C-C. Su and C. R. Kime, "Multiple path sensitization for hierarchical circuit testing," *Proc. Int. Test Conf.,* 1988, pp. 152-161.

[3] M. S. Abadir and M. A. Breuer, "A knowledge-based system for designing testable VLSI chips," *IEEE Design Test,* Aug. 1985, pp. 56-68.

[4] M. S. Abradir and M. A. Breuer, "Test schedules for VLSI circuits having built-in test hardware," *IEEE Trans. Comput.,* vol. c-35, no. 4, pp.361-367, Apr.1986.

[5] S. Lin, C. Njinda and M. A. Breuer, "Generating a family of testable designs using the BILBO methodology," *J. Electronic Testing: Theory and Applications.,* vol. 4, pp.71-88,1993.

[5'] M. A. Breuer and A. D. Friedman, "Functional level primitives in test generation," *IEEE Trans. Comput.,* vol. c-29, no.3, pp.223-235, March 1980.

[6] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital systems testing and testable design,* Computer Science Press,1990.

[7] P. Vishakantaiah, J. A. Abraham, and M. Abadir, "Automatic test knowledge extraction from VHDL(ATKET)," *Proc. 29th Design Automation Conf.,* 1992, pp. 273-278.

[8] P. Vishakantaiah, J. A. Abraham, and D. G. Saab, "CHEETA: composition of hierarchical sequential tests using ATKET," *Proc. Int. Test Conf.,* 1993, pp.606-615.

[9] R. P. Kunda, P. Narain, J. A. Abraham, and B. D. Rathi, "Speed up of test generation using high-level primitives," *Proc. 27th Design Automation Conf.,* 1990, pp.594-599.

[10] P. Narian, D. G. Saab, R. Kunda and J. Abraham, "A high level approach to test generation," *IEEE Trans. Circuits and Systems-1: Fundamental Theory and Applications,* vol. 40, no.7, pp.483-492, July 1993.

[11] S. M. Thatte and J. Abraham, "Test generation for microprocessors," *IEEE Trans. Comput.,* vol. c-29, no.6, pp.429-441, June 1980.

[12] C. Chen and D. G. Saab, "A novel behavioral testability measure," *IEEE Trans. CAD of Integrated Circuits and Systems,* vol.12, no.12, pp.1960-1970, Dec. 1993.

[13] C. Chen, T. Karnik and D. G. Saab, "Structural and behavioral synthesis for testability techniquies," *IEEE Trans. CAD of Integrated Circuits and Systems,* vol.13, no.6, pp.777-785, June1994.

[14] B. T. Murray and J. P. Hayes, "Hierarchical test generation using precomputed tests for modules," *IEEE Trans. CAD,* vol.9,no.6, pp.594-603, June 1990.

[15] B. T. Murray and J. P. Hayes, "Test propagation through modules and circuits," *Proc. Int. Test Conf.,* 1991, pp. 748-757.

[16] M. J. Batek and J. P. Hayes, "Test-set preserving logic transformations," *Proc. 29th Design Automation Conf.*, 1992, pp.454-458.

[17] D. Bhattacharya and J. P. Hayes, "Hierarchical modeling for VLSI circuit testing," Kluwer Academic Publishers,1990.

[18] Y. H. Levendel and P. R. Menon, "Test generation algorithm for computer hardware description languages," *IEEE Trans. Comput.,* Vol. c-31, pp.577-588, July 1982.

[19] P. N. Anirudhan and P. R. Menon, "Symbolic test generation for hierarchically modeled digital system," *Proc. Int. Test Conf.,* 1989, pp. 461-469.

[20] T. W. Williams and K. P. Parker, "Design for testability -- A survey," *IEEE Trans. Comput.,* vol. c-31, pp.2-15, Jan.1982.

[21] J. Lee and J. H. Patel, "An architectural level test generator for hierarchical design environment," *21th Symposium on Fault-Tolerant Computing* , 1991, pp. 44-51.

[22] J. Lee and J. H. Patel, "ARTEST: an architectural level test generator for data path faults and control faults," *Proc. Int. Test Conf.,* 1991, pp. 729-738.

[23] V. Chickermane, E. M. Rudnick, P. Banerjee and J. H. Patel, "No-scan design-for-testability techniques for sequential circuits," *Proc. 30th Design Automation Conf.*, 1993, pp.236-241.

[24] J. Lee and J. H. Patel, "An architectural level test generator based on nonlinear equation solving," *J. Electronic Testing: Theory and Applications.*, vol. 4, pp.137-150, 1993.

[25] V. Chickermane, J. Lee and J. H. Patel, "Addressing design for testability at the architectural level," *IEEE Trans. CAD of Integrated Circuits and Systems,* vol.13, no.7, pp.920-934, July1994.

[26] J. Lee and J. H. Patel, "Architectural level test generation for microprocessors," *IEEE Trans. CAD of Integrated Circuits and Systems,* vol.13, no.10, pp.1288-1300, Oct.1994.

[27] J. P. M. Silva and K. A. Sakallah, "Dynamic search-space pruning techniques in path sensitization," *Proc. 31st Design Automation Conf.*, 1994, pp.705-711.

[28] J. Steensma, W. Geurts, F. Catthoor and H. D. Man, "Testability analysis in high level data path synthesis," *J. Electronic Testing: Theory and Applications.*, vol. 4, pp.45-56, 1993.

[29] R. S. Ramchandani and D. E. Thomas, "Behavioral test generation using mixed integer non-linear programming," *Proc. Int. Test Conf.*, 1994, pp. 958-967.

[30] T. M. Sarfert, R. Markgraf, E. Trischler and M. H. Schulz, "Hierarchical test pattern generation based on high-level primitives," *Proc. Int. Test Conf.*, 1989, pp. 470-479.

