LIFETIME VALIDATION OF DIGITAL SYSTEMS VIA FAULT MODELING AND TEST GENERATION

by

Hussain Said Al-Asaad

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1998

Doctoral Committee:

Professor John P. Hayes, Chairman
Professor Richard B. Brown
Professor Trevor N. Mudge
Professor Karem A. Sakallah
Dr. Brian T. Murray, General Motors Research

For my family

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to my advisor John P. Hayes for his continuous guidance throughout my career. His vision, wisdom, and research style have been always an inspiration for me. Also, I would like to thank Dr. Brian T. Murray for his continuous support and advice, especially because I have learned a lot from cooperating with him on several research projects. Furthermore, I would like to thank my committee members for their time and effort: Professor Trevor Mudge, Professor Karem Sakallah, and Professor Richard Brown.

During my stay at Michigan, numerous friends and colleagues have made significant contribution to my learning process including: Gheith Abandah, Shawn Blanton, David Van Campenhout, Krish Chakrabarty, Amit Chowdhary, Avaneendra Gupta, Mark Hansen, Jonathan Hauke, Hyungwon Kim, Chih-Chieh Lee, Matt Postiff, Saqib Syed, Steve Raasch, and Hakan Yalcin. I thank them all for their help.

I would like to thank my mother, Wasfieh, and father, Said, as well as my entire family for their love and support throughout my education. Without their prayers, I would not have achieved anything. Furthermore, I would like to thank my friends who have made my stay at Michigan unforgettable including Hassan and Salwa El-Hor, Zahra Jishi, Omar Qasaimeh, Ghassan Shahine, and Gameel Zindani.

Finally, I would like to gratefully acknowledge the financial support, at different stages of my research work, of the National Science Foundation, General Motors R&D Center, DARPA, and the University of Michigan EECS department.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

ABSTRACT

LIFETIME VALIDATION OF DIGITAL SYSTEMS VIA FAULT MODELING AND TEST GENERATION

by

Hussain Said Al-Asaad

Chair: John P. Hayes

The steady growth in the complexity of digital systems demands more efficient algorithms and tools for design verification and testing. Design verification is becoming increasingly important due to shorter design cycles and the high cost of system failures. During normal operation, digital systems are subject to operational faults, which require regular on-line testing in the field, especially for high-availability and safety-critical applications. Fabrication fault testing has a well-developed methodology that can, in principle, be adapted for efficient design validation and on-line testing. This thesis investigates a comprehensive “lifetime” validation approach that uses fabrication fault testing and simulation techniques, and accounts for design errors, fabrication faults, and operational faults. The validation is achieved by the following sequence of steps: (1) explicit error and fault modeling, (2) model-directed test generation, and (3) test application.

We first present a hardware design validation methodology that follows the foregoing validation approach. We analyze the gate-level design error models used in prior research and show how to map them into single stuck-line (SSL) faults. We then describe an extensive set of experiments, which demonstrate that high coverage of the modeled gate-level errors can be achieved with small test sets obtained with standard test generation and sim-

ulation tools for fabrication faults. Due to the absence of published error data, we have systematically collected design errors from a number of microprocessor design projects, and used them to construct high-level error models suitable for design validation. Experimental results indicate that very high coverage of actual design errors can be obtained with test sets that are complete for a small number of design error models. We further present a new error model for control errors in microprocessors and a validation approach that uses it.

We next show how to achieve built-in validation by embedding the test application mechanism within the circuit under test (CUT). This is realized by built-in self-test (BIST), a design-for-testability technique that places the testing functions physically within the CUT. We demonstrate how BIST, which in the past has been typically used only for fabrication faults, can be applied to on-line testing. On-line BIST can provide full error coverage, bounded error latency, low hardware and time redundancy. We present a method for the design of efficient test sets and test generators for BIST, especially for high-performance scalable datapath circuits. The resultant test generator designs meet the following goals: scalability, small test set size, full fault coverage, and very low hardware overhead. We apply our method to various datapath circuits including a carry-lookahead adder, an arithmetic-logic unit, and a multiplier-adder.

CHAPTER 1
INTRODUCTION

The field of digital systems has undergone a major revolution in recent decades. Circuits are shrinking in physical size while growing both in speed and range of capabilities. This rapid advancement is not without serious problems, however. Especially worrisome are verification and testing, which become more important as the system complexity increases and time-to-market decreases. The inadequacy of existing verification methods is illustrated by the 1994 Pentium microprocessor's FDIV design error, which cost its manufacturer (Intel) an estimated $500 million [29]. The FDIV error involved a set of missing entries in a lookup table used in the hardware algorithm implementing the divide operation, and caused the Pentium's floating-point divide instructions to produce inaccurate results for certain input data [61]. The inadequacy of existing testing methods is also illustrated by the 1990 breakdown of AT&T's long distance network, which cost AT&T around $75 million [38].

Due to the high cost of failure, verification and testing now account for more than half of the total lifetime cost of an integrated circuit (IC) [111]. Increasing emphasis needs to be placed on finding design errors and physical faults as early as possible in the life of a digital system, new algorithms need to be devised to create tests for logic circuits, and more attention should be paid to synthesis for test and on-line testing. On-line testing requires embedding logic that continuously checks the system for correct operation. Built-in self-test (BIST) is a technique that modifies the IC by embedding test mechanisms directly into it. BIST is often used to detect faults before the system is shipped and is potentially a very efficient way to implement on-line testing.

This thesis presents a comprehensive validation approach for digital systems that is based on fault modeling and test generation. Contrary to most prior research, the approach aims at detecting design errors and physical faults throughout the lifetime of a digital system. In this chapter, we review the types of faults and errors that arise during the system's lifetime, and the relevant methods for detecting and simulating these faults and errors. We then discuss our proposed approach to lifetime validation.

1.1 System Development

Digital systems are manufactured on either a single chip, called a system-on-a-chip (SOC), or on several chips. In this thesis, we assume that digital systems follow the SOC-style, however, the results can be easily extended to multiple-chip systems. Figure 1.1 shows the lifetime of a typical IC-based digital system (SOC) divided into three phases: design, manufacturing, and operation. During the design phase, an initial design concept is transformed in a top-down manner into an IC specification (mask layout). The design phase for a new system involves an extensive requirements analysis, resulting in a detailed system specification describing what the system must do. Then a behavioral description is prepared, which describes the system's operation in detail. Following the behavioral design, a register-transfer level (RTL) design is created that includes modules such as buses, registers, logic blocks, and finite state machines. The RTL components are in turn implemented using gate-level components, such as gates and flip-flops. Finally, a mask layout targeting a certain IC technology, such as CMOS, is generated.

Systems designed in the above fashion are said to be *hierarchical*. We use the term *module* to refer to a design block at any abstraction level whose function is clearly defined. A module at a certain level of the system's hierarchy abstracts away the details of

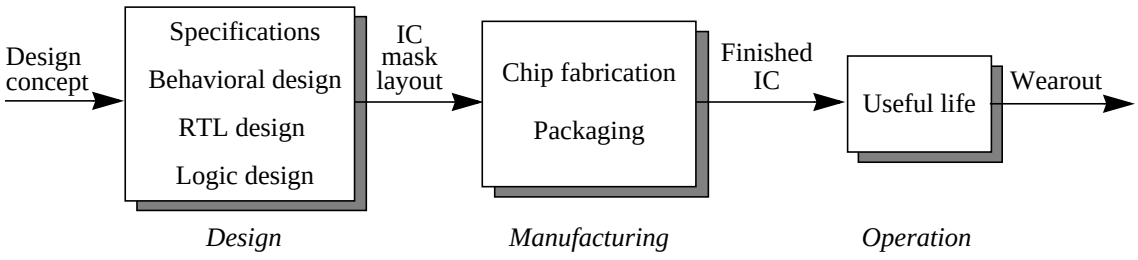


Figure 1.1 Lifetime of a typical system-on-a-chip (SOC).

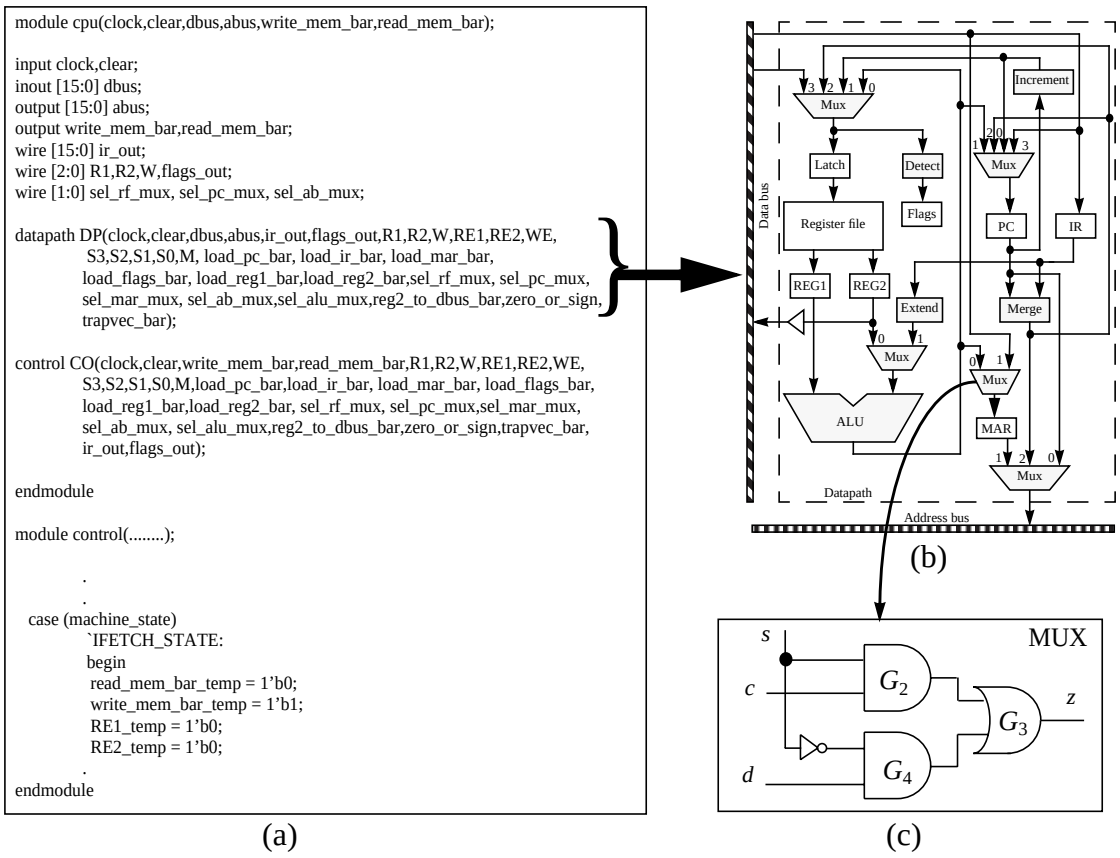


Figure 1.2 Microprocessor design at the (a) behavioral, (b) RTL, and (c) gate levels.

the lower level which implements it. An example is given in Figure 1.2, where the hierarchical structure of a microprocessor is shown along with some modules at different abstraction levels. Figure 1.2a shows a behavioral Verilog description of the microprocessor and Figure 1.2b shows the RTL design of its datapath module. Figure 1.2c shows the gate-level netlist of one of the multiplexers in the datapath module. Designs at the behavioral and register-transfer levels are often considered as *high-level*.

Computer-aided design (CAD) tools are normally used throughout the design process for synthesis, optimization, and verification. Synthesis tools speed up the design cycle and reduce the human design effort and cost. For example, a synthesis tool transforms a design from a higher level of abstraction, such as the microprocessor behavioral design in Figure 1.2a, to a lower one such as the gate-level design in Figure 1.2c. Optimization tools enhance the design quality, while verification tools ensure the correctness of the final design.

The manufacturing phase in the lifetime of a digital system takes the IC mask layout and yields a finished IC. First, the system is fabricated on a chip and then it is packaged into the finished IC that is ready to be used. The final phase of the system's lifetime is normal operation, where the system performs its intended job.

Faults occur throughout the lifetime of a digital system. They can be classified by the phase in which they occur as follows: design faults (more commonly called design *errors*) which appear in the design phase, fabrication faults which appear in the manufacturing phase, and operational faults which occur during normal operation. Fabrication and operational faults are normally considered to be "physical" faults.

Design faults. The three major types of design faults in a system are those "inherited" by the system, those made by human designers, and those made by the computers that aid in the design process [18]. Inherited faults are those that exist before starting the design process. For example, conflicting specifications are considered as inherited faults. These faults cannot be completely eliminated because no system is completely new. Human design faults fall into two major categories: data preparation faults and transcription faults. Data preparation faults usually result from making wrong decisions, miscalculations, etc. Transcription faults are the result of transferring data from one medium to another without changing its content. Faults due to mistakes in keying design data into a computer are considered transcription faults. Examples of data preparation and transcription faults are shown in Figure 1.3. Human design faults must be detected as early as possible because it costs a lot to detect and correct them later. They can happen at any stage of the design process and can remain undiscovered throughout the lifetime of the system. The third source of design faults is the CAD system used to automate and speed up the design cycle. Bugs

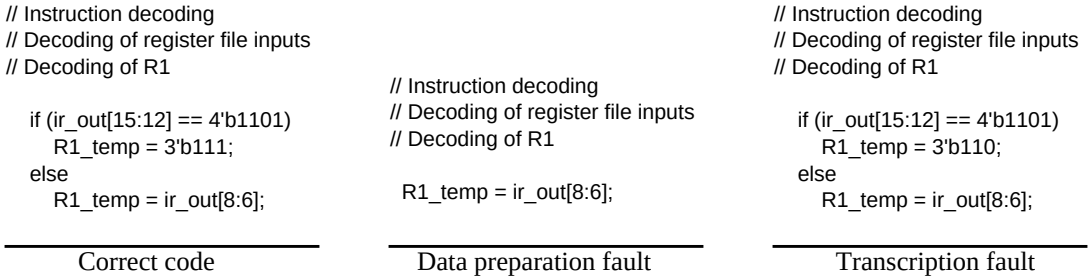


Figure 1.3 Examples of data preparation and transcription faults.

in the CAD software (simulators, translators, layout generators, etc.) can lead to incorrect design. A hardware malfunction in the CAD workstation such as a bad storage sector on the disk can also cause design faults.

Fabrication faults. These defects are not directly attributable to human error; instead they result from an imperfect manufacturing process. For example, shorts and opens are common defects in the manufacture of very large-scale integrated (VLSI) circuits using CMOS technology, the industry standard. These defects can have a severe effect on the behavior of an IC. For example, if the transistor T_2 of the NOR gate circuit shown in Figure 1.4 is shorted, then there will be a direct conducting path from VDD to GND, when $AB = 01$. Such a path will not only produce an erroneous value at the output Z, but also may cause the overall integrated circuit to fail due to the increase in static power consumption and heat. Other CMOS fabrication defects include incorrect transistor threshold voltage, improper doping profiles, mask alignment errors, and poor encapsulation. Accurate identification of fabrication defects is important in improving the manufacturing yield [64].

Operational faults. Most of these faults are caused by external disturbance during the normal operation of the digital system. Common sources of operational faults are electromagnetic interference, operator mistakes, environmental extremes, and wearout. For example, if a digital system is subjected to extreme temperature variations, the system can produce incorrect results. Moreover, excessive temperature and humidity accelerate the aging of components. Some operational faults arise due to the movement of the system, especially in mobile applications. Also, some IC faults are due to electron migration, where metal connectors inside an IC package thin out with time and break. Operator mistakes are consid-

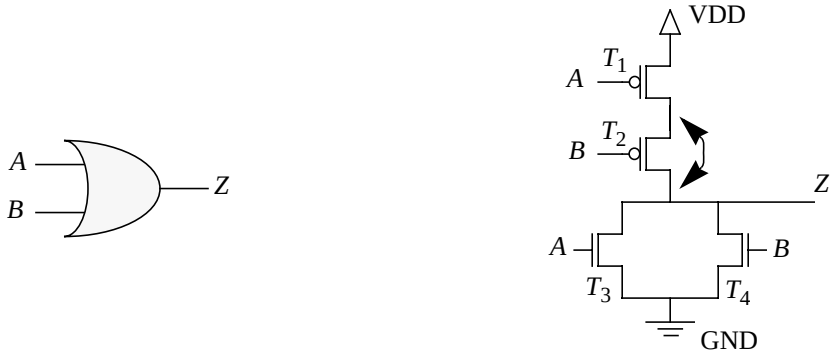


Figure 1.4 A NOR gate and its transistor implementation.

ered in this class because an operator may provide incorrect commands which lead to system failure.

Operational faults are usually classified according to their duration:

- *Permanent* faults remain in existence indefinitely if no corrective action is taken. Many of these are residual design or manufacturing faults. Those that are not most frequently occur during changes in system operation, for instance, after system start-up or shutdown, or as a result of a catastrophic environmental disturbance such as a collision.
- *Intermittent* faults appear, disappear, and reappear repeatedly. They are difficult to predict, but their effects are highly correlated. Most intermittent faults are due to marginal design or manufacturing. The system works well most of the time, but fails under atypical environmental conditions.
- *Transient* faults appear and disappear quickly, and are not correlated with each other. They are most commonly induced by random environmental disturbances.

To detect faults, we need to apply input stimuli (tests) that will force the circuit under test to fail. A circuit is said to fail if the function it implements differs from the function it was designed to implement. Fault models provide a consistent and technology-independent mechanism for how a logic function might fail, as well as a standard yardstick for measuring the quality of a set of tests. In developing a fault model, it is important to strike a balance between accuracy and complexity. The model must also match the characteristics of the design level(s) at which it is used.

The modeling of design errors has rarely been considered before due to the lack of published error data. Abadir et al. [2] defined a set of likely design errors for combinational logic and have shown that complete test sets for fabrication faults detect many, but not all, such errors. In Chapter 2, we reduce most of the known gate-level design errors to five classes. Al Hayek and Robach [15] have adapted mutation errors from the software testing method called mutation testing, to hardware design verification in the case of small VHDL modules. Mutation testing [44][45] generates tests that distinguish a program under test from its mutants, where a mutant is created by injecting a small error (mutation) such as

changing an add to subtract. The rationale for the approach is based on two controversial hypotheses: 1) programmers write programs that are close to correct ones, and 2) a test set that distinguishes a program from all its mutants is also sensitive to more complex errors. Current mutation-testing tools are slow and are only suitable for testing relatively small programs [112].

Developing fault models for fabrication faults has received a lot of attention in the past. The most common such fault model is the *single stuck-line* (*SSL*) fault model [4], under which any single signal line in a logic-level system model can become permanently fixed (stuck) at a logical 1 or 0 value. It is a simple, technology-independent, logical fault model. While it represents only a small number of different manufacturing faults directly, tests derived for SSL faults detect most faults occurring in practice. Since the number of SSL faults is proportional to the number of lines in the circuit, it is feasible to consider all possible SSL faults, even in large-scale designs.

Another model for fabrication faults is the *input pattern* (*IP*) fault model [22], under which a fault changes a module's response to some input pattern. Formally, an IP fault in a single-output module M changes the response of M to the input pattern V from F_V to $\overline{F}_V$. The number of IP faults in a circuit C is proportional to $G \times 2^p$, where G is the number of modules in C and p is the average number of inputs to the modules. A *functional fault* in a module M changes the function implemented by M into a known faulty function, and can be represented by a set of IP faults. On the other hand, a *cell fault* in M changes the function implemented by M into an unknown faulty function. To detect a cell fault, exhaustive testing of M is needed.

Few formal higher-level fabrication fault models exist, and those that do are often not sufficient to detect all actual faults. Thatte and Abraham [108] classified faults in microprocessors according to their effect on some register-level components. These effects include such symptoms as register decoding errors, and data transfer errors. Other higher-level fault models are extensions of the gate-level fault models. For example, Bhattacharya and Hayes [21] extended the SSL fault model to include all bits of a bus, leading to the concept of bus faults.

Since operational faults and fabrication faults are physical in nature, fabrication fault models are also used for operational faults. The SSL fault model is also the most commonly used model for operational faults.

Testing is the process of error/fault detection. It involves exercising a system with input patterns (test vectors) and observing the resulting response vectors to ascertain whether the system behaves correctly. Testing methods can be classified by the types of faults addressed: design verification for design faults, manufacture testing for fabrication faults, and on-line testing for operational faults.

1.2 Design Verification

Design verification is the process of ensuring that a design exhibits certain required or "correct" behavior. There are two broad approaches to hardware design verification: formal methods and simulation-based methods. Formal methods try to verify the correctness of a system by using mathematical proofs [117]. Such methods implicitly consider all possible behavior of the models representing the system and its specification. The accuracy and completeness of the system and specification models are a fundamental limitation for any formal method. Furthermore, formal methods are not yet feasible for large, complex designs due to their excessive time and memory requirements.

An example of a formal verification method is boolean comparison [28][94][115], where verification becomes proving the equivalence of two logical representations of the same design. Most proposed algorithms for boolean comparison apply to gate-level designs and are based on ordered binary decision diagrams (OBDDs) [28]. Since an OBDD is a canonical representation of a logic function, OBDD-based verification methods aim at constructing the OBBD for each of the two design representations and then proving their equivalence. These algorithms often fail for large circuits due to the large memory requirements for storing the OBDDs.

Recently, Kunz and Pradhan [71][72] introduced a procedure called recursive learning, which they use to prove the equivalence of two gate-level designs. The main idea is to use structural methods to capture similarity between two sub-circuits and then use an OBDD-based functional approach to prove the equivalence of the two circuits. Such a method is

useful for verifying the functionality of a circuit after simple modifications have been made to the circuit. However, recursive learning cannot be used to verify the equivalence of two designs at different levels of abstraction.

Simulation-based design verification tries to uncover design errors by detecting a circuit's faulty behavior when tests (simulation vectors) are applied. Several types of tests can be used for verification:

- *Exhaustive tests*: Simulation using all possible input combinations as tests is a possibility, at least for small combinational circuits.
- *Focused tests*: These are hand-written by the designers focusing on basic functionality and important exceptional or "corner" cases in the design. These tests may be effective; however, the process of generating such tests is far from being fully automated. Recently, tools have been developed to assist in the generation of focused tests [35][58].
- *Random tests*: Random vectors can cover a substantial number of design faults, but their coverage is uncertain even with very large test sets [65]. Random simulation provides a cheap way to take advantage of the billion-cycles-a-day simulation capacity of networked workstations available in many big design organizations. Sophisticated systems have been developed that are biased towards corner cases, thus improving the quality of the tests significantly [7].
- *Universal tests*: Implementation-independent "universal" test sets [24][36] exploit any unateness properties of the functions being implemented, but the tests become exhaustive when, as is often the case, there are no unate variables.
- *Physical-fault-oriented tests*: Another approach is to use specific, deterministic test sets generated for a physical (fabrication) fault model like the SSL model to verify the design. It has been shown that many, but not all, gate-level design errors can be detected by using test sets derived for SSL faults [2]. We verify this result experimentally in Chapter 2.

Instead of the usual binary values for the tests described above, symbolic simulation [63] uses logical expressions for the state and input variables. The expressions must cover all valid test cases and avoid those that violate the circuit's input constraints. For example,

the expressions must cover the test cases {00, 01, 10} and avoid {11} for the selection bus of a 3-input multiplexer. This technique is suitable for applications where input constraints can be easily determined. Another simulation-based comparison approach, called probabilistic design verification [62], uses integer values for input variables. This method establishes a transformation from the boolean function realized by a circuit to an arithmetic function. To compare two gate-level designs, the circuits are first transformed to arithmetic functions and then simulated with integer values for inputs instead of the usual binary values.

Common to all the tests mentioned above is that they are not targeted at specific actual design errors. This poses the problem of quantifying the effectiveness of a test set, such as the number of errors detected or "covered". Various coverage metrics have been proposed to address this problem. These include code coverage metrics from software testing [7][20][32], finite state machine coverage [58][66][96], architectural event coverage [66], and observability-based metrics [46]. A shortcoming of all these metrics is that the relationship between the metric and the detection of actual design errors is not well understood.

To overcome the problems of the above approaches, model-based design verification attempts to model design errors directly and generate tests for the synthetic models. An example of model-based design verification is Al Hayek and Robach's method [15] which was adapted from mutation testing [44]. Although mutation testing is considered too costly for wide-scale industrial use, it is one of the few approaches that has yielded an automatic test generation system for software testing, as well as a quantitative measure of error coverage (mutation score) [68].

Although model-based design verification is intended for design error detection, the generated deterministic test sets also appear to be useful for error location, diagnosis, and correction [39][40][73]. This is the case since these test sets (simulation vectors) can be surprisingly small and can guarantee the detection of broad categories of design errors. In contrast, random vectors [59][98] do not guarantee the detection of all errors, and the use of exhaustive tests [98] is rarely feasible.

Design verification via model-based testing suffers from a major limitation. Since no complete set of design error models is known, a system that passes the testing is correct only with respect to the considered error models. Hence, correctness with respect to unmodeled errors cannot be guaranteed. In spite of this, simulation is an effective technique for design verification, and experience has shown that it helps discover most design errors early in the design process.

1.3 Manufacture Testing

Manufacture testing, also called acceptance testing, deals with the detection of malfunctions in a digital system due to fabrication faults. In principle, it is possible to generate tests without the use of an explicit fabrication fault model. For example, exhaustive, universal, and random tests can be used to detect fabrication faults. However, in practice, testing usually employs a fault model, and tests are generated to detect all occurrences of the modeled faults. If the system passes the tests, it is declared free from faults and can be shipped to customers. Otherwise, the system is diagnosed to identify the causes of failure and improve the yield of future production.

Most deterministic fabrication-fault-oriented test generation algorithms are based on the following three basic steps: (1) activate the currently selected fault, (2) propagate an error signal from the site of the fault to an observable output, and (3) justify the internal signals by assigning values to the primary inputs. We next describe the test generation and fault simulation methods used in prior research.

Gate-level test generation. The most studied approaches to test generation employ gate-level structural models; nearly all commercial test generators do so. The most widely known gate-level test generation algorithms are the *D*-algorithm and PODEM (Path Oriented DEcision Making) [4].

If a line in a circuit is 0 (1) when it should be 1 (0), the error signal value on that line is represented by the symbol D ($\overline{D}$) for discrepancy. Consider the 2-input multiplexer circuit in Figure 1.5. A stuck-at-1 fault at the output of gate G_2 can be activated by attempting to make the output 0. A signal on this line will be detected as an error if it is assigned the

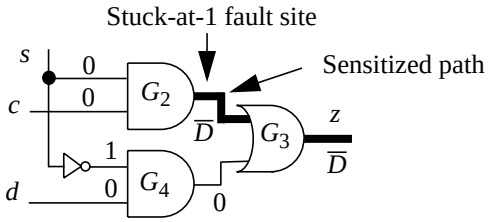


Figure 1.5 A 2-input multiplexer circuit.

value $\overline{D}$. The fault G_2 stuck-at-1 is activated by assigning 0 to one or both of *s* and *c*. Since the output of G_2 is not a primary output, we need to propagate the $\overline{D}$ error signal from the output of G_2 to the primary output *z* so that it can be observed. This is done by assigning values to signals in the circuit to sensitize the output of G_2 to G_3's output, i.e. by assigning 0 to the output of G_4. The error propagation process just described is called *D*-propagation. After propagating the error, we need to assign the primary inputs of the circuit to satisfy the assignments made to internal signals. So, we need to assign 0 to input *d* to satisfy the value 0 at G_4's output. This process of determining complete and consistent specifications of circuit signal values is called justification. After justification is completed, the values of the primary input signals form the test for the fault. Hence the test for the fault G_2 stuck-at-1 is *scd* = 000.

The *D*-algorithm provides a systematic implementation of the *D*-propagation and justification steps described above. In the case of *D*-propagation, several *D*'s ($\overline{D}$'s) may be propagated simultaneously, since sometimes an error signal must be propagated along more than one path to reach an observable output. In the *D*-algorithm, the *D*-propagation and justification operations make only local assignments of signal values. To justify a value on the output of gate *G*, the *D*-algorithm makes assignments to the inputs of *G*. If these are not primary inputs, assignments to them become objectives for subsequent justification steps.

Both *D*-propagation and justification involve decisions or choices. Whenever there are several alternative ways to justify a line or propagate an error, we choose one of them to try. But in doing so we may select a decision that leads to an inconsistency or conflict. Therefore most search strategies use backtracking to systematically explore the complete

space of possible solutions and recover from incorrect decisions. Most gate-level testing algorithms use chronological backtracking where after a conflict is detected, the test generation algorithm returns to and alters the last decision made.

The *D*-Algorithm uses a greedy value assignment policy—it assigns signal values at the earliest opportunity. This reduces the number of signal evaluations but this makes the decision-making more vulnerable to conflicts and hence increases backtracking. The PODEM test generation algorithm avoids this problem by backtracking only at primary inputs. PODEM does not justify internal values explicitly, as in the *D*-algorithm. To satisfy an internal objective such as a D or $\overline{D}$ on some internal line, a value is assigned to a primary input and the circuit is simulated. If the simulation proves that the assignment does not satisfy the objective, PODEM assigns another input value. If during simulation, two values conflict on a line, the algorithm backtracks by changing the value of the last assigned input. When both values have been tried unsuccessfully, the algorithm backtracks to the next-to-last assigned input. In this way, PODEM can exhaustively explore all possible circuit states, but only implicitly.

A number of test generation techniques have been developed that extend PODEM. Their goal is to reduce the number of backtracks by identifying choices a test generation algorithm might make that cannot lead to a solution, without actually pursuing every decision. For example, the FAN algorithm [4] seeks to identify conflicts at fanout branches within a circuit, thereby avoiding backtracks at the primary inputs and the cost of simulating large parts of the circuit. Conflicting assignments at fanout branches cannot be satisfied by any assignment at primary inputs.

The *D*-algorithm and PODEM can be extended to generate tests for synchronous (clocked) sequential circuits. The extension is based on a modeling technique which transforms a sequential circuit into an iterative combinational array, one cell of which is called a time frame. In this transformation a flip-flop is modeled as a combinational element having an additional input q to represent its current state and an additional output q^+ to represent its next state, which becomes the current state in the next time frame. An input vector of the iterative combinational array represents an input sequence for the sequential circuit.

High-level test generation. Due to the high complexity of gate-level test generation and the hierarchical nature of the design process, several high-level or functional test generation methods have been introduced. The design is then described by an interconnection of high-level (RTL) modules, which include word gates, decoders, multiplexers, encoders, demultiplexers, tristate buffers, comparators, 1-bit adders, and buses. An example of a small high-level design is shown in Figure 1.6. High-level test generation has the following potential advantages:

- *Fast module evaluation*: Since modules are described at the functional level, they can be evaluated faster than their gate-level equivalents. For example, evaluating the code in Figure 1.6a is faster than evaluating the approximately 90 gates in a gate-level equivalent of Figure 1.6b.
- *High-level implication*: Implication at the high level may lead to finding values of signals where low-level implication fails. For example, $A = 0$ and $D = 5$ in Figure 1.6b imply that $B = 5$. However, it is not possible to reach to this implication using an equivalent gate-level design of Figure 1.6b.
- *Unique sensitization*: At the high level, efficient procedures can be developed to determine the signals necessary to propagate fault effects at the inputs of a high-level module to its outputs. For example, to propagate a fault effect from input *C* of the multiplexer in Figure 1.6b, we need to set *s* to 0. Moreover, a propagation check routine may also be developed to anticipate conflicts earlier and hence reduce the number of backtracks.
- *Reduced backtracking*: This is due to the following: (1) high-level descriptions enclose reconvergent fan-out and hence leads to fewer poor decisions, and (2) module-level decision making leads to improved global implication and conse-

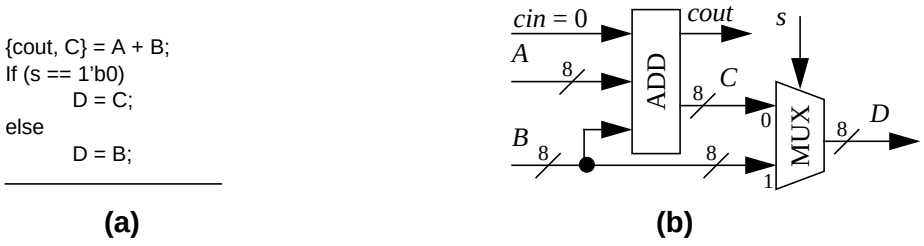


Figure 1.6 A high-level design example: (a) behavioral and (b) RTL.

quently conflicts are detected earlier and alternatives are tried sooner.

Several high-level branch-and-bound combinational test generation methods for SSL faults have been introduced [31][91][103]**.** In these methods, the design description is an interconnection of high-level modules. Each module is represented by a data structure using some form of a high-level representation, which may be expanded to the gate level once the SSL faults inside the module are targeted. To target high-level modules whose gate-level design is unknown and to minimize the need of dynamically expanding those whose gate-level design is known, either faults inside the module are transferred to its input(s)/output(s) or a complete test set for SSL faults is precomputed for the module. Typical experimental results show that high-level test generation produce tests for SSL faults with less CPU time, less memory, and better coverage than gate-level test generation.

The test generation algorithm of Sarfert et al. [103] is divided into two phases: a random phase and a deterministic phase. The random phase applies pseudo-random patterns in parallel for all SSL faults. The deterministic phase targets the remaining SSL faults heuristically in the following order: faults at the primary inputs of the design, faults inside modules by expanding one module at a time to gate level, faults in input(s) and output(s) of modules. The test generation algorithm of Calhoun and Brglez [31], an extension of PODEM that is called MODEM, is similar to that of Sarfert et al.

Narain et al. [91] use precomputed test sets for modules in the form of one test for every SSL fault. The advantage is that the bad value is known, hence the error propagation is simpler. The disadvantages are: (1) precomputed tests may not be justifiable at the high-level and hence the coverage may be decreased, (2) the test generation time may increase since we need to justify more precomputed tests, and (3) overspecification of signals, where no unknown bits are allowed, may result in a large number of backtracks.

The test generation algorithm of Narain et al. is an extension to gate-level algorithms with a justification-first strategy, as in PODEM, or a propagation-first strategy, as in the *D*-Algorithm. To minimize the effect of backtracking on the test generation algorithm, a method called *dependency-directed backtracking* is introduced. Unlike the usual chrono-

logical backtracking method, dependency-directed backtracking causes the branch-and-bound algorithm to jump immediately to the decision nodes that are responsible for a conflict.

Thatte and Abraham [108] propose a high-level test generation scheme for microprocessors based on a system graph model. It uses knowledge about the register-transfer operations that are normally present in the high-level description of the instruction-set architecture. The system graph model has a vertex for every register in the microprocessor. An edge is inserted between nodes *A* and *B* if an operation to transfer data from register *A* to register *B* is possible. So, data transfer operations in the microprocessor are mapped to paths in the system graph. It is assumed that physical failures can corrupt the high-level operations of the microprocessor. Fault models are defined for the following functions: register decoding, instruction decoding and control, data storage, and data transfer. For example, a fault in register decoding leads to reading from or writing to the wrong register. The test generation algorithm produces sequences of instructions to detect the above faults in the microprocessor with the hope of detecting the low-level SSL faults. The approach has the following limitations: (1) it is only applicable to microprocessors, (2) it tends to generate large sequences of instructions for certain faults, and (3) it is unable to deal directly with datapath faults.

Lee and Patel [77] present an architecture-level test generator (ARTEST) for a hierarchical design environment based on precomputed tests for high-level modules. The system model used by ARTEST is composed of a gate-level control unit and a high-level datapath unit. The faults considered are limited to the datapath only. Lee and Patel assume that all possible error signals associated with each module is unknown. Testing involves interaction between a high-level test generator for the datapath and a gate-level test generator for control, with a complex interface algorithm that transfers objectives between these two test generators. ARTEST tries to minimize calls to the interfacing algorithm, hence high-level dependency-directed backtracking is used first until a maximum number of backtracks is reached and then gate-level backtracking is used.

In a subsequent paper [76], Lee and Patel suggest that a high-level branch-and-bound algorithm is likely to be inefficient in making high-level search decisions when the mod-

ule diagram of the circuit under test is complex, in particular where the data and control are highly intertwined. As an alternative to the branch-and-bound algorithm, they propose a signal-driven discrete relaxation technique for the architecture-level test generation problem. An underdetermined system of non-linear equations is derived for each control unit instruction, using symbolic simulation. The resulting system of equations is solved iteratively using a Gauss-Seidel algorithm.

Lee and Patel [78] further present another high-level technique to generate tests for datapath faults in microprocessor-like circuits. This method separates the hierarchical test generation into two phases: (*i*) an instruction-sequence assembling algorithm at the architecture level and (*ii*) a relaxation-based algorithm that produces a fully-specified instruction sequence. The technique may be summarized as follows:

1. Perform symbolic simulation for each instruction to derive a system of equations that represent the instruction behavior in the datapath.
2. Derive a structural data flow graph (DFG) for each instruction. The inputs (outputs) of DFG include the primary inputs (outputs) of the microprocessor and present- and next-state lines. The DFG is used only for path selection without explicitly examining the detailed functionality of the DFG nodes.
3. Calculate the justification and propagation cost for state lines.
4. Inject a test vector at the input of module under test.
5. Assemble an instruction sequence for both fault propagation and signal justification. The sequence is heuristically assembled based on testability measures.
6. Derive a complete system of equations for the instruction sequence. Use discrete relaxation algorithm to solve it.

Murray and Hayes [87] present a test generation algorithm *PathPlan* that processes test data, including precomputed test stimulus and response values, as indivisible units contained in structures called test packages. High-level module inputs and outputs are identified as control or data. The signal values carried by buses are considered to be vector sequences. The test, propagation, and control information for a module are often grouped together into a test package. *PathPlan* uses test packages for faults of a module *M* to activate errors and other test packages, called propagation test packages, to propagate the

responses of *M* to primary outputs. Justification is treated the same as propagation—test packages are used to determine the inputs of modules once their outputs are known. *Path-Plan* requires that all modules have transparent paths where fault responses are either unchanged or inverted when passing through modules en route to primary outputs. Moreover, it can only handle combinational circuits with regular fanout.

In [88], Murray and Hayes present an improved test generation algorithm *PathPlan2* to handle the problems of error propagation through modules with no transparent mode and those with irregular fanout. It is noted that modules are normally partially transparent—some input combinations at the input of a module cannot be distinguished at the output of the module. A propagation theory is developed to determine if error propagation can be achieved by a path through partially transparent modules to primary outputs. This also leads to a method for complete propagation of error information over multiple non-transparent paths.

Hansen and Hayes [52][53] present a high-level functional fault modeling and test generation method that ensures full detection of low-level SSL faults. In this method, a set of independent functional faults, called SSL-induced faults (SIFs), are derived or "induced" from the gate-level SSL faults. The method is illustrated by manually deriving a set of complete functional circuit models and tests for representative 74X-series and ISCAS-85 benchmark circuits. The results demonstrate that functional testing can, with less effort than conventional methods, produce near-minimal test sets that provide complete coverage of SSL faults in practical circuits. A fault generator *SIFgen* was developed to generate SIFs automatically from circuit description and a test generation algorithm SWIFT for SIFs was proposed, however, it was not completely implemented.

Fault Simulation. Fault simulation [4] consists of modeling a circuit's behavior in the presence of faults. By comparing the faulty response of the circuit to its fault-free response using the same test set *T*, we can determine the faults detected by *T*. Fault simulation has many applications such as test set evaluation, fault-oriented test generation, and fault dictionary construction.

There are several general methods for fault simulation such as serial, parallel, deduc-

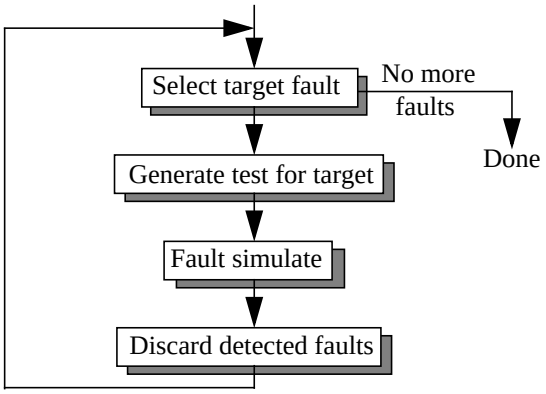


Figure 1.7 Use of fault simulation in test generation.

tive, and concurrent [4]. Serial fault simulation is the slowest method of all, but uses the least memory. It is based on simulating the fault-free circuit and the circuit in the presence of one fault at a time, and then comparing the responses of the faulty and the fault-free circuits; if they differ, the fault is detected. The process is repeated for all faults in sequence, hence the execution time is proportional to the number of faults in the circuit. Parallel fault simulation simulates the good circuit and a fixed number, say *W*, of faulty circuits simultaneously. The values of a signal in the good circuit and the values of the corresponding signals in the *W* faulty circuits are packed together in the same memory location of the host computer. It is faster than serial simulation but it needs more memory and more complex code. The deductive and concurrent fault simulation techniques determines all faults in the circuit detected by a given test in one forward pass through the circuit. These methods are fast, but have unpredictable memory requirements.

