REGISTER-TRANSFER LEVEL FAULT MODELING AND TEST EVALUATION TECHNIQUE FOR VLSI CIRCUITS

by

Pradipkumar Arunbhai Thaker

B.E. December 1989, The Maharaja Sayajirao University (India)

M.S. May 1993, The George Washington University

A Dissertation submitted to

The Faculty of

The Department of Electrical and Computer Engineering

of The George Washington University in partial satisfaction

of the requirements for the degree of Doctor of Science

May 21, 2000

Dissertation directed by

Mona E. Zaghloul

Professor of Engineering and Applied Science

© Copyright by

PRADIPKUMAR ARUNBHAI THAKER

2000

All Rights Reserved

ABSTRACT

Test patterns for large VLSI systems are often determined from the knowledge of the circuit function. A fault simulator is then used to find the effectiveness of the test patterns in detecting gate-level “stuck-at” faults. Existing gate-level fault simulation techniques suffer prohibitively expensive performance penalties when applied to the modern VLSI systems of larger sizes. Also, post-synthesis findings of such test generation and fault simulation efforts are too late in the design cycle to be useful for Design-For-Test (DFT) related improvements in the architecture. Therefore, an effective Register-Transfer Level (RTL) fault model is highly desirable.

In this thesis, a novel procedure that supports RTL fault simulation and generates an estimate of the gate-level fault coverage for a given set of test patterns is proposed. This procedure is based on new RTL fault model, fault-injection algorithm, application of stratified sampling theory, and stratum weight extraction techniques. The VLSI system consists of interconnections of modules described in an RTL language. The proposed RTL fault model and the fault-injection algorithm are developed such that the RTL fault-list of a module becomes a representative sample of the collapsed gate-level fault-list. In other words, the RTL faults of a module have a distribution of detection probabilities similar to that of the collapsed gate-level faults. The RTL fault coverage of the proposed fault model tracks the gate-level fault coverage within error bounds predicted by the random sampling technique. An application of the stratified sampling theory supports RTL fault modeling for VLSI systems that consist of interconnected modules. The RTL fault coverages of all modules in a VLSI system are added according to their respective stratum weights as per the stratified sampling theory. Several stratum weight extraction techniques are developed to support the application of the stratified sampling theory to the RTL fault modeling for VLSI systems. The stratified RTL fault coverage serves as an estimate of the gate-level fault coverage of the VLSI system within statistical error bounds.

Performing fault simulation at the RT level using the proposed procedure has several advantages: (a) a significant performance gain in fault simulation compared to the prevailing gate-level approach, (b) the possibility of improving tests prior to logic synthesis, and (c) the early detection of testability problems enabling design for testability in the pre-synthesis phase of the VLSI design cycle. These significant advantages give the proposed procedure potential for application.

To my parents

Urmilaben and Arunbhai

ACKNOWLEDGEMENT

This thesis has become a reality due to the generous help and support of many individuals. It is my pleasure to acknowledge and thank them for their contributions, which are embodied in concept, thoughts and goals of this dissertation.

I sincerely thank my advisor, Dr. Mona E. Zaghloul, whose technical advice, guidance and encouragement made my doctoral studies a meaningful learning experience. I thank the other members of the examination committee -- Dr. Vishwani D. Agrawal, Prof. Arthur D. Friedman, Prof. Abdou Youssef, and Prof. Can E. Korman -- for the time and effort spent by them in serving on my thesis committee. I am grateful to the administrative staff -- particularly Ms. Merilyn Henry and Ms. Deborah Swanson -- at the George Washington University for being exceptionally supportive during my graduate studies.

I express my deepest appreciation and gratitude to Dr. Vishwani D. Agrawal of Lucent Technologies for having spent much time educating me in the area of design-for-test. His expertise in testing and his generous guidance throughout this research were crucial in steering me toward the completion of this work. I will be indebted to him forever for his kindness and generosity.

I especially thank Dr. Arthur D. Friedman for his time and effort in reviewing the progress of this dissertation and providing valuable feedback. I also thank Dr. Rabindra K. Roy for intriguing technical conversations and encouragement.

I thank my present employer Hughes Network Systems (HNS) for partially supporting the research presented in this dissertation through research and development fund. I am grateful to Mr. Larry Blue and Mr. Ted Hockey at HNS for motivating me, for providing computer equipment, and for granting me time-off from work to complete this research. I also thank Mr. John Vogel and his associates at Cadence® Design Systems for granting fault-simulator licenses for the purpose of this research and for providing technical support during the usage of *Verifault-XL™*.

I acknowledge with gratitude and affection, the love, encouragement and support given to me by my parents, my Guru Shri P. Rajagopalachari, my brother Darshit, and my wife Pragna, during my graduate studies. A seed of ambition sown by my late grandfather Chimanbhai in my young mind for the pursuit of doctoral studies has finally come to fruition. The immense

sacrifices made by my parents for the education of their children are incomparable. I can never thank them enough for all they have done for me. One extra line of appreciation for my mother for being a great friend, philosopher and guide. I owe much of my success to her for imbibing the values of hard work and persistence in me. Last but not the least, I thank my dear wife Pragna for being very patient and supportive. I promise to make up for the quantity as well as in quality for the precious time lost during the first five years of our marriage. I conclude by thanking my aunt and uncle, Mrs. and Mr. Bhatt for their loving and caring support during my initial years as a student in the United States of America.

TABLE OF CONTENTS

	page no.
ABSTRACT	iii
DEDICATION	iv
ACKNOWLEDGEMENT	v
TABLE OF CONTENTS	vii
LIST OF FIGURES	ix
LIST OF TABLES	x
LIST OF SYMBOLS	xi
CHAPTER 1 INTRODUCTION	1
1.1 Role of Testing in VLSI Design	1
1.2 Fault Modeling and Test Generation	2
1.3 Fault Simulation and Test Evaluation	5
1.4 Design-for-Test	6
1.5 RTL-based Top-Down Design and Test Methodology	7
1.6 Goal of Thesis Research	9
1.7 Overview of the Thesis	9
1.8 Organization of the Thesis	12
CHAPTER 2 REVIEW OF THE LITERATURE	13
CHAPTER 3 RTL FAULT MODEL, FAULT-INJECTION ALGORITHM AND FAULT SIMULATION METHOD	17
3.1 RT Level Hardware Design Modeling	17
3.2 RTL Fault Model and Fault-injection Algorithm	23
3.2.1 Fault Modeling for Synthetic Operators	26
3.2.2 Fault Modeling for Boolean Operators	27
3.2.3 Fault Modeling for Logical Operators	28
3.2.4 Fault Modeling for Sequential Elements	28
3.2.5 Stem and Fan-out Fault Modeling	29
3.2.6 RTL Fault Collapsing	32
3.3 RTL Fault Simulation Method	32

3.4 Study of Efficacy of RTL Fault Model and Fault-injection Algorithm 33
3.4.1 Comparison of RTL and Gate Fault Lists 33
3.4.2 Comparison of RTL and Gate Fault Coverages 44
CHAPTER 4 STRATIFIED SAMPLING: THEORY AND APPLICATION 54
4.1 Stratified Sampling Theory 55
4.2 Application of Stratified Sampling to RTL Fault Modeling 58
CHAPTER 5 STRATUM WEIGHT EXTRACTION TECHNIQES 64
5.1 Technology-dependent Stratum Weight Extraction 64
5.2 Technology-independent Stratum Weight Extraction 65
5.2.1 Weights Based on Logic Synthesis 65
5.2.2 Weights Based on Entropy Measure 68
5.2.3 Weights Based on Floor-planning 68
CHAPTER 6 EXPERIMENTAL PROCEDURE AND RESULTS 69
6.1 Estimation of the Gate-level Fault Coverage of a VLSI System 69
6.1.1 Procedure 69
6.1.2 Experimental Data 70
6.1.3 Discussion 73
6.2 Stratum Weight Estimation 75
6.3 Performance Gain 78
CHAPTER 7 CONCLUSIONS AND FUTURE WORK 80
7.1 Conclusions 80
7.2 Future Work 83
REFERENCES 84
APPENDIX A VALIDATION VECTOR GRADE A-1
APPENDIX B C++ CODE FOR FAULT-INJECTION ALGORITHM B-1

LIST OF FIGURES

	page no.
Figure 1 Cost of Failure	2
Figure 2 Logical Fault Model	4
Figure 3 RTL-based Top-Down Design Methodology	8
Figure 4 RTL Description of a 16-bit Counter	18
Figure 5 RTL Description of a 4-to-1 Multiplexer	19
Figure 6 Design Hierarchy	22
Figure 7 Example of Fault Detection Probability Distribution	24
Figure 8 Example of Fault Modeling for Synthetic Operators	26
Figure 9 Example of Fault Modeling for Boolean Operators	27
Figure 10 Example of Fault Modeling for Logical Operators	28
Figure 11 Example of Fault Modeling for Sequential Elements	29
Figure 12 RTL Modeling of Stem and Fan-out Faults	31
Figure 13 RTL and Gate-level Implementations of 2-to-1 Mux (Example 1)	35
Figure 14 RTL and Gate-level Implementations of 2-to-1 Mux (Example 2)	38
Figure 15 RTL Code Example	41
Figure 16 RTL Faults in a Symbolic Representation	42
Figure 17 Fault-injected RTL Code	43
Figure 18 Gate Level Schematic: 4-bit Counter Module	46
Figure 19 RTL and Gate Level Fault Coverage vs. Test Vectors: Transmit Buffer Module	48
Figure 20 Error vs. Fault Coverage: Transmit Buffer Module	48
Figure 21 RTL and Gate Level Fault Coverage vs. Test Vectors: SDRAM Controller Module	50
Figure 22 Error vs. Fault Coverage: SDRAM Controller Module	50
Figure 23 RTL and Gate Level Fault Coverage vs. Test Vectors: DSP Interface Module	52
Figure 24 Error vs. Fault Coverage: DSP Interface Module	52
Figure 25 Logic Synthesis Flow	67

LIST OF TABLES

page no.

Table 1 Verilog RTL Constructs 21

Table 2 RTL-Gate Fault Equivalence for Implementation I (Example 1) 36

Table 3 RTL-Gate Fault Equivalence for Implementation II (Example 1) 36

Table 4 RTL-Gate Fault Equivalence for Implementation I (Example 2) 39

Table 5 RTL-Gate Fault Equivalence for Implementation II (Example 2) 39

Table 6 Empirical Data: 4-bit Counter Module 45

Table 7 Empirical Data: Transmit Buffer Module 47

Table 8 Empirical Data: SDRAM Controller Module 49

Table 9 Empirical Data: DSP Interface Module 51

Table 10 Inaccurate Estimation of Gate Fault Coverage in a VLSI System 55

Table 11 Values of t for Range of Estimator 58

Table 12 Empirical Data: Frame-Timing-Control ASIC 71

Table 13 Empirical Data: System Timing Controller ASIC 72

Table 14 Empirical Data: Digital Signal Processor ASIC 74

Table 15 Estimation Error in Stratum Weights: Design D3 76

Table 16 Impact of Error in Stratum Weights on Stratified RTL Fault Coverage and Error Bounds: Design D3 77

Table 17 Performance Comparison 78

Table 18 Software Code-Coverage Metrics vs. VVG A-5

LIST OF SYMBOLS

ASIC	Application Specific Integrated Circuit
ATE	Automatic Test Equipment
ATPG	Automatic Test Pattern Generation
BIST	Built-in Self-Test
CDFG	Control Data Flow Graph
CMOS	Complementary Metal Oxide Silicon
DFT	Design-for-Test
DUT	Device-Under-Test
HDL	Hardware Description Language
HLTS	High Level Test Synthesis
IC	Integrated Circuit
RTL	Register Transfer Level
SSF	Single Stuck Fault
VLSI	Very Large Scale Integration
VVG	Validation Vector Grade

CHAPTER 1

INTRODUCTION

Recent innovations in the Complementary Metal Oxide Silicon (CMOS) technology have enabled the fabrication of large integrated circuits. The high-level design methodology [BHA99] based on the Register Transfer Level (RTL) design description provides means to design large and complex Application Specific Integrated Circuits (ASICs) in a reasonable time. With the growing size and complexity of Very Large Scale Integration (VLSI) circuits, it has become increasingly challenging to deliver a circuit that is free from any defects or flaws. Validation and test are two essential steps that help to ensure that the final product in the form of a fabricated Integrated Circuit (IC) meets high quality requirements. Comprehensive validation procedures -- formal verification methods and heuristic validation techniques like simulation-based code coverage metrics -- provide a high degree of confidence about the circuit being functionally correct [KUR94, CLA99]. Test addresses issues related to the screening of fabricated devices for manufacturing related defects. This thesis makes a primary contribution to the area of test and a secondary contribution to validation (see Appendix A).

1.1 Role of Testing in VLSI Design

Even though a circuit is designed error-free, manufactured circuits may not function correctly. Since the manufacturing process is not perfect, some defects such as short-circuits, open-circuits, open interconnections, pin shorts, etc., may be introduced. Davis [DAV82] points out that the cost of detecting a faulty component increases ten times at each step between pre-package component test and system warranty repair (see Figure 1). It is important to identify a faulty component as early in the manufacturing process as possible. Therefore, testing has become a very important aspect of any VLSI manufacturing system.

The testing of digital logic involves the application of the appropriate stimuli to a Device-Under-Test (DUT) and the comparison of the resulting response to the expected one. Manufacturing defects tend to alter the circuit behavior and, therefore, when the response of a DUT does not match the expected response, it is considered faulty. For digital circuits, the

stimuli are sequences of logic levels 0 and 1, called test patterns or vectors that are applied to the inputs of the circuit. Test pattern generation is a complex process with three main aspects: the cost of test generation, the cost of test application and the quality of test.

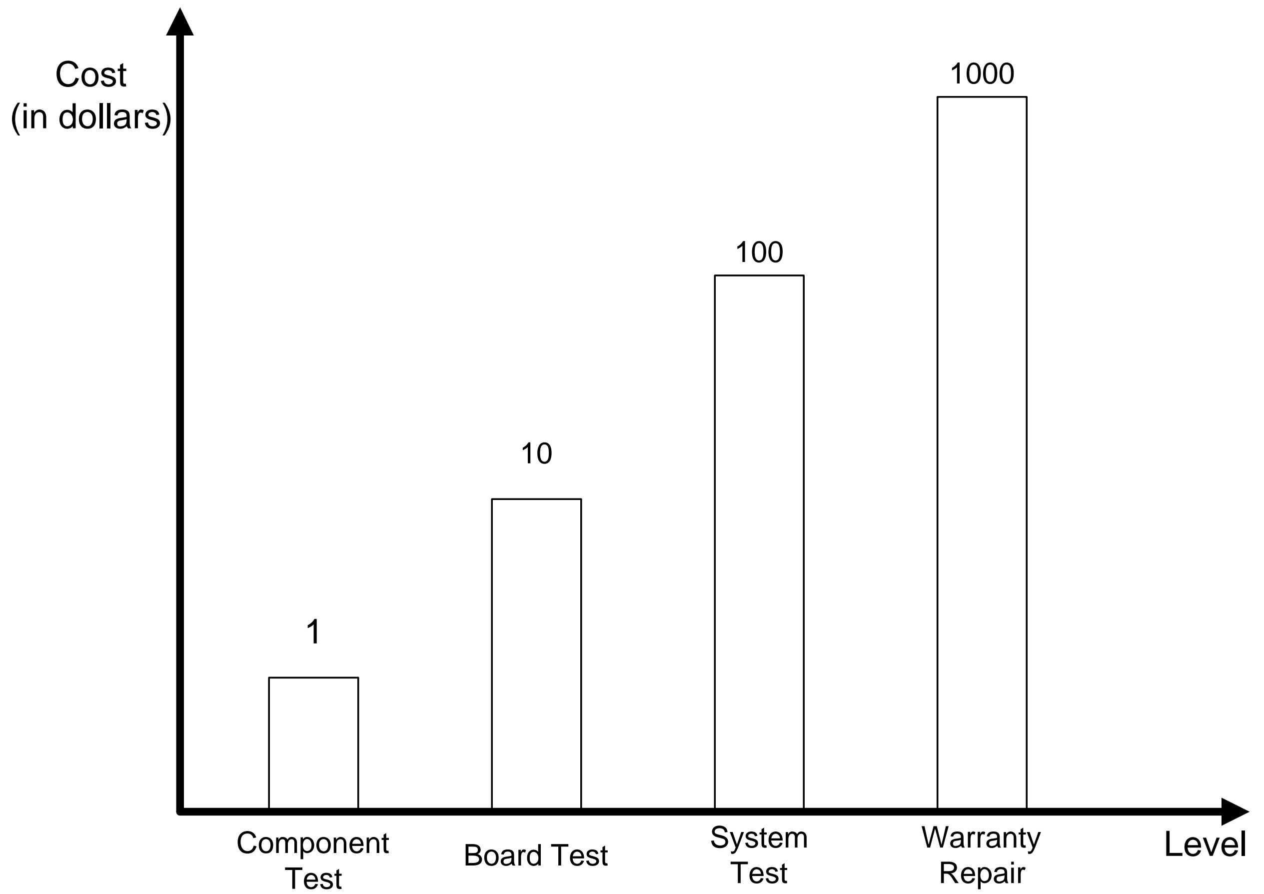


Figure 1 Cost of Failure

1.2 Fault Modeling and Test Generation

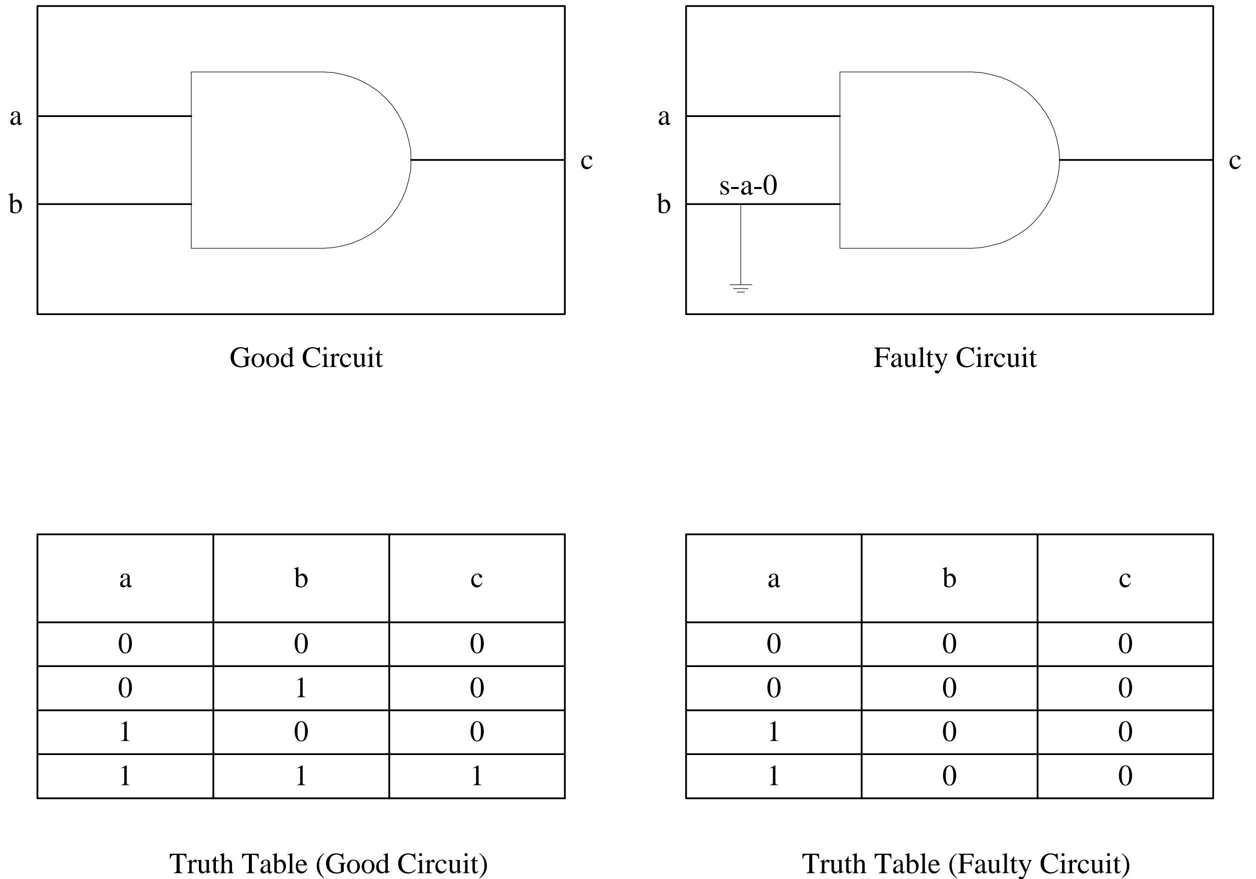
The creation of effective stimuli is an important part of test. Miczo [MIC86] states that the early test generation approaches involved either the creation of all possible combinations for device inputs or the creation of stimuli targeted to verify certain functional features of the device.

The application of 2^n test vectors to a device with n inputs is effective if n is small and the device does not contain memory elements. Modern VLSI circuits are large and complex; they have outgrown both of these restrictions. An exhaustive test pattern set for a modern VLSI circuit could result into a hundred million or more vectors. Each circuit is tested for a given set of test patterns using Automatic Test Equipment (ATE). This equipment is very expensive and contains a limited amount of memory to store test patterns. The amount of time that each circuit requires for testing is also an important factor in determining the overall test cost. In order to reduce the test application cost (which primarily comprises of the tester time and the cost of test equipment), it is important to keep the number of test patterns to a minimum.

Eldred [ELD59] suggested an efficient test generation approach that targets hardware faults rather than the function. This is done by creating test patterns for specific faults. Commonly occurring physical faults are represented by logical fault models. Logical faults represent the effect of physical faults on the behavior of the digital circuit. Then, input stimuli are created to distinguish between the fault-free and faulty circuits. Test pattern generation based on the logical fault model assumes the presence of a single fault in the circuit at a given time. Test patterns derived under the single-fault assumption are generally considered useful for detecting multiple faults because a test derived for an individual single fault can detect a multiple fault containing that single fault as a component. There are, however, specific multiple faults where the components can mask each other and detection by a single fault test is not guaranteed [ABR90].

Among many different types of logical fault models, the "stuck at" model is prominently used for test pattern generation and evaluation. "Stuck-at" is a gate level fault model, which assumes Boolean components to be fault-free and only their input or output ports to be fixed at a specific logic level. When the faulty logic level is 0, the fault is called stuck-at-zero (s-a-0) and when the faulty logic level is 1, the fault is called stuck-at-one (s-a-1). Under the single fault assumption, only one fault is applied at a time when a test set is being created. This approach is called "Single Stuck Fault (SSF)" modeling. For example, a good circuit and the corresponding faulty circuit are shown in Figure 2. A stuck-at-zero fault is being applied on port b of the AND gate. The test that can distinguish between the good and faulty circuits is the one in which the logic value applied at input ports a and b is 1 and the expected response at the output port c is also 1. In the presence of a s-a-0 fault on port b, the response will be altered to the logic value of

0. When this test is applied for device testing, it will be able to identify components with many physical defects that are modeled by this logical fault. The same test vector may also detect other faults in the circuit that produce an identical faulty behavior.



a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

Truth Table (Good Circuit)

a	b	c
0	0	0
0	0	0
1	0	0
1	0	0

Truth Table (Faulty Circuit)

Figure 2 Logical Fault Model

Popular test generation approaches based on the SSF model can be classified into random and deterministic methods [ABR90]. Random test generation depends only on the number of circuit inputs and works without taking into account the topology of the circuit to be tested [AGR88]. Deterministic test generation produces tests by processing a model of the circuit. Deterministic test generation can produce shorter tests of a higher quality in comparison to the random test generation approach. Deterministic test generation can be fault-oriented or fault-independent. Tests are generated for specific faults from a given fault universe in fault-oriented deterministic test generation. The goal of the fault-independent deterministic test generation

approach is to derive a set of tests that detect a large set of faults in the fault universe without targeting individual faults. The test generation process can be manual or automatic. The cost of the tools used and the time spent during test generation add to the overall cost of testing.

1.3 Fault Simulation and Test Evaluation

Fault simulation is the process of simulating a circuit with a given set of test patterns and a set of faults, and comparing the response of the circuit with each fault to that of the fault-free circuit. If the response does not match, the fault is considered detected by the given set of test patterns. Thus, fault simulation can be used to evaluate or grade test patterns. The quality of test patterns is determined by fault coverage, which is a ratio of the number of detected faults to the total number of simulated faults. Fault coverage bears a nonlinear relationship to defect coverage, which is the probability that a test set detects any physical fault in the circuit. According to Abramovici et al. [ABR90], a test with high coverage for SSFs also achieves high defect coverage. The quality of the test influences the quality of shipped ASICs. The probability of a defective ASIC passing a test whose defect coverage is *T*, is given by Williams and Brown [WIL81] as,

$$\textit{Defect Level (DL)} \;=\; 1 - Y^{(1-T)} \tag{1.1}$$

In the equation (1.1), *Y* indicates the true yield, that is, the probability that a manufactured circuit is defect-free. *DL* indicates the defect level which is the probability of shipping a defective circuit, and *T* is the defect coverage. Directly substituting the fault coverage for *T* in the equation (1.1) can lead to a very pessimistic defect level. In general a real defect in a device can produce several faults. Agrawal et al. [AGR82] assumed a statistical relationship between stuck-at faults and defects. In their analysis, a defect, on an average, produces n_0 stuck-at faults. Then, the defect level is given by

$$DL = \frac{(1-f)(1-Y)e^{-(n_0-1)f}}{Y + (1-f)(1-Y)e^{-(n_0-1)f}} \tag{1.2}$$

where f is the fault coverage determined by a fault simulator. The value of n_0 is related to the technology, the design style and the processing parameters. Agrawal et al. [AGR82] derived it from the test data. For any realistic values of Y and n_0, equation (1.2) shows that to achieve a DL lower than 500 parts per million (a desirable quality level for commercial parts), the fault coverage must be in the range of 98-100%. Other more complex models of defect level have been given by Das et al. [DAS93].

In addition to providing a measure of test quality, fault simulation also plays an important role in test generation. Many test generation systems use a fault simulator to evaluate tests, and make changes according to the results of fault simulation until the obtained coverage is considered satisfactory. Tests are modified by adding new patterns and/or by discarding some of the patterns that did not contribute to the increase in coverage. These changes may be made automatically by a program or in an interactive mode by a test engineer. Often test vectors generated by a fault-oriented algorithm for a specified fault may detect many other faults. A fault simulator is essential for finding such information.

1.4 Design-for-Test

It is important to achieve very high fault coverage (in the 95-100% range) for a given circuit with a minimum number of test patterns. The high quality test offers product reliability at a minimum test application cost. However, there is a cost associated with generating such good quality tests. The test generation cost depends on the complexity of the circuit. Methods of reducing the complexity of a circuit for testing purposes are referred to as design-for-testability techniques. Circuits that are highly testable offer easier test access (controllability from primary inputs and observability at primary outputs) for internal nodes.