[31] L. H. Goldstein, "Controllability/observability analysis of digital circuits," *IEEE Trans. Circuits and Systems,* vol. cas-26, no.9, pp.685-693, Sept. 1979.

[32] J. R. Armstrong and F. G. Gray, *Structured logic design with VHDL*, PTR Prentice Hall, 1993.

[33] F. E. Norrod, "An automatic test generation algorithm for hardware description languages," *Proc. 26th Design Automation Conf.*, 1989, pp.429-434.

[34] C. H. Cho and J.R. Armstrong, "VHDL semantics for behavioral test generation," *Proc. CHDL-91*, pp.427-444.

[35] C. H. Cho and J. R. Armstrong, "The B algorithm: A behavioral test generation algorithm," *Proc. Int. Test Conf.*, 1994, pp.968-979.

[36] S. Rao, B. Pan,and J. R. Armstrong, "Hierarchical test generation for VHDL behavioral models," *Proc. European Design Automation Conf.*,Feb.22-24,1993, pp.175-183.

[37] S. Kapoor, J. R. Armstrong, and S. R. Rao,"An automatic test bench generation system,"*Proc. VHDL Int. Users Forum,*1994, pp.8-17.

[38] F. G. Gray, J. R. Armstrong, and A. W. Bennett, "Microprocessor self test," RADC Contract No.F30 602-80-C0200, Final Report,1981.

[39] A. Guptal and J. R. Armstrong, "Functional fault modeling and simulation for VLSI devices," *Proc. 1985 Design Automation Conf.,* pp.720-726.

[40] D. Barclay and J. R. Armstrong, "A heuristic chip-level test generation algorithm," *Proc. 23rd Design Automation Conf.* June 1986, pp.257-262.

[41] M. D. O'Neil, D. Jani, C. H. Cho,and J. R. Armstrong, "BTG: a behavioral test generator," *Proc. CHDL-89,* pp.347-361.

[42] P. Ward and J. R. Armstrong, "Behavioral fault simulation in VHDL," *Proc. 26th Design Automation Conf.,* June 1989, pp.587-593.

[43] C. H. Chao and F. G. Gray, "Microperation perturbations in chip level fault modeling," Proc. 25th Design Automation Conf., pp.579-582.

[44] B. Singh, J. Wicks, P. Wright, and J. R. Armstrong, "The Modeler's Assistant: A CAD tool for behavioral model development," 14 pages, Proc. CHDL 93, April 1993, Ottawa, Canada.

[45] F. G. Gray and J. R. Armstrong, "Realization of VHDL testbench and library components," *Proc. AIAA Computing in Aerospace 10,* 1995, pp.691-700.

[46] V. Pitchummani, P. Mayor, and N. Radia, "A system for fault diagnosis and simulation of VHDL description," *Proc. 28th Design Automation Conf.*, 1991, pp.144-150.

[47] P. C. Maxwell and R. C. Aitken, "Test set and reject rates: All fault coverages are not created equal," *IEEE Design & Test of Computers,* Vol.10, pp.42-51, March 1993.

[48] *VHDL compiler reference manual,* Version 3.0, Synopsys Inc., Nov. 1992.

[49] *Design compiler reference manual,* Version 3.0, Synopsys Inc., Dec. 1992.

[50] *Designware databook*, Version 3.0, Synopsys Inc., Dec. 1992.

[51] *System HILO system reference manual*, GenRad Limited, 1991.

[52] *VHDL tool integration platform (VTIP) / Design library system (DLS) manual*, CAD language systems, Inc., 1993.

[53] *IEEE standard VHDL language reference manual,* IEEE, Inc., March 1988.

[54] K. C. Chang and E. J. Olson, "Tutorial C : VHDL synthesis for digital ASIC designs," *VHDL International Users'Forum*, May 1994.

[55] J. P. Roth, "Diagnosis of automata failures: a calculus and a method," *IBM J. Res. Develop.*,vol. 10, pp. 278-291, July 1966.

[56] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Trans. Comput.*, vol. c-30, pp.215-222, March 1981.

[57] H. Fujiwara and T. Shimono, "On the acceleration of test generation algorithms," *IEEE Trans. Comput.*, vol. c-32, pp.1137-1144, Dec. 1983.

[58] J.-L. Baer, *Computer systems architecture*, Computer Science Press, 1980.

[59] A. Lioy, P. L. Montessoro, and S. Gai, "A complexity analysis of sequential ATPG," *Proc. Int. Symp. on Circuits and System,* pp. 1946-1949, May 1989.

[60] A. Miczo, "The sequential ATPG: a theoretical limit," *Proc. Int. Test Conf.*, pp. 143-147, 1983.

[61] H.-K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli, "Test generation for sequential circuits," *IEEE Trans. Computer-Aid Design.*, vol.7, no. 10, pp.1081-1093, Oct. 1988.

[62] S. Mallela and S. Wu, "A sequential circuit test generation system," *Proc. Int. Test Conf.*, pp. 57-61, 1985.

[63] W.-T. Cheng and T. J. Chakraborty, "Gentest: an automatic test-generation system for sequential circuits," *Computer,* vol. 22, no. 4, pp. 43-49, April 1989.

[64] W. Li and J. R. Armstrong, "Behavioral Test Generation with Fault Coverage Enhancement," *The Second Workshop on Hierarchical Test Generation*, Duisburg, Germany, Sept. 25-26, 1995.

[65] N. D. Dutt and D. D. Gajski, "Design synthesis and silicon compilation," *IEEE Design & Test of Computers*, pp. 8-22, Dec. 1990.