An important and relatively new use of simulation is in test generation for both design errors and physical faults (Figure 1.7). We develop an error/fault simulator ESIM (Chapter 2 and Appendix A) and use it to evaluate the coverage of modeled gate-level design errors by specific test sets. The underlying algorithm of ESIM is critical path tracing, a fault simulation method that simulates the fault-free circuit under a test set *T* and uses the computed signal values for tracing sensitized paths from primary outputs towards primary inputs to determine detected faults by *T*. The method has received attention [5][74][81] because it directly identifies the faults detected by a test without simulating all possible faults, and thus is faster than serial fault simulation.

1.4 On-Line Testing

On-line testing addresses the detection of *operational* faults, and is found in computers that support critical or high-availability applications. The goal of on-line testing is to detect fault effects, that is, errors, quickly and take appropriate corrective action. For example, in some safety-critical applications, the computer system is shut down after an error is detected. In other applications, error detection triggers a reconfiguration mechanism that allows the system to continue its operation, perhaps with some degradation in performance. On-line testing can be performed by external or internal monitoring using either hardware or software; internal monitoring is referred to as *self-testing*. Monitoring is internal if it takes place on the same substrate as the circuit under test (CUT). This is usually considered to be inside an IC.

There are four primary parameters to consider in the design of an on-line testing scheme:

- *Error coverage (EC)*: This is defined as the fraction of all modeled errors that are detected, usually expressed in percent. Critical and highly available systems require very good error detection or *error coverage* to minimize the impact of errors that lead to system failure.
- *Error latency (EL)*: This is the difference between the first time the error is activated and the first time it is detected. *EL* is affected by the time taken to perform a test and by how often tests are executed. A related parameter is *fault latency* (*FL*), defined as the difference between the onset of the fault and its detection. Clearly, $FL \geq EL$, so when *EL* is difficult to determine, *FL* is often used instead.
- *Hardware redundancy (HR)*: This is the extra hardware (IC chip area) needed to perform on-line testing.
- *Time redundancy (TR)*: This is the extra time needed to perform on-line testing.

An ideal on-line testing scheme would have 100% error coverage, error latency of 1 clock cycle, no hardware redundancy, and no time redundancy. It would require no redesign of the CUT, and impose no functional or structural restrictions on the CUT. Most on-

line test methods meet some of these constraints without addressing others. Consideration of all the parameters discussed above in the design of an on-line testing scheme can create conflicting goals. High coverage can require high *EL*, *HR* and *TR*. Schemes with immediate detection ($EL = 1$) minimize time redundancy, but require more hardware. On the other hand, schemes with delayed detection ($EL > 1$) reduce the time and hardware redundancy on the expense of increased error latency.

To cover all classes of operational faults described earlier, two different modes of on-line testing are employed: *concurrent testing* which takes place during normal system operation, and *non-concurrent testing* which takes place while normal operation is temporarily suspended. These modes can often be combined to provide a comprehensive on-line testing strategy at acceptable cost.

Non-concurrent testing is either event- or time-triggered, and is characterized by low hardware and time redundancy. Event-triggered testing is initiated by key events or state changes in the life of a system, such as start-up or shutdown, and its goal is to detect permanent operational faults. It is usually advisable to detect and repair permanent faults as soon as possible. Event-triggered tests resemble manufacturing tests. Any such test can be applied on-line, as long as the required testing resources are available. Typically the hardware is partitioned into components, each of which is exercised by tests specific to that component. Figure 1.8 depicts a taxonomy of on-line testing techniques for microcontrollers. RAMs, for instance, are tested by manufacturing tests specifically designed for RAMs, such as March tests [93].

Time-triggered or periodic testing is activated at predetermined times during system operation. It is done periodically to detect permanent operational faults using the same types of tests applied by event-triggered testing (see Figure 1.8). This testing approach is useful in systems that run for extended periods, where no significant events occur that can trigger testing. Periodic testing is also essential for detecting intermittent faults. Such faults typically behave as permanent faults for short time intervals. Since they usually represent conditions that must be corrected, diagnostic resolution is important. Periodic testing can identify latent design or manufacturing flaws that only appear under the right

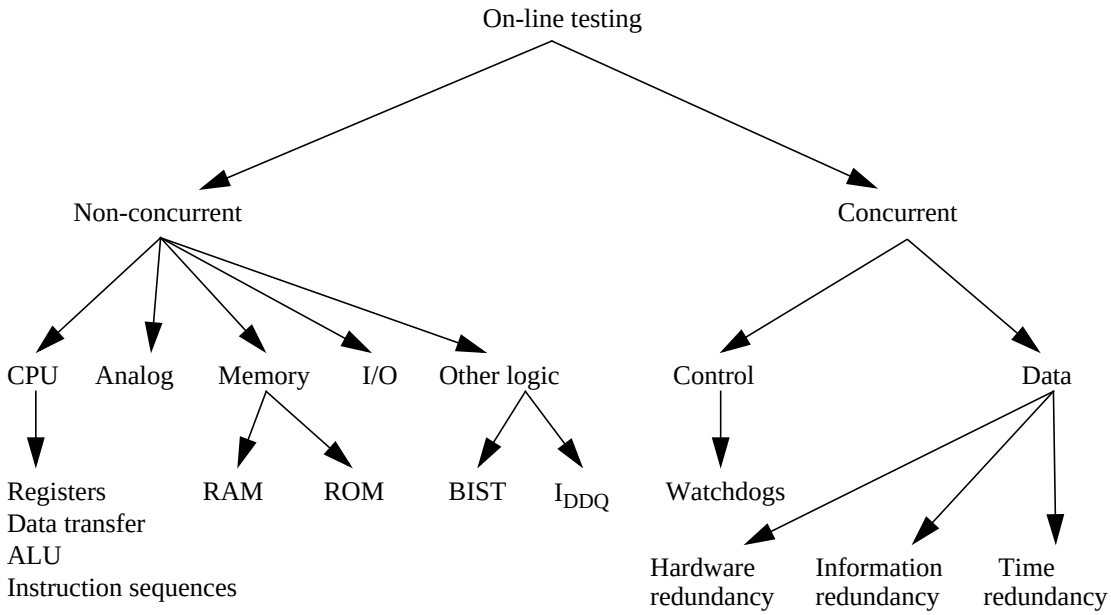


Figure 1.8 Taxonomy of on-line testing methods for microcontrollers.

environmental conditions.

Non-concurrent testing cannot detect transient or intermittent operational faults whose effects disappear quickly. Concurrent testing, on the other hand, continuously checks for errors due to such faults. However, concurrent testing is not by itself particularly useful for diagnosing the source of errors, so it is often combined with diagnostic software. It may also be combined with non-concurrent testing to detect or diagnose complex faults of all types.

A common method of providing hardware support for concurrent testing, especially for detecting software control errors and hardware residual design errors, is a watchdog timer [80]. This is a counter that must be reset by the system periodically to indicate that the system in question is functioning properly. A watchdog timer is based on the assumption that the system is fault-free—or at least alive—if it is able to perform the simple function of resetting the timer at appropriate intervals. Proper system sequencing can be monitored with very high precision by combining watchdog timer reset operations with various software checks. More complex hardware watchdogs can be constructed that implement these software checks in hardware [82].

A key element of concurrent testing for data errors is redundancy. For example, *duplication with comparison* (DWC) [64] can detect any single error at the expense of 100% hardware redundancy. In many applications, this high degree of hardware overhead is unacceptable due to its impact on weight, cost, and power consumption. Moreover, it is difficult to prevent minor variations in timing between duplicated modules from invalidating comparisons. A possible lower-cost alternative is time redundancy. For example, *recomputing with shifted operands* (RESO) [97] achieves almost the same error coverage of DWC with 100% time redundancy but very little hardware redundancy. Testing techniques based on time redundancy have been proposed for regular circuits such as iterative logic arrays and trees [64]. However, their usefulness in on-line testing for general logic circuits has not been demonstrated. A third form of redundancy which is very widely used is information redundancy, that is, the addition of redundant information such as a parity check bit to form error detecting codes [64]. Such codes are particularly effective for detecting memory and data transmission errors, since memories and networks are susceptible to transient errors. Coding methods are also widely used to detect errors in data computed during critical operations.

As noted above, for critical or highly available systems, it is desirable to have a comprehensive approach to on-line testing that covers all expected permanent, intermittent, and transient faults. In recent years, BIST [4] has emerged as an important method for testing manufacturing faults, and it is increasingly promoted for on-line testing as well. BIST is a design-for-testability technique that places the testing functions in the CUT, including test pattern generation, response compaction, response analysis, and test control. On-line BIST targets residual design errors/faults, i.e. errors that escape detection in the design phase, and physical faults arising during the normal operation of the system. Testing is thus performed concurrently to detect faults as soon as they occur. For on-line BIST to be feasible, we usually want to design hardware test generators that provide complete coverage of the modeled faults, low hardware overhead, and short elapsed time between the occurrence of a fault and its detection.

1.5 Lifetime Validation

The methodology of manufacture testing is fairly well developed and understood. Fabrication faults are first represented by well-defined fault models. Then automatic test pattern generation (ATPG) and simulation techniques are used to generate tests for the modeled faults. Finally, the tests are applied to the circuit under test and the resulting responses are compared with those of the specification to determine if the manufactured chip is fault-free. The similarity between design verification, manufacture testing, and on-line testing, as illustrated in Figure 1.9, suggests that design errors and operational faults can be modeled in a similar way to fabrication faults. Hence, if we combine the design development, manufacturing, and field operation phases of system lifetime and develop a common approach to testing and verification, we achieve a comprehensive verification approach that we call “lifetime validation”. This approach will, in principle, systematically detect all types of faults that arise during the lifetime of a system, as suggested in Figure 1.9.

Safety-critical systems, such as some automotive controllers, attempt either to avoid failures completely or else to detect them fast enough to prevent system crashes. Hence,

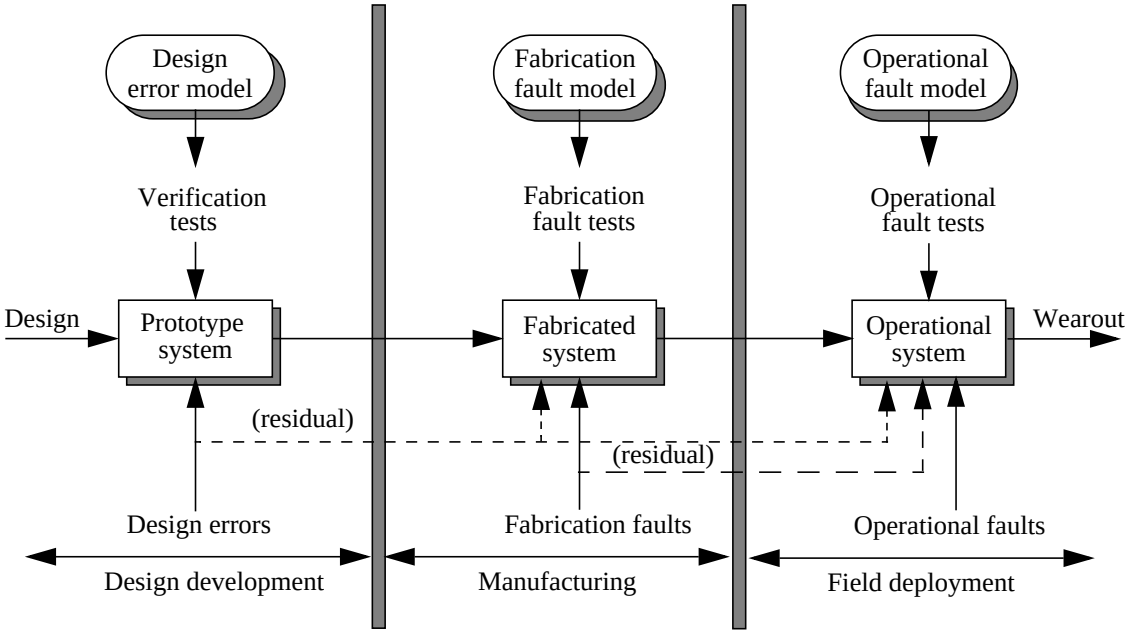


Figure 1.9 Relation between design verification, manufacture testing, and on-line testing.

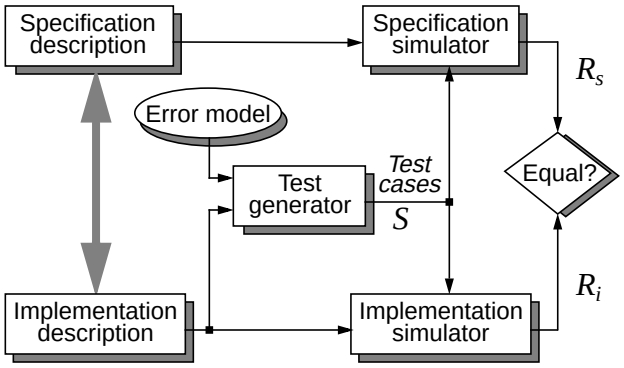


Figure 1.10 Block diagram of the proposed design verification method.

freedom from design errors is a primary goal where high-quality verification is required in the design development phase. However, with the current shorter design cycles and the increased complexity of digital systems, leakage of design errors from the design phase to the operational system is anticipated. Hence, a second goal in safety-critical design is to detect operational faults and respond to them. The combined verification/testing approach that we are proposing can deal with errors in both the development and operational phases. To detect and respond to faults and errors, we may need to carry out the combined tests on-line or concurrently. Hence, we also need to investigate on-line testing and develop built-in hardware test generators.

Our lifetime validation approach is thus based on the following sequence of steps: (1) explicit error and fault modeling, (2) model-directed test generation, and (3) test application. For the case of design verification, we employ software (simulatable) models for both the implementation and specification. A block diagram of the design verification methodology is shown in Figure 1.10. Error models are developed and then used to guide test generation. The resulting test sequence S is applied to the simulatable models of both the implementation and the specification to produce the outcomes R_s and R_i, respectively. A discrepancy between R_s and R_i indicates an error, either in the implementation or in the specification. We generate design verification tests targeting a set of design error models using conventional automatic test pattern generation techniques for physical (fabrication) faults. An important advantage of this approach is that it produces a small set of test vectors that can reveal possible design errors.

For the case of manufacture testing, the implementation is a single-chip SOC. The test sequence S and the response R_s are either supplied by an external tester or generated by hardware within the chip; the latter case corresponds to BIST. The sequence S is applied to the inputs of the CUT and the corresponding response R_i is compared against R_s to detect physical faults. On-line testing for residual design errors and physical faults can take advantage of the BIST hardware used for manufacture testing. On-line BIST, however, requires additional control hardware so that the actual testing is performed in a periodic fashion to detect transient and intermittent operational faults. We investigate this topic in Chapter 4.

1.6 Thesis Outline

This thesis develops a systematic approach to lifetime verification of digital systems with stringent safety and availability requirements. The approach aims to use testing and simulation techniques to improve the quality of error detection throughout the lifetime of a digital system.

In this chapter, we divided the lifetime of a digital system into three phases: design, manufacturing, and operation. We also identified the types of faults that arise during these phases and discussed the abstraction of fault effects into fault models. We also discussed the methods for detecting all types of faults, and proposed a lifetime validation methodology that targets the faults using manufacture testing and simulation methods.

Chapter 2 discusses our results on design verification for gate-level circuits. We present a simulation-based design verification method that uses conventional ATPG techniques for fabrication faults to generate the verification tests. We present an extensive study of the design error models at the gate level and analyze their detection requirements. We show how to systematically map the modeled design errors into SSL faults, and present experimental data showing that the verification test sets generated are small in size and have high coverage of the modeled errors.

Chapter 3 presents a design verification methodology that extends our gate-level validation method to high-level designs. We show how actual error data can be gathered and how design error models suitable for design verification testing can be derived. We

present experiments that indicate that high coverage of actual design errors is achieved with test sets that are complete for a small number of synthetic error models. We also present a new error model for microprocessors and a validation approach that uses it.

In Chapter 4, we examine built-in validation where test generation and application occurs within the CUT. We explore the design of efficient test sets and test-pattern generators for BIST with the target applications being high-performance, scalable datapath circuits for which fast and complete fault coverage is required. We show how to apply our technique to various datapath circuits including a carry-lookahead adder, an arithmetic-logic unit, and a multiplier-adder.

Chapter 5 summarizes the research contributions of this thesis and discusses future research directions.

CHAPTER 2
GATE-LEVEL DESIGN VALIDATION

Manufacture testing for fabrication faults is well understood. Fabrication fault models, such as the SSL model, have been extensively studied and validated. Moreover, excellent automatic test pattern generation (ATPG) tools have been developed. As discussed in Chapter 1, the similarity between manufacture testing and design verification suggests that manufacture-testing techniques can be adapted to model-based design validation. For this purpose, we need to evaluate and improve the existing design error models and develop ATPG methods to detect them. This chapter investigates an automated model-based design verification scheme for gate-level logic circuits that borrows methods from simulation and test generation for fabrication faults, and verifies a circuit with respect to a modeled set of design errors. The next chapter extends this approach to high-level design validation.

In Section 2.1, we examine the previously proposed design error models, and reduce them to five types. Then we study in detail the detection requirements of these error types. Section 2.2 describes the mapping of design errors into SSL faults, as well as the process of generating tests for them using standard test generation and simulation tools for SSL faults. Section 2.3 presents the results of applying our method to representative combinational and sequential benchmark circuits. Finally, Section 2.4 summarizes the contributions of this chapter.

2.1 Tests for Design Errors

Many types of design errors affecting logic circuits are identified in the research literature [1][2][36][65]. These error types are not necessarily complete, but they are believed

to be common in both manual and automated logic synthesis. We condense the errors identified by Abadir et al. [2] into four categories. (A similar classification is given independently in [36]). We also add a fifth category for sequential circuits.

- *Gate substitution error (GSE):* This refers to mistakenly replacing a gate by another gate with the same number of inputs. The extra and missing inverter errors of [1][2][36][65] are considered as substitution of an inverter for a buffer, and a buffer for an inverter, respectively.
- *Gate count error (GCE):* This corresponds to incorrectly adding or removing a gate, and includes the extra and missing gate errors of [2]. This category is combined with gate substitution in [36], where, unlike here, XOR and XNOR gates are not considered. A class of "local" errors is defined in [65] which includes only some of the errors in this category.
- *Input count error (ICE):* This corresponds to using a gate with more or fewer inputs than required.
- *Wrong input error (WIE):* This error corresponds to connecting a gate input to a wrong signal. The "signal-like-source" error [65], is a special case of WIE. Although a WIE may be viewed as a multiple ICE, a multiple ICE cannot model a WIE in an inverter.

We further identify the following error model for sequential circuits:

- *Latch count error* (*LCE*): This error occurs when a latch is incorrectly added or omitted, due to human error or using imperfect CAD tools for synthesis or (re) timing analysis.

The errors in each category are studied next, and the tests needed to detect them are determined. The following assumptions are made concerning the design to be verified:

- A gate-level implementation is available that is either combinational or synchronous sequential.
- The gate types used are AND, OR, XOR, NAND, NOR, XNOR, BUF (buffer) and NOT.
- As in [2][36][65], a functional specification of the design is available which is

completely simulatable, that is, any input pattern (sequence) can be applied and produces a completely specified output pattern (sequence).

- At most one design error is present. This assumption is made in the standard SSL model and, indeed, in most other models used in testing for fabrication faults.

Notation. Let E be the set of all 2^n input vectors of an n-input gate G. We divide E into the disjoint subsets V_0, V_1,..., V_n, where V_k contains all input vectors with exactly k 1s in their binary representation, $0 \le k \le n$. Particularly useful are the disjoint sets V_{null}, V_{all}, V_{odd}, and V_{even} defined as follows:

$$V_{null} = V_0 \qquad V_{all} = V_n$$

$$V_{odd} = \bigcup_{i = odd \wedge i \neq n} V_i \qquad V_{even} = \bigcup_{i = even \wedge i \neq 0 \wedge i \neq n} V_i$$

For example, in the case of 3-input NAND gate, V_{null} = {000}, V_{all} = {111}, V_{odd} = {001,010,100}, and V_{even} = {011, 101, 110}. We call V_{null}, V_{all}, V_{odd}, and V_{even} the *characterizing sets* or *C-sets* of G.

Table 2.1 shows the output responses of each gate type to its various C-sets. The sets V_{null} and V_{all} are nonempty and always have cardinality one. For the single-input gates, V_{even} and V_{odd} are empty. For multiple-input gates, the set V_{odd} contains at least two elements, while the set V_{even} is empty only when $n = 2$. The cardinality of V_{even} (V_{odd}) is $2^{n-1} - 1$ $(2^{n-1} - 1)$ when n is odd, and $2^{n-1} - 2$ (2^{n-1}) when n is even. Finally, v_k denotes an arbitrary vector of the set V_k.

The above notation enables us to express sets of vectors in a concise way. For example, the complete test set for SSL faults in an n-input NAND gate is $V_n \cup V_{n-1}$. When

Table 2.1 Responses of the various gate types to their C-sets.

C-set	$n = 1$		n even (n odd and $n \ge 3$)					
	NOT	BUF	AND	NAND	OR	NOR	XOR	XNOR
V_{null}	1	0	0 (0)	1 (1)	0 (0)	1 (1)	0 (0)	1 (1)
V_{even}	n/a	n/a	0 (0)	1 (1)	1 (1)	0 (0)	0 (0)	1 (1)
V_{odd}	n/a	n/a	0 (0)	1 (1)	1 (1)	0 (0)	1 (1)	0 (0)
V_{all}	0	1	1 (1)	0 (0)	1 (1)	0 (0)	0 (1)	1 (0)

$n = 3$, we can also write these tests as $V_{all} \cup V_{even} = \{111, 011, 101, 110\}$. In general, to verify the identity of a gate *G*, that is, to determine the tests required for its verification, we use the above notation in conjunction with Table 2.1.

Gate Substitution Errors (GSEs). According to experiments reported in [1], the most frequent error made by human designers is gate substitution, accounting for around 67% of all errors. Gate substitution refers to mistakenly replacing a gate symbol (in a schematic diagram) or a gate operator (in an HDL description) *G* with another gate *G'* that has the same number of inputs. We represent this error by G/G'. For gates with multiple inputs, a *multiple-input GSE* (*MIGSE*) can have one of six possible forms: G/AND, G/NAND, G/OR, G/NOR, G/XOR, and G/XNOR. Each multiple-input gate can have five MIGSEs. For example, all MIGSEs can occur on an AND gate except G/AND, which is not considered an error. For gates with a single input, i.e., buffers and inverters, a *single-input GSE* (*SIGSE*) can have one of two possible forms: G/NOT and G/BUF. Each single-input gate can have only one SIGSE. To cover extra or missing inverters in GSEs, a buffer can be inserted in each of a gate's fanout branches as well as in inputs that fan out.

It has been suggested that most GSEs can be detected by a complete test set for SSL faults [2]. Our simulation study (Section 2.3) shows that such a test set can cover 80% to 100% of MIGSEs and 100% of SIGSEs. The actual coverage of MIGSEs is a function of the circuit structure, as well as the types of gates used in the circuit. Our goal here is to achieve 100% coverage for GSEs.

A single-input gate can be identified by one test vector from either V_{null} or V_{all}. On the other hand, a multiple-input gate can be identified by three test vectors: one from V_{null}, one from V_{odd}, and one from V_{all} (if *n* is even) or V_{even} (if *n* is odd). Hence, three test vectors are sufficient to identify an *n*-input gate. Two test vectors suffice in some cases. For example, an AND gate can be identified by applying one test vector from V_{null} and one from V_{odd}.

The number of tests needed to test an *n*-input gate for SSL faults is $n + 1$ for the gates AND, NAND, OR, and NOR, while it is two or three for XOR and XNOR depending on the parity of *n*. So, the number of tests needed to test for SSL faults is greater than or equal to the number of tests needed to test for MIGSEs in most cases.

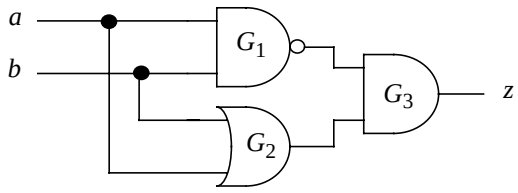


Figure 2.1 Circuit realizing the XOR function.

We now introduce some notation to specify the effects of C-sets on a gate within a circuit. If the inputs of gate G in circuit C can be forced to the pattern v by assigning the primary inputs of C, then G is *controllable* by v; otherwise, it is *uncontrollable* by v. If the output of G with respect to the pattern v is sensitizable to a primary output then the response of v is said to be *observable* at G; otherwise, it is *unobservable*. A gate G is *V-controllable* if G is controllable by at least one vector v in the input vector set V. If v is also observable at G, then G is *excitable* by V (*V-excitable*). A gate G is *fully excitable* if G is excitable by every nonempty C-set of G; otherwise, it is *partially excitable*.

To illustrate these definitions, consider the circuit in Figure 2.1. G_1 and G_2 are both controllable by the pattern 00, while G_3 is uncontrollable by 00. The response to the pattern 00 is observable at G_2 but it is not observable at G_1. The gate G_3 is {00,11}-excitable because G_3 is controllable by 11, and the response of 11 is observable at G_3. However, G_3 is not {00}-excitable because G_3 is uncontrollable by all the elements of the set {00}. The gates G_1, G_2, and G_3 are partially excitable.

The following theorem solves the verification problem for GSEs:

Theorem 2.1 *A necessary and sufficient condition for a test set S to verify a fully excitable gate is that S produce the test vectors T_1 shown in Table 2.2 at the inputs of the gate and sensitize the gate output to a primary output.*

Proof: We prove the case for an AND gate with an odd number of inputs only; the other cases can be proved similarly. Sufficiency follows directly from Table 2.2. To prove necessity, assume that S verifies the AND gate G but does not produce either $\{v_{all}, v_{odd}\}$ or $\{v_{null}, v_{odd}\}$ at G's inputs. It is clear from Table 2.2 that there is no single vector capable of verifying an AND gate. Hence, the set S produces one of the following sets at the inputs of

Table 2.2 The test vectors required to verify an *n*-input gate.

Gate	Fanin *n*	Test set T_1	Test set T_2	Test set T_3
NOT (BUF)	$n = 1$	$\{v_{all}\}$ or $\{v_{null}\}$	$\{v_{all}, v_{null}\}$	$\{v_{all}, v_{null}\}$
AND (NAND)	$n = 2$	$\{v_{null}, v_{odd}\}$	$\{v_{all}, v_{odd}, v_{null}\}$	$\{v_{all}, v_{null}, v_{odd}\}$
	n odd	$\{v_{all}, v_{odd}\}$ or $\{v_{null}, v_{odd}\}$	$\{v_{all}, v_{odd}, v_{null}\}$	$\{v_{null}, v_{odd}, v_{all}, v_{even}\}$
	n even & $n \neq 2$	$\{v_{null}, v_{odd}\}$ or $\{v_{all}, v_{even}\}$	$\{v_{null}, v_{odd}, v_{all}, v_{even}\}$	$\{v_{null}, v_{odd}, v_{all}, v_{even}\}$
OR (NOR)	$n = 2$	$\{v_{all}, v_{odd}\}$	$\{v_{all}, v_{odd}, v_{null}\}$	$\{v_{all}, v_{null}, v_{odd}\}$
	n odd	$\{v_{null}, v_{even}\}$ or $\{v_{all}, v_{even}\}$	$\{v_{null}, v_{even}, v_{all}\}$	$\{v_{null}, v_{odd}, v_{all}, v_{even}\}$
	n even & $n \neq 2$	$\{v_{null}, v_{even}\}$ or $\{v_{all}, v_{odd}\}$	$\{v_{null}, v_{even}, v_{all}, v_{odd}\}$	$\{v_{null}, v_{odd}, v_{all}, v_{even}\}$
XOR (XNOR)	$n = 2$	$\{v_{null}, v_{all}\}$	$\{v_{all}, v_{odd}, v_{null}\}$	$\{v_{all}, v_{null}, v_{odd}\}$
	n odd	$\{v_{odd}, v_{even}\}$	$\{v_{odd}, v_{even}, v_{all}\}$ or $\{v_{odd}, v_{even}, v_{null}\}$	$\{v_{null}, v_{odd}, v_{all}, v_{even}\}$
	n even & $n \neq 2$	$\{v_{null}, v_{all}\}$ or $\{v_{even}, v_{odd}\}$	$\{v_{null}, v_{all}, v_{even}, v_{odd}\}$	$\{v_{null}, v_{odd}, v_{all}, v_{even}\}$

G: $\{v_{null}, v_{all}\}$, $\{v_{null}, v_{even}\}$, $\{v_{even}, v_{all}\}$, $\{v_{odd}, v_{even}\}$, or $\{v_{null}, v_{even}, v_{all}\}$. Table 2.1 shows that none of these are capable of verifying the AND gate. Hence S must produce $\{v_{all}, v_{odd}\}$ or $\{v_{null}, v_{odd}\}$ at G's inputs. Note that the above analysis implies that two test vectors are sufficient to verify a fully excitable gate. □

All gates in a fanout-free circuit are fully excitable. In a circuit with fanout, it is possible that some input combinations cannot be forced at the inputs of some gates. For example, no element of V_{null} can be forced at the inputs of the AND gate G_3 in Figure 2.1. From Table 2.1 we see that V_{null} is necessary to distinguish a 2-input AND gate from an XNOR gate, so, the replacement of the AND by an XNOR gate cannot be detected. This replacement does not change the function of the circuit, hence it is considered to be an *undetectable* MIGSE. Likewise, some input combinations can be forced at the inputs of some gates but their responses cannot be observed. For example, the pattern 00 can be forced at the inputs of G_1 in Figure 2.1, but the response of G_1 cannot be propagated to the primary output. The above examples show that it is natural to have gates which are not fully excitable and therefore have undetectable design errors. It also suggests a modification of the test

vectors T_1 in Table 2.2 to verify a partially excitable gate.

If a partially excitable gate G is excitable by all but one of its nonempty C-sets, then G is called *strong partially excitable*, otherwise, it is called *weak partially excitable*. To illustrate, consider again the circuit in Figure 2.1. The gates G_1, G_2, and G_3 are strong partially excitable because they are excitable by two out of the three nonempty C-sets of the respective gates. An example of a weak partially excitable gate is a 3-input XOR with all inputs connected to a single source. In this case, the gate is excitable by only two (V_{null} and V_{all}) of its four C-sets.

Since a strong partially excitable gate G is not excitable by one of the nonempty C-sets, one of its MIGSEs is undetectable. The remaining four MIGSEs on G can be detected with at least two vectors; Table 2.1 implies that an arbitrary vector detects only three of G's five MIGSEs. Therefore, we have to apply at least three test vectors to G, so that if G is not controllable by one of the vectors or one of the vectors' responses is not observable, then the other two will detect the detectable MIGSEs. This leads to the following result.

Theorem 2.2 *If all gates of a circuit are either fully excitable or strong partially excitable, then the test set T_2 shown in Table 2.2 detects all detectable GSEs in the circuit.*

Proof: If a gate G is fully excitable, then G is controllable by the test vectors in T_2 and their responses are observable at G. Since each test set in T_2 for a particular gate is a superset of the test set in T_1 for the same gate, all GSEs will be detected. If, on the other hand, G is strong partially excitable, then it is not controllable by one of the test vectors in T_2, or the response of one of the vectors is not observable at G. It follows from Table 2.1 that if we remove a test vector for G from T_2, then the remaining vectors detect all the detectable GSEs on that gate. □

A further analysis of T_2 shows that to verify a weak partially excitable gate, we have to apply the patterns T_3 shown in Table 2.2. Since we cannot always assert that the gates in the design under test are fully excitable or strong partially excitable, we may have to apply the patterns T_3 to detect all GSEs. Note that a test set generated for GSEs assuming that the gates are weak partially excitable, will detect all GSEs in the circuit. On the other hand, a test set generated for GSEs by assuming the gates are fully excitable or strong partially

excitable may not detect all GSEs.

A complete test set for SSL faults guarantees the detection of all SIGSEs [2]. Tests for MIGSEs also cover many SIGSEs. A complete test set T for MIGSEs in an SSL-irredundant circuit, i.e. a one with no undetectable SSL faults, is also a complete test set for SIGSEs on all circuit lines except inputs with fanout, if T produces v_{all} at the input of every AND and NAND gate, v_{null} at the input of every OR and NOR gate, and their responses are observable. From this result we conclude that detection of most SIGSEs is ensured by test sets T_2 and T_3, but not by T_1. Our experiments show that the test set T_3 detects all SIGSEs in all SSL-irredundant benchmark circuits considered in Section 2.3.

Gate Count Errors (GCEs). We distinguish two types of gate count errors: extra-gate and missing-gate errors. An *extra-gate design error* (EGE) is defined as inserting a gate *G'* that has its *m* inputs taken from the *n* inputs of a gate *G* and feeding the output of *G'* to *G*. As a consequence, the number of inputs of gate *G* becomes $n - m + 1$. We represent an EGE by EG(*G'*,*G*). It is easily seen that EG(AND,AND), EG(AND,NAND), EG(OR,OR), EG(OR,NOR), EG(XOR,XOR), and EG(XOR, XNOR) are undetectable. Explicit test generation for EGEs is not needed due to the following result.

Theorem 2.3 *A complete test set for GSEs is also a complete test set for EGEs.*

Proof: An EGE can be mapped easily into a GSE. EG(*G'*,*G*) is nothing but the GSE *G"*/*G'*, where *G"* is determined by *G* as follows: (1) if *G* is an AND or NAND, then *G"* is an AND; (2) if *G* is an OR or NOR, then *G"* is an OR; (3) if *G* is an XOR or XNOR, then *G"* is an XOR. Hence, any test set that detect all GSEs will detect all EGEs. □

Most, but not all, EGEs can also be detected by a complete test set for SSL faults. A complete test set for SSL faults in the circuit of Figure 2.2 is {000, 100, 001, 010}. This test set does not detect if the XOR gate is an extra gate. For that, we need the test 011.

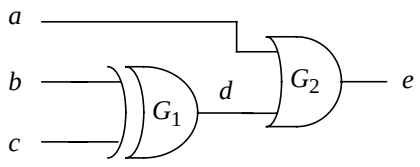


Figure 2.2 Example showing an EGE that is not detected by a complete test set for SSL faults.

Figure 2.3 The missing-gate design error (MGE).

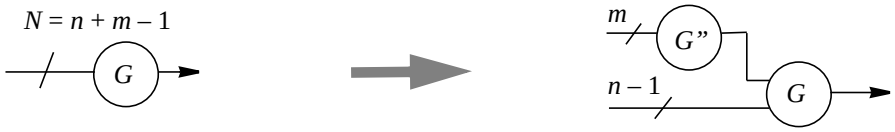


Figure 2.4 Reducing the problem of detecting MGEs to detecting GSEs.

A *missing-gate design error* (MGE) is defined as removing a gate *G'* that has *m* inputs and feeds an *n*-input gate *G*, and then changing the inputs of *G'* into inputs of *G*; see Figure 2.3. As a consequence, the number of inputs of *G* becomes $N = n + m - 1$. We represent the MGE by MG(*G'*,*G*). As in the extra-gate case, the errors MG(AND, AND), MG(AND, NAND), MG(OR, OR), MG(OR, NOR), MG(XOR, XOR), and MG(XOR, XNOR) are undetectable.

Consider the problem of finding a minimal set of vectors that detect all MGEs in an *N*-input gate *G*. For each MG(*G'*,*G*), we insert a gate *G"* as shown in Figure 2.4, where *G"* is chosen so that the function of the circuit is not changed. For example, if *G* is an AND or NAND, then *G"* is an AND gate. We have to detect the GSE G"/G' in order to detect MG(*G'*,*G*).

Theorem 2.4 *The test sets* $V_N \cup V_{N-1} \cup V_{N-2}$, $V_0 \cup V_1 \cup V_2$, *and* $V_0 \cup V_2 \cup V_N$ *are each sufficient and near-minimal for detecting MGEs on an N-input fully excitable AND (or NAND), OR (or NOR), and XOR (or XNOR) respectively.*

Proof: We prove the NAND case; the proofs for the other cases are similar. First, we prove that V_{N-2} is a subset of every test set detecting all MGEs on an *N*-input NAND gate. Since *G* is a NAND gate, *G"* of Figure 2.4 is an AND gate. To detect gate substitutions for the 2-input AND gate *G"*, we need to apply the vector v_{null} at its inputs. Since *G"* can have any two inputs of *G*, we have to apply 00 to any combination of two inputs of *G*. Since the other

inputs of *G* must be 1s to propagate the output signal of *G"* to the output of *G*, all the vectors that contain two 0s must be applied. Hence, V_{N-2} must be applied to *G*.

Second, we need to prove that at least $N-1$ of the *N* vectors of V_{N-1} must belong to every test set detecting all MGEs on an *N*-input NAND gate. To detect the GSEs of the 2-input AND gate *G"*, we need to apply the vector v_{odd} to it. So, we have to apply 01 or 10 to any two inputs of *G*. Since the other inputs of *G* must be 1s to propagate the output signal of *G"* to the output of *G*, the vectors that have one 0 must be applied. It remains to prove that we need at least $N-1$ vectors of V_{N-1}. An arbitrary vector of V_{N-1} will ensure the presence of v_{odd} in $N-1$ of the $\binom{N}{2}$ configurations. Another vector will ensure the presence of v_{odd} in another $N-2$ configurations. Continuing with this way, we find that the one before the last vector will ensure v_{odd} in the last configuration. Since it is irrelevant which $N-1$ vectors are used, and since a near-minimal test set is acceptable, we will simply employ all the *N* vectors of V_{N-1}.We conclude that the test set $V_{N-1} \cup V_{N-2}$ ensures the detection of all 2-input MGEs on an *N*-input NAND gate.

Third, we must prove that the tests $V_{N-1} \cup V_{N-2}$ ensure the existence of v_{odd} and v_{even} for any *m*-input gate *G"* according to Figure 2.4. If *m* is odd, then there are *m* v_{odd} vectors for *G"* in V_{N-2} and $\binom{m}{2}$ v_{even} vectors for *G"* in V_{N-1}. On the other hand, if *m* is even, then there are *m* v_{odd} vectors for *G"* in V_{N-1} and $\binom{m}{2}$ v_{even} vectors for *G"* in V_{N-2}.

Finally, we need to prove that a minimal test set for all *m*-input MGEs ($m > 2$) in an *N*-input NAND gate must include v_{all}. Table 2.2 implies that to detect the gate substitution on *G"*, we need the set $\{v_{all}, v_{odd}\}$ or $\{v_{null}, v_{odd}\}$ when *m* is odd, and the set $\{v_{null}, v_{odd}\}$ or $\{v_{all}, v_{even}\}$ when *m is* even and $m \neq 2$. Since $\{v_{odd}, v_{even}\}$ is guaranteed, then we need only either v_{all} or v_{null} on *G"*. Forcing v_{all} on *G"* requires just forcing v_{all} on *G*. On the other hand, forcing v_{null} on *G"* requires forcing $5 \times \sum_{i=3}^{N-1} \binom{N}{i}$ vectors on *G*. Hence, v_{all} must be selected to minimize the test set size. □

To keep the theorem simple, it is stated it in terms of a near-minimum number of tests. In fact, each test set defined by this theorem has one test more than the minimum. For example, the 11-member test set generated for MGEs in a 4-input NAND gate *G* is *S* = {1111, 1110, 1101, 1011, 0111, 1100, 1010, 1001, 0110, 0101, 0011}. If one of the tests

{1110, 1101, 1011, 0111} is dropped, *S* still detects all MGEs. However, all MGEs in *G* cannot be detected with fewer than 10 vectors. In general, Theorem 2.4 gives near-minimal test sets for an *N*-input fully excitable gate. It is easy to prove that these test sets detect all the MGEs of an *N*-input partially excitable gate with high probability.

Input Count Errors (ICEs) and Wrong Input Errors (WIEs). Input count errors (ICEs) are classified into extra input and missing input errors. An *extra input design error* (EIE) is defined as the replacement of an *n*-input gate ($n \geq 2$) by an ($n + 1$)-input gate with the additional input connected to an arbitrary signal in the circuit. A *missing input design error* (MIE) is the replacement of a gate of ($n \geq 3$) inputs by an ($n - 1$)-input gate whose $n - 1$ inputs are connected to an arbitrary subset of the original *n*. We represent an EIE of a gate *G* by EI(*e*,*G*) where *e* is the extra input. We represent an MIE of a gate *G* by MI(*m*,*G*) where *m* is the source of the missing input.

To detect an EIE at a given input of an AND or NAND gate, that input must be set to 0 to activate the error, the other inputs must be forced to 1, and the gate's output signal must be propagated to a primary output. This is exactly the requirement of a test for a stuck-at-1 fault at the input of the gate in question. Similarly, testing for EIEs at input *x* of an OR or NOR gate is the same as testing for *x* stuck-at-0. To test for an MIE on an AND gate *G*, the inputs of *G* are set to 1, the signal considered to be missing is set to 0, and *G*'s output signal is propagated to a primary output. This is more restrictive than a test for stuck-at-0 at the output of *G*. Similarly, testing for an MIE on a NAND, OR, and NOR is more restrictive than testing the gate output for stuck-at-1, stuck-at-1, and stuck-at-0, respectively.