Structural analysis techniques are often used to compute numerical measures for the testability of the internal nodes of circuits [GOL79]. Designs with insufficient testability measures rarely attain high fault coverage even with a large number of test patterns. Besides the higher costs associated with longer test times and expensive test equipment, the lack of good coverage results in the poor screening of manufacturing defects. As a result, the compromised quality of the manufactured ASICs reduces the reliability of the product that uses those components. Such products have higher failure rates, increasing overall costs. The testability

problems of a design can be identified using analysis techniques such as the SCOAP algorithm described by Goldstein [GOL79] or by fault simulation-based test generation. SCOAP calculates the controllability and observability measures for each node in the circuit. Circuits that are not test-friendly require architectural and/or structural design changes to improve testability. These changes may be guided by the testability measure, which can often point to areas of poor testability in the circuit.

1.5 RTL-based Top-Down Design and Test Methodology

The RTL-based top-down design methodology as described by Bhatnagar [BHA99] (see Figure 3) enables the design of the state-of-the-art ASICs in a reasonable time. The design cycle starts with the identification of functional specifications. Behavioral modeling is the next step followed by an RT level description of individual modules that make up the ASIC. Modeling a circuit's function at the behavioral level means modeling functionality without regard to the hardware structure, electrical signals and detailed timing. Such models are useful for proof-of-concept. In a register-transfer level model, the operations of a sequential circuit are described as synchronous transfers controlled by an ideal clock between functional modules. The RTL modules are validated as stand-alone components before integrating them into an ASIC. The RT level validation effort is followed by logic synthesis. Logic synthesis plays a critical role in the RTL-based top-down design methodology. Logic synthesis transforms the RTL description into an optimized technology-specific hardware description, generally in the form of a gate-level netlist (connectivity description of Boolean gates). Logic synthesis iterations with different optimization constraints can produce different gate-level representations from the same RTL description. Therefore, the gate level structure of the design stabilizes only after the synthesized circuit has been verified through logic simulation. Thus, a verified netlist is produced.

Once the design is verified, gate-level SSF models are used for test generation and fault simulation using the technology-specific (gate-level) netlist. In addition, the gate-level netlist serves as a common database for various post-synthesis steps such as timing simulation, placement, routing, static timing analysis, etc., until a prototype is fabricated.

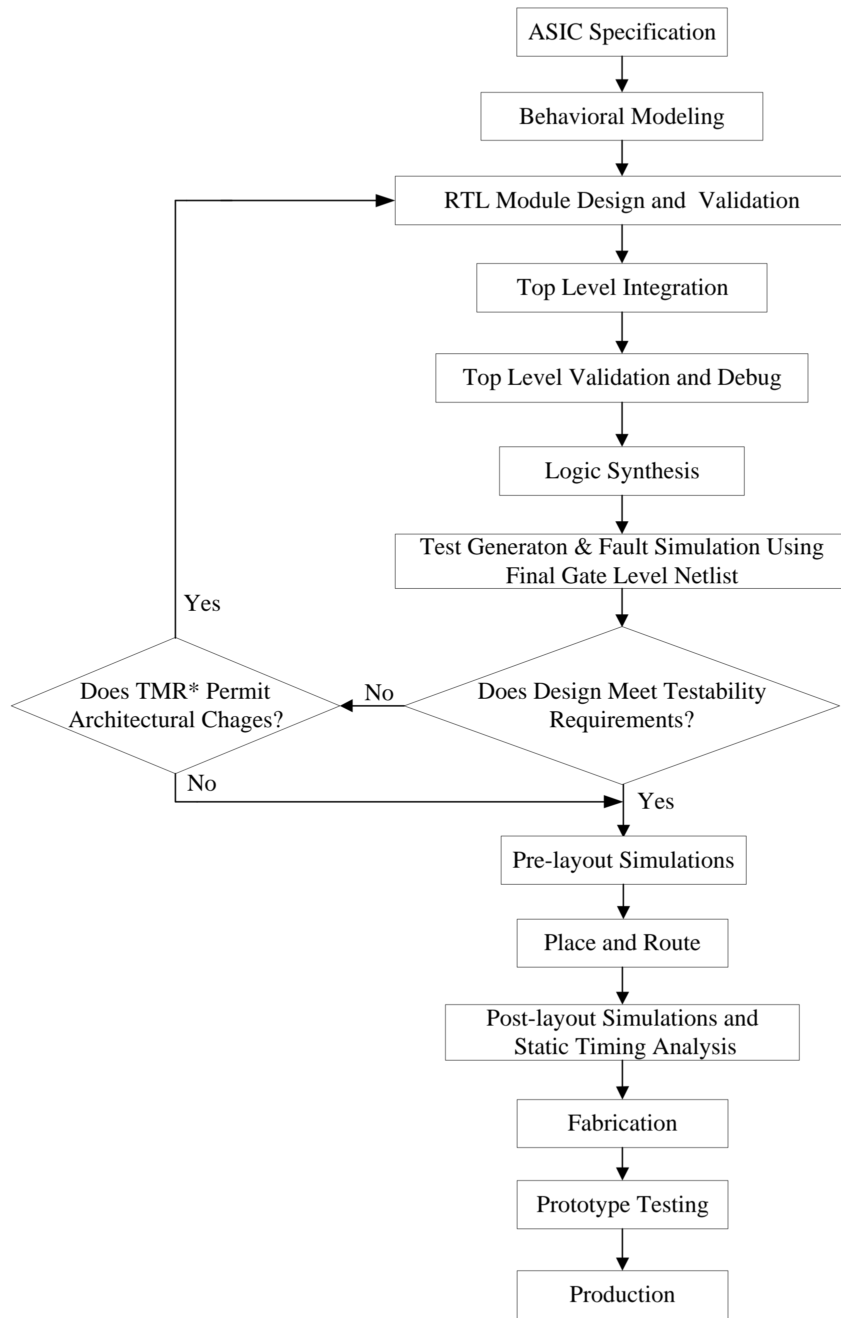


Figure 3 RTL-based Top-Down Design Methodology

In the RTL-based design methodology, test issues are explored only in the post-synthesis phase due to the lack of an effective RTL fault model. Existing fault simulation techniques, which are based on the gate-level SSF model, require a large memory and a lot of CPU time. These computing requirements grow at a rate that is proportional at least to the square of the number of gates in the design [GOE80]. For modern VLSI circuits of larger sizes, this is a

prohibitively expensive limitation. However, random sampling techniques proposed by Agrawal [AGR81] reduce such complexity. Even though the random sampling techniques reduce the size of the fault-list used for fault simulation, they assume the availability of a complete gate-level fault-list and therefore can not be used prior to logic synthesis. Post-synthesis findings of such fault grading and test generation efforts are too late in the design cycle to be used for improving in the design architecture to enhance testability features. Such problems seriously impact the business-related aspects of VLSI devices such as the time-to-market requirements (TMR) as shown in Figure 3. Clearly, there is a need for an RTL fault model to perform the test-related tasks earlier in the design cycle.

1.6 Goal of Thesis Research

The goal of this thesis research can be summarized by the following words: "To develop an RTL fault model that supports test pattern evaluation through RT level fault simulation prior to final logic synthesis. This RTL fault coverage should be an estimate of the post-synthesis gate-level fault coverage within predictable error bounds." The RT level fault simulation performed using the proposed RTL fault model is expected to provide a large performance gain over the gate-level approach, support test generation prior to logic synthesis, and offer early detection of testability problems, enabling architectural improvements for Design-for-Test (DFT) reasons during the pre-synthesis phase of the VLSI design cycle.

1.7 Overview of the Thesis

Modern VLSI systems consist of interconnections of modules described in an RTL language. At the present time, of the three main aspects of VLSI circuit realization, namely, design, validation and test, the first two are performed at the RT level while the test related issues are still addressed only in the post-synthesis phase at the gate-level due to the lack of an effective RTL fault model. A review of the state-of-the-art of research in RTL/behavioral/high-level fault modeling shows that there is no widely accepted solution. A novel procedure proposed in this thesis fills this void. The proposed procedure supports test pattern evaluation through RTL fault

simulation. The RTL fault coverage serves as an estimate of the gate-level fault coverage for a given set of test patterns. This procedure is based on new RTL fault model, fault-injection algorithm, the application of stratified sampling theory and stratum weight extraction techniques.

Some of the previous efforts in the RTL fault modeling area, as described by Hayne and Johnson [HAY99], were focused on developing a model that, when applied to RT level description, could produce the behavior of all possible gate-level single "stuck-at" faults. The RTL fault models that fall short of achieving this goal have been considered incomplete. The procedure presented in this thesis works on the premise that all hardware (gate-level) faults may not be represented at the RT level since the RT level description is a higher level of abstraction and may not contain the low-level structural information needed to exactly replicate all gate-level failures. Also, since the gate-level netlist changes drastically with every logic synthesis iteration, efforts to model all possible gate faults at RT level are inefficient. Instead, in this thesis, an RTL fault model and fault-injection algorithm are developed such that the RTL fault-list of a module becomes a representative sample of the corresponding collapsed gate-level fault-list. The proposed RTL fault model and the fault-injection algorithm are developed from an analysis of the properties of the gate-level SSF model and mapping of RTL constructs onto gate-level structures during logic synthesis. The proposed RTL faults of a module have a distribution of detection probabilities similar to that for collapsed gate faults of a corresponding gate-level netlist. The difference between RTL and gate-level fault coverages of a module for a given set of test patterns is expected to be within the error bounds established for the random-sampling technique by Agrawal and Kato [AGR90]. The effectiveness of the proposed RTL fault model is verified using several real-life industry-application VLSI circuits.

It is observed that the total number of RTL faults in a module does not represent the size of the gate-level fault-list. This lack of a clearly defined relationship between the number of RTL faults and the number of possible gate-level faults presents a problem for a VLSI system which consists of several modules. Although the RTL fault-list of each module in a VLSI system is a representative sample of the corresponding gate-level fault-list of that module, the overall RTL fault-list of the system does not constitute a representative sample of the overall gate-level fault-list. This observation led us to consider a technique known as *stratified sampling* [COC77]. According to the stratified sampling theory, a VLSI system is divided into several non-overlapping strata according to RTL module boundaries. The stratum weights for these modules

are determined using any of the proposed techniques described in Chapter 5. RTL fault coverages of modules are added according their respective stratum weights to determine the stratified RTL fault coverage for the VLSI system. The stratified RTL fault coverage serves as an estimate of the gate-level fault coverage of the VLSI system for the given set of test patterns. The error bounds for this estimate are statistically calculated. The stratified RTL fault coverages of several real-life industry-application VLSI systems are compared with the corresponding gate-level fault coverages for various test pattern sets. A variety of VLSI systems was selected for these experiments such that it reflected variations of RTL coding styles, complexity and types of functionality, e.g., data-path logic, state-machines, controllers, etc., that are generally encountered in the state-of-the-art ASIC design arena.

The RTL fault simulation method is analogous to the gate-level approach in which good and faulty circuits are created on the basis of the single stuck-at fault assumption and simulated with a given set of test patterns. When the responses of a good and a faulty circuit do not match, the fault is considered detected. The RTL fault coverage of a module is defined as the ratio of number of detected faults to the total number of simulated faults. The RTL fault simulator used in this thesis is *Verifault-XL™* [CAD97]. *Verifault-XL™* is suitable for use as an RTL fault simulator due to its capability of propagating fault effects through RTL circuit description [CAD97]. Any other fault simulator with similar capability can also be used with the proposed fault model. The RTL faults are identified to *Verifault-XL™* as zero-delay buffers inserted between variables and executable statements. A C++ parser (see Appendix B) was developed based on the proposed RTL fault model and the fault-injection algorithm. This parser judiciously places faults in the RTL code in the form of zero-delay buffers in order to identify them to the targeted fault simulator *Verifault-XL™*.

In this thesis, the Verilog HDL is used as a means to describe the proposed procedure. However, the fundamental concept of RTL fault modeling and the application of stratified sampling developed in this research can be applied to any other Hardware Description Language (HDL).

1.8 Organization of the Thesis

Chapter 1 provides background information in the area of VLSI testing. It contains a brief introduction to fault modeling, test generation and design-for-test. The RTL-based VLSI design methodology is summarized with the need for RTL fault model highlighted. The objective of research is briefly and clearly stated followed by an overview of the thesis.

Chapter 2 outlines previous research carried out in the area of RTL and behavioral fault modeling in recent years as well as in the past several decades. A comparison-based analysis is provided which concludes with the exposure of an acute need for an effective RTL fault model.

Chapters 3, 4 and 5 describe the research contribution of this thesis. Chapter 3 contains a detailed description of the proposed RTL fault model, the fault-injection algorithm and the RTL fault simulation methodology. The relationship of the proposed RTL faults and the traditional single stuck-at gate faults is elaborated using several examples. Chapter 4 summarizes the relevant concepts of stratified sampling theory and presents their application to RTL fault modeling. Chapter 5 outlines the stratum weight extraction techniques developed to support the application of stratified sampling theory to RTL fault modeling.

Chapter 6 describes the experimental work and results generated for several real-life industry-application VLSI systems. The impact of errors in stratum weight estimates on stratified RTL fault coverage and statistical error bounds is illustrated using an example. The issues related to the performance of RTL fault simulation are discussed.

Chapter 7 gives conclusions and describes issues that can be potentially addressed in future research.

A list of all references cited in the thesis follows Chapter 7.

Appendix A contains the secondary contribution of this thesis in the area of VLSI design validation. A new code coverage metric titled “Validation Vector Grade (VVG)” is proposed in that work.

Appendix B contains segments of C++ code developed on the basis of the fault-injection algorithm described in Chapter 3.

CHAPTER 2

REVIEW OF THE LITERATURE

VLSI circuits continue to increase in both size and complexity. Testing of these circuits is becoming more difficult and costly. The design and validation of integrated circuits are done at a higher level of abstraction than the gate-level, using hardware description languages. However, testing has been historically performed using traditional gate-level fault models [ABR90]. Existing gate-level test generation and evaluation techniques suffer prohibitively expensive performance penalties when applied to the modern VLSI circuits of larger sizes.

Sampling techniques, where a randomly selected subset of faults is simulated to estimate the fault coverage, can reduce the performance penalty of gate-level fault simulation. Case [CAS75] introduced the sampling technique to gate-level fault simulation to decide whether or not the fault coverage of a given test exceeds a given bound. This technique was elaborated by Agrawal [AGR81] to provide upper and lower bounds for the coverage. He also proposed a method that uses a fault sample of a fixed size. The estimation of fault coverage by simulating only a fraction of gate-level faults requires only a fraction of the time and resources required for the complete gate-level fault simulation. Similar approaches based on statistical sampling techniques are proposed by McNamer et al. [MCN89] and Daehn [DAE91]. Even though the fault-sampling technique reduces the size of the fault-list used for simulation, it requires a complete gate-level fault-list and, therefore, cannot be used prior to logic synthesis. Post-synthesis findings of test generation and fault simulation efforts are too late in the design cycle to be utilized for architectural changes to improve testability. It is, therefore, desirable to develop the fault models at a higher level of abstraction than the gate level.

Several high-level fault models and fault simulation techniques have been proposed by Thatte and Abraham [THA80], Chakraborty and Ghosh [CHA88], Ghosh [GHO88], Ward and Armstrong [WAR90], Armstrong et al. [ARM92], Cho and Armstrong [CHO94], and, Sanchez and Hidalgo [SAN96]. None of these techniques establish the relationship between high-level fault coverage and gate-level fault coverage. Khoche et al. [KHO92] proposed a behavioral fault model and a deductive fault simulator. Levendel and Menon [LEV82] extended D-algorithm to circuits described by hardware description languages using fault models such as function variables stuck at 0 or 1, control faults, etc. Chang et al. [CHA86] presented a structured

functional testing methodology and a new functional fault model based on binary decision diagrams. The proposed fault model had no connection with the low-level implementation. It required structural implementation to improve the quality of generated test patterns. The gate-level technique such as STAFAN by Jain and Agrawal [JAI85] offers a good alternative to traditional fault simulation approaches but still requires the knowledge of the structural implementation. Mao and Gulati [MAO96] proposed an RTL fault model and a simulation methodology but failed to establish the relationship of RTL faults to gate level faults. Their approach also required one to run fault simulation twice (first in an optimistic and then in a pessimistic mode) and to use the average of the results to reduce the difference between the RTL and the gate-level fault coverages. This is an inefficient solution derived purely empirically. The authors did not establish any theoretical basis to generalize the application of their fault model. Their experimental data indicated as much as a 10 % error between the actual gate-level fault coverage and the RTL fault coverage. Hayne and Johnson [HAY99] developed a fault model based on finding an abstraction of the industry standard single-stuck-line faults in the behavioral domain. This fault model was developed such that for every possible gate-level fault in the circuit there is a corresponding faulty RTL circuit. The gate-level netlist changes drastically with every synthesis run and there are numerous possible structural implementations for the RTL code. The modeling of all possible gate-level failure mechanisms at RT level is clearly inefficient and one can use only limited cases.

Several techniques have been proposed by Armstrong [ARM93], Bhatia and Jha [BHA94], Chen et al. [CHE94], Lee and Patel [LEE94], and Vishakantaiah et al. [VIS93] to generate test patterns using the high-level description of a design. Recent research attempts also focus at combining software testing based techniques at the high level with test enhancement techniques at the gate level [RUD98], exploiting high-level design information to generate a fully-automated, simulation-based Automatic Test Pattern Generation (ATPG) system [CHI99] and applying genetic-algorithms to high-level test generation [COR97]. Although this research seems to hold promise for test generation, several decades of research has not produced a well-established RTL fault model for test evaluation. As a consequence, test generation and evaluation are still performed during the post-synthesis phase of the design cycle of a top-down VLSI design methodology.

In the meanwhile, the focus of the research has shifted to developing an efficient high-level fault model for testability analysis and enhancement. New areas, namely, behavioral synthesis and High-Level Test Synthesis (HLTS) have emerged. As described by Dey et al. [DEY98], during behavioral synthesis, an algorithmic specification of the design's functionality is transformed into RTL specification as an interconnection of macro-blocks (e.g., functional units, registers, multiplexers, etc.,) and random logic. HLTS is defined as a process of design modification based on high-level information to improve testability. In HLTS, the controllability and observability of the design are first analyzed at the high-level to identify hard-to-test portions of the system. The behavioral description is then modified to improve testability.

Testability considerations at the behavioral level are complicated by the absence of a behavioral fault model with a close correlation to silicon defects. Therefore, recent focus has been on methods to include sequential Automatic Test Pattern Generator (ATPG) or Built-in Self-Test (BIST) objectives into the behavioral synthesis process.

The goal of ATPG-oriented approaches is to reduce the test generation complexity by increasing the transparency of modules, breaking loops and increasing the controllability/observability of registers. Hsu and Patel [HSU98] describe a high-level testability analysis technique that evaluates the testability of the design using controllability and observability measures. After identifying hard-to-test areas based on Control-Data Flow Graph (CDFG) constructed from behavioral description of a design, the proposed testability enhancement methods for data-path as well as controller logic are applied to improve testability. Lee et al. [LEE92] propose to improve the controllability and observability of data-path registers by assigning variables of the CDFG such that most registers are connected to primary I/Os. The sequential depth from an input register to an output register can also be minimized during register assignment. This improves the controllability and observability of all registers of the data path. Loops in the data-path contribute significantly to the difficulty of sequential ATPG. Approaches based on assigning boundary variables to a scan register in order to break the loops are described by Lee et al. [LEE93]. A simultaneous scheduling/assignment technique to avoid formation of the loop is proposed by Potkonjak et al. [POT95]. Chen et al. [CHE94] proposed adding test statements to behavioral description based upon the testability analysis to improve controllability and observability of all variables in the design. The modified behavioral description upon synthesis produces circuits with higher fault coverage and efficiency than those produced by the

original description. Ghosh et al. [GHO96] and Ravi et al. [RAV97] describe a controller resynthesis technique to enhance testability of register-transfer level controller/data-path circuits. Makris and Orailoglu [MAK99] propose an RTL analysis methodology that identifies the test justification and propagation bottlenecks, facilitating a judicious DFT insertion process. They describe a traversal algorithm that examines satisfiability of the test justification and propagation requirements at each module's boundary. The proposed methodology handles both combinational and sequential, data and control path modules in a uniform manner. It addresses problems related to the traversal process, such as feedback loops and reconvergent paths.

The goal of the BIST-based high-level test synthesis approach is to increase the probability of each module in a system to be fully tested by the BIST structure. Papachristou et al. [PAP98] proposed a technique for BIST insertion in the behavioral description of a design. Test generators and test response analyzers are embedded in the design. Testability is achieved by improving the controllability/obsevability of registers and by improving the randomness/transparency properties of modules. Drawback of this approach is that it requires additional test hardware.

While research results in the area of high-level synthesis show great promise, the proposed techniques are mostly applicable to data-flow intensive designs. More work is needed before high-level test synthesis can be used in the mainstream ASIC design arena. Most of the VLSI design work is still done at the RT level while high-level test synthesis aims at facilitating DFT for behavioral designs. Though high-level test synthesis holds great promise for futuristic behavioral level designs, the fundamental problem of the lack of an RTL fault model for test generation and evaluation needs to be solved for the contemporary mainstream RT level designs.

CHAPTER 3

RTL FAULT MODEL, FAULT-INJECTION ALGORITHM AND FAULT SIMULATION METHOD

In this chapter, the scope of application of the RTL fault model is outlined. This is followed by an elaborate description of the proposed fault model and the fault-injection algorithm. An RTL fault simulation method based on a commercial fault simulator is also described. The effectiveness of the proposed fault model is illustrated using several examples of RTL circuit descriptions. The RTL fault-list of each of these circuits is compared with the respective gate-level fault-lists of several post-synthesis structural implementations to study the relation of the RTL faults to the gate-level hardware faults. RTL fault coverages of several real-life industry circuits are compared with the respective gate-level fault coverages. In all circuits, the difference between the RTL and the gate-level fault coverages is found within acceptable and predictable error bounds.

3.1 RT Level Hardware Design Modeling

Hardware description languages (HDLs) are used to model VLSI circuits. As described by Sternheim et al. [STE93], HDL constructs are classified into three types: structural, register-transfer level (RTL) and behavioral. The structural, RTL and behavioral constructs of an HDL describe the design at the lowest, the intermediate and the highest levels of abstractions, respectively.

The structural features of HDLs are primarily used to describe the gate-level netlist during the post-synthesis phases of the design cycle. Boolean gates such as AND, OR, NOR, etc., constitute the structural constructs of the HDL.

RTL constructs represent a subset of HDL constructs with the corresponding design guidelines meant to ensure the consistent synthesis of gate-level netlists by logic synthesis tools. In an RTL model, the operations of a sequential circuit are described as synchronous transfers between functional units such as an arithmetic unit and a register file. With event scheduling and resource allocation information built-in, an RTL model represents the micro-architecture of a

circuit. Figure 4 and Figure 5 show RTL descriptions of a 16-bit counter and a 4-to-1 multiplexer, respectively.

The behavioral constructs of HDL are used for system modeling aimed at proof-of-concept, testbench generation to validate RTL designs and for design modeling to be used with high-level synthesis tools.

```
// Module Name: 16-bit counter

module counter (count, din, load, enable, clock, reset_);

// i/o Declrations

input[15:0] din;
input load, enable, clock, reset_;

output[15:0] count;


// Data Type Declarations

reg[15:0] count;

wire[15:0] din;
wire load, enable, clock, reset_;


// Functionality Implementation

always @(posedge clock or negedge reset_)
 if(!reset_)
 count[15:0] <= 16'h0000;
else if(load)
      count[15:0] <= din[15:0];
else if(enable)
      count[15:0] <= count[15:0] + 16'h0001;
else
      count[15:0] <= count[15:0];

endmodule
```

Figure 4 RTL Description of a 16-bit Counter

```
// Module Name: 4-to-1 Multiplexer

module multiplexer(e, a, b, c, d, s);


// i/o Decleartion

input[3:0] a, b, c, d;
input[1:0] s;

output[3:0] e;


// Data Type Declarations

reg[3:0] e;

wire[3:0] a, b, c, d;
wire[1:0] s;


// Functionality Implementation

always @(a or b or c or d or s)
 case (s[1:0])
 2'b00: e = a;
 2'b01: e = b;
 2'b10: e = c;
 2'b11: e = d;
endcase

endmodule
```

Figure 5 RTL Description of a 4-to-1 Multiplexer

Some of the previous research in the area of high-level fault modeling scoped its application to behavioral design modeling [GHO88, SAN96], while the other research aimed

towards RTL design modeling [MAO96, HAY99]. The research presented in this thesis focuses its application on RTL design modeling.

The proposed RTL fault model and test evaluation technique can be used as an integral part of both, top-down as well as high-level design methodologies. The top-down design methodology requires design to be specified using RT level HDL constructs, which is then converted into a gate-level implementation by a logic synthesis tool using cost constraints such as speed and area. In the high-level design methodology, design is described with behavioral HDL constructs. The behavioral description is converted into register-transfer level specification by a high-level (or behavioral) synthesis tool using cost constraints such as area, performance and testability. The obtained RTL specification of the design is then converted into a gate-level implementation by logic synthesis.

In this thesis, the Verilog HDL is used as a medium to explain the proposed RTL fault model. However, the concepts developed and described here can be applied to any other hardware description language. Table 1 lists RTL constructs of the Verilog HDL [STE93].

It is essential to establish terminology before describing the fault model. A few clarifications on the terminology used in this thesis are offered here:

Language Operators: Language operators described in Table 1 are classified into Boolean (&, |, ^, ~), synthetic (+, -, *, >=, <=, <, >, = =, !=), and logical (&&, ||, !) operators. A further classification of these operators, though used in other contexts, is unnecessary for the purpose of RTL fault model description.

Design Hierarchy: A VLSI system consists of several interconnected components. Each component may contain design description using RTL constructs described in Table 1 or interconnected sub-components (see Figure 6). At any level of hierarchy, when a design is described using RTL constructs, it is referred to as a module. As a good contemporary practice, the RTL description of a module should not exceed some reasonable length of program such that the synthesized gate-level netlist may not exceed 30,000 gates. However, the RTL fault model proposed here is universal in application to design descriptions of any program length and any gate count after logic synthesis.

Table 1 Verilog RTL Constructs

RTL Construct Type	RTL Construct
Data Type	reg
	wire
Arrays	one-dimensional array
Basic Operators	+ - *
	> >= < <=
	! && ||
	== !=
	?:
	{ }
	~ & | ^
	<< >>
Procedural Statements	always
	assign
Assignment Operators	blocking (=)
	non-blocking (<=)
Conditional Statements	if-else
	case
	casex
	casez
Event Expression	@
String Substitution	`define
Parameterization	parameter
Event Control	posedge
	negedge
	level sensitive

Identifiers: Identifiers are the names that one gives to the objects like wires, gates and functions in the circuit. All identifiers that specify signal names will be referred to as "variables" in this thesis.