[66] R. Camposano and W. Rosenstiel, "Synthesizing circuits from behavioral descriptions," *IEEE Trans. Computer-Aid Design.*, vol.8, no. 2, pp.171-180, Feb. 1989.

[67] T. H.-Y. Meng, R. W. Brodersen, and D. G. Messerschmitt, "Automatic synthesis of asynchronous circuits from high-level specifications," *IEEE Trans. Computer-Aid Design.*, vol.8, no. 11, pp.1185-1205, Nov. 1989.

[68] A. Ghosh, S. Devadas, and A. R. Newton, *Sequential logic testing and verification*, Kluwer Academic Publishers, 1992.

[69] X. Zhong, "Propagation and justification algorithms of boolean expression for test generation of VHDL behavioral model," Project report, VPI&SU, Sep. 1995.

[70] G. Baweja, "Gate level coverage of a behavioral test generator," Master's Thesis, VPI&SU, Mar. 1993.

[71] S. Kapoor, "Process level test generation for VHDL behavioral models," Master's Thesis, VPI&SU, Mar. 1994.

[72] C. H. Cho, "A formal model for behavioral test generation," PhD's Dessertation, VPI&SU, Feb. 1994.

[73] D. A. Patterson and J. L. Hennessy, *Computer architecture: A quantitative approach,* Morgan Kaufmann Publishers, Inc. 1996.

[74] N. H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI design: A systems perspective,* Addison-Wesley Publishing Company, 1993.

Appendix A: User's Guides

1. Obtaining VHDL code

- Use the Modeler's Assistant to construct VHDL model.
- Use "VHDL Dump" in the menus to get the VHDL code for each process and the unit. If the model contains only a single process, just dump the code of the unit.

2. Process level test generation

In this section and following sections, the *italic* parts represent the outputs from the programs and the **bold** parts indicate the inputs from users.

Case 1: single process models

Compile the VHDL code first and then run **ptg**.

%> **vhdl upcnt4.vhd**

%> **ptg**
Enter the filename: **upcnt4**

Enter unit name: **upcnt4**
Enter process name:(default=upcnt4)
Enter the Library name:(default=work)
Enter the Entity name:(default=upcnt4)
Enter the Architecture name:(default=behavior) **BEHAVIORAL**

Case 2: multiple process models

For each process, firs run **mvhd** to construct a complete VHDL. Then compile the VHDL code. Finally run **ptg**.

%> **mvhd**
Enter unit name: **incadd**
Enter module name: **add4**

%> **vhdl add4.vhd**

%> **ptg**
Enter the filename: **add4**
Enter unit name: **incadd**
Enter process name:(default=add4)
Enter the Library name:(default=work)
Enter the Entity name:(default=add4)
Enter the Architecture name:(default=behavior)

The **ptg** will generate two files for each process:

add4.tst --- symbolic tests

add4.vec --- fixed valued tests

3. Entity level test generation

Run **etg**:

%> **etg**

Enter name of the unit: **incadd**

The **etg** will generate two files for a unit:

incadd_sym.test --- symbolic entity tests

incadd.test --- fixed valued entity tests

4. HILO waveform generation

Run **fct** to get DWL file *.dwl (incadd.dwl).

%> **fct**
*VHDL source file (???.vhd):***incadd**
test sequence file (default incadd.test):
please set inputs (exact bits of binary number) :
notes: h ~ all '1' ; l ~ all '0' ; other number ;
*need another set of inputs(y/n)?***n**

5. Integrated menu

The program **tg** provides a menu to guide user use the previous programs.

%> **tg**
please enter the name of the unit:
incadd

----- Unit Name : incadd -----

********** *Select Activity* **********
*
* *1. Generate .tst (test set for process)* *
* *2. Generate .test (test sequence for entity)* *
* *3. Convert to DWL format* *
* *4. Generate All* *

```
*   5. Display                                          *
*   0. Quit                                            *
*                                                      *
********************************************************
Your choice:
```

6. Gate level circuit generation

First, use the Synopsys synthesis tools to get gate level circuit and save it as EDIF netlist format(incadd.edif). Then, use HILO TX_EDIF to get a preliminary GHDL format of the circuit (incadd.cct). Finally, invoke **tx** to get the standard GHDL format (incadd_n.cct).

%> tx incadd.cct

7. Fault grading

First use HILO CIRCUIT to compile the circuit (incadd_n.cct). Then use HILO HISIM to get fault free values. The following blocks in HILO HISIM need to set.

Name: **incadd**

SRC Waveform file: **incadd.dwl**

LOG Log file: **incadd_s.log**

Next invoke **rsd** to get the final DWL waveform for fault simulation (incaddn.dwl)

%> **rsd**
*Old DWL file (???.dwl):***incadd**
HSIM log file (default incadd_s.log):

Finally use the HILO HIFAULT to get fault coverage. The following blocks in HILO HIFAULT need to set.