The foregoing tests are complete for AND, NAND, OR, and NOR gates. Hence, a complete test set for ICEs in a given circuit detects all SSL faults at AND, NAND, OR, and NOR gates. A complete test set for ICEs also detects some SSL faults affecting XOR and XNOR gates. For example, testing for EIEs at the input of an XOR or XNOR gate is equivalent to testing for stuck-at-0 fault at the same input.

A *wrong input error* (WIE) is defined as connecting a gate input to a wrong signal source. We represent a WIE on a gate *G* by WI(*u*,*w*,*G*), where *u* is the wrong input of the gate and *w* is the correct input. If a test vector *v* detects WI(*u*,*w*,*G*), then it must set *u* and *w*

to opposite values and propagate the signal at u to a primary output. WIE appears to be the second most common design error—around 17% of the errors reported in [1]. The relationship between MIEs and WIEs is as follows: A complete test set for MIEs on gates of type AND, NAND, OR, or NOR is a complete test set for WIEs on the same gates. To prove this relationship, consider an AND gate G with inputs $x_1, x_2,..., x_n$ and output y. Let z be an arbitrary signal in the circuit. The complete test set for MIEs on G will detect MI(z,G) and hence set the inputs of the gate to 1s, propagate y to a primary output, and set z to 0 with at least one vector v of the test set. Since v sets x_i and z to opposite values, and propagates x_i to a primary output, WI(x_i,z,G) is detected for every i. A similar argument holds for the other gate types.

In practice, it is hard to find a complete test set for MIEs. The fact that a given MI(x,G) is undetectable does not imply that WI(u,x,G) is undetectable for every u. Also, a complete test set for MIEs does not guarantee the detection of WIEs in XOR, XNOR, NOT, and BUF gates. Hence, we cannot conclude that a test set for MIEs covers all WIEs.

The numbers of ICEs and WIEs in a circuit are large—approximately $O(k^2)$, where k is the number of distinct signals in the circuit. Hence, we use simulation to extract the errors detected by the test set $S_T = S_{SSL} \cup S_{GSE} \cup S_{MGE}$, where S_{SSL}, S_{GSE}, and S_{MGE} are complete test sets for SSL faults, GSEs, and MGEs, respectively. In fact, all EIEs are detected by the test set for SSL faults alone [2], hence, we only have to generate tests for the undetected MIEs and WIEs. Our experimental results show that most MIEs and WIEs are detected by the set S_T.

A basic question concerning MIEs (WIEs) is the source of the missing (wrong) input. It must not depend on the erroneous gate's output, otherwise, the circuit can become sequential and asynchronous. Errors that make a circuit sequential can be detected by a levelization procedure [4].

The coverage relationships among the various design errors are summarized as follows. A complete test set for MIGSEs detects all EGEs. On the other hand, a complete test set for SSL faults detects all EIEs and SIGSEs. Complete test sets for MIEs, MGEs, and WIEs do not guarantee the detection of other error types. For example, a test for MIEs

detects many, but not necessarily all, SSL faults.

Latch Count Errors (LCEs). Latch count errors (LCEs) are classified into extra and missing latch errors. We assume that all latches are of the clocked D type, synchronized by a common clock signal. An *extra latch design error* (*ELE*) is defined as the insertion of a latch into any line in the circuit. A *missing latch design error* (*MLE*) is the replacement of a latch by a line linking its data input and output terminals. It is impractical to consider all possible MLEs due to their impact on the circuit's state space and test generation complexity. Hence, we only consider MLEs affecting the circuit's primary inputs and outputs.

In contrast to the design errors studied in the previous subsections, to check for LCEs, a test sequence rather than an unordered set of test patterns is needed. To test for an ELE, a transition sequence, either $0 \rightarrow 1$ or $1 \rightarrow 0$, is applied at the input of the latch and its response is propagated to a primary output. Similarly, to test for an MLE, a transition sequence is applied at the line where the latch may be missing and the transition sequence is propagated to a primary output.

Design Error Undetectability. We noted earlier that some design errors are undetectable. This leads to a type of redundancy that is quite different from that previously studied [54].

A gate G in a circuit C has *redundant inputs* if the function implemented by C is not changed when a proper subset of the inputs of G are removed. A circuit C is called *GI-irredundant* if no gate in C has redundant inputs. GI-redundancy does not imply SSL-redundancy. For example, a 5-input XOR with all inputs connected to the same source is GI-redundant but SSL-irredundant. Similarly, SSL-redundancy does not imply GI-redundancy. For example, a buffer whose input is connected to ground is SSL-redundant but GI-irredundant.

An *undetectable design error* is one for which no test vector exists. For example, the substitution of an XNOR gate for G_3 in Figure 2.1 cannot be detected by any input vector. Hence, the MIGSE G_3/XNOR is undetectable. The following theorem characterizes undetectable GSEs:

Theorem 2.5 *In a GI-irredundant and SSL-irredundant circuit C, the following holds: (1) C has no undetectable SIGSEs; (2) If G/G' is an undetectable MIGSE then every other*

MIGSE on G is detectable, and if $G \in \{XOR, XNOR\}$ then $G' \in \{AND, NAND, OR, NOR\}$ and vice versa.

Proof: If there is an undetectable SIGSE G/G' in C, then the output of G is not sensitizable to a primary output. Hence, SSL faults cannot be detected at the output of G, consequently C is SSL-redundant. Therefore, if C is SSL-irredundant, then it must be free from undetectable SIGSEs. For the case of MIGSEs, let us consider a 2-input AND gate. Since the circuit is SSL-irredundant, the AND gate is excitable by the C-sets V_{all} and V_{odd}. This implies that the MIGSEs AND/NAND, AND/OR, AND/NOR, and AND/XOR are detectable. The only possible undetectable MIGSE is AND/XNOR, which requires that the AND gate not be excitable by V_{null}. Figure 2.1 shows an example of this redundant MIGSE. A similar analysis leads to the other possible undetectable MIGSEs shown in Table 2.3.

Although XOR and XNOR have two possible undetectable MIGSEs, only one undetectable MIGSE can be found in a gate in an SSL-irredundant and GI-irredundant circuit. Let us prove this for the case of an XOR gate G with odd number of inputs. Assume that G is in a circuit C in which both XOR/OR and XOR/AND are undetectable. This implies that G is only excitable by V_{all} and V_{null}. So, if two of the inputs to G are removed, G's output is not changed. Hence G has redundant inputs and the circuit C is not GI-irredundant. Therefore, only one undetectable MIGSE can be found on a gate for a GI-irredundant and SSL-irredundant circuit. The final part of Theorem 2.5 follows directly from Table 2.3. □

Table 2.3 Possible redundant MIGSEs on an *n*-input partially excitable gate.

Gate type	$n = 2$		n even and $n \neq 2$		n odd	
	Strong	**Weak**	**Strong**	**Weak**	**Strong**	**Weak**
AND	AND/XNOR	Impossible	Impossible	AND/XNOR	AND/XOR	AND/XOR
NAND	NAND/XOR	Impossible	Impossible	NAND/XOR	NAND/XNOR	NAND/XNOR
OR	OR/XOR	Impossible	Impossible	OR/XOR	OR/XOR	OR/XOR
NOR	NOR/XNOR	Impossible	Impossible	NOR/XNOR	NOR/XNOR	NOR/XNOR
XOR	XOR/OR or XOR/NAND	Impossible	Impossible	XOR/OR or XOR/NAND	XOR/OR or XOR/AND	XOR/OR or XOR/AND
XNOR	XNOR/NOR or XNOR/AND	Impossible	Impossible	XNOR/NOR or XNOR/AND	XNOR/NOR or XNOR/NAND	XNOR/NOR or XNOR/NAND

From the above theorem we can infer that if the gates in a GI-irredundant and SSL-irredundant circuit *C* are restricted to AND, NAND, OR, NOR, NOT, and BUF, then *C* contains no undetectable GSEs.

The number of gates that can have undetectable MIGSEs in a circuit *C* varies with the circuit structure and the types of gates in *C*. For example, fanout-free circuits have no undetectable GSEs. On the other hand, a 2-input XOR circuit implemented using four 2-input NAND gates has up to four undetectable MIGSEs: each NAND gate can be replaced with an XOR without affecting the overall XOR function.

2.2 Verification Test Generation

This section describes our method for modeling and detecting design errors. In order to use standard ATPG tools, we map the error types under consideration into SSL faults. The mapping process consists of modifying the target circuit's netlist (or equivalent description) and injecting a predefined set of SSL faults. A test set is then generated for these faults in the modified netlist which detects all errors in the original design.

To map MIGSEs and MGEs into SSL faults, each gate in the original netlist is replaced by a functionally equivalent circuit called a *gate replacement module*. A few selected SSL faults are injected in the gate replacement module, so that the test for each injected fault forces the input of the gate to be a vector from one of the sets required to verify the gate. To cover all possible MIGSEs in a circuit, we must assume that the gates are weak partially excitable. Consider, for example, the AND replacement module shown in Figure 2.5. The faults *c* stuck-at-0, *d* stuck-at-0, and *e* stuck-at-0 force the inputs of the AND replacement module to v_{null}, v_{odd}, and v_{all}, respectively. These input patterns determine if the AND gate in the circuit is correct or not, i.e., the presence of any MIGSE on the gate is detected.

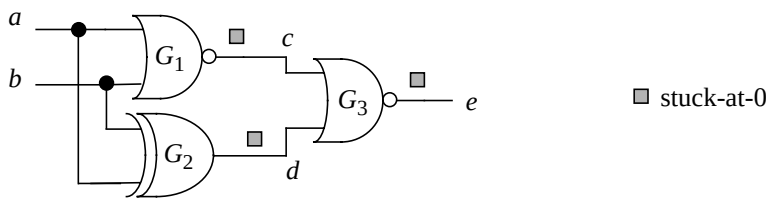


Figure 2.5 The replacement module for detecting GSEs in a 2-input AND gate.

The requirements to be met by a gate replacement module *M(G)* of a gate *G* are as follows:

- The function of *M(G)* must be the same as that of *G*.
- A test for an injected SSL fault in *M(G)* must force the input of *G* to a certain vector that is needed to verify *G*.
- The injected SSL faults must be sensitizable to the output of *M(G)*.
- If an injected SSL fault in M(G) is detected by a vector $v \in V_i$, then it must be detected by any vector of V_i. This requirement simplifies the detection of the injected SSL faults by the test generator, and leads to smaller test sets.

The gate replacement modules for MIGSEs and MGEs on all gate types can be designed systematically using the "detection signals" Y_{all}, Y_{odd}, Y_{even}, and Y_{null} that are shown in Figure 2.6. A *detection signal* Y_i is defined to be 1 if and only if the input pattern *v* belongs to the characterizing set V_i. Since the characterizing sets are disjoint, only one of the detection signals Y_{all}, Y_{odd}, Y_{even}, and Y_{null} can be 1 for a given *v*. A test for a stuck-at-0 fault at one of the detection signals will force the input of the gate to *v*. The functions of the gates in terms of the detection signals (Table 2.4) are used in designing the gate replacement modules. The equations of Table 2.4 can be simplified for the special case of 2-input gates when Y_{even} is always 0. For example, consider a 2-input AND gate whose gate replacement module is shown in Figure 2.5. From Table 2.4, $Y_{all} = \overline{Y}_{null}\overline{Y}_{odd}\overline{Y}_{even}$. Since $Y_{even} = 0$, then $Y_{all} = \overline{Y}_{null}\overline{Y}_{odd}$. The signals *c* and *d* in Figure 2.5 are Y_{null} and Y_{odd}, respectively, hence gate G_3 implements the equation $Y_{all} = \overline{Y}_{null}\overline{Y}_{odd}$.

Figure 2.7 shows the GSE replacement modules for a 2-input XOR and an *n*-input

Figure 2.6 Generation of the detection signals for an *n*-input gate.

Table 2.4 Equations for *n*-input gate replacement modules for GSEs.

Gate		Equation
Type	**Fanin *n***	
AND	--	$Y_{all} = \overline{Y}_{null}\overline{Y}_{odd}\overline{Y}_{even}$
NAND	--	$\overline{Y}_{all} = Y_{null} + Y_{odd} + Y_{even}$
OR	--	$\overline{Y}_{null} = Y_{all} + Y_{odd} + Y_{even}$
NOR	--	$Y_{null} = \overline{Y}_{all}\overline{Y}_{odd}\overline{Y}_{even}$
XOR	even	$Y_{odd} = \overline{Y}_{all}\overline{Y}_{null}\overline{Y}_{even}$
	odd	$Y_{odd} + Y_{all} = \overline{Y}_{null}\overline{Y}_{even}$
XNOR	even	$\overline{Y}_{odd} = Y_{all} + Y_{null} + Y_{even}$
	odd	$\overline{Y}_{odd}\,\overline{Y}_{all} = Y_{null} + Y_{even}$

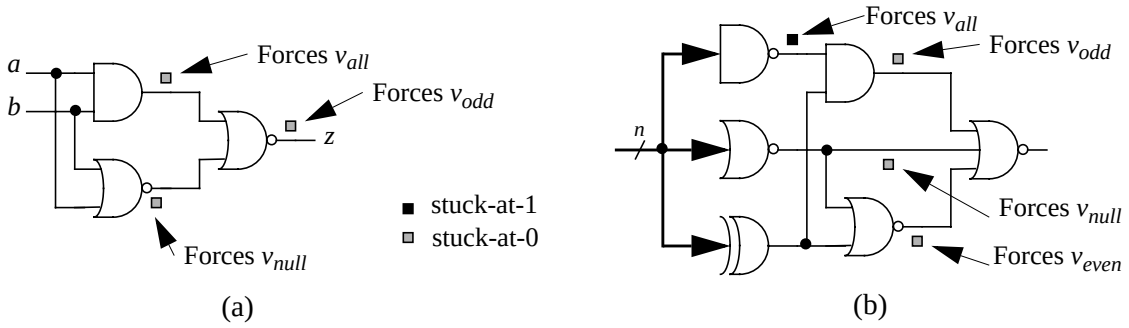


Figure 2.7 Gate replacement module for detecting GSEs in (a) a 2-input XOR and (b) an *n*-input AND (*n* odd).

AND (*n* odd). GSE replacement modules for the other gates can be constructed in a similar manner using Table 2.4. The gate replacement modules for MGEs can also be designed in a systematic way similar to that for GSEs. The gate replacement modules for MGEs are more complex due to the requirement of generating Y_2 and Y_{N-2} signals that detect the presence of v_2 and v_{N-2} at the inputs. Figure 2.8 shows the MGE replacement module for a 3-input AND gate.

The mapping of MIGSEs and MGEs into SSL faults is many-to-one. Detecting a given set of injected SSL faults detects a larger set of MIGSEs and MGEs. For example, detection of the three SSL faults in Figure 2.5 detects five MIGSEs. There is a one-to-one correspondence between net errors (EIEs, MIEs, and WIEs) and SSL faults. The mapping of an EIE into an SSL fault is very simple: to detect whether an AND or NAND gate's input *x* is extra, we need to set *x* to 0, set every other input to 1, and propagate the gate's output

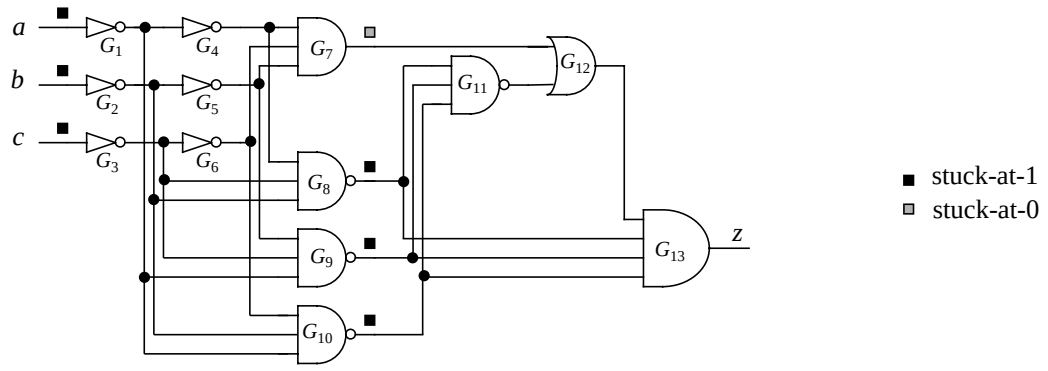


Figure 2.8 Gate replacement module for detecting MGEs in a 3-input AND gate.

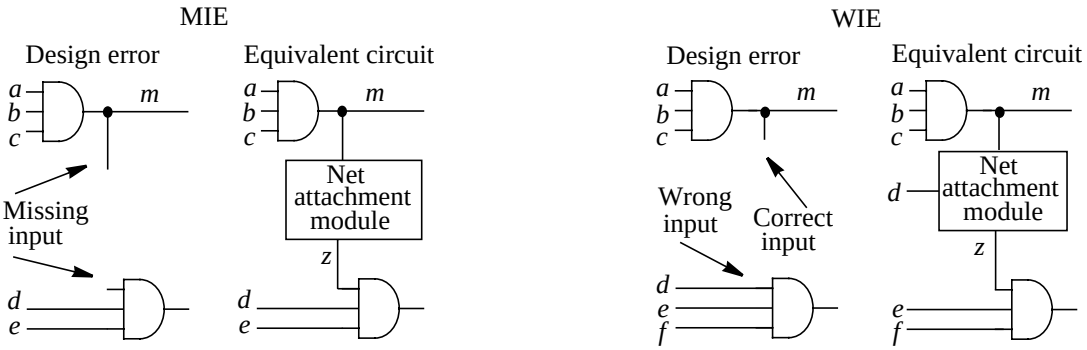


Figure 2.9 Mapping MIEs and WIEs into SSL faults.

signal to a primary output. This is the same as testing for *x* stuck-at-1. Also, to test for an extra input in an OR, NOR, XOR, or XNOR gate, a test for the input stuck-at-1 is required.

The detection of MIEs and WIEs is modeled by a mapping circuit called a *net attachment module*, as shown in Figure 2.9. Let *C* and *C'* be the circuits obtained before and after adding the net attachment module. The following requirements must be met:

- The function of circuit *C* must be the same as that of *C'*.
- A test for the injected SSL fault in the net attachment module must detect the MIE or WIE.
- The injected SSL fault must be sensitizable in the net attachment module.

A typical design of a net attachment module for MIEs appears in Figure 2.10a. If G_2 is an AND or NAND, then G_1 must be an XNOR and the fault *p* stuck-at-1 is injected. On the other hand, if G_2 is any of the gates {OR, NOR, XOR, XNOR}, then G_1 must be an

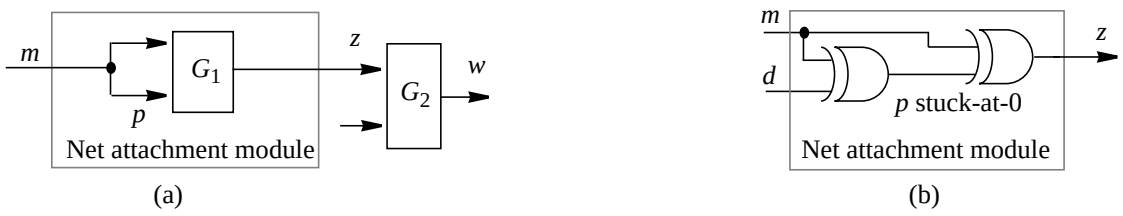


Figure 2.10 A net attachment module (a) for MIEs and (b) for WIEs.

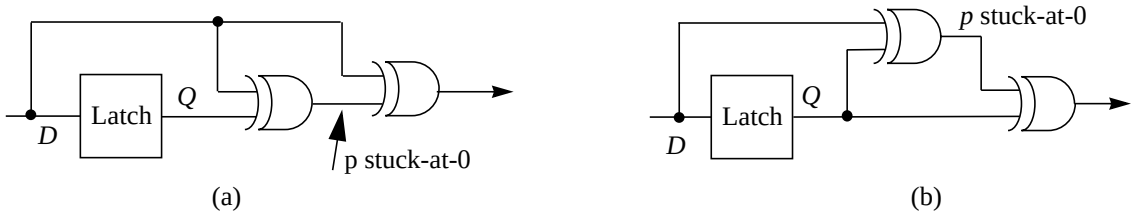


Figure 2.11 (a) Latch and (b) line replacement modules to detect ELEs and MLEs, respectively.

XOR and the fault *p* stuck-at-0 is injected. In both cases, the output of G_2 is independent of *z* and hence the function of the circuit is not changed. Also, the SSL fault is sensitizable to the output of the net attachment module and the vector testing it detects MI(m,G_2). A typical design of the net attachment module for a WIE is shown in Figure 2.10b. The output of the net attachment module is $z = d$, hence the circuit function is preserved. The test for *p* stuck-at-0 forces opposing values on *m* and *d*, and hence the corresponding WIE will be detected by the same test.

The detection of ELEs is performed by replacing the latch by the *latch replacement module* shown in Figure 2.11a, and then generating a test sequence for the SSL fault *p* stuck-at-0, which is also a test sequence for the ELE. Similarly, the detection of MLEs is performed by replacing the line by the *line replacement module* as shown in Figure 2.11b. The test sequence generated for the SSL fault *p* stuck-at-0 is also a test for the MLE. Hence, LCEs are easily mapped into SSL faults.

The overall verification process is divided into two phases. The first phase generates tests for gate errors (MIGSEs and MGEs) and is shown in Figure 2.12. If the circuit is sequential, additional tests for LCEs are generated. The second phase performs the error simulation for net errors (MIEs, WIEs) and then generates tests for the undetected ones;

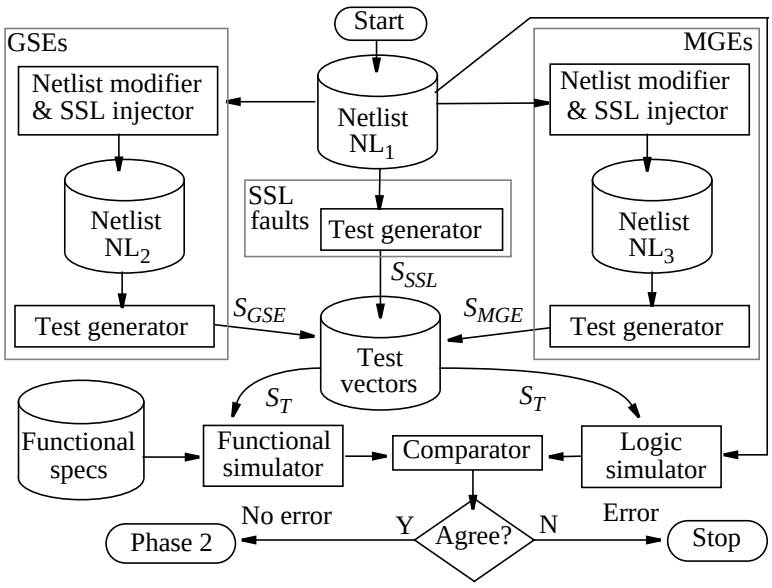


Figure 2.12 First phase of the design verification process.

the flowchart of phase 2 is similar to that of phase 1. Complete coverage of net errors may require several iterations through phase 2. If after checking for all modeled errors, the implementation is found to match the functional specifications, we can conclude with high confidence that the circuit is correct as designed.

2.3 Experimental Results

In this section we describe the experiments performed to support the preceding analysis; these experiments used the combinational ATPG tool ATALANTA [75] and the sequential ATPG tool ATTEST [17]. To determine the ability of a given test set to detect design errors and SSL faults, we developed an error/fault simulator ESIM. For combinational circuits, the simulator uses parallel-pattern evaluation and critical path tracing [4]. It simulates the circuit with multiple vectors concurrently and determines the detected errors/faults without explicit simulation of each error/fault. ESIM uses parallel fault simulation [4] for sequential circuits. Additional details of ESIM, as well as experiments and examples to demonstrate its capabilities, can be found in Appendix A.

The circuits used for the experiments are the ISCAS 85 combinational benchmarks [25], some standard, combinational 74X-series circuits [107], and the ISCAS 89 sequen-

Table 2.5 Design error coverage in combinational benchmarks using complete SSL test set generated by ATALANTA.

Circuit	Test set size	Detected SSL faults	Detected GSEs		Detected GCEs		Detected ICEs		Detected WIEs
			SIGSE	MIGSE	EGE	MGE	EIE	MIE	
c17	5	100.0	100.0	80.0	100.0	n/a	100.0	57.5	88.0
c432nr	44	100.0	100.0	89.1	100.0	95.5	100.0	73.1	96.9
c499nr	52	100.0	100.0	97.9	46.2	93.8	100.0	88.8	98.9
c880	47	100.0	100.0	90.3	100.0	94.6	100.0	84.9	98.6
7485	25	100.0	100.0	88.4	100.0	89.8	100.0	83.4	92.7
74181	18	100.0	100.0	96.2	88.9	90.6	100.0	81.8	94.0
74283	12	100.0	100.0	91.3	100.0	84.1	100.0	74.5	92.2

tial benchmarks [26]. We conducted a preliminary experiment to determine the coverage of design errors using a complete test set for all detectable SSL faults. The resulting data given in Table 2.5 show that a complete test set for SSL faults guarantees the detection of all SIGSEs and EIEs, confirming results in [2]. The detection of the other design error types is not guaranteed but they are likely to be detected because the test set does exercise each net in the circuit. Note that all the circuits in Table 2.5 are relatively small and SSL-irredundant. The circuits c432nr and c499nr are the irredundant equivalents of c432 and c499 respectively.

Our next experiments are concerned with generating nearly complete test sets for all modeled design errors. They use the method described in the previous section to generate test vectors targeting specific errors. The modified netlist is supplied to ATALANTA which generates a test set. The generated test sets are then evaluated using simulation to find their coverage of GSEs, as shown in Table 2.6. Since tests for MIGSEs cover EGEs (Theorem 2.3), the results on detecting EGEs are shown in Table 2.6. The coverage of MGEs is also shown in Table 2.6. Testing for MIEs, and WIEs is performed only for those errors that are not detected by error simulation using the set $S_T = S_{SSL} \cup S_{GSE} \cup S_{MGE}$. The coverage of EIEs is the same as that shown in Table 2.5 because a complete test set for SSL faults detects all EIEs. The error simulation results for MIEs and WIEs also appear in Table 2.6. Tests were generated using ATALANTA for the remaining undetected

Table 2.6 Design error coverage in combinational benchmarks using verification tests generated by ATALANTA.

Circuit	Tests targeting GSEs				Tests targeting MGEs		Error simulation for MIEs and WIEs using S_T		
	Test set size	Detected MIGSEs	Detected SIGSEs	Detected EGEs	Test set size	Detected MGEs	Test set size	Detected MIEs	Detected WIEs
c17	5	100.0	100.0	100.0	n/a	n/a	10	82.5	95.7
c432nr	39	92.8	100.0	100.0	92	99.9	174	88.8	99.5
c499nr	39	99.8	100.0	46.2	43	98.4	133	93.2	99.7
c880	49	92.8	100.0	100.0	66	100.0	162	95.0	99.8
7485	14	88.4	100.0	100.0	47	94.4	85	89.3	96.4
74181	15	98.5	100.0	88.9	36	99.5	69	94.9	98.8
74283	10	94.7	100.0	100.0	31	100.0	51	88.7	95.2

Table 2.7 Improved coverage of MIEs and WIEs after the second phase of test generation using ATALANTA.

Circuit	Tests targeting MIEs not detected by S_T		Tests targeting WIEs not detected by S_T	
	Total test set size	Detected MIEs	Total test set size	Detected WIEs
c17	13	95.0	12	100.0
c432nr	190	89.9	195	99.6
c499nr	220	95.8	147	99.8
c880	225	96.5	192	99.9
7485	91	91.2	92	96.4
74181	83	96.6	78	98.9
74283	58	90.0	56	96.4

MIEs and WIEs after the error simulation. ATALANTA reported that a large percentage of those errors are undetectable. Adding the generated tests to S_T improves the coverage of MIEs and WIEs, as shown in Table 2.7.

The coverage of design errors using the generated test sets is quite high, 80%–100% in most cases. We are confident that most detectable design errors of the modeled types are actually detected. To explore this further, we analyzed the 7485, 74181, and 74283 circuits in depth. We found that MIGSEs and EGEs not detected by our test sets are undetectable,

Table 2.8 Design error coverage in combinational benchmarks using verification tests generated by ATTEST.

Circuit	Size of S_T	Testing time[a]	Detected SIGSEs	Detected MIGSEs	Detected EGEs	Detected MGEs	Detected EIEs	Detected MIEs	Detected WIEs
c1355	265	3.2	100.0	82.3	100.0	97.1	99.2	83.5	99.3
c1908	465	3.7	100.0	84.8	97.6	90.3	99.2	90.8	97.3
c2670	797	7.9	99.7	87.5	87.6	91.6	93.2	90.4	98.7
c3540	650	11.2	99.3	89.7	90.6	81.6	94.2	88.4	98.6
c5315	1263	8.1	99.8	89.6	98.9	93.8	98.3	99.8	99.5
c6288	324	55.7	99.6	85.8	100.0	n/a	99.3	97.9	99.7
c7552	1364	23.5	100.0	86.6	97.4	91.2	96.4	99.4	98.7

a. In minutes on a HALstation 300.

and hence, these test sets cover 100% of the detectable MIGSEs and EGEs.

It is difficult to compare the coverage results obtained in this section to related work in the literature for several reasons: (1) different error models are used; (2) test set sizes are missing from the results of [65]; and (3) standard benchmarks are not used in most prior work. The test generation times for the circuits in Table 2.6 and Table 2.7 were found to range from a few seconds to a few minutes on a HALstation 300.

To check that our method can use any standard SSL test generator and to determine the design error coverage for the large SSL-redundant ISCAS 85 circuits, we performed the test generation experiments using the advanced SSL test generator ATTEST [17]. The error simulation results of the generated test sets are shown in Table 2.8. As expected, the generated test sets are small and have high coverage of the modeled errors.

We further experimented with the proposed method using a representative set of non-scan sequential benchmarks from the ISCAS 89 suite [26]. To simplify test generation, we attached a single clear (reset) input to all storage elements in each circuit. We also used the commercial ATTEST SSL test generator [17] to generate the verification test sequences $S_T = S_{SSL} \cup S_{GSE} \cup S_{MGE} \cup S_{ELE} \cup S_{MLE}$. The simulation results (Table 2.9) were determined by the error simulator ESIM, and demonstrate the effectiveness of the generated test sequences. The coverage of design errors is high for all circuits, except for s420

Table 2.9 Design error coverage in sequential benchmarks using verification test sequences generated by ATTEST.

Circuit	Size of S_T	Testing time[a]	Detected SSLs	Detected SIGSEs	Detected MIGSEs	Detected EGEs	Detected MGEs	Detected EIEs	Detected MIEs	Detected WIEs	Detected ELEs	Detected MLEs
s27	49	0	100.0	100.0	95.0	100.0	n/a	100.0	74.7	94.4	100.0	100.0
s208	448	8	95.6	91.3	87.9	87.8	83.8	91.6	72.4	93.0	87.5	81.8
s298	448	27	86.4	91.1	93.6	100.0	96.92	77.5	67.5	84.6	100.0	100.0
s344	264	180	93.9	97.3	90.7	100.0	85.0	91.9	76.9	94.2	100.0	95.0
s349	352	28	94.9	97.7	90.8	100.0	85.0	93.5	79.9	95.9	100.0	95.0
s386	500	132	85.1	97.1	92.7	78.4	98.2	69.6	67.1	87.6	100.0	92.9
s420	499	114	54.5	58.1	69.4	71.9	48.1	51.8	32.8	52.8	81.3	36.8
s641	360	14	87.6	93.5	94.8	73.5	90.6	81.7	65.2	90.4	78.9	89.8

a. In minutes on a HALstation 300.

whose internal nets have low controllability and observability.

2.4 Discussion

We have presented an error-based method for verifying logic circuits using standard simulation and ATPG tools. We showed that all common gate-level design errors can readily be mapped into SSL faults, and presented a systematic method to perform this mapping. Our experimental results show that complete test sets for the SSL faults detect almost all detectable design errors. The test sets are small and provide high coverage—the percentage of detected design errors from all modeled errors, detectable and undetectable, is greater than 90% for most of the benchmark circuits. The experiments also show that the fraction of undetectable design errors is significant in practical circuits, even when they are SSL-irredundant. For example, 11.6% of the MIGSEs in the 7485 comparator circuit are undetectable. We ensure full detectability of design errors by injecting SSL faults into a modified netlist and apply an ATPG program to it. Any such program can be used off the shelf, so future improvements in ATPG tools can be applied directly to this type of design error detection. Furthermore, as we show in the next chapter, the verification method considered here can be extended to higher levels of abstraction such as the register-transfer or behavioral level of design.

CHAPTER 3
HIGH-LEVEL DESIGN VALIDATION

In this chapter, we extend the work in Chapter 2 to high-level design validation. Like the gate-level validation approach, our high-level methodology is based on modeling design errors and generating simulation vectors for them via testing techniques for fabrication faults. Section 3.1 presents a review of high-level design verification and testing techniques. Section 3.2 describes a method for design error collection and presents some design error statistics that we have collected. Section 3.3 discusses design error modeling and illustrates test generation with these models. An experimental evaluation of the proposed methodology and error models is presented in Section 3.4. Section 3.5 introduces a new error model for microprocessors and a validation approach that uses it. Section 3.6 discusses the experimental results and gives some concluding remarks.

3.1 Introduction

Simulation-based design verification tries to uncover design errors by detecting a circuit's faulty behavior when deterministic or pseudo-random tests (simulation vectors) are applied. Microprocessors are usually verified by simulation-based methods, but require an extremely large number of simulation vectors whose coverage is often uncertain.

Hand-written test cases form the first line of defense against bugs, focusing on basic functionality and important corner (exceptional) cases. These tests are very effective in the beginning of the debug phase, but lose their usefulness later. Recently, tools have been developed to assist in the generation of focused tests [35][58]. Although these tools can significantly increase design productivity, they are far from being fully automated.

The most widely used method to generate verification tests automatically is random test generation. It provides a cheap way to take advantage of the billion-cycles-a-day simulation capacity of networked workstations available in many big design organizations. Sophisticated systems have been developed that are biased towards corner cases, thus improving the quality of the tests significantly [7]. Advances in simulator and emulator technology have enabled the use of very large sets as test stimuli such as existing application and system software. Successfully booting the operating system has become a common quality requirement [49][70].

Common to all the test generation techniques mentioned above is that they are not targeted at specific design errors. This poses the problem of quantifying the effectiveness of a test set, such as the number of errors covered. Various coverage metrics have been proposed to address this problem. However, the relationship between the metrics and the classes of design errors they detect is not well understood.

A different approach is to use synthetic design error models to guide test generation as we have done in the previous chapter. Such a method is also found in the area of software testing. Mutation testing [44] considers programs, termed mutants, that differ from the program under test by a single small error, such as changing the operator from add to subtract. Although considered too costly for wide-scale industrial use, mutation testing is one of the few approaches that has yielded an automatic test generation system for software testing, as well as a quantitative measure of error coverage (mutation score) [68]. Recently, Al Hayek and Robach [15] have adapted mutation testing to hardware design verification in the case of small VHDL modules.

This chapter addresses design validation via error modeling and test generation for complex high-level designs such as microprocessors. The implementation to be verified and its specification are assumed to be given. For microprocessors, the specification is typically the instruction set architecture (ISA), and the implementation is a description of the new design in a hardware description language (HDL) such as VHDL or Verilog. Synthetic error models are used to guide test generation, and the tests are applied to simulated models of both the implementation and the specification. A discrepancy between the two simulation outcomes indicates an error, either in the implementation or in

the specification.

As discussed in Chapter 1, several high-level manufacture testing techniques for fabrication faults have been proposed. Most of these methods use high-level knowledge about the design in the test generation algorithm to detect gate-level fabrication faults. A few other methods introduce high-level fault models to speed up the test generation. For example, Thatte and Abraham [108] defined high-level fault models for the following functions: register decoding, instruction decoding and control, data storage, and data transfer. The corresponding test generation algorithm produces sequences of instructions which detect the above faults in the microprocessor with the hope of detecting the low-level SSL faults. A general problem of high-level manufacture testing is the absence of high-level ATPG techniques and supporting software tools.

From the above discussion, we can conclude that high-level validation is more complex than gate-level validation due to the following reasons: (*i)* the lack of high-level design error data and good design error models, and (*ii*) the inadequacy of high-level ATPG tools. In the rest of this chapter, we develop a set of high-level design error models based on actual error data and show how to generate tests for them.

3.2 Design Error Collection

As discussed earlier, hardware design verification and physical fault testing are closely related conceptually. The basic task of physical fault testing (hardware design verification) is to generate tests that distinguish the correct circuit from faulty (erroneous) ones. The class of faulty circuits to be considered is defined by a logical fault model. Logical fault models represent the effect of physical faults on the behavior of the system, and free us from having to deal with the plethora of physical fault types directly. The most widely used logical fault model, the SSL model, combines simplicity with the fact that it forces each line in the circuit to be exercised. Typical hardware design methodologies employ hardware description languages as their input medium and use previously designed high-level modules. To capture the richness of this design environment, the SSL model needs to be supplemented with additional error models.

The lack of published data on the nature, frequency, and severity of the design errors

occurring in large-scale projects is a serious obstacle to the development of error models for hardware design verification. Although bug reports are collected and analyzed internally in industrial design projects the results are rarely published. Examples of user-oriented bug lists can be found in [60][84]. Some insight into what can go wrong in a large processor design project is provided in [41].

The above considerations have led us to implement a systematic method for collecting design errors. Our method uses the CVS revision management tool [33] and targets ongoing design projects at the University of Michigan, including the PUMA high-performance microprocessor project [27] and various class projects in computer architecture and VLSI design, all of which employ Verilog as the hardware description medium. Designers are asked to archive a new revision via CVS whenever a design error is corrected or whenever the design process is interrupted, making it possible to isolate single design errors. We have augmented CVS so that each time a design change is entered, the designer is prompted to fill out a standardized multiple-choice questionnaire, which attempts to gather four key pieces of information: (1) the motivation for revising the design; (2) the method by which a bug was detected; (3) a generic design-error class to which the bug belongs, and (4) a short narrative description of the bug. A uniform reporting method such as this greatly simplifies the analysis of the errors. A sample error report using our standard questionnaire is shown in Figure 3.1. The error classification shown in the report is the result of the analysis of error data from several earlier design projects.

Design error data has been collected from four VLSI design class projects that involve implementing the DLX microprocessor [57], from the implementation of the LC-2 microprocessor [99] which is described later, and from preliminary designs of PUMA's fixed-point and floating-point units [27]. The distributions found for the various representative design errors are summarized in Table 3.1. Error types that occurred with very low frequency were combined in the "others" category in the table. The number of design errors recorded per day for the duration of one particular class project is shown in Figure 3.2 [113]. The graph reflects the somewhat sporadic nature of student design projects.

```
(replace the _ with X where appropriate)

MOTIVATION:

X bug correction
_ design modification
_ design continuation
_ performance optimization
_ synthesis simplification
_ documentation

BUG DETECTED BY:

_ inspection
_ compilation
X simulation
_ synthesis

BUG CLASSIFICATION:

Please try to identify the primary source
of the error. If in doubt, check all
categories that apply.

X combinational logic:

     X wrong signal source
     _ missing input(s)
     _ unconnected (floating) input(s)
     _ unconnected (floating) output(s)
     _ conflicting outputs
     _ wrong gate/module type
     _ missing instance of gate/module

_ sequential logic:

     _ extra latch/flipflop
     _ missing latch/flipflop
     _ extra state
     _ missing state
     _ wrong next state
     _ other finite state machine error

_ statement:

     _ if statement
     _ case statement
     _ always statement
     _ declaration
     _ port list of module declaration

_ expression (RHS of assignment):

     _ missing term/factor
     _ extra term/factor
     _ missing inversion
     _ extra inversion
     _ wrong operator
     _ wrong constant
     _ completely wrong

_ buses:

     _ wrong bus width
     _ wrong bit order

_ verilog syntax error

_ conceptual error

_ new category (describe below)

BUG DESCRIPTION: Used wrong field from
instruction
```

Figure 3.1 Sample error report.

Table 3.1 Actual error distributions from three groups of design projects.