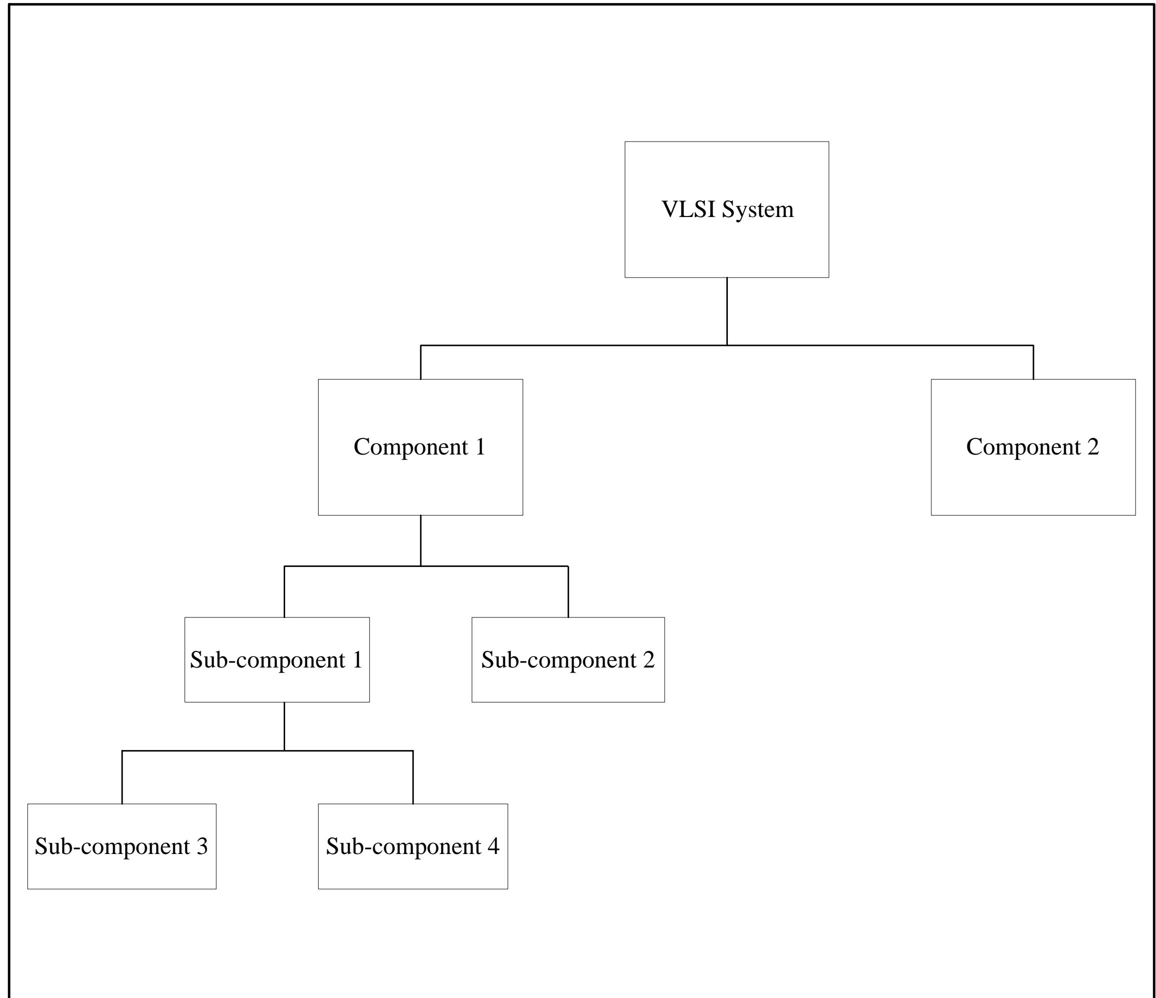


Figure 6 Design Hierarchy

Output Variable: The term "output variable" is commonly used in two separate situations. A variable declared as an "output" of a module in an I/O declaration is considered an output variable. In the case of an RTL construct in a module, a variable that appears on the left-hand side of an assignment (variable that is being assigned in a given construct) is considered the output variable of that construct.

3.2 RTL Fault Model and Fault-injection Algorithm

As described by Hayne and Johnson [HAY99], previous research efforts in the RTL fault modeling arena have taken the approach of modifying RTL code to model all gate-level failure mechanisms. These efforts have not been successful, primarily due to the fact that the gate-level netlist changes drastically with every synthesis iteration, creating many distinct gate-level fault-lists. It is impossible to model all the gate faults of every possible netlist at the RT level. Instead, in this thesis, an RTL fault model and a fault-injection algorithm are developed such that the RTL fault-list of a module becomes a representative sample of the collapsed gate-level fault-list. The proposed RTL fault model and fault-injection algorithm are derived from an analysis of the properties of the gate-level SSF model and mapping of the RTL constructs onto gate-level structures during logic synthesis. The classical definition of the term "representative sample" in the context of statistical theory is given by Stephan and McCarthy [STE58] as:

"A representative sample is a sample which, for a specified set of variables, resembles the population from which it is drawn to the extent that certain specified analyses that are to be carried out on the sample (computation of means, standard deviations, etc., for particular variables) will yield results which will fall within acceptable limits set about the corresponding population values, except that in a small proportion of such analyses of samples (as specified in the procedure used to obtain this one) the results will fall outside the limits."

In order for the RTL fault-list of a module to be a representative sample of the collapsed gate-level fault-list, RTL faults should have a distribution of detection probabilities (see Figure 7) similar to that for collapsed gate faults. The detection probability of a fault is defined by Seth et al. [SET90] as the "probability of detecting a fault by a randomly selected pattern." In other words, if a given test set contains n patterns and a fault is detected k times during fault simulation using this pattern set, the detection probability of the fault is given as k/n. When two fault-lists (an RTL and collapsed gate-level) with similar detection probability distributions are simulated for a given set of test patterns, the respective fault coverages are expected to track each other closely within statistical error bounds. Agrawal and Kato [AGR90] established error bounds for the random fault-sampling technique in which detection probability distributions for a random

sample and that for the entire gate fault-population are expected to be similar. If the RTL fault-list of a module is indeed a representative sample of the collapsed gate-level fault-list, the difference between RTL and gate-level fault coverages of a module for a given set of test patterns should be within the error bounds established for the random sampling technique by Agrawal and Kato [AGR90]. This assumption is supported by the data given in a later section.

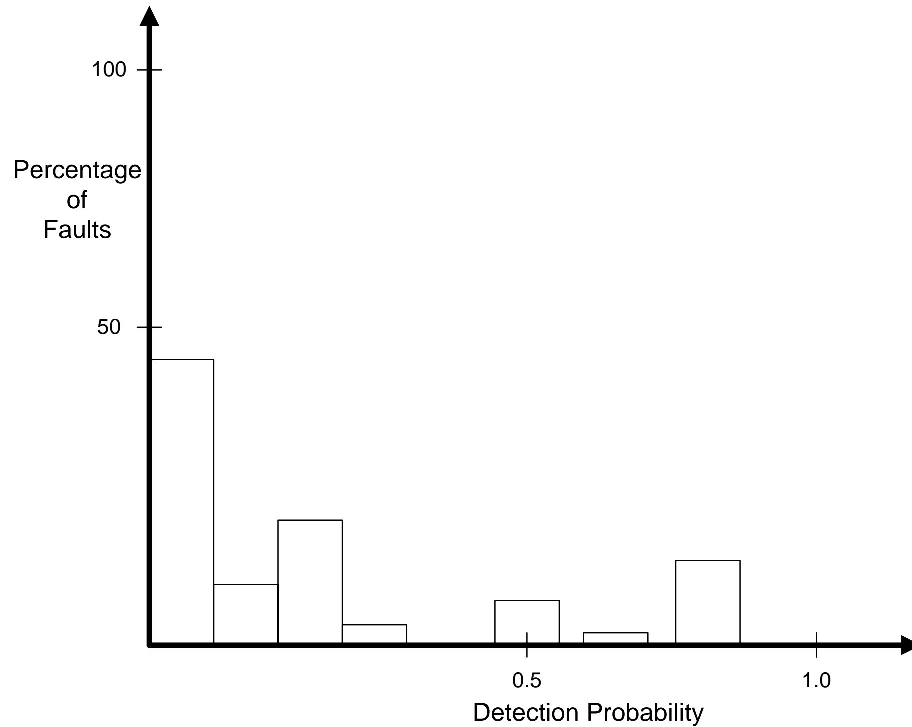


Figure 7 Example of Fault Detection Probability Distribution

RT level design description dictates the micro-architecture of the gate-level representation. During logic synthesis, RTL operators map onto Boolean components of varying complexities, e.g., Boolean and logical RTL operators map onto Boolean gates, synthetic operators map onto components such as adders, comparators, etc. The RTL variables map onto signal lines in the gate-level netlist although the relationship may not be a one-to-one mapping. The goal of the proposed fault model is to judiciously place RTL faults in the design description of a module. This is assured by mirroring properties of the gate-level SSF model in the RTL fault model. These properties are,

Properties of the Gate Level SSF Model:

(1) Boolean components are assumed to be fault-free.

(2) Signal lines contain faults:

- a stuck-at-zero (s-a-0) fault when the logic level is fixed at value 0
- a stuck-at-one (s-a-1) fault when the logic level is fixed at value 1

(3) According to the single fault (SSF) assumption, only one fault is applied at a time when a test set is either being created or evaluated.

(4) The fault-list is collapsed using the "check-point" theorem [ABR90]. The collapsed fault-list of a module contains input as well as fan-out faults. Further collapsing may be done using structural equivalence and dominance relationships.

Properties of the RTL Fault Model:

(1) Language operators (which map onto Boolean components in the gate-level netlist) are assumed to be fault-free.

(2) Variables (which map onto signal lines in the gate-level netlist) contain faults:

- a stuck-at-zero (s-a-0) fault when the logic level is fixed at value 0
- a stuck-at-one (s-a-1) fault when the logic level is fixed at value 1

(3) The proposed RTL fault model follows the single fault assumption and therefore only one fault is applied at a time when a test set is evaluated.

(4) The RTL fault-list of a module contains input as well as fan-out faults. RTL variables that are used more than once in executable statements or the instantiations of lower level modules of the design-hierarchy are considered to have fan-out. Input faults of a module at the RT level have one-to-one equivalence to input faults of the module at the gate-level. The fan-out faults of a module at the RT level represent a subset of the fan-out faults of a possible gate-level implementation.

The definition of the RTL fault model and the fault-injection algorithm encompasses modeling of faults for synthetic, Boolean and logical operators, sequential elements and fan-out/stem variables, as well as the collapsing of RTL faults.

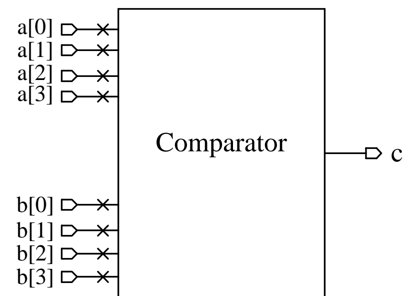
3.2.1 Fault Modeling for Synthetic Operators

When RTL constructs contain synthetic operators, faults are injected only on the input variables of such operators. During logic synthesis, the synthetic operators are replaced by combinational circuits implementing the respective functions, e.g., adder, subtracter, comparator, etc. The internal details of such functions represented by synthetic operators are not available at the RT level and therefore only the subset of the checkpoint faults of the gate-level representation of these operators, namely, the primary input faults are modeled.

Figure 8(a) shows the description of a 4-bit comparator using a synthetic operator. Figure 8(b) shows RTL faults placed in the code using a symbolic description. A cross ("x") indicates the RTL fault location. The fault on the output port "c" is intentionally omitted. Criteria for fault placement on output ports will be discussed later.

```
module comp(c, a, b);

input[3:0] a, b;
output c;

reg c;

always @(a or b)
 if (a = = b)
    c = 1'b1;
 else
    c = 1'b0;

endmodule
```

(a) RTL Description



(b) RTL Faults in a Symbolic Description

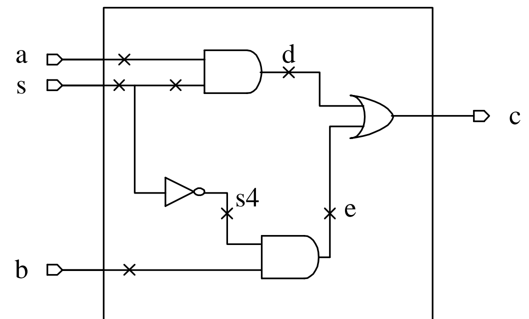
Figure 8 Example of Fault Modeling for Synthetic Operators

3.2.2 Fault Modeling for Boolean Operators

When a function is represented using RTL constructs that contain Boolean operators, faults are injected on variables that form Boolean equations. Some internal signals of these constructs are available at the RT level and therefore RTL faults are placed at primary inputs and internal nodes including signal stems and fan-outs. The post-synthesis gate-level representation of such a construct may be structurally different from the RTL Boolean representation. However, some RTL faults have equivalent faults in a collapsed gate-level fault-list of any post-synthesis design. Figure 9(a) shows the RTL description of a 2-to-1 multiplexer using Boolean operators. Figure 9(b) shows RTL faults placed symbolically in the code by making fault-specific addition to the code. Figure 9(b) shows a logical representation of the RTL code.

```
module mux(c, a, b, s);

input a, b, s;
output c;

wire a, b, c, s, d, e, s4;

assign d = a & s;
assign e = s4 & b;
assign s4 = !s;
assign c = d | e;

endmodule
```

(a) RTL Description



(b) RTL Faults in a Symbolic Description

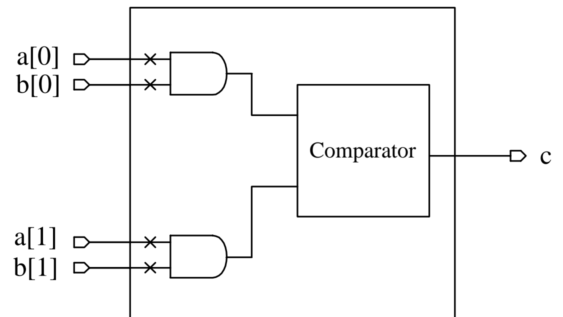
Figure 9 Example of Fault Modeling for Boolean Operators

3.2.3 Fault Modeling for Logical Operators

When RTL constructs contain logical operators, faults are injected on variables that constitute inputs of such operators. Sometimes the post-synthesis gate-level implementation of a function described using logical operators maintains the structure implied in the RTL description. In such cases, RTL faults have a one-to-one equivalence to the collapsed gate faults of the synthesized logic. Figure 10(a) shows the RTL description of a function using logical operators. Figure 10(b) shows RTL faults placed in the code using a symbolic representation of the RTL code.

```
module logic_func(c, a, b);

input[1:0] a, b;
output c;

reg c;

always @(a or b)
begin
 if((a[1:0] && b[1:0]) = = 2'b01)
  c = 1'b1;
 else
  c = 1'b0;
end

endmodule
```

(a) RTL Description



(b) RTL Faults in a Symbolic Description

Figure 10 Example of Fault Modeling for Logical Operators

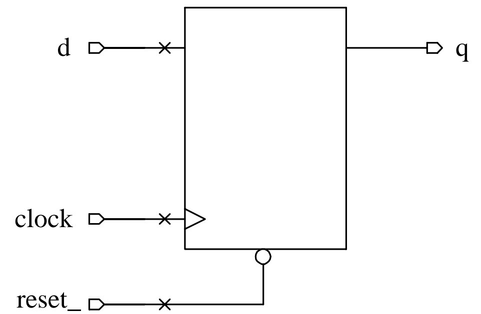
3.2.4 Fault Modeling for Sequential Elements

Hardware description languages support both types of sequential elements, latch as well as flip-flop. In both cases, RTL faults are placed on input ports of these components. In the case

of the flip-flop, faults are placed on clock as well as the reset variables. Figure 11(a) shows the RTL description of a flip-flop. Figure 11(b) shows the RTL fault location using a symbolic representation of the RTL code.

```
module flip_flop(q, d, clock, reset_);

input d, clock, reset_;
output q;

reg q;

always @(posedge clock or negedge reset_)
begin
  if (!reset_)
   q <= 1'b0;
  else
   q <= d;
end

endmodule
```

(a) RTL Description



(b) RTL Faults in a Symbolic Description

Figure 11 Example of Fault Modeling for Sequential Elements

3.2.5 Stem and Fan-out Fault Modeling

In the gate-level SSF model, stem and fan-out faults are unique since neither equivalence nor dominance relations exist between them. Stem and fan-out faults are treated as special cases in the RTL fault model as well. RTL variables that are used more than once in executable statements or instantiations of lower level modules of the design-hierarchy are considered to have fan-out. In these cases, a separate RTL fault is injected on each fan-out of each bit of the variable so that a fan-out fault only affects one executable statement using that fan-out. If the same statement uses the same variable more than once, then each instance of every bit of that variable will receive a unique RTL fault. Stem faults are treated with special attention as well. The fault-injection algorithm evaluates fan-out from each stem in the RTL code to determine fault-

injection not only on the fan-out but also on the stem. The various scenarios for stem faults are as follows:

1) RTL constructs in a given module are interconnected. An output variable from one RTL construct may be used as an input to one or more RTL constructs. In such a case, a unique RTL fault is placed on the stem in addition to faults on fan-out branches. Figure 12(a) shows this scenario. The dotted lines indicate boundaries of RTL constructs inside a given module.
2) The output variable of an RTL construct may be used as input to more than one RTL construct, each having multiple fan-outs within. In such a case, a unique RTL fault is placed on the stem which is the output variable of the RTL construct. Figure 12(b) shows this scenario.
3) An RTL variable in one of the RTL constructs may be used multiple times within that construct itself. Also, in some cases, this variable may fan-out to other RTL construct. A unique RTL fault is placed on the stem. Figure 12(c) shows this scenario.
4) The duplication of an RTL fault is avoided at all times. The output variable of one RTL construct with no fan-out within the construct may be used in another construct with a single fan-out. In such a case, only one RTL fault is placed on the line. This fault is placed inside the destination construct as a fault on input variable. Figure 12(d) shows this scenario.

Many other possible scenarios are similarly resolved by placing a unique RTL fault on each stem and all branches of the fan-out. In a VLSI system, several modules are interconnected. The interconnecting signals between modules have similar issues about stem and fan-out fault modeling. The properties described for stem and fan-out fault modeling for RTL constructs are applied for interconnecting signals between modules as well.

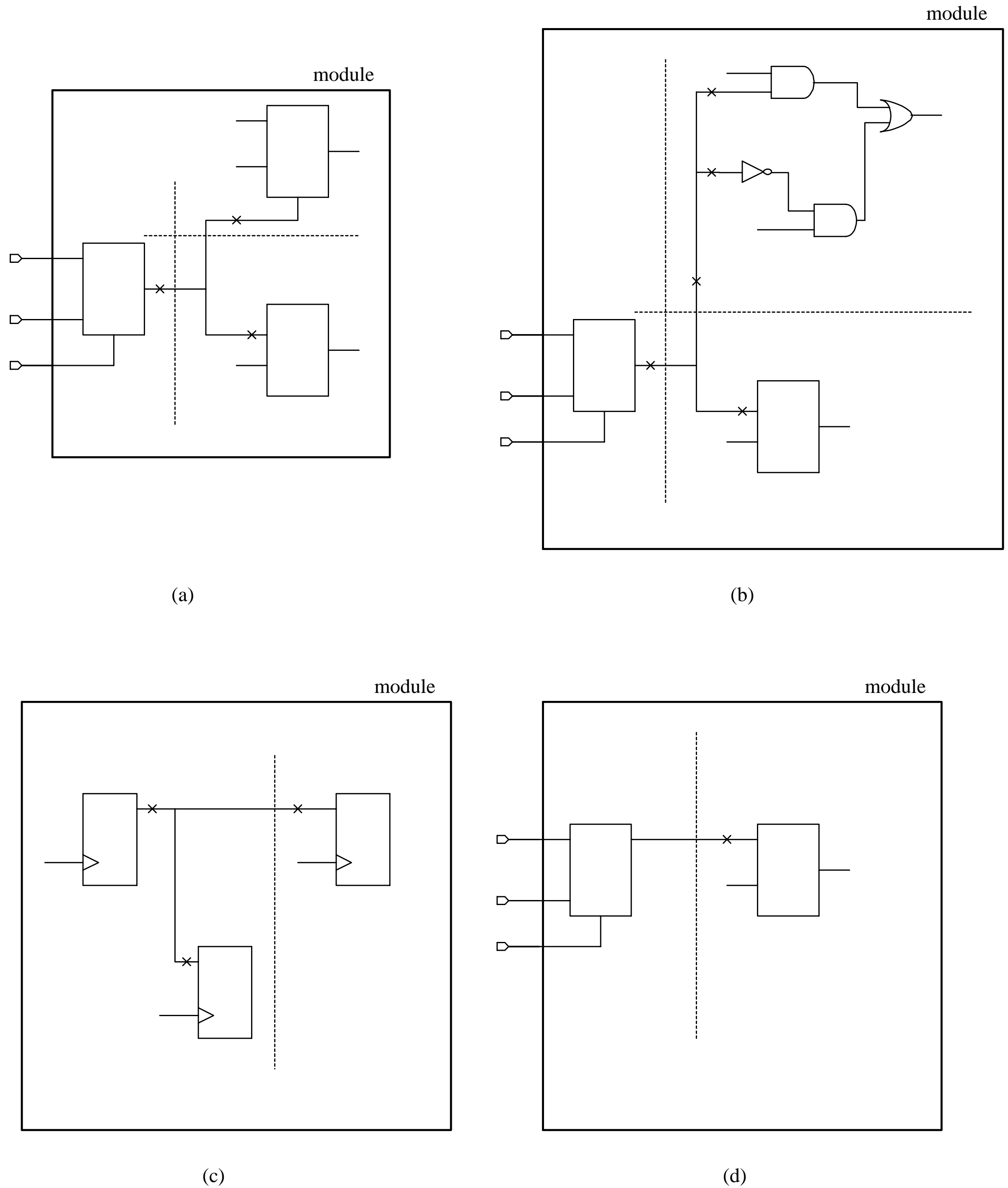


Figure 12 RTL Modeling of Stem and Fan-out Faults

3.2.6 RTL Fault Collapsing

The fault collapsing technique is widely used during gate-level fault simulation to reduce the size of the fault-list [ABR90]. Smaller fault-lists require lesser resources, and improve run-time performance. Although it is desirable, fault collapsing is not performed at the RT level since the structural information needed to analyze the equivalence of faults is missing. At the minimum, the proposed RTL fault model inherently avoids generating duplicate faults.

3.3 RTL Fault Simulation Method

The RTL fault simulator accepts the fault-injected RTL circuit description and the test pattern set as inputs. The RTL fault simulation method is analogous to the gate level approach, in which good and faulty circuits are created based on the single stuck-at fault assumption and simulated with a given set of test patterns. When the responses of a good circuit and a faulty circuit do not match, the fault is considered detected. Fault simulation is continued until all faults are evaluated for the given set of test patterns. At the completion of fault simulation, a fault report is generated which contains statistics and other information on detected as well as undetected RTL faults. The RTL fault coverage of a module is defined as the ratio of the number of detected RTL faults to all RTL faults.

The RTL fault simulator used in this thesis is *Verifault-XL*™. *Verifault-XL*™ is suitable for use as an RTL fault simulator due to its capability of propagating fault effects through RTL circuit description. It accepts the fault-injected RTL code as input along with the test pattern set. RTL faults are identified to *Verifault-XL*™ as zero-delay buffers inserted between variables and executable statements as per the fault-injection algorithm described in Section 3.2. For more details on the method of running RTL fault grading using *Verifault-XL*™, one may refer to Mao and Gulati [MAO96] and *Verifault-XL*™ *User's Guide* [CAD97].

A C++ parser (see Appendix B) was developed based on the RTL fault model and the fault-injection algorithm described in Section 3.2. Parser is a program that accepts the RTL code as input and places faults at appropriate locations without altering circuit behavior. The output of the parser is the modified RTL code. RTL faults are injected in the circuit description using zero-delay buffers in order to identify them to the targeted fault simulator, *Verifault-XL*™. Figure 17

describes an example of a fault-injected RTL code generated by the C++ parser. The original RTL code input to the C++ parser is given in Figure 15.

3.4 Study of Efficacy of RTL Fault Model and Fault-injection Algorithm

In this section, the effectiveness of the RTL fault model and fault-injection algorithm (proposed in Section 3.2) in modeling hardware faults at the RT level is discussed.

3.4.1 Comparison of RTL and Gate Fault Lists

In this sub-section, the relationship of the RTL faults with possible hardware gate faults is elaborated using two examples. In these examples, RTL fault-lists are created for the given RTL design descriptions using the proposed fault model and fault-injection algorithm. RTL design descriptions are then synthesized using a commercial logic synthesis tool, *Design Compiler*™ [BHA99], and a 0.35 micron CMOS library. For each RTL design description, two distinct gate-level implementations are obtained using the speed and area cost constraints, respectively. Gate-level collapsed fault-lists are generated from those implementations. The RTL and gate-level faults in the two lists are compared for possible relationships.

Example 1

Figure 13 (a) contains the RTL description of a 2-to-1 multiplexer. Constraint-driven logic synthesis may produce many different gate-level implementations from this RTL description. Two such implementations, I and II, are shown in Figure 13(b) and Figure 13(c). When the proposed RTL fault model and injection algorithm are applied to the code, the RTL faults are generated as given in Table 2 and Table 3 along with the collapsed gate-level fault-lists of implementations I and II.

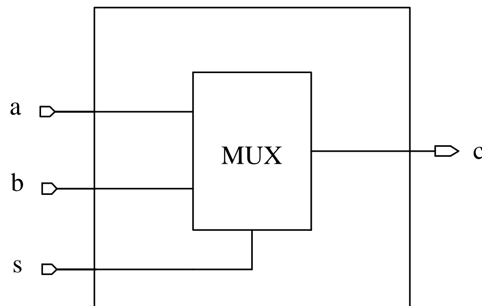
An analysis of the RTL and gate-level fault-lists reveals that individual RTL faults, when applied one at a time to the RTL design, produce behaviors that match the corresponding behaviors of faulty gate-level circuits resulting from individual stuck-at gate faults applied one at a time. Such RTL and gate faults are considered equivalent. Upon comparing the RTL fault-list

to the collapsed gate-level fault-list of implementation I (see Table 2), it is found that each RTL fault matches a unique gate-level fault. In fact, RTL and gate-level fault-lists are equivalent. Therefore, equivalence mapping between the RTL fault-list and the gate-level fault-list is one-to-one. A set of test patterns that produces 100% RTL fault coverage when applied to gate-level implementation-I, will also produce 100% coverage.