Name: **incadd**

SRC Waveform file : **incaddn.dwl**

LOG Log file: **incadd_f.log** (or else name)

Appendix B: VHDL models and Test Results

```
use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- ************************************************************
entity csi is
 port (B: in BIT_VECTOR(3 downto 0);
   RST: in BIT;
   CLOCK: in BIT;
   INX: in BIT_VECTOR(3 downto 0);
   OP: out BIT_VECTOR(3 downto 0));
end csi;
-- ************************************************************

architecture BEHAVIORAL of csi is
 signal SEL2: BIT;
 signal SEL1: BIT;
 signal COM: BIT_VECTOR(1 downto 0);
 signal S2: BIT_VECTOR(3 downto 0);
 signal S1: BIT_VECTOR(3 downto 0);
begin

 -- ------------------------------------------------------
 -- Process Name: dec2
 -- ------------------------------------------------------
 dec2_4: process (COM)
 begin
    if COM(1)='0' and COM(0)='0' then
       SEL1<='0';
       SEL2<='0';
    elsif COM(1)='0' and COM(0)='1' then
       SEL1<='0';
       SEL2<='1';
    elsif COM(1)='1' and COM(0)='0' then
       SEL1<='1';
       SEL2<='0';
    else
       SEL1<='1';
       SEL2<='1';
    end if;
```

```
 end process dec2_4;

 -- ---------------------------------------------------
 -- Process Name: shift4
 -- ---------------------------------------------------
 shift4_9: process (SEL2,SEL1,S1)
 begin
   S2 <= SHIFT(S1,SEL2,SEL1);
 end process shift4_9;

 -- ---------------------------------------------------
 -- Process Name: comp4
 -- ---------------------------------------------------
 comp4_15: process (B,S1)
 begin
   COM<=COMP(S1,B);
 end process comp4_15;

 -- ---------------------------------------------------
 -- Process Name: reg
 -- ---------------------------------------------------
 reg_20: process (RST,CLOCK)
 begin
   if RST='1' then
     S1<="0000";
   elsif CLOCK'EVENT and CLOCK='1' then
     S1<=INX;
   end if;
 end process reg_20;

 -- ---------------------------------------------------
 -- Process Name: inc4
 -- ---------------------------------------------------
 inc4_26: process (S2)
 begin
   OP <= INC(S2);
 end process inc4_26;

end BEHAVIORAL;




use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- ********************************************************
entity xyz is
 port (RESET: in BIT;
  CLK: in BIT;
  INZ: in BIT_VECTOR(7 downto 0);
  INY: in BIT_VECTOR(3 downto 0);
  INX: in BIT_VECTOR(3 downto 0);
```

```
   SEL: in BIT;
   F: out BIT_VECTOR(7 downto 0));
end xyz;
-- ************************************************************

architecture BEHAVIORAL of xyz is

 signal RST: BIT;
 signal CLOCK: BIT;
 signal OUTZ: BIT_VECTOR(7 downto 0);
 signal OUTY: BIT_VECTOR(3 downto 0);
 signal OUTX: BIT_VECTOR(3 downto 0);
 signal D_OUT: BIT_VECTOR(7 downto 0);
begin

 -- ---------------------------------------------------
 -- Process Name: RSTBUF
 -- ---------------------------------------------------

 RSTBUF_4: process (RESET)
 begin
   RST<=RESET;
 end process RSTBUF_4;


 -- ---------------------------------------------------
 -- Process Name: BUFCLK
 -- ---------------------------------------------------
 BUFCLK_8: process (CLK)
 begin
   CLOCK<=CLK;
 end process BUFCLK_8;


 -- ---------------------------------------------------
 -- Process Name: regz
 -- ---------------------------------------------------
 regz_12: process (RST,CLOCK)
 begin
   if RST='1' then
     OUTZ<="00000000";
   elsif CLOCK'EVENT and CLOCK='1' then
     OUTZ<= INZ;
   end if;
 end process regz_12;


 -- ---------------------------------------------------
 -- Process Name: regy
 -- ---------------------------------------------------
 regy_18: process (RST,CLOCK)
 begin
```

```
    if RST='1' then
      OUTY<="0000";
    elsif CLOCK'EVENT and CLOCK='1' then
      OUTY<=INY;
    end if;
  end process regy_18;


  -- ---------------------------------------------------
  -- Process Name: regx
  -- ---------------------------------------------------
  regx_24: process (RST,CLOCK)
  begin
    if RST='1' then
       OUTX<="0000";
    elsif CLOCK'EVENT and CLOCK='1' then
      OUTX<=INX;
    end if;
  end process regx_24;


  -- ---------------------------------------------------
  -- Process Name: m_mult4
  -- ---------------------------------------------------
  m_mult4_30: process (SEL,OUTY,OUTX)
  begin
    if SEL='0' then
      D_OUT<=OUTX&OUTY;
    else
      D_OUT<=mult(OUTX,OUTY);
    end if;
  end process m_mult4_30;


  -- ---------------------------------------------------
