FUNCTIONAL DESIGN VERIFICATION FOR MICROPROCESSORS BY ERROR MODELING

by

David Van Campenhout

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering)
in The University of Michigan
1999

To my parents.

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my advisor Trevor Mudge for his mentoring and guidance. Trevor has also given me broad freedom in my research and has been much more than just a research advisor. My appreciation and thanks also goes to John Hayes. Working with John has been a great learning experience. I would like to thank Karem Sakallah and Richard Brown for serving on my committee.

I would like to thank those who volunteered their time and effort to participate in our error collection effort: Hussain Al-Asaad, Todd Basso, Mary Brown, Juan Antonio Carballo, Subhachandra Chandra, Robert Chappell, Jim Dundas, David Greene, Jonathan Hauke, Rohini Krishna Kaza, Michael Kelley, Matt Postiff, and Steve Raasch.

I thank my friends and colleagues in ACAL: Hussain Al-Alsaad, Jeff Bell, I-Cheng Chen, Brian Davis, Jim Dundas, Jonathan Hauke, Tom Huff, Jim Huggins, Hyungwon Kim, Keith Kraver, Victor Kravets, Chih-Chieh Lee, Charles Lefurgy, João Paulo Marques da Silva, Phiroze Parakh, Steeve Raasch, Mike Riepe, Mike Upton, and Hakan Yalcin. These individuals together with the faculty provided an intellectually stimulating environment where it was also fun to work. I will never forget Tim Stanley who left us much too early. I am grateful for several close friends I met during my years in Michigan and hope that they will continue to be part of my life.

My deepest appreciation goes to my parents for their unconditional love and encouragement. I am grateful to my brothers for their understanding.

I would like to acknowledge the Belgian American Educational Foundation for having given me the opportunity to study in the US and supporting my first year of study.

TABLE OF CONTENTS

LIST OF FIGURES

<u>**Figure**</u>

LIST OF TABLES

Table:

LIST OF APPENDICES

CHAPTER 1
Introduction

Information technology is drastically changing our world. Economies are shifting from the industrial age of steel and cars to the information age of computer networks and ideas [Eco96]. In 1998, the gross domestic product (GDP) in the US due to computers, semiconductors, and electronics reached that of the automobile industry: 3.5% [Baum98]. The microprocessor, which saw its birth in 1971 with the Intel 4004, plays a central role in this information revolution. Indeed, Intel now dominates the hardware side of the computer industry. It is the largest (by dollar volume) chipmaker in the world. Microprocessors have become commodity products and are essential parts, not just of computers, but also of cars, cellular phones, personal digital assistants, and video games, to name just a few. Continuous technological improvements have led to integrated circuits becoming smaller, faster and cheaper. Simultaneously, people have continued to find new uses of microchips and computers. The Internet, with its explosive growth, is just the latest example.

The markets for microprocessors demand low cost and high performance, and are changing rapidly. To meet these demands, microprocessor design houses have to overcome great technological challenges: Circuits need to be designed that operate at very high speed. New design methodologies to deal with signal integrity and timing issues are becoming necessary now that the minimum feature size has dropped well into the deep sub-micron regime. The number of transistors integrated on a single chip is doubling every 18 months. This has led to an enormous growth in functional complexity. Furthermore, the pressure put on the design cycle by time-to-market is enormous.

Functional verification, which is concerned with ensuring that the design implements the intended functional behavior, is considered one of microprocessor design's major bottlenecks. A verification team today is the size of what an entire microprocessor design team was 10 years ago [Wolf98b]. The technology and knowledge required to do verification ‘right’ is expected to become a differentiator among companies [Wolf98b]. Successful companies will have to put together a complete set of complementing technologies to ensure their designs’ functional quality. Also, microprocessors are increasingly being used in (safety-) critical applications, which directly influence the need for better design verification.

The cost of doing verification ‘wrong’ can be very high. Manufacturing an extra silicon revision of a chip is becoming increasingly expensive [Wils99]. Far more severe can be the loss in expected sales and margin due to the delay in planned shipping date when an unplanned silicon revision is required. Finally, production volumes are ramping up so rapidly, that letting the first customers do the debugging is no longer an option. To cover the replacement of faulty chips containing the infamous FDIV bug, Intel announced a charge against earnings of $475 million [Beiz95].

The subject of this thesis is a new methodology for functional design verification of microprocessors. To put functional design verification into perspective, we first examine the design process. We then define functional design verification in Section 1.2. The next three sections discuss some important issues in functional design verification: test generation (Section 1.3), correctness checking (Section 1.4), and quality measurement (Section 1.5). We examine related work in the areas of physical fault testing and software testing in Sections 1.6 and 1.7, respectively. We conclude this introduction with an outline of the thesis.

1.1 Microprocessor design

To put functional design verification into perspective, we first examine the design process of microprocessors. Our discussion is based on an article by Bose and Conte [Bose98]. Referring to Table 1.1, four major phases can be identified: 1) architectural exploration, 2) microarchitecture definition, 3) design implementation, and 4) post tape-out.

Table 1.1: Phases in the design of a microprocessor

Phase	Synthesis activities	Analysis (verification) activities	Design representations
Architectural exploration	Application selection Workload selection Trace generation	Validation of traces	Performance, cost, and power goals
Microarchitecture definition	Development of trace-driven simulator Microarchitecture definition	Performance verification	ISA Trace-driven simulation model (μarch.)
Design implementation	Logic design Circuit design Physical design	Func. design verif. Func. impl. verif. Timing verification Electrical verification Layout verification	Behavioral RTL Structural RTL Transistor-level schematics Layout
Post tape-out	Machine-specific compiler tuning	Functional testing Electrical charact. Performance charact.	First silicon

Architectural exploration

Both economical and technological forces set the design targets for a new microprocessor. The specifications include performance goals for certain applications, cost and power consumption constraints.

One of the first tasks of the architects is to select representative applications, generate corresponding workloads, and reduce these to *benchmarks* that are small enough for microarchitectural simulation. Other activities include the analysis of the dataflow of key applications and the development of a crude model to evaluate the performance of the architecture. Architects bound the design space based on technological considerations.

Microarchitecture definition

The second phase is concerned with the definition of the microarchitecture. Architects work on microarchitectural innovations, determine which microarchitectural features need to be considered, and build simulation models for them. These models are the building blocks of a simulation model for the complete microarchitecture. The purpose of the microarchitectural simulator, also referred to as a *timer* or a *performance simulator*, is to compute an accurate estimate of the execution time, in number of clock cycles, of a given benchmark on a given concrete microarchitecture [Burg97, John91]. As the number of design points is very large and the size of the benchmarks is very large as well, simulation speed is important. A popular simulation technique is trace-driven simulation [John91]. A concrete microarchitecture is defined by a set of parameters that further specify the features used, such as the number and latency of integer execution units, or the size of a cache. Also associated with each microarchitectural feature is its cost in terms of hardware, but also in terms of design and verification effort. To accurately estimate the cost some floorplanning and circuit design studies may be required.

The design problem is that of finding the microarchitecture that meets all the constraints and provides the best trade-off between performance and cost. Verification at this stage of the design is mainly concerned with the correctness of the microarchitectural simulator; the systematic study of this problem, also referred to as performance verification, has only recently gained interest [Bose98].

Design implementation

In the third phase the microarchitecture is implemented. The microarchitectural specification is typically not formal, and consists of textual descriptions, block diagrams, and parameter values. The first step in this phase is to design the first register-transfer level (RTL) description. Standardized hardware description languages (HDL's), such as Verilog [IEEE96] and VHDL [IEEE88], or C/C++ are commonly used to describe the RTL model. This activity is sometimes referred to as *control design*. Logic designers refine this behavioral RTL description to a structural RTL description. Circuit designers generate transistor-level netlists that implement the structural RTL. Layout designers generate layouts for the transistor-level schematics. The refinement from behavioral RTL to layout differs significantly among industrial design methodologies.

There are numerous verification problems at this stage. They include functional (logic) verification, timing verification, electrical verification, physical design rule verification. In functional verification we distinguish between design verification, which is concerned with the functional correctness of the initial RTL description, and implementation verification, which is concerned with checking the functional equivalence between two versions of the implementation. The latter includes comparing the RTL view against the structural view, the structural RTL view vs. transistor level schematics, and the schematics vs. the layout.

Post tape-out

Functional design verification continues after the design has been taped out. Once first silicon is available, extensive functional testing can begin. The main difference with pre-silicon design verification is the vast increase in test throughput. Diagnosing the root cause of a discrepancy can be difficult. Functional test suites are complemented with test suites aimed at measuring performance. Other activities in this phase include electrical characterization, and physical fault testing and diagnosis.

Software activities that critically depend on the hardware, such as machine-specific compiler tuning, and post-hardware measurements, can start as soon as the silicon has been found sufficiently functional. This may involve engineering workarounds for remaining functional bugs, or fabrication of corrected versions of the chip.

1.2 Functional verification

Functional verification has gained a lot of interest in recent years as evidenced by the surge in publications detailing industrial experience with the topic:

- AMD’s K5: [Gana96]
- DEC’s Alpha: [Kant96, Tayl98]
- HP’s PA RISC: [Alex96, Bass95, Mang97, Weir97]
- IBM’s S/390: [Shep97, Wile97]
- Metaflow’s Sparc: [Pope96]
- Motorola/IBM’s PowerPC: [Mall95, Mona96]
- SGS Thomson’s Chameleon: [Casa96]

Although concrete methodologies differ from company to company, some common themes can be identified as we shall see.

Functional implementation verification. Functional *implementation* verification refers to checking the functional equivalence between two versions of the design. The two versions may be representations of the design at a different level of abstraction (below the microarchitectural level), such as behavioral and structural RTL. Alternatively they may be different versions of the design at the same level of abstraction; for example, one may be a retimed version of the other.

Efficient methods have been developed to formally check the boolean equivalence of large combinational circuits [Kuel97], and have recently become available commercially [Goer95]. For library-based logic design methodologies these tools are readily applicable. In custom methodologies a significant effort may be involved in automatically extracting an accurate gate-level view from the transistor level netlist. Nevertheless, these methods are becoming a favorable alternative for regression verification using a (switch-level) simulator.

Combinational equivalence checkers can be used to check the equivalence of two sequential circuits if there is a one-to-one mapping between the state registers. If such a mapping does not exist, the complexity of the problem greatly increases. For the special case of circuits whose state registers differ because of retiming, more specialized methods have been developed [Bisc97, Hosk95].

Functional design verification. Functional *design* verification (FDV) is concerned with verifying the functional correctness of the first RTL model of the design. For microprocessor design correctness, this means conformance to the instruction set architecture (ISA) and to some (incomplete) microarchitectural specification.

Functional design verification undergoes several phases as the project progresses. The complete effort can be divided into a pre-silicon and a post-silicon phase. The former phase is further divided into unit verification and system verification.

During *unit* verification a portion of the design is verified in isolation. For larger units, another (lower) level of integration (designer macros) may be appropriate. Basically the same verification techniques as those at the system level are applicable. The advantages of this bottom-up approach are as follows. Functional bugs that are confined to a single unit are usually easier to detect and diagnose when that unit is exercised in isolation than in the system. Furthermore, verification does not have to wait until every unit is completed. A disadvantage is the overhead required to set-up testbenches for all of the units. This may require abstract models to mimic other units interfacing with the one being verified. Once functional quality criteria for the units have been met, *system* verification can be started.

The aim of *pre-silicon* functional verification is not to eliminate every functional design error, but to raise the functional quality of the design to a level that facilitates swift hardware bring-up and test. Consequently the system has to be able to perform most functions perfectly, but a small number of remaining errors can be tolerated. Wile et al. [Wile97] report three key elements in achieving this goal: 1) The strengths and weaknesses of verification methods used need to be understood. 2) The priorities of hardware bring-up need to be understood and verified. 3) Work-around mechanisms, which allow for avoidance of failing aspects of the system behavior, need to be understood and fully functional. If these conditions are met, a much higher confidence in the functional quality of the design can be obtained in a given amount of time by fabricating the chip and performing extensive functional testing than by any of the pre-silicon verification techniques. This is essential for meeting time-to-market. The throughput of test cases during *post-silicon* verification is typically 3 to 5 *orders of magnitude* greater than that during pre-silicon verification. On the other hand, diagnosing an error may take significantly more time. The first step is usually to try to reconstruct the conditions that led to the discrepancy in the RTL simulation model. The observability offered by the RTL model can then be used to diagnose the error to its root cause.

Approaches to functional design verification

There are two broad approaches to functional hardware design verification: formal and simulation-based. Formal methods try to verify the correctness of a system by using mathematical proofs [Clar96, Kurs97, McFa93, McMi94, Yoel90]. Such methods implicitly consider all possible behaviors of the models representing the system and its specification, whereas simulation-based methods can only consider a limited range of behaviors. The accuracy and completeness of the system and specification models is a fundamental limitation for any formal method. The spectrum of formal methods for FDV is broad. At one end there are methods that are highly automated, but address only a very restricted problem space [Beer96, EET94, Goer97, Hard96, Kuel97, McMi93]. Methods at the opposite end, such as theorem proving, use formalisms to address a richer class of problems, and have mechanisms to use hierarchy and abstraction, but require a great amount of expertise to apply them [Cohn87, Cohn89, Owre96, Wind95].

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. All of the microprocessor manufacturers mentioned in the beginning of Section 1.2 report that they still rely heavily on simulation-based methods to verify their products. Simulation-based methods are readily applicable as typical design flows use HDL descriptions that can be simulated using standard logic simulation tools, or C/C++ descriptions in conjunction with a proprietary (cycle-based) simulator. Also, logic simulation is an area hardware designers are very familiar with.

In the following sections we examine some important issues in simulation-based functional design verification: test generation, checking correctness, and measuring functional quality.

1.3 Test generation for FDV

Manual test generation. Hand-written tests have been used with great success in the early days of computer design. With an intimate understanding of the design, a designer can write a very powerful and yet small set of verification tests that are very effective in exercising the design. Hand-written tests also have the advantage of being easier to debug. However, as the size and complexity of designs have dramatically increased, it becomes harder for one person to comprehend the complete design at a detailed level. Moreover, modern simulation technology, such as cycle-based simulation [McGe95], is able to simulate millions of clock cycles per day on a single workstation, even for the biggest designs. To take advantage of this simulation capability, automated test generation methods are needed.

Pseudo-random test generation. In the area of physical fault testing, it has long ago been recognized that test patterns can easily be generated randomly. The method requires very little effort, but its efficiency and effectiveness, are rather low compared to algorithmic approaches. Furthermore, the effectiveness and efficiency decrease with increasing design size [Abra90]. Nevertheless, random test pattern generation can be very useful to complement manual test generation in the absence of better methods. Sophisticated pseudo-random exercisers have been used very successfully to validate complex microprocessor designs [Ahar91, Kant96]. Taylor et al. [Tayl98] report that 79% of the functional bugs in the DEC Alpha 21264 microprocessor were found by pseudo-random tests. To achieve this high effectiveness such pseudo-random test generators incorporate knowledge about the instruction set architecture and the concrete micro-architecture. They typically have many parameters that allow the verification engineer to *bias* test generation towards "interesting behaviors," such as corner cases. A strength of random test generators is that they can generate test cases that verification engineers might have never thought of. On the other hand, most random test generators have so-called *holes*: these are areas in the space of valid test sequences that are covered only with an extremely low probability, or even are not covered at all. Random tests are more difficult to debug than hand-written tests.

Template-based test generation. Certain aspects of designs are difficult to cover with biased random tests. This may be the case if the space of valid input sequences is highly constrained. Specialized tools have been developed to help automate the generation of such focused tests. One example is a code generator described in [Chan94, Chan95]. The user provides so-called symbolic instruction graphs that compactly describe a set of instruction sequences that exhibit certain properties. The tool generates actual instruction sequences that satisfy all the properties by using constraint solving techniques. One property might be that the third instruction is a load-class instruction, which causes a cache miss, and that the fourth instruction is an arithmetic instruction using the result produced by the load instruction. Free variables, such as the register that serves as the load target, are chosen in a biased random manner. The tool incorporates knowledge about the micro-architecture, in the form of implicit constraints and biasing functions. A similar tool is described in [Hoss96].

Operating system (OS) and application code. Other sources of verification tests are operating system code and application code. However, booting an OS requires on the order of ten billion cycles [Kuma97], therefore the use of this type of verification has only recently become feasible through hardware emulation [Gana96, Bass95, Kuma97]. Demonstrating that the design correctly boots several OS's is a great confidence builder. Furthermore, for architectures that are not very well documented, such as the Intel x86 architecture [Wolf98a], successfully running application software with the OS in place is a common practice [Gana96]. In spite of the fact that the x86 architecture dominates the industry, there are some subtle features [X86] which are not officially documented and can cause compatibility problems. Ultimately application and OS software are the yardsticks for compatibility. In-circuit emulation is one step closer towards real system operation. The emulator is hooked up to a (modified) system board and hence receives real external events from other devices on the system bus.

Coverage-directed test generation. Pseudo-random test generators are typically deployed in conjunction with extensive coverage measurements. Coverage is a measure of the completeness of a test suite for a design. A discussion of prevalent coverage metrics is given in Section 1.5. Coverage data is analyzed to identify regions of the behavior that are not (well) covered. Usually, verification engineers manually tune the pseudo-random test generators, or write a focussed test to cover the verification hole.

Error-oriented test generation. A different approach is to use synthetic design error models to guide test generation. This exploits the similarity between hardware design verification and physical fault testing. For example, Al-Asaad and Hayes [AA95] define a class of design error models for gate-level combinational circuits. They describe how each of these errors can be mapped onto single-stuck line (SSL) faults that can be targeted with standard automated test pattern generation (ATPG) tools. This provides a method to generate tests with a provably high coverage for certain classes of modeled errors.

A second method in this class stems from mutation testing, which is an error-oriented structural approach to software testing. Mutation testing will be discussed in greater detail in Section 1.7. Recently, Al Hayek and Robach [AH96] have successfully applied mutation testing to hardware design verification in the case of small VHDL modules.

1.4 Checking the outcome of a simulation

A nontrivial task in simulation-based FDV is to determine the outcome of simulating a verification test, i.e., did the verification test detect an error?

Manual inspection. Manual inspection of the simulation output is still a commonly used method, especially in the early stages of the verification effort. The engineer, typically the designer, inspects the simulation output through an interface similar to that of a logic analyzer. This method is very flexible way for tracking down an error to its source. The interface allows the designer to explore the entire design. Every signal in the circuit can be examined. Once the outcome of a simulation run has been validated, it can be stored together with the test for later use (regression testing). Although manual inspection is error-prone and impractical for large test sets, it is still necessary to diagnose the root cause of discrepancies detected by the methods discussed below.

Self-checking tests. A first approach to automated correctness checking is to make the tests self-checking. The tests start by setting up the system’s initial state. This is followed by the bulk of the verification test. At the end, part of the system’s final state is compared to a precomputed final state included with the test. If the test was generated manually, it is not uncommon that the test writer computes the expected final state himself. For larger tests, and for tests that were generated with tool assistance or even completely automatically, the final state is usually computed by running the test through a suitable high-level simulation model of the system, such as an interpreter for the ISA. The outcome of a self-checking test is basically pass or fail. In case of failure, the test needs to be simulated again, this time with full visibility. Tracking down the error (error diagnosis) can be very tedious and time consuming. Another complication of the approach is the problem of *error masking*. At some point in the simulation, a verification test may uncover a design error (make it observable in the visible part of the machine state), but the error effect might get annihilated in the remaining part of the simulation. To overcome this difficulty, tests are usually restricted in length. An advantage of self-checking tests is that only the implementation needs to be simulated. The absence of a suitable simulatable reference model (specification) makes it the only choice for correctness checking. The use of self-checking tests has been reported in [Chan94, Ho96a]. Kantrowitz and Noack [Kant96] worked on the functional verification of a commercial superscalar microprocessor. They reported some of these difficulties associated with self-checking tests, and mainly used the *co-simulation* approach to correctness checking.

Co-simulation. The second approach for automated checking is to simulate the implementation and the specification together and to compare the states of both machine constantly. This approach has been referred to as co-simulation [Ho96a]. A discrepancy between implementation and specification states indicates either an error in the implementation (a design error), or an error in the reference model. Provided that the reference model and the implementation are developed independently, it is very unlikely that both models exhibit exactly the same error and hence that an error would pass unnoticed. The main difficulty with this approach is that the reference model and the implementation are usually at different levels of abstraction. Synchronizing the simulation of both models, and providing an appropriate mapping (both in time and in space) between the state of the reference model and the implementation may require a significant amount of effort. A superscalar processor with out-of-order execution might be working on several tens of instructions at any given time, whereas the sequential reference model processes only a single instruction at a time. The benefits of this approach are as follows. Co-simulation allows large verification tests to be run without supervision. Changes in the implementation only require changing the state mapping. No individual verification tests need modification, as might be the case for self-checking tests. Co-simulation is typically used in verification methodologies that use pseudo-random test generation [Kant96, Tayl98].

Assertion checkers. Assertion checkers, also referred to as *watchdogs* or *snoopers*, are agents that check certain properties about the design during the simulation. Checkers increase observability, help diagnosis, and prevent wasted simulation cycles by aborting the simulation as soon as a violation occurs. They can be used in conjunction with self-checking tests or co-simulation. A module that monitors the signals on a bus and checks these signals against the bus protocol is an example of a checker. Another example is a unit that monitors a finite-state machine and halts the execution as soon as the machine enters an illegal state.

1.5 Measuring and predicting functional quality

Perhaps the most challenging problem in simulation-based FDV is to estimate the confidence in the functional quality of a design after a certain amount of verification. Bass et al. report the following acceptance criteria that were used in the verification effort of the HP PA 7100LC microprocessor [Bass95]:

- all failures are diagnosed to their root cause
- no chip failures exist
- all handwritten tests pass
- random code generators have run for a “long time” without finding any failures
- application software has run without any indication of hardware bugs

The problem can be stated informally as two questions: “How thoroughly has the design been verified?” and “When am I (going to be) done simulating?” The first question is concerned with *coverage*, the completeness of a verification test set. Given that exhaustive simulation is not practical, and that therefore any simulation-based verification method is incomplete, the second question is concerned with predicting the effort required to achieve a certain level of functional quality. Determining coverage and predicting the verification effort are closely related aspects. Both analysis of bug detection data, and analysis of coverage have been used to gauge the (expected) confidence in the functional quality of a design.

Analysis of bug detection data

Upton collected design error data from the Aurora GaAs microprocessor designs at the University of Michigan [Upto94, Upto97]. He observed error rates of one design error per every 100 to 200 lines of Verilog code. This figure has been confirmed by industrial sources [Bent97]. Upton also analyzed the cumulative number of detected bugs over time and suggested that the bug detection process can be modeled as a function exponentially tapering off in time.

Malka and Ziv apply techniques from software reliability engineering for statistical analysis of bug detection data from two industrial microprocessor design projects [Malk98]. They use trend analysis to gauge the effect of the introduction of a new test generation techniques on reliability growth. Modeling of the bug discovery process is used to make short term predictions, such as the mean time to the next failure, and long term predictions, such as when a certain level of reliability can be expected. They conclude that statistical analysis of bug detection data can provide very relevant information for determining tapeout dates.

The use of quality criteria based on analysis of bug detection data is widespread. However very little data of this type has been published [Mona96]. One reason might be that this data is highly dependent on the design and verification methodology, the nature of the design, and the designers themselves. Also, bug detection data is only meaningful to the extent that a detailed verification plan has been carefully designed and implemented, and that continuous efforts have been made to improve and extend techniques to exercise the design. Biased-random test generators can generate new tests indefinitely, but these tests tend to loose their effectiveness over time. Analysis of coverage provides another means to assess the functional quality of a design.

Analysis of coverage

Coverage is a measure of the completeness of a test suite for a design. Moundanos, Abraham and Hoskote [Moun98] give an idealized definition of coverage as the ratio of the exercised behaviors over the total number of specified behaviors. A behavior can be modeled as an execution trace of the design. Unfortunately, attempting to exercise all possible execution paths is an intractable problem. Practical coverage metrics are needed to expose shortcomings of test suites and to spur further directed test generation. They are also needed to complement the methods discussed above for evaluating the state of completion of the verification effort.

Code coverage metrics from software testing

Software design also poses the problem of measuring the effectiveness of testing [Beiz90]. Classical structural metrics such as *statement*, *branch* and *path coverage* also apply to hardware design, as designs are usually represented in hardware description languages today. It is well known that many design errors may still go undetected even though complete statement and branch coverage has been achieved. Full path coverage is an impractical goal as the number of paths can be exponential. An advantage of these metrics is that their computation imposes only a small overhead on logic simulation. A typical flow for code coverage measurement is as follows. First the original HDL description is instrumented. The instrumented code is then simulated for the given test suite using an standard logic simulator that is augmented with library functions provided by the coverage tool vendor. Part of the simulation outcome is coverage data that can be examined using a coverage analysis tool.

OCCOM

Although code coverage metrics such as statement coverage can be computed efficiently, they suffer from not taking into account observability. A verification test that activates a particular statement, but fails to propagate the effect of executing that statement to a part of the machine state that is truly observable (those signals that are also part of the specification), cannot be considered to have "covered" that statement. To address this shortcoming Devadas et al. [Deva96] propose a code coverage metric based on *tag propagation*, which was later refined in [Fall98a] and is called OCCOM, which stands for observability-based code coverage metric. Errors are associated with assignment statements in the code. The effect of an error is represented by a "tag" that can propagate through the circuit according to a set of rules similar to the D-calculus [Abra90]. The metric measures the fraction of tags that have been propagated to the observable state over the number of tags injected. The major extension of [Fall98a] to the earlier work in [Deva96] is an efficient method for computing OCCOM coverage using a standard logic simulator. The efficiency of computation is closely related to the definition of the tag propagation rules. The propagation rules are defined so that, in essence, the erroneous machine stays on the same execution path as the error-free machine. Experimental results on small examples show a modest overhead factor of 1.5-4 over logic simulation, which is a much smaller overhead than that incurred by fault simulation.

FSM based metrics

Microprocessor designs typically have a natural partition: datapaths and controllers. Controllers have been found to be particularly prone to design errors [Ho96a]. An appropriate model for small controllers is that of a finite-state machine (FSM). Coverage can then be measured as the fraction of states or state transitions visited by a test sequence. FSM transition coverage is not a meaningful metric for microprocessors, which can easily contain thousands of state registers. Even if it were possible to compute the set of reachable states, any coverage measurement with respect to complete state graph would be negligibly small. However, most of the state registers are part of the datapath. Ho [Ho95] worked on the verification of the protocol processor in the FLASH project [Kusk94]. Even after abstracting the datapath, he was still faced with the state explosion problem. Control is usually distributed and consists of a number of interacting smaller FSM’s. Ho proposed an incremental strategy in which coverage with respect to the individual state machines is attempted first. This type of coverage measurement is referred to as FSM coverage in industry [Hoss96, Kant96, Nels97, Paln94, SP92], and is also supported by EDA vendors [Clar98]. Next, larger composite state machines can be considered. To reduce the size of that state graph further, Ho defined an equivalence relation on states. All states that apply the same control signals to the datapath are considered equivalent. Provided that the machine has been partitioned in such a way that the datapath does not store any control state, this is a very reasonable assumption. Ho applied his methodology in combination with hand-generated self-checking tests. He found that designers presented with state transitions that were not covered had great difficulty generating tests to exercise these transitions. An automated method for test generation would have helped a lot. This direction of research set in [Ho95] has gained considerable following: [Ho96b, Ho96a, Geis96, Gupt97, Lewi96, Moun98]. Of particular interest is a study of the relationship between reduced-FSM coverage and design error coverage by Gupta et al. [Gupt97]. However, theoretical results in this area tend to be weak and extensive experimental studies have not appeared in the literature.