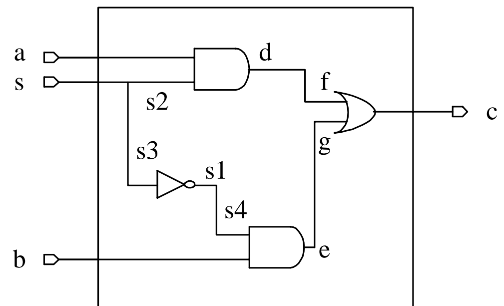
Upon comparing the RTL fault-list to the collapsed gate-level fault-list of implementation II (see Table 3), it becomes clear that all RTL faults have unique equivalent faults in the collapsed gate-level fault-list. However, two of the gate-level faults do not have equivalent RTL faults. In this case, the RTL fault-list is viewed as a subset of the gate-level fault-list.

```
module mux(c, a, b, s);

input a, b, s;
output c;

reg c;

always @ (a or b or  s)

if (s)
 c = a;
else
 c = b;

endmodule
```

(a) RTL Description



(b) Gate Level Implementation I



(c) Gate Level Implementation II

Figure 13 RTL and Gate-level Implementations of 2-to-1 Mux (Example 1)

Table 2 RTL-Gate Fault Equivalence for Implementation I (Example 1)

RTL Fault	Gate Fault
a s@0	a s@0
b s@0	b s@0
s s@0	s s@0
c s@0	c s@0
a s@1	a s@1
b s@1	b s@1
s s@1	s s@1
c s@1	c s@1

Table 3 RTL-Gate Fault Equivalence for Implementation II (Example 1)

RTL Fault	Gate Fault
a s@0	a s@0
b s@0	b s@0
s s@0	s s@0
c s@0	c s@0
a s@1	a s@1
b s@1	b s@1
s s@1	s s@1
c s@1	c s@1
No equivalent fault	s2 @1
No equivalent fault	s3 s@1

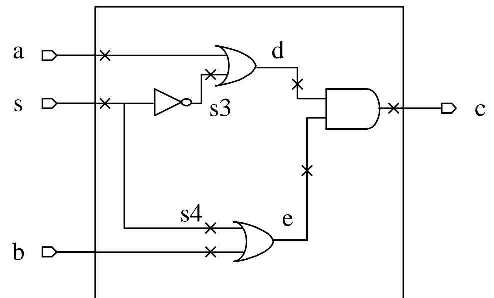
Example 2

Figure 14(a) contains the RTL description of a 2-to-1 multiplexer with Boolean operators. Figure 14(b) shows a symbolic representation of RTL code with injected faults indicated by crosses (“x”). Two of the possible post-synthesis structural implementations are presented in Figure 14(c) and Figure 14(d).

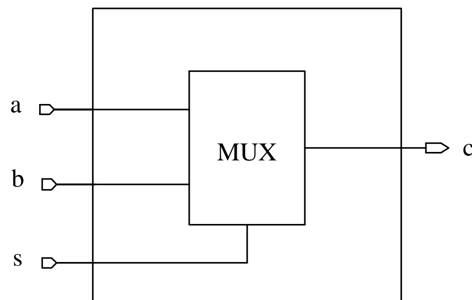
Comparison of the RTL fault-list to the collapsed gate-level fault-list of implementation-I (see Table 4) reveals a one-to-one equivalence between most of the RTL faults and all of the gate-level faults. A comparative study of the RTL fault-list and the collapsed gate-level fault-list of implementation II (see Table 5) indicates a one-to-one equivalence between most of the RTL and gate faults. Two RTL and two gate-level faults do not have equivalent faults on the other side.

```
module mux(c, a, b, s);

input a, b, s;
output c;

wire a, b, s, s3, s4, e, d, c;

assign e = b | s;
assign s3 = !s;
assign d = a | s3;
assign c = e & d;

endmodule
```

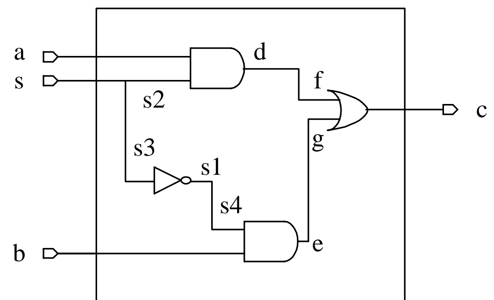
(a) RTL Description



(b) RTL Faults in a Symbolic Representation



(c) Gate Level Implementation I



(d) Gate Level Implementation II

Figure 14 RTL and Gate-level Implementations of 2-to-1 Mux (Example 2)

Table 4 RTL-Gate Fault Equivalence for Implementation I (Example 2)

RTL Fault	Gate Fault
a s@0	a s@0
b s@0	b s@0
s s@0	s s@0
c s@0	c s@0
a s@1	a s@1
b s@1	b s@1
s s@1	s s@1
c s@1	c s@1
s4 s@0	No equivalent fault
s3 s@0	No equivalent fault

Table 5 RTL-Gate Fault Equivalence for Implementation II (Example 2)

RTL Fault	Gate Fault
a s@0	a s@0
b s@0	b s@0
s s@0	s s@0
c s@0	c s@0
a s@1	a s@1
b s@1	b s@1
s s@1	s s@1
c s@1	c s@1
No equivalent fault	s2 s@1
No equivalent fault	s3 s@0
s4 s@0	No equivalent fault
s3 s@0	No equivalent fault

The two preceding examples illustrate the relationship between the RTL faults and the hardware gate faults. In both cases, the RTL modules describe a very simple hardware function. These simple examples of circuit-descriptions were used for better illustration. In practice, the RTL modules are far more complex and may contain more than hundred lines of code and possibly result in several hundred or thousand gates after logic synthesis. It may appear from the illustrated examples that the proposed RTL fault model and fault-injection algorithm consider only pin faults. However, this notion is not correct. In practice, an RTL module contains several interconnected Boolean components described using various constructs. RTL faults are judiciously placed at input ports of the module, in the input variables of the Boolean components, and on the fan-outs of interconnecting signals between Boolean components. The RTL faults may also be placed on fan-outs of variables internal to the Boolean components if they are described using Boolean RTL operators. Therefore, an RTL fault-list of a module contains not just pin faults but also various internal faults. This is illustrated by the examples provided in Figure 15 and Figure 16. A hardware function is described at the RT level abstraction in Figure 15. Figure 16 contains a symbolic representation of this RTL code. The RTL fault model and fault-injection algorithm proposed in Section 3.2, when applied to the code, judiciously place faults in the RTL code. RTL faults are depicted with crosses ("x") in Figure 16. As can be observed in Figure 16, the RTL fault set consists of pin faults of the module as well as internal faults. The logic synthesis of the RTL code given in Figure 15 will produce a gate-level netlist. The gate-level netlist derives the micro-architecture from the RTL description. An analysis similar to the one presented in Examples 1 and 2, reveals that most RTL faults in Figure 16 have equivalent faults in the collapsed gate-level fault-list, making the RTL fault-list a representative subset of the collapsed gate-level fault-list.

```
// Module Name: Miscellaneous Function

module misc_func(out_sig1, out_sig2, a, b, c, d, clk, reset_);

// i/o declarations

input[1:0] a, b, c, d;
input clk, reset_;

output out_sig1, out_sig2;

// Data Type Declarations

wire[3:0] e;
wire[2:0] f;
wire[2:0] g;
wire[1:0] a, b, c, d;
wire v, w, k;

reg out_sig1, out_sig2;
reg h, i, j;

// Functionality Implementation

assign e[3:0] = a[1:0] * c[1:0];
assign f[2:0] = a[1:0] + b[1:0];
assign g[2:0] = c[1:0] - d[1:0];


always @(e or f)
 if(e < f)
   h = 1'b1;
 else
   h = 1'b0;

always @(f or g)
 if(f == g)
  i = 1'b1;
 else
  i = 1'b0;

always @(e or g)
 if(e > g)
  j = 1'b1;
 else
  j = 1'b0;

always @(posedge clk or negedge reset_)
 if(!reset_)
  out_sig1 <= 1'b0;
 else
  out_sig1 <= (i & j) ? h : out_sig1;

assign v = i & h;
assign w = (!h) & j;
assign k = v | w;

always @(posedge clk or negedge reset_)
 if(!reset_)
  out_sig2 <= 1'b0;
 else
  out_sig2 <= k;

endmodule
```

Figure 15 RTL Code Example

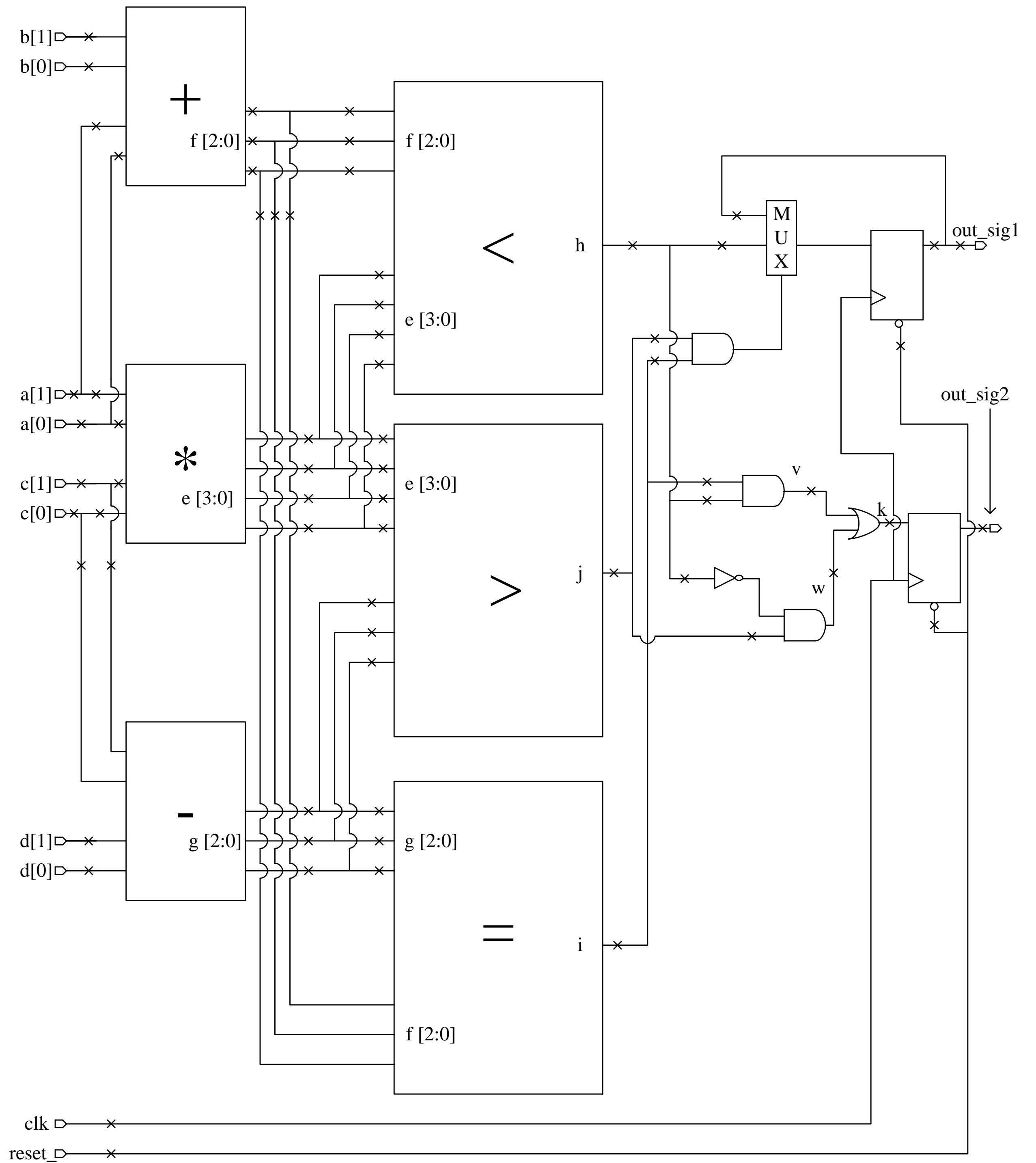


Figure 16 RTL Faults in a Symbolic Representation

```
// Module Name: Miscellaneous Function

module misc_func(out_sig1, out_sig2, a, b, c, d, clk, reset_);

// i/o declarations

input[1:0] a, b, c, d;
input clk, reset_;

output out_sig1, out_sig2;

// Data Type Declarations: New Data Type Declarations for some existing variables and other
new variables to
// be described here.

// Functionality Implementation

assign e[3:0] = a_fault2[1:0] * c_fault2[1:0];
assign f[2:0] = a_fault3[1:0] + b_fault1[1:0];
assign g[2:0] = c_fault3[1:0] - d_fault1[1:0];


always @(e_fault2 or f_fault2)
 if(e_fault2 < f_fault2)
   h = 1'b1;
 else
   h = 1'b0;

always @(f_fault3 or g_fault2)
 if(f_fault3 == g_fault2)
  i = 1'b1;
 else
  i = 1'b0;

always @(e_fault3 or g_fault3)
 if(e_fault3 > g_fault3)
  j = 1'b1;
 else
  j = 1'b0;

always @(posedge clk_fault1 or negedge reset_fault2)
 if(!reset_fault2)
  out_sig1_fault1 <= 1'b0;
 else
  out_sig1_fault1 <= (i_fault2 & j_fault2) ? h_fault2 : out_sig1_fault2;

assign v = i_fault3 & h_fault3;
assign w = (!h_fault4) & j_fault3;
assign k = v_fault1 | w_fault1;

always @(posedge clk_fault1 or negedge reset_fault3)
 if(!reset_fault3)
  out_sig2_fault1 <= 1'b0;
 else
  out_sig2_fault1 <= k_fault1;

// Generic "Buffer" primitives placed to indicate presence of  RTL faults to Verifault-XL
buf a_fault1_b0(a_fault1[0], a[0]);
buf a_fault1_b1(a_fault1[1], a[1]);

buf a_fault2_b0(a_fault2[0], a_fault1[0]);
buf a_fault2_b1(a_fault2[1], a_fault1[1]);

buf a_fault3_b0(a_fault3[0], a_fault1[0]);
buf a_fault3_b1(a_fault3[1], a_fault1[1]);
.
.
.
buf k_fault1_b0(k_fault1, k);

endmodule
```

Figure 17 Fault-injected RTL Code

3.4.2 Comparison of RTL and Gate Fault Coverages

As illustrated in Section 3.4.1, the proposed RTL fault model and fault-injection algorithm are developed such that the RTL fault-list becomes a representative sample of the collapsed gate-level fault-list. As described earlier, in order for the RTL fault-list to be a representative sample, the detection probability distributions of RTL and gate-level fault-lists must be similar. In such a case, RTL and gate fault coverages for a given set of test patterns are expected to track each other closely within statistical error bounds. Agrawal and Kato [AGR90] established error bounds for the random fault sampling technique in which the detection probability distributions of the random samples and that of the entire gate fault-population are expected to be similar. Since the proposed RTL fault model is expected to resemble statistical properties of the random sample, the difference between the RTL and the gate-level fault coverages of a module for a given set of test patterns is expected to be within the error bounds established for the random-sampling technique. Agrawal and Kato [AGR90] established the range of coverage for the random sampling technique as,

$$c \pm \frac{\alpha^2 k}{2N}\sqrt{1+4Nc(1-c)/(\alpha^2 k)} \tag{3.1}$$

where, $k = 1 - N/M$, when N gate faults are sampled from a total of M gate faults in the circuit, $\alpha = 3.00$ obtained from the Gaussian probability distribution function for a confidence probability of 0.998, and c is the ratio of detected to total among sampled gate faults.

When equation (3.1) is used for error bounds of the gate-level fault coverage estimated by the proposed RTL fault modeling technique, N represents the number of RTL faults in a module, M represents the number of gate faults in a module and c represents the ratio of the number of detected RTL faults to all RTL faults.

In this section, the RTL and the gate fault coverages of several real-life industry-application circuits are compared. An RTL fault-list is created for each RTL module using the proposed fault model and fault-injection algorithm. RTL modules are then synthesized using the commercial logic synthesis tool *Design Compiler*™ [BHA99] and a 0.35 micron CMOS technology library. A gate-level implementation is arbitrarily selected to measure the gate-level

fault coverage of the given test patterns. RTL and gate-level circuits are simulated using the fault simulator *Verifault-XL*™. Test pattern sets were written for circuits using their functional specifications. The error between RTL and gate-level fault coverage is expected to be within $\pm 3\sigma$ range with a confidence probability of 99.8% as per equation (3.1).

Module M1: 4-bit Counter

This module contains an RTL design description of a 4-bit counter with active high polarity load and enable signals as well as active low polarity asynchronous reset signal. It has 8 primary input and 4 primary output pins. It is designed using the Verilog HDL and the number of lines of code is 28. The number of RTL faults for this design is 24. The post-synthesis gate-level netlist contains 4 flip-flops and its size is equiavalent to 55 two-input CMOS gates. The size of the gate-level fault-list is 72 non-collapsed and 62 collapsed faults. Figure 18 shows the schematic of the gate-level netlist. Table 6 shows empirical data of RTL and gate-level fault simulations. For all data-points, the true error is found to be within statistically calculated error bounds.

Table 6 Empirical Data: 4-bit Counter Module

RTL Fault Coverage (%)	Gate Fault Coverage (%)	True Error (RTL Cov – Gate Cov) (%)	Error Bound For $\pm 3\sigma$ Range (%)
20.8	25.3	−4.5	±22.6
58.3	75.8	−17.5	±26.3
79.2	86.3	−7.1	±22.6
91.7	92.9	−1.2	±17.5
95.8	96.8	−1.0	±14.9

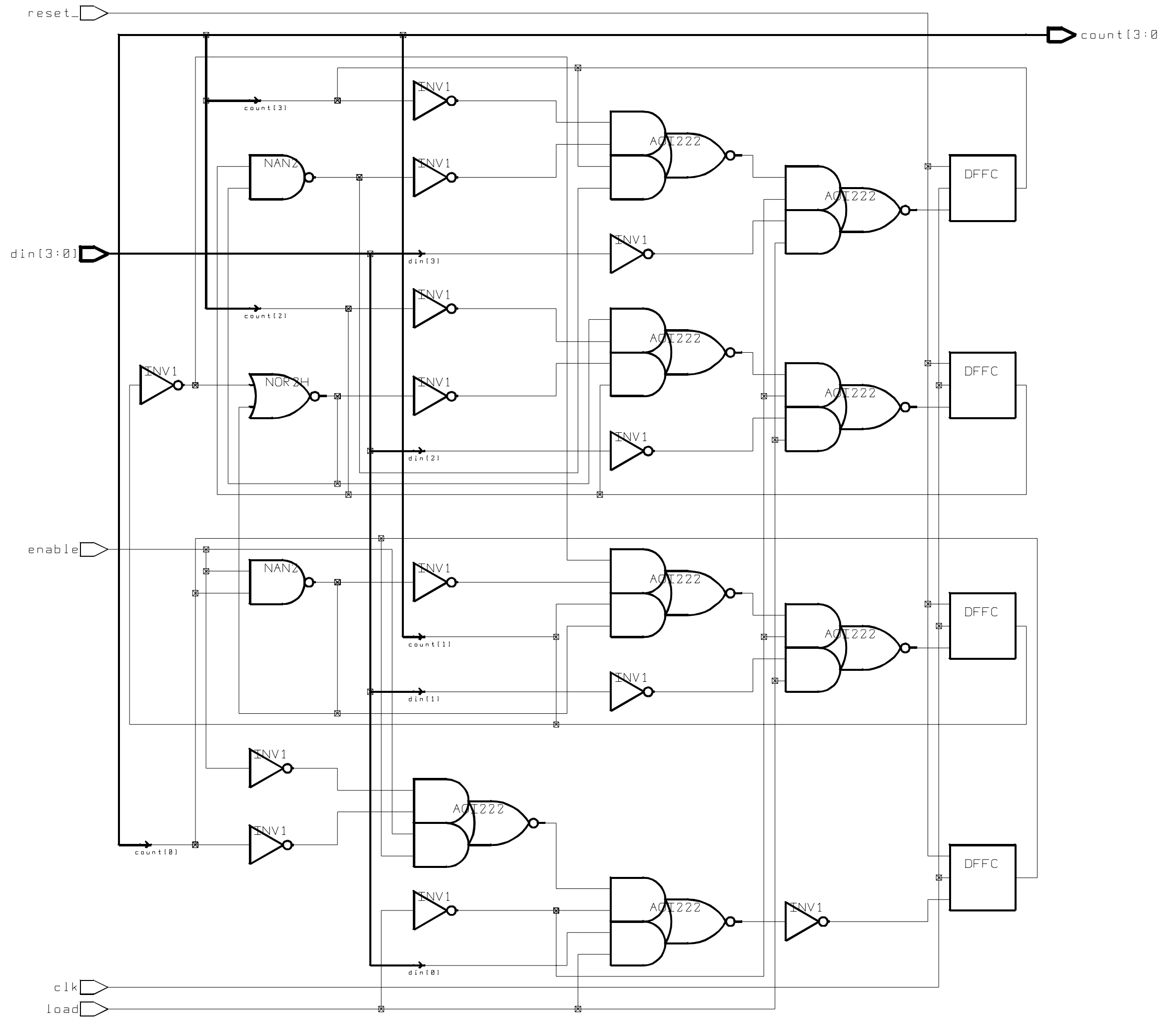


Figure 18 Gate Level Schematic: 4-bit Counter Module

Module M2: Transmit Buffer Module

This module contains the RTL design description of a buffer interface with the primary function of providing temporary storage and synchronization prior to data transmission. It has 37 primary input and 10 primary output pins. It is designed using the Verilog HDL and the number of lines of code is 214. The number of RTL faults for this design is 390. The post-synthesis gate-

level netlist contains 21 flip-flops and its size is equiavalent to 428 two-input CMOS gates. The size of the gate-level fault-list is 1192 non-collapsed and 651 collapsed faults. Table 7 shows several empirical data-points from the RTL and the gate-level fault simulation. Figure 19 shows a gradual improvement of the RTL and gate-level fault coverages. It can be observed that RTL fault coverage agrees with the true gate-level fault coverage within the calculated error bounds in most cases. Figure 20 plots the error between RTL and gate-level fault coverages versus RTL fault coverage.

Table 7 Empirical Data: Transmit Buffer Module

RTL Fault Coverage (%)	Gate Fault Coverage (%)	True Error (RTL Cov – Gate Cov) (%)	Error Bound For $\pm 3\sigma$ Range (%)
5.4	8.9	−3.5	±2.2
25.1	28.0	−2.9	±4.2
49.5	54.1	−4.6	±4.8
77.9	80.8	−2.9	±4.0
82.6	83.6	−1.0	±3.7

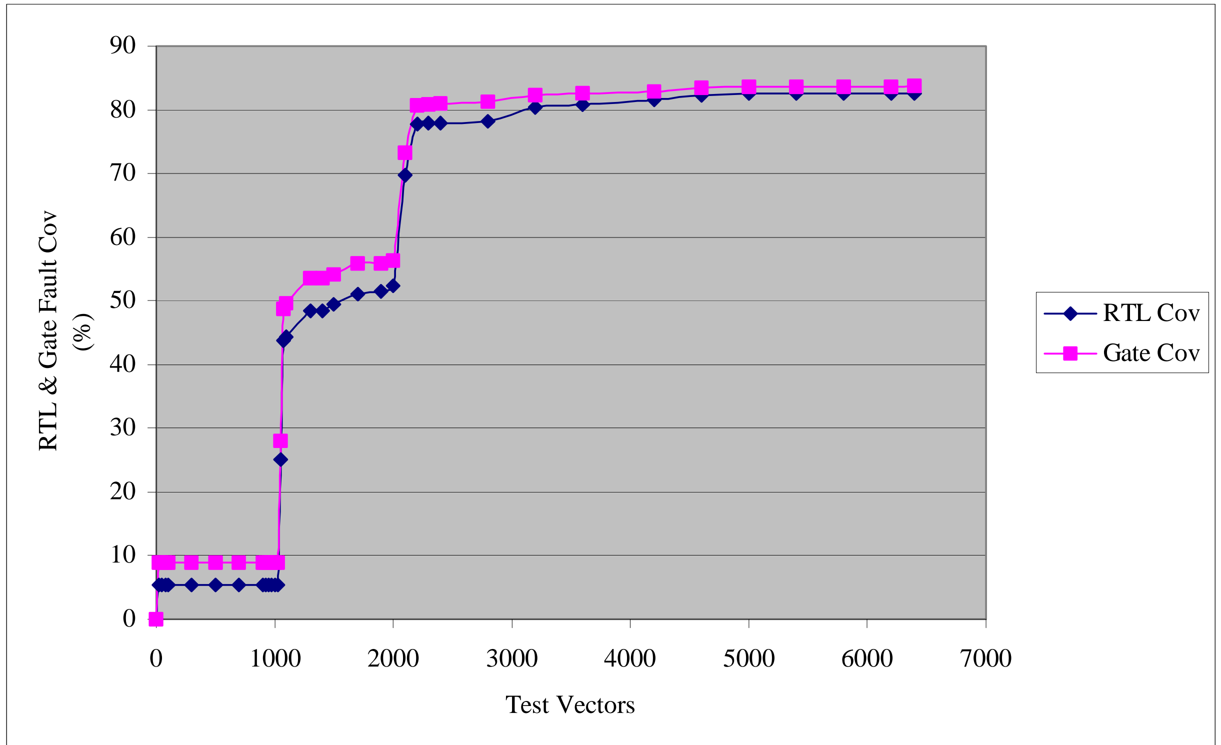


Figure 19 RTL and Gate Level Fault Coverage vs. Test Vectors: Transmit Buffer Module

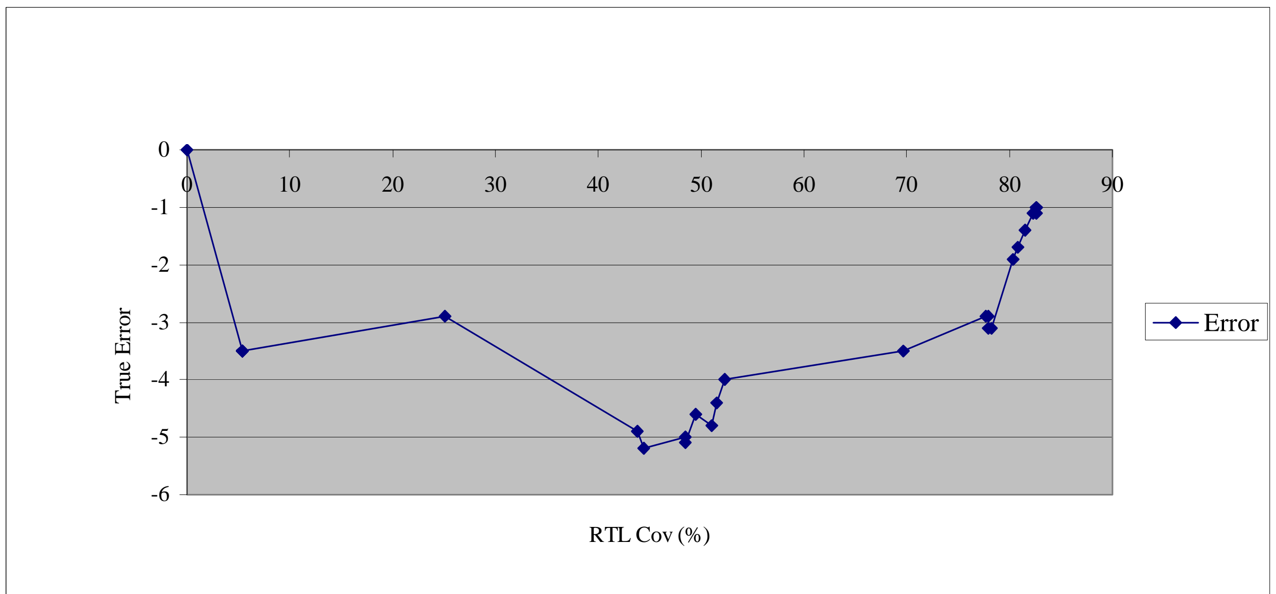


Figure 20 Error vs. Fault Coverage: Transmit Buffer Module

Module M3: SDRAM Controller Module

As the name indicates, this module is an SDRAM controller. It has 312 primary input and 68 primary output pins. It is designed using the Verilog HDL and the number of lines of code is 819. The number of RTL faults for this design is 976. The post-synthesis gate-level netlist contains 16 flip-flops and its size is equivalent to 981 two-input CMOS gates. The size of the gate-level fault-list is 4100 non-collapsed and 2262 collapsed faults. Table 8 shows several empirical data-points of RTL and gate-level fault simulations. Figure 21 shows a gradual improvement of the RTL and gate-level fault coverages. It can be observed that the RTL fault coverage agrees with the true gate-level fault coverage within the calculated error bounds in all cases. Figure 22 plots the error between RTL and gate-level fault coverages versus RTL fault coverage.