Design error category	Relative frequency [%]		
	DLX	PUMA	LC-2
1. Wrong signal source	29.9	28.4	25.0
2. Conceptual error	39.0	19.1	0.0
3. Case statement	0.0	10.1	0.0
4. Gate or module input	11.2	9.8	0.0
5. Wrong gate/module type	12.1	0.0	5.0
6. Wrong constant	0.4	5.7	10.0
7. Logical expression wrong	0.0	5.5	10.0
8. Missing input(s)	0.0	5.2	0.0
9. Verilog syntax error	0.0	3.0	0.0
10. Bit width error	0.0	2.2	15.0
11. If statement	1.1	1.6	5.0
12. Declaration statement	0.0	1.6	0.0
13. Always statement	0.4	1.4	5.0
14. FSM error	3.1	0.3	0.0
15. Wrong operator	1.7	0.3	0.0
16. Others	1.1	5.8	25.0

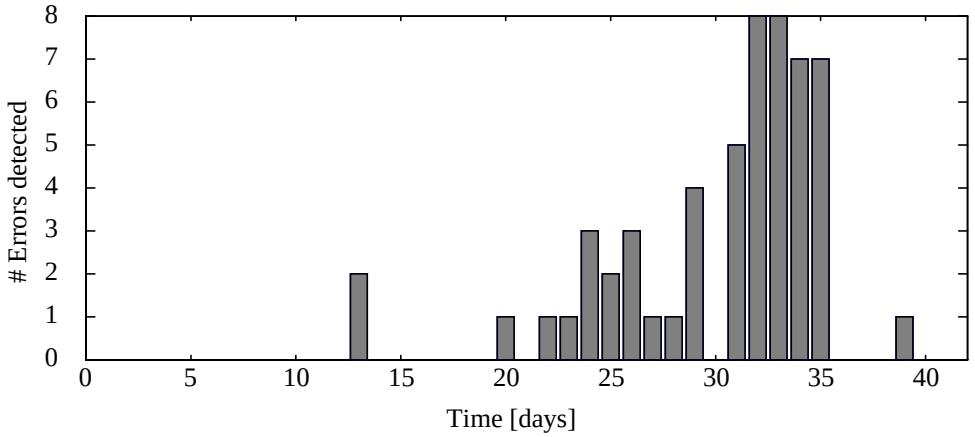


Figure 3.2 Number of errors detected per day for the duration of one class project.

3.3 Error Modeling

Standard simulation and logic synthesis tools have the side effect of detecting some design error categories of Table 3.1, and hence there is no need to develop models for those particular errors. For example a simulator such as Verilog-XL [30] flags all Verilog syntax errors (category 9), declaration statement errors (category 12), and incorrect port list of modules (category 16). Also, logic synthesis tools, such as those of Synopsis, usually flag all wrong bus width errors (category 10) and sensitivity-list errors in the *always* statement (category 13).

To be useful for design verification, error models should satisfy three requirements: (1) tests (simulation vectors) that provide complete coverage of the modeled errors should also provide very high coverage of actual design errors; (2) the modeled errors should be amenable to automated test generation; (3) the number of modeled errors should be relatively small. The error models need not mimic actual design bugs precisely, but the tests derived from complete coverage of modeled errors should provide very good coverage of actual design bugs.

Basic error models. A set of error models that satisfy the requirements for the restricted case of gate-level logic circuits was introduced in Chapter 2. Several of these models appear useful for the higher-level (RTL) designs found in Verilog descriptions as well. From the actual error data in Table 3.1, we derive the following set of five basic error models:

- *Bus SSL error (SSL)*: A bus of one or more lines is (totally) stuck-at-0 or stuck-at-1 if all lines in the bus are stuck at logic level 0 or 1. This generalization of the standard SSL model was introduced in [21] in the context of physical fault testing. Many of the design errors listed in Table 3.1 can be modeled as SSL errors (categories 4 and 6).
- *Module substitution error (MSE):* This refers to mistakenly replacing a module by another module with the same number of inputs and outputs (category 5). This class includes word gate substitution errors and extra/missing inversion errors.
- *Bus order error (BOE)*: This refers to incorrectly ordering the bits in a bus (category 16). Bus flipping appears to be the most common form of BOE.
- *Bus source error (BSE):* This error corresponds to connecting a module input to a wrong source (category 1).
- *Bus driver error (BDE)*: This refers to mistakenly driving a bus from two sources (category 16).

To detect a basic error *e* in a circuit *C*, we need to activate *e*, propagate the erroneous values to an observable output in *C*, and justify the corresponding internal signals of *C*. We next study the activation conditions for the basic design error models.

- SSL: For an *m*-wide bus to be stuck-at 0, there are $2^m - 1$ tests that can activate the error, namely, any *m*-bit vector that has at least one bit set to 1. Hence the bus SSL error can be easily activated.
- MSE: The number of tests needed to detect a substitution error of module M_1 by module M_2 is one; any input vector that distinguishes M_1 from M_2 suffices as a test. Hence, if M_1 can be replaced by other *k* modules, we need at most *k* tests to detect MSEs on M_1. The number of possible tests that can detect the substitution of M_1 by M_2 is equal to the number of minterms of the difference function *f*, which is defined as the logical OR of the Exclusive-OR of the corresponding outputs of M_1 and M_2. For the special case of standard word gates, to verify an *n*-input *m*-wide $n, m \geq 2$, word gate *G* using the results in Chapter 2, we need to apply each of the four tests v_{null}, v_{all}, v_{odd}, and v_{even} to an arbitrary gate of *G*. This requirement can be satisfied by a single test if $m \geq 4$ and the output of the word gate can be propa-

gated through a transparent path to a primary output. In this case, a single test can detect 5 MSEs on a word gate. Note that for the case of an *m*-wide inverter, any test will activate the error.

- BOE: A single test is sufficient to activate a bus order error. However, the number of possible tests is dependent on the number of possible ways a wrong order can occur. For the case of incorrectly flipping the order of an *m*-wide (*m* even) bus, any non-symmetrical vector is a test. (A vector *v* is non-symmetrical if there exists a bit *i* of *v* such that $v_i \neq v_{m-i-1}$.) For example, the test vector 1000 detects incorrectly flipping the order of a 4-bit bus. Since the number of *m*-wide non-symmetrical vectors is $2^m - 2^{m/2}$, which amounts to 93.75% of the possible vectors on an 8-bit bus, then BOEs are likely to be activated by random vectors.
- BSE: A single test that places different values on the wrong and correct buses is sufficient to activate this error. The number of possible tests for a BSE on an *m*-wide bus is equal to the number of instances where the wrong and correct buses have different values, i.e. $2^{2m} - 2^m$. This number amounts to 99.60% of the possible vectors on an 8-bit bus, hence BSEs are likely to be activated by random vectors.
- BDE: Any test that enables more than two bus drivers simultaneously and produce conflicting bus signals is sufficient to activate this error.

After activating the basic error, we need to propagate the resulting erroneous signal to an observable output. We therefore need to define a criterion for propagating error values (D or $\overline{D}$) through modules. A simple criterion is to maximize the number of $D/\overline{D}$ error signals propagated from the initial error site. To illustrate, consider a 2-input *m*-wide AND word gate with inputs *A* and *B* and output *Z*. To propagate an error signal from *A* to *Z*, the input *B* is set such that a maximum number of $D/\overline{D}$ is propagated to *Z*. For example, if $A = 01XD\overline{D}1$, then we set $B = XXX11X$. In general, if $A_i = D$ or $A_i = \overline{D}$ then $B_i = 1$ otherwise $B_i = X$. To illustrate further, consider a multiplexer with output *Z*, selection bus *S*, and data inputs A_0, A_1, A_2, ... To propagate an error signal from data bus A_k to *Z*, we need to set the *S* bus to the fixed value *k* while setting all the other input buses to *X*'s. On the other hand, to propagate an error from the *S* bus to *Z*, we need to set several

data buses to specific values. For example, if $S = 01D$, then we set A_2 to the complement of A_3. This propagates a maximum number of $D/\overline{D}$ signals to Z, since every bit of Z is either D or $\overline{D}$.

After propagating the error to an observable output, we must justify the internal signals of the circuit to reach to a consistent solution. We define the criterion for justifying inputs of modules to obtain the desired outputs as maximizing the number N of X's appearing at the inputs. The justification algorithm for a module enumerates all solutions, sorts them based on the values of N, and returns the test with the largest N once it is executed for the first time and the test with the largest N among the remaining tests in every consequent execution. To illustrate, consider a standard 4-bit adder with a carry out c_4 and input vector $a_3b_3a_2b_2a_1b_1a_0b_0c_0$. To justify $c_4 = 1$, the first and best test is 11XXXXXXXXX which corresponds to $N = 2$, and the next test is 1011XXXXXXX which corresponds to $N = 4$.

Direct generation of tests for the basic error models is difficult, and is not supported by currently available CAD tools. While the errors can be easily activated as we have shown above, propagation of their effects can be difficult, especially when modules or behavioral constructs do not have transparent operating modes [88]. In the following, we illustrate manual test generation for various basic error models.

Test generation examples. Because of their relative simplicity, the foregoing error models allow tests to be generated and error coverage evaluated for RTL circuits of moderate size. We consider the test requirements of two representative combinational circuits: a carry-lookahead adder and an ALU. The test generation is done manually here, but in a systematic manner that can potentially be automated. Three basic error models are considered: BOEs, MSEs, and BSEs. Test generation for SSL faults is discussed in [4][21]; no tests are needed for BDEs, since the circuits under consideration do not have tristate buses.

Our first example is the 74283 4-bit fast adder [107]. An RTL model [51] of the adder appears in Figure 3.3. It consists of a carry-lookahead generator (CLG) and a few word gates. We show how to generate tests for some design error models in the adder and then discuss the overall coverage of the targeted error models.

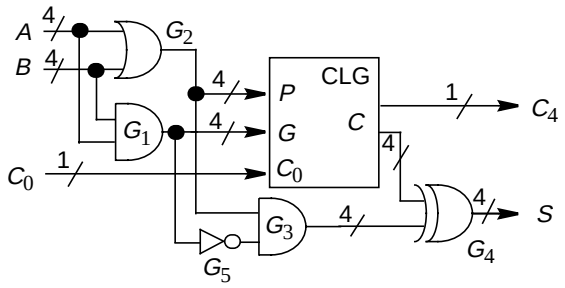


Figure 3.3 High-level model of the 74283 carry-lookahead adder.

- **BOE on bus *A*:** A possible non-symmetrical vector that activates this error is $A_g = 0XX1$, where X denotes an unknown value. The erroneous value of A is then $A_f = 1XX0$. Hence, we can represent the error by $A = \bar{D}XXD$, where D ($\bar{D}$) is 1 (0) in the good circuit and 0 (1) in the erroneous circuit. One way to propagate the error signal through the AND gate G_1 is to set $B = 1XX1$. Hence, we get $G_2 = 1XX1$, $G_5 = DXX\bar{D}$, and $G_3 = DXX\bar{D}$. Now for the module CLG we have $P = 1XX1$, $G = \bar{D}XXD$, and $C_0 = X$. The resulting outputs are $C = XXXX$ and $C_4 = X$. This implies that $S = XXXX$ and hence the error signal is not detected at the primary outputs. We need to assign more input values for error signal propagation. If we set $C_0 = 0$, we get $C = XXD0$, $C_4 = X$, and $S = XXX\bar{D}$. Hence, the error signal is propagated to S and the complete test vector is A, B, $C_0 = 0XX11XX10$.
- **BSE on bus *P* with correct source G_3:** To activate the error we need to apply opposite values to at least one bit of the P and G_3 buses. If we start with $P_f = XXX0$ and $G_3 = P_g = XXX1$, we reach to a conflict through implications. If we try $P_f = XXX1$ and $G_3 = P_g = XXX0$, we obtain $P = XXX\bar{D}$, $A = XXX1$, and $B = XXX1$. However, no error is propagated through the CLG module since $G = XXX1$. After all the activation conditions are explored, we conclude that the error is redundant (undetectable).
- **MSE G_3/XNOR**: To distinguish the word AND gate G_3 from an XNOR gate, we need to apply the all-0 pattern to one of the gates forming G_3. So, we start with the values $G_5 = 0XXX$ and $G_2 = 0XXX$. By making implications, we find that there is a conflict when selecting the values of A and B. We then change to another set of activation condition $G_5 = X0XX$ and $G_2 = X0XX$. This also leads to a conflict. After

trying all possible combinations, we conclude that no test exists, hence the error is redundant.

On generating tests for all BSEs in the adder we find that just 2 tests detect all 33 detectable BSEs, and a single BSE is redundant as shown above. We further targeted all MSEs in the adder and found that 3 tests detect all 27 detectable MSEs; the MSE G_3/XNOR is redundant. Finally, we found that all BOEs are detected by the tests generated for BSEs and MSEs. Therefore, complete coverage of BOEs, BSEs, and MSEs is achieved with only 5 tests.

In our second example, we try to generate tests for some modeled design errors in the c880 ALU, a member of the ISCAS 85 benchmark suite [25]. A high-level model based on a Verilog description of the ALU [67] is shown in Figure 3.4. The c880 is composed of six modules: an adder, two multiplexing units, a parity unit, and two control units. The circuit has 60 inputs and 26 outputs, and its standard gate-level implementation has 383 gates. The design error models to be considered in the c880 are again BOEs, BSEs, and MSEs (inversion errors on 1-bit signals). We next generate tests for these error models.

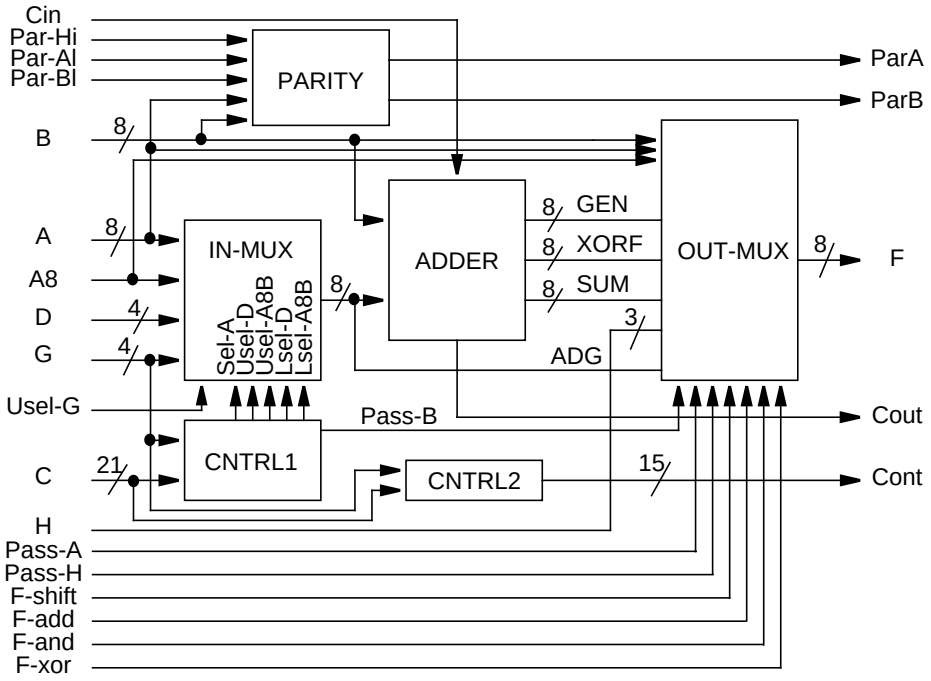


Figure 3.4 High-level model of the c880 ALU.

- **BOEs:** In general, we attempt to determine a minimum set of assignments needed to detect each error. Some BOEs are redundant such as the BOE on *B* (PARITY), but most BOEs are easily detectable. Consider, for example, the BOE on *D*. One possible way to activate the error is to set *D*[3] = 1 and *D*[0] = 0. To propagate the error to a primary output, the path across IN-MUX and then OUT-MUX is selected. The signal values needed to activate this path are:

Sel-A = 0	*Usel_D* = 1	*Usel_A8B* = 0	*Usel_G* = 0
PassB = 0	*PassA* = 1	*PassH* = 0	*F-shift* = 0
F-add = 0	*F-and* = 0	*F-xor* = 0	

  Solving the gate-level logic equations for *G* and *C* we get:

G[1:2] = 01	*C*[3] = 1	*C*[5:7] = 011	*C*[14] = 0

  All signals not mentioned in the above test have don't care values. We generated tests for all BOEs in the c880. We found that just 10 tests detect all 22 detectable BOEs and serve to prove that another 2 BOEs are redundant.
- **BSEs:** The buses in the ALU were grouped according to their size since the correct source of a bus must have the same size as the incorrect one. We targeted BSEs with bus widths of 8 and 4 only. We found that by adding 3 tests to the 10 tests generated for BOEs, we are able to detect all 27 BSEs affecting the c880's multibit buses. Since the multiplexing units are not decoded, most BSEs on their 1-bit control signals are detected by the tests generated for BOEs. Further tests are needed to get complete coverage of BSEs on the other 1-bit signals.
- **MSEs:** Tests for BOEs detect most but not all inversion errors on multibit buses. In the process of test generation for the c880 ALU, we noticed a case where a test for an inversion error on a bus *A* can be found even though the BOE on *A* is redundant. This is the case when an *n*-bit bus (*n* odd) is fed into a parity function. Testing for inversion errors on 1-bit signals needs to be considered explicitly, since a BOE on a 1-bit bus is not possible. Most inversion errors on 1-bit signals in the c880 ALU are detected by the tests generated for BOEs and BSEs. This is especially true for the control signals to the multiplexing units.

Conditional error model. The preceding examples, as well as prior work on SSL error detection [2][21], show that the basic error models can be used with RTL circuits, and that high, but not complete, error coverage can be achieved with very small test sets. These results are further reinforced by our experiments on microprocessor verification (Section 3.4) which indicate that a large fraction of actual design errors (67% in one case and 75% in the other) is detected by complete test sets for the basic errors. To increase coverage of actual errors to the very high levels needed for design verification, additional error models are required to guide test generation. Many more complex error models can be derived directly from the actual data of Table 3.1 to supplement the basic error types, the following set being representative:

- *Bus count error (BCE):* This corresponds to defining a module with more or fewer input buses than required (categories 4 and 8).
- *Module count error (MCE):* This corresponds to incorrectly adding or removing a module (category 16), which includes the extra/missing word gate errors and the extra/missing registers.
- *Label count error (LCE)*: This error corresponds to incorrectly adding or removing the labels of a case statement (category 3).
- *Expression structure error (ESE)*: This includes various deviations from the correct expression (categories 3, 6, 7, 11, 15), such as extra/missing terms, extra/missing inversions, wrong operator, and wrong constant.
- *State count error (SCE)*: This error corresponds to an incorrect finite state machine with an extra or missing state (category 14).
- *Next state error (NSE)*: This error corresponds to incorrect next state function in an FSM (category 14).

Although this extended set of error models increases the number of actual errors that can be modeled directly, we have found them to be too complex for practical use in automated test generation. For example, it is impractical to enumerate missing modules (MCEs) since the possible instances depend on many module parameters including type, number of inputs, sources of inputs, number of outputs, and destination of outputs.

The more difficult actual errors are often composed of multiple basic errors, where the

component basic errors interact in such a way that a test to detect the actual error must be much more specific (have fewer don't cares) than a test to detect any of the component basic errors. Modeling these difficult composite errors directly is impractical as the number of error instances to be considered is too large, and such composite modeled errors are too complex for automated test generation. However, as noted earlier, a good error model does not necessarily need to mimic actual errors accurately. What is required is that the error model necessitates the generation of these more specific tests. To be practical, the complexity of the new error models should be comparable to that of the basic error models. Furthermore, the (unavoidable) increase in the number of error instances should be controlled to allow trade-offs between test generation effort and verification confidence. We found that these requirements can all be met in many practical situations by augmenting the basic error models with a condition.

A *conditional error* (*C*,*E*) consists of a condition *C* and a basic error *E*; its interpretation is that *E* is only active when *C* is satisfied. In general, *C* is a predicate over the signals in the circuit during some time period. To limit the number of error instances, we restrict *C* to a conjunction of terms of the form $(y_i = w_i)$, where y_i is a signal in the circuit and w_i is a constant of the same bit-width as y_i and whose value is either all-0s or all-1s. The number of terms (condition variables) appearing in *C* is said to be the *order* of (*C*,*E*). Specifically, we consider the following conditional error types:

- Conditional single-stuck line errors (CSSL*n*) of order *n*;
- Conditional bus order errors (CBOE*n*) of order *n*;
- Conditional bus source errors (CBSE*n*) of order *n*.

When $n = 0$, a conditional error (*C*,*E*) reduces to the basic error *E* from which it is derived. Higher-order conditional errors enable the generation of more specific tests, but lead to a greater test generation cost due to the larger number of error instances. Although the total set of all *N* signals we consider for each term in the condition can possibly be reduced, CSSL*n* errors where $n > 2$ are probably not practical.

For gate-level circuits (where all signals are 1-bit), it can be shown that CSSL1 errors cover the following basic error models: MSEs (excluding XOR and XNOR gates), missing

2-input gate errors, BSEs, single BCEs (excluding XOR and XNOR gates), bus driver errors. Higher-order CSSL*n* errors improve coverage even further.

3.4 Coverage Evaluation

The effectiveness of a verification methodology can be measured by its ability to uncover actual design errors in an unverified design. An experiment was designed to evaluate the effectiveness of our verification methodology when applied to two student-designed microprocessors. A block diagram of the experimental set-up is show in Figure 3.5. As design error models are used to guide test generation, the effectiveness is closely related to the synthetic error models used.

To evaluate our methodology, a circuit is chosen for which design errors are to be systematically recorded during its design. Let D_0 be the final, presumably correct design. From the CVS revision database, the actual errors are extracted and converted such that they can be injected in the final design D_0. In the evaluation phase, the design is restored to an (artificial) erroneous state, D_1, by injecting a single actual error into the final design D_0. This set-up approximates a realistic on-the-fly design verification scenario. The experiment answers the question whether given D_1, the proposed methodology would produce a test that determines D_1 to be erroneous. This is done by examining the actual error in D_1,

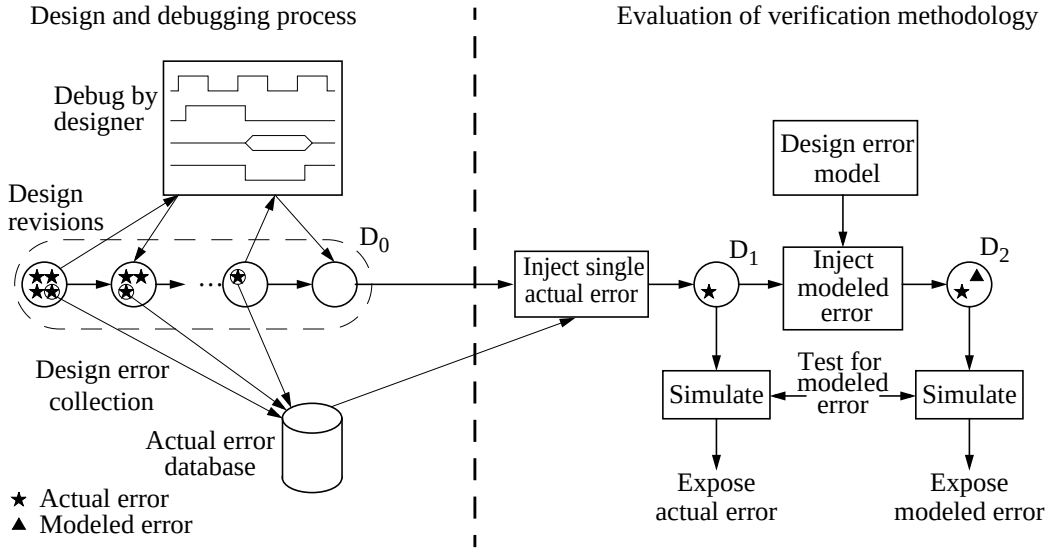


Figure 3.5 Experimental set-up to evaluate the proposed design verification methodology.

and determining if a modeled design error exists that is dominated by the actual error. Let D_2 be the design constructed by injecting an error model M into D_1. If any test that detects the modeled error M in D_2 also detects the actual error in D_1, then M is called a *dominated* error. Consequently, if we were to generate a complete test set for every error defined on D_1 by M, D_1would be found erroneous by that test set. Note that the concept of dominance in the context of design verification is slightly different than in physical fault testing. Unlike the testing problem, we cannot remove the actual design error from D_1 before injecting the dominated modeled error. This distinction is important because generating a test for an error of omission, which is generally very hard, becomes relatively easy if given D_0 instead of D_1.

The erroneous design D_1 considered in this experiment is somewhat artificial. In reality a design evolves over time as bugs are introduced and eliminated. Only at the very end of the design process, is the target circuit in a state where it differs from the final design D_0 in just a single design error. Prior to that time, the design may contain more than one design error. To the extent that the design errors are independent, it does not matter if we consider a single or multiple design errors in each verification step. Furthermore, our results are independent of the order in which one applies the generated tests.

The preceding coverage-evaluation experiment was implemented for two small but representative designs: a simple microprocessor and a pipelined microprocessor. We present our results in the remainder of this section.

A simple microprocessor. The Little Computer 2 (LC-2) [99] is a simple computer used for teaching purposes at the University of Michigan. It has a representative set of 16 instructions that are subset of the instruction sets of most current microprocessors. To serve as a test case for design verification, we designed behavioral and RTL synthesizable Verilog descriptions for the LC-2 microprocessor (Appendix B). The behavioral model of the LC-2 consists of 235 lines of behavioral Verilog code. The RTL design consists of a datapath unit composed of library modules and a few custom modules, and a control unit described as a finite-state machine with five states and 27 output control signals. The RTL design consists of 921 lines of Verilog code, excluding the models for library modules such as adders, register files, etc. A gate-level model of the LC-2 can thus be obtained using logic synthesis

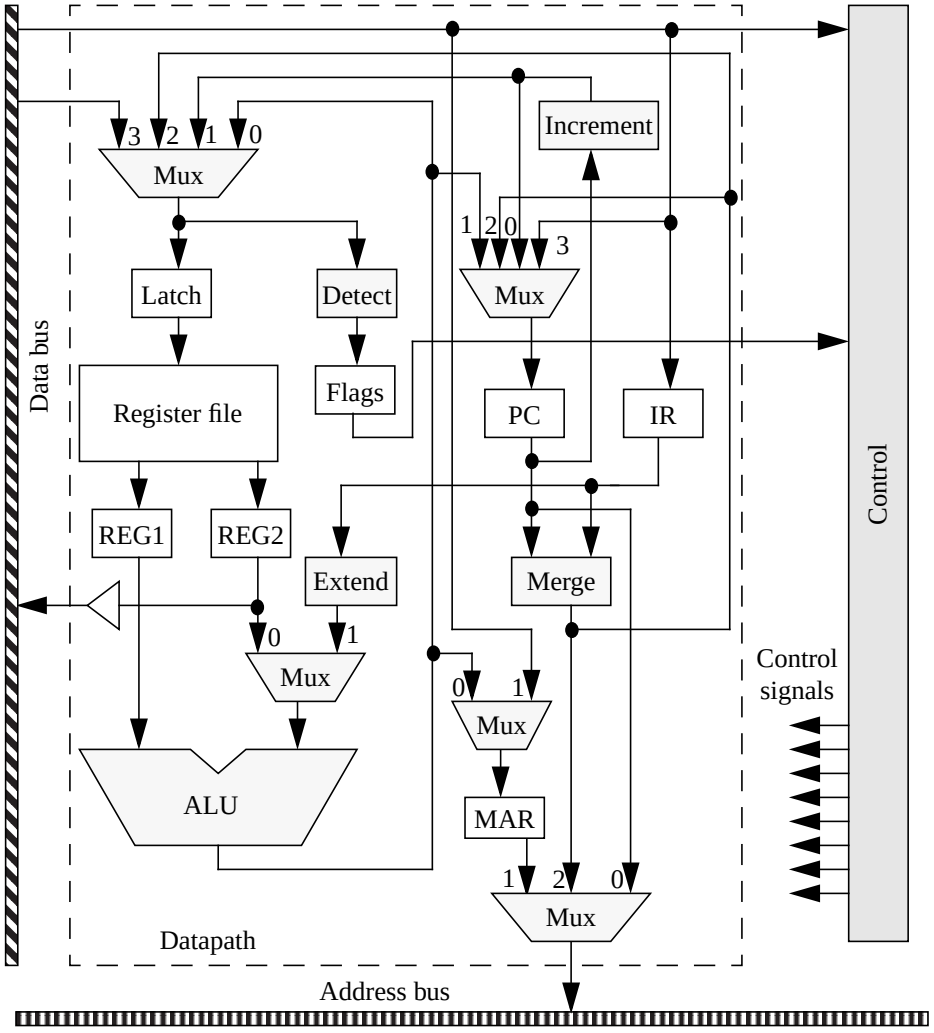


Figure 3.6 RTL block diagram of the LC-2 microprocessor.

tools. A simplified block diagram of the design is shown in Figure 3.6 The design errors made during the design of the LC-2 were systematically recorded using our error collection system (Section 3.2).

For each actual design error recorded, we derived the necessary conditions to detect it. An error is detected by an instruction sequence *s* if the external output signals of the behavioral model (specification) and the RTL model (implementation) are distinguished by *s*. We found that some errors are undetectable since they do not affect the functionality of the microprocessor. The detection conditions are used to determine if a modeled error that is dominated by the actual error can be found. An example where we were able to do

Incorrect design Correct design

Figure 3.7 An example of an actual design error that is dominated by an SSL error.

```
// Instruction decoding
// Decoding of register file
inputs
// 1- Decoding of R1

CORRECT CODE:

    if (ir_out[15:12] == 4'b1101)
         R1_temp = 3'b111;
     else
         R1_temp = ir_out[8:6];

ERRONEOUS CODE:

  R1_temp = ir_out[8:6];
```

Design error

(a)

```
// Instruction sequence

@3000
main:
      JSR sub0
      ........
      ........
sub0:
      Not R0, R7
      RET        //1101 0000 0000
0000
//
// After execution of instructions
// PC = 3001 in correct design
// PC = CFFE in incorrect design
```

Test sequence

(b)

Figure 3.8 An example of (a) an actual design error for which no dominated modeled error was found, and (b) an instruction sequence that detects the actual error.

that is shown in Figure 3.7. The error is a BSE on data input 1 of a multiplexer (mux) attached to the program counter PC. Testing for input 1 stuck-at-1 will detect the BSE since the outputs of PC and the increment unit are always different, i.e., the error is always activated, and testing for the SSL will propagate the signal on input number 1 of the multiplexer to a primary output of the microprocessor. A case where we were not able to find a basic or conditional modeled error dominated by the actual error is shown in Figure 3.8a. Here the error occurs when a signal is assigned a value independent of any condition. However, the correct implementation requires an if-then-else construct to control the signal assignment. To activate this error, we need to set ir_out[15:12] == 4’b1101, ir_out[8:6] ≠ 3’b111, and RF[ir_out[8:6]] ≠ RF[3’b111], where RF[*i*] refers to the contents of the register *i* in the register file. An instruction sequence that detects this error is shown in Figure 3.8b.

Table 3.2 Actual design errors and the corresponding dominated modeled errors for LC-2.

Actual design errors				Corresponding dominated modeled errors			
Category	Total	Easily detected	Unde-tectable	SSL	BSE	CSSL1	Un-known
Wrong signal source(s) (1)	4	0	0	2	2	0	0
Expression error (7)	4	0	0	2	0	1	1
Bit width error (10)	3	3	0	0	0	0	0
Missing assignment(s) (16)	3	0	0	0	0	2	1
Wrong constant(s) (6)	2	0	0	2	0	0	0
Unused signal (16)	2	0	2	0	0	0	0
Wrong module (5)	1	0	0	1	0	0	0
Always statement (13)	1	1	0	0	0	0	0
Total	20	4	2	7	2	3	2

We analyzed the actual design errors in both the behavioral and RTL designs of the LC-2, and the results of the experiment are summarized in Table 3.2. A total of 20 errors were made during the design process, of which four errors are easily detected by the Verilog simulator and/or logic synthesis tools and two errors are undetectable. The actual design errors are grouped by category; the numbers in parentheses refer to the corresponding category in Table 3.1. The columns in the table give the type of the simplest dominated modeled error corresponding to each actual error. For example, among the 4 remaining wrong-signal-source errors, two dominate an SSL error and two dominate a BSE error.

We can infer from Table 3.2 that most errors are detected by tests for SSL errors or BSEs. About 75% of the actual errors in the LC-2 designs can be detected after simulation with tests for SSL errors and BSEs. The coverage increases to 90% if tests for CSSL1 is also used.

A pipelined microprocessor. The second design case study was mainly carried out by David Van Campenhout [113]. It considers the well-known DLX microprocessor [57] which has more of the features found in contemporary microprocessors. The particular DLX version considered is a student-written design that implements 44 instructions, has a five-stage pipeline and branch prediction logic, and consists of 1552 lines of structural Verilog code, excluding the models for library modules such as adders, register-files, etc. A simplified block diagram of the design is shown in Figure 3.9. The design errors committed

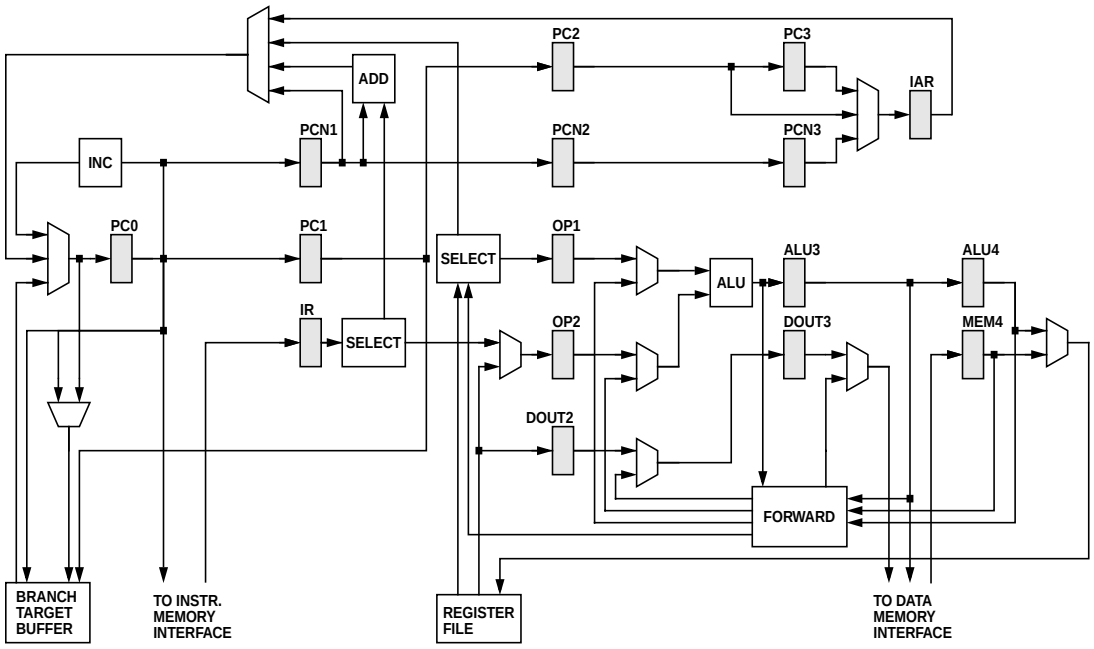


Figure 3.9 Block diagram of the DLX microprocessor.

Table 3.3 Actual design errors and the corresponding dominated modeled errors for our DLX implementation.

Actual design errors		Corresponding dominated modeled errors						
Category	**Detectable**	**INV**	**SSL**	**BSE**	**CSSL1**	**CBOE**	**CSSL2**	**Un-known**
Missing module(s) (2)	14	0	2	0	6	1	0	1
Wrong signal source(s) (1)	11	1	4	5	1	0	0	0
Complex (2)	3	0	3	0	0	0	0	0
Inversion (5)	3	3	0	0	0	0	0	0
Missing input(s) (4)	3	0	0	0	1	0	0	0
Unconnected input(s) (4)	3	3	0	0	0	0	0	0
Missing minterm (2)	1	0	0	0	0	0	1	0
Extra input(s) (2)	1	0	1	0	0	0	0	0
Total	39	7	10	5	8	1	1	1

by the student during the design were systematically recorded using our error collection system.

As in the previous experiment, Van Campenhout analyzed the detection requirements of each actual error and constructed a modeled error dominated by the actual error, wherever possible. The results of this experiment are summarized in Table 3.3. A total of 39 detectable design errors were recorded by the designer. The actual design errors are

grouped by category; the numbers in parentheses refer again to Table 3.1. The correspondence between the categories is imprecise, because of inconsistencies in the way in which different student designers classified their errors. Also, some errors in Table 3.3 are assigned to a more specific category than in Table 3.1, to highlight their correlation with the errors they dominate. ‘Missing module’ and ‘wrong signal source’ errors account for more than half of all errors. The columns give the type of the simplest dominated modeled error corresponding to each actual error. Among the 10 detectable ‘missing module(s)’ errors, two dominate an SSL error, six dominate a CSSL1 error, and one dominates a CBOE; for the remaining one, no dominated modeled error was found.

A conservative measure of the overall effectiveness of our verification approach is given by the coverage of actual design errors by complete test sets for modeled errors. Any complete test set for the inverter insertion errors (INV) also detects at least 21% of the (detectable) actual design errors. Any complete test set for the INV and SSL errors covers at least 52% of the actual design errors. If a complete test set for all INV, SSL, BSE, CSSL1 and CBOE is used, at least 94% of the actual design errors will be detected.

3.5 Mutation Control Errors

The preceding error models can, in principle, be used with all types of designs. In this section, we describe a related error model intended specifically for microprocessor-like circuits. This model targets control errors in designs where datapath and control are clearly separated. It is similar to the conditional error model with the condition being dependent on a single instruction and its cycles. We next define the model, present a mutation-based validation approach using it, and illustrate the validation approach on the LC-2.

Mutation control error model. A *mutation control error (MCE)* denoted (*i*,*c*,*s*,*vc*,*ve*) is a change in the control signal *s* in the cycle *c* of the instruction *i* of the microprocessor from the correct value *vc* to the erroneous value *ve*. For example, in an ADD instruction, the MCE (ADD, execute, load_flags, 1’b1, 1’b0) corresponds to incorrectly maintaining the contents of the flags in the ADD’s execute cycle.

MCEs are classified by their detectability as redundant (undetectable), invalid, or testable. Of these, only testable MCEs are targeted for test generation. A *redundant MCE* for

instruction *i* does not change the functions performed by *i*. The following conditions typically lead to redundant MCEs:

- *Unchanged visible state*: MCEs which do not affect the processor or memory state are redundant. These include: (*i*) reading a register or memory without storing a new result, (*ii*) loading a register or memory multiple times without reading it until some final value is loaded, and (*iii*) changing registers not visible to the instruction set, which are not used across several instructions.
- *Disabled signals*: MCEs on disabled signals are redundant. For example, an MCE that changes a select signal of a register file with a disabled read port will not affect instruction behavior.

Invalid MCEs violate usage constraints on modules, buses, or the overall microprocessor, for example:

- *Module input constraints*: These prevent inconsistencies such as: (*i*) reading and writing to memory in the same clock cycle, and (*ii*) setting the select bus of a 3-input multiplexer to 11.
- *Bus constraints*: These are bus usage rules such as: (*i*) a bus cannot have multiple active drivers at the same time, and (*ii*) a bus cannot be read if it has no data source, e.g., if it is in the high-impedance state.
- *Microprocessor constraints*: These are global operating constraints such as: (*i*) an instruction must be fetched every instruction cycle, and (*ii*) one and only one of the flags must be set.

Testable MCEs change a correct design to one with different functionality that meets all the specified design constraints. Detection of such MCEs requires instruction sequences that distinguish the correct design from erroneous ones. These sequences constitute tests for the modeled errors.

MCE evaluation. We evaluated the effectiveness of MCEs by an experiment similar to the one discussed in Section 3.4. The actual design errors are injected manually one at a time in the final, presumed correct design of LC-2. We then determine whether testing for all MCEs guarantees the detection of the injected design errors. This is done by deriving the

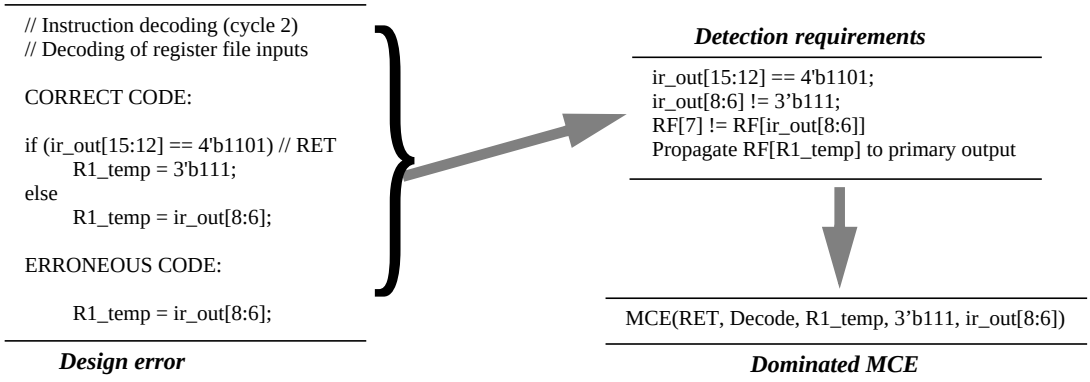


Figure 3.10 Example of an actual design error, its detection requirements, and the corresponding dominated MCE.

Table 3.4 Actual design errors and the number of corresponding dominated MCEs for LC-2.