Design-specific coverage analysis

The functional complexity of commercial microprocessors has been increasing continuously. Functional verification of these designs poses formidable challenges that verification teams tackle with very pragmatic approaches that are specific to the concrete design to be verified, and build on past experience. We list some of the coverage analysis techniques from the functional verification effort of the DEC Alpha processors [Dohm98, Kant96, Tayl98]:

- *State transition analysis.* State transitions coverage is measured on individual FSM's in the design. Some aspects of the design may not be directly implemented as a localized FSM, but may have a natural abstract FSM view that can then be used to measure transition coverage. The analysis may include checking for transitions that are not supposed to occur.
- *Sequence analysis.* A sequence of microarchitectural events in a particular window of time is measured. This type of analysis can be used, for example, to ensure that a behavioral model for an external device on the system bus is fully randomizing events.
- *Occurrence analysis.* Occurrence analysis refers to counting events without any time relationship. An example is checking that a carry-out has been generated for every stage of an adder.

Similar work appears in [Mona96, Pope96].

Design error coverage

A class of simulation-based verification approaches [Abad88, AA95, Kang94, VC98] use synthetic error models to guide test generation. Coverage is then defined as the ratio of detected synthetic errors to the total number synthetic errors. This is similar to a fault grade in physical fault testing [Abra90]. An advantage of design error coverage is that it addresses observability. A limitation is its high computational cost, as in fault simulation.

1.6 Related area: physical fault testing

Physical fault testing addresses the problem of detecting physical errors in digital systems. Physical faults may be introduced during the manufacturing process or may appear over time “in the field” due to wear, etc. Direct analysis of physical faults is a physical problem. Furthermore, a wide variety of technology-dependent physical faults exist. Logical fault models and delay fault models have been developed to model in a technology-independent way the physical faults that affect a system’s function and operating speed, respectively [Abra90]. In the following we restrict the discussion to testing for logical faults. Logical fault models greatly simplify the testing problem by moving the problem from the physical domain to the Boolean domain.

A logical fault model together with the circuit under test, defines a set of faulty circuits. The test generation problem is to find tests that distinguish each of the faulty circuits from the fault-free circuit. Likewise, in error-directed FDV, tests need to be generated that distinguish the given design from a number of erroneous circuits defined by a design error model. Despite this close relationship, the problems have major differences concerning 1) the reference model, 2) the nature of the circuit: combinational vs. sequential, 3) fault/error models, and 4) hierarchy.

Reference model. In physical fault testing the fault-free circuit is given and completely specified. In design verification, a design (implementation) that needs to be verified is given; the correct design is unknown. Instead, and at best, a complete model of the correct design at a higher abstraction level is given also. Often only a partial specification of the system’s behavior is available.

Combinational vs. sequential circuits. Test generation for sequential circuits is a much harder problem than test generation for combinational circuits [Micz86, Chen96, Marc96]. Although several test generators are commercially available that are able to generate very high quality tests for very large combinational circuits, test generation for sequential circuits the size of modern microprocessor is well beyond the reach of any current automatic test pattern generation (ATPG) system. However, design for testability techniques (DFT) [Abra90] can greatly reduce the complexity. In full scan design, every register is replaced by a scan register and the registers are linked in a chain, thereby making every register observable and controllable. This effectively reduces the test generation problem to one for combinational circuits. Unfortunately these DFT techniques do not apply to design verification, since typically there is no one-to-one correspondence between the state elements of the design implementation and the reference model (specification).

Fault/error models. A third difference between the two areas is that physical fault testing has a proven and widely accepted logical fault model, the single-stuck line (SSL) model. The SSL combines simplicity with the property that it forces each line in the circuit to be exercised. A large body of research has been based on this model. Design verification, as yet, does not have such a fault model. The success of the SSL model provides a motivation to develop error models for design verification, which can potentially benefit from the work on SSL faults.

Hierarchy. Physical fault testing and FDV also differ in the role hierarchy plays. In physical fault testing the goal is always complete (SSL) fault coverage at the gate-level. Hierarchical test generation approaches have been proposed that carry the promise of being able to handle larger designs than purely gate-level methods. Typically, test sets are precomputed for gate-level descriptions of individual modules. System tests are then derived at the high-level representation that apply the test stimuli to the module under test, and propagate the error effects to the system’s primary outputs [Murr90, Murr92]. FDV can be done in a bottom-up fashion. First, the units constituting the design are verified. Verification tests are applied to the unit in isolation during this phase. Next, the complete design is verified, shifting the focus towards the interaction of the units.

1.7 Related area: software testing

Since the introduction of HDL’s to mainstream design methodologies in the 80’s, hardware design has started to resemble software design. HDL’s, such as Verilog and VHDL, take after general-purpose programming languages, such as C/C++ and Ada. Complex mechanisms, such as dynamic memory allocation, recursion, arbitrary user defined data type and pointers, are readily supported by general-purpose programming languages and are commonly used in software design. In hardware designs, however, these mechanisms need to be implemented explicitly by the designers so that they readily map onto hardware. Functional verification of software is therefore substantially more complex than hardware verification.

The task of software testing is to ensure the reliability of software. A large number of methodologies and techniques have been proposed; an overview can be found in [Beiz90]. However, most of these techniques are not supported by tools that automate the testing process. This is in contrast to most areas of hardware testing and verification. One explanation is that software tends to be more complex and more diverse. Testing methods that are both practical and effective tend to rely on expert knowledge about the software under test that is very difficult to automate. Another explanation is that most testing methods crucially depend on a *specification*. Written specifications are now considered a cornerstone of any software development project [Post96b], but there has been a time when the software community stubbornly tried to avoid taking time to record a description of how software was supposed to behave.

Programmers obviously need specifications to write code, but these specifications should also be written down to facilitate making changes and repairs later. Testers need written specifications to determine whether an observed behavior conforms to the intended behavior.

Informal specifications, plain English descriptions of the requirements, have the benefit of being easy to read. Unfortunately they are a major obstacle to automation of software testing. Recently, formal languages that are still readable, such as Semantic Transfer Language (STL) that appears in [IEEE94], have been proposed to capture specifications. Such formal specifications can automatically be checked not only for syntax problems, but also for semantic inconsistencies. Formal specifications are essential to systematic test case generation and functional coverage measurement.

Beizer [Beiz90] distinguishes two major approaches to software testing: functional and structural. Functional methods are specification-directed and view the implementation as a black box. Structural methods are driven by the implementation. In the remainder of this section we discuss some of the few techniques that 1) are general, i.e., not specific to a particular application domain, and 2) have substantial support for automation.

Control flowgraph path testing. Path testing methods are based on the use of the control flowgraph of the program. In this graph nodes represent branching points or junction points in the programs. Arcs represent branch-free code with a single entry and exit. Path testing is the oldest of all structural test techniques. Beizer references work at IBM from 1964. Tests are targeted at bugs that make the program take a different path than intended. Completeness of test sets is measured in terms of their statement, branch, and path coverage. Statement coverage requires that all statements in the program be executed at least once. Branch coverage require that each alternative at each branch in the program is exercised at least once. Branch coverage implies statement coverage. Path coverage requires that all control flow paths through the program are exercised. This is in general not practical to achieve as the number of paths can be exponential. Full statement and branch coverage are common targets during unit testing. Additional paths are selected by other test methods, such as dataflow testing, or logic-based testing. Coverage measurement is widely supported by software development tool vendors [Paxs98, SR].

Mutation testing. Mutation testing is an error-oriented structural approach to software testing introduced by DeMillo et al. in [DeMi78]. Mutation testing considers programs, termed mutants, that differ from the given program by only simple errors, such as replacing ‘<‘ by ‘≤’ in one conditional expression. The task of the tester is to construct tests that distinguish the mutants from the given program. Mutation testing provides a metric, mutation coverage, to grade test sets. King and Offutt described in [King91] a system to automatically generate tests using constraint solving techniques.

Mutation testing is predicated on two hypotheses. The competent programmer hypothesis assumes that programmers write code that is very close to correct code. The coupling effect hypothesis states that a test that distinguishes all programs differing from a correct one by only simple errors (mutants), will also be sensitive for more complex errors.

Mutation testing has several high costs associated with it: 1) the large number of mutants that need to be considered, 2) the requirement to execute tests on all of the mutants still alive, and 3) the complexity of test generation. The first problem is addressed by selective mutation [Offu96]. Selective mutation considers only mutation operators, which generate a number of mutants which is linear in the size of the program. A careful experimental study indicates that tests having good coverage with respect to selective mutation provides a good coverage for (non-selective) mutation as well. The second source of the cost is addressed by weak mutation [Wood93]. In weak mutation only the activation of the error is considered, not the propagation. This makes weak mutation a much easier criterion to satisfy. In firm mutation [Wood93], the error is introduced in the program, during the execution, and it only persists for a limited duration of the execution, not till completion of the execution. The correlation between coverage with respect to weak or firm mutation and that with respect to (strong) mutation has not been studied.

Run-time debugging. Errors involving dynamically allocated memory are notoriously difficult to debug because they are hard to analyze statically. Examples of such errors are leaking memory, using uninitialized variables, using freed memory, and reading or writing beyond array boundaries. The widely used Purify tool [Rat97] is designed for this type of errors. Purify instruments the object code of a program so that during program execution all code is checked for these run-time errors and memory leaks.

A common type of late-cycle error in hardware designs involves complex control interactions. A register leak occurs when a shared register latches corrupted data, or when it is overwritten before its data has been used. To combat this type of error, 0-In Design Automation [0In] proposes to automatically synthesize assertions that check for these errors. The instrumented design containing the assertions is then simulated with a standard HDL simulator.

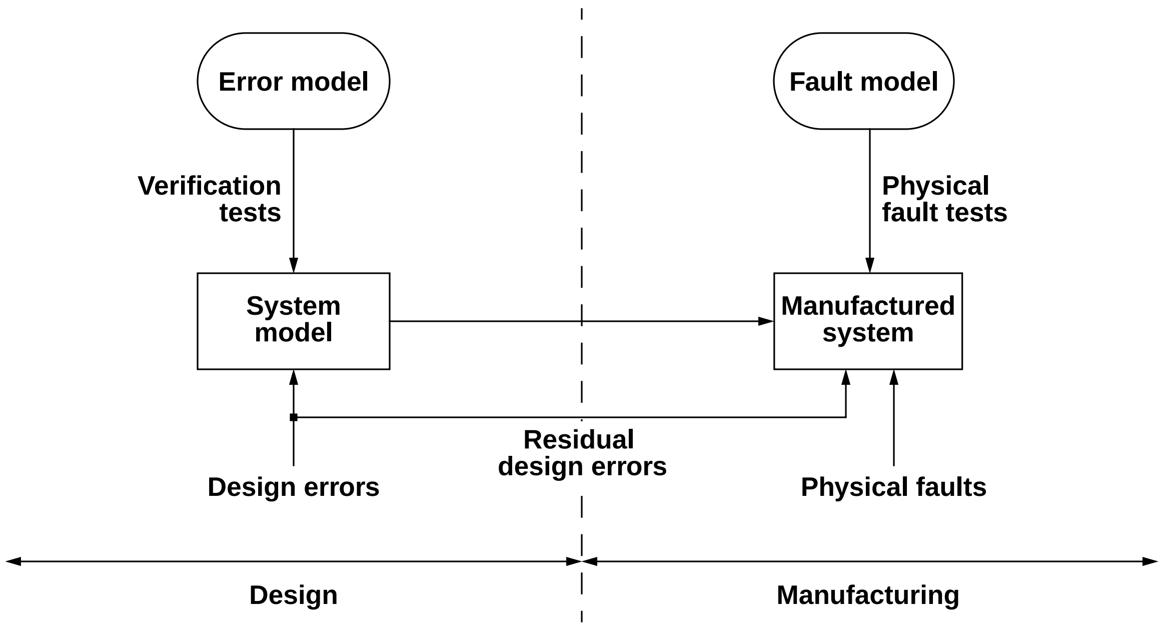


Figure 1.1: Relationship between physical design verification and physical fault testing

1.8 Thesis outline

This thesis explores functional design verification by modeling design errors and generating functional vectors for modeled errors using methods adapted from physical fault testing techniques. The close relationship between physical fault testing and design verification is illustrated in Figure 1.1. The task of physical fault testing is to identify parts that are functionally defective due to imperfections in the manufacturing process. For this purpose, tests are generated that are targeted instances of logical fault models, such as SSL faults. Likewise, the task of FDV is to detect functional design errors. We develop design error models based on empirical error data.

The deployment of our methodology is illustrated in Figure 1.2. An implementation to be verified and its specification are given. For microprocessors, the specification is typically the ISA, and the implementation is a description of the new design in an HDL, such as VHDL or Verilog. In this approach, synthetic error models are used to guide test generation. The tests are applied to simulated models of both the implementation and the

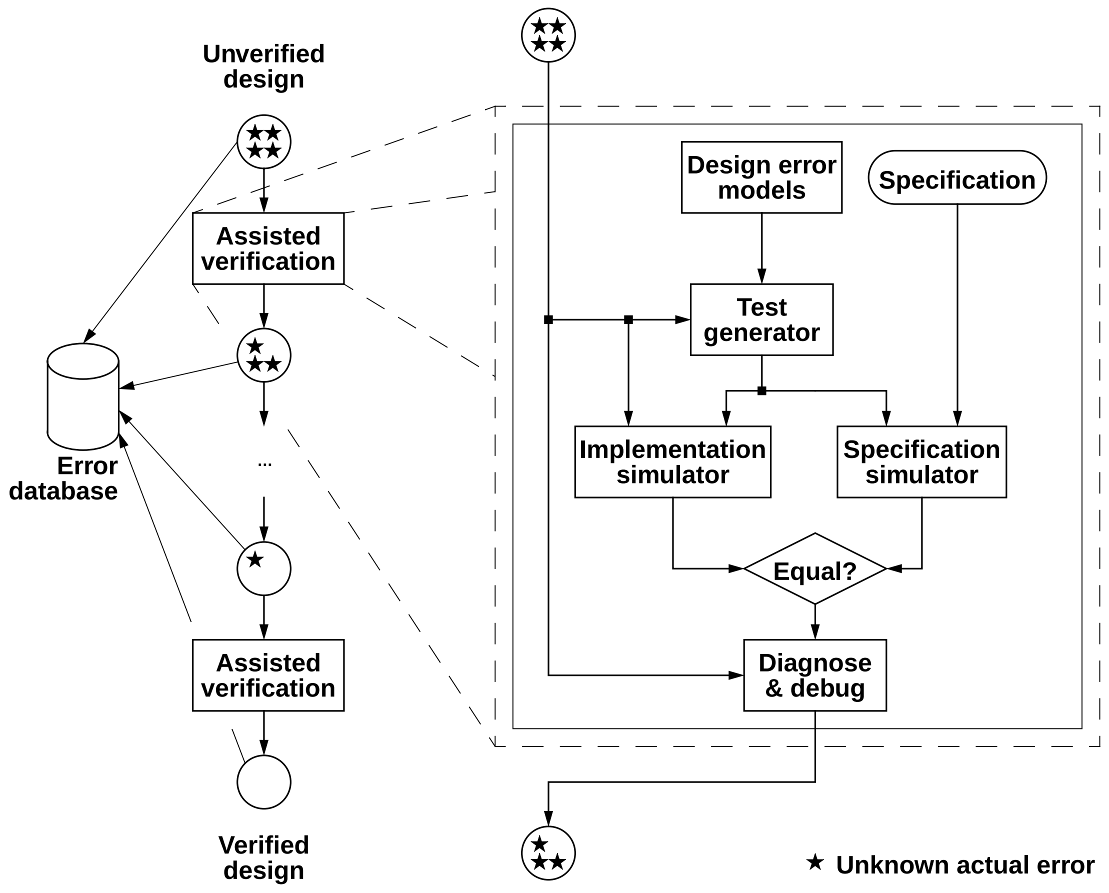


Figure 1.2: Deployment of the proposed verification system

specification. A discrepancy between the two simulation outcomes indicates an error, either in the implementation or in the specification. The figure also shows that throughout the verification process, actual errors are recorded. This information can be used to tune error models.

Chapter 2 examines design error data. Published design error data lack the detail needed to derive structural error models. We therefore devised a systematic method to collect error data. We present and analyze error data we collected from design projects at the University of Michigan.

Chapter 3 develops synthetic error models based on the empirical data. We identify requirements that error models must satisfy to be useful for design verification. We show how well each of the proposed error models meets these requirements.

Chapter 4 considers the problem of generating verification tests for synthetic errors in microprocessors. As we will see, this problem is not unlike test generation for SSL faults in a very large sequential circuit. To cope with this complexity, we consider a limited, but important, class of pipelined microprocessors, and develop a test generation method specific to these designs. To this end, we introduce a model that captures high-level information about the structure of pipelined microprocessors. We then develop a high-level test generation method that exploits the high-level knowledge. We describe experiments to evaluate the effectiveness of this algorithm.

Chapter 5 summarizes our research contributions and presents some directions for future research.

CHAPTER 2
Design error data

Our design verification approach uses design error models to direct test generation. Good design error models should result in test sets that detect many actual design errors. To construct such design error models a good understanding of the nature, frequency, and severity of actual design errors is required. Despite the abundance of design errors in large-scale projects, very little data has been published on these errors. It is common practice in industry to record design errors, but this information is considered proprietary and, perhaps, embarrassing, so it rarely appears in public. These considerations led us to collect error data from design projects at the university. Section 2.1 presents published error data from industry. Section 2.2 describes our method to systematically collect design errors. Section 2.3 presents the error data we collected. Section 2.4 offers some lessons learned. A summary and a discussion of our results is given in Section 2.5.

2.1 Published error data

Although design errors that make their way into final products are common, manufacturers have not always been forthcoming about them. This has changed since MIPS began to publish their bug list beginning with [MIP94]; the Pentium bug [Beiz95] also influenced this change. To give a feel for these errors, we present a few examples of design errors that appeared in major commercial microprocessors.

The errata list for the MIPS R4000PC and R4000SC microprocessors (revisions prior to revision 3.0) [MIP94] documents 55 bugs. Many of these require a rare combination of events before they become visible. The following is a representative bug: If an instruction sequence which contains a load causing a data cache miss is followed by a jump, and the jump instruction is the last instruction on the page and, further, the delay slot of the jump

```
lw          // data cache miss
noop        // one or two noops
jr          // last instruction in the page
------      // page boundary
noop        // first instruction (delay slot of jump)
            // on the next page
```

Figure 2.1: Example of instruction sequence that exposes an error

is not mapped at the time, then the (VM) exception vector is incorrectly overwritten by the jump address. The R4000 will use the jump address as the exception vector. The workaround suggested in [MIP94] is that jump and branch instructions should never be the last location in a page.

Early versions of the Intel 8086 were shipped with the following bug [Ham94]: The architecture specifies that for `MOV` and `POP` instructions to a segment register, interrupts are not to be sampled until completion of the *following* instruction [Int89]. This feature allows a 32-bit pointer to be loaded to the stack pointer registers `SS` and `SP` without the danger of an interrupt occurring between the two loads. However, early versions of the 8086 do not disable interrupts following a `MOV` to a segment register. This causes them to crash when an interrupt uses the stack between `MOV SS, reg` and `MOV SP, op`. A workaround is to insert instructions to temporarily disable the interrupts when reloading `SS`. An uncorrectable problem occurs when an unmaskable interrupt takes place while executing the instruction pair.

These published bug lists are inadequate for error model construction for two reasons: 1) Errata lists typically provide only a programmer's view on the errors. Our error models depend on the design implementation. Therefore, more detailed information about the errors is required, namely the concrete modification to the design implementation that fixes the error. 2) Errata lists only concern errors in the final product. Microprocessor companies go to great efforts to functionally validate their designs. Those design errors that remain undetected before the product is shipped to customers tend to be very subtle and difficult to detect. The majority of all design errors are detected before reaching the customer, and hence are not documented in errata lists. Consequently, errata lists are not representative for the overall population of design errors. These considerations led us to systematically collect design errors from design projects at the university.

2.2 Collection method

The most suitable point to collect design error data is immediately after the design error is discovered and corrected. At that point, all relevant information about the design error should be recorded. This record-keeping requirement conflicts with the interests of the designer. Overhead has to be reduced to a minimum in order to overcome designers' natural reluctance to cooperate.

Our error collection method uses the revision management tool CVS [Cede93]. The revision management tool archives successive revisions of the design. Designers were asked to submit a new revision of their design whenever a design error was corrected and whenever they interrupted work on the design. Some designers resist the system because they see it as a way their work can be monitored. We defused this potential problem by providing designers with a handout [VC97] explaining the use of the revision management system, and by explaining our objectives to obtain the designers' cooperation.

Our first design error collection effort took place during the summer of 1996. Only the bare revision management system was in place. Experience with that project motivated the system described above. It was clear that a standardized form was needed to accompany each revision so that interesting revisions, i.e., those involving a design error correction, can be separated from other revisions. We therefore augmented the revision management system so that each time a new revision is submitted, the user is prompted to fill out a questionnaire. The questionnaire, in the form of a multiple choice form shown in Figure 2.2, gathers four pieces of information: 1) the motivation for revising the design; in the case of a bug, the following apply as well: 2) the method by which the bug was detected, 3) the class to which the bug belongs, 4) a short description of the bug. Design errors can be detected by reading the HDL code (inspection), by syntax checking performed by the HDL simulator (compilation) or a synthesis tool (synthesis), or by logic

```
(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:

     _ wrong signal source
     x 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:

   Forgot to select NOP in case of stall
```

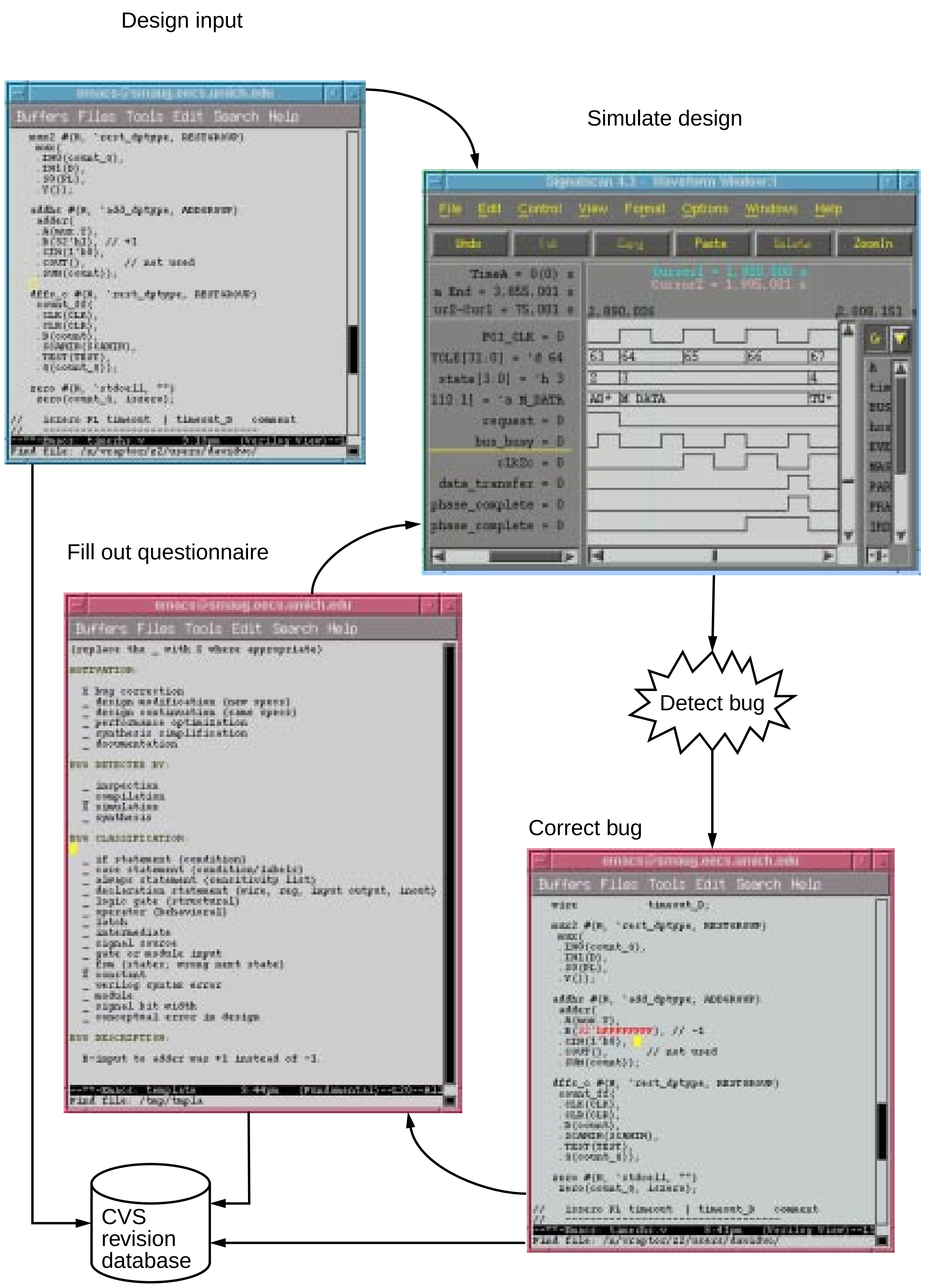
Figure 2.2: Bug report example

simulation. The operation of our error collection method within the design cycle is illustrated in Figure 2.3.

From the raw revision management data, we identified the design modifications to fix each error by computing the differences between successive revisions. The analysis of the design error data lead to a preliminary classification of design errors. This classification was used in our first major design error collection effort, which took place in the fall term of 1996. Analysis of this design error data lead us to revise our classification. The result is shown in Figure 2.2. The categories are not completely disjoint, so designers were asked to check all applicable categories.

2.3 Collected error data

Design projects. Design error data was collected from both class design projects and research projects at the University of Michigan. All of the designs were described in Verilog [IEEE96]. Table 2.1 lists these projects. LC2 concerns the design of the Little Computer 2 (LC-2) [Post96a], which is a small microprocessor used for teaching purposes at the University of Michigan. The design of both a behavioral and a synthesizable register transfer level model was carried out by Hussain Al-Asaad [AA98] in the summer of 1997. DLX1, DLX2, and DLX3 concern design projects that were undertaken as part of the senior/first-year-graduate level computer architecture course (EECS470) in the fall of 1996. Students designed a pipelined implementation of the DLX [Henn90] microprocessor at the structural level. X86 concerns an EECS470 design project carried out in the Fall of 1997. Students designed a pipelined implementation of a subset of the Intel x86 architecture [Int89]. FPU concerns the design of a floating-point unit for the PUMA processor [Brow96], which is a PowerPC microprocessor implemented in complementary GaAs process technology, and was undertaken as part of the graduate level VLSI design class (EECS627). Both a purely behavioral and a mixed synthesizable behavioral/structural model were designed. FXU concerns the design of the fixed-point unit of the PUMA processor. James Dundas and Todd Basso wrote the synthesizable behavioral description in the Fall of 1996. For each of the projects the table lists the


Design input
Simulate design
Fill out questionnaire
Detect bug
Correct bug
CVS revision database
Buffers Files Tools Edit Search Help
File Edit Control View Format Options Windows Help
Undo Cut Copy Paste Delete ZoomIn

```
  mux2 #([illegible], `[illegible]_dptype, [illegible])
   mux(
    .IN0(count_[illegible]),
    .IN1(D),
    .S0(PL),
    .Y());

  adder #([illegible], `add_dptype, [illegible])
   adder(
    .A(mux_[illegible]),
    .B(32'h1), // +1
    .CIN(1'b0),
    .COUT(),      // not used
    .SUM(count));

  dffr_c #([illegible], `[illegible]_dptype, [illegible])
   count_ff(
    .CLK(CLK),
    .CLR(CLR),
    .D(count),
    .SCANIN(SCANIN),
    .TEST(TEST),
    .Q(count_[illegible]));

  zero #([illegible], `stdcell, "")
   zero(count_[illegible], [illegible]);

//   [illegible] timeout | timeout_b   comment
//   ------------------------------------
```

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

MOTIVATION:

  X bug correction
  _ design modification (new specs)
  _ design continuation (same specs)
  _ performance optimization
  _ synthesis simplification
  _ documentation

BUG DETECTED BY:

  _ inspection
  _ compilation
  X simulation
  _ synthesis

BUG CLASSIFICATION:

  _ if statement (condition)
  _ case statement (condition/labels)
  _ always statement (sensitivity list)
  _ declaration statement (wire, reg, input output, inout)
  _ logic gate (structural)
  _ operator (behavioral)
  _ latch
  _ intermediate
  _ signal source
  _ gate or module input
  _ fsm (states; wrong next state)
  X constant
  _ verilog syntax error
  _ module
  _ signal bit width
  _ conceptual error in design

BUG DESCRIPTION:

  B-input to adder was +1 instead of -1.