Table 8 Empirical Data: SDRAM Controller Module

RTL Fault Coverage (%)	Gate Fault Coverage (%)	True Error (RTL Cov – Gate Cov) (%)	Error Bound For $\pm 3\sigma$ Range (%)
19.4	19.0	+0.4	±2.9
42.0	42.8	–0.8	±3.6
52.9	54.6	−1.7	±3.6
65.7	67.2	−1.5	±3.4
78.0	75.8	+2.2	±3.0
96.6	95.5	+1.1	±1.3

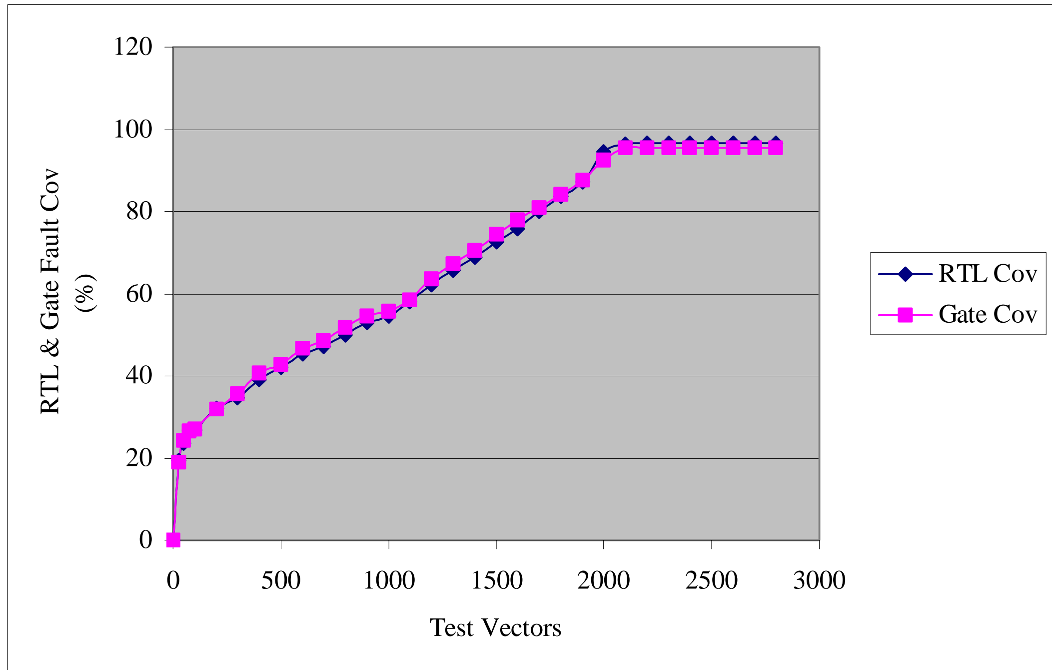


Figure 21 RTL and Gate Level Fault Coverage vs. Test Vectors: SDRAM Controller Module

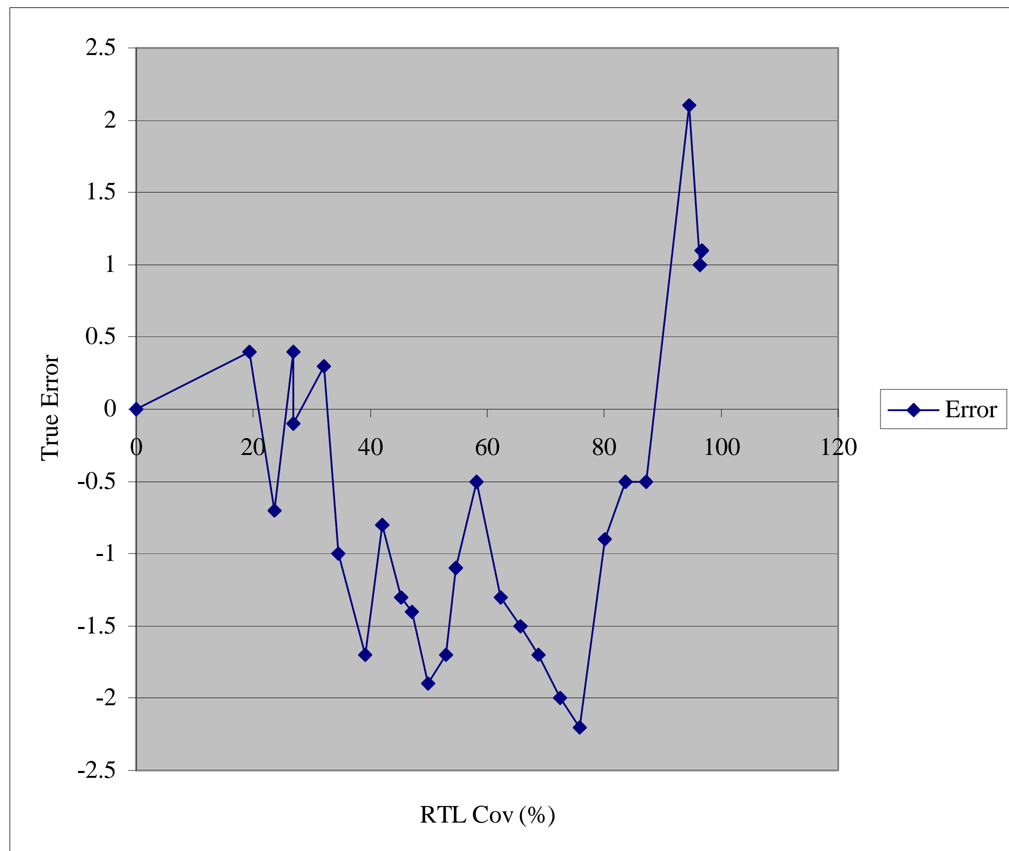


Figure 22 Error vs. Fault Coverage: SDRAM Controller Module

Module M4: DSP Interface Module

As the name indicates, this module is an interface circuit to a digital signal processor. It has 122 primary input and 190 primary output pins. It is designed using the Verilog HDL and the number of lines of code is 648. The number of RTL faults for this design is 1490. The post-synthesis gate-level netlist contains 228 flip-flops and its size is equivalent to 3168 two-input CMOS gates. The size of the gate-level fault-list for this design is 7002 non-collapsed and 5500 collapsed faults. Table 9 shows several empirical data-points of RTL and gate-level fault simulations. Figure 23 shows a gradual improvement of RTL and gate-level fault coverages. It can be observed that the RTL fault coverage agrees with the true gate-level fault coverage within the calculated error bounds in most cases. In cases, when the errors are outside the calculated bound, the deviation is very small. Figure 24 plots the error between RTL and gate-level fault coverages versus RTL fault coverage.

Table 9 Empirical Data: DSP Interface Module

RTL Fault Coverage (%)	Gate Fault Coverage (%)	True Error (RTL Cov – Gate Cov) (%)	Error Bound For $\pm 3\sigma$ Range (%)
16.2	15.4	+0.8	±2.5
24.2	23.9	+0.3	±2.9
47.9	44.5	+3.4	±3.3
70.7	68.9	+1.8	±3.0
84.1	81.6	+2.5	±2.4
87.3	84.5	+2.8	±2.2

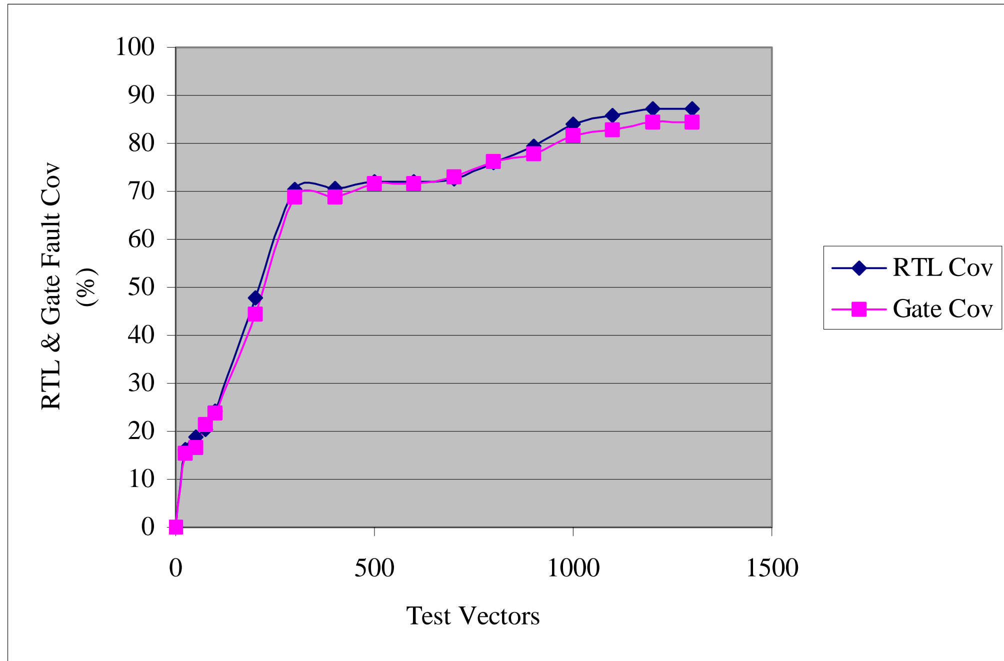


Figure 23 RTL and Gate Level Fault Coverage vs. Test Vectors: DSP Interface Module

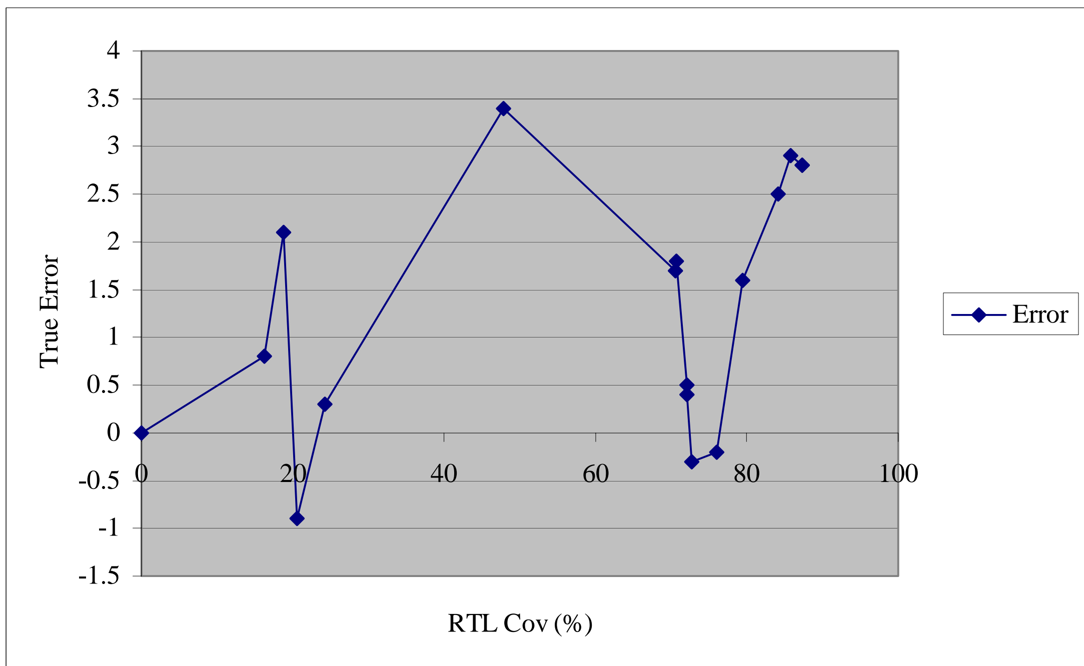


Figure 24 Error vs. Fault Coverage: DSP Interface Module

In this chapter, new RTL fault model and fault-injection algorithm are presented. The relationship of RTL faults to the hardware gate faults is also analyzed, and the effectiveness of the proposed fault model is verified by comparing RTL and gate-level fault coverages of several industry-application circuits. From the experimental data, it can be conclusively derived that the RTL fault coverage is a good estimate of the gate-level fault coverage for medium to large size modules. It is observed that for very small modules such as module M1 (4-bit counter), the error between the RTL and the gate-level fault coverages, though within statistically calculated error bounds, is large (more than 5%). This characteristic is a function of the sample size as well as population size and it is present in the random sampling technique as well [AGR81]. However, modern VLSI systems contain many modules of a variety of sizes and large estimation errors in a few small modules are insignificant while calculating the overall fault coverage. This argument is further clarified in later chapters. The main conclusion of this chapter is that the proposed RTL fault model can be used to estimate the gate-level fault coverage of a module within statistical error bounds. In the next chapter, the effectiveness of the proposed RTL fault model is analyzed for large VLSI systems that contain many modules.

CHAPTER 4

STRATIFIED SAMPLING: THEORY AND APPLICATION

To verify that the RTL fault model introduced in the previous chapter will provide a good representation of the gate-level faults, several experiments were conducted. The results reveal that the RTL fault coverage provides a good estimate of the gate-level fault coverage. The experimental data also reveals that there is no straightforward relationship between the number of RTL faults and that of gate-level faults. Considering the fault modeling method, the number of RTL faults is a measure of the size of the RTL description of the module. However, this number does not correlate with the gate count. For example, for module M1, the number of gates and gate faults are more than twice the number of RTL faults. For modules M2 and M3, the gate count is closer to the number of RTL faults, but there are almost twice as many gate-level faults. For module M4, the number of gate-level faults is more than twice the number of RTL faults and the gate count is almost twice the number of gate faults. The main conclusion of these experiments is that the proposed RTL fault model can be used to estimate the gate-level fault coverage of a module. But, the total number of RTL faults in the module does not represent the size of the gate-level fault-list.

In general, a large VLSI system consists of many modules. The lack of a defined relationship between the number of RTL faults and the number of possible gate-level faults presents a problem. A module with a large contribution of RTL faults to the overall RTL fault-list of the VLSI system may get synthesized into a relatively smaller gate-level implementation and subsequently make a smaller contribution of gate-level faults to the overall gate-level fault-list of that VLSI system. Similarly, a module that contributes fewer faults to the RTL fault-list of the VLSI system may result in a relatively larger gate-level implementation after logic synthesis, and thus contribute a larger percentage of faults to the overall gate-level fault-list of the system. Therefore, although the RTL fault-list of each module in a VLSI system is a representative sample of the corresponding gate-level fault-list of that module, the overall RTL fault-list of the system does not constitute a representative sample of the overall gate-level fault-list. Table 10 illustrates this problem for a hypothetical VLSI system consisting of two modules, m1 and m2. Based on the observation of the experiments in the previous chapter, it is assumed that the measured RTL fault coverage is close to the gate-level fault coverage in each module. The

overall RTL fault coverage of the system is obtained as $(91 \times 100 + 39 \times 100)/200 = 65\%$. The overall gate-level fault coverage of the system is calculated as $(90 \times 150 + 40 \times 400)/550 = 54\%$. It is found that the overall coverages are quite different. This is because the two modules, although equal in size at the RT level, translate into quite different sizes at the gate-level. In order to find the gate-level fault coverage, modules' RTL coverages should be weighted according to their relative gate-level sizes. Weighted RTL fault coverage can be obtained as $91 \times (150/550) + 39 \times (400/550) = 53\%$.

Table 10 Inaccurate Estimation of Gate Fault Coverage in a VLSI System

Mod. Level	m1 Faults		m2 Faults		m1+ m2 Coverage
	Total	Covered	Total	Covered	
RTL	100	91%	100	39%	65%
Gate	150	90%	400	40%	54%

The above observation leads us to consider a technique known as *stratified sampling* [STU84]. Random fault sampling is frequently used for fault coverage estimation as described by Agrawal [AGR81], Case [CAS75], McNamer et al. [MCN89] and Daehn [DAE91]. Generally, a randomly selected subset of faults is simulated and the coverage in the subset is used as an estimate of the total coverage. The error of this estimate can be mathematically quantified and is known to decrease as $O(n^{-0.5})$, where *n*, called the sample size, is the number of faults that are simulated. This procedure works well for any one type of faults, i.e., single stuck-at faults, as is the case usually. However, when there are several different types of fault population*s* as in defect-oriented fault modeling, one must use stratified sampling [GON98]. Stratified sampling theory is described in Section 4.1. The application of stratified sampling theory to determine the RTL fault coverage of the VLSI system is developed in Section 4.2.

4.1 Stratified Sampling Theory

Stratified sampling [MUR67, COC77] consists of classifying the population units into several non-overlapping groups, called *strata*, and then selecting samples independently from

each group (stratum). An appropriate estimator of the characteristic under consideration for the population as a whole is obtained by suitably combining the stratum-wise estimators. Suppose, a population of N units is first divided into L sub-populations of N_1, N_2, . . .,N_L units, respectively. These sub-populations are non-overlapping such that,

$$N_1 + N_2 + N_3 + ... + N_L = N \tag{4.1}$$

When the strata have been determined, a sample is drawn from each, the drawings being made independently in each stratum. The sample sizes within the strata are denoted by n_1, n_2, n_3, ...,n_L, respectively. In the following notation, the subscript h denotes the stratum and i the unit within the stratum,

N	Population
N_h	Number of units in the hth stratum
n_h	Number of units in a sample from h^{th} stratum
y_{hi}	Value of the ith unit in h^{th} stratum
$W_h = \frac{N_h}{N}$	Weight of the hth stratum
$f_h = \frac{n_h}{N_h}$	Sampling fraction in h^{th} stratum
$\hat{Y}_h = \frac{\sum_{i=1}^{N_h} y_{hi}}{N_h}$	True mean for h^{th} stratum
$\hat{y}_h = \frac{\sum_{i=1}^{n_h} y_{hi}}{n_h}$	Sample mean for h^{th} stratum
$S_h^{\ 2} = \frac{\sum_{i=1}^{N_h} (y_{hi} - \hat{Y}_h)^2}{N_h - 1}$	True variance for h^{th} stratum

For the population mean, the estimate used in stratified sampling is $\hat{y}_{st}$ ("*st*" for stratified), given by

$$\hat{y}_{st} = \frac{\sum_{h=1}^{L} N_h \hat{y}_h}{N} = \sum_{h=1}^{L} W_h \hat{y}_h \tag{4.2}$$

The estimate $\hat{y}_{st}$ is not in general the same as the sample mean. The sample mean, $\hat{y}$, can be written as,

$$\hat{y} = \frac{\sum_{h=1}^{L} n_h \hat{y}_h}{n} \tag{4.3}$$

where, $n = \sum_{h=1}^{L} n_h$. The difference between the two means is that in $\hat{y}_{st}$ the estimates from the individual strata receive their correct weights N_h / N . In order for $\hat{y}$ to coincide with $\hat{y}_{st}$, the sampling fraction must be the same in all strata. The principal properties of the estimate $\hat{y}_{st}$ as given by Cochran [COC77] are outlined below:

1) If in every stratum the sample estimate $\hat{y}_h$ is unbiased, then $\hat{y}_{st}$ is an unbiased estimate of the population mean $\hat{Y}$. Here, $\hat{Y}$ is the true value of the quantity one is trying to measure.

2) If simple random samples are taken within each stratum, an unbiased estimate of true variance for h^{th} stratum is given as,

$$s_h{}^2 = \frac{\sum_{i=1}^{n_h} (y_{hi} - \hat{y}_h)^2}{n_h - 1} \tag{4.4}$$

3) In stratified random sampling theory, an unbiased estimate of the true variance of $\hat{y}_{st}$ is,

$$s^2(\hat{y}_{st}) = \frac{1}{N^2} \sum_{h=1}^{L} \frac{N_h (N_h - n_h) s_h{}^2}{n_h} \tag{4.5}$$

4) The range within which the true mean of the population lies with some given confidence probability is

$$\hat{y}_{st} \pm ts(\hat{y}_{st}) \tag{4.6}$$

The estimate of the population mean, $\hat{y}_{st}$, is assumed to be normally distributed and $s(\hat{y}_{st})$ can be determined from equation (4.5). The multiplier "*t*" is the value of the deviation from the mean for a normal random variable with mean $\hat{y}_{st}$ and standard deviation $s(\hat{y}_{st})$, corresponding to the desired confidence probability. The value of "*t*" can be easily found from published probability table. Table 11 gives a few examples.

Table 11 Values of *t* for Range of Estimator

Confidence Prob. (%)	50	80	90	95	99	99.8
t	0.67	1.28	1.64	1.96	2.58	3.00

4.2 Application of Stratified Sampling to RTL Fault Modeling

In the present application, the fault population is divided according to RTL module boundaries. Thus, each module is considered a stratum. Within a stratum, the RTL faults are considered as a sample of all (i.e., gate-level) faults. The ratio of the number of RTL faults to the number of gate-level faults will be the sampling fraction for the stratum. In general, if the number of RTL faults in a module is larger, then a random sample of those can also be used.

Suppose the VLSI system has G gate-level faults distributed among M modules (or strata.) The mth module has G_m gate-level faults. Then,

$$G = \sum_{m=1}^{M} G_m \tag{4.7}$$

and,

$$Weight\ of\ m^{th}\ module,\ W_m = \frac{G_m}{G} \quad (4.8)$$

From the mth module, r_m RTL faults are simulated. These can either be all RTL faults (100% sample) or a random sample of all RTL faults in the mth module. In either case,

$$m^{th}\ \text{module's sampling fraction,}\ f_m = \frac{r_m}{G_m} \quad (4.9)$$

Further, two coverages for the mth module are defined as

$$C_m = \frac{Detected\ gate\ faults\ in\ m^{th}\ module}{G_m} \quad (4.10)$$

$$c_m = \frac{1}{r_m}\sum_{i=1}^{r_m} y_{mi} \quad (4.11)$$

where y_{mi} are random variables whose values are determined by fault simulation: $y_{mi} = 1$ if the ith sampled fault in m^{th} module is detected, or $y_{mi} = 0$ if that fault is not detected. Thus, y_{mi} is a random variable defined as

$$Prob(y_{mi} = 1) = C_m$$
$$Prob(y_{mi} = 0) = 1 - C_m$$

Also,

$$E(y_{mi}) = C_m$$
$$VAR(y_{mi}) = E(y_{mi} - C_m)^2 = C_m(1 - C_m)$$

Taking expectation of equation (4.11), we obtain

$$E(c_m) = C_m$$

That is, c_m is an unbiased estimate for the gate-level fault coverage of the module m. C_m is the true gate-level fault coverage in the mth module which provides the total gate-level coverage of the VLSI system as

$$C = \frac{\sum_{m=1}^{M} G_m C_m}{G} = \sum_{m=1}^{M} W_m C_m \tag{4.12}$$

Here module coverages have been weighted according to their sizes to eliminate the type of error illustrated in Table 10. The estimated value for the true gate-level coverage C of the VLSI system is called stratified RTL fault coverage and is obtained as

$$\overline{C} = \frac{\sum_{m=1}^{M} G_m c_m}{G} = \sum_{m=1}^{M} W_m c_m \tag{4.13}$$

Notice that the stratified RTL fault coverage $\overline{C}$ is different from the overall non-stratified RTL fault coverage, which is given by

$$C_{RTL} = \frac{\sum_{m=1}^{L} r_m c_m}{\sum_{m=1}^{M} r_m} \tag{4.14}$$

In general, the non-stratified RTL coverage can be quite different from the true gate-level coverage. Using equations (4.9), (4.13) and (4.14), it can be shown that the non-stratified RTL coverage will coincide with the stratified RTL coverage if the sampling fraction, f_m, was identical for all modules. This is a known result in the theory of stratified sampling [COC77]. However, f_m is the ratio of the number of sampled RTL faults to the total number of gate-level faults. So, if RTL faults are sampled in proportion to the gate-level faults in modules, then the RTL coverage of equation (4.14) can be directly used as an estimate of the gate-level coverage. In practice, some modules may be small and such a scheme will take very small fault samples from those modules, adversely affecting the accuracy of the estimate. It is desirable to take 100% RTL fault

samples in small modules, while smaller percentage samples are taken from very large modules. Thus, the sampling fraction will differ from module to module and equation (4.13) should be used.