  -- Process Name: add8
  -- ---------------------------------------------------
  add8_36: process (OUTZ,D_OUT)
  begin
    F <= ADD(D_OUT,OUTZ);
  end process add8_36;

end BEHAVIORAL;




use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- ***********************************************************
entity amd2910b is
```

```
  port (FULL: out TSL;
    G_RESET: in BIT;
    DATAIN: in TSL_VECTOR(1 to 12);
    CLKIN: in BIT;
    CARRYIN: in TSL;
    INTERENABLE: out TSL;
    MAPPINENABLE: out TSL;
    PIPENABLE: out TSL;
    LOAD: in TSL;
    CCEN: in TSL;
    CC: in TSL;
    INSTRUCTION: in BIT_VECTOR(0 to 3);
    OUTADDR: inout TSL_VECTOR(1 to 12));
end amd2910b;
-- ************************************************************

architecture BEHAVIORAL of amd2910b is

  signal STACK_OUT: TSL_VECTOR(1 to 12);
  signal UPC_OUT: TSL_VECTOR(1 to 12);
  signal STACK_CNTL: BIT_VECTOR(0 to 1);
  signal CLOCK: BIT;
  signal GLB_RESET: BIT;
  signal DATA: TSL_VECTOR(1 to 12);
  signal UPC_CNTL: BIT;
  signal REGCNT_ZERO: TSL;
  signal REGCNT_OUT: TSL_VECTOR(1 to 12);
  signal REGCNT_CNTL: BIT_VECTOR(0 to 1);
  signal MUX_CNTL: BIT_VECTOR(0 to 2);
begin

  -- ------------------------------------------------------
  -- Process Name: STACK3
  -- ------------------------------------------------------
  STACK3_4: process (CLOCK,GLB_RESET)
     variable mem: ADDRESS_VECTOR;
     variable stk_ptr: STACK_VECTOR_SIZE;
     variable psh_ptr: STACK_VECTOR_SIZE;
  begin
     if (GLB_RESET = '1') then
          for i in stack_vector_size loop
                mem(i) := "000000000000";
          end loop;
          stk_ptr := stack_vector'low;
          psh_ptr := stack_vector'low;
          FULL <= '0';
          STACK_OUT <= "000000000000";

     elsif (CLOCK'event and CLOCK = '1') then
          FULL <= '0';
          case STACK_CNTL is
                when "00" =>
```

```
                    NULL;
            when "01" =>
                    stk_ptr := stack_vector'LOW;
                    psh_ptr := stack_vector'LOW;
            when "11" =>
                    mem(psh_ptr) := UPC_OUT;
                    if (psh_ptr < stack_vector'HIGH) then
                            stk_ptr := psh_ptr;
                            psh_ptr := INC(psh_ptr);
                    else
                            FULL <= '1';
                            stk_ptr := stack_vector'high;
                    end if;
            when "10" =>
                    if (psh_ptr > stack_vector'LOW and
                            psh_ptr /= stk_ptr) then
                            psh_ptr := DEC(psh_ptr);
                    end if;
                    if (stk_ptr > stack_vector'LOW) then
                            stk_ptr := DEC(stk_ptr);
                    end if;
        end case;

        STACK_OUT<=mem(stk_ptr);

   end if;
end process STACK3_4;


-- ------------------------------------------------------
-- Process Name: GRETBUF
-- ------------------------------------------------------
GRETBUF_15: process (G_RESET)
begin
   GLB_RESET <= G_RESET;
end process GRETBUF_15;


-- ------------------------------------------------------
-- Process Name: DATABUF
-- ------------------------------------------------------
DATABUF_19: process (DATAIN)
begin
   DATA <=DATAIN;
end process DATABUF_19;


-- ------------------------------------------------------
-- Process Name: CLKBUF
-- ------------------------------------------------------
CLKBUF_23: process (CLKIN)
begin
```

```
   CLOCK <= CLKIN;
end process CLKBUF_23;


-- ------------------------------------------------------
-- Process Name: UPC
-- ------------------------------------------------------
UPC_27: process (CLOCK,GLB_RESET)
begin
   if (GLB_RESET = '1') then
        UPC_OUT <= "000000000000";
   elsif (CLOCK'event and CLOCK = '1' ) then
        if (UPC_CNTL = '1') then
                if CARRYIN='1' then
                        UPC_OUT <= INC(OUTADDR);
                else
                        UPC_OUT <= OUTADDR;
                end if;
        else
                UPC_OUT <= "000000000000";
        end if;
   end if;
end process UPC_27;


-- ------------------------------------------------------
-- Process Name: REGCNT2
-- ------------------------------------------------------
REGCNT2_35: process (REGCNT_OUT)
begin
   if (REGCNT_OUT = "000000000000") then
        REGCNT_ZERO <= '1';
   else
        REGCNT_ZERO <= '0';
   end if;
end process REGCNT2_35;


-- ------------------------------------------------------