Find file: /tmp/tmpl[illegible]
```

```
  wire           timeout_b;

  adder #([illegible], `add_dptype, [illegible])
   adder(
    .A(mux_[illegible]),
    .B(32'hffffffff), // -1
    .CIN(1'b0),
    .COUT(),      // not used
    .SUM(count));
```

Figure 2.3: Error collection system

Table 2.1: Design projects for which error data was collected

Project	Class	Date	Duration [days]	No. of designers	Code size [lines]	No. of errors
LC2	N/A	Summer ‘97	11	1	1,179	22
DLX1	EECS 470	Fall ‘96	16	1	3,010	39
DLX2	EECS 470	Fall ‘96	21	1	3,015	35
DLX3	EECS 470	Fall ‘96	29	1	5,210	13
X86	EECS 470	Fall ‘97	42	3	6,071	59
FPU	EECS 627 / PUMA	Fall ‘96	96	2	5,607	17
FXU	PUMA	Fall ‘96 - Winter ‘97	135	2	27,587	113

number of designers, the duration of the design entry and logic debug part, the size of the design description, and the number of errors that were logged. Design verification for the class projects relied on simulating the design for a few handwritten assembly programs. Simulation outcome was checked by comparing the final state of the processor, and by examining internal signals over the duration of the simulation. For the FXU project, designers also wrote a random program generator, and used that to augment the handwritten test cases.

Data of one project in detail. In this section we examine the data obtained from design project X86. This was chronologically the latest project we collected data from, and hence it benefited the most from past experience.

Table 2.2 lists the design files created in this project. For each file, we list its size, the total number of revisions it underwent, and the number of design errors recorded, broken down by detection method. Note that in this project no synthesis tools were used; hence no errors were detected this way. Errors of interest are those detected by inspection or simulation. The designers were aware that syntax errors are of very little value to our work. We can therefore assume that many syntax errors were corrected without recording a new design revision, and hence do not appear in the table the column “compilation.”

Figure 2.4 shows the difference between a design revision motivated by an error correction and the previous revision. In revision 1.49, NOR gate Controls_NOPsel_nor2 misses input Stallin. Revision 1.50 corrects this error.

Table 2.2: Design files written for the X86 project

Design file	Code size [lines]	No. of rev-isions	Errors-detection method			
			Inspec-tion	Comp-ilation	Simu-lation	Syn-thesis
decode.v	984	63	1	2	18	0
datapath.v	530	54	0	9	12	0
stages1.v	294	19	0	1	9	0
modules1.v	1750	27	1	3	8	0
smallmodules.v	1010	21	0	4	2	0
fetch.v	140	23	1	2	2	0
datacaches.v	674	13	0	0	1	0
exe1.v	135	8	0	1	1	0
modules.v	554	27	3	1	0	0
Total	6071	255	6	23	53	0

```
Index: project/decode.v
===================================================================
RCS file: /x/users/davidvc/repositories/repositories_470_f97/jhauke/470_reposit
ory_98/project/decode.v,v
retrieving revision 1.50
retrieving revision 1.49
diff -r1.50 -r1.49
3c3
< $Revision: 1.50 $
---
> $Revision: 1.49 $
5c5
< $Date: 1997/12/13 22:45:54 $
---
> $Date: 1997/12/13 20:43:41 $
878c878
<    nor4$ Controls_NOPsel_nor2(Controls_NOPsel_nor2_out,
CounterInput,HLT_NOP,ScoreNOP,Stallin);
---
>    nor3$ Controls_NOPsel_nor2(Controls_NOPsel_nor2_out,
CounterInput,HLT_NOP,ScoreNOP);
```

Figure 2.4: Difference between two successive revisions

Table 2.3 gives the distribution of design errors by error category. The dominant type of design error is wrong signal source. Errors involving missing logic are also notable and amount to 31%.

Figure 2.5 shows the evolution of the project over time. HDL coding and debugging spanned 42 days in this project. The chart shows the total size of the design at the end of each day. Also shown is the number of lines of code that were touched over the duration of

Table 2.3: Error distribution in X86

Error category	Frequency
Wrong signal source	32.8%
Missing instance of gate/module	14.8%
Missing input(s)	11.5%
Wrong gate/module type	9.8%
Unconnected (floating) input(s)	8.2%
Missing latch/flipflop	6.6%
Conceptual error	4.9%
Wrong next state	3.3%
Other finite state machine error	1.6%
Extra term/factor	1.6%
Extra inversion	1.6%
Wrong bit order	1.6%
Other	1.6%

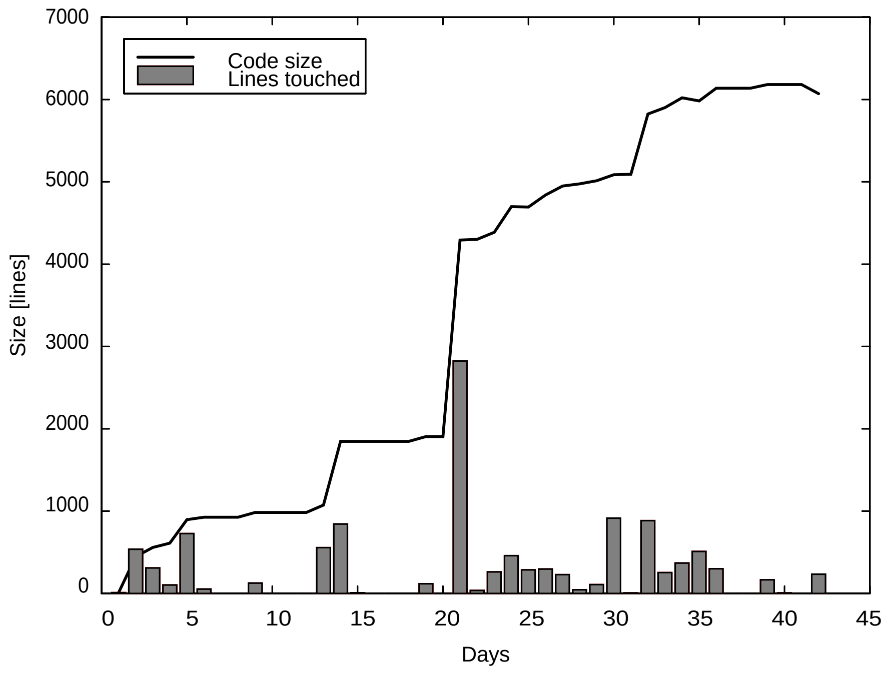


Figure 2.5: Project evolution: code size [lines] and lines touched over time

each day. Most of the design description is in place by day 21, and integration testing can start.

Figure 2.6 shows the number of revisions over the duration of the project. The number of revisions logged on any day is broken up into revisions that are due to bug corrections

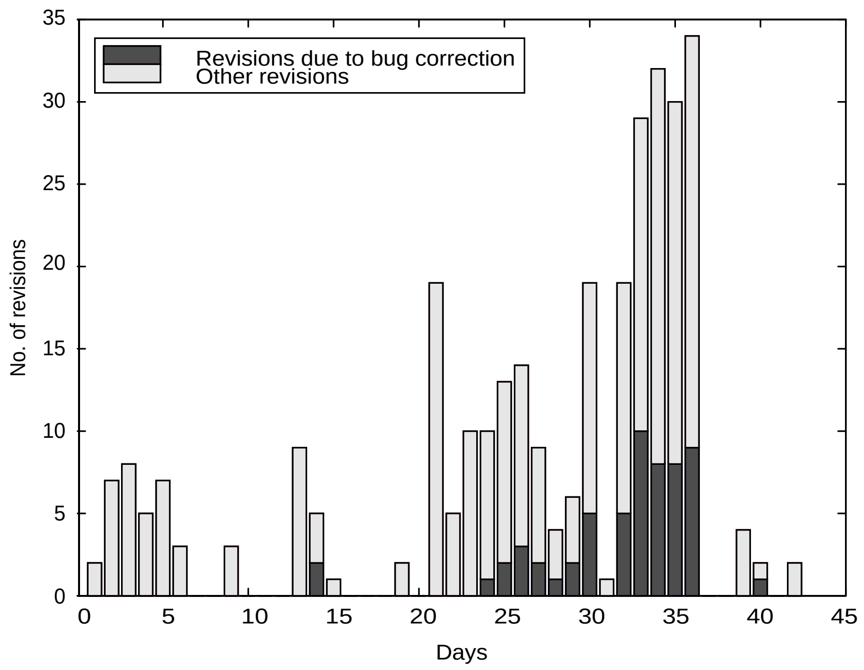


Figure 2.6: Revisions motivated by bug correction and other revisions over time

and those due to other reasons. Ideally, there is a one-to-one correspondence between uncovered design errors and revisions motivated by error correction. Hence the bar for number of revisions logged due to error corrections also gives the total number of bugs corrected during the corresponding day. It can be seen that most of the bugs were discovered and corrected in the second half of the project.

Figure 2.7 plots the time at which each error was corrected versus the number of lines of code that were touched to correct the error. The vertical coordinate is an indication of the structural complexity of the error. Although easy to compute, this metric is far from ideal. It does not distinguish between lines of code that have merely been reformatted and lines that have truly been changed. More accurate measures, such as the minimum number of ‘atomic’ modifications needed to remove the error from the control dataflow graph of erroneous circuit, would be more appropriate but are also much harder to compute. For about half of the errors fewer than ten lines of code were involved, and only four errors resulted in modification to the design involving more than 100 lines of code.

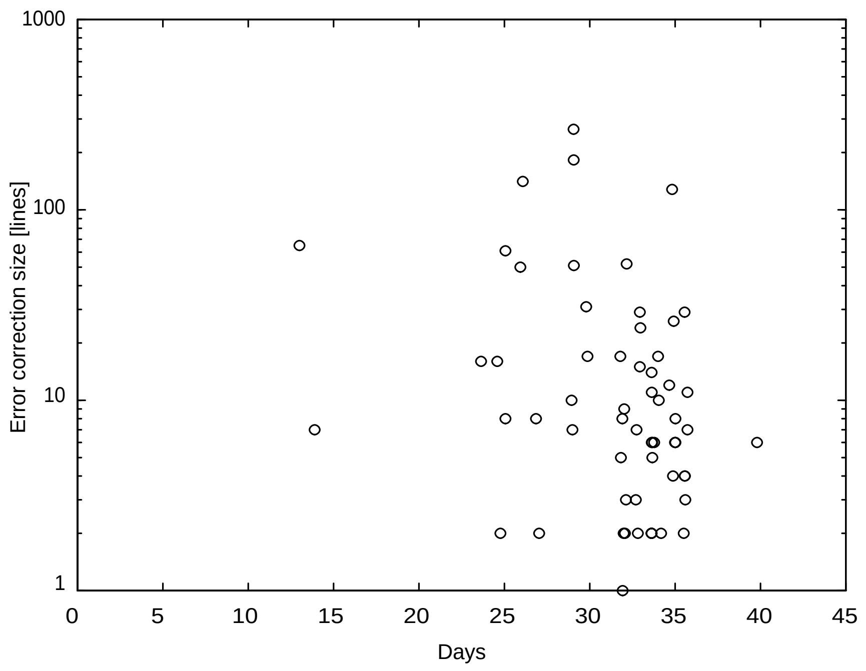


Figure 2.7: Design errors: time to discovery [days] vs. error size [lines]

We further characterize these design errors based on purely structural properties. We define the *size* of an error as the order of the polynomial that computes the number of similar errors as a function of the size of the circuit. For example, single inversion errors and single-stuck errors both are of size 1, because there are $O(N^1)$ such errors in a circuit with N lines. Signal source errors are of size 2 as there are $O(N^2)$ such errors. We noted that some actual errors consist of multiple instances of the same type of error. An example is an inversion error on a port connection of a module instance that is repeated for all instances of the module. We define the *multiplicity* of an actual error as the number of identical and repeated instances of a simpler error that constitute the actual error. Figure 2.8 plots the frequency of design errors when binned according to size and multiplicity. We observe that design errors of higher multiplicity are rare. Design errors with multiplicity 1 and sizes 1 or 2 account for more than half of all design errors. Only about 12% of the errors are very complex, as indicated by a size of 10 or greater.

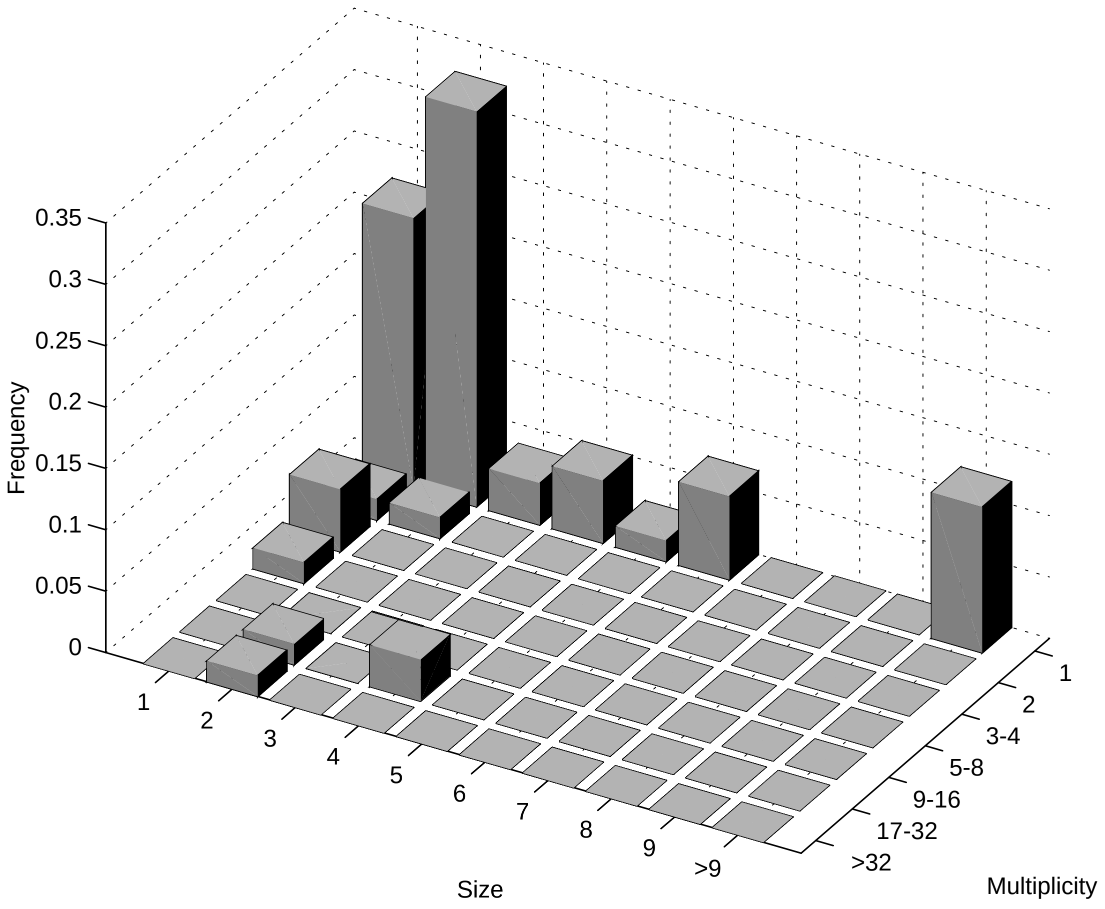


Figure 2.8: Frequency of design errors in function of their size and multiplicity

2.4 Guidelines for implementing an error collection system

Revision management. A revision management system has proven priceless. Not only does it allow detailed analysis of concrete design errors, but it also came to be valued by the designers. One reservation some designers have with these systems is that they see it as a way to monitor their work. This can usually be overcome by explaining the intent and the benefits.

The stigma of bugs. A key factor to success is to remove the stigma usually associated with design errors. We made an effort to make designers feel engaged with our research project, and explained to them the need for collecting error data. The participation of class projects in error collection was on a voluntary basis.

Overhead to the designer. The need to minimize the overhead of error logging for the designer cannot be underestimated. Although the designer is, in principle, in the best position to classify each newly discovered error, this small effort, from which the designer may not see any immediate benefit, may be felt as burden or threat. Consequently, the designation of errors often becomes imprecise. We observed that for periods some designers marked all of there errors as *conceptual error*, even if the actual error involved a single inversion error. This led us to reassess the raw revision data, and explains the discrepancies between the data reported here and that in our earlier work [VC98]. The reassessment also corrected the counts assigned to errors that spanned multiple design files. Previously, these errors had been overrepresented. This adjustment primarily affects the bigger designs where such errors occurred more often.

A key element in an error collection effort is to encourage designers to adopt the habit of *systematically* recording every *single* design error that is not a syntax error. Simple errors such as single inversion errors don't require much explanation. For more elaborate errors a brief textual description of the error, already in the present error template, is very helpful to analyze the error afterwards. Additional pieces of information could include a measure of the difficulty of detecting the error, and the root source of the error. Typical root sources include: oversight, failure to consider certain behavior, wrongly implemented behavior, misunderstanding of specification, or miscommunication between designers.

Designers should not be burdened with classifying the errors with respect to their structural aspects (item 3 of our questionnaire). This task can be performed by those analyzing the error data provided that a new design revision has systematically been recorded for each detected error.

Practical considerations. Finally some practical considerations need to be pointed out. Fixing a single design error may require multiple modify / simulate cycles, and hence multiple revisions. The designer should record information to distinguish such revisions. Fixing a single design error may require modifications to multiple files. Designers should

Table 2.4: Design error distributions [%]

Category	LC2	DLX1	DLX2	DLX3	X86	FPU	FXU	Average
Wrong signal source	27.3	31.4	25.7	46.2	32.8	23.5	25.7	30.4
Missing instance		28.6	20.0	23.1	14.8	5.9	15.9	15.5
Missing inversion		8.6				47.1	16.8	10.3
New category	9.1	8.6		7.7	6.6	11.8	4.4	6.9
Unconnected input(s)		8.6	14.3	7.7	8.2	5.9	0.9	6.5
Missing input(s)	9.1	8.6	5.7	7.7	11.5			6.1
Wrong gate/module type	13.6		11.4		9.8			5.0
Missing term/factor	9.1	2.9	5.7				4.4	3.2
Always statement	9.1		2.9				2.7	2.1
Wrong constant	9.1						5.3	2.1
Missing latch/flipflop					4.9	5.9	0.9	1.7
Wrong bus width	4.5						7.1	1.7
Missing state	9.1							1.3
Conflicting outputs				7.7				1.1
Wrong constant			2.9				4.4	1.0
Conceptual error			2.9		3.3		0.9	1.0
Signal declaration			5.7					0.8
Extra term/factor		2.9			1.6		0.9	0.8
Wrong operator							4.4	0.6
Gate or module input			2.9					0.4
Case statement							2.7	0.4
Other FSM error					1.6			0.2
Extra inversion					1.6			0.2
Wrong bit order					1.6			0.2
Wrong next state					1.6			0.2
Latch							0.9	0.1
If statement							0.9	0.1
Expres. completely wrong							0.9	0.1

submit new revisions for all of these files together. Otherwise, these revisions data can wrongly be interpreted as concerning multiple errors.

2.5 Discussion

Table 2.4 shows the error distributions for all projects. Also listed is the average error frequency over all projects. We observe that signal source errors are the most common type of error at 30%. Errors involving missing logic (missing instance, missing input, missing term, missing state) are the second most common group at 26%. Also notable is that apparently very simple errors, such as extra/missing inversions and unconnected inputs, account for 17% of all errors. More detailed analysis of these simple errors shows that some of these were detected late in the project. This indicates that the behavior of some parts of the design is not properly exercised, since these simple errors do not require any activation conditions. Among the errors marked as *new category* are timing errors, and errors that required very elaborate corrections.

The limitations of our error collection effort are as follows. Student designers have limited experience. Class projects are short in duration and the verification effort in these projects is modest. Consequently our data may contain a disproportionately small number of hard-to-detect errors, compared to data from industrial design projects. This concern also applies to the data from the projects related to PUMA, but to a lesser extent.

CHAPTER 3
Design error models

Manufacturing testing uses logical fault models to guide test generation. 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. Similarly, we use design error models to drive verification test generation. This chapter presents and studies design error models that are based on the error data described in the previous chapter.

Section 3.1 presents four requirements that error models should satisfy to be useful for design verification. Section 3.2 proposes three classes of error models: *basic*, *extended*, and *conditional* error models. The following sections analyze how well these error models meet the requirements: Section 3.3 analyzes the number of error instances defined by each model (requirement 4). Section 3.4 analyzes test generation with the error models (requirement 2). Section 3.5 analyzes error simulation and presents an efficient error simulation technique for conditional error models called CESIM (requirement 3). Section 3.6 presents an analytical coverage evaluation of one conditional error model (requirement 1). Section 3.7 presents an experimental coverage evaluation using error simulation (requirement 1). Another experimental study with the same goal but a different approach is detailed in Section 3.8 (requirement 1). Our findings are summarized in Section 3.9.

3.1 Error model requirements

A *design error model* defines a class of *modeled errors*, also referred to as *synthetic errors*, for a given design. In design verification, design error models play the role of fault models in physical fault testing. The different terminology, *error* vs. *fault*, is to underscore the different contexts. To be useful for design verification, error models should satisfy the following four requirements:

1. Tests (simulation vectors) that are complete for 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 modeled errors should be amenable to automated error simulation.
4. The number of modeled errors should be relatively small.

Since exhaustive simulation is prohibitive for practical verification problems, any simulation-based method can only aspire to produce test sets that, at the end of the debug process, give very high confidence in the functional correctness of the design. For our approach, this goal crystallizes in requirement 1 on design error models and in the necessity for an efficient method to generate complete tests for the synthetic errors. A naive attempt to satisfy requirement 1 would be direct modeling of concrete design errors based on error statistics (see Chapter 2). Unfortunately this would lead to a vast array of models, thus greatly complicating automation of test generation (requirement 2), and an extremely large number of synthetic errors to be targeted. However, error models are only a means to generate high quality tests. This is similar to physical fault testing. Studies have shown that despite the fact that very few physical defects behave precisely like SSL faults, complete test sets for SSL faults detect most manufacturing defects. This motivates us to attempt to develop design error models that balance requirement 1 with the three other requirements.

3.2 Design error models

Standard logic simulation and synthesis tools have the side effect of detecting some classes of design errors (Table 2.4); hence there is no need to develop models for those particular errors. For example a logic simulator such as Verilog-XL [Cad94] flags all Verilog syntax errors, and incomplete port lists of modules. Also, logic synthesis tools, such as those of Synopsys [Syn97], usually flag wrong bus-width errors and sensitivity-list errors in the *always* statement.

Basic error models. A set of error models that satisfy the requirements for the restricted case of gate-level logic circuits was developed in [AA95]. Motivated by our empirical design error data, similar error models for higher-level (RTL) designs appear to be useful. We propose the following 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 [Bhat85] in the context of physical fault testing.
- *Module substitution error (MSE):* This refers to mistakenly replacing a module by another module with the same number of inputs and outputs. 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; mistakenly reversing the order 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.
- *Bus driver error (BDE)*: This refers to mistakenly driving a tristate bus from two sources at the same time.

Extended error models. Prior work on SSL error detection [Abad88, Bhat85], shows that basic error models can be used to generate test sets that provide high, but not complete, error coverage. These results are further reinforced by our experiments on microprocessor verification (Section 3.8), 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 2.4 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.
- *Module count error (MCE):* This corresponds to incorrectly adding or removing a module, which includes extra/missing word gate errors and extra/missing registers.

- *Label count error (LCE)*: This error corresponds to incorrectly adding or removing the labels of a case statement.
- *Expression structure error (ESE)*: This includes various deviations from the correct expression, such as extra/missing terms, extra/missing inversions, wrong operator, and wrong constant.
- *Next state error (NSE)*: This error corresponds to incorrect next state function in a finite-state machine (FSM).

Although targeting these extended error models can increase coverage of actual errors, we have found them too complex for practical use in manual or automated test generation. Analysis of the more difficult actual errors revealed that these errors are often composed of multiple basic errors, and that the component basic errors interact in such a way that a test to detect the actual error must be much more specific than a test to detect any of the component basic errors. An effective error model should necessitate the generation of these more specific tests without resorting to direct modeling of the composite errors. The complexity of the new error models should be comparable to that of the basic error models and the (unavoidable) increase in the number of error instances should be controlled to allow trade-offs between test generation effort and verification confidence. These requirements can be combined by augmenting the basic error models with a condition.

Conditional error models. 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 ($y_i = w_i$), where y_i is a signal in the circuit that is not in the transitive combinational fanout of the basic error[1], and w_i is a constant of the same signal-width as y_i and whose value is either all-0's or all-1's. 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 (*CE*) types:

- conditional single-stuck line error of order n (CSSLn)

1. The requirement that condition signals are not to be part of the transitive combinational fanout of the basic error, eliminates problems of combinational feedback, and thus ensure that all conditional errors are well defined. This requirement also facilitates efficient error simulation, as we will see in Section 3.5.

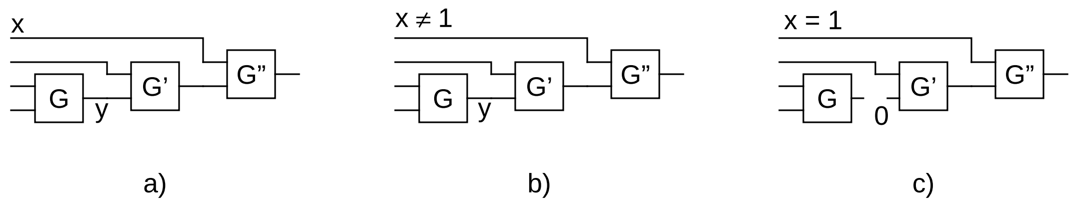


Figure 3.1: CSSL1 error ($x = 1$, y / 0): a) error-free design; b) erroneous design with $x \neq 1$; c) erroneous design with $x = 1$

- conditional bus order error of order n (CBOEn)
- conditional bus source error of order n (CBSEn)

Figure 3.1 gives an example of a CSSL1 error, ($x = 1$, y / 0). If the condition does not hold, $x \neq 1$, the erroneous circuit operates as the error-free. If the condition holds, $x = 1$, line y is stuck at 0.

3.3 Number of error instances defined by error model

The fourth requirement on error models states that the number of modeled errors should be sufficiently small. Consider a design N signals; we denote by #M the number of error instances defined by error model M on the design.

Basic error models

- #SSL = $O(N)$
- #MSE = $O(N)$
- #BOE = $O(N)$
- #BSE = $O(N^2)$
- #BDE = $O(B.D^2)$, where B is the number of tristate buses, and D is the rms[1] number of drivers on a bus.

1. $\sum_{i=1}^{B} D_i^2 = B\left(\sqrt{(1/B)\sum_{i=1}^{B} D_i^2}\right)^2 = BD^2$

Extended error models

- #BCE = $O(N^2)$
- #MCE = $O(N^2)$
- LCE. Consider a case statement with an n-bit signal making the selection; let L be the number of labels (branches). The simplest type of missing label error would occur if the statements selected by the missing label are identical to those of another (not missing) label. Hence, there are $(2^n$-$L)$ missing label errors and L extra label errors. For a circuit with C case statements, we have #LCE = $O(C.2^n)$ LCE's with n appropriately averaged.
- ESE. Consider an expression with L literals and E subexpressions. If we restrict missing and extra term errors to a single literal, and further require that the missing literal appears elsewhere in the expression, then there are $O(L)$ extra term errors and $O(E.L)$ missing term errors.
- NSE. Consider an FSM with S states, and E distinct state transitions. A simplest next state error would occur if for one of the state transitions, the next state was wrong. There are $O(S.E)$ of this type.

Conditional error models

The number of instances defined by a conditional error model (C,E) is given by the product of the number of basic errors and the number of conditions:

- #CSSLn = $O(2^{n+1}N^{n+1})$
- #CBOEn = $O(2^nN^{n+1})$
- #CBSEn = $O(2^nN^{n+2})$

For $n = 0$, 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. For example, the CSSL1 model defines a number of instances quadratic in the size of the circuit. Although the total set of all signals we consider for each term in the condition can possibly be reduced, CSSLn errors where $n > 1$ are probably not practical.

3.4 Test generation

The second requirement on design error models states that error models should be amenable to automated test generation. In this section we investigate how well our error models meet this requirement.

The basic and conditional error models have the property that each error instance together with the given design completely specifies an erroneous design, and that both designs structurally differ in a very localized part. Consequently, the test generation problem is well defined and very similar to that of test generation for SSL faults. For combinational circuits the D-calculus suffices, and for sequential circuits a 9-valued logic is sufficient. Targeted test generation for SSL fault can be decomposed into three subproblems: fault excitation, fault propagation, and line justification. Test generation for basic and conditional errors differs only in the excitation subproblem.

Some of the extended error models, e.g., ESE, need to be specified further so that they completely define a set of erroneous designs. Although test generation for these errors is in principle similar to test generation for SSL faults, automation is greatly complicated by the variety and complexity of the excitation conditions associated with these errors. We therefore conclude that the extended error models are not practical, and drop them from further discussion.

Error instances of more abstract error models such as the universal fault model [Abra90] and the tag model used in OCCOM [Fall98b] do not completely specify an erroneous machine. Instead, an instance together with the given design specifies a class of erroneous designs. The aim is then to generate a test that will distinguish each element of that class from the original design. In case of the tag model, the error effect propagation is approximate. Consequently, an interpretation of tag coverage in terms of what concrete design errors have been tested for is not possible.

There are two main approaches for generating tests for our synthetic error models. The first approach is to modify the original design and to apply existing ATPG tools; Al-Asaad and Hayes follow this approach in their work on verification of gate-level designs [AA95]. Another approach is to adapt existing ATPG algorithms to directly target the synthetic errors. The major advantage of the first approach is that it leverages existing ATPG technology. When the error model defines a super-linear number of error instances, this approach may suffer from the significant increase in the size of the modified netlist. The second approach overcomes this problem. For the CSSL*n* models the effort required to modify an SSL test generator is modest: error excitation requires the justification of *n* extra lines over SSL excitation. For MSE’s, BSE’s, BOE’s, BDE’s a larger effort is required.

Another consideration about test generation is that most commercial ATPG tools operate on gate-level netlists. Our design error models are primarily targeted for use with behavioral and structural RTL descriptions. The structure in these higher-level descriptions can be used beneficially during test generation, and therefore high-level test generation carries the promise of being able to handle larger designs than gate-level test generation. Nevertheless, gate-level test generators can still be used as follows: The higher level description can be mapped (synthesized) to a gate-level design that preserves all signals of the high-level version. Such gate-level representations have the property that for each basic or conditional error instance, we can find a corresponding set of one or more error instances at the gate-level. More than one error instance is required at the gate level if the original error instance involves multibit buses. For example *A* stuck-at-0, maps onto *w* SSL errors at the gate-level: A_i stuck-at-0 $i = 0 \ldots (w - 1)$, where *w* is the width of *A*.

3.5 Error simulation

The problem addressed by error simulation is as follows. Given a design, a set of (synthetic) design errors, and a sequence of test vectors, determine which errors are detected by the test sequence. Fault simulation addresses a similar problem in physical fault testing, but differs in the error/fault models. Whereas physical fault testing is concerned with SSL faults, bridging faults, open faults, our design verification methodology needs to consider other errors, such as the conditional errors introduced earlier. In this section we address error simulation with conditional errors. First, we motivate error simulation.

Augmenting targeted test generation with error simulation can reduce overall run times. Test generators typically target one error at a time. A targeted test may detect errors other than just the targeted error. These errors can be identified by an error simulator so that they do not need to be considered by the test generator any more.

A stand-alone use of error simulation is the computation of design error coverage of a given test suite. This is useful in regression testing, where one might be interested in selecting a subset of a given set of test sequences that provides coverage of design errors similar to those of the complete test set. Error simulation can also reveal areas of the design that are not sufficiently tested by a given test suite, and hence spur further targeted test generation.

Error simulation needs to be efficient. Not only the length of test suites, which is extremely large for pseudo-random tests, but also the nature of the error models, and the number of error instances to be considered affect the size of the task. It is clear that better methods are required than simple serial error simulation, which simulates the erroneous designs for the complete test suite one by one.

In the remainder of this section we first discuss related work and we then present our method for error simulation with conditional errors; we conclude with experimental results.

Related work

Representative approaches to fault simulation for sequential circuits [Abra90, Nier91a] are parallel, concurrent, deductive, and differential fault simulation. *Parallel* fault simulation takes advantage of the word-level parallelism of the computer used. On a 32-bit computer, 32 faulty machines can be simulated in parallel. This method lacks the ability to drop errors. The other methods are motivated by the observation that as long as a fault is not detected, the good and faulty circuit differ in only a fraction of the number of signals present. For this purpose, such methods process the complete set of faulty machines one vector at a time. Both *concurrent* and *deductive* fault simulation compute the node values of a faulty machine for the current vector, based on the good circuit's node values for the current vector, and the faulty machine's node values for the previous vector. A drawback of both methods is high memory requirement. *Differential* fault simulation, a variant of concurrent fault simulation, addresses the memory problem, but suffers from the inability to drop detected faults.

Niermann, Cheng and Patel [Nier90, Nier91a] described a fault simulator, called PROOFS, that combines ideas of concurrent, differential and parallel fault simulation. As our error simulation method for conditional errors derives from PROOFS, we briefly describe its main features, referring to Figure 3.2.

Given is a gate-level sequential circuit, a fault list, and a test vector sequence, PROOFS maintains two sets of signal values: one for the good, and one for a faulty machine. For each undetected fault, PROOFS also stores the difference in present state between the good machine and the corresponding faulty machine.

The outermost loop of PROOFS processes one test vector at a time. First, the good machine is simulated for the current vector. Next, faults that are active for the current test vector are identified. A fault is considered *active* if one or both of the following two conditions holds: 1) the present state of the faulty machine is different from that of the good machine; 2) the fault is excited by the current vector, and the faulty line is sensitized through the first two levels of logic. Checking condition 1 is straightforward since we have saved the faulty circuit's state while processing the previous vector. If condition 1 does not hold, that is, if the faulty circuit's present state is identical to that of the good circuit, checking condition 2 is inexpensive too, as it is very localized and requires only the good circuit's values.

Faults that are not active for the current vector have the property that they are not detected by the current vector *and* the next states of the corresponding faulty machines are identical to the next state of the good machine. Consequently, there is no need to simulate these faulty machines for the current vector.

Each active fault is processed as follows: First, the fault is injected into faulty circuit. The event list is initialized to reflect the fault injection and the present state lines whose values differ in the good and the faulty machine. The event-driven simulation of the faulty machine in PROOFS typically has a very low event activity, as in concurrent fault