The discussion and empirical data provided in Chapter 3 shows that the RTL fault coverage of a module is a close estimate of the gate-level coverage of that module. That is, given a vector set, almost identical fractions of the two types of faults are detected, or a randomly selected RTL fault has the same probability of being detected as a gate-level fault. By taking the statistical expectation of equation (4.13), it can be shown that $\overline{C}$ is an unbiased estimate of the true coverage C, or

$$E(\overline{C}) = \sum_{m=1}^{M} W_m E(c_m) = \sum_{m=1}^{M} W_m C_m = C \tag{4.15}$$

The variance of the estimate is evaluated as

$$\sigma^2 = \sigma^2(\overline{C}) = E(\overline{C} - C)^2 \tag{4.16}$$

On simplification [COC77], this leads to

$$\sigma^2 = \frac{1}{G^2} \sum_{m=1}^{M} \frac{G_m (G_m - r_m) \sigma_m^2}{r_m} \tag{4.17}$$

where $\sigma_m{}^2$ is the unbiased estimate for the variance of y_{mi}, given by

$$\sigma_m^2 = \frac{1}{r_m - 1} \sum_{i=1}^{r_m} (y_{mi} - c_m)^2 \tag{4.18}$$

In equation (4.18), $y_{mi} = 1$ for $c_m r_m$ detected faults and $y_{mi} = 0$ for $(1 - c_m) r_m$ undetected faults. Therefore,

$$\sigma_m{}^2 = \frac{1}{r_m - 1} (c_m r_m (1 - c_m)^2 + (1 - c_m) r_m c_m{}^2) = \frac{r_m c_m (1 - c_m)}{r_m - 1} \tag{4.19}$$

Equation (4.17) can be expanded as,

$$\sigma^2 = \sum_{m=1}^{M}\left(\frac{G_m{}^2}{G^2} - \frac{G_m r_m}{G^2}\right)\frac{\sigma_m{}^2}{r_m} = \sum_{m=1}^{M}\left(W_m{}^2 - \frac{W_m r_m}{G}\right)\frac{\sigma_m{}^2}{r_m} \tag{4.20}$$

Since RTL module description is significantly more compact compared to gate-level description, $r_m << G_m$. Therefore,. Simplifying equation (4.20) based on the approximation,

$$\sigma^2 = \sum_{m=1}^{M}\frac{W_m{}^2\sigma_m{}^2}{r_m} \tag{4.21}$$

Substituting equation (4.19) into (4.21),

$$\sigma^2 = \sum_{m=1}^{M}\frac{W_m{}^2}{r_m - 1}c_m(1 - c_m) \tag{4.22}$$

With the above analysis, the stratified sampling procedure is outlined as follows:

- The VLSI system consists of interconnections of M modules described in an RTL language. The RTL fault-lists are generated for all modules according to the fault model and fault-injection algorithm proposed in Chapter 3.
- For each module a set of r_m RTL faults is selected, where $m = 1 \ldots M$. These can be either all RTL faults in the module or a random subset of those. Selected RTL faults are simulated and module coverages c_m are determined.
- The gate-level coverage estimate $\overline{C}$ is computed from equation (4.13). The gate-level coverage estimate is referred to as stratified RTL fault coverage. The determination of weights, W_m, that are used in this equation are discussed in Chapter 5.
- The variance σ^2 for the estimate is computed from equation (4.22). For a given confidence probability, the range of coverage is given as

$$\overline{C} \pm t\sigma \tag{4.23}$$

where t is the limit for which the area of the normalized (zero mean and unity variance) Gaussian probability density equals some given confidence probability. The values of t can be selected from Table 11.

As elaborated in Chapter 3, smaller RTL modules have larger error bounds. However, smaller modules have proportionately smaller stratum weights reducing the effect of larger error bounds on the stratified RTL fault coverage estimate of the VLSI system.

It is self-evident from equations (4.13), (4.22) and (4.23) that none of the parameters that are crucial in determining stratified RTL fault coverage and error bounds require a knowledge of the absolute values of G_m or G. They require the ratio of the two quantities in the form of stratum weights. Techniques developed to determine the stratum weights of modules in a given VLSI system are described in Chapter 5.

CHAPTER 5

STRATUM WEIGHT EXTRACTION TECHNIQES

The stratified RTL fault coverage serves as a good estimate of the gate-level fault coverage of a VLSI system. As illustrated in Chapter 4, the stratified RTL fault coverage can be determined using the RTL fault coverage and the stratum weight of each of the modules of a VLSI system. The RTL fault coverage of modules can be determined using the fault model and the fault simulation technique depicted in Chapter 3. The stratum weight of a module is the ratio of the number of gate faults of the module to the total number of gate faults of the VLSI system. In this chapter, several different approaches are proposed for determining the stratum weights of modules of a VLSI system. An approach that requires the use of the CMOS technology library in determining stratum weights is considered to be a technology-dependent approach. Approaches that do not require the use of the CMOS technology library are considered technology-independent. The technology library is a collection of pre-designed gates and sequential elements available in a particular implementation technology supported by a semiconductor manufacturer.

5.1 Technology-dependent Stratum Weight Extraction

As described in Chapter 4, the gate-level fault population of a VLSI system is divided into non-overlapping sub-populations called "strata" at module boundaries. The weight of the module *m* is given as,

$$W_m = \frac{G_m}{G} = \frac{\text{Number of gate faults in the } m^{th} \text{ module}}{\text{Total number of gate faults in the VLSI system}} \tag{5.1}$$

The information needed in equation (5.1) to calculate the weight of a stratum could be obtained from the gate-level fault-list of the VLSI system. The gate-level fault-list can be generated from the gate-level netlist as is usually done by fault simulators. The gate-level netlist of the VLSI system can be obtained by performing logic synthesis on the RTL code for a specific technology library. The absolute values of G_m and G can change when the synthesis is performed using different optimization (or cost) constraints. Since the synthesis usually involves iterations with

changes of constraints, accurate stratum weights can only be obtained after the final logic synthesis is completed. However, the final logic synthesis is performed very late in the design cycle. The goal of the proposed RTL fault model and test evaluation technique is to enable an accurate estimation of the gate-level fault coverage early in the design cycle. Therefore, there is a need to generate stratum weights prior to final logic synthesis. As illustrated in Chapter 6 with an example, even though the absolute values of G_m and G obtained from different gate-level netlists of the same RTL code vary, their ratio given as, W_m , does not vary as much. The impact of variation in W_m on the stratified RTL fault coverage and error bounds is assumed to be negligible. Therefore, it is proposed that the RTL description of the given VLSI system should be synthesized for the preferred technology library, using several different optimization constraints. Several gate-level netlists, each of which is unique in its structure to meet specific cost constraints will be generated from such an effort. Unique fault-lists are created using a traditional fault simulator from each gate-level netlist. The stratum weight can be calculated from each gate-level fault-list. For the stratum, the mean weight is obtained by taking a statistical average of weights obtained from those fault-lists.

5.2 Technology-independent Stratum Weight Extraction

5.2.1 Weights Based on Logic Synthesis

It is important to understand logic synthesis flow (see Figure 25) in order to establish a technology-independent stratum weight extraction technique. As described by Devadas et al. [DEV94], in a typical synthesis flow, RTL code is first translated into a mixture of combinational logic gates and sequential memory elements. The RTL code that is input to the translator describes the micro-architecture of the circuit with pre-determined and built-in information about event-scheduling and resource allocation. Therefore, the style used in the RTL design description, controls the translation process. Gate level description resulting from the translation process often contains a sub-optimal logic implementation. The goal of next step -- that is, logic optimization -- is to transform this sub-optimal design description into a closer-to-optimal implementation of the circuit in terms of area and speed. Logic optimization aims at minimizing

area while meeting the speed constraints. The combinational logic implementation can either be a two-level implementation or a multilevel implementation. In the post-translation phases of logic synthesis, the synthesized sequential memory elements remain unchanged. The result of the logic optimization step is a gate-level netlist of combinational and sequential elements. The next step is technology mapping in which the optimized netlist is efficiently mapped onto a CMOS technology library of pre-designed standard elements available from a semiconductor manufacturer. It is assumed that the Boolean network has gone through significant technology-independent optimization and therefore the structure of the circuit will not be changed drastically during technology mapping. The primary challenge of this phase is to choose the fastest gates from the CMOS technology library along the critical paths and to use the most area-efficient combination of gates along non-critical paths.

In a VLSI system, modules with larger gate counts contribute a proportionately large percentage of gate faults to the overall gate fault-list. Similarly, smaller size modules contribute a proportionately smaller percentage of faults to the overall gate fault-list. Thus, the distribution of gate-faults of a VLSI system among various modules is proportional to their respective sizes measured in terms of gate counts. Therefore, the stratum weight of a module can be approximated as a ratio of the number of gates in the module to the total number of gates in the VLSI system,

$$W_m \cong \frac{A_m}{A} = \frac{\textit{Area of the } m^{th} \textit{ module}}{\textit{Total area of the VLSI system}} \tag{5.2}$$

As described earlier, in the integral process of logic synthesis, the gate-level representation of the RTL code stabilizes prior to the technology mapping. The stratum weights can be extracted from this pre-technology-mapped representation using the gate count information of the module and the overall VLSI system. Contemporary industry-standard logic synthesis tools support this approach [BHA99]. An error introduced in stratum weights due to approximation inherent in the proposed approach is observed to have insignificant impact on the stratified RTL fault coverage and predicted error bounds. Experimental data is presented in Chapter 6.

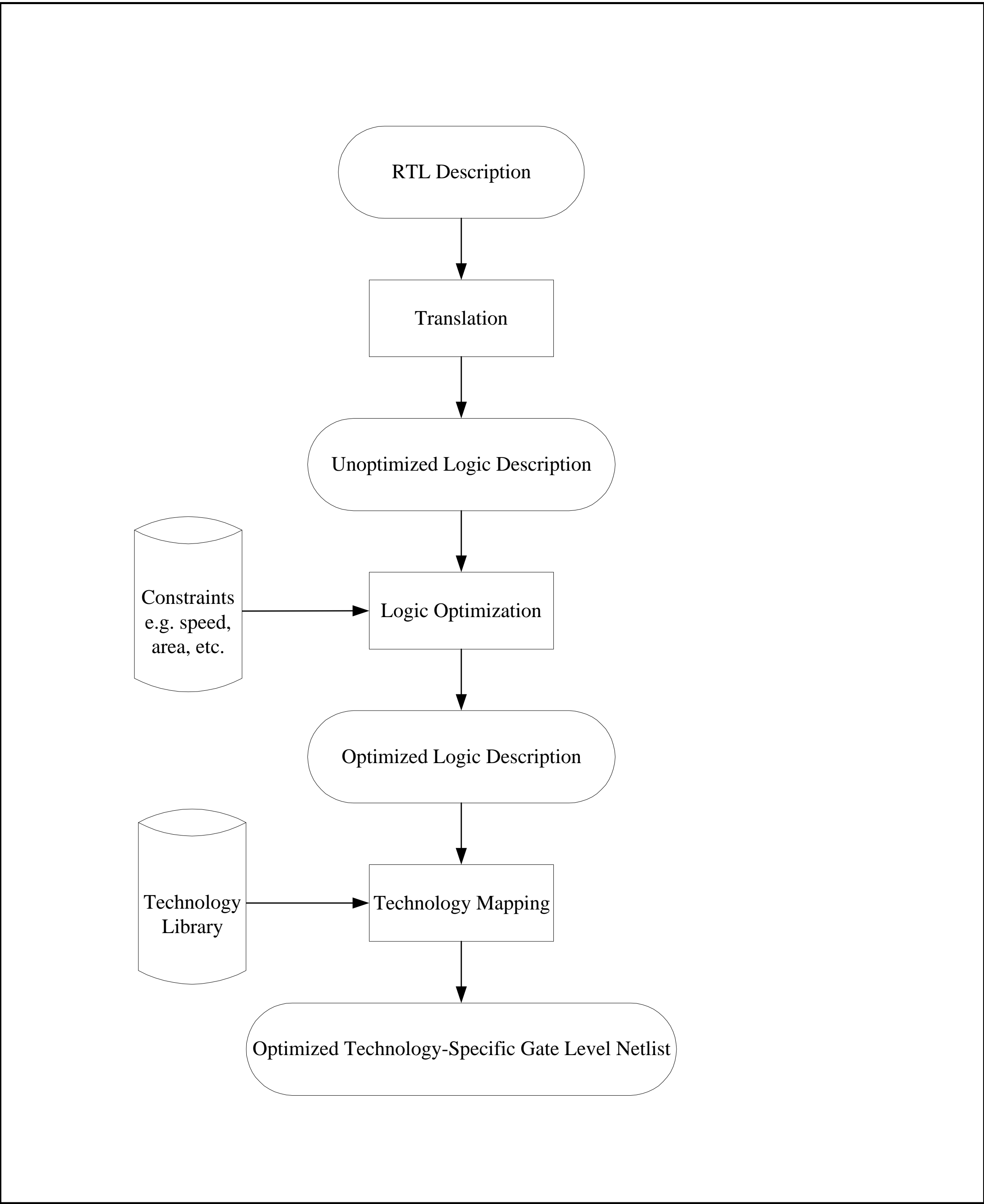


Figure 25 Logic Synthesis Flow

5.2.2 Weights Based on Entropy Measure

The complexity or the size of a Boolean function can be expressed in terms of computational work. Computational work is measured based on the information theoretic entropy of the output signals. Computational work is shown to have a direct relation to the hardware needed to implement the function. Although statistical measures have not been used in modern logic synthesis, Cheng and Agrawal [CHE90], and Pippenger [PIP77] claim that it is possible to estimate the area of a circuit (which is proportional to the number of gates) from its functional description. An approach for area estimation based on statistical measures using functional description can be employed to derive stratum weights. The stratum weights can be obtained using equation (5.2). This approach as well as the next one needs further investigation.

5.2.3 Weights Based on Floor-planning

The architecture (a block diagram consisting of several functional modules) of a VLSI system is an early step in the design. Floor-planning tools estimate the areas of these modules based on the function, complexity, number of inputs and outputs, etc. This information is used to plan the layout of the ASIC. Even though, the area information is rather approximate and is subject to change as the design progresses, the relative areas of the modules can be used to determine their weights by using the equation (5.2).

CHAPTER 6

EXPERIMENTAL PROCEDURE AND RESULTS

The stratified RTL fault coverage provides a good estimate of the gate-level fault coverage of a VLSI system. As illustrated in Chapter 4, the stratified RTL fault coverage can be determined using the RTL fault coverages and the stratum weights of all modules of the VLSI system. The RTL fault coverage of a module can be determined using the fault model and the fault simulation technique described in Chapter 3. The stratum weights of modules can be determined by using the techniques illustrated in Chapter 5. In this chapter, data generated from several experiments conducted to verify the theory proposed in Chapters 3, 4 and 5, is presented.

6.1 Estimation of the Gate-level Fault Coverage of a VLSI System

6.1.1 Procedure

Several industry-application VLSI systems ranging in size from 1,000 gates to 120,000 gates were used for empirical studies. The experimental procedure developed on the basis of the proposed RTL fault model, the fault-injection algorithm and the application of stratified sampling theory is as follows,

(1) The RTL code is run through a C++ parser (see Appendix B). This parser processes the RTL code and generates the fault-injected RTL code without altering the circuit behavior. The parser is developed according to the fault model and fault-injection algorithm proposed in Chapter 3.

(2) The VLSI system consists of interconnections of modules described in an RTL language. For each module, a set of RTL faults is selected. These can be all RTL faults in the module or a random subset of them. Selected RTL faults are simulated for the given set of test patterns and module coverages are determined.

(3) Stratum weights for modules are computed using any of the techniques given in Chapter 5.

(4) The stratified RTL fault coverage is computed using equation (4.13). The stratified RTL fault coverage serves as an estimate of the gate-level fault coverage of the VLSI system. The tolerance range of the estimated coverage is obtained using equation (4.23).

The experimental procedure described above was carried out using commercial electronic design automation tools. The RTL fault simulation was performed using the *Verifault-XL™* simulator. Among the proposed techniques for stratum weight extraction, technology-dependent and logic-synthesis based technology-independent approaches were used. The stratum weight extraction procedures were performed using *Verifault-XL™* and *Design Compiler™*. The stratified RTL fault coverages for each VLSI system were calculated using both technology-dependent and technology-independent stratum weights. These estimates of the gate-level fault coverage were compared with the non-stratified RTL fault coverage calculated using equation (4.14) as well as the actual gate-level fault coverage. The actual gate-level fault coverage of each VLSI system was obtained by fault grading the gate-level netlist. Gate-level fault simulations were preformed using *Verifault-XL™*. The gate-level netlist for each VLSI system was obtained by performing logic synthesis of the RTL code for an arbitrary set of optimization constraints. *Design Compiler™* was used for logic synthesis. Test patterns used for the RTL and gate-level fault simulations were manually generated using functional specifications of the systems. The procedure outlined here is typically used in many actual design environments in the industry.

6.1.2 Experimental Data

The experimental data for three real-life industry-application VLSI systems is presented in this section.

VLSI System D1: Frame-Timing Control ASIC

This VLSI system contains several watchdog timers, scheduling/control status read/update function and frame synchronization logic. It has 36 primary input and 71 primary output pins. It is designed using the Verilog HDL and the number of lines of code is 673. The number of RTL faults for this design is 678. The post-synthesis gate-level netlist contains 88 flip-flops and its size is equivalent to 1,170 two-input CMOS gates. The size of the gate-level fault-

list is 2,254 non-collapsed and 1,332 collapsed faults. This system consists of two modules, m1 and m2. The modules, m1 and m2 contain 440 and 238 RTL faults, respectively. Their stratum weights are determined to be 0.797 and 0.202, respectively, using the technology-dependent technique, and 0.809 and 0.1909, respectively, using the logic-synthesis based technology-independent technique. Table 12 contains the empirical data generated from the experiments.

Table 12 Empirical Data: Frame-Timing-Control ASIC

Test	C_{RTL} (%)	$\overline{C}$ (%)		C (%)	Estimation Error (%)			
					W_{tech}		$W_{tech-ind}$	
		W_{tech}	$W_{tech-ind}$		$\overline{C}-C$	3σ Bound	$\overline{C}-C$	3σ Bound
T1	24.3	23.8	23.8	25.6	–1.8	±5.1	–1.8	±5.2
T2	35.8	27.7	27.1	29.3	–1.6	±4.6	–2.2	±4.7
T3	44.1	32.9	32.1	35.4	–2.5	±4.5	–3.3	±4.5
T4	42.6	32.2	31.4	35.9	–3.7	±4.6	–4.5	±4.6
T5	43.4	35.3	34.8	38.9	–3.6	±5.2	–4.1	±5.2
T6	87.3	81.9	81.6	82.0	–0.1	±4.8	–0.4	±4.9
T7	84.8	82.0	81.9	82.7	–0.7	±4.8	–0.8	±4.8

Legend:

C_{RTL} = Non-stratified RTL fault coverage calculated using equation (4.14)

$\overline{C}$ = Stratified RTL fault coverage calculated using equation (4.13)

C = Gate-level fault coverage

W_{tech} = Stratum weights determined using the technology-dependent approach

$W_{tech-ind}$ = Stratum weights determined using the technology-independent approach

Error bounds are calculated for $\pm 3\sigma$ range using equation (4.23).

True Error = $\overline{C}-C$

T1 through T7 indicate test pattern sets generated using functional specifications.

VLSI System D2: System Timing Control ASIC

This VLSI system contains several state-machine based controller functions. It is highly programmable and supports a total of 2^{18} different modes. It has 16 bi-directional, 15 primary input and 7 primary output pins. It is designed using the Verilog HDL and the number of lines of code is 9,000. The number of RTL faults for this design is 24,982. The post-synthesis gate-level netlist contains 750 flip-flops and its size is equivalent to 17,126 two-input CMOS gates. The size of the gate-level fault-list is 45,688 non-collapsed and 29,670 collapsed faults. This system consists of six modules. Table 13 contains empirical data generated from the experiments.

Table 13 Empirical Data: System Timing Controller ASIC

Test	C_{RTL} (%)	$\overline{C}$ (%)		C (%)	Estimation Error (%)			
		W_{tech}	$W_{tech-ind}$		W_{tech}		$W_{tech-ind}$	
					$\overline{C}-C$	3σ Bound	$\overline{C}-C$	3σ Bound
T1	48.9	52.6	54.4	51.7	+0.9	±0.94	+2.7	±0.94
T2	39.7	42.6	43.6	43.9	−1.3	±0.99	−0.3	±1.00
T3	29.0	30.6	31.4	31.5	−0.9	±0.95	−0.1	±0.97
T4	55.2	59.5	61.1	60.2	−0.7	±0.89	+0.9	±0.88
T5	52.3	55.9	57.6	54.7	+1.2	±0.93	+2.9	±0.92
T6	56.8	60.9	62.4	61.5	−0.6	±0.90	+0.9	±0.89

Legend:

C_{RTL} = Non-stratified RTL fault coverage calculated using equation (4.14)

$\overline{C}$ = Stratified RTL fault coverage calculated using equation (4.13)

C = Gate-level fault coverage

W_{tech} = Stratum weights determined using the technology-dependent approach

$W_{tech-ind}$ = Stratum weights determined using the technology-independent approach

Error bounds are calculated for $\pm 3\sigma$ range using equation (4.23).

True Error = $\overline{C} - C$

T1 through T6 indicate test pattern sets generated using functional specifications.

VLSI System D3: Digital Signal Processor ASIC

This VLSI system contains various types of signal processing functions such as filtering, interpolation, wave-shaping circuits, etc., implemented in hardware. It contains data-path as well as control functions. It has 10 bi-directional, 34 primary input and 51 primary output pins. It is designed using the Verilog HDL and the number of lines of code is 8,580. The number of RTL faults for this design is 27,108. The post-synthesis gate-level netlist contains 2,600 flip-flops and its size is equivalent to 120,000 two-input CMOS technology gates. The size of the gate-level fault-list is 107,290 non-collapsed and 73,374 collapsed faults. This system consists of twelve modules. Table 14 shows empirical data generated from the experiments.

6.1.3 Discussion

Experimental data provided in Table 12, Table 13 and Table 14 indicates that the stratified RTL fault coverage is a good estimate of the gate-level fault coverage. Also, in all except four cases, the true error is within statistically determined error bounds. As can be noted in Table 14, the error bounds for test pattern sets T1, T5, T6 and T7 are significantly wider than that for T2, T3 and T4. The RTL fault simulations for test pattern sets T1, T5, T6 and T7 were performed using a very small random sample of the overall RTL fault-list. As per the equation (4.22), the size of the RTL fault sample in each module and the error bounds of the overall stratified RTL coverage of the system are related. The smaller the number of RTL faults used in simulation, the wider the error bounds. This is confirmed by the data presented in Table 14. Therefore, in order to estimate the gate-level fault coverage within narrow error bounds, a reasonable number of RTL faults should be selected during simulations.

Table 14 Empirical Data: Digital Signal Processor ASIC

Test	C_{RTL} (%)	$\overline{C}$ (%)		C (%)	Estimation Error (%)			
					W_{tech}		$W_{tech-ind}$	
		W_{tech}	$W_{tech-ind}$		$\overline{C}-C$	3σ Bound	$\overline{C}-C$	3σ Bound
T1	56.4	55.1	52.3	57.7	−2.6	±8.0	−5.4	±7.5
T2	23.9	19.4	19.4	18.0	−1.4	±1.7	+1.4	±1.6
T3	57.7	59.2	59.7	61.4	−2.2	±2.4	−1.7	±2.3
T4	48.7	50.2	47.7	50.9	−0.7	±2.1	−3.2	±1.9
T5	59.4	57.2	54.4	57.7	−0.5	±7.9	−3.3	±7.4
T6	68.8	69.8	68.6	70.9	−1.1	±9.3	−2.3	±9.3
T7	73.3	74.9	73.6	72.7	+2.2	±8.8	+0.9	±8.9

Legend:

C_{RTL} = Non-stratified RTL fault coverage calculated using equation (4.14)

$\overline{C}$ = Stratified RTL fault coverage calculated using equation (4.13)

C = Gate-level fault coverage

W_{tech} = Stratum weights determined using the technology-dependent approach

$W_{tech-ind}$ = Stratum weights determined using the technology-independent approach

Error bounds are calculated for $\pm 3\sigma$ range using equation (4.23).

True Error = $\overline{C}-C$

T1 through T7 indicate test pattern sets generated using functional specifications.

6.2 Stratum Weight Estimation

The accurate stratum weights of the modules of a VLSI system can be obtained only after final logic synthesis. However, since the goal of the proposed RTL fault model and the test evaluation technique is to enable an accurate estimation of the gate-level fault coverage early in the design cycle, alternative approaches are proposed in Chapter 5. The stratum weights estimated from the proposed technology-dependent and logic-synthesis based technology-independent methods may be different from the ones obtained from the final gate-level netlist. In the theoretical analysis proposed in Chapter 4, the error resulting from such estimation is ignored with the assumption that even though the absolute values of G_m and G may vary significantly for different gate-level netlists, their ratio represented as stratum weights does not vary as much. The impact of difference between the estimated and actual stratum weights on the stratified RTL fault coverage and error bounds is assumed to be insignificant. These assumptions are illustrated in this chapter with empirical data.

For the VLSI system D3, several unique gate-level netlists were obtained by logic synthesis. Each netlist is generated using a different set of optimization constraints. Stratum weights were obtained for each netlist using the technology-dependent stratum weight extraction technique described in Chapter 5. The stratum weights were also obtained using a logic-synthesis based technology-independent technique. Table 15 presents a comparison of the number of gate-level faults and the stratum weights obtained from this experiment. Table 16 presents a comparison of stratified RTL coverages and error bounds obtained using equation (4.13) and (4.23) for the stratum weights illustrated in Table 15.

Table 15 Estimation Error in Stratum Weights: Design D3

Module	Use of Technology-dependent Technique						Use of Technology-independent Technique		Final Netlist	
	Netlist1		Netlist2		Netlist3					
	G_m	W_m	G_m	W_m	G_m	W_m	A_m	W_m	G_m	W_m
m_1	630	0.006	680	0.007	572	0.005	652	0.005	602	0.006
m_2	45	0.000	40	0.000	44	0.000	575	0.004	38	0.000
m_3	8052	0.073	7050	0.068	6890	0.061	8643	0.072	7002	0.065
m_4	11384	0.103	11342	0.110	12294	0.109	11328	0.095	11384	0.106
m_5	21012	0.189	18825	0.183	21152	0.188	28016	0.234	21152	0.197
m_6	27024	0.243	22683	0.220	26108	0.232	27392	0.228	22902	0.213
m_7	4280	0.039	3980	0.039	4310	0.038	3584	0.029	4280	0.039
m_8	21220	0.191	19734	0.192	21220	0.189	20535	0.171	21220	0.198
m_9	5010	0.045	6424	0.062	6616	0.059	6653	0.055	6424	0.059
m_{10}	11116	0.100	11116	0.108	12227	0.109	11253	0.094	11116	0.104
m_{11}	504	0.005	504	0.005	504	0.005	695	0.006	504	0.005
m_{12}	739	0.007	640	0.006	612	0.005	143	0.001	666	0.006
Total	111016	1.0	103018	1.0	112549	1.0	119469	1.0	107290	1.0

Legend:

G_m = Number of gate faults in the mth module

A_m = Area of the mth module measured in terms of number of gates

W_m = Stratum weight of the mth module

Table 16 Impact of Error in Stratum Weights on Stratified RTL Fault Coverage and Error Bounds: Design D3

Module	Technology-dependent Stratum Weights						Technology-independent Stratum Weights		Actual Stratum Weights from the Final Netlist	
	Netlist1		Netlist2		Netlist3					
	$\overline{C}$	3σ Bound	$\overline{C}$	3σ Bound	$\overline{C}$	3σ Bound	$\overline{C}$	3σ Bound	C	3σ Bound
T1	55.8	±7.9	55.8	±8.1	55.8	±8.0	52.3	±7.5	55.1	±8.0
T2	20.7	±1.7	19.9	±1.7	19.9	±1.7	19.4	±1.6	19.4	±1.7
T3	59.8	±2.3	59.1	±2.3	59.6	±2.3	59.7	±2.3	59.2	±2.4
T4	51.3	±2.0	51.3	±2.1	51.5	±2.1	47.7	±1.9	50.2	±2.1
T5	58.3	±7.9	58.1	±8.1	58.2	±7.9	54.4	±7.4	57.2	±7.9
T6	70.2	±9.1	69.9	±9.2	70.2	±9.2	68.6	±9.3	69.8	±9.3
T7	75.5	±8.6	75.2	±8.7	75.6	±8.6	73.6	±8.9	74.9	±8.8

Legend:

$\overline{C}$ = Stratified RTL fault coverage calculated using equation (4.13)

C = Gate-level fault coverage

Error bounds are calculated for $\pm 3\sigma$ range using equation (4.23).