	Actual design errors					No. of corresponding dominated MCEs
	Category	Total	Easily detected	Unde-tectable	Test-able	
Control Unit	Expression error	2	0	0	2	2
	Bit width error	1	1	0	0	0
	Missing assignment(s)	3	0	0	3	3
	Wrong constant(s)	1	0	0	1	1
	Unused signal	1	0	1	0	0
	Always statement	1	1	0	0	0
Datap-ath Unit	Wrong signal source(s)	3	0	0	3	1
	Bit width error	2	2	0	0	0
	Unused signal	1	0	1	0	0
	Wrong module	1	0	0	1	1
	Total	**16**	**4**	**2**	**10**	**8**

detection conditions for every actual error *e* and then determining if an MCE exists that is dominated by e. We applied this process to the complex actual error described earlier (Figure 3.8) and we were able to find a dominated MCE for it as shown in Figure 3.10. We analyzed manually all design errors in the test implementation of the LC-2 and the results are summarized in Table 3.4. A total of 16 design errors were found, nine in the control unit and the rest is in the datapath unit. Four of these errors are easily detected by the Verilog simulator, two are redundant, and the rest are testable. We can infer from Table 3.4 that all testable design errors in the LC-2 control unit are detected after simulation with tests for eight MCEs, and only two testable errors in the datapath unit are not guaranteed to be detected. However, by analyzing their detection requirements, we found that the probability of these two errors being undetected or masked is extremely low.

Validation approach. We now outline a microprocessor validation algorithm that generates test sequences for MCEs. As usual, the microprocessor's instruction set *IS* is defined by its ISA. The design constraints *CT* are derived from the ISA and the bus/module usage rules. We assume that a microprocessor implementation IM is given that consists of a control unit and a datapath unit; the problem is to verify IM. Both the ISA and IM are specified by a simulatable hardware description language (Verilog in our case).

The proposed verification algorithm is described in Figure 3.11 in five phases. The first phase identifies all relevant control/data symbols in each instruction. For example, the

Procedure MV(instruction set architecture *ISA*, constraints *CT*, implementation *IM*)

Phase 1	1	extract *IS* from *ISA*
	2	preprocess every instruction in *IS* to identify its fields
Phase 2	3	**for** every instruction *i* in *IS*
	4	**begin**
	5	**for** every instruction cycle
	6	**begin**
	7	simulate control and datapath units
	8	**if** any constraint from *CT* is violated **then**
	9	report {erroneous *IM*} and then stop
	10	**end**
	11	*MSI* := processor state in *IM* after simulating all cycles of *i*
	12	**for** every instruction cycle
	13	**begin**
	14	**for** every control signal *c* in *IM*
	15	**begin**
	16	*ci* := value of *c* in *IM*
	17	**for** every possible value *cm* of *c* not equal to *ci*
	18	**begin**
	19	inject the MCE (i.e. set *c* := *cm*) to form a mutant
	20	perform complete simulation of the mutant under *i*
	21	*MSM*:= final processor state in mutant
	22	**if** any constraint from *CT* is violated **then** MCE is INVALID
	23	**else if** (*MSI* == *MSM*) **then** MCE is REDUNDANT
	24	**else** add the TESTABLE MCE to error list
	25	**end**
	26	**end**
	27	**end**
	28	**end**
Phase 3	29	collapse the MCE list via dominance relations
Phase 4	30	set overall test sequence *S* := ϕ
	31	**while** there are more MCEs in the list
	32	**begin**
	33	select an MCE *m*
	34	generate an instruction sequence *s* to detect *m*
	35	remove all MCEs that are detected by *s*
	36	add *s* to *S*
	37	**end**
Phase 5	38	apply *S* to *IM* and *ISA*
	39	**if** the responses are different then report {erroneous *IM*}
	40	**else** report {correct *IM*}

Figure 3.11 The microprocessor validation algorithm.

16-bit LC-2 instruction ADD DR, SR1, SR2 is represented by a sequence of (name, location, value) symbols as follows:

(opcode,[15:12],0001), (DR,[11:9],N), (SR1,[8:6],N), (SR2,[2:0],N), (M,[5],0)

This sequence indicates that bits 15:12 of the instruction specify the opcode which is 0001 for ADD, bits 11:9 specify the destination register DR which is an unsigned integer (N), bits 8:6 and 2:0 specify the source registers (which are also unsigned integers), and finally bit 5 is a mode bit *M* which is set to 0. (*M* distinguishes ADD DR, SR1, SR2 from the instruction ADD DR, SR1, imm5, where imm5 is a signed 5-bit constant.) Note that phase 1 is based only on the microprocessor's ISA.

The second phase performs symbolic simulation of IM and its mutants, where a mutant is IM with a single injected MCE. For every instruction *i*, we first simulate the control unit cycle by cycle, and evaluate the resulting control signals originating from the control unit. Each such signal has the value undefined, constant, or symbolic; it is undefined if it is never assigned a value in the instruction cycle under consideration. We then simulate the datapath unit to compute the processor state at the end of the instruction cycle, and consequently determine if IM violates any specified design constraint. After simulating all cycles of *i*, we compute the final processor state *MSI*. For example, after simulating the ADD instruction described above, we end up with RF[DR] = RF[SR1] + RF[SR2], where RF denotes the register file.

Next the possible MCEs are injected one at a time and the resulting mutants are simulated for all cycles of *i* to obtain the final processor state *MSM*. By checking the constraints and comparing *MSI* to *MSM*, we can determine whether the current MCE is redundant, invalid, or testable. Redundant and invalid MCEs are dropped at this stage, while testable MCEs are inserted in the error list for later test generation.

The third phase in the verification algorithm is error collapsing to reduce the number of MCEs. Dominance among MCEs in the same instruction can be established for this purpose. An error e_1 is *dominated* by an error e_2 if any test for e_1 is also a test for e_2, in which case, e_2 can be dropped from the error list. Normally, some MCEs in cycle *i* of an instruction dominate others in cycle *j*, $i \leq j$, of the same instruction.

The fourth phase of the algorithm is test generation. Applying the instruction *i* is generally necessary to activate an MCE affecting *i*. We then may need instructions that justify the processor state needed to activate the MCE, and other instructions to propagate error values to the primary outputs of the processor.

The final phase of the algorithm applies the generated instruction sequence to both *IM* and *ISA*. If a difference is detected in the responses, the implementation is erroneous.

Example: To illustrate our validation methodology, we apply it here to the LC-2 instruction ADD DR, SR1, SR2. We define the state of the LC-2 microprocessor as the contents of all its storage elements, including the program counter (PC), instruction register (IR), memory-address register (MAR), flags register (FLAGS), register file (RF), and temporary registers (REG1 and REG2). The LC-2's initial state is thus (PC_0, IR_0, MAR_0, $FLAGS_0$, RF_0, $REG1_0$, $REG2_0$). Table 3.5 shows the control signal values in the implementation for the ADD instruction and the corresponding datapath actions. For every possible MCE *m*, we inject *m* into the implementation to form a mutant that is manually simulated to determine the type of *m*. The ADD instruction has a total of 58 MCEs of which 18 are testable. Examples of these MCEs include: (*i*) MCE32 (ADD, execute, load_pc_bar,0,1) which cor-

Table 3.5 Simulation of the instruction ADD DR, SR1, SR2: control signal values and corresponding datapath actions.

Simulation results	Instruction cycles		
	1: Fetch	2: Decode	3: Execute
Control signal values	read_mem_bar := 1'b0 write_mem_bar := 1'b1 load_pc_bar := 1'b1 RE1 := 1'b0 RE2 := 1'b0 WE := 1'b0 load_ir_bar := 1'b0 load_flags_bar := 1'b1 load_reg1_bar := 1'b1 load_reg2_bar := 1'b1 reg2_to_bus_bar := 1'b1 sel_ab_mux := 2'b00 R1 := SR1 R2 := SR2 W := DR	read_mem_bar := 1'b1 write_mem_bar := 1'b1 load_pc_bar := 1'b1 RE1 := 1'b1 RE2 := 1'b1 WE := 1'b0 load_ir_bar := 1'b1 load_flags_bar := 1'b1 load_reg1_bar := 1'b0 load_reg2_bar := 1'b0 reg2_to_bus_bar := 1'b1 R1 := SR1 R2 := SR2 W := DR	read_mem_bar := 1'b1 write_mem_bar := 1'b1 load_pc_bar := 1'b0 RE1 := 1'b0 S3 := 1'b1 RE2 := 1'b0 S2 := 1'b0 WE := 1'b1 S1 := 1'b0 load_ir_bar := 1'b1 S0 := 1'b1 load_flags_bar := 1'b0 M := 1'b1 load_reg1_bar := 1'b1 R1 := SR1 load_reg2_bar := 1'b1 R2 := SR2 reg2_to_bus_bar := 1'b1 W := DR zero_or_sign := 1'b1 sel_alu_mux := 1'b0 sel_rf_mux := 2'b00 sel_pc_mux := 2'b00
Corresponding datapath actions	MEM := MEM_0 IR := MEM[PC_0] PC := PC_0 FLAGS := $FLAGS_0$ REG1 := $REG1_0$ REG2 := $REG2_0$ RF := RF_0 MAR := MAR_0	MEM := MEM_0 PC := PC_0 IR := IR_P FLAGS := $FLAGS_0$ REG1 := RF[SR1] REG2 := RF[SR2] RF := RF_0 MAR := MAR_0	MEM := MEM_0 PC := PC_0 + 1 IR := IR_P FLAGS := Detect(REG1 + REG2) REG1 := $REG1_P$ REG2 := $REG2_P$ RF[DR] := REG1 + REG2 MAR := MAR_0

responds to the PC being stuck at the address of the ADD instruction in main memory and hence executing the ADD instruction indefinitely, and (ii) MCE48 (ADD, execute, S1, 0, 1) which corresponds to changing the ALU operation from plus to logical OR. To reduce the number of testable MCEs, dominance relations among MCEs are used. Of the 18 testable MCEs, only MCE32 can be removed by dominance— any instruction sequence that detects MCE48 will also detect MCE32.

In generating a test sequence for the MCEs of an instruction *i*, we first target MCEs in the last cycle of *i* with the hope that other MCEs in earlier cycles of *i* are detected by the generated sequence. The specifications of LC-2 give the starting PC address as 3000H. So, we start our PC value with a number larger than 3000H to give some space for justification of instructions, say 3080H. We generated manually the 10-instruction test sequence shown in Figure 3.12 to detect all 15 MCEs on control signals having constant values in the ADD instruction.

To get some idea of the total number of MCEs in the LC-2, we analyzed its instruction set and found that 430 MCEs (18.9%) are testable, 763 MCEs (33.5%) are invalid, and 1085 MCEs (47.6%) are redundant.

3.6 Discussion

The preceding experiments indicate that very good coverage of actual design errors in

```
307A:  0010 000 100000000          LD R0, 105H
307B:  0010 001 100000001          LD R1, 106H
307C:  0101 001 001 0 00 000       AND R2, R1, R0
307D:  1010 010 100000011          LDI R2, 103H
307E:  0010 000 100000000          LD R0, 100H
307F:  0010 001 100000001          LD R1, 101H
3080:  0001 001 001 0 00 000       ADD R1, R1, R0
3081:  0011 001 100000010          ST R1, 102H
3082:  0001 011 010 0 00 010       ADD R3, R2, R2
3083:  1000 010 100000100          BRZ 104H

3100:  0000 0000 0000 0110         Data = 6
3101:  0000 0000 0000 0101         Data = 5
3102:  xxxx xxxx xxxx xxxx         Storage
3103:  0011 0001 0000 0100         Data = 3104H
3104:  0000 000000000000           NOP
3105:  0000 0000 0000 0001         Data = 1
3106:  0000 0000 0000 0010         Data = 2
```

Figure 3.12 A test sequence for most MCEs in the ADD DR, SR1, SR2 instruction.

high-level designs can be obtained by complete test sets for a limited number of modeled error types, such as those defined by our basic and conditional error models. Thus our methodology can be used to construct focused test sets aimed at detecting a broad range of actual design bugs. More importantly, perhaps, it also supports an incremental design verification that can be implemented as follows: First, generate tests for SSL errors. Then generate tests for other basic error types such as BSEs. Finally, generate tests for conditional errors. As the number of SSL errors in a circuit is linear in the number of signals, complete test sets for SSL errors tend to be relatively small. In our experiments such test sets already detect at least half of the actual errors. To improve coverage of actual design errors and hence increase the confidence in the design, an error model with a quadratic number of error instances, such as BSE and CSSL1, can be used to guide test generation.

The conditional error models proved to be especially useful for detecting actual errors that involve missing logic. Most ‘missing module’ and ‘missing input’ errors in Table 3.3 cannot be covered when only the basic error types are targeted. However, all but one of them is covered when CSSL1 and CBOE errors are targeted as well. The same observation applies to the ‘missing assignment(s)’ errors in Table 3.2.

Moreover, our experimental results suggest that high coverage of data as well as control errors can be obtained by a test set for MCEs. An interesting observation is that most MCEs are either invalid or redundant—only 18.9% of the MCEs in the LC-2 are testable. This can significantly reduce the number of MCEs that need to be targeted by test generation. Moreover, the MCE model proved to be especially useful for detecting errors that involve missing logic—all ‘missing assignments(s)’ errors in the LC-2 control unit are covered by tests for MCEs.

The MCE error model and validation approach are, at least in principle, expandable to microprocessors with instruction pipelines, multiple instruction issue, etc. The definition of the MCE then needs to be generalized to (*I*,*c*,*s*,*vc*,*ve*), where *I* represents a sequence of one or more instructions. However, the complexity of the MCE model increases rapidly, so the applicability of this approach remains to be seen.

The designs used in the experiments are small, but appear representative of real indus-

trial designs. An important benefit of small-scale designs is that they allow us to analyze each actual design error in detail. The coverage results obtained strongly demonstrate the effectiveness of our model-based verification methodology. Furthermore the analysis and conclusions are independent of the manner of test generation. Nevertheless, further validation of the methodology using industrial-size designs is desirable, and will become more practical when CAD support for design error test generation becomes available.

Error models of the kind introduced here can provide metrics to assess the quality of a given verification test set. For example, full coverage of basic (unconditional) errors provides one level of confidence in the design, coverage of conditional errors of order $n \geq 1$ provides another, higher confidence level. Such metrics can also be used to compare test sets and to spur further directed test generation.

We envision the proposed methodology eventually being deployed as suggested in Figure 3.13. Given an unverified design and its specification, tests targeted at modeled design

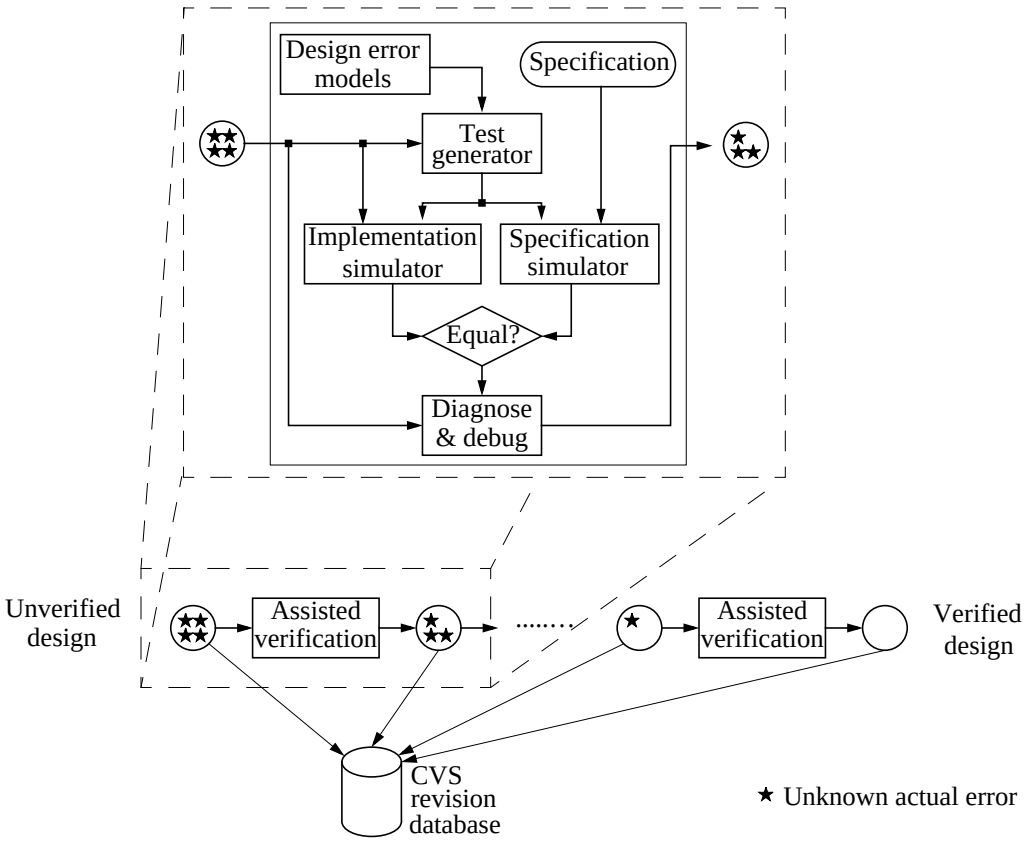


Figure 3.13 Deployment of proposed design verification methodology.

errors are automatically generated and applied to the specification and the implementation. When a discrepancy is encountered, the designer is informed and perhaps given guidance on diagnosing and fixing the error.

CHAPTER 4
BUILT-IN VALIDATION

Chapters 2 and 3 present gate-level and high-level validation methods aimed at detecting design errors by generating tests for them. These methods' goals were to generate a small number of tests that have high coverage of design errors and to apply the generated tests to software models of the specification and implementation. In this chapter, we are interested in detecting residual design errors and fabrication faults that may have escaped the detection during the design and manufacturing phases, and operational faults that appear during normal operation. However, to apply the tests on-line, i.e during normal operation, we need to efficiently generate the tests using built-in hardware or software.

Section 4.1 discusses the need for built-in validation and its achievement via built-in self-test (BIST). Section 4.2 reviews previous work on designing hardware test generators for BIST and Section 4.3 describes a new approach to designing scalable test sets and test generators. In Section 4.4 we apply this approach to carry-lookahead adders and several other examples.

4.1 Built-In Self-Test (BIST)

To reduce the cost of testing, design for testability (DFT) techniques are often used, where testability criteria are considered early in the design phase. BIST is a design-for-testability technique that places the testing functions physically with the circuit under test (CUT). It has several advantages over the alternative, external testing: (*i*) the ability to test in-system and at-speed, (*ii*) reduced test application time, (*iii*) less dependence on expensive test equipment, and (*iv*) the ability to automatically test devices on-line or in the field. On-line testing is especially important for high-integrity applications such as automotive

systems, in which we are interested.

When BIST is employed, a digital system is usually partitioned into a number of CUTs, each of which is logically configured as shown in Figure 4.1. In normal mode, a CUT receives its inputs *X* from other modules and performs the function for which it was designed. In test mode, a test pattern generator (TG) circuit applies a sequence of test patterns *S* to the CUT, and the test responses are evaluated by a response monitor (RM). This chapter concentrates on the design of TG, although we also consider some relevant aspects of RM. In the most common type of BIST, test responses are compacted in RM to form response signatures. The signatures are compared with reference signatures generated or stored on-chip, and the error signal indicates any discrepancies detected. We assume this type of response processing in the following discussion.

Although BIST is sometimes considered as a technique to facilitate manufacture testing, it is also useful for on-line testing. In on-line BIST, testing occurs during normal functional operating conditions. Non-concurrent BIST is carried out while the system is in an idle state. For example, the CUT can be configured for event-triggered testing, in which case, the BIST control can be tied to the system reset signal, so that testing occurs during system start-up or shutdown. Alternatively, concurrent BIST is carried out in parallel with normal system operation. For example, BIST can be designed as a periodic testing technique with low fault latency. This requires incorporating a testing process into the CUT that guarantees the detection of all target faults within a fixed time. A full test sequence need not be applied to the CUT all at once; instead, it can be partitioned and applied in

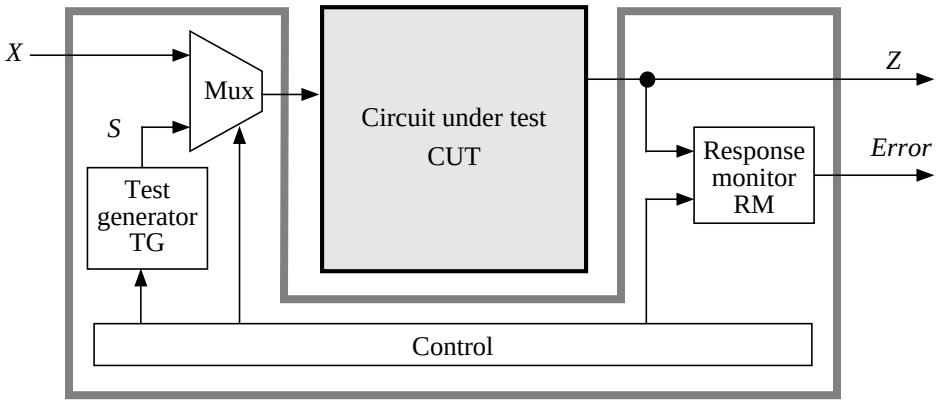


Figure 4.1 Generic BIST scheme.

periodic bursts.

Four primary parameters should be considered in developing a BIST methodology for digital systems; these correspond with the design parameters for on-line testing discussed earlier in Section 1.4.

- *Fault coverage*: This is the fraction of faults/errors of interest that can be exposed by the test patterns produced by the TG and detected by the RM. Most RMs produce the same signature for some faulty response sequences as for the correct response, a property called aliasing. This reduces fault coverage even if the tests produced by the TG provide full fault coverage. Safety-critical applications require very high fault coverage, typically 100% of the modeled faults.
- *Test set size*: This is the number of test patterns produced by the TG. This parameter is linked to fault coverage: generally, large test sets imply high fault coverage. However, for on-line testing either at system start-up or periodically during normal operation, test set size must be kept small to minimize impact on system resources and reduce fault latency.
- *Hardware overhead*: This is the extra hardware needed for BIST. In most applications, low hardware overhead is desirable.
- *Performance penalty*: This is the impact on performance of the normal circuit function, such as critical path delays, due to the inclusion of BIST hardware. In on-line BIST, the performance penalty is directly related to the extra time needed for testing, i.e. time redundancy.

We have been investigating the design of TGs in the four-dimensional design space defined by the above parameters with the goals of 100% fault coverage, very small test sets, and low hardware overhead. The specific CUTs we are targeting are high-speed data-path circuits to which most existing BIST methods are not applicable. Our CUTs are N-input, scalable, combinational circuits with large values of N (64 or more). They also employ carry lookahead, a common structure in high-performance datapaths. It is well-known that such circuits have small deterministic test sets that can be computed fairly easily. For example, it is shown in [51] that the standard n-bit carry-lookahead adder (CLA) design, which has $N = 2n + 1$ inputs, has easily-derived and provably minimal test sets for

all stuck-line faults; these test sets contain $N + 1$ test patterns. Some low-cost, scalable TG designs for datapath circuits based on C-testability (a constant number of test patterns independent of N) are known [50] [114], but they do not apply when CLA is used.

In the rest of this chapter, we describe a novel TG design methodology that addresses all the above issues, and illustrate it with several examples. The TG's structure is based on a twisted ring counter, and is tailored to generate a regular, deterministic test set of near-minimum size. Its hardware overhead is low enough to suggest that the TG can be incorporated into a standard cell or core design, as has been done for RAMs [90], adders [92] and multipliers [50]. For a modest increase in hardware overhead and test set size, our method can also minimize the performance penalty. The proposed approach covers the major types of fast arithmetic circuits, and appears to be generalizable to other CUT types as well.

4.2 Test Generator Design

A generic TG structure applicable to most BIST styles is shown in Figure 4.2 [34]. The sequence generator SG produces an m-bit-wide sequence of patterns that can be regarded as compressed or encoded test patterns, and the decoder DC expands or decodes these patterns into N-bit-wide tests, where N is the number of inputs to the CUT. Generally, $m \leq N$, and the SG can be some type of counter that produces all 2^m m-bit patterns.

The most common TG design is a counter-like circuit that generates pseudorandom sequences, typically a maximal-length linear feedback shift register (LFSR) [19], a cellular automaton [23], or occasionally, a nonlinear feedback shift register [42]. These designs basically consist of a sequence generator only, and have $m = N$. The resulting TGs are extremely compact, but they must often generate excessively long test sequence to achieve

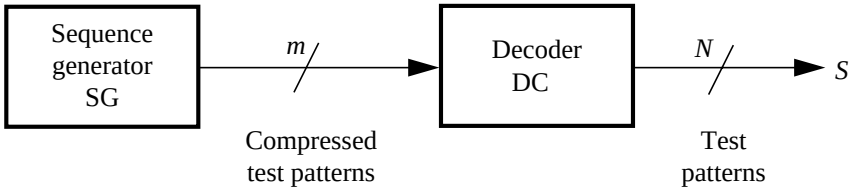


Figure 4.2 Basic structure of a test generation circuit.

acceptable fault coverage. Some CUTs, including the datapath circuits of interest, contain hard-to-detect faults that are detected by only a few test patterns T_{hard}. An N-bit LSFR can generate a sequence S that eventually includes $2^N - 1$ patterns (essentially all possibilities), however the probability that the tests in T_{hard} will appear early in S is low. Two general approaches are known to make S reasonably short. Test points can be inserted in the CUT to improve controllability and observability; this, however, can result in a performance loss. Alternatively, some determinism can be introduced into S, for example, by inserting "seed" tests for the hard faults. Such methods aim to preserve the cost advantages of LFSRs while making S much shorter. However, these objectives are difficult to satisfy simultaneously. It can also be argued that pseudorandom approaches represent "overkill" for datapath CUTs, which, like RAMs [90], seem much better suited to directed deterministic approaches.

Weighted random testing adds logic to a basic LFSR to bias the pseudorandom sequence it generates so that patterns from the desired test set T appear near the start of S [19]. In a related method proposed by Dufaza and Cambon [47], an LFSR is designed so that T appears as a square block at the beginning of S. A test set must usually be partitioned into many square blocks, and the feedback function of the LFSR must be modified after the generation of each block, making this method complex and costly. The approach of Hellebrand et al. [55] [56] modifies the seeds used by the LFSR, as well as its feedback function. In other work, Touba and McCluskey [110] describe mapping circuits that transform pseudorandom patterns to make them cover hard faults.

Another large group of TG design methods, loosely called deterministic or nonrandom, attempt to embed a complete test T of size P in a generated sequence S. A straightforward way to do this is to store T in a ROM and address each stored test pattern using a counter. SG is then a $\lceil \log P \rceil$-bit address counter and the ROM serves as DC. Unfortunately, ROMs tend to be too expensive for storing entire test sequences. Alternatively, a $\lceil \log P \rceil$-state finite state machine (FSM) that directly generates T can be synthesized. However, the relatively large values of P and N, and the irregular structure of T, are usually more than current FSM synthesis programs can handle.

Several methods have been proposed that, like a ROM-based TG, use a simple counter

for SG and design a low-cost combinational circuit for DC to convert the counter's output patterns into the members of *T* [9] [43]. Chen and Gupta [37] describe a test-width compression technique that leads to a DC that is primarily a wiring network. Chakrabarty et al. [34] explore the limits of test-pattern encoding, and develop a method for embedding *T* into test sequences of reasonable length.

Some TG design methods strive for balance between the straightforward generation of *T* using a ROM or FSM, and the hardware efficiency of an LFSR or counter. Perhaps the most straightforward way to do this was suggested by Agarwal and Cerny [6]. Their scheme directly combines the ROM and the pseudorandom methods. The ROM provides a small number of test patterns for hard-to-detect faults and the LFSR provides the rest of *T.*

None of the BIST methods discussed above explicitly addresses the scalability of the TG as the CUT is scaled to larger data word sizes. Scalable TGs based on C-testability have been described for iterative (bit-sliced) array circuits, such as ripple-carry adders [92] and array multipliers [50]. However, no technique has been proposed to design deterministic TGs that can be systematically rescaled as the size of a non-bit-sliced circuit, such as a CLA, is changed.

This next section introduces a class of TGs where SG is a compact (n + 1)-bit twisted ring counter and DC is CUT-specific. The output of SG can be efficiently decoded to produce a carefully crafted test sequence *S* that contains a complete test set for the CUT. As we will see, both SG and DC have a simple, scalable structure of the bit-sliced type. *S* is constructed heuristically to match a DC design of the desired type, so we can view this process as a kind of "co-design" of tests and their test generation hardware.

4.3 Scalable Test Generators

We first examine the scalability of the target datapath circuits and their test sets. A circuit or module $M(n)$ with the structure shown in Figure 4.3 is loosely defined as *scalable* if its output function $Z(n)$ is independent of the number n of its input data buses. Each such bus is w bits wide; there may also be a v-bit control bus, where w and v are constants independent of n. Bit-sliced arrays are special cases of scalable circuits in which each w-bit

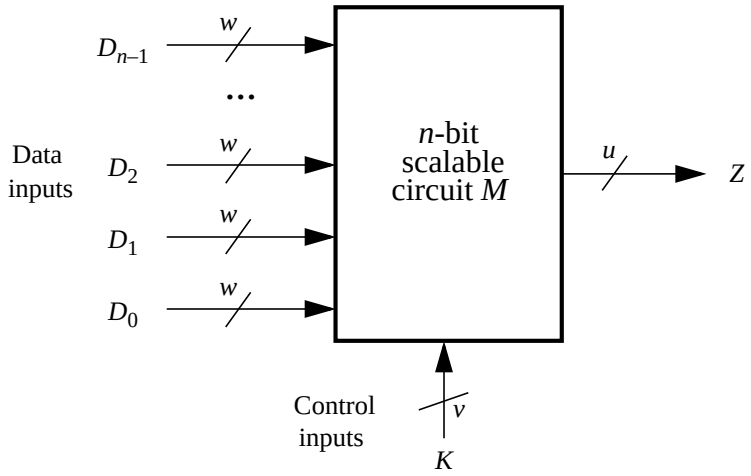


Figure 4.3 General scalable circuit.

input data bus corresponds to a slice or stage. Most datapath circuits compute a function $Z(A(n), B(n))$, where $A(n) = A_{n-1}\ldots A_1A_0$ and $B(n) = B_{n-1}\ldots B_1B_0$, and are scalable in the preceding sense. They can be expressed in a recursive form such as

$$Z(A(n+1), B(n+1)) = z[Z(\mathrm{A}(n), B(n)), A_n, B_n]$$

For example, if Z is addition, we can write

$$Z_{\text{add}}(A(n+1), B(n+1)) = Z_{\text{add}}(A(n), B(n)) + 2^n \times A_n + 2^n \times B_n$$

where the 2^n factor accounts for the shifted position of the new operand $D_n = (A_n,B_n)$. Similarly, a test sequence $S(n)$ for a scalable circuit $M(n)$ can be represented in recursive form. $S(n)$ is considered to be scalable if

$$S(A(n+1), B(n+1)) = s[S(\mathrm{A}(n), B(n)), A_n, B_n]$$

As we will see, the test scaling functions s and S can take a few regular, shift-like forms for the CUTs of interest.

To introduce our method, we use the very simple example of a ripple-carry incrementer shown in Figure 4.4. Here the carry-in line C_0 is set to 1 in normal operation, but is treated as a variable during testing. The increment function Z_{inc} can be expressed as

$$Z_{\text{inc}}(A(n+1)) = Z_{\text{inc}}(A(n)) + 2^n \times A_n + C_0 \qquad (4.1)$$

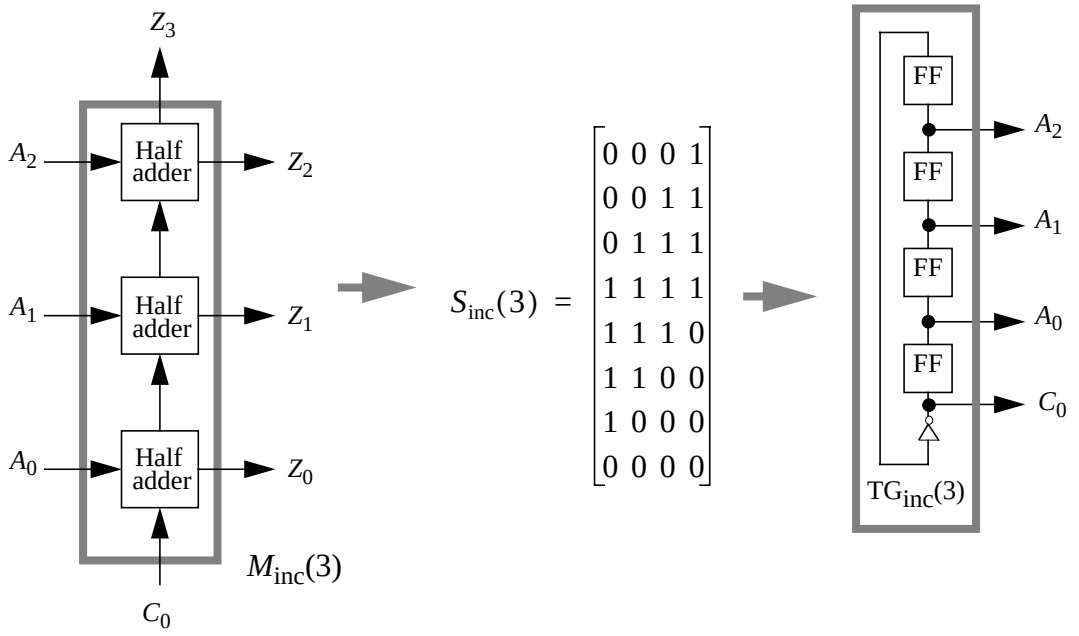
When $n = 1$, Equation (4.1) reduces to the half-adder equation

$$Z_{\text{inc}}(A(1)) = A_0 + C_0 \qquad (4.2)$$

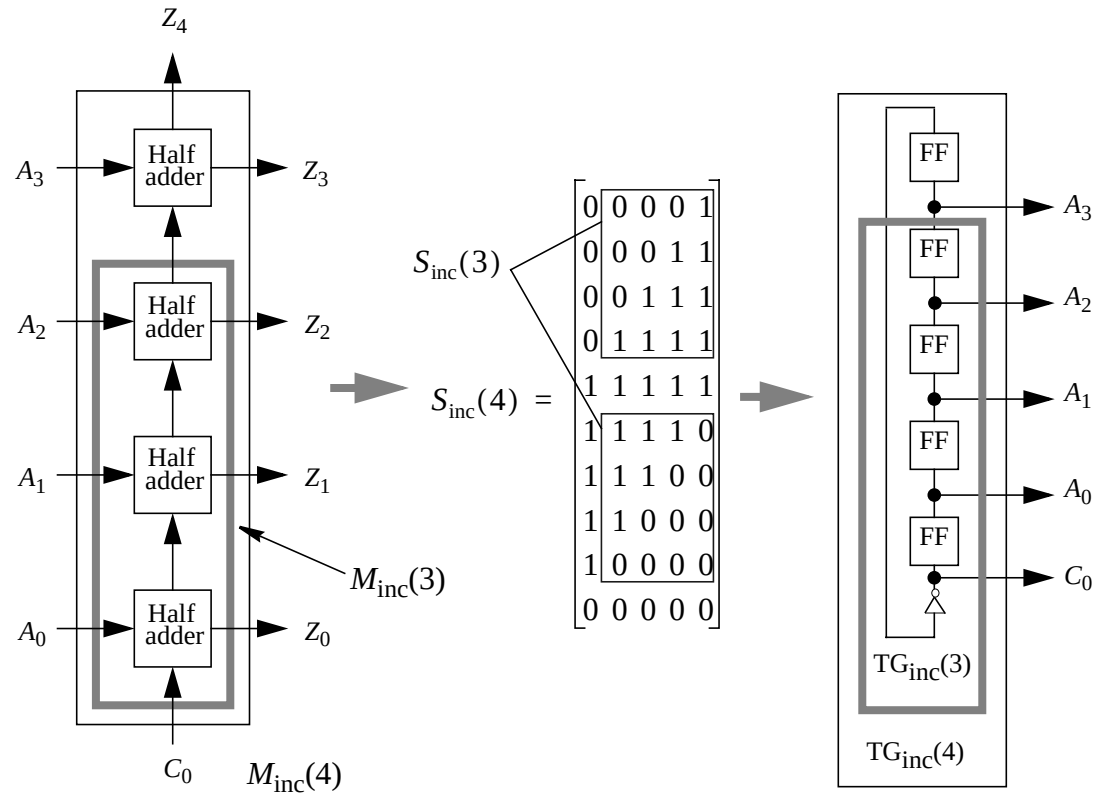
and (4.2) is realized by a single half-adder. An $(n + 1)$-bit incrementer $M_{\text{inc}}(n)$ is obtained by appending a half-adder stage to $M_{\text{inc}}(n - 1)$. Figure 4.4 shows how $M_{\text{inc}}(3)$ is scaled up to implement $M_{\text{inc}}(4)$.

A corresponding scaling of a test sequence $S_{\text{inc}}(n)$ for $n = 3$ to 4 is also shown in the figure. $S_{\text{inc}}(n)$ consists of $2n + 2$ test patterns of the form $A_{n-1}A_{n-2}\ldots A_0C_0$, each corresponding to a row in the binary matrices of Figure 4.4. These tests exhaustively test all half-adder slices of $M_{\text{inc}}(n)$ by applying the four patterns {00,01,10,11} to each half-adder and propagating any errors to the Z outputs. For example, the first test pattern $A_3A_2A_1A_0C_0 = 00001$ in $S_{\text{inc}}(4)$ applies 00 to the top three half-adders, and 01 to the bottom one. The next test 00011 applies 00 to the top two half-adders, 01 to the third half-adder from top, and 11 to the bottom one, and so on. If a fault is detected in, say, the bottom half-adder HA_0 by some pattern, an error bit appears either on Z_0 or on HA_0's carry-out line; in the latter case, the error will propagate to output Z_1, provided the fault is confined to HA_0. Thus $S_{\text{inc}}(n)$ detects 100% of all cell faults in the incrementer and, by extension, all SSL faults in $M_{\text{inc}}(n)$, independent of the internal implementation of the half-adder stages. The members of $S_{\text{inc}}(n)$ can easily be shown to constitute a minimal complete test with respect to cell faults, SSL faults, IP faults, GSEs, EGEs, and EIEs. Moreover, they also provide high coverage of MGEs, MIEs, and WIEs. Note that, unlike a ripple-carry adder, a ripple-carry incrementer such as $M_{\text{inc}}(n)$ is *not* C-testable, and can be shown to require at least $2n + 2$ tests for 100% fault coverage. This linear testing requirement is unusual in bit-sliced circuits, but is typical of CLA designs.

Each test in the sequences $S_{\text{inc}}(n)$ shown in Figure 4.4 has been carefully chosen to be a shifted version of the test above it. Moreover, the first $n + 1$ tests have been chosen to be bitwise complements of the second $n + 1$ tests. (We will see later that these special properties of $S(n)$ can be satisfied in other, more general datapath circuits.) The sequence of the $2(n + 1)$ test patterns of S is exactly the state sequence of an $(n + 1)$-bit twisted ring (TR) counter—this well-known circuit is also called a switch-tail, Johnson or Moebius counter



(*a*)



(*b*)

Figure 4.4 Scalable incrementer and the corresponding test sequence and test generator (twisted ring counter) for (a) *n* = 3 and (*b*) *n* = 4.

[83]. This immediately suggests that a suitable test generator $TG_{inc}(n)$ for $M_{inc}(n)$ is an $(n + 1)$-bit TR counter, as shown in Figure 4.4. Clearly $TG_{inc}(n)$ is also a scalable circuit. Thus we have a TG design conforming to the general model of Figure 4.2, in which SG is a TR counter and DC is vacuous.

Although at first glance, a TG like $TG_{inc}(4)$ seems to embody a large amount of BIST overhead given the small size of $M_{inc}(4)$, we can argue that, in fact, $TG_{inc}(4)$ is of near-minimal (if not minimal) cost. Assuming 10 test patterns are required, any TG in the style of Figure 4.2 requires an SG of 10 states, implying $\lceil \log_2 10 \rceil = 4$ flip-flops, plus an indeterminate amount of logic to implement DC. Our design uses 5 flip-flops—one more than the minimum—plus a single inverter. The fact that DC is vacuous in this particular case is consistent with a basic property of the TR counter: it is almost fully decoded. In contrast, a comparable $(2n + 2)$-state ring counter has $2n + 2$ flip-flops and is fully decoded, whereas an ordinary (binary) counter has $\lceil \log_2(2n + 2) \rceil$ flip-flops but is fully encoded. Thus we can hope to use TR counters in TGs that require little decoding logic.

As discussed in Chapters 1-3, tests for SSL faults detect several important types of design errors and physical faults. Hence, detecting all SSL faults is a primary goal in our general approach to designing TGs. We can now outline this approach for scalable datapath circuits. It uses high-level information about the CUT to explore in a systematic, but still heuristic, fashion a large number of its possible complete test sets to find a test set that has a regular, *shift-complement* (*SC)* structure resembling that illustrated by $S_{inc}(n)$ in Figure 4.4. The main steps involved are as follows:

1. Obtain a high-level, scalable model of the CUT $M(n)$.
2. Analyze this model using high-level functional analysis to derive a complete SSL-fault test set $T(n)$ for $M(n)$ for some small value of n. Use don't cares in the test patterns wherever feasible.
3. Convert $T(n)$ to an SC-style test sequence $S(n)$ as far as possible.
4. Synthesize a test generator $TG(n)$ for $S(n)$ in the style of Figure 4.5.