```
            PROOFS(circuit, faultList, testVectorSequence)

1.          while (vectors left) {
1.1             read next vector
1.2             simulate good circuit
1.3             determine which faults are active
1.4             for each active fault {
1.4.1               inject fault
1.4.2               add faulty node events
1.4.3               simulate faulty circuit
1.4.4               drop detected faults
1.4.5               store faulty next state
1.4.6               remove fault
1.5             }
2.          }
```

Figure 3.2: PROOFS' error simulation algorithm

simulation. If the fault is detected by the current vector, it is dropped. Otherwise, the difference between the next state of the faulty machine and that of the good machine is saved.

The basic algorithm, as discussed above, can be augmented to take advantage of the word-level parallelism available on the computer executing the fault simulator. On a 32-bit machine, up to 32 iterations the simulation step 1.4.3 of loop 1.4 can be executed in parallel. This is done by assigning the values of different faulty machines to different bit positions within a word. The other steps of loop 1.4 still have to be executed serially. A more detailed description of one implementation is given in [Nier90, Nier91a].

Extension to conditional errors

It is straightforward to modify PROOFS to handle conditional errors, such as CSSL1. For a given circuit and a given test sequence, the average run time per error for CSSL1 error simulation is very close to that for SSL error simulation. As the number of CSSL1 errors is quadratic in the size of the circuit, the cost of error simulation for CSSL1 may be prohibitively large. To address this, we develop an error simulation algorithm for conditional errors, called CESIM, that exploits the close relationship among CSSL1 errors derived from the same CSSL0 error. Its key features are processing of sets of conditional errors, and the injection of basic (instead of conditional) errors. We will demonstrate that this leads to improved performance over the naive extension of PROOFS.

First, we define two equivalence relations on conditional errors. Two conditional errors are *PS-equivalent* with respect to the current vector iff the present states of the corresponding erroneous machines are identical. Two conditional errors are *PSBE-equivalent* with respect to the current vector, iff they are derived from the same basic error and the present states of the corresponding erroneous machines are identical. The two equivalence relations define a hierarchical partition on the set of conditional errors; PSBE-equivalence refines the partition defined by PS-equivalence. CESIM maintains the set of undetected errors in partitioned form.

We redefine the activity criterion of PROOFS as follows: A conditional error is *active* for the current vector iff 1) its condition holds in the erroneous circuit, and 2) the corresponding basic error is excited in the erroneous machine for the current vector, and 3) that basic error is sensitized through the first two levels of logic.

The inner loop 1.4 of PROOFS (Figure 3.2) that iterates over individual active faults is replaced in CESIM, outlined in Figure 3.3, by one that iterates over sets of PS-equivalent conditional errors:

Given is a set S_1 of undetected PS-equivalent conditional errors, we process this set of errors for the current vector as follows. First, we simulate the erroneous machine[1] with no errors injected, starting from the present state associated with S_1 for the current vector (steps 2.3.1 and 2.3.2 of Figure 3.3).

For each conditional error in S_1, we check if it is active. Activation is determined by three conditions (see above). Conditions 2 and 3 only depend on the basic error, and hence are identical for all PSBE-equivalent errors. We therefore check conditions 2 and 3 first (one check for each class of PSBE-equivalent errors). Only if both conditions hold do we have to check condition 1 (one check per individual conditional error). Note that lines

1. Note that CESIM uses a single copy of the circuit structure but associates two values with each signal, one corresponding to the error-free machine, the other to an erroneous machine. Hence by simulating the *erroneous machine* we mean simulating the circuit using the set of *erroneous values*.

```
          **CESIM**( circuit, errorList, testVectorSequence)