It can be observed from Table 15, that the absolute value of G_m and G vary significantly for different gate-level netlists. However, their ratios represented as stratum weight W_m do not vary as much. Also, an observation of the data in Table 16 indicates that the impact of small variations in stratum weights on the stratified RTL fault coverage and error bounds is negligible. None of the parameters used in equations (4.13) and (4.23) to determine the stratified RTL fault coverage and error bounds depend upon absolute values of G_m and G. They use only the ratio, W_m. If parameters in equations (4.13) and (4.23) relied on absolute values of G_m and G, it would be impossible to estimate the gate-level fault coverage within predictable error bounds due to the large possible variations in the values of G_m and G among the different netlists.

6.3 Performance Gain

The RTL fault simulation is expected to offer superior performance compared to the gate-level approach. Experiments were conducted to measure the improvements in the simulation speed. Table 17 shows the comparison of simulation time measured in "seconds" for RTL as well as gate-level fault simulations for several modules and VLSI systems.

Table 17 Performance Comparison

Design	RTL Fault Simulation (seconds)	Gate-level Fault Simulation (seconds)
4-bit counter module	0.3	1.2
Transmit Buffer Module	45.7	181
SDRAM Controller Module	91.2	138.2
DSP Interface Module	40.3	265.7
Frame-Timing-Control ASIC	22.6	24.2
Digital Signal Processor ASIC	2964	5493
	17305	41018

For each test case, an equal number of RTL and gate faults were used during fault simulation for a given set of test patterns. During various experiments, the RTL fault simulation showed 2 to 6 times speed improvement over the gate-level approach. The RTL and gate-level fault simulations were performed using *Verifault-XL™* of the Cadence® Design Systems. The *Verifault-XL™* is a gate-level fault simulator designed to provide optimum performance when used with a gate-level netlist. Its unique ability for behavioral fault propagation also allows one to use it as an RTL fault simulator. The use of *Verifault-XL™* for the comparison of the simulation speeds of RTL and gate-level abstractions may not be adequate since it is optimized for use with gate-level fault-lists while offering support for RTL fault simulation only as a secondary feature. A true comparison of RTL and gate-level fault simulation performances can be obtained if fault-simulators optimized for RTL as well as gate-level fault-models are used. Currently, such an option is not available. Fault simulators designed to work optimally with the proposed RTL fault model may offer large performance improvements. In the meanwhile, it can

be stated that RTL fault simulation is observed to offer 2 to 6 times simulation speed improvement over the gate-level approach.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

In this thesis, a novel procedure that supports RTL fault simulation and generates an estimate of the gate-level fault coverage for a given set of test patterns is proposed. This procedure is based on new RTL fault model, fault-injection algorithm, application of stratified sampling theory, and stratum weight extraction techniques. The proposed RTL fault model and the fault-injection algorithm are derived from an analysis of the properties of the gate-level SSF model and a study of mapping of the RTL constructs onto the gate-level structures during logic synthesis. The proposed fault model and injection algorithm are developed such that the RTL fault-list of a module becomes a representative sample of the collapsed gate-level fault-list. The RTL fault coverage of a module generated on the basis of the proposed fault model tracks gate-level fault coverage within the error bounds predicted by the random sampling technique [AGR90]. An application of the stratified sampling theory is developed to support RTL fault modeling for VLSI systems, that consist of interconnected modules. The RTL fault coverages of all modules in a VLSI system are processed along with their respective stratum weights as per the stratified sampling theory. Several stratum weight extraction techniques are suggested to support this application of the stratified sampling theory to the RTL fault modeling. The stratified RTL fault coverage serves as an estimate of the gate-level fault coverage of the VLSI system within statistical error bounds. The novel procedure proposed in this thesis was verified using several real-life industry-application VLSI systems. The experimental data supports the proposed theory.

Test generation for VLSI circuits has traditionally been performed at the gate level. Although gate-level fault models and test generation algorithms yield high quality tests, their application to modern VLSI systems have become prohibitively expensive in terms of CPU time and memory resources [CHI99]. High-level test generation has recently been the subject of various research activities. The novel procedure proposed in this thesis can be used as an integral part of the high-level test generation system. The test patterns can be generated either using traditional approaches, such as, random test generation and deterministic fault-independent test generation, or state-of-the-art high-level test generation approaches, such as the ones given by

Hayne and Johnson [HAY99], Cho and Armstrong [CHO94], Chiusano et al. [CHI99], Rudnick et al. [RUD98] and Corno et al. [COR97]. The results of test-evaluation using the proposed procedure can provide feedback for further improving the quality of test patterns.

If the circuit is poorly testable, that is generally known only after the gate-level netlist is generated. Possible changes have to be made in the RT-level description, which must then be re-validated and re-synthesized, increasing the design cycle time [COR97]. As established by Thaker et al. [THA99a] with empirical data, the architectural testability properties and the subsequent test weaknesses of a gate-level netlist are derived from the RTL description and remain unchanged by the constraint-driven logic synthesis. The RTL fault simulation using the proposed fault model provides means to identify testability problems at the RT level prior to logic synthesis. The circuits that do not attain high fault coverage even with a large number of test patterns indicate test-related flaws inherent in the design architecture. The undetected RTL faults indicate hard-to-test functional areas of the design. It has been empirically shown by Thaker et al. [THA99a] that various test problems such as,

- observability and/or controllability blockages resulting from the untestable embedded components,
- illegal states of state-machines,
- combinational feedback loops,
- blockage of observability of lower bits of arithmetic components due to truncation hardware,
- redundant logic,
- reconverging signals,
- global redundancies, and/or
- observability/controllability blockages resulting from design modules with poor function dependent testability properties such as error coding and correction algorithm, etc.,

present in the gate-level netlist can be traced back to their causes in the RTL design space. The identification of these test problems early in the design cycle prompts necessary architectural changes prior to logic synthesis, reducing the impact on time-to-market.

Existing fault simulation techniques, which are based on gate-level SSF models, require a large memory and a lot of CPU time. These computing requirements grow at a rate that is

proportional at least to the square of the number of gates in the design [GOE80]. For modern VLSI circuits of larger sizes, this is a prohibitively expensive limitation. The RTL fault simulation approach presented in this thesis holds promise for significantly reducing the performance penalties of the gate-level fault simulation approach. The experiments conducted to compare the speed of RTL and gate-level fault simulations for an equal number of faults, show that the RTL fault simulation runs 2 to 6 times faster than the gate-level fault simulation. However, the commercial fault simulator used for these experiments is designed for optimum performance at the gate level, while supporting RTL fault simulation as only a secondary feature. A true comparison of RTL and gate-level fault simulation performances can be obtained if fault simulators optimized for RTL as well as gate-level fault models were used. Currently, such options are not available. A fault simulator designed to work optimally with the proposed RTL fault model may offer larger performance improvements.

The proposed research meets its stated goals and provides a technique for estimating the gate-level fault coverage of VLSI circuits using RTL fault simulation. The RT level fault simulation performed using the proposed procedure provides significant performance gain over the gate level approach, can support test generation prior to logic synthesis, and offers early detection of testability problems enabling architectural improvements for Design-for-Test (DFT) reasons during the pre-synthesis phase of the VLSI design cycle. The limitation of the proposed procedure is that it requires one to preserve the partitioning of the VLSI systems across RTL module-boundaries in order to be able to use stratum weights reliably. Some of the advanced optimization techniques recommend flattening of the RTL structure/partitions during logic synthesis to maximize the area reduction. This and other similar logic synthesis/optimization techniques that require removal or restructuring of design partitions built into the RTL description as modules, should be avoided if the proposed procedure is to be effectively used.

The secondary contribution of this thesis is in the area of design validation. A superior code coverage metric (see Appendix A) is proposed. This observability-based code coverage metric, Validation Vector Grade (VVG), overcomes several critical limitations of the contemporary software code-coverage metrics used in hardware design validation [THA99b]. The new metric helps develop a comprehensive validation vector-set and thus improves the design quality.

7.2 Future Work

The success of the proposed RTL fault model in estimating gate-level fault coverage within statistical error bounds opens new avenues for further research. Some of the obvious extensions of the proposed fault model are in the area of high-level test generation and testability analysis. The RTL fault-list generated based on the proposed fault model represents a subset of the gate faults. Techniques similar to the one developed by Seth et al. [SET90] to determine the sample size of gate-faults for random and deterministic test generation can be used for high-level test generation with the proposed RTL fault model. Also, an RTL testability measure similar to the one proposed by Seth et al. [SET90] that establishes relationship between probabilistic testability and gate fault coverage can be explored.

The RTL fault model proposed in this thesis is used to estimate the gate-level fault coverage. It is also used to derive a code coverage metric, VVG, for validation. Thus, the proposed fault model provides a common model for test and validation, two aspects of VLSI design methodology perceived to be unrelated until recently. Some of the recent research in the area of high-level test generation attempts at combining software testing based techniques for test sequence generation at a high-level with gate-level techniques for test sequence enhancement [RUD98]. A similar approach using the proposed software coverage metric, VVG, can be developed to guide test generation. Such a test generation technique, combined with the proposed test evaluation technique, will provide a complete solution based on the common RTL fault model.

REFERENCES

[ABR90] Abramovici, M., Breuer, M., and Friedman, A., *Digital Systems Testing and Testable Design*, IEEE Press, New York, NY., 1990.

[AGR81] Agrawal, V.D., “Sampling Techniques for Determining Fault Coverage in LSI Circuits,” *J. Digital Systems*, vol. 5, Sept 1981, pp. 189-202.

[AGR82] Agrawal, V.D., Seth, S.C., and Agrawal, P., “Fault Coverage Requirement in Production Testing of LSI Circuits,” *IEEE Journal of Solid-State Circuits,* vol. SC-17, no. 1, Feb 1982, pp. 57-61.

[AGR88] Agrawal, V.D., and Seth, S.C., *Test Generation for VLSI Chips*, IEEE Computer Society Press, Los Alamitos, CA., 1988.

[AGR90] Agrawal, V.D., and Kato H., “Fault Sampling Revisited,” *IEEE Design & Test of Computers*, vol. 7, August 1990, pp. 32-35.

[ARM92] Armstrong, J.R., Lam, F.S., and Ward, P.C., “Test Generation and Fault Simulation for Behavioral Models,” *Performance and Fault Modeling with VHDL*, Prentice-Hall, Englewood Cliffs, NJ., 1992, pp. 240-303.

[ARM93] Armstrong, J.R., “Hierarchical Test Generation: Where We Are, And Where We Should Be Going,” *Proc. EURO-DAC*, Sept 1993, pp. 434-439.

[BHA94] Bhatia, S. and Jha, N.K., “Genesis: A Behavioral Synthesis System for Hierarchical Testability,” *Proc. European Design and Test Conference*, Feb 1994, pp. 272-276.

[BHA99] Bhatnagar, H., *Advanced ASIC Chip Synthesis*, Kluwer Academic Publishers, Boston, MA., 1999, pp. 2-4.

[CAD97] _________, Cadence® Design Systems, Inc., *Verifault-XL™ User's Guide*, San Jose, CA., 1997.

[CAS75] Case, G.R., "A Statistical Method for Test sequence Evaluation," *Proc. 12th Design Automation Conference*, June 1975, pp. 257-260.

[CHA86] Chang, H.P., Rogers, W.A., and Abraham, J., " Structured Functional Level Test Generation Using Binary Decision Diagrams," *Proc. International Test Conference*, 1986, pp. 97-104.

[CHA88] Chakraborty, T., and Ghosh, S., "On Behavior Fault-Modeling for Combinational Digital Designs," *Proc. International Test Conference*, Sept 1988, pp. 593-600.

[CHE90] Cheng, K.T., and Agrawal, V.D., "An Entropy Measure for the Complexity of Multi-output Boolean Functions," *Proc. 27th Design Automation Conference*, June 1990, pp. 302-305.

[CHE94] Chen, C.H., Karnik, T., and Saab, D.G., "Structural and Behavioral Synthesis for Testability Techniques," *IEEE Transactions on Computer-Aided Design*, vol. 13, no. 6, June 1994, pp. 777-785.

[CHI99] Chiusano, S., Corno, F., and Prinetto, P., "RT-level TPG Exploiting High-Level Synthesis Information," *Proc. 17th IEEE VLSI Test Symposium*, April 1999, pp. 341-346.

[CHO94] Cho, C.H., and Armstrong, J.R., "B-algorithm: A Behavioral Test Generation Algorithm," *Proc. International Test Conference*, Oct 1994, pp. 968-979.

[CLA99] Clarke, Jr., E.M., Grumberg, O., Peled, D.A., *Model Checking*, MIT Press, Cambridge, MA., 1999.

[COC77] Cochran, W.G., *Sampling Techniques*, John Wiley & Sons, Inc., New York, NY., 1977.

[COR97] Corno, F., Prinetto, P., and Reorda, M.S., "Testability Analysis and ATPG on Behavioral RT-level VHDL," *Proc. International Test Conference*, Nov 1997, pp. 753-759.

[DAE91] Daehn, W., "Fault Simulation Using Small Fault Samples," *J. Electronic Testing: Theory and Applications (JETTA)*, vol. 2, no. 2, June 1991, pp. 191-203.

[DAS93] Das, D., Seth, S.C., and Agrawal, V.D., "Accurate Computation of Field Reject Ratio Based on Fault," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 1, no. 4, Dec 1993, pp. 537-545.

[DAV82] Davis, B., *Economics of Automatic Testing*, McGraw Hill, 1982.

[DEV94] Devadas, S., Ghosh, A., and Keutzer, K., *Logic Synthesis*, McGraw-Hill, Inc., Washington, D.C., 1994, pp. 6-185.

[DEY98] Dey, S., Raghunathan, A., and Roy R.K., "Considering Testability During High Level Design," *Proc. ASP-DAC*, Feb 1998, pp. 205-210.

[DRA98] Drako, D., and Cohen, P., "HDL Verification Coverage," *Integrated System Design*, June 1998, pp. 46-52.

[ELD59] Eldred, R.D., "Test Routines Based on Symbolic Logic Statements," *J. ACM*, vol. 6, no. 1, Jan 1959, pp. 33-36.

[FAL98] Fallah, F., Devadas, S., and Keutzer, K., "OCCOM: Efficient Computation of Observability-based Metrics for Functional Verification," *Proc. 35th Annual Design Automation Conference*, June 1998, pp.152-157.

[GHO88] Ghosh, S., "Behavioral-level Fault Simulation," *IEEE Design & Test of Computers*, vol. 5, no. 3, June 1988, pp. 31-42.

[GHO96] Ghosh, I., Raghnathan, A., and Jha, N.K., "A Design for Testability Technique for RTL Circuits Using Control/Data Flow Extraction," *Proc. IEEE International Conference on Computer-Aided Design*, Nov 1996, pp. 329-336.

[GOE80] Goel, P., "Test Generation Costs Analysis and Projections," *Proc. 17th Design Automation Conference*, June 1980, pp. 77-84.

[GOL79] Goldstein, L.H., "Controllability/Observability Analysis of Digital Circuit," *IEEE Trans. on Circuits and Systems*, vol. CAS-26, no. 9, Sept 1979, pp. 685-693.

[GON98] Goncalves, F.M., and Teixeira, J.P., "Sampling Techniques of Non-Equally Probable Faults in VLSI Systems," *Proc. 16th IEEE VLSI Test Symp.*, 19998, pp. 283-288.

[HAY99] Hayne, R.J., and Johnson, B.W., "Behavioral Fault Modeling in a VHDL Synthesis Environment," *Proc. 17th VLSI Test Symposium*, April 1999, pp. 333-340.

[HSU98] Hsu, F.F., and Patel, J.H., "High-level Controllability and Observability Analysis for Test Synthesis," *J. Electronic Testing: Theory and Applications (JETTA)*, vol. 13, no. 2, Oct 1998, pp. 93-103.

[JAI85] Jain, S., and Agrawal, V.D., "Statistical Fault Analysis," *IEEE Design & Test of Computers*, vol. 2, no. 1, Feb 1985, pp. 38-44.

[JAL91] Jalote, P., *An Integrated Approach to Software Engineering*, Springer-Verilog, New York, NY., 1991.

[KAN96] Kantrowitz, M., and Noack, L.M., "I'm done simulating: Now What? Verification Coverage Analysis and Correctness of Checking of the DEC Chip 21164 Alpha Microprocessors," *Proc. 33rd Annual Design Automation Conference*, June 1996, pp. 325-330.

[KHO92] Khoche, A., Sherlekar, S.D., Venkateshesh, G., and Venkateswaran, R., "A Behavioral Fault Simulator for Ideal," *IEEE Design & Test of Computers*, vol. 9, no. 4, Dec 1992, pp. 14-21.

[KUR94] Kurshan, R.P., *Computer Aided Verification of Coordinating Processes*, Princeton University Press, Princeton, NJ., 1994.

[LEE92] Lee, T.C., Wolf, W., Jha, N.K., "Behavioral Synthesis for Easy Testability in Data Path Scheduling," *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 1992, pp. 616-619.

[LEE93] Lee, T.C., Jha, N.K., and Wolf, N.H., "Behavioral Synthesis of Highly Testable Data Paths Under Non-scan and Partial scan Environments," *Proc. Design Automation Conference*, June 1993, pp. 292-297.

[LEE94] Lee, J., and Patel, J.H., "Architectural Level Test Generation for Microprocessors," *IEEE Trans. on Computer-Aided Design*, vol. 13, no. 10, Oct 1994, pp. 1288-1300.

[LEV82] Levendel, Y.H., and Menon, P.R., "Test Generation Algorithms for Computer Hardware Description Languages," *IEEE Trans. on Computers*, vol. C-31, July 1982, pp. 577-589.

[MAK99] Makris, Y., and Orailoglu, A., "RTL Test Justification and Propagation Analysis for Modular Designs," *J. Electronic Testing: Theory and Applications (JETTA)*, vol. 13, no. 2, Oct 1999, pp. 105-120.

[MAO96] Mao, W., and Gulati, R., "Improving Gate Level Fault Coverage by RTL Fault Grading," *Proc. International Test Conference*, Oct 1996, pp. 150-159.

[MCN89] McNamer, M.G., Roy, S.C., and Nagle, H.T., "Statistical Fault Sampling," *IEEE Transaction on Industrial Electronics*, May 1989, pp. 141-150.

[MIC86] Miczo, A., *Digital Logic Testing and Simulation*, John Wiley & Sons, New York, NY., 1986.

[MUR67] Murthy, M.N., *Sampling Theory and Methods*, Statistical Publishing Society, Calcutta, India, 1967.

[PAP98] Papachristou, C.A., Bklashav, M., and Lai, K., "High-Level Test Synthesis for Behavioral and Structural Designs," *J. Electronic Testing: Theory and Applications (JETTA)*, vol. 13, no. 2, Oct 1998, pp. 167-188.

[PIP77] Pippenger, N., "Information Theory and the Complexity of Boolean Functions," *Mathematical Systems Theory*, vol.10, 1997, pp.129-167.

[POT95] Potkonjak, M., Dey, S., and Roy, R., "Behavioral Synthesis of Area-Efficient Testable Designs Using Interaction Between Hardware Sharing and Partial Scan," *IEEE Trans. on Computer-Aided Design*, vol. 14, no. 9, Sept 1995, pp. 1141-1154.

[RAV97] Ravi, S., Ghosh, I., Roy, R.K., and Dey, S., "Controller Resynthesis for Testability Enhancement of RTL Controller/Data Path Circuits," *Proc. International Conference on VLSI Design*, Jan 1997, pp. 193-198.

[RUD98] Rudnick, E.M., Vietti, R., Ellis, A., Corno, F., Prinetto, P., and Reorda, M.S., "Fast Sequential Circuit Test Generation Using High-Level and Gate-Level Techniques," *Proc. IEEE European Design Automation and Test Conference*, Feb 1998.

[SAN96] Sanchez, P., and Hidalgo, I., "System Fault Simulation," *Proc. International Test Conference*, Oct 1996, pp. 732-740.

[SET90] Seth, S.C., Agrawal, V.D., and Farhat, H., "A Statistical Theory of Digital Circuit," *IEEE Trans. on Computers*, vol. 39, no. 4, April 1990, pp. 582-586.

[STE58] Stephan, F., and McCarthy P., *Sampling Opinions: An Analysis of Survey Procedure*, John Wiley & Sons, Inc., New York, NY., 1958.

[STE93] Sternheim, E., Singh, R., Madhavan, R., and Trivedi, Y., *Digital Design and Synthesis with Verilog HDL*, Automata Publishing Company, San Jose, CA., 1993.

[STU84] Stuart, A., *The Ideas of Sampling*, Charles Griffin and Company, Ltd., High Wycombe, Great Britain, 1984.

[THA80] Thatte, S., and Abraham, J., "Test Generation for Microprocessors," *IEEE Transactions on Computers*, vol. C-29, June 1980, pp. 429-441.

[THA99a] Thaker, P.A., Zaghloul, M.E., and Amin, M.B., "Study of Correlation of Testability Aspects of RTL Description and Resulting Structural Implementations," *Proc. 12th International Conference on VLSI Design*, Jan 1999, pp. 256-259.

[THA99b] Thaker, P.A., Agrawal, V.D., and Zaghloul, M.E., "Validation Vector Grade (VVG): A New Coverage Metric for Validation and Test, " *Proc. 17th IEEE VLSI Test Symposium*, April 1999, pp. 182-188.

[VIS93] Vishakantiah, P., Abraham, J.A., and Saab, D.G., "CHEETA: Composition of Hierarchical Sequential Tests Using ATKET," *Proc. International Test Conference*, Oct 1993, pp. 606-615.

[WAN95] Wang, T.H., and Tan, C.G., "Practical Code Coverage for Verilog*," Proc. 4th International Verilog HDL Conference*, March 1995, pp. 99-104.

[WAR90] Ward, P.C., and Armstrong, J.R., "Behavioral Fault Simulation in VHDL," *Proc. 27th Design Automation Conference*, June 1990, pp. 587-593.

[WIL81] Williams, T.W., and Brown, N.C., "Defect Level as a Function of Fault Coverage," *IEEE Trans. on Computers*, vol. C-30, no.12, Dec 1981, pp. 987-988.

APPENDIX A

VALIDATION VECTOR GRADE

A NEW CODE COVREAGE METRIC: VALIDATION VECTOR GRADE

Abstract

Contemporary code coverage metrics used in the top-down VLSI design methodology are based on statement, branch, toggle and condition coverage of the HDL code obtained by simulating validation vectors. These measures allow the designer to find sections of the HDL code not executed during validation. Feedback from the code coverage analysis helps generate additional vectors to exercise previously unexercised functionality. However, these controllability-based software code coverage metrics are inadequate for hardware validation effort. Observability requirements are very critical for hardware validation effort due to the fact that the activation of a statement with a design-bug does not guarantee that its effect will be observed at outputs or designated internal nodes where responses are analyzed. Thus, controllability-based code coverage metrics measure only partial effectiveness of the stimuli. They only measure if the input stimuli activate particular function but they fail to measure if the activation of an RTL construct which may or may not have design errors is observed at the designated internal nodes or output ports. These code coverage metrics have several other limitations such as redundancy and ambiguity in reported results, prohibitively expensive run-time overheads and the inability to identify meaningless tests that may falsely generate high coverage. A superior observability-based code coverage metric, *Validation Vector Grade (VVG)*, is proposed. VVG overcomes the limitations of software code coverage metrics and helps improve the quality of hardware design validation.

1.0 Introduction

Advances in EDA technology have increased the size and the complexity of VLSI circuits making it challenging to develop comprehensive validation procedures. An HDL-based design approach has reduced the actual design time, allowing designers to spend relatively more time on validation. As outlined by Jalote [JAL91], the code coverage analysis technique has been popular in the software arena. With the use of hardware description languages, the code coverage technique of software verification serves as a feedback mechanism for hardware design engineers

in developing comprehensive validation suites. Various practical approaches [KAN96] and coverage analysis methods [DRA98, WAN95] have been proposed in recent years. As described by Drako and Cohen [DRA98], the software code coverage analysis technique offers five quality metric numbers: statement execution coverage, condition coverage (also known as expression coverage), branch coverage (also known as decision coverage), toggle coverage and path coverage. If any coverage is found to be low, an analysis of the unused code helps determine the necessity of additional tests.

The controllability-based software code coverage metrics are inadequate for hardware validation efforts since they do not address observability requirements. The fact that a statement with a design-bug has been activated by input stimuli does not mean that the observed response at the outputs or designated internal nodes captures the incorrectness of the functionality. As explained by Fallah et al. [FAL98], the incorrect behavior due to a design error may have been executed by stimuli but may not have been propagated to designated observation nodes for it to be caught during validation. Contemporary software code coverage metrics report the measurement of functional features stimulated by validation vectors but overlook whether incorrect functional behavior has propagated to designated observation node or not. Also, the controllability-based code coverage technique has several other limitations such as redundancy and ambiguity in reported results, prohibitively expensive run-time overheads due to its coverage tracking mechanism that insets “callbacks”, “hooks” or “probes” in the code, and the inability to identify tests that may falsely generate high coverage without properly exercising the function. Since the contemporary code coverage technique reports the overall quality of vectors using five metrics, the results are often ambiguous since one or more of the metrics may report 100% coverage with the remaining metrics reporting coverage to be less than 10%. These numbers do not clearly indicate the amount of effort needed to improve the quality of validation vectors. Only an in-depth analysis can reveal if there is a need for additional vectors. Such an analysis can also sometimes reveal that conditions not covered are functionally redundant and do not require any effort for improvement. The contemporary code coverage analysis technique inserts “callbacks” into the code that count occurrences of particular conditions such as transitions, statement executions, etc., during simulations. Since designs are getting larger, the performance overhead of this technique on simulators has increased exponentially. Also, the code coverage technique does not give a proper indication of whether the logic is being functionally used during

simulation. One can write meaningless or poor tests and still achieve a very high coverage. For example, an *enable* signal of a counter will be reported covered even if it toggled while the counter is in reset state (due to active reset signal) since the effect of the *enable* signal is not observed while measuring code coverage.

Fallah et al. [FAL98] proposed an observability-based code coverage analysis method, OCCOM, as an alternative to the five-metrics based software code coverage technique. The proposed technique relied on an approximate D-calculus in a graph analysis program that computes the observability of internal variables. The algorithm of tag propagation presented by Fallah [FAL98] focuses on the reduction of computing effort at the cost of accuracy. The metric proposed in this thesis focuses on the accuracy of results at a slightly higher performance cost compared to OCCOM. The performance penalty of the proposed approach is many folds lower than the contemporary approach that uses software code coverage metrics.