-- Process Name: REGCNT1
-- ------------------------------------------------------
REGCNT1_39: process (CLOCK,GLB_RESET)
begin
   if (GLB_RESET='1') then
        REGCNT_OUT <= "000000000000";
   elsif (CLOCK'event and CLOCK='1') then
        if REGCNT_CNTL ="01" then
                REGCNT_OUT <= DATA;
        elsif REGCNT_CNTL="10" then
                REGCNT_OUT <= DEC(REGCNT_OUT);
        end if;
   end if;
```

```
end process REGCNT1_39;


-- ------------------------------------------------------
-- Process Name: MUX
-- ------------------------------------------------------
MUX_46: process (UPC_OUT,STACK_OUT,REGCNT_OUT,DATA,MUX_CNTL)
begin
   case MUX_CNTL is
        when "000" => OUTADDR <= DATA;
        when "001" =>OUTADDR <= REGCNT_OUT;
        when "011" =>OUTADDR <= STACK_OUT;
        when "010" => OUTADDR <=UPC_OUT;
        when others => OUTADDR <= "000000000000";
   end case;
end process MUX_46;


-- ------------------------------------------------------
-- Process Name: CONTROL
-- ------------------------------------------------------
CONTROL_54: process (REGCNT_ZERO,LOAD,CCEN,CC,INSTRUCTION)
begin
   PIPENABLE <= '1';
   MAPPINENABLE     <='1';
   INTERENABLE      <='1';
   UPC_CNTL  <='1';
   STACK_CNTL       <="00";
   MUX_CNTL <="010";

   if (LOAD = '1') then
        REGCNT_CNTL <= "00";
   else
        REGCNT_CNTL <= "01";
   end if;

   case INSTRUCTION is
        when "0000" =>
                UPC_CNTL <= '0';
                STACK_CNTL <= "01";
                MUX_CNTL <= "100";
                PIPENABLE <= '0';
        when "0001" =>
                PIPENABLE <= '0';
                if not (CCEN = '0' and CC ='1') then
                        STACK_CNTL <= "11";
                        MUX_CNTL <= "000";
                end if;
        when "0010" =>
                MAPPINENABLE <= '0';
                MUX_CNTL <= "000";
        when "0011" =>
```

```
        PIPENABLE <= '0';
        if not (CCEN='0' and CC='1') then
                MUX_CNTL <="000";
        end if;
when "0100" =>
        STACK_CNTL <= "11";
        PIPENABLE <= '0';
        if not (CCEN='0' and CC='1') then
                REGCNT_CNTL <= "01";
        end if;
when "0101" =>
        PIPENABLE <='0';
        STACK_CNTL <= "11";
        if (CCEN ='0' and CC = '1') then
                MUX_CNTL <= "001";
        else
                MUX_CNTL <= "000";
        end if;
when "0110" =>
        INTERENABLE <= '0';
        if not (CCEN ='0' and CC='1') then
                MUX_CNTL <= "000";
        end if;
when "0111" =>
        PIPENABLE <= '0';
        if (CCEN ='0' and CC ='1') then
                MUX_CNTL <= "001";
        else
                MUX_CNTL <= "000";
        end if;
when "1000" =>
        PIPENABLE <= '0';
        if (REGCNT_ZERO = '0') then
                MUX_CNTL <= "011";
                REGCNT_CNTL <= "10";
        else
                STACK_CNTL <= "10";
        end if;
when "1001" =>
        PIPENABLE <= '0';
        if (REGCNT_ZERO = '0') then
                MUX_CNTL <= "000";
                REGCNT_CNTL <= "10";
        end if;
when "1010" =>
        PIPENABLE <='0';
        if not (CCEN = '0' and CC ='1') then
                MUX_CNTL <= "011";
                STACK_CNTL <= "10";
        end if;
when "1011" =>
        PIPENABLE <='0';
```

```
                if not(CCEN ='0' and CC='1') then
                        MUX_CNTL <="000";
                        STACK_CNTL <= "10";
                end if;
        when "1100" =>
                PIPENABLE <='0';
                REGCNT_CNTL <= "01";
        when "1101" =>
                PIPENABLE <='0';
                if (CCEN='0' and CC='1') then
                        MUX_CNTL <="011";
                else
                        STACK_CNTL <= "10";
                end if;
        when "1110" =>
                PIPENABLE <='0';
        when "1111"=>
                PIPENABLE <= '0';
                if (REGCNT_ZERO ='0') then
                        REGCNT_CNTL <="10";
                        if (CCEN ='0' and CC='1') then
                                MUX_CNTL<="011";
                        else
                                STACK_CNTL <="10";
                        end if;
                else
                        STACK_CNTL <= "10";
                        if (CCEN ='0' and CC='1') then
                                MUX_CNTL <= "000";
                        end if;
                end if;
    end case;
  end process CONTROL_54;

end BEHAVIORAL;
```