1.        U = errorlist  /* hierarchically partitioned set of undetected errors */
2.        while (vectors left) {
2.1           read next vector
2.2           simulate good circuit
2.3           for each set $S_1$ of PS-equivalent contional errors in U {
2.3.1             add the erroneous present-state events
2.3.2             simulate the erroneous machine (no errors injected)
2.3.3             partition $S_1$ into an active and an inactive subset, A, and D, resp.
2.3.4             if error effect is exposed {
2.3.4.1               drop all errors in D
2.3.5             }
2.3.6             else {
2.3.6.1               save next state for D
2.3.6.2               insert D in nextU
2.3.7             }
2.3.8             for each set $S_2$ of PSBE-equivalent conditional errors in A {
2.3.8.1               inject the corresponding basic error
2.3.8.2               add the erroneous node events
2.3.8.3               simulate the erroneous circuit
2.3.8.4               if basic error is detected {
2.3.8.4.1                 drop all errors in $S_2$
2.3.8.5               }
2.3.8.6               else {
2.3.8.6.1                 save the erroneous next state for $S_2$
2.3.8.6.2                 insert $S_2$ in nextU
2.3.8.7               }
2.3.8.8               remove the error
2.3.9             }
2.4           }
2.5           U = nextU
3.        }
```

Figure 3.3: CESIM error simulation algorithm for conditional errors

appearing in the condition of a conditional error are not part of the transitive combinational fanout of the basic error. Therefore, the activation conditions can be evaluated using the values computed in step 2.3.2. This partitions S_1 into a subset A of the active conditional errors, and a subset D of dormant (not active) errors (step 2.3.3).

If any outputs computed in 2.3.2 differ from those of the good circuit (step 2.2) all errors in D are detected and can be dropped. Otherwise, we record the erroneous next state corresponding to D, and insert D into the *nextU*, the set of undetected errors for the next vector.

For each set S_2 of PSBE-equivalent errors in A, we inject the *basic* error corresponding to S_2, apply the erroneous present state corresponding to S_2, and simulate the erroneous circuit. If any outputs differ from those in the good circuit, all errors in S_2 are dropped. Otherwise, we record the erroneous next state for S_2, and insert S_2 into *nextU*.

Example. Figure 3.4 illustrates CESIM. Consider sets of conditional errors derived from three basic errors e_1, e_2, and e_3. Initially, the corresponding erroneous machines are all in the same present state, namely the unknown state s_0^0. The initial PS-partition has a single class, which is further partitioned with respect to PSBE-equivalence. First, the error-free machine is simulated for the first vector; the next state is s_0^1. This allows us to separate those conditional errors that are active (shaded in the figure) for the first vector from those that are not. For the dormant errors no further work is required: none of them is detected, and the next state of the corresponding erroneous machines is s_0^1. For each PSBE class that contains active conditional errors, the corresponding *basic* error is injected and the erroneous circuit is simulated for the current vector. In the example, none of these errors is detected, and the next states s_1^1 and s_2^1 are distinct. This process is repeated for the next vector. In the example, the active errors in PSBE class (s_2^1, e_2) are detected by the second vector; all other errors remain undetected. Note that there is a one-to-one correspondence between a single transition in the state transition diagram in Figure 3.4 and a circuit simulation step in the algorithm (steps 2.2, 2.3.2, or 2.3.8.3 in Figure 3.3).

Analysis. CESIM minimizes the overall computational cost by exploiting PS- and PSBE-equivalence of conditional errors. We now analyze the algorithm's complexity. The two major components of the cost of one iteration of the top-level loop (step 2) are the simulation cost of steps 2.2, 2.3.2, and 2.3.8.3, and the partition cost of step 2.3.3. The partition cost is proportional to the number of conditional errors for which we have to check activation condition 1, which is typically a small fraction of the total number of conditional errors. The event-driven simulator is called as many times as there are PSBE partition classes on all sets A; this is a fraction of the number of PSBE partition classes of U. In summary, the cost of one iteration has one component with complexity sublinear in the size of the error list (partition cost), and a second component proportional to the size of

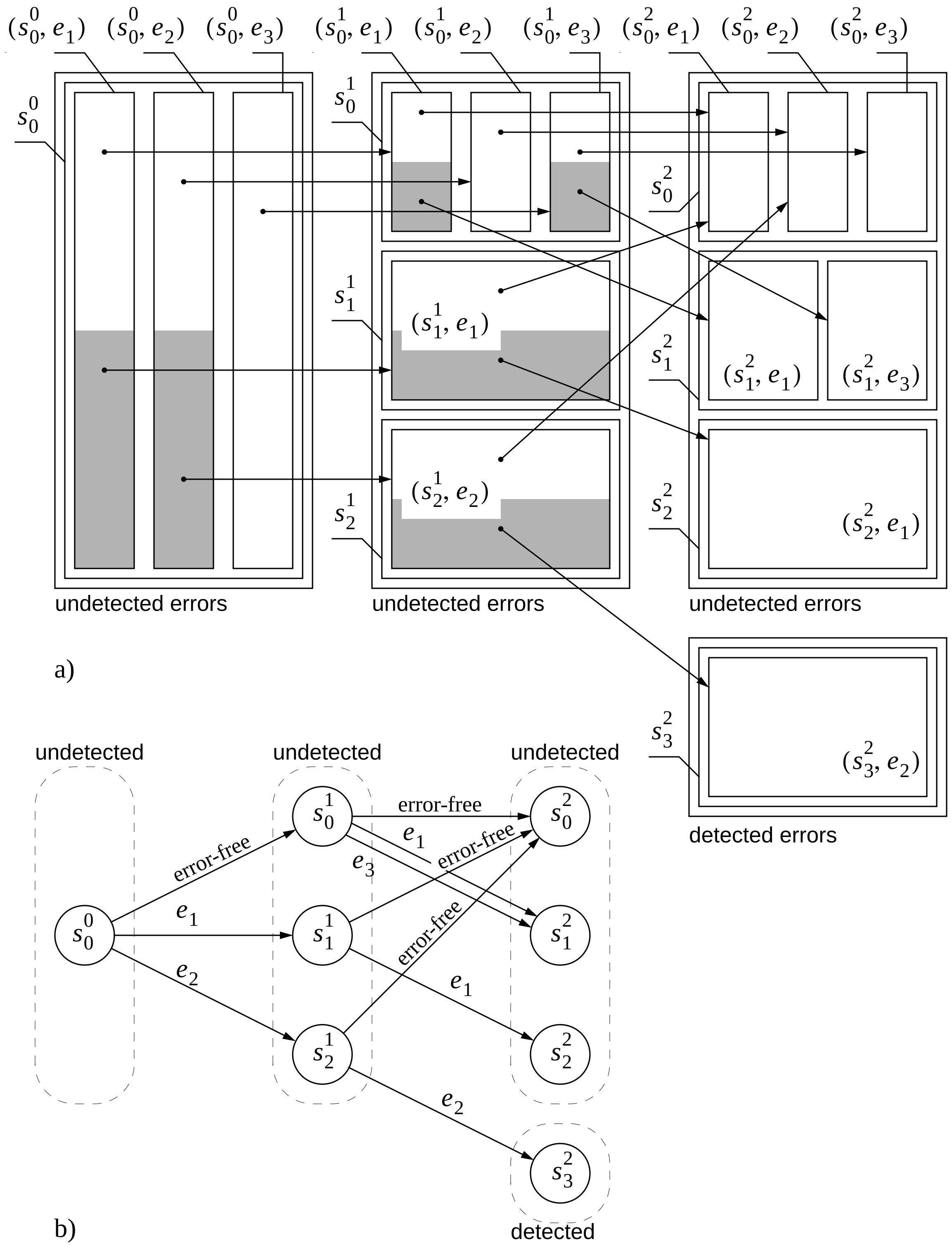


Figure 3.4: Example execution of CESIM for a 3-vector test sequence: a) PS- and PSBE-partitions of errors, b) corresponding state transitions

the circuit and the product of the number of basic errors and the number of distinct states (simulation cost). In our experiments, we observed that 90% of the execution time is due to partitioning, while only 10% is due to simulation. The algorithm requires maintaining both partitions (PS and PSBE) on the set of undetected errors. All partitions are implemented using hash tables, which allow for constant time insertions of error sets.

Initially, all errors are undetected and the corresponding erroneous machines all start from an unknown present state. Hence all errors are PS-equivalent initially, and all errors derived from the same basic error are PSBE-equivalent. In the partition step (2.3.3), the number of error sets (PSBE equivalence classes) may increase. The worst case occurs when 1) neither A nor D is empty, 2) neither of them is detected, and 3) the next states generated in steps 2.3.8.3 and 2.3.2 are all distinct. For this case, the number of error sets can double in a single iteration of step 2, leading to an exponential growth in the number of vectors. However, the total number of PSBE-equivalence classes can never exceed the total number of individual conditional errors we started with. Our experimental results (see below) show that, in practice, the number of error sets remains fairly constant.

Optimizations. As in PROOFS we take advantage of the word-level parallelism of the host computer; hence multiple iterations of 2.3.2 and of 2.3.8.3 are executed in parallel. To further reduce execution time, static dominators [Nier91a] could be used to identify redundant errors during a preprocessing step.

Experiments. We used the ISCAS’89 benchmarks to evaluate the performance of CESIM. First, we generated test sequences for SSL faults using HITEC [Nier91b, Nier91a]. We then error-simulated these test sequences using CESIM for CSSL0 and CSSL1 errors. The error list for CSSL0 errors is identical to the collapsed SSL fault list. The CSSL1 error list was constructed as follows. For each CSSL0 error, we considered a maximum of 500 lines to derive CSSL1 errors. The smaller circuits have fewer than 500 lines, so every line in the circuit is considered as condition line. This leads to a maximum of 1000 CSSL1 errors per CSSL0 error. However, some CSSL1 errors are rejected because their condition is part of the transitive fanout of the error site. A more detailed description of the experiments is given in Appendix B.

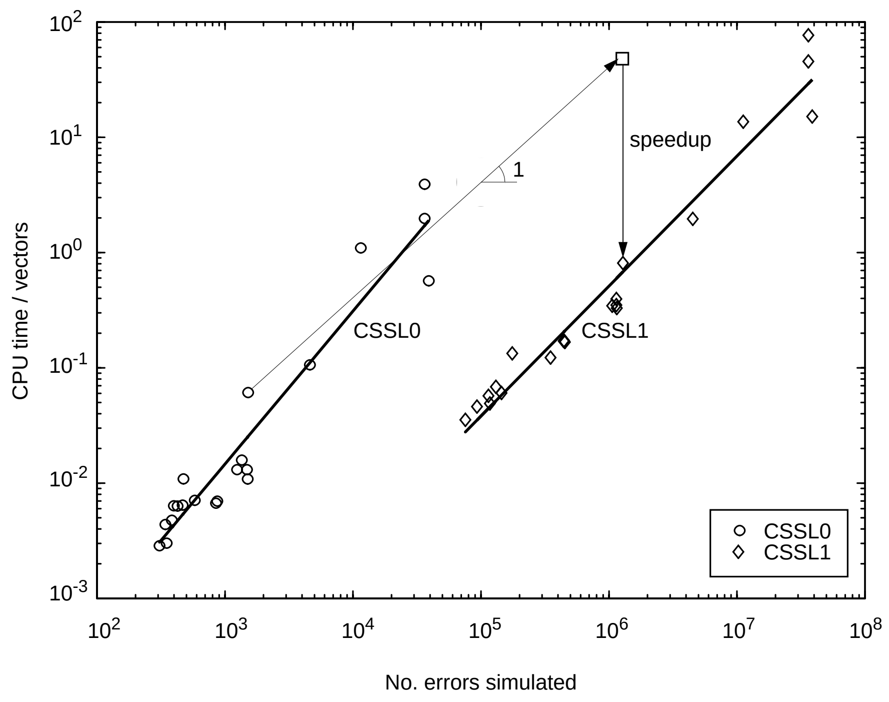


Figure 3.5: Run-time analysis of CESIM on the ISCAS'89 benchmarks

The efficiency of CESIM can best be seen by plotting the CPU time per test vector versus the total number of errors simulated for each benchmark, as in Figure 3.5. There are two sets of data: the first concerns CSSL0 error simulation, the other CSSL1 error simulation. The plot also shows a least square fit (linear regression) for each data set. The execution time of CSSL0 error simulation is dominated by event-driven simulation of faulty circuits. However, when simulating the CSSL1 errors, checking whether the condition of each CSSL1 error holds dominates the execution time. Least-square analysis shows that the CPU time per test vector is proportional to the number of CSSL0 errors to the power 1.33. This superlinear behavior reflects the fact that those data with a larger number of CSSL0 errors correspond to larger circuits, and hence the execution time of each event-driven simulation increases. For the CSSL1 execution time we find that the CPU time per test vector is proportional to the number of CSSL1 errors to the power 1.13. This near-linear behavior is because checking if CSSL1 errors are active, which is independent of the size of the circuit, dominates the execution time.

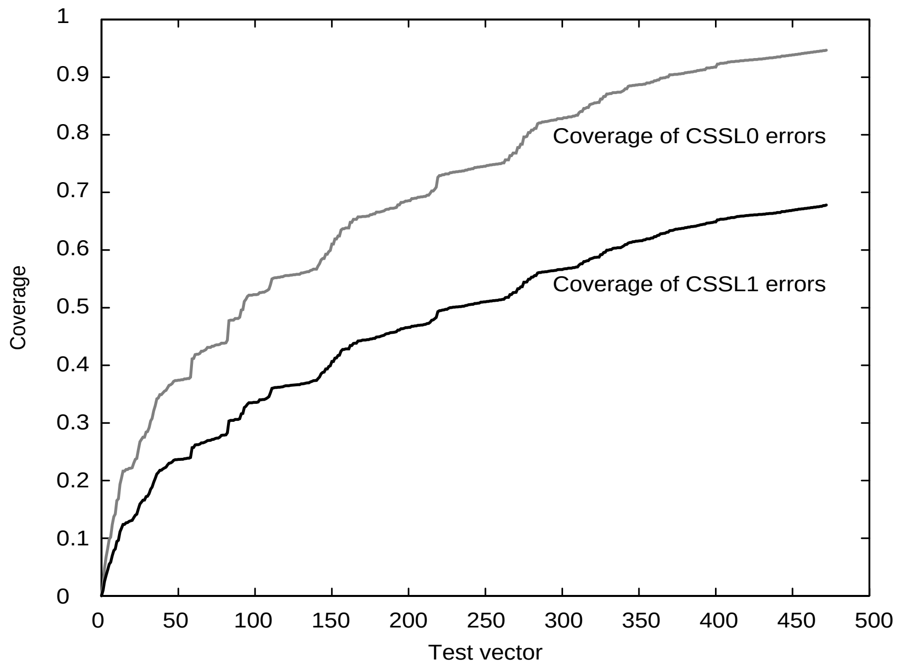


Figure 3.6: Coverage of CSSL0 and CSSL1 errors on s1238 by a CSSL0 test set generated by HITEC

Figure 3.5 also allows us to compare CESIM with the straightforward extension of PROOFS for CSSL1 error simulation, which we will refer to here as CPROOFS. CPROOFS treats CSSL1 errors the same way CESIM treats CSSL0 errors. The increase in execution time of CPROOFS for CSSL1 errors compared to the execution time of PROOFS for CSSL0 errors is therefore proportional to the ratio of the number of CSSL1 errors to the number of CSSL0 errors. The figure shows the execution time of CPROOFS for only one benchmark (s1423); the speedup for that circuit is 64. We can see that the speedup is roughly equal to the vertical distance between the linear regression lines for CSSL0 and CSSL1 datasets. We conclude that CESIM outperforms the CPROOFS by a wide margin.

We further analyze the behavior of CESIM for a representative circuit, s1238. This circuit has 14 inputs, 14 outputs, 18 D-type flip-flops, 80 inverters and 428 gates. Figure 3.6 shows the error coverage as a function of the number of test vectors applied. The ratio of coverage of CSSL1 errors to coverage of CSSL0 errors varies between 0.49 and 0.72.

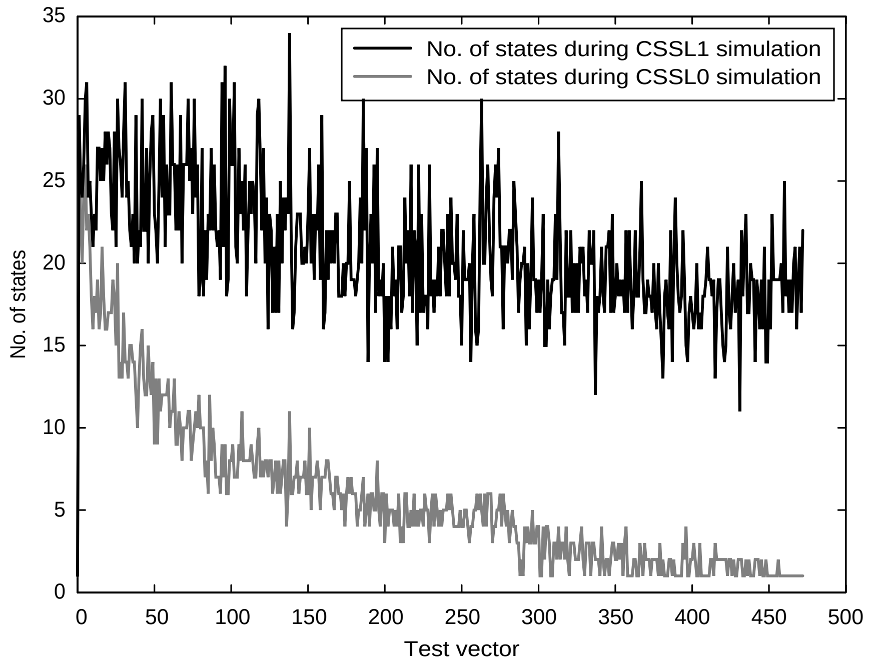


Figure 3.7: Error simulation on s1238 with CSSL0 and CSSL1:
number of distinct states

Figure 3.7 shows the number of distinct states as a function of the number of test vectors applied. For CSSL0 error simulation, the number of states rapidly drops; after vector 300 there are at most five distinct states among the present states of the remaining undetected erroneous machines. For CSSL1 error simulation, we observe that the number of states hovers around 20 but never becomes larger than 35 (about twice the number of flip-flops in the circuit).

Figure 3.8 details the number of error sets occurring during the execution of CESIM. We show both the total number of error sets, and the number of error sets in use. Both are normalized with respect to the total number of errors. The number of error sets in use is the number of PSBE-equivalence classes of the set of undetected errors U in loop 2.3 of Figure 3.3. The total number of error sets is the number of error sets in use plus the number of errors sets detected by previous vectors (those error sets are dropped in steps 2.3.4.1 and 2.3.8.4.1 of Figure 3.3). For CSSL0 simulation, the total number of error sets remains constant at the number of errors, whereas the number of error sets in use drops as coverage increases. For CSSL1 simulation we observe that the total number of error sets

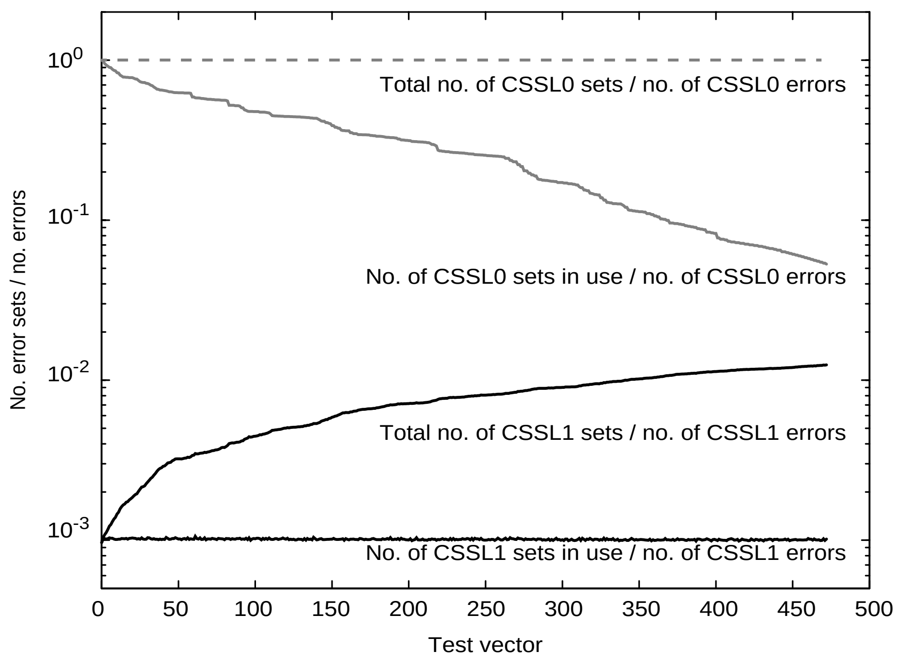


Figure 3.8: Number of error sets during error simulation on s1238
with CSSL0 and CSSL1 errors

increases steadily, as coverage increases. However, the number of error sets in use remains fairly constant and hovers around the total number of basic errors, which is about 1000 times smaller than the total number of errors.

3.6 Analytical coverage evaluation of CSSL1

The first and foremost requirement for design error models is that complete test sets for the modeled errors should also provide very high coverage of actual design errors. In this section we analyze the detection of basic design errors by complete test sets for CSSL1 errors in gate-level circuits. Let D_0 be a gate-level circuit; construct D_1 by injecting a single error e_1 into D_0, where e_1 is an instance of error model M_1. Let T_0 and T_1 be test sets that provide complete coverage of all detectable CSSL0 and CSSL1 errors, respectively, in D_1. We analyze the coverage provided by test sets T_0 and T_1 with respect to the error models M_1 proposed in [AA95]. In particular, we are interested in those error classes covered by T_1, but not by T_0.

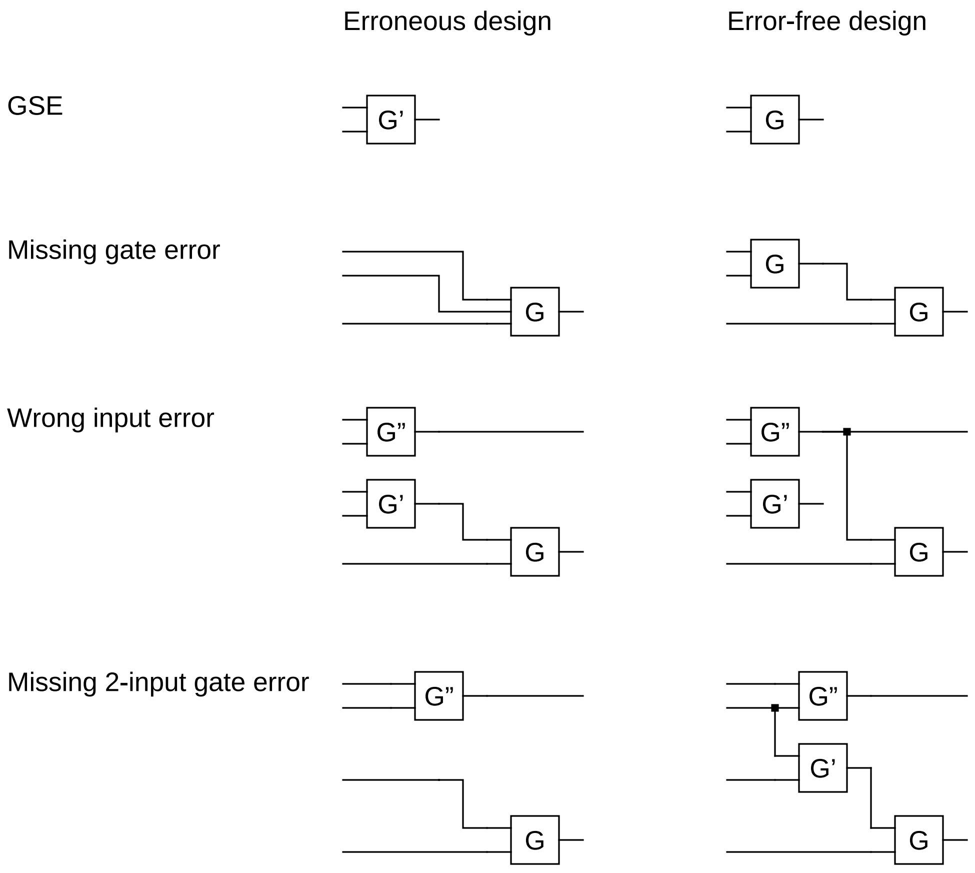


Figure 3.9: Some basic error types [AA95]

We use the notation introduced in [AA95], and refer the reader to that paper for further details of the error models. Let $y = G(x_1,\ldots,x_n)$ be a gate in the error-free circuit. A gate substitution error G/G' occurs if a gate G is erroneously replaced with a gate G' that has the same number of inputs but is of a different type. The set of all 2^n input vectors of an n-input gate is divided into disjoint subsets $V_0, V_1,\ldots,V_n$, where V_k contains all input vectors with exactly k 1s in their binary representation, $0 \leq k \leq n$. The disjoint sets V_{null}, V_{all}, V_{odd}, and V_{even} are defined as follows:

$$V_{null} = V_0;\ V_{all} = V_n;\ V_{odd} = \bigcup_{i = odd \wedge i \neq n} V_i;\ V_{even} = \bigcup_{i = even \wedge i \neq 0 \wedge i \neq n} V_i .$$

The sets V_{null}, V_{all}, V_{odd}, and V_{even} are called the *characterizing sets* or *C-sets* of G.

Consider the following sets of CSSL0 and CSSL1 errors in the erroneous circuit:

- $E_1 = \{(1, y / 0)\}$
- $E_2 = \{(1, y / 1)\}$
- $E_3 = \{(1, x_i / 0) \mid i=1\ldots n\}$
- $E_4 = \{(1, x_i / 1) \mid i=1\ldots n\}$
- $E_5 = \{(x_i = 0, y / 0) \mid i=1\ldots n\}$
- $E_6 = \{(x_i = 1, y / 0) \mid i=1\ldots n\}$
- $E_7 = \{(x_i = 0, y / 1) \mid i=1\ldots n\}$
- $E_8 = \{(x_i = 1, y / 1) \mid i=1\ldots n\}$

Let T_i, where $i = 1\ldots8$, be a complete test set for E_i.

Single-input gate substitution errors. Gate substitution errors (*GSE'*s) involving buffers or inverters are called *single-input GSE'*s. A necessary and sufficient condition to detect *G*/*G'* is to sensitize *y*. Any test in T_1 or T_2 must sensitize *y* and hence detects *G*/*G'*. If both T_1 and T_2 are empty, *y* is not sensitizable and hence *G*/*G'* is undetectable.

Multiple-input GSE's. Consider a *multiple-input GSE* (*MIGSE*) *G*,AND, i.e., gate *G* is erroneously replaced by an AND gate. To detect (*G*,AND), we have to identify the AND gate in the erroneous circuit. Note that:

- any test in T_1 or T_3 or T_6 excites *G'* for V_{all}
- any test in T_2 or T_7 excites *G'* for $V_{null} \cup V_{even} \cup V_{odd}$
- any test in T_4 excites *G'* for V_{n-1}
- any test in T_8 excites *G'* for $V_{even} \cup V_{odd}$
- $T_5 = \varnothing$

Case 1: $T_4 \neq \varnothing$, $T_1 \neq \varnothing$. $T_1 \cup T_4$ uncovers all detectable *MIGSE's* with the exception of XNOR/AND (XOR/AND) for *n* even (odd). Detection of this last error requires exciting *G'* for $V_{null} \cup V_{even}$ ($V_{null} \cup V_{odd}$). None of the error sets considered can enforce this condition.

Case 2: $T_4 \neq \varnothing$, $T_1 = \varnothing$. T_4 uncovers all detectable *MIGSE's* with the exception of NOR/AND and XNOR/AND (XOR/AND) for *n* even (odd). Detection of these last two errors requires exciting *G'* for both V_{null} and V_{even} (V_{odd}). None of the error sets considered can enforce this condition.

Case 3: $T_4 = \emptyset$, $T_8 \neq \emptyset$, $T_1 \neq \emptyset$. $T_1 \cup T_8$ uncovers all detectable *MIGSEs* with the exception of XNOR/AND (XOR/AND) for n even (odd). $T_1 \cup T_2$ might fail to uncover OR/AND as well.
Case 4: $T_4 = \emptyset$, $T_8 \neq \emptyset$, $T_1 = \emptyset$. T_8 might fail to detect NOR/AND and either XNOR/AND or XOR/AND.
Case 5: $T_4 = \emptyset$, $T_8 = \emptyset$. *G'* cannot be excited for $V_{even} \cup V_{odd}$. $T_1 \cup T_2$ excites *G'* for each remaining C-set.

The analysis for *MIGSE's* where *G'*=NAND,OR,NOR,XOR,XNOR is similar to that presented above.

We conclude that the coverage of *MIGSEs* provided by complete test sets for CSSL1 errors is only marginally better (see case 3) than that provided by test complete test sets for CSSL0 errors.

Gate count errors. Two types of gate count errors are defined in [AA95]: extra-gate errors and missing gate errors. Extra-gate errors are shown to be detected by any complete test set for *GSE's*. Hence the same conclusions with respect to coverage by complete test sets apply to CSSL1 errors.

It is shown in [AA95] that detection of missing gate errors requires applying either V_2 or V_{n-2}. These sets cannot be enforced using a single CSSL1 error, and hence complete test sets for CSSL1 errors do not provide more coverage for missing gate errors than test sets for CSSL0 errors.

Input count errors. Extra input errors have been shown to be covered by any complete test set for CSSL0 errors [AA95]. The coverage for missing input errors was shown to be only partial. A missing input error occurs when an n-input gate ($n \geq 3$) is replaced by an ($n - 1$)-input gate with its $n - 1$ inputs connected to an arbitrary subset of the original n inputs. The error detection requirements of this type of error map exactly onto those of a CSSL1 error, except in the case of XOR or XNOR gates.

Wrong input error. A wrong input error occurs when a single gate input is connected to the wrong signal: in the error-free circuit, $y = G(x_1,\ldots,x_n)$, while in the erroneous circuit: $y = G(z, x_2,\ldots,x_n)$. A necessary and sufficient condition to detect this error is to sensitize z while z and x_1 have opposite values. This is equivalent to detecting either of the CSSL1 errors ($x_1 = 0$, z / 0) or ($x_1 = 1$, z / 1).

Missing 2-input gate error. This error occurs if the error-free circuit contains a gate $y = G(x_1, x_2)$ that is completely missing in the erroneous circuit and $y = x_1$. It can be shown that the error detection requirements for this error are equivalent to those of a CSSL1 error. For example if G=AND, the corresponding CSSL1 error is ($x_2 = 0$, y / 0). A complete test set for CSSL0 errors may fail to detect this error.

Conclusion. Complete test sets for CSSL1 errors also detect all wrong input errors, all missing input errors on gates that are not of type {XOR,XNOR}, and all missing 2-input gate errors; complete test sets for CSSL0 errors can fail to detect these errors. For the other error types, no increased coverage is guaranteed by complete test sets for CSSL1 errors.

3.7 Coverage evaluation using error simulation

From the analysis presented in the previous section, one could conclude that the class of design errors that is guaranteed to be detected by a complete test set for CSSL1 errors is very limited. In fact, most actual design errors do not fall in this class. However, as our analytical study tries to establish properties that hold for any design, its results are conservative. For a concrete design, complete test sets for CSSL1 errors may detect many more design errors than those reported in the previous section.

To compare the effectiveness of two design error models, we could take an unverified design, and generate test sets that are complete with respect to the two error models. The test set that uncovers more (and harder) design errors in a fixed amount of time is more effective. However, for such a comparison to be practical, fast and efficient high-level test generation tools for our error models appear to be necessary. Although this type of test generation is feasible, it has yet to be automated. Instead we consider test sets that were not specifically targeted, and compute their coverage of modeled design errors as well as of actual design errors.

Table 3.1: Characteristics of two modules of the DLX microprocessor implementation

Parameter	Module 1: `top`	Module2: `decode`
No. of lines of code	302	263
No. of CSSL0 errors	574	816
No. of CSSL1 errors	141,756	238,732
No. of restricted CSSL0 errors	178	82
No. of restricted CSSL1 errors	21,864	18,788
No. of detectable actual errors	8	16

In this section we present a set of experiments whose goal is to compare different design error models and investigate the relationship between coverage of modeled design errors and coverage of more complex actual errors.

The test vehicle for this study is the well-known DLX microprocessor [Henn90]. The particular DLX version considered is a student-written design that implements 44 instructions, has a five-stage pipeline and branch prediction logic. The design errors made by the student during the design process were systematically recorded. They were presented earlier in Chapter 2 (DLX1 in Table 2.1 and Table 2.4). Some characteristics of two of the modules of the design are shown in Table 3.1. Module `top` integrates the different pipeline stages and contains the forwarding logic. Module `decode` describes the decode stage of the pipeline. These modules are analyzed here because 75% of all actual errors were made within these two modules. A simplified block diagram of the design, indicating both modules, is shown in Figure 3.10.

For these experiments, we modified the original design description to allow us to automatically inject synthetic errors into the design. The modifications do not cause a significant overhead during simulation and do not require recompilation of the simulator when a new error is injected. On the other hand, this approach requires a simulation run for each error considered.

The error models considered in this study are the CSSL0 and CSSL1 models. Even for moderately sized modules under consideration, the number of CSSL1 errors is very large; for example, there are 141,756 CSSL1 errors in `top`. Given our error simulation approach, the number of errors needs to be reduced to make the experiment practical. A subset of the CSSL1 errors was selected by imposing the following constraints: 1) lines

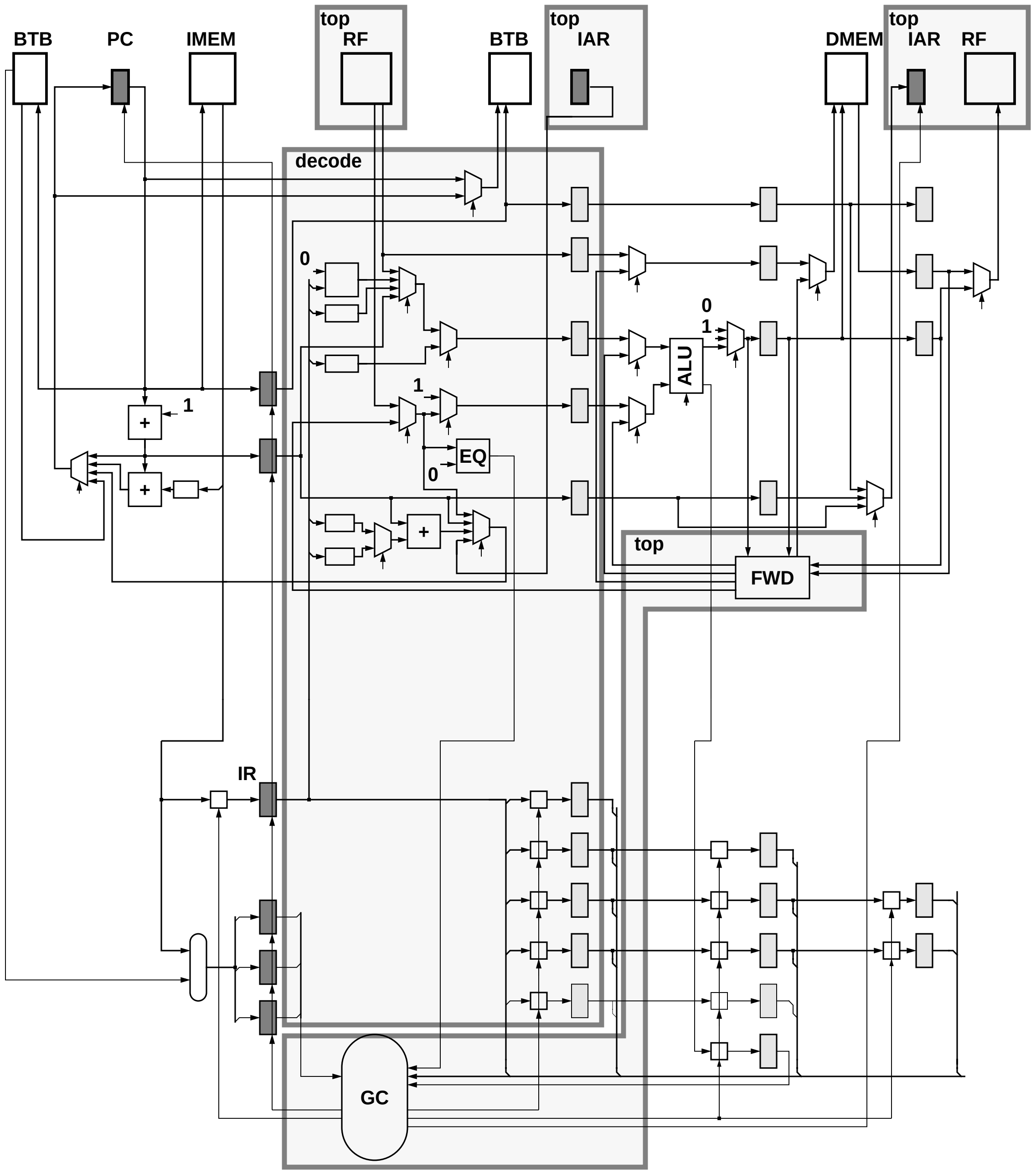


Figure 3.10: Simplified schematic of DLX implementation showing modules `decode` and `top`

considered in the condition are restricted to signals of bit-width 1, and 2) lines considered as error sites are restricted to signals with bit width > 1. *CE's* of this type are referred to as *restricted CE's*. This reduces the number of CSSL1 errors by about an order of magnitude. For example, there are 21,864 restricted CSSL1 errors in `top`. Error simulation for

Table 3.2: Coverage of synthetic and actual errors by biased random tests T0-T13

Parameter	Module 1: `top`	Module2: `decode`
CSSL0 errors	77%	64%
Restricted CSSL0 errors	86%	93%
Restricted CSSL1 errors	72%	69%
Actual errors	75%	69%

restricted CSSL1 errors in `top` and the test set described below took 34 hours on a HAL300 workstation; an average simulation speed of 140 simulated clock cycles per CPU second was observed.

We developed a tool for generating random but valid assembly programs for the DLX instruction set architecture. The tool is biased towards generating 'interesting' cases, such as data dependencies, control dependencies, exceptions, and boundary data values. To satisfy the requirement that the programs generated be valid, some structure is imposed on the programs which limits their variety. A number of parameters allow the user to vary the size and structure of the programs. We constructed a sequence of test programs increasing in size and complexity. The combined execution length of the programs was 3445 cycles. We then computed the coverage for synthetic and actual errors of the test set.

The results of error simulation are summarized in Table 3.2. A significant fraction of CSSL0 errors is not covered by the test set. The coverage of restricted CSSL1 errors is, as expected, lower than that of restricted CSSL0 errors. Figure 3.11 shows the coverage of three error sets as a function of the number of test programs applied. The first (and shortest) test program uncovers many easy-to-detect errors. Coverage increases slowly as more test programs are applied. The profiles for both synthetic error models appear similar. Both error models reveal areas insufficiently exercised by the test programs. For instance, none of the test programs contains instructions with illegal opcodes.

Our analytical evaluation of the CSSL1 error model shows that complete test sets for CSSL1 errors also detect a number basic errors for which complete test sets for CSSL0 errors provide only partial coverage. However, as the CSSL1 error model defines a number of error instances quadratic in the size of the circuit, further evidence is needed to demonstrate the model's merits for design verification. We therefore have conducted error

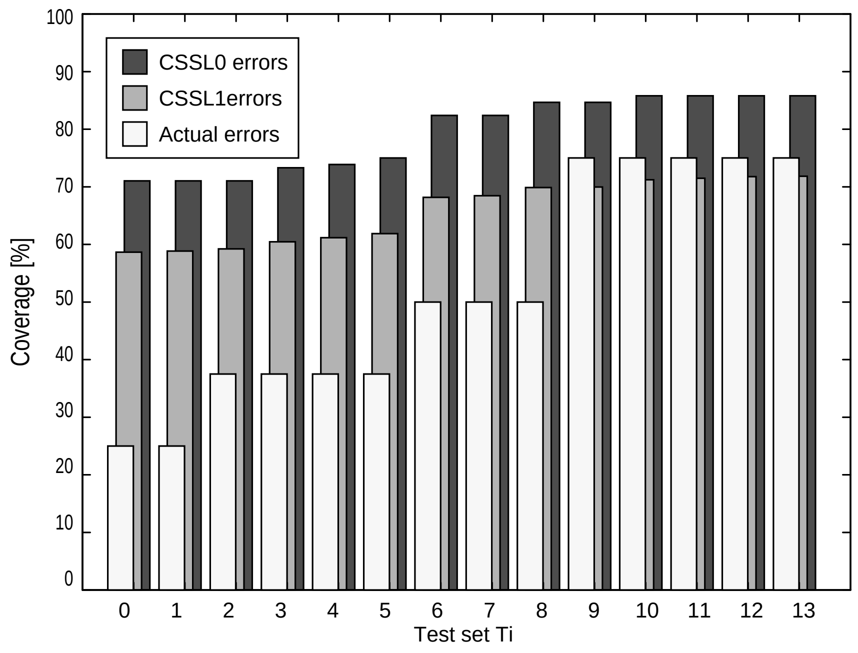


Figure 3.11: Coverage of restricted CSSL0, restricted CSSL1, and actual errors by 14 biased random test sets T0-T13 for `top`

simulation experiments in which the coverage of CSSL0, CSSL1, and actual errors by biased random test sets is computed. The correlation between coverage of SSL errors and coverage of actual errors is very similar to that between coverage of CSSL1 errors and coverage of actual errors. Hence this study does not provide any grounds to justify the use of the CSSL1 error model instead of the SSL model.

3.8 Coverage evaluation by analysis of actual errors

To show the effectiveness of a verification methodology, one can apply it and a competing methodology to an unverified design. The methodology that uncovers more or harder design errors in a fixed amount of time is more effective. However, for such a comparison to be practical, fast and efficient high-level test generation tools for the error models are necessary. We have discussed such test generation tools in Section 3.4 but they have to be automated. We therefore designed a controlled experiment that approximates the conditions of the discussed experiment, while avoiding the need to automate test generation. The experiment evaluates 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 3.12. As design error models are used to guide test generation, the effectiveness of our design verification approach is closely related to the synthetic error models used.

To evaluate our methodology, a circuit was chosen for which design errors were systematically recorded during its design. Let D_0 be the final, presumably correct, design. From the CVS revision database, the actual errors were extracted and converted so that they can be injected into the final design D_0. In the evaluation phase, the design was 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: given D_1, can the proposed methodology produce a test that determines D_1 to be erroneous? This is achieved by examining the actual error in D_1 to determine if a modeled design error exists that is *dominated* by the actual error. Let D_2 be the design constructed by injecting the dominated modeled error in D_1, and let M be the error model that defines the dominated modeled error. Such a dominated modeled error has the property that any test that detects the modeled error in D_2 also detects the actual error in D_1. Consequently, if we were to generate a complete test set for every error defined on D_1 by error model 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 case with 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 easy if given D_0 instead of D_1.

The erroneous design D_1 considered in this experiment is somewhat artificial. In reality the 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

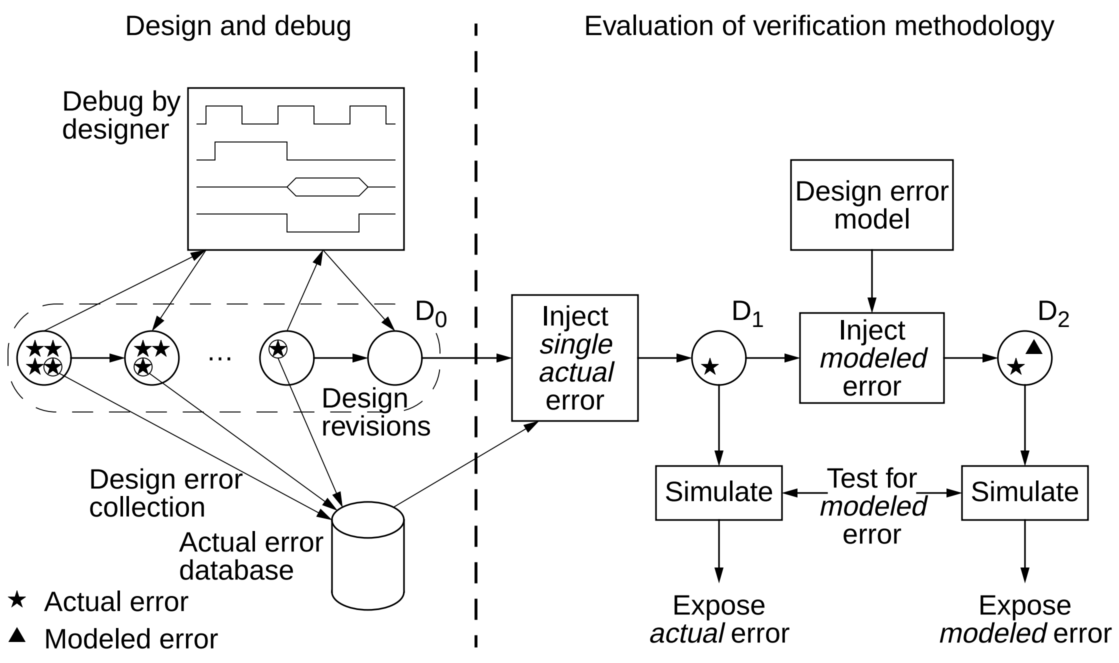


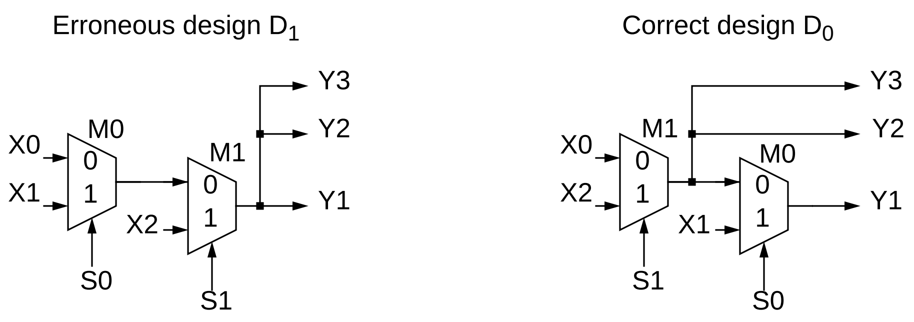
Figure 3.12: Experiment to evaluate the proposed design verification methodology

consider a single error or multiple design errors at a time. Furthermore, our results are independent of the order in which one applies the generated test sequences.

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

A pipelined microprocessor. Our first design case study considers the well-known DLX microprocessor [Henn90]. 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, registerfiles, etc. The design errors committed by the student during the design process were systematically recorded using our error collection system.

For each actual design error we painstakingly derived the requirements to detect it. Error detection was determined with respect to one of two reference models (specifications). The first reference model is an ISA model that is not cycle-accurate: only



S1,S0	Y1	Y2,Y3
0 0	X0	X0
0 1	X1	X1
1 0	X2	X2
1 1	X2	X2

S1,S0	Y1	Y2,Y3
0 0	X0	X0
0 1	X1	X0
1 0	X2	X2
1 1	X1	X2

Figure 3.13: Example of an actual design error in our DLX implementation

the changes made to the ISA-visible part of the machine state, that is, to the register file and memory, can be compared. The second reference model contains information about the microarchitecture of the implementation and gives a cycle-accurate view of the ISA-visible part of the machine state (including the program counter). We determined for each actual error whether it is detectable with respect to each reference model. Errors undetectable with respect to both reference models may arise for the following two reasons: (1) Designers sometimes make changes to don't care features, and log them as errors. This happens when designers have a more detailed specifications (design intent) in mind than that actually specified. (2) Inaccuracies can occur when fixing an error requires multiple revisions.

We analyzed the detection requirements of each actual error and constructed a modeled error dominated by the actual error, wherever possible. One actual error involved multiple signal source errors, and is shown in Figure 3.13. Also shown are the truth tables for the immediately affected signals; differing entries are shaded. Error detection via fanout $Y1$ requires setting $S1 = 1$, $S0 = 1$, $(X1 \neq X2)$, and sensitizing $Y1$. However, the combination $(S1 = 1, S0 = 1)$ is not achievable and thus error detection via $Y1$ is not possible. Detection

Table 3.3: Actual design errors and the corresponding dominated modeled errors for DLX

Actual errors				Corresponding dominated modeled errors						
Category	ISA	ISAb	Total	INV	SSL	BSE	CSSL1	CBOE	CSSL2	Unknown
Missing instance	8	2	14	0	2	0	6	1	0	1
Wrong signal source	9	2	11	1	4	5	1	0	0	0
Complex	3	0	3	0	3	0	0	0	0	0
Inversion	1	2	3	3	0	0	0	0	0	0
Missing input	1	0	3	0	0	0	1	0	0	0
Unconnected input	3	0	3	3	0	0	0	0	0	0
Missing minterm	1	0	1	0	0	0	0	0	1	0
Extra input	1	0	1	0	1	0	0	0	0	0
Total	27	6	39	7	10	5	8	1	1	1

via $Y2$ or $Y3$ requires setting $S1 = 0$, $S0 = 1$, $(X0 \neq X1)$, and sensitizing $Y2$ or $Y3$. However, $S0 = 1$ blocks error propagation via $Y2$ further downstream. Hence, the error detection requirements are: $S1 = 0$, $S0 = 1$, $(X0 \neq X1)$, and sensitizing $Y3$.

Now consider the modeled error $E_1 = S0$ s-a-0 in D_1. Activation of E_1 in $D1$ requires $S1 = 0$, $S0 = 1$. Propagation requires $(X0 \neq X1)$, and sensitizing $Y1$, $Y2$ or $Y3$. As mentioned before, $S0 = 1$ blocks error propagation via $Y2$. But as E_1 can be exposed via $Y1$ without sensitizing $Y3$, E_1 is not dominated by the given actual error. To ensure detection of the actual error, we can condition $S0$ s-a-0 such that sensitization of $Y3$ is required. The design contains a signal *jump_to_reg_instr* that, when set to 1, blocks sensitization of $Y1$, but allows sensitization of $Y3$. Hence the CSSL1 error (*jump_to_reg_instr* = 1, $S0$ s-a-0) is dominated by the actual error.

The results of this experiment are summarized in Table 3.3. A total of 39 design errors were recorded by the designer. The actual design errors are grouped by category. ‘Missing instance’ and ‘wrong signal source’ errors account for more than half of all errors. The column headed ‘ISA’ indicates how many errors are detectable with respect to the ISA-model; ‘ISAb’ lists the number of errors only detectable with respect to the micro-architectural reference model. The sum of ‘ISA’ and ‘ISAb’ does not always equal ‘Total’; the difference corresponds to actual errors that are not detectable with respect to either reference model. The remaining columns give the type of the simplest dominated modeled error corresponding to each actual error. Among the ten detectable missing instance errors, two dominate an SSL error, six dominate a CSSL1 error, and one dominates a CBOE. For the remaining one, we were not able to find a sufficiently simple dominated modeled error.

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. From Table 3.3 it can be concluded that for this experiment, 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.

A simple microprocessor. Al-Asaad [VC98] performed a similar experiment for the Little Computer 2 (LC-2) [Post96a], a small microprocessor of conventional design used for teaching purposes at the University of Michigan. It has a representative set of 16 instructions which, are a subset of the instruction sets of most current microprocessors. To serve as a test case for design verification, behavioral and RTL synthesizable Verilog descriptions for the LC-2 were designed. The behavioral model of the LC-2 consists of 235 lines of behavioral Verilog code. The RTL design (implementation) consists of a datapath module described as an interconnection of library modules and a few custom modules, and a control module described as an FSM with five states. The implementation comprises 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 tools. The design errors made during the design of the LC-2 were systematically recorded using our error collection system (Chapter 2).

The actual design errors in both the behavioral and RTL designs of the LC-2 were analyzed, and the results are summarized in Table 3.4. A total of 20 design errors were made during the design, of which four are easily detected by the Verilog simulator and/or logic synthesis tools, and two are undetectable. The actual design errors are grouped by category. The columns in the table give the type of the simplest dominated modeled error corresponding to each actual error. For example, among the four remaining wrong-signal-source errors, two dominate an SSL error and two dominate a BSE error.

Table 3.4: Actual design errors and the corresponding dominated modeled errors for LC2

Category	Actual errors			Corresponding dominated modeled errors			
	Total	Easily detected	Unde-tectable	SSL	BSE	CSSL1	Un-known
Wrong signal source	4	0	0	2	2	0	0
Expression error	4	0	0	2	0	1	1
Wrong bus width	3	3	0	0	0	0	0
Missing assignment	3	0	0	0	0	2	1
Wrong constant	2	0	0	2	0	0	0
Unused signal	2	0	2	0	0	0	0
Wrong module	1	0	0	1	0	0	0
Always statement	1	1	0	0	0	0	0
Total	20	4	2	7	2	3	2

We can infer from Table 3.4 that most errors are detected by tests for SSL errors or BSEs. About 75% of the actual errors in the LC-2 design can be detected after simulation with tests for SSL errors and BSEs. The coverage increases to 90% if tests for CSSL1 are added.

3.9 Conclusions

Unlike the case with other simulation-based design validation methodologies, we use design error models to direct test generation. We have identified four key requirements that error models should satisfy to be useful for design validation: 1) complete test sets for the modeled errors should also provide very high coverage of actual errors, 2) the error models should be amenable to automated test generation 3) the error models should be amenable to error simulation, and 4) the number of modeled errors should be sufficiently small.

Based on the error data presented in the previous chapter, we have proposed three classes of design error models: basic, extended and conditional design error models. We have analyzed how well each error model satisfies the four requirements. The extended error models were found too difficult for automated test generation, and have been discarded on that ground. Test generation for the other two classes of models was found similar to test generation for SSL errors. We have developed an error simulation algorithm for conditional errors called CESIM. Our experimental results show that CESIM outperforms a state-of-the-art fault simulation algorithm by a wide margin (a factor 34 on average).

We conducted three studies to assess how well the error models meet requirement 1. An analytical study of CSSL1 errors shows that complete test sets for CSSL1 errors do provide higher coverage for common design errors in gate-level designs over test sets that are complete for SSL errors. A second study used error simulation, and compared the coverage of SSL, CSSL1 and actual errors on a microprocessor design. The correlation between coverage of SSL errors and coverage of actual errors was found to be very similar to that between coverage of CSSL1 errors and coverage of actual errors.

A final study analyzed actual errors in microprocessor designs, and investigated whether our methodology can detect such errors. The results indicate that complete test sets for synthetic errors provide a very high coverage of actual errors (97% for one design and 90% for another). The results also show the conditional error models are especially useful for detecting actual errors that involve missing logic, which are often difficult to detect using basic errors only.

Table 3.5 summarizes our findings. Each error model is graded with respect to the four requirements relative to the SSL model. The SSL model scores the highest on requirements 2 and 3, since standard ATPG tools use the SSL model. The scores of the other models reflect the effort required to either modify the design and to use standard tools, or to modify the tools to handle the new models. Our methodology supports incremental design validation: First, generate tests for SSL errors. Then generate tests for other basic error types such as MSE. Finally, generate tests for conditional errors.

Our studies suggest that the CSSL1 model is a good candidate to improve on the coverage provided by a complete test sets for SSL errors. The CSSL1 model provides a natural extension of the SSL model; standard ATPG algorithms can easily be modified for CSSL1; we have demonstrated efficient error simulation with CSSL1 errors. The number of CSSL1 errors is quadratic in the size of the circuit. Although, the number of CSSL1

Table 3.5: Comparison of practical design error models[a]

Error model		Req. 1: Coverage	Req. 2: Test generation	Req. 3: Error simulation	Req. 4: No. of instances
Basic	SSL	+	+	+	$O(N)$
	MSE	+	--	--	$O(N)$
	BOE	-	---	---	$O(N)$
	BSE	++	---	---	$O(N^2)$
	BDE	+	--	--	$O(B.D^2)$
Conditional	CSSL1	++	-	-	$O(2^2N^2)$
	CBOE	+	---	---	$O(2N^2)$
	CSSL2	+++	-	-	$O(2^3N^3)$

a. *N* is the number of signals in the circuit; *D* is the average number of drivers on a tristate bus.

errors for a flattened design hierarchy is extremely large, the design hierarchy provides a natural means to reduce the number of CSSL1 errors. If the stuck line and the condition line that constitute a CSSL1 error are restricted to signals belonging to the same hierarchical module the number of CSSL1 errors to be targeted during test generation is typically small enough for practical use of CSSL1 errors.

CHAPTER 4
High-level test generation for design verification of pipelined microprocessors

Our design validation methodology uses design error models to direct the generation of verification tests. In the previous chapter we presented error models suitable for design verification and showed that test generation for these errors is similar to test generation for SSL faults.

The area of automatic test pattern generation (ATPG) for combinational circuits is very mature and several commercial ATPG tools are available that are able to handle very large designs. ATPG for sequential circuits is a much harder problem [Chen96, Marc96], but design for testability (DFT) techniques, such as *scan* [Abra90], greatly reduce this complexity. In *full scan* design, every register is replaced by a scan register and the registers are linked in a chain, thereby making every register observable and controllable. This effectively reduces the test generation problem to one for combinational circuits.