The test generator $TG(n)$ adds to the TR counter of Figure 4.4 a decoding array DC of identical combinational cells $DC_0, DC_1, \ldots, DC_{n-1}$ that modify the counter's output as needed by

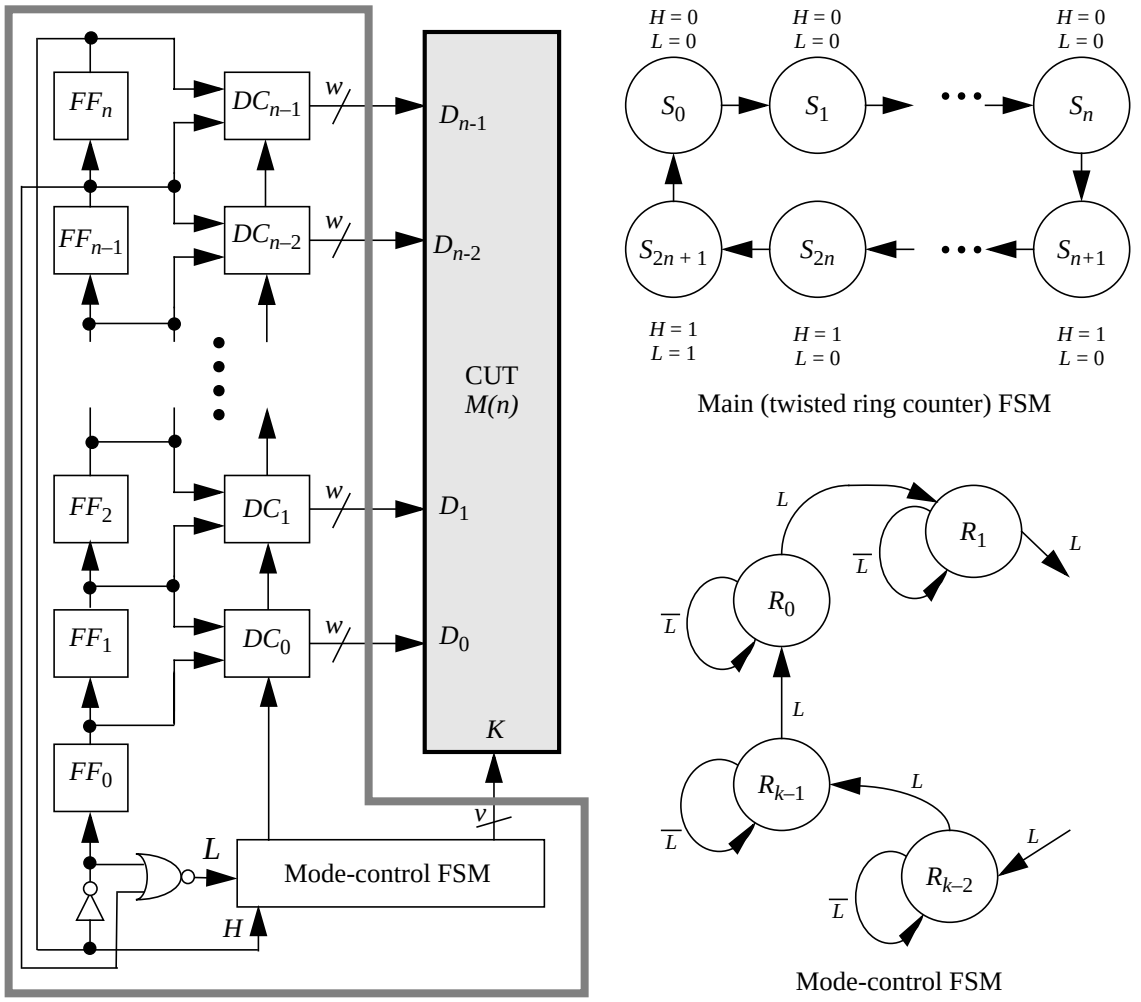


Figure 4.5 General structure of TG(*n*) and its state behavior.

a particular CUT. The array structure of *DC* ensures the scalability of TG. There is also a small mode-control FSM to allow *DC* to be modified for complex cases like multifunction circuits. The only inputs to the mode-control FSM are the signals *H* and *L*, which are active in the second half of the states of the TR counter and the last state, respectively. The state behavior of the TR counter and the mode-control FSM are shown in Figure 4.5; they have $2n + 2$ and k states, respectively, where k is a fixed number independent of n. The total number of states for TG(n) is thus $k(2n + 2)$, which approximates the number of tests in the test set $T(n)$.

Our use of functional, high-level circuit models to derive test sets (Step 1 and 2 above) is based on the work of Hansen and Hayes [52], who show that test generation for datapath

circuits can be done efficiently at the functional level while, at the same time, providing 100% coverage of low-level SSL faults for typical implementations. The model required for Step 1 is usually available for these types of circuits, since their scalable nature is exploited in their specification and carries through to high-level modeling during synthesis as illustrated by our incrementer example (Figure 4.4). Step 3 is perhaps the most difficult to formalize. It requires modifying and ordering the tests from Step 2 to obtain a sequence of shifted test patterns that resemble the output of the TR counter, but retain the full fault coverage of the original tests.

4.4 Design Examples

In this section, we apply the preceding approach to derive scalable test sets and test generators for CLAs and some other common datapath circuits.

Carry-Lookahead Adder. A CLA is a key component of many high-speed datapath circuits, including arithmetic-logic units and multipliers. A high-level model of a generic n-bit CLA $M_{CLA}(n)$, with the 4-bit 74283 [107] serving as a model, was derived in [51] and is shown in Figure 4.6. It is composed of (*i*) a module $M_{PGX}(n)$ that realizes the functions $P_i = A_i + B_i$, $G_i = A_i B_i$, and $X_i = A_i \oplus B_i$, (*ii*) a carry-lookahead generator (CLG) module $M_{CLG}(n)$ that computes all carry signals, and (*iii*) an XOR word gate that computes the sum outputs. The CLG module $M_{CLG}(n)$ contains the adder's hard-to-detect faults, and so is the focus of the test-generation process. Its testing requirements can be satisfied by generating tests for the SSL faults on the input lines of $M_{CLG}(n)$ that propagate the fault effects along the path to C_n, which is the longest and "hardest" fault-detection path. The resulting test set $T_{CLG}(n)$ contains $2n + 2$ tests and detects all faults in the CLG logic. For example, when $n = 2$, $T_{CLG}(2) = \{10101, 10110, 11000, 10100, 10001, 00111\}$, where the

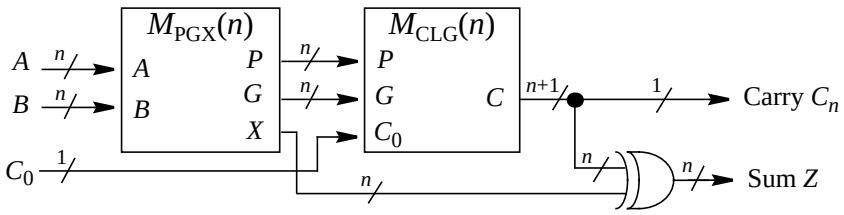


Figure 4.6 High-level model of the *n*-bit CLA.

test patterns are in the form $P_1G_1P_0G_0C_0$. Hansen and Hayes [52] have proven that such a test set detects all SSL faults in typical implementations of $M_{CLG}(n)$. Their method induces high-level functional faults from the SSL faults, and generates $T_{CLG}(n)$ for a small set of functional faults that cover all SSL faults. Because the carry functions are unate, it follows that $T_{CLG}(n)$ is a "universal" test set in the sense of [8], hence it covers all SSL faults in any inverter-free AND/OR implementation of $M_{CLG}(n)$.

Once the tests for $M_{CLG}(n)$ are known, they are traced back to the primary inputs of the $M_{CLA}(n)$ through the module $M_{PGX}(n)$; the resulting test sets for $n = 2$, are shown in Table 4.1a. The table gives a condensed representation of $M_{CLG}(2)$'s test requirements within $M_{CLA}(2)$, and specifies implicitly all possible sets of 6 tests (the minimum number) that cover all SSL faults in $M_{CLG}(2)$. For example, the first row in Table 4.1a defines the tests for the fault "C_0 fails to propagate 0 to C_2", which requires $C_0 = 1$ and $A_iB_i = 10$ or 01 for $i = 0$ and 1. Hence the potential tests for this fault are {10101, 10011, 01101, 01011}. The second row specifies the test for the faults "A_0 or B_0 fails to propagate 1 to C_2", which requires $A_0B_0 = 00$, but $A_iB_i = 10$ or 01 as before to ensure error propagation to C_2. To test for all SSL faults in module $M_{PGX}(n)$, each pair of bits A_iB_i must be exhaustively tested. The tests for $M_{CLG}(n)$ guarantee the application of 00 and 11 on each A_iB_i of $M_{PGX}(n)$, as we can see from Table 4.1a for the case of $n = 2$. Therefore, the remaining requirement for testing $M_{PGX}(n)$ is to apply 01 and 10 to each A_iB_i, as shown in Table 4.1b. The n XOR gates that feed the sum output Z are automatically covered by the tests for $M_{CLG}(n)$ and $M_{PGX}(n)$, and also provide non-blocking error propagation paths for these modules.

Once we know the possible test sets for $M_{CLA}(n)$, our next goal is to obtain a specific

Table 4.1 Condensed representation of complete test sets in (a) $M_{CLG}(2)$ and (b) $M_{PGX}(2)$. (c) Specific test sequence for the CLA that follow the SC style.

A_1 B_1	A_0 B_0	C_0
{10,01}	{10,01}	1
{10,01}	00	1
00	11	1
{10,01}	{10,01}	0
{10,01}	11	0
11	00	0

(a)

A_1 B_1	A_0 B_0	C_0
01	xx	x
10	xx	x
xx	01	x
xx	10	x

(b)

Test #	A_1 B_1	A_0 B_0	C_0
1	10	10	1
2	10	00	1
3	00	11	1
4	01	01	0
5	01	11	0
6	11	00	0

(c)

Table 4.2 Complete and minimal SC-style test sequence for the 74283 4-bit CLA and the corresponding responses.

Test #	Input pattern										Response				
	A_3	B_3	A_2	B_2	A_1	B_1	A_0	B_0	C_0		C_4	Z_3	Z_2	Z_1	Z_0
1	1	0	1	0	1	0	1	0	1		1	0	0	0	0
2	1	0	1	0	1	0	0	0	1		0	1	1	1	1
3	1	0	1	0	0	0	1	1	1		0	1	1	1	1
4	1	0	0	0	1	1	1	1	1		0	1	1	1	1
5	0	0	1	1	1	1	1	1	1		0	1	1	1	1
6	0	1	0	1	0	1	0	1	0		0	1	1	1	1
7	0	1	0	1	0	1	1	1	0		1	0	0	0	0
8	0	1	0	1	1	1	0	0	0		1	0	0	0	0
9	0	1	1	1	0	0	0	0	0		1	0	0	0	0
10	1	1	0	0	0	0	0	0	0		1	0	0	0	0

test sequence that follows the SC style. Such a test sequence of size 6 is extracted in Table 4.1c. This sequence is minimal and complete for SSL faults in the CLA [51], as can be verified by simulation. Tests 1, 2, and 3 are selected to make the 00 pattern applied to A_iB_i shift from right to left, as the shading in the table shows. Tests 4, 5, and 6 are selected to be the complements of tests 1, 2, and 3 respectively. Hence these tests shift the pattern 11 on A_iB_i from right to left. The specific test sequence $S_{\text{CLA}}(2)$ in Table 4.1c can be easily extended to a complete test sequence $S_{\text{CLA}}(n)$ of size $2n + 2$ for any $n > 2$. For example, Table 4.2 shows how $S_{\text{CLA}}(2)$ is scaled up to $S_{\text{CLA}}(4)$ to obtain a complete SC-style test sequence for the 74283 CLA.

The functional tests in $S_{\text{CLA}}(4)$ give complete coverage of SSL faults and high coverage of several design errors. Error simulation via ESIM shows that $S_{\text{CLA}}(4)$ detects more than 90% of the detectable gate-level design errors in the 74283 CLA.

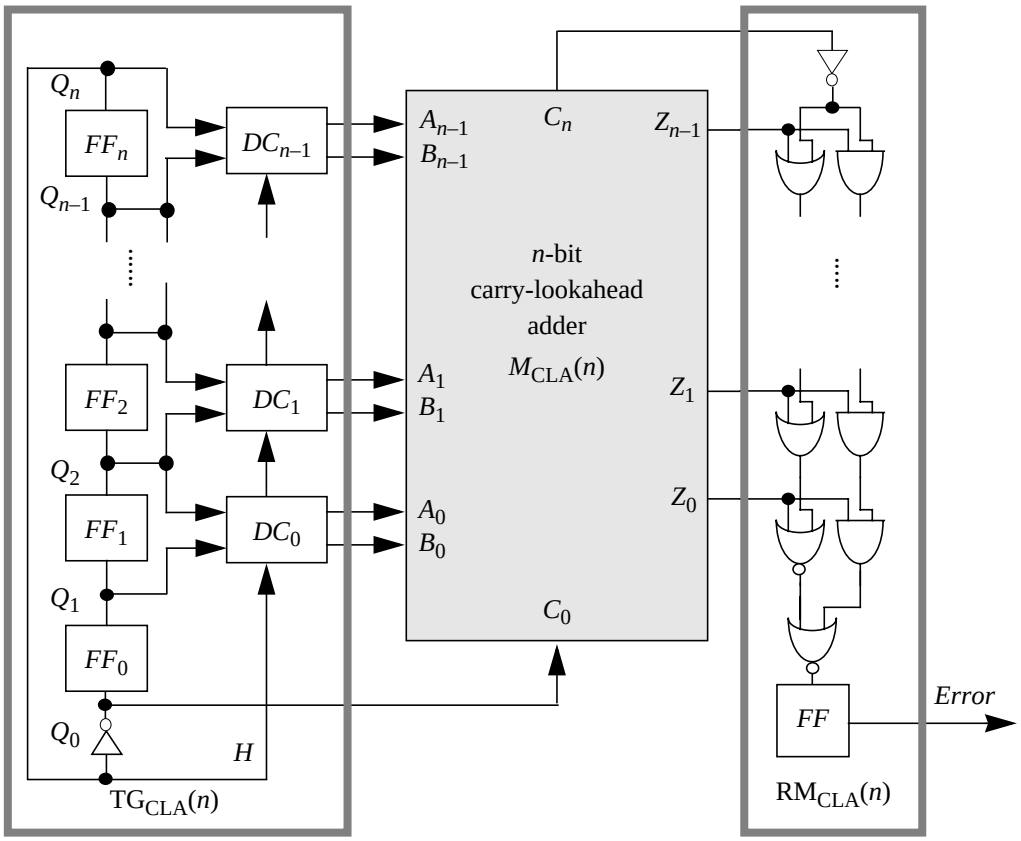
A test generator $TG_{\text{CLA}}(n)$ for $M_{\text{CLA}}(n)$ can now be synthesized from $S_{\text{CLA}}(n)$ following the general structure in Figure 4.5. As in the incrementer example, the sequence generator is an $(n + 1)$-bit TR counter. Note, however, that the number of input lines has almost doubled from $N = n + 1$ to $N = 2n + 1$. The size of $S_{\text{CLA}}(n)$ is $2n + 2$, which is the number of states of the TR counter, so no mode-control FSM is needed. Table 4.3 lists the CLA test sequence side by side with the TR counter's output sequence for the 4-bit case; the truth table of a decoder cell DC_i can be extracted directly, as shown in Figure 4.7. The combina-

Table 4.3 Mapping of the CLA test sequence to the TR counter's output sequence.

Test #	TR counter outputs					TG outputs (CLA test sequence)				
	H	Q_4Q_3	Q_3Q_2	Q_2Q_1	Q_1Q_0	$A_3\ B_3$	$A_2\ B_2$	$A_1\ B_1$	$A_0\ B_0$	C_0
1	0	0 0	0 0	0 0	0 0	1 0	1 0	1 0	1 0	1
2	0	0 0	0 0	0 0	0 1	1 0	1 0	1 0	0 0	1
3	0	0 0	0 0	0 1	1 1	1 0	1 0	0 0	1 1	1
4	0	0 0	0 1	1 1	1 1	1 0	0 0	1 1	1 1	1
5	0	0 1	1 1	1 1	1 1	0 0	1 1	1 1	1 1	1
6	1	1 1	1 1	1 1	1 1	0 1	0 1	0 1	0 1	0
7	1	1 1	1 1	1 1	1 0	0 1	0 1	0 1	1 1	0
8	1	1 1	1 1	1 0	0 0	0 1	0 1	1 1	0 0	0
9	1	1 1	1 0	0 0	0 0	0 1	1 1	0 0	0 0	0
10	1	1 0	0 0	0 0	0 0	1 1	0 0	0 0	0 0	0

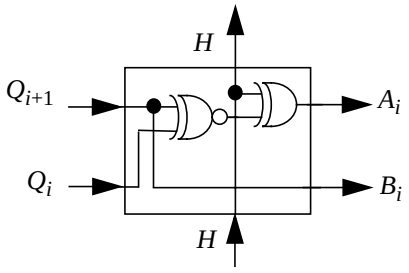
tions ($HQ_{i+1}Q_i$) = {010, 101} never appear at the inputs of the decoder cells, hence the outputs of DC_i are considered don't care for these combinations. Furthermore, the patterns ($HQ_{i+1}Q_i$) = {011, 100} never appear at the inputs of the high-order decoder cell DC_{n-1}, however, we choose not to take advantage of this, since our goal is to keep the decoder logic DC simple and regular. The carry-in signal C_0 can be seen from Table 4.3 to be $C_0 = \overline{H}$. The resulting design for $TG_{CLA}(n)$ shown in Figure 4.7 requires $n + 1$ flip-flops and n small logic cells that form DC. The hardware overhead of TG, as measured by transistor count in a standard CMOS implementation, amounts to 35.8% for a 32-bit CLA. This overhead decreases as the size of the CLA increases, a characteristic of all our TGs.

Our TGs, like the underlying TR counters, produce two sets of complementary test patterns. Such tests naturally tend to detect many faults because they toggle all primary inputs and outputs, as well as many internal signals. An n-bit adder also has the interesting property that A ***plus*** B ***plus*** $C_{in} = C_{out}S$ implies $\overline{A}$ ***plus*** $\overline{B}$ ***plus*** $\overline{C}_{in} = \overline{C}_{out}\overline{S}$, where ***plus*** denotes addition modulo 2^n. Hence the adder's outputs are complemented whenever a test is complemented, implying that there are only two distinct responses, 100...0 and 011...1, to all the tests in $TG_{CLA}(n)$, as can be seen from Table 4.2. Consequently, a simple, low-cost and scalable RM can be designed for the CLA adder as depicted in Figure 4.7. This example shows that some of the benefits of scalable, regular tests carry over to RM design.



H	$Q_{i+1}Q_i$	A_i B_i
0	0 0	1 0
0	0 1	0 0
0	1 1	1 1
1	1 1	0 1
1	1 0	1 1
1	0 0	0 0

Truth table of DC_i



DC_i circuit

Figure 4.7 Scalable test generator and response monitor for an *n*-bit CLA.

Arithmetic Logic Unit. We first consider an n-bit ALU $M_{ALU}(n)$ that employs the standard design represented by the 4-bit 74181 [107]. This ALU is basically a CLA with additional circuits that implement all 16 possible logic functions of the form $f(A,B)$. A high-level model for the 74181 is shown in Figure 4.8 [51], and consists of a CLG module M_2, a function select module M_1, and several word gates. Following the approach of the previous section, the tests needed for the CLG module M_2 are traced back to the ALU's primary

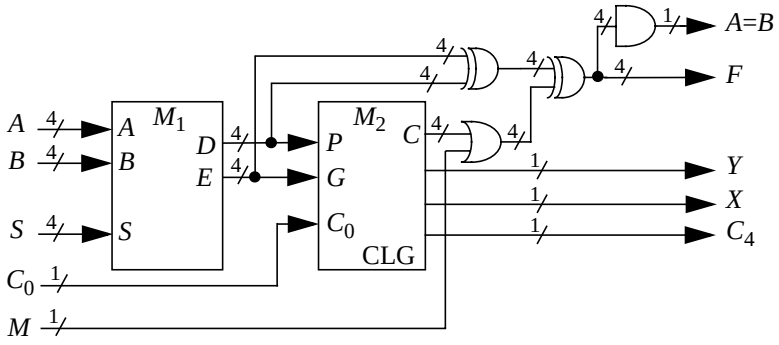


Figure 4.8 High-level model for the 74181 4-bit ALU.

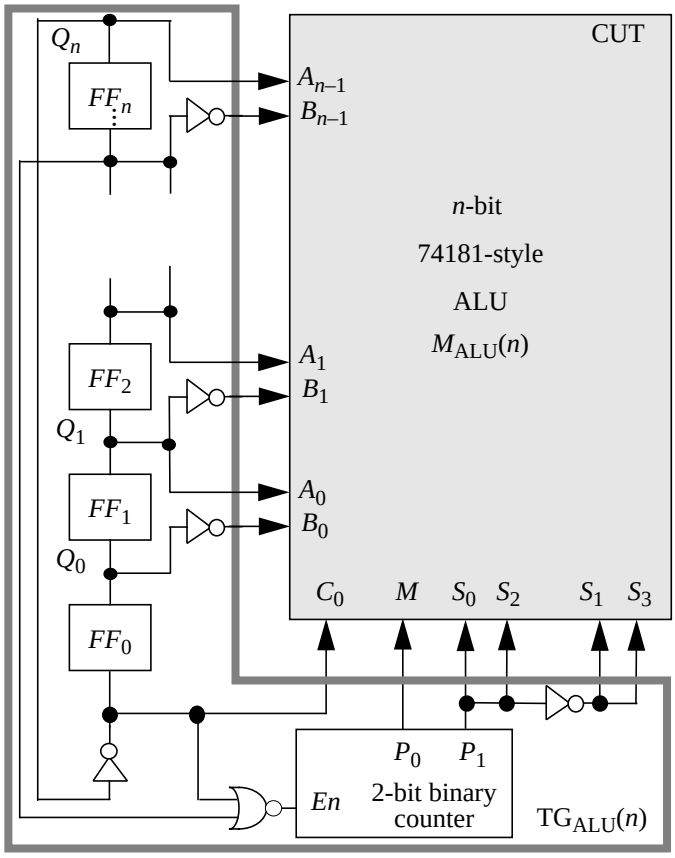
inputs. During this process, the signal values applied to the function-select control bus S are chosen to satisfy the testing needs for M_1 as well. An obvious choice is to make S select the add ($S_3S_2S_1S_0$ = 1001) and subtract ($S_3S_2S_1S_0$ = 0110) modes of the ALU. However, we found by trial and error that the assignments $S_3S_2S_1S_0$ = 1010 and 0101 lead to a TG design with less overhead. The testing needs of the word gates in the high-level model of the ALU must be also considered. The final test sequence $S_{\text{ALU}}(n)$ has an SC structure that closely resembles that of the CLA. Table 4.4 shows $S_{\text{ALU}}(4)$; note how the tests exhibit the same shifting property as before for the patterns $A_iB_i = 11$ and $A_iB_i = 00$. Moreover, tests 1:20 are the complements of tests 21:40. The test sequence $S_{\text{ALU}}(4)$ is not minimal, however, since 12 tests are sufficient to detect all SSL faults in the 74181 [51]. The tests in $S_{\text{ALU}}(4)$ are functional, so they have high coverage of several design errors. Error simulation via ESIM shows that $S_{\text{ALU}}(4)$ detects more than 95% of the detectable gate-level design errors in the 74181 ALU. $S_{\text{ALU}}(4)$ can be easily extended to $S_{\text{ALU}}(n)$ with a near-minimal size of $8n + 8$.

A test generator $\text{TG}_{\text{ALU}}(n)$ for $M_{\text{ALU}}(n)$ is shown in Figure 4.9, which again follows the general test generator model of Figure 4.5. Since the test sequence size is $8n + 8$ and the general test generator has $k(2n + 2)$ states, the mode-select FSM of $\text{TG}_{\text{ALU}}(n)$ has $k = 4$ states. The state table of the mode-select FSM and the truth table of the decoder cell are shown in Figure 4.9. The decoder cell DC_i turns to be extremely simple in this case—a single inverter. The overall test generator $\text{TG}_{\text{ALU}}(n)$ requires $n + 3$ flip-flops, n inverters, and a small amount of combinational logic whose size is independent of n. The hardware

Table 4.4 Complete and near-minimal SC-style test sequence for the 74181 ALU.

Test #	A_3	B_3	A_2	B_2	A_1	B_1	A_0	B_0	C_0	M	S_3	S_2	S_1	S_0
1	0	1	0	1	0	1	0	1	1	0	1	0	1	0
2	0	1	0	1	0	1	0	0	1	0	1	0	1	0
3	0	1	0	1	0	0	1	0	1	0	1	0	1	0
4	0	1	0	0	1	0	1	0	1	0	1	0	1	0
5	0	0	1	0	1	0	1	0	1	0	1	0	1	0
6	1	0	1	0	1	0	1	0	0	0	1	0	1	0
7	1	0	1	0	1	0	1	1	0	0	1	0	1	0
8	1	0	1	0	1	1	0	1	0	0	1	0	1	0
9	1	0	1	1	0	1	0	1	0	0	1	0	1	0
10	1	1	0	1	0	1	0	1	0	0	1	0	1	0
11	0	1	0	1	0	1	0	1	1	1	1	0	1	0
12	0	1	0	1	0	1	0	0	1	1	1	0	1	0
13	0	1	0	1	0	0	1	0	1	1	1	0	1	0
14	0	1	0	0	1	0	1	0	1	1	1	0	1	0
15	0	0	1	0	1	0	1	0	1	1	1	0	1	0
16	1	0	1	0	1	0	1	0	0	1	1	0	1	0
17	1	0	1	0	1	0	1	1	0	1	1	0	1	0
18	1	0	1	0	1	1	0	1	0	1	1	0	1	0
19	1	0	1	1	0	1	0	1	0	1	1	0	1	0
20	1	1	0	1	0	1	0	1	0	1	1	0	1	0
21	0	1	0	1	0	1	0	1	1	0	0	1	0	1
22	0	1	0	1	0	1	0	0	1	0	0	1	0	1
23	0	1	0	1	0	0	1	0	1	0	0	1	0	1
24	0	1	0	0	1	0	1	0	1	0	0	1	0	1
25	0	0	1	0	1	0	1	0	1	0	0	1	0	1
26	1	0	1	0	1	0	1	0	0	0	0	1	0	1
27	1	0	1	0	1	0	1	1	0	0	0	1	0	1
28	1	0	1	0	1	1	0	1	0	0	0	1	0	1
29	1	0	1	1	0	1	0	1	0	0	0	1	0	1
30	1	1	0	1	0	1	0	1	0	0	0	1	0	1
31	0	1	0	1	0	1	0	1	1	1	0	1	0	1
32	0	1	0	1	0	1	0	0	1	1	0	1	0	1
33	0	1	0	1	0	0	1	0	1	1	0	1	0	1
34	0	1	0	0	1	0	1	0	1	1	0	1	0	1
35	0	0	1	0	1	0	1	0	1	1	0	1	0	1
36	1	0	1	0	1	0	1	0	0	1	0	1	0	1
37	1	0	1	0	1	0	1	1	0	1	0	1	0	1
38	1	0	1	0	1	1	0	1	0	1	0	1	0	1
39	1	0	1	1	0	1	0	1	0	1	0	1	0	1
40	1	1	0	1	0	1	0	1	0	1	0	1	0	1

overhead decreases as the number of inputs *n* of the ALU increases, and it amounts to 11.4% for a 32-bit ALU.



Present state	Next state	Present outputs				
		M	S_3	S_2	S_1	S_0
R_0	R_1	0	1	0	1	0
R_1	R_2	1	1	0	1	0
R_2	R_3	0	0	1	0	1
R_3	R_0	1	0	1	0	1

State table of the mode-select FSM

H	$Q_{i+1}Q_i$	A_i B_i
0	0 0	0 1
0	0 1	0 0
0	1 1	1 0
1	1 1	1 0
1	1 0	1 1
1	0 0	0 1

Truth table of DC_i

Figure 4.9 Test generator for an *n*-bit 74181-style ALU.

Multiply-Add Unit. Our next example introduces another important arithmetic operation, multiplication. The high-level model and some implementation details of the target $n \times n$-bit multiply-add unit (MAU) $M_{MAU}(n)$ are shown in Figure 4.10. The MAU composed of a cascaded sequence of carry-save adders followed by a CLA in the last stage. This design is faster than a normal multiply-add unit where the last stage is a ripple-carry adder [16] [69].

Following our general methodology, we first analyze a small version of MAU, in this instance, the 4-bit case. Again the tests for the CLA (Table 4.2) are traced back to the primary inputs through the cell array. The primary input signals are selected to preserve the shifting structure of the CLA tests. The resulting MAU tests do not test the cell array completely—two SSL faults per cell remain undetected. These undetected faults require two extra tests, leading to a complete test set of size 12. Once the possible test sets are determined, a sequence that has the desired SC structure is constructed. Table 4.5 shows a pos-

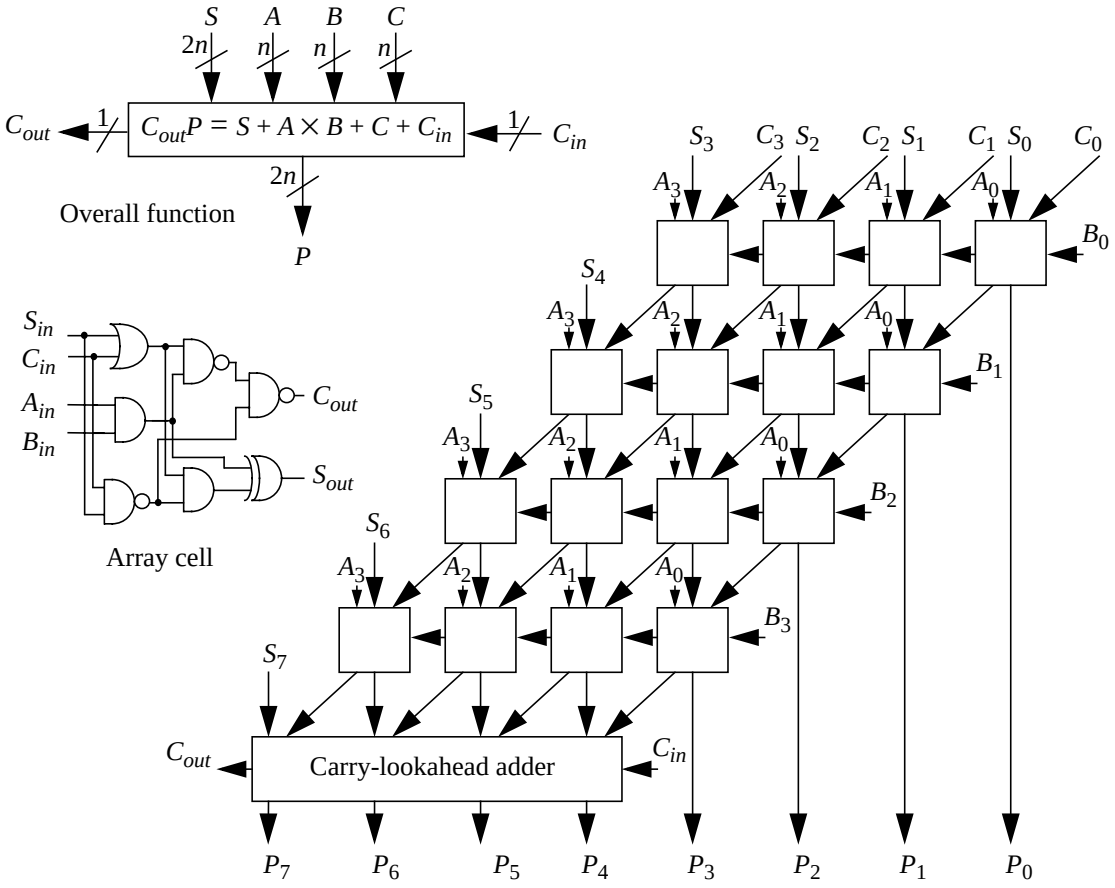


Figure 4.10 High-level model for the multiply-add unit.

sible test sequence $S_{\text{MAU}}(4)$ of size 20 for $M_{\text{MAU}}(4)$. This test sequence can be easily extended to $M_{\text{MAU}}(n)$, resulting in a test of size $4n + 4$.

A test generator $\text{TG}_{\text{MAU}}(n)$ for $M_{\text{MAU}}(n)$ in the target style is shown in Figure 4.11. Since the test sequence size is $4n + 4$ and the general test generator TG(n) has $k(2n + 2)$ states, the mode-select FSM has $k = 2$ states (one flip-flop). The state table of the mode-select FSM and the truth table for DC_i are shown in Figure 4.11. The hardware overhead of $\text{TG}_{\text{MAU}}(n)$ is estimated to be only 0.8% for a 32×32-bit multiply-add unit.

Booth multiplier. Our technique can also be applied with some minor modifications to a fast Booth multiplier that is composed of a cascaded sequence of carry-save adders followed by a final stage consisting of a 2n-bit CLA [16]. This design is faster than the usual Booth multiplier where the last stage is a ripple-carry adder; test generation has been stud-

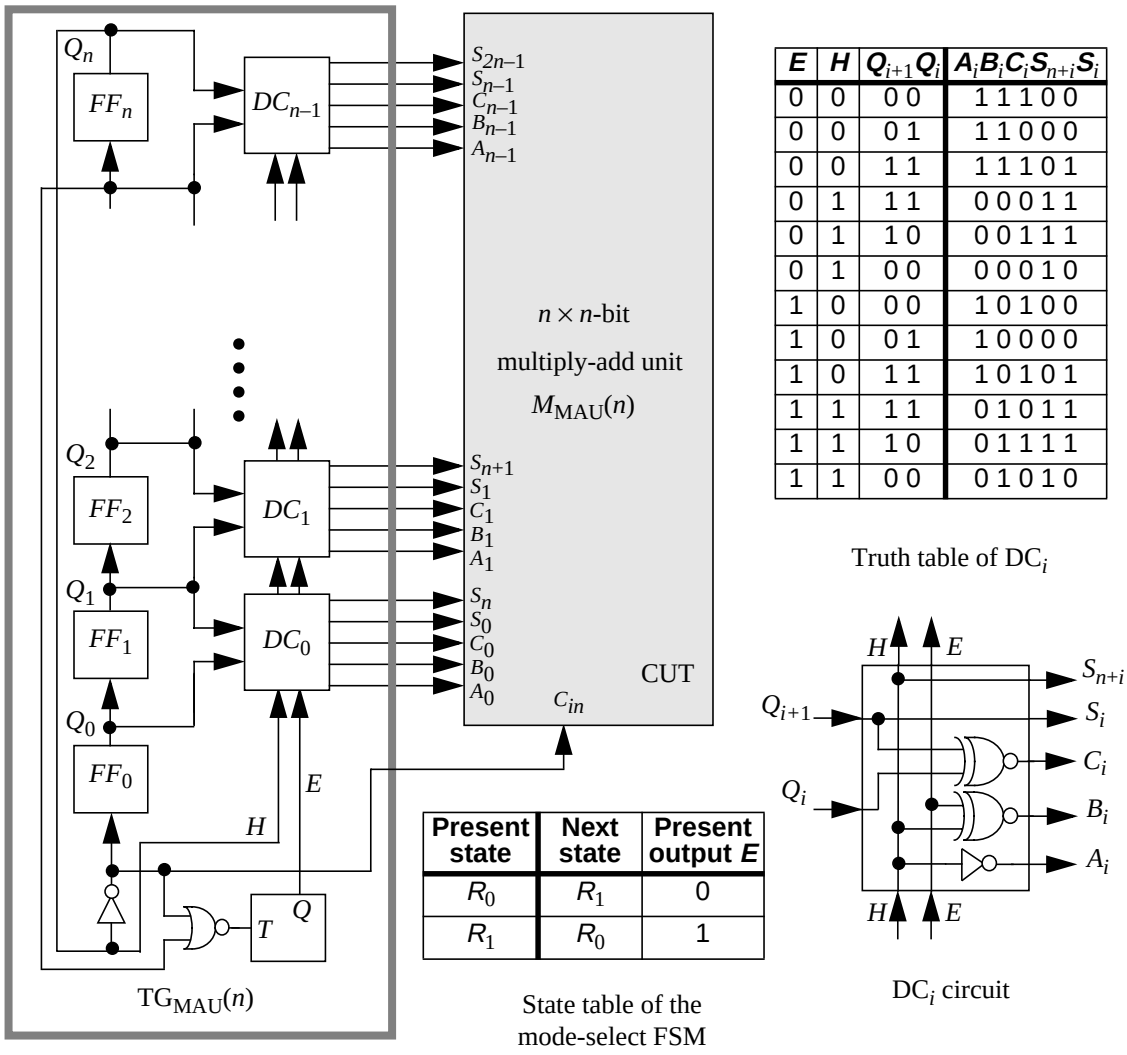
Table 4.5 Complete and near-minimal SC-style test sequence for the multiply-add unit.

Test #	$A_3B_3C_3S_7S_3$	$A_2B_2C_2S_6S_2$	$A_1B_1C_1S_5S_1$	$A_0B_0C_0S_4S_0$	C_{in}
1	11100	11100	11100	11100	1
2	11100	11100	11100	11000	1
3	11100	11100	11000	11101	1
4	11100	11000	11101	11101	1
5	11000	11101	11101	11101	1
6	00011	00011	00011	00011	0
7	00011	00011	00011	00111	0
8	00011	00011	00111	00010	0
9	00011	00111	00010	00010	0
10	00111	00010	00010	00010	0
11	10100	10100	10100	10100	1
12	10100	10100	10100	10000	1
13	10100	10100	10000	10101	1
14	10100	10000	10101	10101	1
15	10000	10101	10101	10101	1
16	01011	01011	01011	01011	0
17	01011	01011	01011	01111	0
18	01011	01011	01111	01010	0
19	01011	01111	01010	01010	0
20	01111	01010	01010	01010	0

ied before only for the slower, ripple-carry design [50]. We have been able to derive a complete scalable test sequence of size $4n + 14$ for the CLA-based Booth multiplier. The corresponding test generator TG(n) contains a TR counter with $n + 1$ flip-flops and a 10-state mode-control FSM with 5 flip-flops. The hardware overhead is estimated to be 5.3% for a 32×32-bit multiplier.

4.5 Discussion

Built-in testing and validation are potentially important features of digital systems, not only for critical applications, but also to satisfy the high-availability requirements of common consumer products as well. To achieve the twin goals of high fault/error coverage and low error latency, hardware features for testing and monitoring must be included. One such hardware feature is BIST, a technique occasionally used in manufacture testing and widely promoted for on-line testing. We have described how BIST can be employed to detect design errors and physical faults. Further research is needed to determine how best



E	H	$Q_{i+1}Q_i$	$A_iB_iC_iS_{n+i}S_i$
0	0	0 0	1 1 1 0 0
0	0	0 1	1 1 0 0 0
0	0	1 1	1 1 1 0 1
0	1	1 1	0 0 0 1 1
0	1	1 0	0 0 1 1 1
0	1	0 0	0 0 0 1 0
1	0	0 0	1 0 1 0 0
1	0	0 1	1 0 0 0 0
1	0	1 1	1 0 1 0 1
1	1	1 1	0 1 0 1 1
1	1	1 0	0 1 1 1 1
1	1	0 0	0 1 0 1 0

Truth table of DC_i

Present state	Next state	Present output E
R_0	R_1	0
R_1	R_0	1

State table of the mode-select FSM

DC_i circuit

Figure 4.11 Test generator for an $n \times n$-bit multiply-add unit.

to systematically integrate hardware features for built-in testing and validation with software to enhance safety and reliability in digital systems.

We have also presented a new approach to the design of scalable hardware test generators for BIST, and illustrated it for several practical datapath circuits. The resulting test generators produce extremely small test sets that have complete coverage of SSL faults and high coverage of design errors; they are of minimal or near-minimal size for all examples covered. Small test sets of this kind are essential for the on-line use of BIST, especially in applications requiring fast arithmetic techniques like carry-lookahead, for which previously proposed BIST schemes are not well suited. The TGs proposed here also have

Table 4.6 Summary of the scalable test generator examples.