Appendix C: Primitive Gate Level GHDL Circuits

```
***********
CCT AO1P (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1;

and(1,1) g1 (w1,A,B);
nor(1,1) g2 (OUT,w1,C,D);

ENDCIRCUIT

***********
CCT AO1 (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1;

and(1,1) g1 (w1,A,B);
nor(1,1) g2 (OUT,w1,C,D);

ENDCIRCUIT

***********
CCT AO2P (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1,w2;

and(1,1) g1 (w1,A,B);
and(1,1) g2 (w2,C,D);
nor(1,1) g3 (OUT,w1,w2);

ENDCIRCUIT

***********
```

```
CCT AO11 (A,B,C,D,E,F,OUT)
input A,B,C,D,E,F;
output OUT;
wire w1,w2,w3;

and(1,1) g1 (w1,A,B);
and(1,1) g2 (w2,C,D);
and(1,1) g3 (w3,E,F);
nor(1,1) g4 (OUT,w1,w2,w3);

ENDCIRCUIT

**********
CCT AO12P (A,B,C,D,E,F,G,H,OUT)
input A,B,C,D,E,F,G,H;
output OUT;
wire w1,w2,w3,w4;

or(1,1) g1 (w1,A,B);
or(1,1) g2 (w2,C,D);
or(1,1) g3 (w3,E,F);
or(1,1) g4 (w4,G,H);
nand(1,1) g5 (OUT,w1,w2,w3,w4);

ENDCIRCUIT

**********
CCT AO2 (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1,w2;

and(1,1) g1 (w1,A,B);
and(1,1) g2 (w2,C,D);
nor(1,1) g3 (OUT,w1,w2);

ENDCIRCUIT

**********
CCT AO3P (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1;
```

```
or(1,1) g1 (w1,A,B);
nand(1,1) g2 (OUT,w1,C,D);

ENDCIRCUIT

**********
CCT AO3 (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1;

or(1,1) g1 (w1,A,B);
nand(1,1) g2 (OUT,w1,C,D);

ENDCIRCUIT

**********
CCT AO4P (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1,w2;

or(1,1) g1 (w1,A,B);
or(1,1) g2 (w2,C,D);
nand(1,1) g3 (OUT,w1,w2);

ENDCIRCUIT

**********
CCT AO4 (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1,w2;

or(1,1) g1 (w1,A,B);
or(1,1) g2 (w2,C,D);
nand(1,1) g3 (OUT,w1,w2);

ENDCIRCUIT

**********
CCT AO6 (A,B,C,OUT)
```

```
input A,B,C;
output OUT;
wire w1;

and(1,1) g1 (w1,A,B);
nor(1,1) g2 (OUT,w1,C);

ENDCIRCUIT

**********
CCT AO6P (A,B,C,OUT)
input A,B,C;
output OUT;
wire w1;

and(1,1) g1 (w1,A,B);
nor(1,1) g2 (OUT,w1,C);

ENDCIRCUIT

**********
CCT AO7P (A,B,C,OUT)
input A,B,C;
output OUT;
wire w1;

or(1,1) g1 (w1,A,B);
nand(1,1) g2 (OUT,w1,C);

ENDCIRCUIT

**********
CCT AO7 (A,B,C,OUT)
input A,B,C;
output OUT;
wire w1;

or(1,1) g1 (w1,A,B);
nand(1,1) g2 (OUT,w1,C);

ENDCIRCUIT

**********
```

```
CCT EO1 (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1,w2;

and(1,1) g1 (w1,A,B);
nor(1,1) g2 (w2,C,D);
nor(1,1) g3 (OUT,w1,w2);

ENDCIRCUIT

**********
CCT EON1 (A,B,C,D,OUT)
input A,B,C,D;
output OUT;
wire w1,w2;

or(1,1) g1 (w1,A,B);
nand(1,1) g2 (w2,C,D);
nand(1,1) g3 (OUT,w1,w2);

ENDCIRCUIT

**********
CCT LD1 (D,G,Q,QN)
input D,G;
output Q,QN;
wire w1,w2,w3;

and (1,1) g1 (w1,D,G);
and (1,1) g2 (w3,w2,Q);

not (1,1) g3 (w2,G);
not (1,1) g4 (QN,Q);

or (1,1) g5 (Q,w1,w3);

ENDCIRCUIT

**********
CCT LD2 (D,G,Q,QN);
input D,G;
output Q,QN;
```

```
wire w1,w2,w3;

and (1,1) g1 (w1,D,w2);
and (1,1) g2 (w3,G,Q);

not (1,1) g3 (w2,G);
not (1,1) g4 (QN,Q);

or (1,1) g5 (Q,w1,w3);


ENDCIRCUIT

**********
CCT LD3 (D,G,CD,Q,QN);
input D,G;
output Q,QN;
wire w1,w2,w3;

and (1,1) g1 (w1,D,w2,CD);
and (1,1) g2 (w3,G,Q,CD);

not (1,1) g3 (w2,G);
not (1,1) g4 (QN,Q);

or (1,1) g5 (Q,w1,w3);

ENDCIRCUIT

**********
CCT FD1(D,CLK,Q,QN);
input D,CLK;
output Q,QN;
wire w1,w2,w3,w4;

NAND (1,1)
        g1 (w1, w4, w2),
        g2 (w2, CLK, w1),
        g3 (w3, w4, CLK, w2),
        g4 (w4, D, w3),
        g5 (Q, w2, QN),
        g6 (QN, w3, Q);
```

```
ENDCIRCUIT

**********
CCT FD2(D,CLK,CLR,Q,QN);
input D,CLK,CLR;
output Q,QN;
wire w1,w2,w3,w4;

NAND (1,1)
        g1 (w1, w4, w2),
        g2 (w2, CLR, CLK, w1),
        g3 (w3, w4, CLK, w2),
        g4 (w4, D, CLR, w3),
        g5 (Q, w2, QN),
        g6 (QN, CLR, w3, Q);


ENDCIRCUIT

**********
CCT FD2P(D,CLK,CLR,Q,QN);
input D,CLK,CLR;
output Q,QN;
wire w1,w2,w3,w4;

NAND (1,1)
        g1 (w1, w4, w2),
        g2 (w2, CLR, CLK, w1),
        g3 (w3, w4, CLK, w2),
        g4 (w4, D, CLR, w3),
        g5 (Q, w2, QN),
        g6 (QN, CLR, w3, Q);


ENDCIRCUIT

**********
CCT FD2S(D,CLK,CLR,TI,TE,Q,QN);
input D,CLK,CLR,TI,TE;
output Q,QN;
wire w1,w2,w3,w4,w5,w6,w7,w8;
```

```
NAND (1,1)
        g1 (w1, w4, w2),
        g2 (w2, CLR, CLK, w1),
        g3 (w3, w4, CLK, w2),
        g4 (w4, w8, CLR, w3),
        g5 (Q, w2, QN),
        g6 (QN, CLR, w3, Q);
not(1,1)
            g7(w5,TE);
and(1,1)
            g8(w6,D,W5),
            g9(w7,TI,TE);
or(1,1)
            g10(w8,w6,w7);


ENDCIRCUIT

**********************
CCT FD3(D,CLK,CLR,SET,Q,QN);
input D,CLK,CLR,SET;
output Q,QN;
wire w1,w2,w3,w4,w5,w6;

NAND (1,1)
        g1 (w1, w4, w2, SET),
        g2 (w2, CLR, CLK, w1),
        g3 (w3, w4, CLK, w2),
        g4 (w4, D, CLR, w3),