Unfortunately, DFT techniques do not apply to design verification. Unlike in physical fault testing, the error-free design is unknown. This may seem contradictory since in logic simulation every signal in the design can be examined. However, a signal can only be considered observable if we can easily discern incorrect from correct values. This means that only the signals that are part of the specification are observable. Typically these include all primary inputs and primary output, as well as a small subset of the state registers. The same observation applies to controllability. Logic simulators typically allow us to override the logic driving any signal, and ‘force’ the signal to an arbitrary value. This might give the illusion that every signal is controllable, but since most of these signals do not appear in the specification, we cannot easily bring the specification in a state that corresponds to that of the implementation. Hence we cannot easily verify the simulation outcome if signals have been forced to arbitrary values. Therefore test vectors can only specify the values of the primary inputs, and the initial state of the subset of implementation registers that is also part of the specification.

We conclude that the test generation problem in design validation has to be solved for a sequential circuit. Although recently significant advances have been made in the area of gate-level sequential ATPG, designs of the size of microprocessors are well beyond the capabilities of present methods. Most previous work in this area attempts to address test generation for general gate-level sequential circuits. An alternative direction of research is to restrict the class of circuits targeted. Domain-specific information about these designs can then be exploited so that larger circuits can be handled than the general methods are capable of. We chose a class of pipelined microprocessors as our design domain. In this chapter, we develop a high-level test generation method for pipelined microprocessors. We first describe a model that captures high-level knowledge about microprocessor structure. We then develop a high-level test generation algorithm that uses this information. The main features of the method are 1) the integration of high-level treatment of the datapath with fully detailed treatment of the controller, 2) its “pipeframe” based iterative organization, 3) the separation of path and value selection.

We review relevant previous work in Section 4.1. Our high-level model for pipelined processors is presented in Section 4.2. The iterative organization of the proposed high-level test generation algorithm is described in Section 4.3. The overall test generation algorithm is described in Section 4.4. Sections 4.5, 4.6, and 4.7 describe the three main components of the algorithm. We present experimental results in Section 4.8, and give some concluding remarks in Section 4.9.

4.1 Related work

Test generation for gate-level sequential circuits

Typical test generators for sequential circuits [Abra90, Chen96] iteratively apply a test generation algorithm for combinational circuits by using a gate-level iterative logic array

(ILA) model of the circuit. Kelsey et al. [Kels93] describe a test generation algorithm for sequential circuits that does not follow the iterative structure of the ILA. For a given fault, an estimate of the test sequence length is computed, and the circuit is unrolled over that many cycles. The PODEM algorithm [Chen96, Goel81] is applied to the resultant circuit, which is treated as a single combinational circuit. Because this approach only makes decisions on primary inputs and only propagates information forward, it can result in a more efficient search. On the other hand, the search process is performed on a much larger and deeper circuit than in conventional approaches, hence its efficiency depends critically on the backtracing heuristics used.

Ghosh et al. [Ghos91] decompose the test generation problem into three subproblems: combinational test generation, fault-free state justification, and fault-free state differentiation. By performing state justification and differentiation in the fault-free machine, their algorithm can re-use a significant amount of computation.

High-level test generation

Lee and Patel describe a high-level test generation method for microprocessors in [Lee92a, Lee94]. They model a processor as an interconnection of high-level modules. During a preprocessing step they symbolically simulate each instruction of the instruction set to derive the control behaviors corresponding to each instruction. These control behaviors can be seen as ‘configurations’ of the processor (datapath) over a number of clock cycles (as many as the corresponding instruction takes to execute). The proposed test generation method has two phases: path selection and value selection. During path selection, a sequence of instructions is assembled so that a set of paths is sensitized to activate the targeted error and propagate its effect. These paths may span multiple clock cycles and may require multiple instructions. The task of computing the concrete values that need to be applied to the primary inputs is delegated to the value selection phase. This second problem can be formulated as a system of non-linear equations. The variables in this problem correspond to the signals in the datapath (in multiple timeframes). The equations correspond to the modules and interconnections of the datapath. They express the relationship between the input and output signals, as defined by the module’s functionality. Lee and Patel propose a simple discrete relaxation method for value selection. The reason why such a simple method works well is that path selection tries to avoid selections that may lead to conflicts during value selection. A limitation of Lee and Patel’s method is that it explicitly enumerates the control behaviors of the processor by considering every instruction in the ISA. Such an enumeration is no longer possible for pipelined processors, as instructions do not execute in isolation.

Hansen and Hayes describe a high-level test generation algorithm, called SWIFT, in [Hans95b]. SWIFT can guarantee low-level fault coverage through the use of a functional fault model, described in [Hans95a]. SWIFT uses high-level information about the circuit in the form a set of (multicycle) operations that the circuit can execute. Given a precomputed test for a module, SWIFT first constructs a partially-ordered set of operations needed to apply that test to the module and propagate the fault effects to the system outputs. It then proceeds with detailed low-level processing (scheduling). Although the results in [Hans95b] are very promising, it is not clear how to derive needed high-level information automatically.

Iwashita et al. [Iwas94] describe a technique for generating instruction sequences to excite given “test cases”, such as hazards, in pipelined processors. Test cases are mapped onto states of a reduced FSM model of the processor. The technique performs implicit enumeration of the reachable states to synthesize the desired test sequences. Some limitations are that the reduced FSM model is derived manually, and that no details are given on the effect of the abstraction on the types of test cases that can be handled.

Chandra et al. [Chan95] present a sophisticated code generator for architectural validation of microprocessors. The user provides symbolic instruction graphs together with a set of constraints; these compactly describe a set of instruction sequences that have certain properties. The system expands these templates into test sequences using constraint solvers, an architectural simulator, and biasing techniques. A similar work is discussed in [Hoss96]. As these techniques operate on the microarchitectural specification of the design only, they are not suitable for generating tests for structural errors in the implementation.

Formal verification

Bhagwati and Devadas [Bhag94] describe an automated method to verify pipelined processors with respect to their ISA specification. The method assumes that a mapping between input and output sequences of the implementation and the specification is given, and that the implementation can be approximated by a *k*-definite[1] FSM. The equivalence of the two machines is checked by symbolic simulation. The assumptions made about the implementation and the lack of abstraction limit the applicability of this approach.

Burch and Dill [Burc94] propose a method for microprocessor verification based on symbolic simulation and the use of a quantifier-free first-order logic with uninterpreted functions. The method requires manually generated abstract models of both the implementation and the specification in terms of uninterpreted functions. Symbolic simulation of the models is used to construct the next-state functions. The verification problem is turned into checking the equivalence of the next-state functions of implementation and specification.

Levitt and Olukotun [Levi97] develop a methodology for verifying the control logic of pipelined microprocessors. The datapath is modeled using uninterpreted functions. Verification is performed by iteratively merging the two deepest stages of the pipeline. After each step a check is made to see whether the newly obtained pipeline is still equivalent to the previous one. The equivalence is proven automatically using induction on the number of execution cycles. To achieve the high degree of automation, the approach of [Levi97] uses high-level knowledge about the design, such as the design intent of a bypass.

Hybrid verification techniques

A class of hybrid verification techniques [Geis96, Gupt97, Ho95, Ho96b, Lewi96, Moun98] that combine simulation with formal verification has recently been proposed. These techniques construct a reduced FSM model of the implementation. Test sequences are then generated to achieve full coverage on the reduced FSM model. A non-trivial

1. A *k*-definite FSM is one that can only remember the last *k* inputs.

problem with these methods is transforming the test sequences so that they can be applied to the implementation. The reduced FSM model may abstract part of the design by replacing the interface signals by primary inputs. The transformed test sequences can only specify primary inputs to the specification; these need to justify the abstracted interface signals specified in the original test sequence. Ho et al. [Ho95] avoid the transformation problem by ‘forcing’ the desired interface signals onto the implementation. A serious drawback of this approach is that the simulation outcome needs to be verified manually. Moundanos et al. [Moun98] use conventional gate-level ATPG to transform test sequences. Lewin et al. [lLewi96] first map the test sequences onto sequences of architectural constraints; these constraints are then used by an architectural test generator [Ahar95] to produce test sequence for the implementation. The implementation and the specification are then simulated for the transformed test set.

Discussion

Our method uses high-level knowledge about pipeline structure similar to that used in Levitt and Olukotun’s work [Levi97]. Among all related work, Lee and Patel’s work [Lee92a, Lee94] is closest to our work. Our method borrows from Lee and Patel the following ideas:

- Separation of path and value selection
- Path selection aimed at avoiding value conflicts
- The use of discrete relaxation for value selection

Our method goes beyond Lee and Patel’s in the following:

- Our method handles pipelined microprocessors.
- The iterative organization of our method is based on what we call pipeframes instead of timeframes, which leads to a reduction of the decision space, as we shall see.
- Our method uses a new formulation of the path selection problem. This formulation allows ATPG to be one with a PODEM-like directed search in a flexible manner.

4.2 Pipelined processor model

In this section we introduce a new model for pipelined processors. The purpose of the model is to facilitate a more efficient test generation method by identifying high-level information about the structure of pipelined microprocessors.

An important element of microprocessor structure is the distinction between *data* and *control*. The merits of treating datapaths and controllers differently have been recognized in many other domains such as high-level synthesis, formal verification, etc. In today’s design methodologies, controllers are often described by behavioral HDL code (case statements). These descriptions are then synthesized into a gate-level or transistor-level netlist either by tools or by hand. Most signals appearing in controller descriptions are unstructured binary signals. Controllers are essentially sets of interacting finite-state machines. Datapaths, on the other hand, process structured data words and so can be represented at a higher level than the gate level, using high-level, multibit modules and buses. This high-level representation drastically reduces the size of the design representation.

From a verification point of view, it is also important to distinguish machine state that is visible to the specification, typically an instruction set architecture (ISA) model, from machine state which is specific to the implementation. In pipelined microprocessors the pipeline registers contain the implementation-specific machine state. Much of the complexity of these processors stems from the interaction between instructions in the pipeline. If instructions were to interact only through the ISA-visible part of the machine state, they could be treated independently for verification test generation.

However, instructions also interact through the implementation-specific machine state, and this is intimately related to pipeline hazards. Hennessy and Patterson [Henn90] describe three standard techniques for dealing with pipeline hazards: *stalling*, *squashing* and *bypassing*; they are illustrated in Figure 4.1. The signals that control these mechanisms are of interest because they reveal the essence of instruction interaction in the

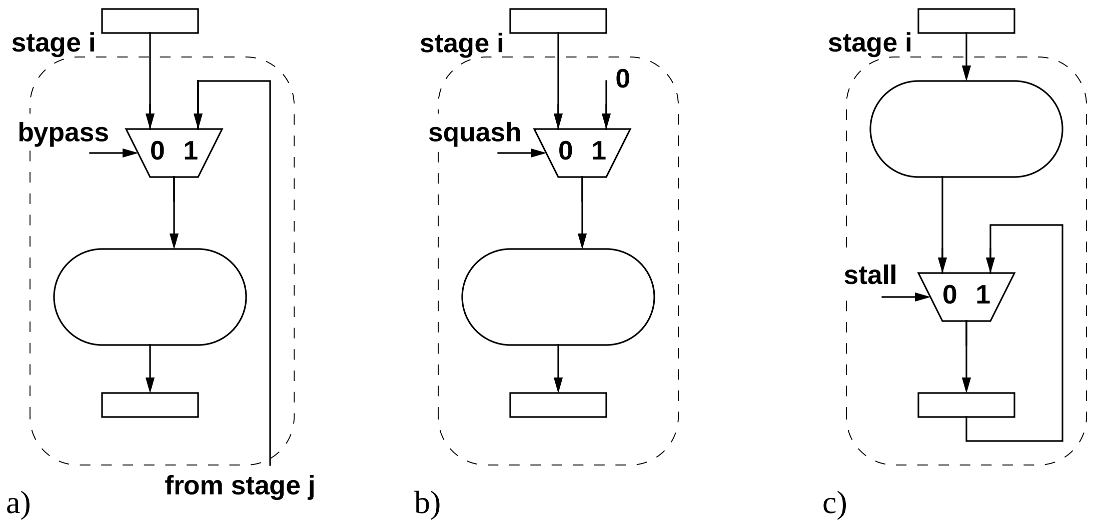


Figure 4.1: Instruction interaction mechanisms: a) bypassing, b) squashing, c) stalling

pipeline. They provide a means to characterize the control state of the pipeline in a much more compact way than by considering all the instructions in the pipeline simultaneously.

Based on these considerations and on the analysis of actual designs, we have developed the model for pipelined processors shown in Figure 4.2, which exposes high-level knowledge that can be used during test generation. We assume that data-stationary control [Kogg77] is chosen as implementation style of the controller, in which case control 'follows' the data through the pipeline providing the control signals at each stage as needed. Such controller implementations mimic the pipeline structure of the datapath. The datapath and controller both exhibit pipeline structure and interact via *status* and *control* signals. The signals at each stage are classified as:

- *primary*: interfacing with the environment
- *secondary*: interfacing with the stage's pipeline registers
- *tertiary*: interfacing with another pipeline stage

The tertiary signals are precisely the signals needed to describe essential instruction interaction. Typical examples of tertiary signals in the controller are squash and stall; typical examples of tertiary signals in the datapath are bypasses. Using the model requires no more than the appropriate labeling of control signals, status signals, and pipe registers, along with appropriate high-level modeling of the datapath. The block labeled 'global

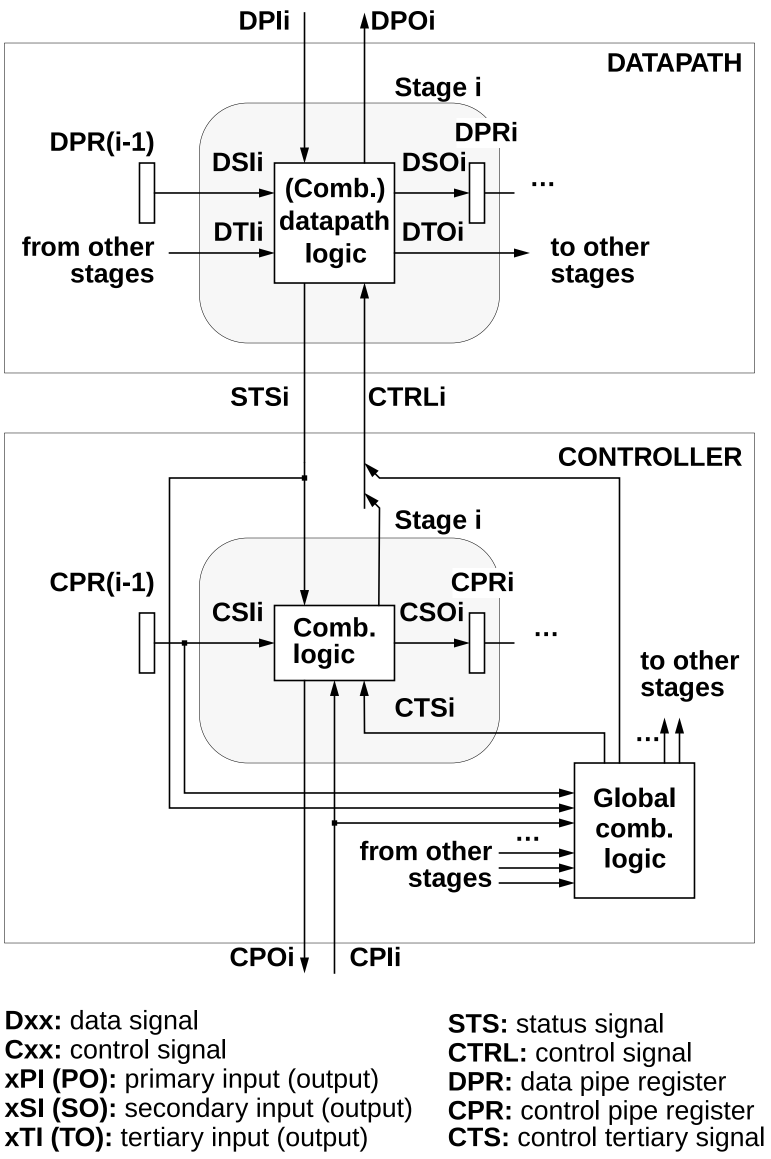


Figure 4.2: Pipelined microprocessor model

combinational logic' generates the CTS's. By isolating this block, the number of tertiary signals can be minimized.

Our test generation method attempts to decouple decisions concerning the interaction of instructions from those concerning only a single instruction. For example, a decision of the former type might be whether the current instruction needs to be stalled by the previous instruction. Such a decisions allow us to defer deciding upon the particular opcode and operand registers of that previous instruction. This is in contrast to an approach where the search is performed in the flat product space of all instructions in the pipeline. In the next section we will show how the tertiary signals can be used to decouple decisions on instruction interaction from those that concern instructions in isolation.

4.3 Pipeframe model

Conventional test generation algorithms for sequential circuits use the ILA model and iteratively apply test generation techniques for combinational circuits in one timeframe. In this section we describe a different organizational model specific to pipelined processors. This *pipeframe* organizational model exploits high-level knowledge about pipeline structure that is captured with the processor model. The advantages of this approach are a reduction of the search space and the elimination of many conflicts.

Consider the application of a conventional test generation algorithm to a pipelined controller circuit without a datapath. Figure 4.3 shows a three-stage pipelined circuit. C_1, C_2 and C_3 are combinational logic corresponding to the three pipe stages. The global combinational logic C_g sources all CPI's and all CSI's. In order not to clutter the figure, only the CPI's sourced by C_1 are shown, and the CPO's produced by C_i have been omitted. The iterative logic array model for this circuit is shown in Figure 4.4a. If PODEM is used as the combinational test generation algorithm, the decision variables are the CPI's and the CSI's in each timeframe. The decision space to be searched during each iteration is that of the CSI's and CPI's. For the controller of pipelined microprocessors, the number of CSI's (state bits) is typically much larger than the number of CPI's. This is because the primary function of the controller is to decode the incoming instructions.

Taking into account that the circuit is pipelined and performs several concurrent, and to a large extent independent, decodes, a different organization of the search, one that is directly in terms of the CPI's, is desirable. When the global control logic C_g is absent, it is easy to see how this can be accomplished. In this case, the iterative array model consists of unconnected (horizontal) slices spanning a number of timeframes equal to the number of pipe stages. These horizontal slices will be referred to as *pipeframes*. It can be seen that the size of the circuit to be considered is exactly the same as that in the conventional time-

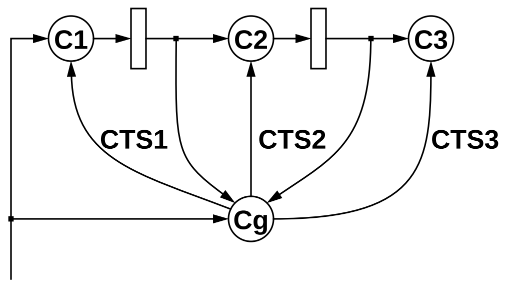


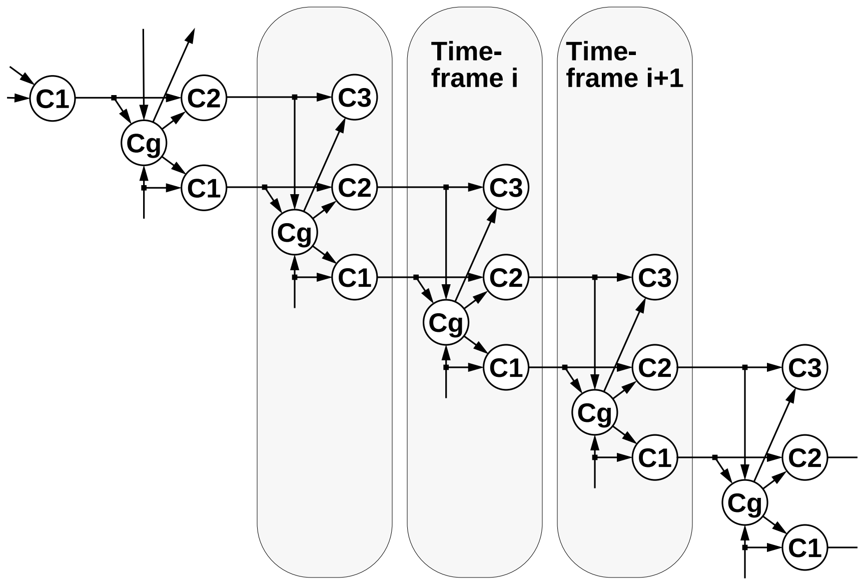
Figure 4.3: Pipelined controller

frame based search, although the depth is greater. However, in the new approach conflicts due to invalid (unreachable) states cannot arise as decisions are made only on the CPI's.

In general, there is interaction between pipestages through the global combinational logic C_g. To organize the search by pipeframe, the tertiary signals CTS_i, $i = 1, \dots 3$, need to be included as decision variables. The iterative array is partitioned into pipeframes by cutting the tertiary signals, as shown in Figure 4.4b. A complication is that a pipeframe directly interacts with a number of other pipeframes via shared primary inputs and via the tertiary signals feeding the pipeframe. In the conventional organization, each timeframe depends directly only on the previous timeframe. To cope with this complication, multiple pipeframes need to be considered simultaneously during the search. The set of pipeframes directly relevant to pipeframe *i* is indicated by window *i* in the figure. The linking of pipeframes via the tertiary signals is shown in Figure 4.5. It can be seen that the tertiary signal CTS_1 to pipeframe $i + 2$ depends on CPI's and CTS's to pipeframes i, $i + 1$ and $i + 2$. (In order not to clutter the figure the indices are omitted.)

Consider a *p*-stage pipelined controller with a total of n_1 CPI's, n_2 CSI's per pipestage, and n_3 CTS's per pipestage. In the usual timeframe organization, there are $n_1 + p.n_2$ decision variables per timeframe, $p.n_2$ of which need justification. In our pipeframe approach, there are $n_1 + p.n_3$ decision variables per pipeframe, $p.n_3$ of which need justification. Our approach is targeted at the circuits with $n_3 << n_2$. For such circuits the following can be observed:

- The size of the search space in the pipeframe organization is significantly smaller than that in the usual timeframe organization. For example in the DLX

a)


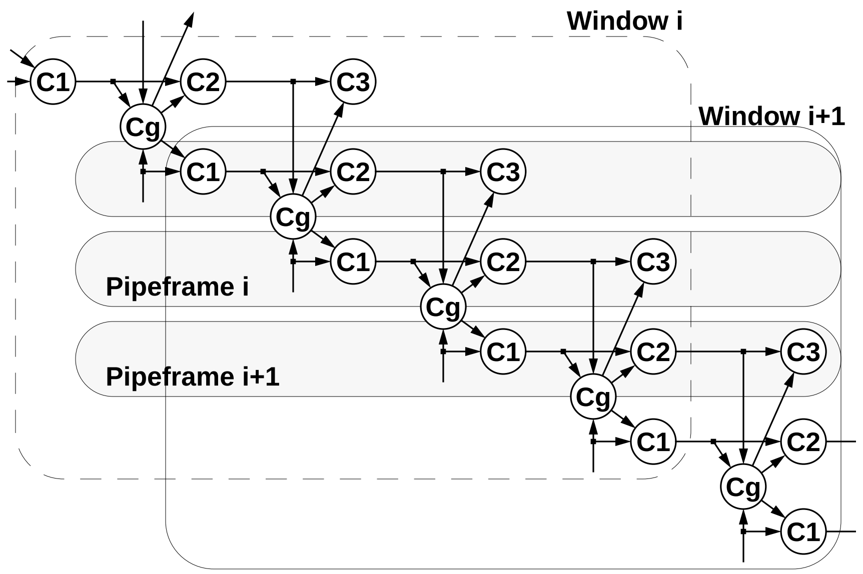
b)


Figure 4.4: Iterative array of pipelined controller: a) conventional organization; b) alternative organization

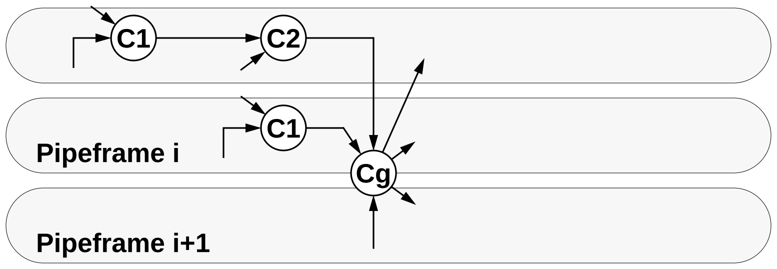


Figure 4.5: Composite circuit dealt with in pipeframe organization

implementation that will be discussed in Section 4.8, there are 95 CSI's per timeframe but only 43 CTS's per pipeframe. The size of the circuit to be dealt with in the pipeframe organization is comparable to that in the conventional organization, although its depth is greater. This can be seen in Figure 4.5.

- For some pipelined controllers the pipeframe approach does not reduce the search space. This is the case when CSO_i depends on CSI_{i+1} (referring to Figure 1) for every pipestage. Circuits in which every pipe stage can be stalled exhibit this property. For such circuits, all CSI's are also CTS's, the pipeframe approach reduces to the usual timeframe approach.

4.4 Test generation algorithm

In this section we give an overview of our high-level test generation algorithm for design verification of pipelined microprocessors. It is targeted at localized errors in the datapath, such as the SSL and CSSL errors described in Chapter 3. The algorithm follows the iterative pipeframe organization described in the previous section and decomposes the test generation problem into three subproblems:

- P_1: path selection in the datapath,
- P_2: value selection in the datapath, and
- P_3: justification of control signals in the controller.

The procedures that solve P_1, P_2, and P_3 are *DPTRACE*, *DPRELAX*, and *CTRLJUST*, respectively. The interaction of the three subproblems is illustrated in Figure 4.6. A flowchart of the overall algorithm is presented in Figure 4.7. The overall algorithm *TG* is built on top of the directed search (*CTRLJUST*) for solving P_3. *DPTRACE* selects justification and propagation paths in the datapath for activating and exposing the error. Part of the solution produced by *DPTRACE* is a set of objectives (s, v), where s is a CTRL signal and $v \in \{0, 1\}$. These objectives are used to guide the search performed by *TG*. *DPRELAX* uses discrete relaxation to determine appropriate data values.

DPTRACE computes an initial path selection and the corresponding set of path objectives. *CTRLJUST* makes decisions on the CPI, CTS and STS signals, guided by the path objectives. These decisions are implied on three fronts. First, they are implied in the controller where they affect the CPO, CTS and CTRL signals. Second, *DPTRACE* checks whether the updated CTRL signals are consistent with the current set of justification and propagation paths in the datapath. If there is consistency, no further action is required. Otherwise, *DPTRACE* computes a new set of justification and propagation paths, taking into account the current values of the CTRL lines. The objectives on the CTRL lines are updated accordingly. Only if *DPTRACE* fails to derive a set of justification and propagation paths will *DPTRACE* cause *TG* to backtrack. The third aspect of implication involves invoking *DPRELAX* to compute data values. Failure to converge will cause *TG* to backtrack. If no inconsistencies arise from the implication step, we check whether the reset state has been reached. If so, and if all objectives are satisfied, we return successfully with a test. The three subalgorithms are described in the remainder of this section.

4.5 DPTRACE: path selection in datapath

The task of the path selection algorithm *DPTRACE* is to determine a set of justification and propagation paths in the datapath to activate the error and expose its effect at a primary output of the datapath. *DPTRACE* does not consider the values that need to be justified and propagated. This task is delegated to *DPRELAX*. This divide-and-conquer approach reduces the problem size significantly, but may fail to find a solution, even if the problem is feasible.

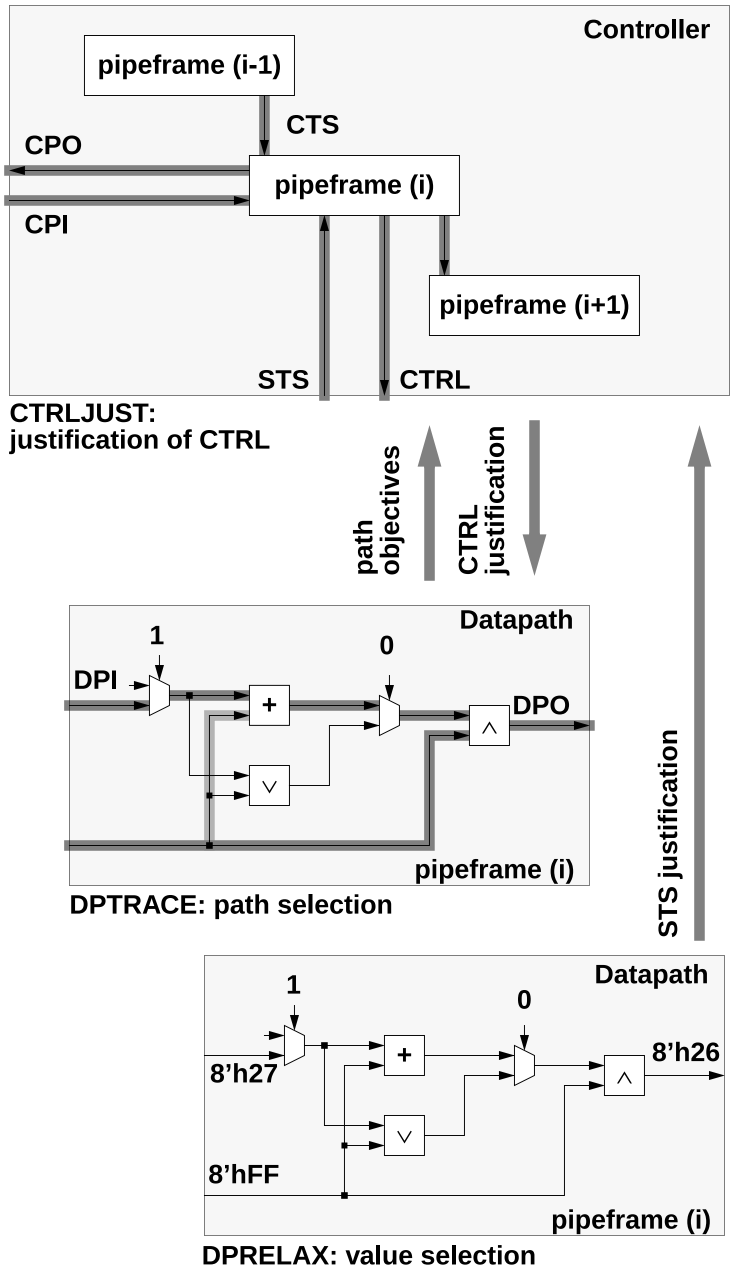


Figure 4.6: The three parts of the test generation algorithm and their interactions

We start by discussing the overall iterative organization of DPTRACE. We then present a controllability / observability graph (COG) for analyzing the path selection problem. This leads to a formulation that can be solved with a PODEM-like directed search.

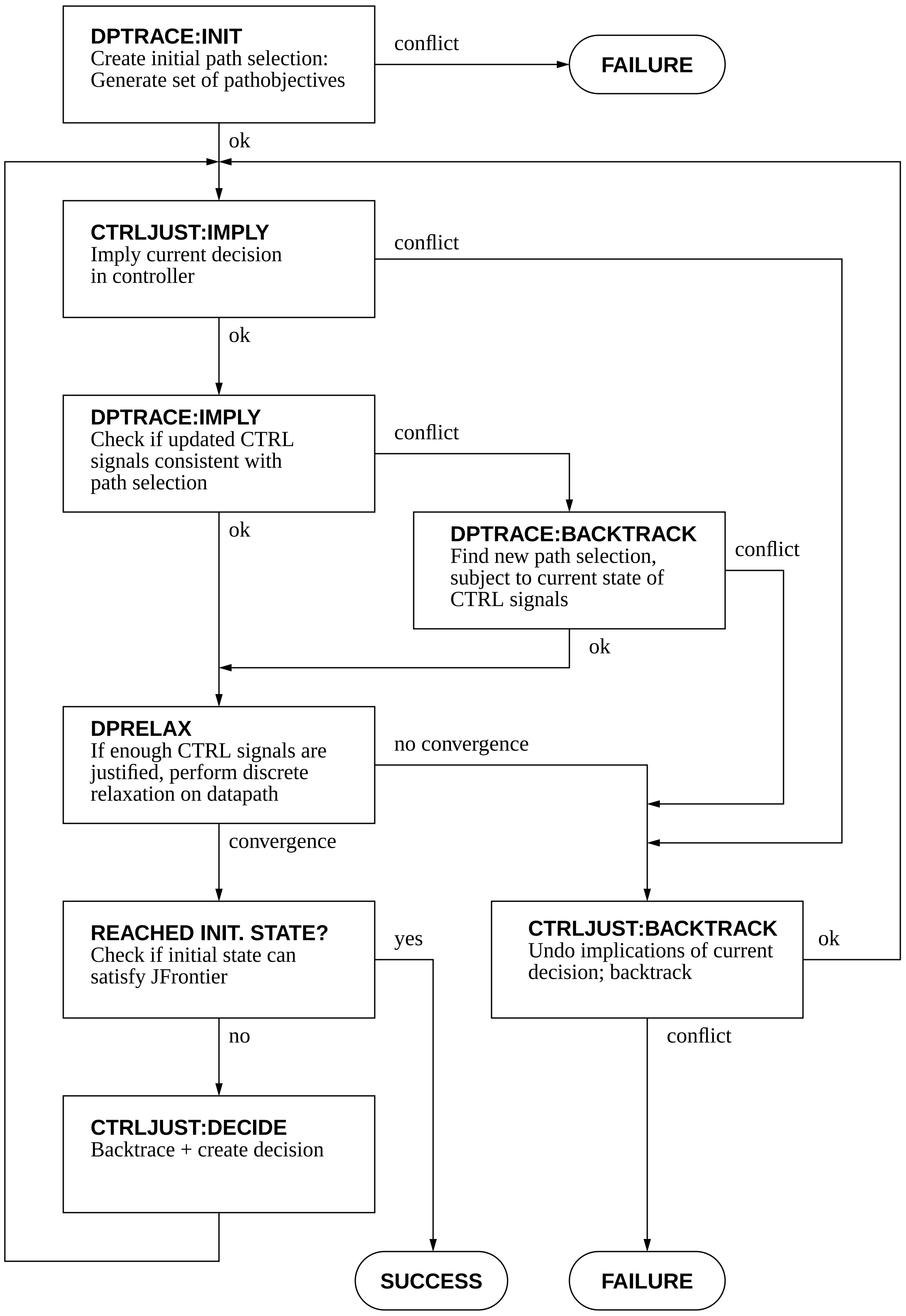


Figure 4.7: Flowchart of overall test generation algorithm

Overall iterative organization

The overall iterative organization of DPTRACE is based on pipeframes. This organization is illustrated in Figure 4.8 for a four-stage pipeline. Initially, only a single pipeframe, the excitation frame, is considered when computing a set of justification paths (Figure 4.8a). If this fails, another pipeframe is added to the COG. The process is repeated until all lines are justified, or until a maximum number of frames has been explored. In the figure a valid path selection is obtained after adding two pipeframes (Figure 4.8b). Next, a set of propagation paths is computed. Again, we start with considering the excitation frame only. If no path selection is found, but the error effect can be exposed at a DTO, another pipeframe is added to the COG. In Figure 4.8c a complete set of paths is obtained.

Controllability/observability graph (COG)

In Lee and Patel’s work [Lee92a, Lee94], justification and propagation paths are created by iteratively extending an instruction sequence with one instruction. A new instruction is chosen to minimize the remaining justification effort. This choice is among the instructions in the ISA.

In our work, decisions are made at a finer grain during path selection. Justification and propagation paths are created by setting individual CTRL signals. During the path selection phase, we wish to determine a partial assignment to the CTRL signals so that the error can be activated, and its effect can be propagated to a primary data output, by applying appropriate values to the data primary inputs. As in Lee and Patel’s work, path selection is aimed at avoiding conflicts during the subsequent value selection phase.

Conflicts during value selection arise when constraints corresponding to the modules and interconnection are violated. We can distinguish four classes of basic datapath modules: ADD, AND, MUX, and FAN. More complex modules such as ALU’s are modeled as a composition of simpler high-level modules, such as word-gates, adders, multiplexers, etc.

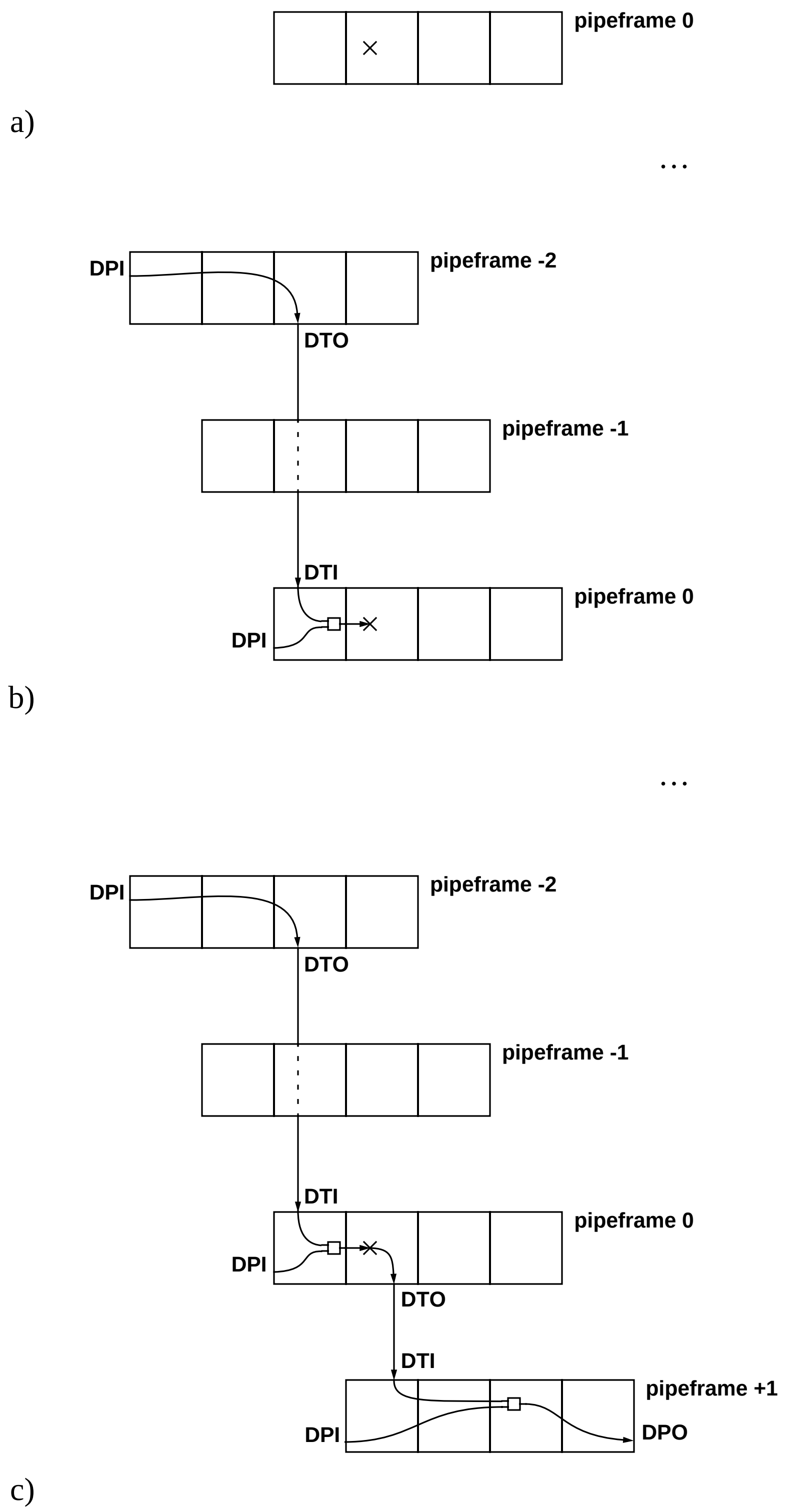


Figure 4.8: Overall iterative organization of path selection

We define a controllability / observability graph (COG) for a sequence of pipeframes in the ILA of the datapath. Its nodes correspond to modules in the datapath, nets in the datapath with multiple fanout, primary inputs and outputs, tertiary inputs and outputs, control and status signals; its edges correspond to pairs of connected ports (module terminals) in the datapth. Note that in the ILA all (pipeline) registers have been eliminated. The boundaries of the ILA are formed by nodes corresponding to DPI, DPO, CTRL, STS signals, exterior CTI (driven by a pipeframe not in the sequence considered), and exterior CTO signals (sourced by pipeframes not part of the sequence considered only).

Modules in the ADD class have one data output, and one or more data inputs. They have the property that the output can be justified (to an arbitrary value) by controlling only a single input, i.e., regardless of the values of the other inputs, the controlled input can be assigned a value that will justify the output. Also, if the output is observable then every input is observable as well. Modules in this class include the buffer, the inverter, the adder, the subtractor, and the X(N)OR word gate. Predicate modules, which take two n-bit inputs A and B and produce a single-bit output $Y = A <op> B$, where $<op> \in \{=, \neq, <, \leq, >, \geq,$ ADDOVF, SUBOVF$\}$, are also in the ADD class for the controllability analysis. For the observability analysis, they are in the AND class. ADDOVF and SUBOVF compute overflow for signed addition and subtraction, respectively.

Modules in the AND class have one data output, and one or more data inputs. In order to justify the output (to an arbitrary value) all inputs need to be controlled. To observe an input, the output needs to be observable and all side inputs need to be controlled. Modules in this class include word gates such as (N)AND, (N)OR, and SHIFT modules.

Modules in the MUX class have one data output, one or more data inputs, and one or more control inputs. The control inputs determine which data input is selected. In order to justify the output, the control inputs need to be assigned and the selected data input needs to be controlled; the other data inputs are free[1]. In order to observe a data input, the output needs to be observable, and the control inputs need to be assigned such that the requested

1. DPRELAX requires that select signals to multiplexers are set. Hence DPTRACE does not consider justification of the output of a multiplexer by controlling all data-inputs and leaving the select input unassigned.

data input is selected. This class contains modules such as multiplexers and tristate buffers.

FAN nodes correspond to nets in the datapath that have multiple fanouts, and are the only type of node in the graph that have more than one outgoing edge. A FAN node has one incoming edge corresponding to the module driving the net, and outgoing edges to each module that sources the net. FAN nodes have the property that only one fanout can be justified by controlling the stem. Conflicts will arise if we attempt to use a stem for different simultaneous justification problems, unless all problems request the same value on the stem, but the concrete values are not known during path selection. Therefore, we associate a so-called *FO-select* variable with FAN nodes to reflect this constraint. This variable indicates which outgoing edge 'uses' the incoming edge for its justification, and assumes values from $\{1,\ldots, n, u\}$, where n is the number of outgoing edges, and u stands for unassigned.

Note that because this analysis does not take into account the concrete values that need to be justified, it is an approximation. Consider for example the predicate module $Y = A < B$, which we classified as an ADD-class module because Y can be justified by controlling just a single input. However, there is a singular case in which $Y = 1$ cannot be justified by controlling only A, namely for B equal to the largest positive value.

Given a COG, and a *complete* assignment to its CTRL and FO-select variables (no u values), we can easily identify potential conflicts when trying to activate an error site and propagating the error effect. First, traverse the COG forwards in levelized order, starting with the primary inputs, and compute the controllability of each edge, using the properties discussed above. If the error site is controlled, we should be able to activate the error during value selection, otherwise conflicts are likely. Next, traverse the COG backwards in levelized order, and compute the observability of each edge. If the error site is determined to be both controllable and observable, we should be able to activate the error and propagate its effect to a primary output, otherwise conflicts are likely.

The path selection problem is that of finding a *partial* assignment to the CTRL variables and the FO-select variables of the COG, such that the error site can be controlled and observed. A PODEM-like search with the CTRL and FO-select variables as decision variables is a good starting point for an algorithm. To make the search efficient, conflicts need to be identified as early as possible. In other words, we need to foresee conflicts that may arise when trying to activate the error and expose its effect for the given partial assignment to the CTRL variables. For this purpose, we have developed the following system:

We attribute to an edge in the COG a symbolic value that encodes controllability information. The attribute *C*-state assumes values from the set {*C*1, *C*2, *C*3, *C*4}. The interpretation is as follows:

- *C*4. The *C*-state of edges that are *controlled* is *C*4. That is, for the current partial assignment of CTRL and FO-select variables, the signal corresponding to such an edge can be set to an arbitrary value by applying appropriate values to the primary inputs. This requires that there is at least one path from a DPI to this edge, for which the *C*-states of all path segments (edges) are *C*4. It also implies that all CTRL variables in the transitive fanin of the edge are assigned.
- *C*3. The *C*-state of edges that are *not-controlled* is C3. That is, for the current partial assignment of CTRL and FO-select variables, the signal corresponding to such an edge can be set to a value by applying appropriate values to the primary inputs, but not to an arbitrary value. This also implies that all CTRL variables in the transitive fanin of the edge are assigned.
- *C*2. The *C*-state of edges that have unassigned decision variables in their transitive fanin, and that can only become not-controlled (*C*3) after completing the current partial assignment of CTRL and FO-select variables, is *C*2.
- *C*1. The *C*-state of edges that have unassigned decision variables in their transitive fanin, and that have the potential to become controlled (*C*4) after completing the current partial assignment of CTRL and FO-select variables, is *C*1.

Similarly, edges in the COG are assigned symbolic values that encode observability information. The attribute *O*-state assumes values from the set {*O*1, *O*2, *O*3}. The interpretation is as follows:

Table 4.1: Initial C- and O- values

Signal	C-state	O-state
DPI	$C4$	$O1$
external DPO	$C1$	$O3$
external DTI	$C3$	$O1$
DTO	$C1$	$O2$
CTRL	N/A	N/A
STS	$C1$	$O2$
Other	$C1$	$O1$

- $O3$. The O-state of edges that are observable for the current partial assignment of CTRL and FO-select variables is $O3$. There exists at least one sensitized path from such an edge to a primary output.
- $O2$. The O-state of edges that are not observable is $O2$. No sensitized path exists from such an edge to a primary output.
- $O1$. The O-state of edges that have the potential to become observable after completing the partial assignment to the decision variables, is $O1$.

Using these value systems, we can formally state the relationship between controllability and observability information of the incoming edges of a node and that of the outgoing edges. We present propagation tables for a representative of each class of modules in Figure 4.9; the bottom two tables are for a net with stem x and two fanouts $y1$ and $y2$. The C- and O-state of terminal nodes are initialized as shown in Table 4.1.

Consider the datapath shown in Figure 4.6; the corresponding COG is shown in Figure 4.10. The goal is to control edge $n12$–$n13$. So far all CTRL variables are still unassigned and one FOselect variable, that associated with $n8$, has been set (to 3). As a result of this decision, edge $n8$–$n12$ is controlled, whereas $n8$–$n9$ and $n8$–$n10$ are not-controlled. Note that the $C3$ state of $n8$–$n10$ propagates to $n10$–$n11$. The $C2$ value on $n10$–$n11$ will avoid setting CTRL variable $n5$ to 1, which may lead to a conflict during value selection. It can be seen that the current state is consistent with the propagation tables of Figure 4.9.

y = ADD (*x*1, *x*2)

C(*y*)		**C(*x*2)**			
		*C*1	*C*2	*C*3	*C*4
C(*x*1)	*C*1	*C*1	*C*1	*C*1	*C*1
	*C*2	*C*1	*C*2	*C*2	*C*1
	*C*3	*C*1	*C*2	*C*3	*C*4
	*C*4	*C*1	*C*1	*C*4	*C*4

y = ADD (*x*1, *x*2)

O(*x*1)		**O(*y*)**		
		*O*1	*O*2	*O*3
C(*x*2)	*C*1	*O*1	*O*2	*O*1
	*C*2	*O*1	*O*2	*O*1
	*C*3	*O*1	*O*2	*O*3
	*C*4	*O*1	*O*2	*O*3

y = AND (*x*1, *x*2)

C(*y*)		**C(*x*2)**			
		*C*1	*C*2	*C*3	*C*4
C(*x*1)	*C*1	*C*1	*C*2	*C*2	*C*1
	*C*2	*C*2	*C*2	*C*2	*C*2
	*C*3	*C*2	*C*2	*C*3	*C*3
	*C*4	*C*1	*C*2	*C*3	*C*4

y = AND (*x*1, *x*2)

O(*x*1)		**O(*y*)**		
		*O*1	*O*2	*O*3
C(*x*2)	*C*1	*O*1	*O*2	*O*1
	*C*2	*O*2	*O*2	*O*2
	*C*3	*O*2	*O*2	*O*2
	*C*4	*O*1	*O*2	*O*3

y = MUX2 (*s*, *x*1, *x*2) (*s*=1 selects *x*2)

C(*y*)		
s	*u*	*C*2 if *C*(*x*1), *C*(*x*2) ∈ {*C*2,*C*3} *C*1 otherwise
	0	*C*(*x*1)
	1	C(*x*2)

y = MUX2 (*s*, *x*1, *x*2)

O(*x*1)		**O(*y*)**		
		*O*1	*O*2	*O*3
s	*u*	*O*1	*O*2	*O*1
	0	*O*1	*O*2	*O*3
	1	*O*2	*O*2	*O*2

(*y*1, *y*2) = FAN(*x*)

C(*y*1)C(*y*2)		***FOsel***		
		u	1	2
C(*x*)	*C*1	*C*1 *C*1	*C*1 *C*2	*C*2 *C*1
	*C*2	*C*2 *C*2	*C*2 *C*2	*C*2 *C*2
	*C*3	*C*3 *C*3	*C*3 *C*3	*C*3 *C*3
	*C*4	*C*1 *C*1	*C*4 *C*3	*C*3 *C*4

(*y*1, *y*2) = FAN(*x*)

O(*x*)		**O(*y*2)**		
		*O*1	*O*2	*O*3
O(*y*1)	*O*1	*O*1	*O*1	*O*3
	*O*2	*O*1	*O*2	*O*3
	*O*3	*O*3	*O*3	*O*3

Figure 4.9: *C*- and *O*-propagation tables

Formulation

A path selection in COG consists of an assignment of the *C*-state and *O*-state of all edges, and of the CTRL signals to the pipeframe. A path selection is valid if it is consistent with the *C*- and *O*- propagation tables of all nodes. Given a localized error in the datapath, the path selection problem is that of finding a valid path selection such that the edge associated with the error is both controllable (*C*4) and observable (*O*3).

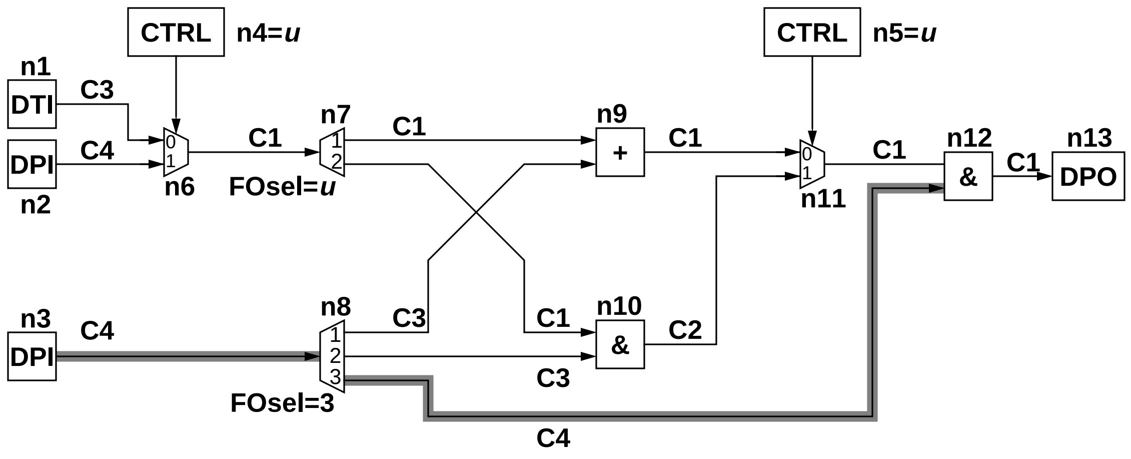


Figure 4.10: Path selection using *C*-values

Path selections in which the *C*-state of CTI edges is set to *C*4 require further justification. Similarly, path selections in which the *O*-state of CTO edges is set to *O*3 require further propagation.

In general, consider a set of edges, called the *J*-frontier, whose *C*-state needs to be justified to *C*3 or *C*4. Also given is a set of edges, called the *E*-frontier, which contains the error effect. At least one line in the *E*-frontier needs to be made observable (*O*3). The path selection problem is as follows: Given the following: a *J*-frontier = $\{(e_i,\ ce_i) \mid i=1\ldots n\}$ where e_i is an data-edge, and $ce_i \in \{C3,\ C4\}$; an *E*-frontier[1] = $\{e_i \mid i=1\ldots m\}$; a partial assignment to the CTRL variables; determine a partial assignment to the decision variables such that the *C*-state of every line in the J-frontier is justified to the specified value, and the *O*-state of at least one line in the E-frontier is justified to *O*3.

Directed search algorithm

The path selection problem can be solved using a directed search similar to PODEM [Goel81, Abra90]. Pseudocode for PODEM is shown in Figure 4.11. The adaptation of

1. In design verification we use the term *error* to differentiate from the term *fault* used in physical fault testing; hence *E*-frontier instead of *D*-frontier.