2.0 Validation Vector Grade

As described by Thaker et al. [THA99b], the Validation Vector Grade (VVG) extends the concept of toggle coverage of the software code coverage technique by adding the observability dimension to it, while dropping the rest of the coverage parameters. Instead of the five quality-metrics used by the contemporary code coverage technique, VVG uses only one metric to report more information than reported by the five metrics. RTL variables that get used more than once in executable statements or instantiations of lower level modules of the design-hierarchy are considered to have fan-out. VVG requires fan-out of each RTL variable to be treated as a unique variable. Under the VVG approach, an RTL variable is reported covered only if it can be controlled from primary inputs and observed at a primary output or a designated internal node using a technique that is similar to fault grading. Fault grading is usually performed on gate-level netlist. However, VVG is generated for a given set of validation vectors by fault grading RTL variables using the stuck-at fault model and the RTL code as a design description. VVG is characterized as the ratio of RTL variables reported covered for a given set of validation vectors to the total number of RTL variables in a design. Faults are injected on RTL variables using the RTL fault model and the fault-injection algorithm described in Chapter 3. RTL fault-grading methodology is described in Chapter 3 as well.

3.0 Experimental Results

RTL designs along with their validation suites were used to compare the effectiveness of VVG and the contemporary code coverage metrics. D1, D2, D3 and D4 are RTL designs with gate sizes ranging from 56 gates up to 4100 gates. TransEDA®'s *Verisure*™ was used to measure statement, branch, condition and toggle coverage for all the designs for a given set of validation vectors. The same validation vector sets were used to measure VVG. Results are detailed in Table 18.

Table 18 Software Code-Coverage Metrics vs. VVG

Design	Software Code-Coverage Metrics (%)					New Metric VVG (%)
	Statement Coverage	Branch Coverage	Sub-condition Coverage		Toggle Coverage	
			Basic	Multiple		
D1	100.0	100.0	100.0	62.5	96.8	96.6
D2	99.1	97.6	100.0	82.5	92.9	87.3
D3	100.0	100.0	100.0	42.2	100.0	89.2
D4	100.0	100.0	90.0	83.3	91.1	82.6

4.0 Discussion

Fallah et al. [FAL98] have identified the need for an observability-based code coverage metric for the functional verification of a VLSI design. Their technique relies on an approximate D-calculus in a graph analysis program, OCCOM, which computes the observability of internal variables. Their algorithm for tag propagation focuses on the reduction of computing effort at the cost of accuracy. The method proposed in this thesis focuses on accuracy with a slightly higher computing cost (1.5 to 4 times for OCCOM vs. 4 to 7 times for VVG, over HDL simulation). As in the case of the software code coverage technique and OCCOM, the VVG approach only provides feedback about the quality of validation vectors. All approaches still require the

designer to validate output responses of the design module. A comparison of the VVG approach with the software code coverage technique follows:

(1) Since the software code coverage technique reports the overall quality of validation vectors using five coverage metrics, the results are often ambiguous as shown in Table 18. For example, for design D3, the statement and toggle coverage are each 100% with very low multiple sub-condition coverage (42.24%). These numbers do not clearly indicate the amount of effort needed to improve the quality of validation vectors to attain complete coverage. Only an in-depth analysis can reveal if there is a need for additional validation vectors. Such an analysis sometimes reveals that conditions not covered are functionally redundant and not required to be covered. It is desirable that the code coverage metrics automatically account for such conditions and exclude them from being reported.

The approach presented in this thesis uses a single code coverage parameter called *Validation Vector Grade (VVG)* to report the quality of validation vectors unambiguously. The VVG-approach excludes many functionally redundant conditions from the computation of the metric and therefore does not flag them. This reduces the complexity and effort required to identify uncovered cases. For example, in the Verilog HDL construct, "*if (a && b) statement1; else statement2;*" the condition coverage of the statement "*if (a && b)*" will be reported fully covered by the contemporary software code coverage metrics only if all four combinations of logic levels on variables *a* and *b* are exercised. However, for complete functional verification, only three conditions are necessary, excluding the need for the vector that applies logic-level zero on both variables simultaneously. Since the proposed code coverage metric is generated using the process of fault simulation, full coverage will be attained with three vectors. It will not report lower coverage for the fourth missing condition in which both variables simultaneously have logic value of zero. Accounting for unnecessary redundant conditions and not flagging them as missing coverage, helps reduce the effort and the complexity required for analysis to identify uncovered conditions and allows the designer to spend more time on valid cases of missing coverage.

(2) The contemporary software-based code coverage technique consumes memory and imposes simulation run-time overheads in a prohibitively expensive fashion. It tracks various types of coverage during simulation by adding callbacks into RTL code as described by Wang and Tan [WAN95]. These callbacks (also called "hooks" or "probes") count occurrences of particular situations such as transitions, conditions, statement executions, etc., throughout the simulation. Since designs are getting larger, the performance overhead on simulators has increased exponentially. Also, the current technique does not provide a method of excluding situations during the current simulation run that are reported covered in previous runs.

    Since the VVG approach uses stuck-at fault models (supported in Cadence®'s *Verifault-XL™*) for every RTL variable, it is free from the performance penalty of the software-based code coverage approach. As soon as a variable is detected, it is dropped from the list of variables. Therefore, as simulation progresses, the performance overhead decreases. However, a method is provided to keep variables on the active list after they are covered if the designer chooses to do so. Also, one may use the random sampling approach for early estimation of vector quality during the validation effort.

(3) Even though a block of code is fully exercised, the validation effort may be meaningless unless the results of code execution propagate through the design. The controllability-based software code coverage technique cannot report such problems.

    Since the VVG approach tracks the observability of all internal RTL variables to primary outputs or designated internal nodes, the problem mentioned above can be detected effortlessly. An RTL variable is not reported as covered until it is activated by the input stimuli and the effect of activation observed at the designated observation node.

(4) As designs get larger and complex, self-checking validation environments are gaining popularity. The self-checking validation suites contain both input stimuli and the expected responses. Self-checking validation environments monitor outputs or selected internal nodes during the simulation and report any deviation from the expected response. The software code coverage technique reports how many times an RTL variable toggled without assessing its impact on the primary outputs or selected internal nodes.

In the VVG approach, a variable is reported as covered only if it can be controlled from primary inputs and its logic transition impacts primary outputs or the selected internal node (strobe probe). It thus provides a method for a complete assessment of the quality of the self-checking validation environment.

(5) The software code coverage technique does not give a proper indication of whether the logic is being functionally used during simulation. One can write meaningless or poor tests and still achieve a very high coverage. For example, an *enable* signal of a counter will be reported as covered even if it toggled while the counter was in reset state due to active reset signal.

Since the VVG-approach draws a parallel from a gate-level fault simulation, it checks for the functional validity of a test to a certain degree. In the above example of a counter, an *enable* signal is reported covered only if it is properly exercised in a functional manner. If the *enable* signal toggles while a counter is in reset state, it will not be covered since it can not be observed at counter outputs.

5.0 Conclusion

The proposed Validation Vector Grade approach overcomes several limitations of the software code coverage technique without imposing new penalties. This observability-based code coverage metric is adequate for hardware design validation. Since it measures the quality of validation vectors not just on the basis of whether a given functionality was stimulated or not, but also on the basis of whether the effect of execution of a function was observed at designated observation nodes or not. Sections of code (or function) reported not covered by this metric need evaluation and additional vectors can be written to improve the quality of the validation suite. Thus, a common RTL model and grading methodology for the validation and test aspect of VLSI design are proposed in this thesis.

APPENDIX B

C++ CODE FOR FAULT-INJECTION ALGORITHM

C++ program contains more than 2500 lines of code. Some of the important segments of the code are presented here.

```
/*********************************
 PROGRAM:    Parser - puts verifaults on input, output and intermediate signals
 FILENAME:   parse.c
Hughes Network Systems
*********************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <ctype.h>
#include <time.h>
#define MAJOR_VERSION  4
#define MINOR_VERSION  01
/* Parser specific declarations */
#define MAX_VARNAME            32767  /* Maximum size of Variable Name */
#define MAX_STRLEN             32767  /* Maximum length of a string */
#define WIRE                   0x01   /* Size of one byte, as a char but just number 1 */
#define REGISTER               0x02   /* number 2... */
#define INPUT                  0x01
#define OUTPUT                 0x02
#define INTERMEDIATE           0x03
#define VAR_NEW                0x01
#define VAR_OLD                0x02
#define ORD_BEFORE             0x01
#define ORD_AFTER                     0x02
#define IN_ALWAYS                     0x01
#define AL_NONE                0x00
#define AL_NEGEDGE             0x01
```

```
#define AL_POSEDGE        0x02
#define AL_AND            0x01
#define AL_OR             0x02
#define INTERMEDIATE_COMMENT    “\nIntermediate variable(s) commented:\n"
#define VAR_DEF           0x01
#define CODE_PARSE        0x02

/* Error codes */
#define NO_PARAMETER      0x81 /* Not enough parameters to run correctly (no input
file indicated)*/
#define GENERAL_ERROR     0x01 /* memory or running error */
#define NO_ERROR          0x00 /* main returns zero if succesful execution */

/* Help string */
#define HELP_STRING  "HNS Parser v%d.%d final release.\nUsage: %s [-switch|filename
[output] [-D<definition>]]\n\t-h\t\thelp (this screen)\n\tfilename\tinput file
name\n\toutput\t\toutput file name (filename_out.extension by default)\n\t-D\t\tprefix before
<definition>\n\t<definition>\tdefinition to use with:\n\t\t\t'ifdef...`elseif...`else...`endif
construct.\n\t\t\tAny number of definitions allowed.\n\t\t\tEveryone of them must be prefixed
with -D\nExample: %s test.v test_out.v -DTEST1 -DTEST2\n"

#define EXCLUDED_STRING "\n\nHNS Parser v%d.%d will NOT support the
following:\n\tsyntax errors\t\tFiles that contain syntax errors will\n\t\t\t\tnot be processed
correctly.\n\tbidirectional signals\r\t\t\t\tFiles that contain bidirectional signals\n\t\t\t\twill not be
processed correctly, parser\n\t\t\t\tmight crash.\n\tnet,nand and buf\r\t\t\t\tFiles that contain
commands net or nand or\n\t\t\t\tbuf will not be processed correctly, the\n\t\t\t\tparser might
crash.\n\t2'b xxxx or 4'h xxxx\r\t\t\t\tFiles that have a space between type\n\t\t\t\tand value will
not be processed correctly,\n\t\t\t\tbecause of the difficulty to distinguish\n\t\t\t\tbetween single
bit undeclared wires and \n\t\t\t\txxxxx following 'b, 'd, or 'h.  The correct\n\t\t\t\tformat is
2b'xxxx without the space.  However,\n\t\t\t\t4'b 001 and anything with digits only
is\n\t\t\t\tacceptable even with a space.\n\tno module(funct. only)\r\t\t\t\tFiles that contain no
```

```
module, only functions\n\t\t\t\tor some other code without module declaration\n\t\t\t\twill not be
parsed. If a file is to be\n\t\t\t\tparsed correctly, it must contain
\"module\n\t\t\t\tmodule_name(input_signals, output_signals);\".\n\tMacro language\r\t\t\t\tSome
extentive Macro language might be\n\t\t\t\tprocessed incorrectly.\n\tmultiple
modules\r\t\t\t\tWhen a file contain multiple modules. First\n\t\t\t\tmodule will be parsed, the rest
will remain\n\t\t\t\tunchanged.\n\tmultiple instuntiation\r\t\t\t\tWhen a file contain multiple
instuntiation, and\n\t\t\t\tsignals are wires used to connect them, one\n\t\t\t\tneed to make the
necessary corrections in the\n\t\t\t\tparsed code for signal direction.\n\twire a = b +
c;\r\t\t\t\tParser does not support this style. Instead\n\t\t\t\tuse:\r\t\t\t\t\t\twire a;\n\t\t\t\t\t\tassign a
= b + c;\n\tparameters->string\r\t\t\t\tParser does not handle use of parameters\n\t\t\t\tfor string
substitution. For instance:\n\t\t\t\t\twire x;\n\t\t\t\t\tparameter string_substitution =
x;\n\t\t\t\tWhen in the code, parameter is used instead of\n\t\t\t\tthe signal, that signal will not
receive a\n\t\t\t\tverifault because parameter is used instead of\n\t\t\t\ta string name.\n\tinclude
files\r\t\t\t\tRTL code file should contain all the\n\t\t\t\tdeclarations and definitions of the
variables.\n\t\t\t\tStyle in which \"include\" files are included in\n\t\t\t\tRTL code with vital
parameters, string\n\t\t\t\tsubstitution or variable declaration\n\t\t\t\tinformation will not be
processed correctly.\n\ttwo-dimensional array\r\t\t\t\tAlthough supported, use of two
dimensional\n\t\t\t\tarray is discouraged.\n"
/* OS Dependents*/
#ifdef WIN32
#define NECESSARY_NUMBER    2    /* Windows count the executable name as 1 */
#define DELIMITER           '\\'    /*  Windows use c:\..... so delimeter is "\" */
#else
#define NECESSARY_NUMBER    2        /* Unix does not count executable name as one
*/
#define DELIMITER            '/'    /*  Unix uses //user/.... so delimeter is "/" */
#endif

/* PARAMETERS List definition */
typedef struct _PARAMETERS
{
```

```
	char name[MAX_VARNAME];
	long value;
	struct _PARAMETERS *next;  /* ptr. to next element in the list */
} PARAMETERS;

/* VARS List definition */
typedef struct _VARS
{
char name[MAX_VARNAME]; /* either original name or name_verifaultX (number of verifault) */
char type;            /* WIRE (0x01)  or REGISTER (0x02)  */
char dir;             /* Direction: INPUT, OUTPUT or INTERMEDIATE */
long low,high;        /* Bit value: [low:high] */
long verifault;  /*if original var.= counter for number of vfaults total; or number of a vfault*/
char howold;          /* VAR_NEW (1) verifault var or VAR_OLD (2) original variable  */
char *where;          /* ptr. to the location of the variable in the file */
char how;/* if signal found in conect fun. then 1 and if it's not used anywhere else becomes 2else 0 */
int  order;    /* For Bits: ORD_BEFORE (reg [low:high] S;)  or ORD_AFTER (reg S[low:high];) */
int  inalways;                  /* 0 if variable not in always @(...), 1 otherwise */
int  al_prefix;
int  al_postfix;
struct _VARS *next;    /* ptr to next element in the list */
} VARS;
typedef struct _FUNCTIONS
{	char name[MAX_VARNAME];
	struct _FUNCTIONS *next;
} FUNCTIONS;
typedef struct _EXTERNALS
{
```

```
        char name[MAX_VARNAME];
        struct _EXTERNALS *next;
} EXTERNALS;

/* Global storage */
char *pfile=NULL; /* will be input file in memmory */
long pindex=0;    /* current location in that file */
long psize=0;     /* total size the input file occupies in memmory */
char state=VAR_DEF;
FILE *fout;       /* declaring output file */
VARS *var_list=NULL,*var_end=NULL; /* var_end and parameter_end will last nodes the list
*/
PARAMETERS *parameter_list=NULL,*parameter_end=NULL;
FUNCTIONS *functions_list=NULL,*functions_end=NULL;
EXTERNALS *externals_list=NULL,*externals_end=NULL;
int IN_IFDEF=0;

/* function prototypes */
void help(char *exename, int exlude_list); /* outputs a message with run the parser on the screen
*/
void error_terminate(char *pre_message,char *error_message); /* outputs an on the screen */
long filesize(FILE *f); /* returns total size of a single file */
char *parse_line(char *tmp); /*gets a line from the file,line ends with ';'or end of file or end of
line*/
long get_declared(void); /* analyzes first part of the always or endmodule statement */
char *parse_spaces(char *line); /* will search for alpha_numeric character, which is letters,
numbers and _, everything else will be skipped */
char *_parse_spaces(char *line); /* same as one above, except it does not skip symbols only
comments and spaces */
char *get_word(char *line,char *v); /* gets a single: alpha_numeric, or _ or ' is allowed for word
*/
```

```
char *get_uword(char *line,char *v); /* NOT USED */
int get_range(char **_line, char *low, char *high);
/* analyzes bits, if a single bit then that bit is copied to both high &low */
long lookup_range(char *name);
/* checks whether the value in [] is completely digits or found in parameters */
VARS *find_var(char *name);  /* searches through the variable list */
PARAMETERS *find_parameter(char *name); /* searches throuh parameter list */
VARS *add_var(char *name, char type, char dir, long verifault, int range, long low, long high,
char howold);/*adds a new elem. into the variable list */
void sort_list(int how_sort);
/* sorts variable list by name (how_sort=2) or by location, which one found in program first (1)
*/
void write_out(long); /* prints to outputs file all connect, always, assigns and buffers */
int is_alnum(char c);
void preprocess(void);
int get_function(char *cword, char **_cline);
void get_fan_out(char **_line);
void add_external(char *name);
int main(int argc, char *argv[])
{
        FILE *fin;  /* input file */
        char outfname[256]; /* name of output file */
        char tmp[MAX_STRLEN],*t; /* temporary variables */
        long fr;
        long pcur;
        char tmp_date[255],tmp_time[255];
        /* 1st if:
******* if only name of the executable is entered or -h will on how to use the parser * ** 2nd if:
******* no output file was specified, the output file will be original name .out eg. gmsk.v -->
gmsk.v.out
        ** 3rd else:
```

```
****** copies second argument the user inputed into the name of an output file ********/
	if(argc<NECESSARY_NUMBER)
help(strrchr(argv[0],DELIMITER)?strrchr(argv[0],DELIMITER)+1:argv[0], 0);
	elseif(!strcmp(argv[1],"-h"))
help(strrchr(argv[0],DELIMITER)?strrchr(argv[0],DELIMITER)+1:argv[0],1);
	if(argc<NECESSARY_NUMBER+1)
	{
		fr=2;
		strcpy(tmp,argv[1]);
		t=strchr(tmp,'.');
		*t='\0';
		sprintf(outfname,"%s_out.%s",tmp,t+1);
	}
	else
	{
		if(strncmp(argv[2],"-D",2))
		{
			fr=3;
			strcpy(outfname,argv[2]);
		}
		else
		{
			fr=2;
			sprintf(outfname,"%s.out",argv[1]);
		}
}
	for(;fr<argc-NECESSARY_NUMBER+2;fr++)
	{
		if(!strncmp(argv[fr],"-D",2)) add_external(argv[fr]+2);
	}
```

```
//sprintf(prefname,"%s.i",argv[1]);
/* opens input file for reading only, if cannot be found outputs error message */
if((fin=fopen(argv[1],"r"))==NULL) error_terminate("cannot open input file",strerror(errno));
/* 1st determines filesize, then one more is needed to assign 0 and three more just in case
the size of file +4 times size of character, which is one byte
**** 2nd the calloc returns a pointer to the allocated memory of type void, it is easier to
work with characters; hence we turn it to character ptr
***  3rd if for some reason unable to get the memory needed, error message */
if((pfile=(char*)calloc((psize=filesize(fin))+4,sizeof(char)))==NULL)error_terminate("ca
nnot allocate memory",strerror(errno));
fr=fread(pfile,sizeof(char),psize+4,fin);
if(fr<0) error_terminate("cannot read whole file",strerror(errno));
fclose(fin);
/* to open the output file, if for some reason is unable to do this output error message */
/* more over the file is opened for writting, if a file with certain name exists, overwrites
*/
//if((fout=fopen(prefname,"w"))==NULL) error_terminate("cannot open preprocessor
output file",strerror(errno));
printf("preprocessing...");
preprocess();
printf(" done.\n");
//fprintf(fout,"%s",pfile);
//fclose (fout);
if((fout=fopen(outfname,"w"))==NULL) error_terminate("cannot open output
file",strerror(errno));
//fprintf(fout,"/*Processed: %s, %s*/\n",_strdate(tmp_date),_strtime(tmp_time));
while(1)
{
    parse_line(tmp); /* get a line (for our purpose it is until ;) */
    t=tmp;
    fprintf(fout,"%s",tmp);
```

```
/* outputs into file everything including first ; which should appear after module */
                for(;*t;t++)
                {
                        if(!*t) continue;
                        while(!strncmp(t,"/*",2) || !strncmp(t,"//",2))
                        {
                                if(!strncmp(t,"/*",2)) for(;*t && strncmp(t,"*/",2);t++);
                                if(!*t) break;
                                if(!strncmp(t,"//",2)) for(;*t && *t!='\n';t++);
                                if(!*t) break;
                        }
                        if(!strncmp(t,"module",6)) break;
                }
                if(!strncmp(t,"module",6)) break;
        }/* stops when equal to module */
        /* reads until first always or assign */
        /* pcur is pointer to the current char that we are analyzing */
        if((pcur=get_declared())==0) error_terminate("cannot get declarations, file",argv[1]);
        /* sorts list by ptr. where the signals are found within file, which first and etc...*/
        sort_list(1);
        write_out(pcur);
/* analyzes plus outputs assign, always plus connected commponents plus buffers */
        free(pfile);
        fclose(fout);
/* important in emergency cases so that everything analyzed thus far out into the output file */
        return(NO_ERROR);
}
void add_external(char *name)
{

if(!externals_list)externals_list=externals_end=(EXTERNALS*)malloc(sizeof(EXTERNALS));
```

```
  else externals_end=externals_end->next=(EXTERNALS *)malloc(sizeof(EXTERNALS));
  strcpy(externals_end->name,name);
        externals_end->next=NULL;
}
typedef struct _def_stat
{
        char name[80];
        char value[80];
        struct _def_stat *next;
} def_stat;
def_stat *def_list=NULL,*def_end=NULL;
def_stat *preprocess_find_def(char *name)
{
        def_stat *t;
        if(!def_list) return(NULL);
        for(t=def_list;t;t=t->next) if(!strcmp(t->name,name)) return(t);
        return(NULL);
}
def_stat *preprocess_add_def(char *name, char *value)
{
        if(!def_list) def_list=def_end=(def_stat *)malloc(sizeof(def_stat));
        else def_end=def_end->next=(def_stat *)malloc(sizeof(def_stat));
        strcpy(def_end->name,name);
        strcpy(def_end->value,value);
        def_end->next=NULL;
        return(def_end);
}
void preprocess_skip_spaces(void)
{
                while(isspace(pfile[pindex])) pindex++;
        }
```

```
char *preprocess_get_word(char *cword)
{
	int i;
	if(!isalpha(pfile[pindex]) && pfile[pindex]!='_') return(NULL);
	for(i=0;is_alnum(pfile[pindex]);i++,pindex++) cword[i]=pfile[pindex];
	cword[i]=0;
	return(cword);
}
char *preprocess_get_value(char *cword)
{
	long i;
	char *p,*pc1,*pc2,*pc;
if(!(p=strstr(&pfile[pindex],"\n"))) return(NULL);
	pc1=strstr(&pfile[pindex],"//");
	pc2=strstr(&pfile[pindex],"/*");
	pc=pc1>pc2?(pc2>NULL?pc2:pc1):(pc1>NULL?pc1:pc2);
	i=((p>pc && pc>NULL)?pc:p)-&pfile[pindex];
	strncpy(cword,&pfile[pindex],i);
	cword[i]=0;
for(i--;isspace(cword[i]);i--) cword[i]=0;
	if(!cword[0]) return(NULL);
return(cword);
}
void preprocess(void)
{
char *p,*t,*rb,*re,cword[80],cvalue[80];
def_stat *tmp;
long pc;
int skob;
pindex=0;
while((p=strchr(&pfile[pindex],'`')) || (p=strstr(&pfile[pindex],"function")))
```

```
{
        rb=re=NULL;
        pc=pindex;
        //Look if 'define or 'constant is inside comments space
        //for '// ... \n' comment type
while((t=strstr(&pfile[pindex],"//")) && t<p)
                {
                pindex+=t-&pfile[pindex]+2;
                rb=t;
                re=strstr(t,"\n");
                }
                if(rb<p && re>p)
                {
                pindex+=re-&pfile[pindex]+1;
                continue;
                }
                rb=re=NULL;
                pindex=pc;
                //for '/* ... */' comment type
                while((t=strstr(&pfile[pindex],"/*")) && t<p)
                {
                pindex+=t-&pfile[pindex]+2;
                rb=t;
                re=strstr(t,"*/");
                }
                if(rb<p && re>p)
                {
                pindex+=re-&pfile[pindex]+2;
                continue;
                }
                pindex+=p-&pfile[pindex];
```

```
                    if(!strncmp(p,"function",8))
                    {
                    if(psize>pindex+8) pindex+=8;
                    else return;
                    preprocess_skip_spaces();
                    if(pfile[pindex]=='[')
                    {
                        skob=1;
                        pindex+=1;
                    while(skob)
                        {
                            if(pfile[pindex]=='[') {pindex++; skob++; continue;}
                            if(pfile[pindex]==']') {pindex++; skob--; continue;}
                    pindex++;
                    }
                    preprocess_skip_spaces();
                    }
                    if(!preprocess_get_word(cword)) continue;
if(!functions_list) functions_list=functions_end=(FUNCTIONS *)malloc(sizeof(FUNCTIONS));
else functions_end=functions_end->next=(FUNCTIONS *)malloc(sizeof(FUNCTIONS));
                    strcpy(functions_end->name,cword);
                    functions_end->next=NULL;
                }
                else if(!strncmp(p,"`define",7))
                {
                    if(psize>pindex+7) pindex+=7;
                    else return;
                    preprocess_skip_spaces();
                    if(!preprocess_get_word(cword)) continue;
                    if(!preprocess_find_def(cword))
                    {
```

```
                        preprocess_skip_spaces();
                        if(!preprocess_get_value(cvalue)) continue;
                        tmp=preprocess_add_def(cword,cvalue);
                    }
                }
                else
                {
                    pindex++;
                    if(!preprocess_get_word(cword)) continue;
                    if(!(tmp=preprocess_find_def(cword))) continue;
                if(strlen(cword)+1<strlen(tmp->value))pfile=(char*)realloc(pfile,psize
                strlen(cword)+strlen(tmp->value));
                memmove(&pfile[pindex-strlen(cword)+strlen(tmp->value)-
                1],&pfile[pindex],psize pindex);
                memcpy(&pfile[pindex-strlen(cword)-1],tmp->value,strlen(tmp->value));
                if(strlen(cword)+1>strlen(tmp->value))        pfile=(char        *)realloc(pfile,psize-
                strlen(cword)+strlen(tmp->value));
                psize+=strlen(tmp->value)-strlen(cword)-1;
                pindex+=strlen(tmp->value)-strlen(cword)-1;
                pfile[psize]=0;
                }
                }
                pindex=0;
                }
EXTERNALS *find_external(char *name)
{
        EXTERNALS *tmp;
        for(tmp=externals_list;tmp;tmp=tmp->next) if(!strcmp(tmp->name,name)) return(tmp);
        return(NULL);
}
```