Circuit(s)		SSL fault coverage	Regular test set size	Hardware overhead %			
				n = 4	*n* = 8	*n* = 16	*n* = 32
Carry-lookahead adder (CLA)		100%	$2n + 2$	45.5	40.1	36.9	35.8
Arithmetic-logic unit (ALU)		100%	$8n + 8$	23.2	16.1	12.9	11.4
Multiply-add unit (MAU)		100%	$4n + 4$	7.8	3.5	1.6	0.8
Booth multiplier		100%	$4n + 14$	32.9	18.0	9.9	5.3
A combination of ALU, MAU, and registers	Separate TGs	100%	$8n + 8$	9.8	5.7	3.3	1.8
	Combined TG			6.2	3.6	2.1	1.1

low hardware overhead, and are easily expandable to test much larger versions of the same target CUT.

Table 4.6 summarizes the results obtained for the scalable TGs we have designed so far. The first part of the table contains the results for the circuits discussed in Section 4.4. The average hardware overhead for the ALU, MAU, and Booth multiplier with $n = 32$ is around 6%. The table also indicates how the overhead decreases as n increases from 4 to 32. The overhead for the MAU shrinks by 90%, and the average decrease for all the circuits is 61%.

When applying BIST in a system, designers usually try to take advantage of existing flip-flops and logic already present in or around the CUT. For a typical datapath in, say, a digital signal processing circuit, all the data inputs to ALUs or multipliers come from a small register file. These registers can be designed to be reconfigured into TR counters like that in Figure 4.5, thus eliminating the need for special flip-flops in SG. Similar schemes have been proposed in prior techniques such as BILBO [19]. Moreover, it may be possible to share the resulting SGs among several CUTs. Multiplexing logic will then be needed to select the DCs for individual CUTs during test mode but circumvent them during normal operation. For a small additional increase in circuit complexity, time-multiplexing can be used to select the DCs in test mode, while avoiding the performance penalty associated with multiplexers.

In some cases, it may be feasible to share the entire TG. To illustrate this possibility,

consider an n-bit ALU, an $n \times n$-bit MAU, and a register file connected to a common bus. A single, reconfigurable TG attached to the bus can test both arithmetic units. The results of this approach are summarized in Table 4.6 for various values of n, and suggest that replacing separate TGs for the ALU and MAU by a single combined TG reduces overhead by about a third.

Our TG designs shed some light on the following interesting, but difficult question: How much overhead is necessary for built-in test generation? As we noted in the incrementer case, the size of the $TG_{inc}(4)$ must be close to minimal for any TG that is required to produce a complete test sequence of near-minimal length. The same argument applies to $TG_{CLA}(4)$, since it has 5 flip-flops in SG and a small amount of combinational logic in DC; any test generator G(4) producing the same number of tests (12) must contain at least 4 flip-flops in its SG. In general, the overhead of a TR-counter-based design TG(n) scales up linearly and slowly with n. The number of flip-flops in some other test generator G(n) may increase logarithmically with n, but the combinational part of G(n) is likely to scale up at a faster rate than that of TG(n). This suggest that even if the overhead of TG(n) is considered high, it may not be possible to do better using other BIST techniques under similar overall assumptions. If the constraints on test sequence length are relaxed, simpler TGs for datapath circuits may be possible, but such designs have yet to be demonstrated.

CHAPTER 5
CONCLUSIONS

This chapter reviews the major contributions of this thesis and discusses some directions for further research.

5.1 Thesis Contributions

Due to increases in design complexity and shorter design cycles, design errors are more likely to escape detection and hence lead to high-cost field failure. Moreover, operational faults, which occur during normal operation, can also lead to high-cost field failure especially in high-availability and safety-critical applications. To increase the reliability of digital systems and to reduce the cost of failure, we have developed a lifetime validation methodology that targets the detection of design errors, fabrication faults, and operational faults throughout the lifetime of a digital system. The major contributions of this thesis are summarized below.

- A simulation-based gate-level design validation method that uses conventional ATPG techniques for SSL faults to generate the verification tests.
- A rigorous analysis of the gate-level design error models and a systematic method to map them into SSL faults.
- A fault/error simulator ESIM that can handle several gate-level design error models and physical fault types.
- A design validation method for high-level designs that is based on modeling design errors and generating simulation vectors for them. The basic and conditional design error models are derived from actual error data.
- The concept of mutation control errors and a validation algorithm based on the

detection of these errors.

- A design method for built-in functional test generation aimed at high-performance, scalable datapath circuits.

The proposed lifetime validation methodology for digital systems is based on the following steps: (1) modeling the faults in the system at each stage in its lifetime, (2) generating tests to detect the modeled faults, and (3) applying the generated tests to the circuit under test and monitoring the responses to detect any deviations from the specifications. This methodology has often been used in manufacture testing; this thesis shows how to apply it to both design verification and on-line testing.

For design verification, we have shown how to model gate-level design errors and generate tests for them. Our results suggest that most gate-level errors are detected by tests aimed at a small number of synthetic and relatively simple error models. We have also shown how to model some basic types of high-level design errors and how to generate tests for them. Our experimental results also indicate that very high coverage of actual design errors can be obtained with test sets that are complete for a few types of error models. We have also developed a systematic validation algorithm for the detection of control errors in microprocessor-like circuits and applied it to a small microprocessor, the LC-2.

We have shown that on-line BIST is an attractive option for detecting residual design errors and physical faults. On-line BIST can achieve full error coverage, bounded error latency, low hardware and time redundancy. We have introduced a method for the design of efficient test sets and test-pattern generators for BIST, where the target applications are high-performance, scalable datapath circuits. Our hardware test generators meet the following desirable goals: scalability, small test set size, full fault coverage, and very low hardware overhead.

5.2 Future Research

In this final section, we discuss several possible extensions of the results presented in the thesis.

Design Error Diagnosis and Correction: In a typical design process, many iterations are

performed before the final design is manufactured on an IC. Since a design error causes all manufactured ICs to be faulty, design error diagnosis is more important than error detection. Each time a design error is detected, the design is inspected to determine the location of the error and correct it. Normally, the correction process is carried out manually by human designers. Once the IC is manufactured, it is also tested to determine if it behaves incorrectly due to fabrication faults. If the production yield is high then detecting fabrication faults is more important than fault diagnosis. However, if the yield is low, then the IC is diagnosed to determine the source of the fault and consequently improve the yield of future manufacturing.

The problem of gate-level design error diagnosis is similar to SSL fault location. Testing is the basic method used for fault diagnosis, where we need tests that distinguish non-equivalent faults from one another [4]. A complete diagnosis test set distinguishes between every pair of (distinguishable) faults in the circuit. The result of the fault diagnosis process is usually a fault dictionary that allow us to identify the fault from observing the output of the faulty circuit in response to the complete diagnostic test set.

Several researchers have addressed the problem of automatic location of gate-level design errors. However, the methods previously suggested are either computationally expensive [79] or do not guarantee finding the location of every error [48][109]. Moreover, all of the proposed design error location methods are only suitable for gate-level combinational circuits.

Chapters 2 and 3 identify several classes of gate- and high-level design errors such as those shown in Table 3.1 and analyze their detection requirements. By combining this analysis with the identification of the sensitized paths in the circuit, using methods such as critical path tracing [5], it is possible to determine the location of the error and correct it. Several questions need to be answered such as: How useful are verification tests in diagnosis? How do we generate efficient diagnostic tests especially for sequential circuits? What are necessary and sufficient conditions for a test set to locate errors of a given type?

Automated Test Generation for Design Error Models: In Chapter 3, we manually generated tests for several basic error models in high-level combinational benchmarks.

Developing a high-level test generation algorithm, similar to D-Algorithm or PODEM, to detect the basic and conditional error models is of great importance. Such an algorithm allows us to automatically generate test sequences for high-level design errors.

We introduced the MCE model in Chapter 3 and validated it experimentally for a small microprocessor; we also presented a general validation algorithm using this model. The MCE error model and validation approach are, at least in principle, expandable to microprocessors with instruction pipelines, multiple instruction issue, etc. Automating our validation algorithm and extending it to more complex microprocessor types is another interesting direction for future research.

High-level symbolic simulation is a basic part of test generation and test evaluation. In Chapter 3, we have manually simulated all the instruction set of the LC-2 microprocessor. Efficient tools to automate and speed up this fault simulation process are needed. Such tools are useful in ISA-based simulation and test generation for MCEs, as well as the basic and conditional error models.

Built-In Validation: Chapter 4 presents a method to design scalable hardware test generators for detecting residual design errors and physical faults in scalable datapath circuits. It also describes a scalable compactor circuit for the case of an adder. Additional research is needed to develop an automated and complete scalable BIST methodology for regular circuits, especially since it is difficult to simultaneously control the hardware complexity of TG and RM while satisfying the requirement of complete fault coverage. Another important task for future research is to develop a complete built-in validation method for microprocessors, especially for those used in safety-critical embedded systems.

In summary, the lifetime validation approach we have developed in this thesis has proven to be very useful in detecting a wide range of physical faults and design errors as early as possible in the lifetime of a digital system. We believe that lifetime validation via testing and simulation will increase in importance in the future, especially for safety-critical applications.

APPENDICES

APPENDIX A
ERROR/FAULT SIMULATOR ESIM

This appendix describes the error/fault simulator ESIM that we developed and used in Chapters 2 and 4 to determine the ability of test sets to detect gate-level design errors. Moreover, several experiments are described to illustrate ESIM's capabilities. Unlike previous simulation programs [65], ESIM is designed to efficiently fault simulate several different types of error/fault models. ESIM is based on parallel-pattern single fault propagation with critical path tracing [5] for combinational circuits and standard parallel fault simulation, with 32 faults at a time, for sequential circuits [4]. The main goal of ESIM is to evaluate the coverage of specified design errors and logical faults by using various test sets that are determined by the following automatic test pattern generation tools:

- ATALANTA [75]: This is a combinational test pattern generator for SSL faults that is characterized by short test generation time as well as small test set size. It is based on the FAN algorithm for test generation.
- ATTEST [17]: This is a powerful sequential test sequence generator for SSL faults that can be used for full-scan, partial-scan, or non-scan circuits. It can generate vectors for synchronous and asynchronous circuits, and for circuits with embedded RAM, bidirectional ports, and complex bus structures. It can operate in either of the classic PODEM or D-Algorithm modes.
- RTESTS and ETESTS: These test pattern generators were developed by us to produce random and exhaustive tests, respectively.

The following error and fault models are handled by ESIM: gate substitution errors (GSEs), gate count errors (GCEs), input count errors (ICEs), wrong input errors (WIEs), SSL faults, and input pattern (IP) faults [22]. Since the sequential part of ESIM is similar

```
/* C is the circuit*/
/* T is the simulation test set */

procedure Group1-Simulation(C,T);
begin
  Form the fault/error list L;
  Form stem list S;
  repeat
    Select a packet P of 32 tests from T;
    T := T - P;
    Set all signals in C as noncritical;
    S := S - {stems with no faults in their fanout
              free regions;}
    Perform fault free simulation using P;
    Determine criticality of stems in S via simulation;
    Traverse C backwards and determine criticality of
      all signals;
    Identify detected faults/errors D;
    L := L - D;
  until T is empty or L is empty;
  Print the results;
end;
```

```
procedure Group2-Simulation(C,T);
begin
  Form gate list GL;
  repeat
    Select a gate G from GL;
    GL := GL - {G};
    CT := T;
    repeat
      Select a packet P of 32 tests from CT;
      CT := CT - P;
      Perform fault free simulation using P;
      Determine criticality of G's output;
      NC := {gates of C not in the cone of influence of G};
      if (G's output is critical) then
      repeat
        Select a gate G' from NC;
        NC := NC - {G'};
        Mark all detected MIEs and WIEs that have the
          output of G' as the wrong source;
      until NC is empty;
    until CT is empty;
    Update the results;
  until GL is empty;
  Print the final results;
end;
```

Figure A.1 Error simulation algorithms for GROUP1 and GROUP2 errors.

to any standard parallel fault simulator, we only discuss the combinational part of ESIM in the rest of this appendix.

The detection of an error/fault in a target circuit is determined by ESIM using the information about the criticality of the lines as well as the activation conditions for the faults/errors. Simulation of GROUP1 errors (GSEs, GCEs, and EIEs) is performed in a similar manner to SSL fault simulation by using parallel-pattern evaluation and critical path tracing. Error simulation for GROUP2 errors (MIEs and WIEs) cannot be done this way because the large number of possible errors prevent the use of complete error lists. In fact, we performed error simulation for GROUP2 errors using a mixture of parallel-pattern evaluation, multiple error activation, and single fault propagation. ESIM is written in C++ and its simulation algorithms for GROUP1 and GROUP2 errors are shown in Figure A.1. The simulation algorithm for IP faults is similar to that of GROUP1.

The benchmark circuits used in the experiments of this appendix and Chapter 2 are described at the end of the appendix. Table A.1 shows the number of design errors and logical faults in these circuits. We now describe several experiments that illustrate the

Table A.1 Numbers of faults and design errors in the circuits used in the experiments.

Circuit	SSL faults	IP faults	GSEs		GCEs		ICEs		WIEs
			SIGSEs	MIGSEs	EGEs	MGEs	EIEs	MIEs	
c17	22	24	11	30	2	0	12	40	92
c432	524	2508	312	600	67	9460	296	18482	52063
c499	758	1072	337	810	104	1500	368	31452	81576
c880	942	1614	586	1470	199	1040	640	120779	299868
c1355	1574	2384	881	2370	216	1500	992	208476	480408
c1908	1879	5374	1467	2205	252	12775	1059	358816	1217410
c2670	2747	4842	1994	3380	476	4485	1559	940307	2881417
c3540	3428	10258	2584	4780	634	23470	2226	1513437	4658069
c5315	5350	11728	3902	7065	986	18110	3492	3454806	10738696
c6288	7744	9600	3904	11920	944	0	4768	4999155	10055805
c7552	7550	14636	5450	10510	1408	14390	4734	7707830	22536439
7485	137	472	86	155	20	1565	97	974	3456
74181	237	454	146	265	36	855	143	3750	11621
74283	128	240	74	150	17	460	76	985	3285

Table A.2 The percentages of SSL faults and design errors detected using exhaustive test sets.

Circuit	Test set size	Detected SSL faults	Detected IP faults	Detected GSEs		Detected GCEs		Detected ICEs		Detected WIEs
				SIGSEs	MIGSEs	EGEs	MGEs	EIEs	MIEs	
c17	32	100	100	100	100	100	N/A	100	95.00	100
7485	2048	100	66.53	100	88.39	100	94.38	100	91.17	97.45
74181	16384	100	96.48	100	98.49	88.89	99.53	100	96.61	99.07
74283	512	100	96.67	100	94.67	100	100	100	90.03	96.93

capabilities of ESIM.

- **Experiment 1** (*Exhaustive simulation*): The first experiment was conducted to investigate exhaustive simulation. The tests are generated using ETESTS. This experiment gives us the percentage of redundant design errors and logical faults in the simulated circuits. The results of the experiment are shown in Table A.2, from which we see that the redundancy of some types of design errors can be as large as 11.61%, and that of IP faults can be as large as 33.47%. This experiment is performed only for those benchmarks where simulation with exhaustive tests is feasible—circuits with approximately 16 or fewer inputs.

Table A.3 The percentages of SSL faults and design errors detected in the 4-bit 74283 adder circuit using random test sets.

Test set size	SSL	SIGSE	MIGSE	EGE	MGE	EIE	MIE	WIE
1	28.52	47.68	48.08	60.94	23.14	17.26	17.66	23.04
2	44.88	61.14	65.17	76.71	35.58	30.21	29.54	37.95
3	54.12	68.32	72.45	86.35	45.72	39.26	36.33	47.18
4	63.77	75.62	80.56	93.41	54.13	48.47	45.28	57.07
5	67.69	78.11	81.87	92.71	56.16	53.11	49.42	60.49
6	72.13	81.03	86.51	96.71	61.34	57.74	54.47	65.65
7	73.62	81.78	87.52	96.47	62.52	59.89	57.09	67.86
8	78.12	86.86	89.25	99.06	70.23	66.89	60.18	72.75
9	76.94	84.54	88.91	97.88	69.66	66.63	60.23	72.18
10	80.87	87.57	90.72	99.76	72.14	70.37	65.82	75.79
11	81.10	87.41	91.39	99.76	73.99	73.21	65.47	76.48
12	83.19	89.62	91.33	99.76	76.03	74.89	68.17	78.56
13	84.04	90.65	92.53	100.00	78.68	75.16	70.35	79.53
14	82.63	90.16	92.13	100.00	79.78	76.58	68.43	78.88
15	85.08	92.00	92.48	100.00	80.70	78.05	71.31	80.99
16	85.54	92.38	93.01	100.00	82.18	78.05	71.78	81.52
17	85.21	91.62	92.67	100.00	80.88	79.26	71.71	81.27
18	87.88	94.38	93.20	100.00	84.68	81.16	74.78	83.89
19	87.35	93.14	93.01	100.00	83.90	81.74	74.81	83.30
20	87.85	93.35	93.28	100.00	84.62	82.16	74.79	83.89

- **Experiment 2** (*Random simulation*): The second experiment evaluates the random simulation approach. Random test sets of sizes 1 through 20 were generated by RTESTS for the c74283 carry-lookahead adder circuit and the coverage of design errors was determined using ESIM. The process was repeated 50 times and the average coverage obtained is shown in Table A.3. The table shows that a small number of vectors provide good (but not full) coverage of design errors. The main problem with random simulation of this type is that it cannot guarantee high coverage with a relatively small number of vectors.
- **Experiment 3** (*Simulation using SSL tests*): A third experiment was conducted to determine the coverage of design errors and logical faults using tests for SSL faults. The effectiveness of a complete test set for SSL faults (determined by ATALANTA) in detecting design errors is shown in Tables A.4 and A.5. As discussed earlier, most of the simulation time is spent in the simulation of GROUP2, especially as the circuits become larger. The effectiveness of the complete test sets for SSL faults in detecting IP faults is shown in Table A.6. The results show that

Table A.4 The percentages of SSL faults and design errors detected using complete SSL tests generated by ATALANTA.

Circuit	Test set size	Detected SSL faults	Detected GSEs		Detected GCEs		Detected ICEs		Detected WIEs
			SIGSEs	MIGSEs	EGEs	MGEs	EIEs	MIEs	
c17	5	100	100	80.0	100	N/A	100	57.5	88.04
c432	46	99.24	100	89.33	100	95.51	98.65	71.33	96.36
c499	52	98.94	100	97.78	46.15	89.60	97.83	88.77	98.55
c880	47	100	100	90.34	100	94.62	100	84.92	98.55
c1355	85	99.49	100	82.03	100	89.60	99.19	82.18	98.55
c1908	115	99.52	100	84.72	97.62	88.69	99.15	85.80	96.95
c2670	106	95.74	99.70	86.51	87.61	88.92	93.20	85.94	97.44
c3540	152	96.00	99.34	89.52	90.54	81.21	94.20	82.73	97.52
c5315	106	98.90	99.97	89.46	98.88	91.67	98.34	94.45	98.93
c6288	35	99.56	99.59	85.57	100	N/A	99.29	89.28	99.63
c7552	199	98.25	99.98	86.62	97.37	90.29	97.21	93.15	98.68
7485	25	100	100	88.39	100	89.78	100	83.37	92.68
74181	18	100	100	96.23	88.89	90.64	100	81.76	94.02
74283	12	100	100	91.33	100	84.13	100	74.54	92.21

Table A.5 The CPU times in seconds spent on a SUN SPARC 20 by ESIM using complete SSL tests generated by ATALANTA.

Circuit	Initialization		Simulation for GROUP1		Simulation for GROUP2		Total time
	Time	%	Time	%	Time	%	
c17	0.04	50	0.02	25	0.02	25	0.08
c432	0.26	1.11	15.81	67.62	7.31	31.27	23.38
c499	0.34	2.28	3.70	24.92	10.81	72.80	14.85
c880	0.55	1.55	1.88	5.28	33.14	93.17	35.57
c1355	0.86	0.71	7.33	6.06	112.84	93.23	121.03
c1908	1.36	0.49	25.41	9.15	250.91	90.36	277.68
c2670	1.99	0.39	18.13	3.53	493.21	96.08	513.33
c3540	2.96	0.22	218.99	16.07	1140.77	83.71	1362.71
c5315	6.38	0.28	89.50	3.90	2199.19	95.82	2295.07
c6288	5.25	0.28	73.76	3.87	1824.62	95.85	1903.63
c7552	8.35	0.12	179.70	2.47	7079.72	97.41	7267.77
7485	0.08	3.56	1.85	82.22	0.32	14.22	2.25
74181	0.12	7.69	0.51	32.69	0.93	59.62	1.56
74283	0.07	11.67	0.29	48.33	0.24	40	0.60

complete test sets for SSL faults do a very poor job in detecting IP faults.

Table A.6 The percentage of IP faults detected using complete SSL tests generated by ATALANTA.

Circuit	Test set size	Detected IP faults	Simulation time on a SUN SPARC 20 (sec)
c17	5	75.00	0.01
c432	46	25.68	1.00
c499	52	80.78	1.12
c880	47	85.32	3.84
c1355	85	75.25	6.19
c1908	115	61.85	17.30
c2670	106	76.46	21.35
c3540	152	50.25	79.05
c5315	106	74.25	183.11
c6288	35	81.73	219.12
c7552	199	78.46	391.84
7485	25	40.68	0.09
74181	18	70.70	0.12
74283	12	56.67	0.04

Although ESIM was designed to handle the simulation of design errors and logical faults, it has capabilities that can be used in other applications.

- *Test grading*: The error simulator can determine the number of faults detected by a given test. This information is useful in applications such as hardware test generation and test set compaction.
- *Fault grading*: This refers to classifying the faults as hard-to-detect (also called random pattern resistant) or easy-to-detect. Fault grading has many applications such as test generation and test point selection.
- *Fault table generation*: The simulator can generate a complete fault table for circuits with 16 or fewer inputs; for larger circuits, partial fault tables can also be generated. This feature of the simulator is used to analyze the faults of a given circuit. Note that ESIM performs simple SSL fault collapsing to reduce the size of the fault table.
- *Test generation*: The simulator also supports a fast, greedy test generation algorithm based on covering the fault table. Experimental results show that the algorithm often produces near-minimal test sets in circuits with small number of

inputs.

- *Dependency evaluation*: This refers to the structural dependency between any two circuit outputs. The idea is to determine the common lines in the cones of influence of two outputs. This is useful in concurrent monitoring, where circuit outputs are compacted to decrease the hardware overhead of the hardware test/signature generator.
- *Netlist translation*: Most CAD tools accept various netlist formats, such as ISCAS 85 and BLIF. ESIM can translate any ISCAS 85 description to ISCAS 89, BLIF, and Verilog. A major use of the translator is in the synthesis of logic circuits using SIS [105], whose input format is BLIF.

ESIM also report some statistics about the circuit being simulated. This can be seen by the sample run shown in Figure A.2, where ESIM determines the coverage of an exhaustive test set for the 74283, a 4-bit carry-lookahead adder circuit.

Circuit description: Table A.7 describes the input-output characteristics of the circuits used in the experiments discussed in this thesis. The circuits c17, c432, c499, c880, c1355, c1908, c2670, c3540, c5315, c6288, and c7552 form the ISCAS 85 benchmark set [25]. The

```
esim c74283.isc c74283.xhv

ESIM Copyright 1995
Programmer: Hussain Al-Asaad

Gates = 104
================================
  Gtype Ngates  Mxfin    Mxfout
================================
  nand    4       2        7
  and    14       5        1
  nor     8       5        5
  or      0       0        0
  xor     4       2        0
  xnor    0       0        0
  inpt    9       0        2
  from   59       1        1
  not     6       1        5
  buff    0       0        0
================================
Levels  = 6
Inputs  = 9
Outputs = 5
```

```
Stems   = 22
Tests   = 512

  SSL     128     128    100.00   0.09
======= ======= ======= ======= ======
 SIGSE     74      74    100.00
 MIGSE    150     142     94.67
  EGE      17      17    100.00
  MGE     460     460    100.00
  EIE      76      76    100.00
 TOTAL    777     769     98.97   0.22
======= ======= ======= ======= ======
  MIE    1143    1029     90.03
  WIE    3285    3184     96.93
 TOTAL   4428    4213     95.14   2.86
======= ======= ======= ======= ======
  IP      240     232     96.67   0.15
======= ======= ======= ======= ======

Initialization Time = 0.27
Simulation Time     = 3.32
Total Time          = 3.59
```

Figure A.2 Output generated by a sample run of ESIM.

Table A.7 Characteristics of the circuits used in the experiments.

Circuit	No. of inputs	No. of outputs	No. of levels	No. of stems[a]
c17	5	2	6	5
c432	36	7	29	96
c499	41	32	16	91
c880	60	26	34	151
c1355	41	32	39	291
c1908	33	25	60	410
c2670	233	140	52	594
c3540	50	22	70	601
c5315	178	123	67	929
c6288	32	32	217	1488
c7552	207	108	61	1408
7485	11	3	8	20
74181	14	8	11	39
74283	9	5	6	22

a. Including the primary outputs.

Table A.8 Gate type distribution in the selected circuits.

Circuit	AND	OR	XOR	NAND	NOR	XNOR	NOT	BUFF	FROM[a]
c17	0	0	0	6	0	0	0	0	6
c432	4	0	18	79	19	0	40	0	236
c499	56	2	104	0	0	0	40	0	256
c880	87	29	0	87	61	0	63	26	437
c1355	56	2	0	416	0	0	40	32	768
c1908	63	0	0	377	1	0	277	162	995
c2670	333	77	0	254	12	0	321	196	1244
c3540	498	92	0	298	68	0	490	223	1821
c5315	718	214	0	454	27	0	581	313	2830
c6288	256	0	0	0	2128	0	32	0	3840
c7552	776	244	0	1028	54	0	876	534	3833
7485	21	0	0	4	6	0	0	0	75
74181	29	1	8	3	12	0	7	5	115
74283	14	0	4	4	8	0	6	0	59

a. Represents a fanout branch.

7485, 74181, and 74283 are a 4-bit comparator, an arithmetic-logic unit, and a carry-lookahead adder respectively; all are in the 74X IC series [107]. Table A.8 shows the gate type distribution of the circuits used.

APPENDIX B
THE LC-2 MICROPROCESSOR

In this appendix, we describe the LC-2 microprocessor [99] which was used in several experiments in Chapter 3. We also present our behavioral and RTL Verilog implementations of it.

B.1 Description of LC-2

The LC-2 microprocessor is used for educational purposes at the University of Michigan. It has 8 general purpose registers, each of which is 16 bits wide. The arithmetic and logic units operate on 16 bit words. Addresses are also 16 bits wide, so the machine has 64K words, or 128 KB of memory. The instruction set contains 16 basic instructions: arithmetic and logic instructions (ADD, AND, NOT, and NOP), data movement instructions (LD, LDI, LDR, ST, STI, STR, LEA), flow control instructions (BR, JSR, JMP, JSRR, JMPR, RET), and a system control instruction (TRAP). The simplest addressing scheme, direct addressing, forms addresses by concatenating the top 7 bits of the program counter with 9 bits of page address specified in the instruction. This means that the memory space is divided into 128 pages of 512 words each. Table B.1 summarizes the instruction formats of the LC-2. Only the following instructions modify the condition codes: ADD, AND, NOT, LD, LDI, LDR, LEA. The fields MBZ and MB1 of the instructions must be set to all-0 and all-1 respectively.

Addressing modes: An addressing mode defines the way in which a data operand is accessed. The LC-2 has five addressing modes: register, immediate, direct, indirect, and base+index. With register addressing, the operand to be accessed is located in an internal register of the LC-2. If a source operand is part of the instruction, it represents an immediate

Table B.1 Summary of instruction formats and semantics of the LC-2.

	Instruction fields														
Mnemonic	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
ADD DR, SR1, SR2	0001				DR			SR1			0	MBZ		SR2		
ADD DR, SR1, imm5											1	imm5				
AND DR, SR1, SR2	0101				DR			SR1			0	MBZ		SR2		
AND DR, SR1, imm5											1	imm5				
BR label	1000				N=0	Z=0	P=0	pgoffset9								
BRN label					N=1	Z=0	P=0									
BRZ label					N=0	Z=1	P=0									
BRP label					N=0	Z=0	P=1									
BRNZ label					N=1	Z=1	P=0									
BRNP label					N=1	Z=0	P=1									
BRZP label					N=0	Z=1	P=1									
BRNZP					N=1	Z=1	P=1									
JMP label	0100				L=0	MBZ		pgoffset9								
JSR label					L=1											
JMPR BaseR, index6	1100				L=0	MBZ		BaseR			index6					
JSRR BaseR, index6					L=1											
LD DR, label	0010				DR			pgoffset9								
LDI DR, label	1010				DR			pgoffset9								
LDR DR, BaseR, index6	0110				DR			BaseR			index6					
LEA DR, label	1110				DR			pgoffset9								
NOP	0000				MBZ											
NOT DR, SR	1001				DR			SR			MB1					
RET	1101				MBZ											
ST SR, label	0011				SR			pgoffset9								
STI SR, label	1011				SR			pgoffset9								
STR SR, BaseR, index6	0111				SR			BaseR			index6					
TRAP trapvec8	1111				MBZ				trapvec8							

operand and is accessed using the immediate addressing mode. With direct addressing, the operand is at a specified address on the current page. A direct address can be placed in the pgoffset9 field (bits 8 to 0 of the instruction, i.e. IR[8:0]) of the LD and ST instructions. The 9 bits directly identify the memory address on the current page which is specified by PC[15:9]. A complete 16-bit absolute address is formed by concatenating the page number PC[15:9] and the pgoffset9, i.e. the final address is PC[15:9]@ IR[8:0]. This address is used as access 16-bit data element. The indirect addressing mode allows the contents of a mem-

ory location to contain the address of the data element. It is implemented with the LDI and STI instructions. Bits IR[8:0] and the page number PC[15:9] are concatenated to form the indirect address whose memory contents are the absolute address of the data element to be fetched (in the case of LDI) or stored (in the case of STI). The base+index addressing mode allows the programmer to specify the absolute address of an operand as an offset from some particular starting address, which is contained in a base register BaseR. The offset is specified by the index IR[5:0] in the instruction. The absolute address is formed by adding the contents of the base register to the zero-extended offset. This address is then used for the LDR or STR operation. It is convenient to use the base+index addressing mode to process sequential data structures such as strings, records, arrays, etc. Supposing that the base register points to the beginning of an array of items, it must be incremented by the size of each element to traverse through the array. The index can also specify a field in each item.

Branching modes: A branching mode defines the way in which a branch or jump instruction is executed. The LC-2 has two basic branching modes, direct and indirect. Direct branches are implemented by the BR and JSR instructions. IR[8:0] directly identifies the target address on the current page specified by PC[15:9]. An absolute 16-bit address is formed by concatenating the page number PC[15:9] and the pgoffset9, i.e. PC[15:9] @ IR[8:0]. This address is placed into the PC. Branches to locations not on the current page are implemented with the JSRR instruction. The new PC contents are formed by adding the contents of the BaseR to the 6-bit index IR[5:0]. The *L* (link) field in the JSR instruction distinguishes it from the BR instruction. If $L = 1$, the current contents of the PC are copied to R7 before the jump takes place. This action forms a link to the calling subroutine for a later return. If $L = 0$, the contents of PC are discarded.

B.2 Behavioral Verilog Description of LC-2

We give next a complete behavioral Verilog model of the LC-2 microprocessor that we constructed. This model was used for the design error data collection in Section 3.2 and for the evaluation of the proposed design error models in Section 3.4.

```
module bcpu(clock,clear,dbus,abus,write_mem_bar,read_mem_bar);

input clock,clear;
```

```
inout [15:0] dbus;
output [15:0] abus;
output write_mem_bar,read_mem_bar;
reg [15:0] pc,ir, R[7:0], rt, tf;
reg P,N,Z;
reg read_mem_bar_temp,write_mem_bar_temp;
reg [15:0] abus_temp,dbus_temp;
reg [3:0] opcode;

`define cpu_delay 5

assign #`cpu_delay abus = abus_temp;
assign #`cpu_delay dbus = dbus_temp;
assign #`cpu_delay read_mem_bar = read_mem_bar_temp;
assign #`cpu_delay write_mem_bar = write_mem_bar_temp;

always
   begin
      if (clear == 1'b0)
      begin
          read_mem_bar_temp = 1'b1;
          write_mem_bar_temp = 1'b1;
          pc = 16'b0;
          N = 1'b0;
          P = 1'b0;
          Z = 1'b0;
          rt = 16'b0000;
          tf = 16'b0000;
          dbus_temp = 16'bzzzz;
          abus_temp = 16'bzzzz;
          @(posedge clear) #1;
          @(posedge clock) #1;
      end
      else
         begin

        // Fetch instruction
        abus_temp = pc;
        read_mem_bar_temp = 1'b0;
        @(posedge clock) #1;
        ir = dbus;
        read_mem_bar_temp = 1'b1;

        // Decode
        opcode = ir[15:12];

        // Execute
        case (opcode)
         4'b0000:; // NOP
         4'b0001: // ADD
          if (ir[5] == 1'b0)
             R[ir[11:9]] = R[ir[8:6]] + R[ir[2:0]];
          else
             begin
                rt = {ir[4],ir[4],ir[4],ir[4],ir[4],ir[4],ir[4],
                      ir[4],ir[4],ir[4],ir[4],ir[4:0]}; // Sign extend
                R[ir[11:9]] = R[ir[8:6]] + rt;
                tf = R[ir[11:9]];
             end
```

```
4'b0010: // LD
 begin
   abus_temp = {pc[15:9],ir[8:0]};
   read_mem_bar_temp = 1'b0;
   @(posedge clock) #1;
   R[ir[11:9]] = dbus;
   tf = R[ir[11:9]];
   read_mem_bar_temp = 1'b1;
 end
4'b0011: // ST
 begin
   abus_temp = {pc[15:9],ir[8:0]};
   dbus_temp = R[ir[11:9]];
   write_mem_bar_temp = 1'b0;
   @(posedge clock) #1;
   write_mem_bar_temp = 1'b1;
   dbus_temp = 16'bzzzz;
 end
4'b0100: // JSR
 begin
   if (ir[11] == 1'b1)
     R[7] = pc + 1;
   pc = {pc[15:9],ir[8:0]};
 end
4'b0101: // AND
 if (ir[5] == 1'b0)
    R[ir[11:9]] = R[ir[8:6]] & R[ir[2:0]];
 else
    begin
      rt = {ir[4],ir[4],ir[4],ir[4],ir[4],ir[4],ir[4],
            ir[4],ir[4],ir[4],ir[4],ir[4:0]};  // Sign extend
      R[ir[11:9]] = R[ir[8:6]] & rt;
      tf = R[ir[11:9]];
    end
4'b0110: // LDR
 begin
   abus_temp = R[ir[8:6]] + ir[5:0];
   read_mem_bar_temp = 1'b0;
   @(posedge clock) #1;
   R[ir[11:9]] = dbus;
   tf = R[ir[11:9]];
   read_mem_bar_temp = 1'b1;
 end
4'b0111: // STR
 begin
   abus_temp = R[ir[8:6]] + ir[5:0];
   dbus_temp = R[ir[11:9]];
   write_mem_bar_temp = 1'b0;
   @(posedge clock) #1;
   write_mem_bar_temp = 1'b1;
   dbus_temp = 16'bzzzz;
 end
4'b1000: // BR
 if (((ir[11] == 1'b1) && (N == 1'b1)) ||
     ((ir[10] == 1'b1) && (Z == 1'b1)) ||
     ((ir[9] == 1'b1) && (P == 1'b1)))
     pc = {pc[15:9],ir[8:0]};
 else
     pc = pc + 1;
```

```
 4'b1001: // NOT
   begin
    R[ir[11:9]] = ~R[ir[8:6]];
    tf = R[ir[11:9]];
  end
 4'b1010: // LDI
  begin
    abus_temp = {pc[15:9],ir[8:0]};
    read_mem_bar_temp = 1'b0;
    @(posedge clock) #1;
    rt = dbus;
    abus_temp = rt;
    @(posedge clock) #1;
    R[ir[11:9]] = dbus;
    tf = R[ir[11:9]];
    read_mem_bar_temp = 1'b1;
  end
 4'b1011: // STI
  begin
    abus_temp = {pc[15:9],ir[8:0]};
    read_mem_bar_temp = 1'b0;
    @(posedge clock) #1;
    rt = dbus;
    read_mem_bar_temp = 1'b1;
    abus_temp = rt;
    dbus_temp = R[ir[11:9]];
    write_mem_bar_temp = 1'b0;
    @(posedge clock) #1;
    write_mem_bar_temp = 1'b1;
    dbus_temp = 16'bzzzz;
  end
 4'b1100: // JSRR
  begin
    if (ir[11] == 1'b1)
       R[7] = pc + 1;
    pc = R[ir[8:6]] + ir[5:0];
  end
 4'b1101: // RET
  pc = R[7];
 4'b1110: // LEA
  begin
    R[ir[11:9]] = {pc[15:9],ir[8:0]};
    tf = R[ir[11:9]];
  end
 4'b1111: // TRAP
  begin
    R[7] = pc + 1;
    abus_temp = {8'b00000000,ir[7:0]};
    read_mem_bar_temp = 1'b0;
    @(posedge clock) #1;
    pc = dbus;
    read_mem_bar_temp = 1'b0;
  end
endcase

case (opcode)
 4'b0001,4'b0010,4'b0101,4'b0110,4'b1001,
 4'b1010,4'b1110:           // ADD, LD, AND, LDR, NOT, LDI, LEA
     begin
```

```
                if (tf == 16'b0)
                begin
                    P = 1'b0;
                    Z = 1'b1;
                    N = 1'b0;
                end
                else if (tf[15] == 1'b0)
                     begin
                        P = 1'b1;
                        Z = 1'b0;
                        N = 1'b0;
                     end
                     else
                     begin
                        P = 1'b0;
                        Z = 1'b0;
                        N = 1'b1;
                     end
            end
        endcase

        case (opcode)
         4'b0000,4'b0001,4'b0010,4'b0011, 4'b0101,4'b0110,4'b0111,4'b1001,
         4'b1010,4'b1011,4'b1110:
                // NOP, ADD, LD, ST, AND, LDR, STR, NOT, LDI, STI, lEA
                pc = pc + 1; // Increment pc
        endcase
    end
    end

endmodule
```