```
PODEM()

1.      if (goal satisfied) {
1.1         return SUCCESS
2.      }
3.      if (goal infeasible) {
3.1         return FAILURE
4.      }
5.      obj ← Select next objective ()
6.      decision ← Backtrace(obj)
7.      Imply( dec)
8.      if (PODEM() = SUCCESS) {
8.1         return SUCCESS
9.      Undo implications of decision
10.     while (untried alternatives to decision exists) {
10.1        dec ← Select untried alternative (decision)
10.2        Imply( dec)
10.3        if (PODEM() = SUCCESS) {
10.3.1          return SUCCESS
10.4        }
11.     }
12.     return FAILURE
```

Figure 4.11: Directed search PODEM

PODEM involves appropriate definition of 1) decision variables, 2) implication procedure, 3) backtrace procedure, and 4) consistency checking.

Decision variables. The decision variables in the path selection problem are:

- CTRL variables, which are associated with the CTRL signals, assume values from $\{0, 1, u\}$.
- FO-select variables, which are associated with multiple-fanout nodes, indicate which outgoing edge uses the incoming edge for its justification. These variables assume values from $\{1,\ldots, n, u\}$, where n is the number of outgoing edges.

Implication procedure. After a decision variable has been assigned, the implications of that decision have to be computed. This is done by an event-driven forward traversal of COG to compute *C*-values, and a similar backward-traversal of the COG to compute *O*-values. The rules are those presented earlier in Figure 4.9. Decisions involving CTRL variables are also implied *functionally.* For each pipeframe a predicate is kept that is the conjunction of the assignments made to the CTRL variables so far. The support of the