        g5 (w5, w2, w6, SET),
        g6 (w6, CLR, w3, w5);
not(1,1)
            g7(QN,w5),
            g8(Q,w6);

ENDCIRCUIT

**********************
CCT FD4(D,CLK,SET,Q,QN);
input D,CLK,SET;
output Q,QN;
wire w1,w2,w3,w4,w5,w6;
```

```
NAND (1,1)
        g1 (w1, w4, w2, SET),
        g2 (w2, CLK, w1),
        g3 (w3, w4, CLK, w2),
        g4 (w4, D, w3),
        g5 (w5, w2, w6, SET),
        g6 (w6, w3, w5);
not(1,1)
            g7(QN,w5),
            g8(Q,w6);

ENDCIRCUIT

**********************

CCT MUX21L(A,B,S,Q);
input A,B,S;
output Q;
wire w1,w2,w3;

AND (1,1)  g1 (w1, A, w3),
      g2 (w2, B, S);
NOT (1,1)  g3 (w3, S);
NOR (1,1)   g4 (Q, w1, w2);

ENDCIRCUIT

**********************
CCT MUX21LP(A,B,S,Q);
input A,B,S;
output Q;
wire w1,w2,w3;

AND (1,1)  g1 (w1, A, w3),
      g2 (w2, B, S);
NOT (1,1)  g3 (w3, S);
NOR (1,1)   g4 (Q, w1, w2);

ENDCIRCUIT

**********************
CCT MUX21H(A,B,S,Q);
input A,B,S;
```

```
output Q;
wire w1,w2,w3;

AND (1,1)  g1 (w1, A, w3),
        g2 (w2, B, S);
NOT (1,1)  g3 (w3, S);
OR (1,1)   g4 (Q, w1, w2);

ENDCIRCUIT

**********************
CCT MUX21HP(A,B,S,Q);
input A,B,S;
output Q;
wire w1,w2,w3;

AND (1,1)  g1 (w1, A, w3),
        g2 (w2, B, S);
NOT (1,1)  g3 (w3, S);
OR (1,1)   g4 (Q, w1, w2);

ENDCIRCUIT

****************************
CCT FJK3(J,K,CLK,CLR,SET,Q,QN);
input J,K,CLK,CLR,SET;
output Q,QN;
wire w1,w2,w3,w4,w5,w6;

NAND (1,1)
            g1 (w1, J, CLK, QN),
            g2 (w2, K, CLK, Q),
            g3 (w3, w1,w4,SET),
            g4 (w4, w2,w3,CLR),
            g5 (w5, w1, w3),
            g6 (w6, w4, w2),
            g7 (Q, w5,QN,SET),
            g8 (QN,w6,Q,CLR);

ENDCIRCUIT

**********************
CCT FJK2(J,K,CLK,CLR,Q,QN);
```

```
input J,K,CLK,CLR;
output Q,QN;
wire w1,w2,w3,w4,w5,w6;

NAND (1,1)
        g1 (w1, J, CLK, QN),
        g2 (w2, K, CLK, Q),
        g3 (w3, w1,w4),
        g4 (w4, w2,w3,CLR),
        g5 (w5, w1, w3),
        g6 (w6, w4, w2),
            g7 (Q, w5,QN),
            g8 (QN,w6,Q,CLR);

ENDCIRCUIT

**********************
CCT IVDA (A,OUT1,OUT)
input A;
output OUT,OUT1;

not(1,1) g1 (OUT1,A),
    g2 (OUT,OUT1);

ENDCIRCUIT

************
CCT IVDAP (A,OUT1,OUT)
input A;
output OUT,OUT1;

not(1,1) g1 (OUT1,A),
    g2 (OUT,OUT1);

ENDCIRCUIT

************
CCT HA1 (A,B,SUM,CY)
input A,B;
output SUM,CY;

xor(1,1) g1 (SUM,A,B);
and(1,1) g2 (CY,A,B);
```

```
ENDCIRCUIT

**********
CCT B3I (A,OUT1,OUT)
input A;
output OUT,OUT1;

not(1,1) g1 (OUT1,A),
      g2 (OUT,OUT1);

ENDCIRCUIT

**********
CCT B3IP (A,OUT1,OUT)
input A;
output OUT,OUT1;

not(1,1) g1 (OUT1,A),
      g2 (OUT,OUT1);

ENDCIRCUIT

**********
CCT MUX31L(D0,D1,D2,A,B,Q);
input D0,D1,D2,A,B;
output Q;
wire w1,w2,w3,w4,w5;

NAND (1,1) g1 (Q, w3,w4,w5);
OR (1,1)   g2 (w3, D0, A, B),
        g3 (w4, D1, w1,B),
           g4 (w5, D2, w2);
NOT (1,1)  g5 (w1, A),
           g6 (w2, B);

ENDCIRCUIT

**********
CCT MUX31LP(D0,D1,D2,A,B,Q);
input D0,D1,D2,A,B;
output Q;
wire w1,w2,w3,w4,w5;
```

```
NAND (1,1) g1 (Q, w3,w4,w5);
OR (1,1)   g2 (w3, D0, A, B),
        g3 (w4, D1, w1,B),
           g4 (w5, D2, w2);
NOT (1,1)  g5 (w1, A),
           g6 (w2, B);

ENDCIRCUIT

**********************
CCT AO5 (A,B,C,OUT)
input A,B,C;
output OUT;
wire w1,w2,w3;

and(1,1) g1 (w1,A,B),
       g2 (w2,A,C),
          g3 (w3,B,C);
nor(1,1) g4 (OUT,w1,w2,w3);

ENDCIRCUIT

***********
CCT AO5P (A,B,C,OUT)
input A,B,C;
output OUT;
wire w1,w2,w3;

and(1,1) g1 (w1,A,B),
       g2 (w2,A,C),
          g3 (w3,B,C);
nor(1,1) g4 (OUT,w1,w2,w3);

ENDCIRCUIT
```

Vita

Wencheng Li was born on October 19, 1961 in ShanXi , China. He entered the Dalian University of Technology, Dalian, China, in October 1978 and received a Bachelor of Engineering degree in Electronic Engineering in July 1982. He then entered graduate school at Dalian University of Technology in September 1982, receiving a Master of Engineering degree in Electronic Engineering in August 1985. He attended graduate school at the Virginia Polytechnic Institute and State University from August 1993 to June 1996, receiving a Master of Science degree and a Ph.D. degree in Electrical Engineering in April 1995 and June 1996 respectively.

From August 1985 to July 1987 he was a Teaching Assistant at Dalian University of Technology, and from August 1987 to July 1993 he was a Research Engineer and Instructor at the same university. He engaged in researches of testing and fault diagnosis, expert systems, microprocessor and microcomputer applications, and controls.

His current interests include modeling with VHDL, VLSI design, testing, synthesis, computer architecture, and CAD tool development.