B.3 Synthesizable Verilog Description of LC-2

We give next the complete synthesizable Verilog model of the LC-2 microprocessor. The datapath unit of the LC-2 is described as an interconnection of RTL components while the LC-2 control unit is described using a single finite-state machine. This model was used for the design error data collection in Section 3.2, the evaluation of the proposed design error models in Section 3.4, and the illustration of our validation approach for microprocessors in Section 3.5.

```
module rtcpu(clock,clear,dbus,abus,write_mem_bar,read_mem_bar);

input clock,clear;
inout [15:0] dbus;
output [15:0] abus;
output write_mem_bar,read_mem_bar;
wire [15:0] ir_out;
wire [2:0] R1,R2,W,flags_out;
wire [1:0] sel_rf_mux, sel_pc_mux, sel_ab_mux;

datapath DP(clock,clear,dbus,abus,ir_out,flags_out,R1,R2,W,RE1,RE2,WE,
```

```
            S3,S2,S1,S0,M, load_pc_bar, load_ir_bar, load_mar_bar,
            load_flags_bar, load_reg1_bar,load_reg2_bar,sel_rf_mux, sel_pc_mux,
            sel_mar_mux, sel_ab_mux,sel_alu_mux,reg2_to_dbus_bar,zero_or_sign,
            trapvec_bar);

control CO(clock,clear,write_mem_bar,read_mem_bar,R1,R2,W,RE1,RE2,WE,
            S3,S2,S1,S0,M,load_pc_bar,load_ir_bar, load_mar_bar, load_flags_bar,
            load_reg1_bar,load_reg2_bar, sel_rf_mux, sel_pc_mux,sel_mar_mux,
            sel_ab_mux, sel_alu_mux,reg2_to_dbus_bar,zero_or_sign,trapvec_bar,
            ir_out,flags_out);

endmodule

module datapath(clock,clear,dbus,abus,ir_out,flags_out,R1,R2,W,RE1,RE2,WE,
            S3,S2,S1,S0,M, load_pc_bar, load_ir_bar, load_mar_bar,
            load_flags_bar,load_reg1_bar,load_reg2_bar,sel_rf_mux,
            sel_pc_mux,sel_mar_mux, sel_ab_mux,sel_alu_mux,reg2_to_dbus_bar,
            zero_or_sign,trapvec_bar);

input clock,clear;
inout  [15:0] dbus;
output [15:0] abus;

// TO CONTROL
output [15:0] ir_out;
output [2:0] flags_out;

// REGFILE
input [2:0] R1, R2, W;
input RE1, RE2, WE;

// ALU
input S3, S2, S1, S0, M;

// REGISTERS
input load_pc_bar, load_ir_bar, load_mar_bar, load_flags_bar,load_reg1_bar,
      load_reg2_bar;

// MUXS
input [1:0] sel_rf_mux, sel_pc_mux, sel_ab_mux;
input sel_alu_mux, sel_mar_mux;

// TRISTATE
input reg2_to_dbus_bar;

// SPECIAL
input zero_or_sign;
input trapvec_bar;

wire [2:0] flags_in;
wire [15:0] pc_in,ir_in,read_port1,read_port2,ALU_B_port,alu_out,write_port,
           rf_port1,rf_port2,mar_in,pc_out,mar_out,merge_out,inc_out,extend_out,
            latch_in;
wire clock_bar;

stdinv STI(clock,clock_bar);
alu #(16) ALU0(read_port1,ALU_B_port,1'b0,M,S0,S1,S2,S3,Dummy_COUT,alu_out);
latch #(16) LA(latch_in,clock_bar,write_port);
regfile2r #(16,8,3) RF(write_port,R1,R2,RE1,RE2,W,WE,rf_port1,rf_port2);
```

```
dffh_c #(16) REG1 (clock,clear,rf_port1,load_reg1_bar,read_port1),
             REG2 (clock,clear,rf_port2,load_reg2_bar,read_port2);
dffh_c #(16) PC (clock,clear,pc_in,load_pc_bar,pc_out),
             IR (clock,clear,dbus,load_ir_bar,ir_out),
             MAR (clock,clear,mar_in,load_mar_bar,mar_out);
dffh_c #(3) FLAGS (clock,clear,flags_in,load_flags_bar,flags_out);
mux4 #(16) RFMUX(alu_out,inc_out,merge_out,dbus,sel_rf_mux[0],sel_rf_mux[1],
            latch_in),
           PCMUX(inc_out,alu_out,merge_out,dbus,sel_pc_mux[0],sel_pc_mux[1],
            pc_in);
mux3 #(16) ABMUX(pc_out,mar_out,merge_out,sel_ab_mux[0],sel_ab_mux[1],abus);
mux2 #(16) ALUMUX(read_port2,extend_out,sel_alu_mux,ALU_B_port),
           MARMUX(alu_out,dbus,sel_mar_mux,mar_in);
tribuf #(16) TRB(reg2_to_dbus_bar,read_port2,dbus);
extend EXT(ir_out,zero_or_sign,extend_out);
detect DTC(write_port,flags_in);
inc #(16) INC0(1'b1,pc_out,Dummy_TC,Dummy_TCBAR,inc_out);
merge MRG(pc_out,ir_out,trapvec_bar,merge_out);

endmodule

module extend(IN,zero_or_sign,OUT);

input [15:0] IN;
input zero_or_sign;
output [15:0] OUT;

assign OUT[0] = IN[0],
       OUT[1] = IN[1],
       OUT[2] = IN[2],
       OUT[3] = IN[3],
       OUT[4] = IN[4];
stdmux2 SM2(IN[5],IN[4],zero_or_sign,OUT[5]);
stdand2 SA2(IN[4],zero_or_sign,TMP);
assign OUT[6]  = TMP,
       OUT[7]  = TMP,
       OUT[8]  = TMP,
       OUT[9]  = TMP,
       OUT[10] = TMP,
       OUT[11] = TMP,
       OUT[12] = TMP,
       OUT[13] = TMP,
       OUT[14] = TMP,
       OUT[15] = TMP;

endmodule

module detect(IN,FLAGS);

input [15:0] IN;
output [2:0] FLAGS;

stdinv SI(IN[15],IN15bar);
zero #(16) ZR(IN,FLAGS[1]);
stdnor2 SN1(IN[15],FLAGS[1],FLAGS[0]);
stdnor2 SN2(IN15bar,FLAGS[1],FLAGS[2]);

endmodule
```

```
module merge(PC,IR,trapvec_bar,OUT);

input [15:0] PC,IR;
input trapvec_bar;
output [15:0] OUT;

assign OUT[0] = IR[0],
       OUT[1] = IR[1],
       OUT[2] = IR[2],
       OUT[3] = IR[3],
       OUT[4] = IR[4],
       OUT[5] = IR[5],
       OUT[6] = IR[6],
       OUT[7] = IR[7];
stdand2 SA8(IR[8],trapvec_bar,OUT[8]),
        SA9(PC[9],trapvec_bar,OUT[9]),
        SA10(PC[10],trapvec_bar,OUT[10]),
        SA11(PC[11],trapvec_bar,OUT[11]),
        SA12(PC[12],trapvec_bar,OUT[12]),
        SA13(PC[13],trapvec_bar,OUT[13]),
        SA14(PC[14],trapvec_bar,OUT[14]),
        SA15(PC[15],trapvec_bar,OUT[15]);

endmodule

module control(clock,clear,write_mem_bar, read_mem_bar,R1,R2,W,RE1,RE2,WE,
        S3,S2,S1,S0,M,load_pc_bar, load_ir_bar, load_mar_bar, load_flags_bar,
        load_reg1_bar,load_reg2_bar,sel_rf_mux,sel_pc_mux,sel_mar_mux,
        sel_ab_mux, sel_alu_mux,reg2_to_bus_bar,zero_or_sign,trapvec_bar,ir_out,
        flags_out);

input clock,clear;

// TO MEMORY
output write_mem_bar, read_mem_bar;

// TO REGFILE
output [2:0] R1, R2, W;
output RE1, RE2, WE;

// TO ALU
output S3, S2, S1, S0, M;

// TO REGISTERS
output load_pc_bar, load_ir_bar, load_mar_bar, load_flags_bar,load_reg1_bar,
       load_reg2_bar;

// TO MUXS
output [1:0] sel_rf_mux, sel_pc_mux, sel_ab_mux;
output sel_alu_mux, sel_mar_mux;

// TO TRISTATE
output reg2_to_bus_bar;

// TO SPECIAL
output zero_or_sign;
output trapvec_bar;

// FROM_temp DATAPATH
```

```
input [15:0] ir_out;
input [2:0] flags_out;

reg [2:0] machine_state;
reg [2:0] next_state;
reg write_mem_bar_temp, read_mem_bar_temp;
reg [2:0] R1_temp, R2_temp, W_temp;
reg RE1_temp, RE2_temp, WE_temp;
reg S3_temp, S2_temp, S1_temp, S0_temp, M_temp;
reg load_pc_bar_temp, load_ir_bar_temp, load_mar_bar_temp,
     load_flags_bar_temp, load_reg1_bar_temp, load_reg2_bar_temp;
reg [1:0] sel_rf_mux_temp, sel_pc_mux_temp, sel_ab_mux_temp;
reg sel_alu_mux_temp, sel_mar_mux_temp;
reg reg2_to_bus_bar_temp;
reg zero_or_sign_temp;
reg trapvec_bar_temp;

// Machine States

`define MRESET_STATE 3'b000
`define IFETCH_STATE 3'b001
`define DECODE_STATE 3'b010
`define EX_MEM_STATE 3'b011
`define MEMORY_STATE 3'b100

// INSTRUCTIONS

`define ADD  4'b0001
`define AND  4'b0101
`define BR   4'b1000
`define JSR  4'b0100
`define JSRR 4'b1100
`define LD   4'b0010
`define LDI  4'b1010
`define LDR  4'b0110
`define LEA  4'b1110
`define NOP  4'b0000
`define NOT  4'b1001
`define RET  4'b1101
`define ST   4'b0011
`define STI  4'b1011
`define STR  4'b0111
`define TRAP 4'b1111

// DELAY
`define FSM_DELAY 6

// STATE MACHINE
always @(posedge clock)
begin
    machine_state = next_state;
end

always @(ir_out[15:5] or ir_out[2:0] or clear or machine_state or flags_out)
begin
   // Generate addresses for register file
   if (ir_out[15:12] == 4'b1101)
       R1_temp = 3'b111;
   else
```

```
    R1_temp = ir_out[8:6];
if (ir_out[13] == 1'b0)
    R2_temp = ir_out[2:0];
else
    R2_temp = ir_out[11:9];
if ((ir_out[14:12] == 3'b100) || (ir_out[15:12] == 4'b1111))
    W_temp = 3'b111;
else
    W_temp = ir_out[11:9];

// Compute next state
if (clear == 1'b0)
      next_state = `MRESET_STATE;
else
   begin
     case (machine_state)
        `MRESET_STATE:next_state = `IFETCH_STATE;
        `IFETCH_STATE: next_state = `DECODE_STATE;
        `DECODE_STATE:
            if (ir_out[15:12] == `NOP)
               next_state = `IFETCH_STATE;
            else
               next_state = `EX_MEM_STATE;
        `EX_MEM_STATE:
           begin
              case (ir_out[15:12])
                `AND,`ADD,`NOT,`LEA,`RET,`BR,`JSR,
                `JSRR, `LD,`ST,`TRAP: next_state = `IFETCH_STATE;
                `LDI,`STI,`LDR,`STR:next_state = `MEMORY_STATE;
              endcase
           end
        `MEMORY_STATE:next_state = `IFETCH_STATE;
     endcase
   end

// Determine control signals for each state
if (clear == 1'b0)
   begin
      read_mem_bar_temp = 1'b1;
      write_mem_bar_temp = 1'b1;
      load_pc_bar_temp = 1'b1;
      load_ir_bar_temp = 1'b1;
      load_mar_bar_temp = 1'b1;
      load_flags_bar_temp = 1'b1;
      reg2_to_bus_bar_temp = 1'b1;
   end
else
   begin
     case (machine_state)
        `MRESET_STATE:
           begin
              read_mem_bar_temp = 1'b1;
              write_mem_bar_temp = 1'b1;
              load_pc_bar_temp = 1'b1;
              load_ir_bar_temp = 1'b1;
              load_mar_bar_temp = 1'b1;
              load_flags_bar_temp = 1'b1;
              reg2_to_bus_bar_temp = 1'b1;
           end
```

```
`IFETCH_STATE:
   begin
      read_mem_bar_temp = 1'b0;
      write_mem_bar_temp = 1'b1;
      RE1_temp = 1'b0;
      RE2_temp = 1'b0;
      WE_temp = 1'b0;
      load_ir_bar_temp = 1'b0;
      load_pc_bar_temp = 1'b1;
      load_flags_bar_temp = 1'b1;
      load_reg1_bar_temp = 1'b1;
      load_reg2_bar_temp = 1'b1;
      reg2_to_bus_bar_temp = 1'b1;
      sel_ab_mux_temp = 2'b00;
   end
`DECODE_STATE:
   begin
      read_mem_bar_temp = 1'b1;
      write_mem_bar_temp = 1'b1;
      RE1_temp = 1'b1;
      RE2_temp = 1'b1;
      WE_temp = 1'b0;
      load_ir_bar_temp = 1'b1;
      load_flags_bar_temp = 1'b1;
      load_reg1_bar_temp = 1'b0;
      load_reg2_bar_temp = 1'b0;
      reg2_to_bus_bar_temp = 1'b1;
      if (ir_out[15:12] == `NOP)
         begin
           load_pc_bar_temp = 1'b0;
           sel_pc_mux_temp = 2'b00;
         end
      else
         begin
           load_pc_bar_temp = 1'b1;
         end
   end
`EX_MEM_STATE:
   begin
      RE1_temp = 1'b0;
      RE2_temp = 1'b0;
      load_ir_bar_temp = 1'b1;
      load_reg1_bar_temp = 1'b1;
      load_reg2_bar_temp = 1'b1;
      case (ir_out[15:12])
        `AND:
           begin
               WE_temp = 1'b1;
               load_pc_bar_temp = 1'b0;
               sel_pc_mux_temp = 2'b00;
               zero_or_sign_temp = 1'b1;
               if (ir_out[5] == 1'b0)
                  sel_alu_mux_temp = 1'b0;
               else
                  sel_alu_mux_temp = 1'b1;
               S3_temp = 1'b1;
               S2_temp = 1'b1;
               S1_temp = 1'b1;
               S0_temp = 1'b0;
```

```
        M_temp  = 1'b0;
        sel_rf_mux_temp = 2'b00;
        load_flags_bar_temp = 1'b0;
        reg2_to_bus_bar_temp = 1'b1;
        read_mem_bar_temp = 1'b1;
        write_mem_bar_temp = 1'b1;
    end
`ADD:
    begin
        WE_temp = 1'b1;
        load_pc_bar_temp = 1'b0;
        sel_pc_mux_temp = 2'b00;
        zero_or_sign_temp = 1'b1;
        if (ir_out[5] == 1'b0)
           sel_alu_mux_temp = 1'b0;
        else
           sel_alu_mux_temp = 1'b1;
        S3_temp = 1'b1;
        S2_temp = 1'b0;
        S1_temp = 1'b0;
        S0_temp = 1'b1;
        M_temp  = 1'b1;
        sel_rf_mux_temp = 2'b00;
        load_flags_bar_temp = 1'b0;
        reg2_to_bus_bar_temp = 1'b1;
        read_mem_bar_temp = 1'b1;
        write_mem_bar_temp = 1'b1;
     end
`NOT:
    begin
        WE_temp = 1'b1;
        load_pc_bar_temp = 1'b0;
        sel_pc_mux_temp = 2'b00;
        S3_temp = 1'b0;
        S2_temp = 1'b0;
        S1_temp = 1'b0;
        S0_temp = 1'b0;
        M_temp  = 1'b0;
        sel_rf_mux_temp = 2'b00;
        load_flags_bar_temp = 1'b0;
        reg2_to_bus_bar_temp = 1'b1;
        read_mem_bar_temp = 1'b1;
        write_mem_bar_temp = 1'b1;
    end
`LEA:
    begin
        WE_temp = 1'b1;
        load_pc_bar_temp = 1'b0;
        sel_pc_mux_temp = 2'b00;
        sel_rf_mux_temp = 2'b10;
        load_flags_bar_temp = 1'b0;
        trapvec_bar_temp = 1'b1;
        read_mem_bar_temp = 1'b1;
        write_mem_bar_temp = 1'b1;
        reg2_to_bus_bar_temp = 1'b1;
    end
`RET:
    begin
        WE_temp = 1'b0;
```

```
            load_pc_bar_temp = 1'b0;
            sel_pc_mux_temp = 2'b01;
            S3_temp = 1'b1;
            S2_temp = 1'b1;
            S1_temp = 1'b1;
            S0_temp = 1'b1;
            M_temp  = 1'b0;
            load_flags_bar_temp = 1'b1;
            reg2_to_bus_bar_temp = 1'b1;
            read_mem_bar_temp = 1'b1;
            write_mem_bar_temp = 1'b1;
        end
    `BR:
        begin
            WE_temp = 1'b0;
            load_pc_bar_temp = 1'b0;
             if ((flags_out[2] & ir_out[11]) |
                 (flags_out[1] & ir_out[10]) |
                 (flags_out[0] & ir_out[9]))
                     sel_pc_mux_temp = 2'b10;
            else
                     sel_pc_mux_temp = 2'b00;
            load_flags_bar_temp = 1'b1;
            trapvec_bar_temp = 1'b1;
            reg2_to_bus_bar_temp = 1'b1;
            read_mem_bar_temp = 1'b1;
            write_mem_bar_temp = 1'b1;
        end
    `JSR:
        begin
            if (ir_out[11] == 1'b1)
               begin
                   sel_rf_mux_temp = 2'b01;
                   WE_temp = 1'b1;
               end
            else
               WE_temp = 1'b0;
            load_pc_bar_temp = 1'b0;
            sel_pc_mux_temp = 2'b10;
            load_flags_bar_temp = 1'b1;
            trapvec_bar_temp = 1'b1;
            reg2_to_bus_bar_temp = 1'b1;
            read_mem_bar_temp = 1'b1;
            write_mem_bar_temp = 1'b1;
        end
    `JSRR:
        begin
            if (ir_out[11] == 1'b1)
               begin
                   sel_rf_mux_temp = 2'b01;
                   WE_temp = 1'b1;
               end
            else
               WE_temp = 1'b0;
            load_pc_bar_temp = 1'b0;
            sel_pc_mux_temp = 2'b01;
            sel_alu_mux_temp = 1'b1;
            zero_or_sign_temp = 1'b0;
            S3_temp = 1'b1;
```

```
            S2_temp = 1'b0;
            S1_temp = 1'b0;
            S0_temp = 1'b1;
            M_temp  = 1'b1;
            load_flags_bar_temp = 1'b1;
            reg2_to_bus_bar_temp = 1'b1;
            read_mem_bar_temp = 1'b1;
            write_mem_bar_temp = 1'b1;
        end
    `LD:
        begin
            WE_temp = 1'b1;
            read_mem_bar_temp = 1'b0;
            load_pc_bar_temp = 1'b0;
            sel_pc_mux_temp = 2'b00;
            sel_rf_mux_temp = 2'b11;
            sel_ab_mux_temp = 2'b10;
            load_flags_bar_temp = 1'b0;
            trapvec_bar_temp = 1'b1;
            reg2_to_bus_bar_temp = 1'b1;
            write_mem_bar_temp = 1'b1;
        end
    `ST:
        begin
            WE_temp = 1'b0;
            write_mem_bar_temp = 1'b0;
            load_pc_bar_temp = 1'b0;
            sel_pc_mux_temp = 2'b00;
            sel_ab_mux_temp = 2'b10;
            load_flags_bar_temp = 1'b1;
            reg2_to_bus_bar_temp = 1'b0;
             read_mem_bar_temp = 1'b1;
            trapvec_bar_temp = 1'b1;
        end
    `TRAP:
        begin
            sel_rf_mux_temp = 2'b01;
            WE_temp = 1'b1;
            read_mem_bar_temp = 1'b0;
            load_pc_bar_temp = 1'b0;
            sel_pc_mux_temp = 2'b11;
            sel_ab_mux_temp = 2'b10;
            load_flags_bar_temp = 1'b1;
            trapvec_bar_temp = 1'b0;
            reg2_to_bus_bar_temp = 1'b1;
            write_mem_bar_temp = 1'b1;
        end
    `LDI,`STI:
        begin
            WE_temp = 1'b0;
            read_mem_bar_temp = 1'b0;
            load_pc_bar_temp = 1'b0;
            sel_pc_mux_temp = 2'b00;
            sel_ab_mux_temp = 2'b10;
            load_flags_bar_temp = 1'b1;
            load_mar_bar_temp = 1'b0;
            sel_mar_mux_temp = 1'b1;
            trapvec_bar_temp = 1'b1;
            reg2_to_bus_bar_temp = 1'b1;
```

```
                        write_mem_bar_temp = 1'b1;
                    end
                `LDR,`STR:
                    begin
                        WE_temp = 1'b0;
                        load_pc_bar_temp = 1'b0;
                        sel_pc_mux_temp = 2'b00;
                        sel_alu_mux_temp = 1'b1;
                        zero_or_sign_temp = 1'b0;
                        S3_temp = 1'b1;
                        S2_temp = 1'b0;
                        S1_temp = 1'b0;
                        S0_temp = 1'b1;
                        M_temp  = 1'b1;
                        load_flags_bar_temp = 1'b1;
                        load_mar_bar_temp = 1'b0;
                        sel_mar_mux_temp = 1'b0;
                        reg2_to_bus_bar_temp = 1'b1;
                        read_mem_bar_temp = 1'b1;
                        write_mem_bar_temp = 1'b1;
                    end
            endcase
        end
    `MEMORY_STATE:
        begin
            RE1_temp = 1'b0;
            RE2_temp = 1'b0;
            load_ir_bar_temp = 1'b1;
            load_reg1_bar_temp = 1'b1;
            load_reg2_bar_temp = 1'b1;
            sel_ab_mux_temp = 2'b01;
            load_pc_bar_temp = 1'b1;
            trapvec_bar_temp = 1'b1;
            sel_rf_mux_temp = 2'b11;
            case (ir_out[15:12])
              `LDI,`LDR:
                    begin
                        WE_temp = 1'b1;
                        read_mem_bar_temp = 1'b0;
                        write_mem_bar_temp = 1'b1;
                        load_flags_bar_temp = 1'b0;
                        reg2_to_bus_bar_temp = 1'b1;
                    end
              `STI,`STR:
                    begin
                        WE_temp = 1'b0;
                        write_mem_bar_temp = 1'b0;
                        read_mem_bar_temp = 1'b1;
                        load_flags_bar_temp = 1'b1;
                        reg2_to_bus_bar_temp = 1'b0;
                    end
            endcase
        end
    endcase
  end
end

// TO MEMORY
assign #`FSM_DELAY write_mem_bar = write_mem_bar_temp;
```

```
assign #`FSM_DELAY read_mem_bar = read_mem_bar_temp;

// TO REGFILE
assign #`FSM_DELAY R1 = R1_temp;
assign #`FSM_DELAY R2 = R2_temp;
assign #`FSM_DELAY W = W_temp;
assign #`FSM_DELAY RE1 = RE1_temp;
assign #`FSM_DELAY RE2 = RE2_temp;
assign #`FSM_DELAY WE = WE_temp;

// TO ALU
assign #`FSM_DELAY S3 = S3_temp;
assign #`FSM_DELAY S2 = S2_temp;
assign #`FSM_DELAY S1 = S1_temp;
assign #`FSM_DELAY S0 = S0_temp;
assign #`FSM_DELAY M = M_temp;

// TO REGISTERS
assign #`FSM_DELAY load_pc_bar = load_pc_bar_temp;
assign #`FSM_DELAY load_ir_bar = load_ir_bar_temp;
assign #`FSM_DELAY load_mar_bar = load_mar_bar_temp;
assign #`FSM_DELAY load_flags_bar = load_flags_bar_temp;
assign #`FSM_DELAY load_reg1_bar = load_reg1_bar_temp;
assign #`FSM_DELAY load_reg2_bar = load_reg2_bar_temp;

// TO MUXS
assign #`FSM_DELAY sel_rf_mux = sel_rf_mux_temp;
assign #`FSM_DELAY sel_pc_mux = sel_pc_mux_temp;
assign #`FSM_DELAY sel_mar_mux = sel_mar_mux_temp;
assign #`FSM_DELAY sel_ab_mux = sel_ab_mux_temp;
assign #`FSM_DELAY sel_alu_mux = sel_alu_mux_temp;

// TO TRISTATE
assign #`FSM_DELAY reg2_to_bus_bar = reg2_to_bus_bar_temp;

// TO SPECIAL
assign #`FSM_DELAY zero_or_sign = zero_or_sign_temp;
assign #`FSM_DELAY trapvec_bar = trapvec_bar_temp;

endmodule
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] E. J. Aas, T. Steen, and K. Klingsheim, "Quantifying design quality through design experiments", *IEEE Design and Test*, Vol. 11, pp. 27-37, Spring 1994.

[2] M. S. Abadir, J. Ferguson, and T. E. Kirkland, "Logic design verification via test generation", *IEEE Transactions on Computer-Aided Design*, Vol. 7, pp. 138-148, January 1988.

[3] M. S. Abadir and H. K. Reghbati, "Functional testing of semiconductor random access memories", *Computing Surveys*, Vol. 15, No. 3, September 1983.

[4] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*, Computer Science Press, New York, 1990.

[5] M. Abramovici, P. R. Menon, and D. T. Miller, "Critical path tracing: An alternative to fault simulation", *IEEE Design and Test*, Vol. 1, pp. 83-93, February 1984.

[6] V. D. Agarwal and E. Cerny, "Store and generate built-in testing approach", *Proc. Fault-Tolerant Computing Symposium*, 1981, pp. 35–40.

[7] A. Aharon et al., "Verification of the IBM RISC System/6000 by dynamic biased pseudo-random test program generator", *IBM Systems Journal*, Vol. 30, No. 4, pp. 527–538, 1991.

[8] S. B. Akers, "Universal test sets for logic networks", *IEEE Transactions on Computers*, Vol. C-22, pp. 835-839, September 1973.

[9] S. B. Akers and W. Jansz, "Test set embedding in a built-in self-test environment", *Proc. International Test Conference*, 1989, pp. 257-263.

[10] H. Al-Asaad and J. P. Hayes, "Design verification via simulation and automatic test pattern generation", *Proc. International Conference on Computer-Aided Design*, 1995, pp. 174-180.

[11] H. Al-Asaad, J. P. Hayes, and B. T. Murray, "Scalable test generators for high-speed datapath circuits", *Journal of Electronic Testing: Theory and Applications*, Vol. 12, Nos. 1/2, pp. 111-125, February/April 1998. Reprinted in M. Nicolaidis, Y. Zorian, and D. K. Pradhan (editors), *On Line-Testing for VLSI*, Kluwer, Boston, 1998.

[12] H. Al-Asaad, J. P. Hayes, and B. T. Murray, “Design of scalable hardware test generators for on-line BIST”, *Digest of Papers: International On-Line Testing Workshop*, 1996, pp. 164-167.

[13] H. Al-Asaad, B. T. Murray, and J. P. Hayes, “On-line BIST for embedded systems”, *IEEE Design and Test*, 1998, to appear.

[14] H. Al-Asaad, D. Van Campenhout, J. P. Hayes, T. Mudge, and R. Brown, “High-level design verification of microprocessors via error modeling”, *Digest of Papers: International High-Level Design Validation and Test Workshop*, 1997, pp. 194-201.

[15] G. Al Hayek and C. Robach, “From specification validation to hardware testing: A unified method”, *Proc. International Test Conference*, 1996, pp. 885–893.

[16] M. Annaratone, *Digital CMOS Circuit design*, Kluwer, Boston, 1986.

[17] ATTEST Software, *ATTEST Software Tools*, Santa Clara, Calif., 1995.

[18] R. Bailey, *Human Error in Computer Systems*, Prentice-Hall, Englewood Cliffs, N. J., 1981.

[19] P. H. Bardell, W. H. McAnney, and J. Savir, *Built-In Self-Test for VLSI: Pseudorandom Techniques*, Wiley, New York, 1987.

[20] B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York, 2nd edition, 1990.

[21] D. Bhattacharya and J. P. Hayes, “High-level test generation using bus faults,” *Proc. Fault-Tolerant Computing Symposium*, 1985, pp. 65–70.

[22] R. D. Blanton, *Design and Testing of Regular Circuits*, Ph.D. dissertation, University of Michigan, 1995.

[23] S. Boubezari and B. Kaminska, “A deterministic built-in self-test generator based on cellular automata structures”, *IEEE Transactions on Computers*, Vol. 44, pp. 805-816, June 1995.

[24] D. Brand, “Exhaustive simulation need not require an exponential number of tests”, *IEEE Transactions on Computer-Aided Design*, Vol. 12, pp. 1635-1641, November 1993.

[25] F. Brglez and H. Fujiwara, “A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran”, *Proc. International Symposium on Circuits and Systems*, 1985, pp. 695-698.

[26] F. Brglez, D. Bryan, and K. Kozminski, “Combinational profiles of sequential benchmark circuits”, *Proc. International Symposium on Circuits and Systems*, 1989, pp. 1929-1934.

[27] R. Brown et al., “Complementary GaAs technology for a GHz microprocessor”, *Technical Digest of the GaAs IC Symposium*, 1996, pp. 313-316.

[28] R. Bryant, “Graph-based algorithms for boolean function manipulation”, *IEEE Transactions on Computers*, Vol C-35, pp. 677-691, August 1986.

[29] R. Bryant, “Division, Pentium style: An analysis of Intel's mistake(s)”, Carnegie Mellon University, Computer Systems Seminar, February 1995. Available from http://www.cs.cmu.edu/afs/cs.cmu.edu/project/verify/www/pentium-bug.html.

[30] Cadence Design Systems, *Verilog-XL Reference Manual*, Vol. 1, Version 1.6, Lowell, Mass., 1991.

[31] J. D. Calhoun and F. Brglez, “A framework and method for hierarchical test generation”, *IEEE Transactions on Computer-Aided Design*, Vol. 11, pp. 45-67, January 1992.

[32] F. Casaubieilh et al., “Functional verification methodology of Chameleon processor”, *Proc. Design Automation Conference*, 1996, pp. 421–426.

[33] P. Cederqvist et al., *Version Management with CVS*, Signum Support AB, Linkoping, Sweden, 1992.

[34] K. Chakrabarty, B. T. Murray, J. Liu, and M. Zhu, “Test width compression for built-in self testing”, *Proc. International Test Conference*, 1997, pp. 328-337.

[35] A. K. Chandra et al., “AVPGEN - a test generator for architecture verification”, *IEEE Transactions on VLSI Systems*, Vol. 3, pp. 188–200, June 1995.

[36] B. Chen, C. L. Lee, and J. E. Chen, “Design verification by using universal test sets”, *Proc. Asian Test Symposium*, 1994, pp. 261-266.

[37] C.-A. Chen and S. K. Gupta, “A methodology to design efficient BIST test pattern generators”, *Proc. International Test Conference*, 1995, pp. 814-823.

[38] T. C. K. Chou, “Beyond fault tolerance”, *IEEE Computer*, Vol. 30, pp. 47-49, April 1997.

[39] P. Chung and I. Hajj, “ACCORD: Automatic catching and correction of logic design errors in combinational circuits”, *Proc. International Test Conference*, 1992, pp. 742-751.

[40] P. Chung, Y. Wang, and I. Hajj, “Logic design error diagnosis and correction”, *IEEE Transactions on VLSI Systems*, Vol. 2, pp. 320-332, September 1994.

[41] R. P. Colwell and R. A. Lethin, “Latent design faults in the development of the Multiflow TRACE/200”, *IEEE Transactions on Reliability*, Vol. 43, No. 4, pp. 557–565, December 1994.

[42] W. Daehn and J. Mucha, “Hardware test pattern generation for built-in testing”, *IEEE Test Conference*, 1981, pp. 110-113.

[43] R. Dandapani, J. H. Patel, and J. A. Abraham, “Design of test pattern generator for built-in self-test”, *Proc. International Test Conference*, 1984, pp. 315-319.

[44] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on test data selection: Help for the practicing programmer”, *IEEE Computer*, pp. 34–41, April 1978.

[45] R. A. DeMillo et al., *Software Testing and Evaluation*, Benjamin/Cummings, Menlo Park, Calif., 1987.

[46] S. Devadas, A. Ghosh, and K. Keutzer, “Observability-based code coverage metric for functional simulation”, *Proc. International Conference on Computer-Aided Design*, 1996, pp. 418–425.

[47] C. Dufaza and G. Cambon, “LFSR based deterministic and pseudo-random test pattern generator structures”, *Proc. European Test Conference*, 1991, pp. 27-34.

[48] M. Fujita, T. Kakuda, and Y. Matsunaga, “Redesign and automatic error correction of combinational circuits”, *Logic and Architecture Synthesis*, Elsevier, New York, pp. 253-262, 1991.

[49] G. Ganapathy et al., “Hardware emulation for functional verification of K5”, *Proc. Design Automation Conference*, 1996, pp. 315–318.

[50] D. Gizopoulos, A. Paschalis, and Y. Zorian, “An effective BIST scheme for Booth multipliers”, *Proc. International Test Conference*, 1995, pp. 824-833.

[51] M. C. Hansen, *Symbolic Functional Test Generation with Guaranteed Low-Level Fault Detection*, Ph.D. dissertation, University of Michigan, 1996.

[52] M. C. Hansen and J. P. Hayes, “High-level test generation using physically-induced faults”, *Proc. VLSI Test Symosium*, 1995, pp. 20-28.

[53] M. C. Hansen and J. P. Hayes, “High-level test generation using symbolic scheduling”, *Proc. International Test Conference*, 1995 , pp. 586-595.

[54] J. P. Hayes, “On the properties of irredundant logic networks”, *IEEE Transactions on Computers*, Vol. C-25, pp. 884-892, Sept. 1976.

[55] S. Hellebrand et al., “Pattern generation for deterministic BIST scheme”, *Proc. International Conference on Computer-Aided Design*, 1995, pp. 88-94.

[56] S. Hellebrand, S. Tarnick, and J. Rajski, “Generation of vector patterns through reseeding of multiple-polynomial linear feedback shift registers”, *Proc. International Test Conference*, 1992, pp. 120–128.

[57] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, San Francisco, 1990.

[58] A. Hosseini, D. Mavroidis, and P. Konas, “Code generation and analysis for the functional verification of microprocessors”, *Proc. Design Automation Conference*, 1996, pp. 305–310.

[59] S.-Y. Huang et al., “ErrorTracer: A fault simulation-based approach to design error diagnosis”, *Proc. International Test Conference*, 1997, pp. 974-981.

[60] Intel Corp., “Pentium processor specification update,” 1998, available from http:// www.intel.com.

[61] Intel Corp., “Statistical analysis of floating point flaw in the Pentium processor”, Technical report, November 1994.

[62] J. Jain et al., “Probabilistic design verification”, *Proc. International Conference on Computer-Aided Design*, 1991, pp. 468-471.

[63] P. Jain and G. Gopalakrishnan, “Efficient symbolic simulation-based verification using the parametric form of boolean expressions”, *IEEE Transactions on Computer-Aided Design*, Vol. 13, pp. 1005-1015, August 1994.

[64] B. W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, Reading, Mass., 1989.

[65] S. Kang and S. A. Szygenda, “The simulation automation system (SAS); concepts, implementation, and results”, *IEEE Transactions on VLSI Systems*, Vol. 2, pp. 89-99, March 1994.

[66] M. Kantrowitz and L. M. Noack, “I’m done simulating; Now what? Verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor”, *Proc. Design Automation Conference*, 1996, pp. 325–330.

[67] H. Kim, “C880 high-level Verilog description”, Internal report, University of Michigan, 1996.

[68] K. N. King and A. J. Offutt, “A Fortran language system for mutation-based software testing”, *Software: Practice and Experience*, Vol. 21, pp. 685-718, July 1991.

[69] I. Koren, *Computer Arithmetic Algorithms*, Prentice-Hall, Englewood Cliffs, N. J., 1993.

[70] J. Kumar, “Prototyping the M68060 for concurrent verification”, *IEEE Design and Test*, Vol. 14, pp. 34–41, January 1997.

[71] W. Kunz and D. K. Pradhan, “Recursive learning: A precise implication technique for efficient solutions to CAD problems — Test, verification, and optimization”, *IEEE Transactions on Computer-Aided Design*, Vol. 13, pp. 1143-1157, September 1994.

[72] W. Kunz, D. K. Pradhan, and S. M. Reddy, "A novel framework for logic verification in a synthesis environment", *IEEE Transactions on Computer-Aided Design*, Vol. 15, pp. 20-32, January 1996.

[73] S. Kuo, "Locating logic design errors via test generation and don't care propagation", *Proc. European Design Automation Conference*, 1992, pp. 466-471.

[74] H. K. Lee and D. S. Ha, "An efficient forward fault simulation algorithm based on the parallel pattern single fault propagation", *Proc. International Test Conference*, 1991, pp. 946-955.

[75] H. K. Lee and D. S. Ha, "On the generation of test patterns for combinational circuits", Dept. of Elec. Eng., Virginia Tech., Rep. 12-93, 1993.

[76] J. Lee and J. H. Patel, "A signal-driven discrete relaxation technique for architectural level test generation", *Proc. International Conference on Computer-Aided Design*, 1991, pp. 458-461.

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

[78] J. Lee and J. H. Patel, "Architectural level test generation for microprocessors", *IEEE Transactions on Computer-Aided Design*, Vol. 13, pp. 1288-1300, October 1994.

[79] H. Liaw, J. Tsaih, and C. Lin, "Efficient automatic diagnosis of digital circuits", *Proc. European Design Automation Conference*, 1990, pp. 464-467.

[80] A. P. Lowell, "The care and feeding of watchdogs", *Embedded Systems Programming*, pp. 38-52, April 1992.

[81] F. Maamari and J. Rajski, "A method of fault simulation based on stem regions", *IEEE Transactions on Computer-Aided Design*, Vol. 9, pp. 212-220, February 1990.

[82] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processors—A survey", *IEEE Transactions on Computers*, Vol. C-37, pp. 160-174, February 1988.

[83] E. J. McCluskey, *Logic Design Principles*, Prentice-Hall, Englewood Cliffs, N. J., 1986.

[84] MIPS Technologies, *MIPS R4000PC/SC Errata, Processor Revision 2.2 and 3.0*, May 1994.

[85] J. Miyake et al., "Automatic test generation for functional verification of microprocessors", *Proc. Asian Test Symposium*, 1994, pp. 292-297.

[86] F. Muradali, V. K. Agarwal, and B. Nadeau-Dostie, "A new procedure for weighted random built-in self-test", *Proc. International Test Conference*, 1990, pp. 660–669.

[87] B. T. Murray and J. P. Hayes, “Hierarchical test generation using precomputed tests for modules”, *IEEE Transactions on Computer-Aided Design*, Vol. 9, pp. 594-603, 1990.

[88] B. T. Murray and J. P. Hayes, “Test propagation through modules and circuits”, *Proc. International Test Conference*, 1991, pp. 748-757.

[89] B. T. Murray and J. P. Hayes, “Testing ICs: Getting to the core of the problem”, *IEEE Computer*, Vol. 29, pp. 32-45, November 1996.

[90] B. Nadeau-Dostie, A. Silburt, and V. K. Agarwal, “Serial interfacing for embedded memory testing”, *IEEE Design and Test*, Vol. 7, no. 2, pp. 52–63, April 1990.

[91] P. Narain et al., “A high-level approach to test generation”, *IEEE Transactions on Circuits and Systems. Part I, Fundamental Theory and Applications*, Vol. 40, pp. 483-492, July 1993.

[92] M. Nicolaidis, “Test pattern generators for arithmetic units and arithmetic and logic units”, *Proc. European Test Conference*, 1991, pp. 61-71.

[93] M. Nicolaidis, “Theory of transparent BIST for RAMs”, *IEEE Transactions on Computers*, Vol. 45, pp. 1141-1156, October 1996.

[94] G. Odawara et al., “A logic verifier based on boolean comparison”, *Proc. Design Automation Conference*, 1986, pp. 208-214.

[95] A. J. Offutt et al., “An experimental determination of sufficient mutant operators”, *ACM Transactions on Software Engineering & Methodology*, Vol. 5, No. 2, pp. 99-118, April 1996.

[96] S. Palnitkar, P. Saggurti, and S.-H. Kuang, “Finite state machine trace analysis program”, *Proc. International Verilog HDL Conference*, 1994, pp. 52–57.

[97] J. H. Patel and L. Y. Fung, “Concurrent error detection in ALU's by recomputing with shifted operands”, *IEEE Transactions on Computers*, Vol. C-31, pp. 417-422, July 1982.

[98] I. Pomeranz and S. M. Reddy, “On error correction in macro-based circuits”, *Proc. International Conference on Computer Aided Design*, 1994, pp. 568-675.

[99] M. Postiff, *LC-2 Programmer’s Reference Manual*, Revision 3.1, University of Michigan, 1996.

[100] D. Pradhan (ed.), *Fault-Tolerant Computing: Theory and Techniques*, Vol. 2, Prentice-Hall, Englewood Cliffs, N. J., 1986.

[101] J. Rajski and J. Tyszer, “Recursive pseudoexhaustive test pattern generation”, *IEEE Transactions on Computers*, Vol. 42, pp. 1517-1521, December 1993.

[102] K. K. Saluja, R. Sharma, and C. R. Kime, "A concurrent testing technique for digital circuits", *IEEE Transactions on Computer-Aided Design*, Vol. 7, pp. 1250-1259, December 1988.

[103] T. M. Sarfert, R. G. Markgraf, and M. H. Schulz, "A hierarchical test pattern generation system based on high-level primitives", *IEEE Transactions on Computer-Aided Design*, Vol. 11, pp. 34-44, January 1992.

[104] N. R. Saxena and J. P. Robinson, "Accumulator compression testing", *IEEE Transactions on Computers*, Vol. C-35, pp. 317-321, April 1986.

[105] E. M. Sentovich et al., "SIS: A system for sequential circuit synthesis", Dept. of Elec. Eng. and Comp. Sci., University of California, Berkeley, Memorandum No. UCB/ERL M92/41, May 1992.

[106] D. Siewiorek and R. Swarz, *Reliable Computer Systems: Design and Evaluation*, Digital Press, Burlington, Mass., 1992.

[107] Texas Instruments, *The TTL Logic Data Book*, Dallas, 1988.

[108] S. M. Thatte and J. A. Abraham, "Test generation for microprocessors", *IEEE Transactions on Computers*, Vol. C-29, pp. 429-441, June 1980.

[109] M. Tomita and H. Jiang, "An algorithm for locating logic design errors", *Proc. International Conference on Computer-Aided Design*, 1990, pp. 468-471.

[110] N. A. Touba and E. J. McCluskey, "Synthesis of mapped logic for generating pseudorandom patterns for BIST", *Proc. International Test Conference*, 1995, pp. 674-682.

[111] J. Turino, "Test economics in the 21st century", *IEEE Design and Test*, Vol. 14, pp. 41-44, July 1997.

[112] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata", *Proc. International Symposium on Software Testing, Analysis, and Verification*, 1993, pp. 139-148.

[113] D. Van Campenhout, H. Al-Asaad, J. P. Hayes, T. Mudge, and R. Brown, "High-level design verification of microprocessors via error modeling", *ACM Transactions on Design Automation of Electronic Systems*, 1998, to appear.

[114] B. Vasudevan et al., "LFSR-based deterministic hardware for at-speed BIST", *Proc. VLSI Test Symposium*, 1996, pp. 201-207.

[115] R. Wei and A. Sangiovanni-Vincentelli, "PROTEUS: A logic verification system for combinational circuits", *Proc. International Test Conference*, 1986, pp. 350-359.

[116] M. R. Woodward, "Mutation testing – its origin and evolution", *Information & Software Technology*, Vol. 35, pp. 163–169, March 1993.

[117] M. Yoeli (ed.), *Formal Verification of Hardware Design*, IEEE Computer Society Press, Los Alamitos, Calif., 1990.