```
BacktraceAdd( port, obj)

1.        nextObj ← obj
2.        switch( obj) {
2.1           C3: nextPort ← Select inPort with cState in {C1,C2}
2.2           C4: nextPort ← Select lowest cost inPort with cState=C1
2.3           O3: if ( oState(outPort)=O1) {
2.3.0.1                 nextPort ← outPort
2.3.1             }
2.4               else { /* oState(outPort)=O3 */
2.4.1.1                 nextPort ← Select side input with cState in {C1,C2}
2.4.1.2                 nextObj ← C3
2.4.2             }
3.        }
4.        decision ← BacktraceNet( nextPort, nextObj)
5.        return decision

BacktraceAnd( port, obj)

1.        nextObj ← obj
2.        switch( obj) {
2.1           C3: nextPort ← Select inPort with cState in {C1,C2}
2.2           C4: nextPort ← Select highest cost inPort with cState=C1
2.3           O3: if ( oState(outPort)=O1) {
2.3.0.1                 nextPort ← outPort
2.3.1             }
2.3.2             else { /* oState(outPort)=O3 */
2.3.2.1                 nextPort ← Select highest cost side input with
                                   cState=C1
2.3.2.2                 nextObj ← C4
2.3.3             }
3.        }
4.        decision ← BacktraceNet( nextPort, nextObj)
5.        return decision
```

Figure 4.12: Backtrace rules for path selection: ADD, AND

predicate is the CPI, CTI and STS signals. Functional implication of a decision involving a CTRL variable involves updating the predicate to reflect the additional assignment. If the updated predicate is false, the search will backtrack. These predicates, one for each pipeframe, constitute the path objectives that DPTRACE returns upon success, and which are used by CTRLJUST to direct its search.

Backtrace procedure. Backtracing is a heuristic procedure that takes an objective, in our case a pair consisting of an edge and a desired *C*- or *O*- value, and produces an

```
          BacktraceFan( port, obj)

1.        switch (obj) {
1.1           C3: if (cState(fanin) in {C1,C2}) {
1.1.0.1               decision ← BacktraceModule (fanin, obj)
1.1.1             }
1.1.2             else { /* FOselect unassigned */
1.1.2.1               decision ← (FOSel,port)
1.1.3             }
1.2           C4: if (FOselect unassigned) {
1.2.3.1               decision ← Select port
1.2.4             }
1.2.5             else {
1.2.5.1               decision ← BacktraceModule( fanin, obj)
1.3           O3: fanOut ← lowest cost fanout with oState = O1
1.3.6             decision ← BacktraceModule (fanout, obj);
2.        }
3.        return decision
```

```
          BacktraceMux( port, obj )

1.        if ( all selects determined ) {
1.1           switch (obj) {
1.1.1             C3,C4:  decision ← BactraceNet( selected inPort, obj)
1.1.2             O3:     decision ← BactraceNet( outPort, obj)
1.2           }
2.        }
3.        else {
3.1           switch (obj) {
3.1.1             C3: inPort ← Select selectable input with smallest cost
3.1.2             C4: inPort ← Select lowest cost slectable input with cState in {C1,C4}
3.1.3             O3: inPort ← port
3.2           }
3.3           decision ← assign select signal  to select inPort
4.        }
5.        return decision
```

Figure 4.13: Backtrace rules for path selection: FAN, MUX

assignment to a decision variable that is likely to help achieve the objective. Figure 4.12 and Figure 4.13 present a set of mutually recursive backtrace procedures. Backtracing is guided by *Ccosts* and *Ocosts* that are discussed below. The procedure for backtracing through a net, BacktraceFan in Figure 4.13, shows how decisions involving a FO-select variable are created. The procedure for backtracing through a multiplexer, BacktraceMux in Figure 4.13, shows how decisions involving a CTRL variables originate.

Objective selection. Initially there are two objectives: 1) to control the error site, and 2) to make the error site observable. As the search progresses, more lines (tertiary signals) may need justification and the error effect may be observable at multiple sites.

Feasibility checks. After the implication phase we need to check if the current partial path selection can still be augmented to satisfy the goals. Such an augmentation is no more feasible if the *C*-state of a line in the *J*-frontier is violated or if the *E*-frontier has become empty.

Controllability and observability measures. Search algorithms such as PODEM exhibit freedom as to the order of the decision variables and the order in which alternative values are tried. The way in which this freedom is used can significantly affect the execution time of the search. The following principles are commonly used [Abra90, Goel81]:

- Among a set of unsolved subproblems, first attack the hardest one.
- Among a set of potential solutions to a problem, first try the easiest one.

To implement these principles, metrics to gauge the difficulty of problems and alternative solutions are required. The subproblems in path selection consist of justifying (controlling) edges in the COG. Alternative solutions arise not only during justification but also during error effect propagation. The SCOAP measures [Gold79] are widely used for analyzing the controllability and observability of a node in gate-level designs. We have adapted these metrics for the path selection problem.

The *Ccost* of an edge estimates the difficulty of controlling the edge. The *Ccost* is computed by traversing the nodes of the COG forwards in level order, starting with the primary inputs. The expressions for the *Ccost* of the outgoing edges in terms of these of the incoming edges are given in Table 4.2. Similarly, the difficulty of observing an edge is measured by the *Ocost*. These measures are computed by traversing the nodes of the COG backwards in level order, starting with the primary outputs. Again the expressions are given in Table 4.2.

Table 4.2: Computation of controllability and observability measures for a node with incoming edges $x_1 \ldots x_m$ and outgoing edges $y_1 \ldots y_n$

Type	*Ccost*(y_j)	*Ocost*(x_j)
DPI	1	N/A
DPO	N/A	0
DTI	constant$_1$	N/A
DTO	N/A	constant$_3$
CTRL	constant$_2$	N/A
STS	N/A	MAXINT
ADD	$1 + \min\{Ccost(x_i) \mid i = 1 \ldots m\}$	$1 + Ocost(y_1)$
AND	$1 + \Sigma\ \{Ccost(x_i) \mid i = 1 \ldots m\}$	$1 + Ocost(y_1) + \Sigma\ \{Ccost(x_i) \mid i \neq j,\ i = 1 \ldots m\}$
MUX	$1 + \min\{Ccost(x_i) \mid i = 1 \ldots m\}$	$1 + Ocost(y_1)$
FAN	$Ccost(x_1)$	$\min\{Ocost(y_i) \mid i = 1 \ldots n\}$

4.6 DPRELAX: value selection in datapath

The task of the value selection algorithm is to determine DPI values that expose the error effect and justify any STS signals assigned by *CTRLJUST*. As in path selection, the problem is solved on a per pipeframe basis.

More precisely, the value selection problem for a single pipeframe is as follows: Given a partial assignment to the CTRL, DPI, and DTI signals, and a set of (s, v) pairs that need to be justified, where s is a STS or DTO signal, and v is an integer value, determine a partial assignment to the DPI and DTI signals that exposes the error effect at a DPO or DTO and justifies every given (s, v).

This problem can be formulated as that of finding a solution to a system of non-linear equations [Lee92b]. For special cases, such as that of datapaths containing only linear modules, efficient deterministic methods can be devised to solve the system. However such techniques are not applicable to the non-linear systems that result in most practical cases. Lee and Patel [Lee92b] suggested the use of discrete relaxation in the context of high-level test generation for physical fault testing. The main advantages of this technique are its simplicity and its ability to deal with any type of combinational datapath modules. A disadvantage is that it is not a complete method: it cannot prove that the system has no solutions, and may fail to find a solution even if one exists. A key observation is that during path selection, appropriate justification and propagation paths are selected so that the system to be solved during value selection is likely to be underdetermined, in which case discrete relaxation is likely to converge quickly.

In our discrete relaxation algorithm, each net in the datapath is characterized by two pairs of variables, one corresponding to the error-free circuit, the other to the erroneous circuit. Each pair consists of an integer in the range specified by the bit-width of the net, and a type which is in the range {*unassigned*, *determined*, *fixed*}. The algorithm iteratively re-evaluates the modules in the circuit until a consistent assignment is obtained or until a maximum iteration count is exceeded. The mechanism is event-driven. Events, associated with the terminals of a net, are triggered when the value of that net is changed. An event is processed by re-evaluating the module to which the corresponding terminal belongs. If the current values of the nets connected to the module are consistent with the module's functionality, no further action is required. Otherwise the values of one or more nets connected to the module are changed in order to make them consistent. New events are generated for all terminals (except those belonging to the module that is being processed) of the nets whose value has been changed.

The choice of which net to update and what value to assign can, in principal, be random, but it strongly influences convergence. We implemented a number of heuristics whose goal is to try to exercise all possible modes of event propagation and to aid convergence.

4.7 CTRLJUST: CTRL line value justification

DPTRACE computes a set of paths to activate and propagate the error in the datapath; during this process values are assigned to some CTRL lines. These assignments, which may involve multiple time frames, need to be justified by CTRLJUST. The CTRL line value justification problem is that of finding an input sequence that, when applied to the circuit, and starting from its reset state, makes the controller produce the desired values on the CTRL lines.

We are only concerned with errors in the datapath, and assume that the error effects do not propagate through the controller. Hence, CTRLJUST is concerned with the error-free controller only. The CTRL line value justification problem is related to the state justification problem in conventional sequential ATPG.

Approach

We propose a PODEM-like search to solve the justification problem. In our approach, however, the decision variables are the CPI, CTS, and STS signals. Decisions on CTS signals need further justification. Decisions on STS signals need to be justified by DPRELAX. This is in contrast to conventional approaches where decisions are made on the primary and *secondary* signals. Our approach can be viewed as a middle ground between two extremes. At one end of the spectrum are iterative methods that process strictly a single timeframe at a time. At the other end of the spectrum, is FASTEST [Kels93], which is not iterative and does not require intermediate justification.

An example of a method of the first type is to apply PODEM iteratively in a single timeframe. This is one of the techniques HITEC [Nier91b] uses for state justification. This allows for a fairly efficient search for a predecessor state. However, if the selected predecessor is not reachable from the reset state, or if it requires a very long transfer sequence, this approach may suffer. This difficulty stems from the locality of the decision process: decisions concerning one timeframe are made without considering their dependencies on signals in the previous timeframe.

FASTEST [Kels93] is at the other end of the spectrum. FASTEST first computes an estimate of the number of time frames required to activate and propagate the fault; it then directly applies PODEM to an ILA of the estimated length. Consequently, no decisions need justification. If the estimated length is accurate, the ability to use information spanning more than one timeframe may lead to a more efficient search. A weakness of this approach is that the decision space may become very large, which may lead to a large amount of backtracking.

Our approach uses information spanning more than one timeframe, but is also iterative. Furthermore for the designs that we target, the number of decision variables that need justification is smaller than in the conventional iterative approach.

A complication of our approach is that tertiary signals, which are the decision variables that need justification, directly depend on primary and tertiary signals in more than just the previous timeframe. We elaborate on this next.

Transition system

In the conventional, timeframe-based organization the secondary signals define the partition on the ILA. This corresponds to the conventional FSM view of the circuit. In the pipeframe organization, the tertiary signals define the partition on the ILA. The transition system corresponding to this organization can be derived as follows.

Consider the FSM view $M(I, O, S, \delta, \lambda, S_0)$ of a pipelined sequential circuit, where I is the input space, O is the output space, S is the state space, δ are the next-state functions, λ are the output functions, and S_0 is the reset state.

Referring to Figure 4.2, the state registers of M are partitioned according to the pipe stage they belong to: $S = S_1 \times S_2 \times \ldots \times S_{n-1}$, where n is the number of pipeline stages. The complete state is composed of components from every stage (except for the last stage n, which does not contain any pipeline registers): $s = (s_1, s_2, \ldots, s_{n-1})$. The component next-state functions are:

$\delta_i : \mathrm{S} \times I \to S_i : s_i' = \delta_i(s, x)$, where $i = 1, \ldots, n-1$.

Let U be the space of the tertiary signals. The next-state functions can be expressed in terms of the tertiary signals as follows:

$\gamma_i : S_{i-1} \times U \times I \to S_i : s_i' = \gamma_i(s_{i-1}, u, x)$, where $i = 2, \ldots, n-1$.

$\gamma_1 : U \times I \to S_1 : s_1' = \gamma_1(u, x)$

$\rho : S \times I \to U : u = \rho(s, x)$

Note how the pipeline structure is exposed: s_i' directly depends only on component s_{i-1} of s. Let s_i^t denote s_i in timeframe t, then $s'^t_i = s_i^{t+1}$.

$s_1^t = \gamma_1(u^{t-1}, x^{t-1})$

$s_i^t = \gamma_i(s_{i-1}^{t-1}, u^{t-1}, x^{t-1})$, where $i = 2, \ldots, n-1$

$u^t = \rho(s_1^t, \ldots, s_{n-1}^t, x^t)$

$$y^t = \lambda(s_1^t, \dots, s_{n-1}^t, x^t)$$

We can iteratively eliminate all s_i, starting with s_1:

$$s_2^t = \gamma_2(\gamma_1(u^{t-2}, x^{t-2}), u^{t-1}, x^{t-1}),$$

$$s_i^t = \gamma_i(s_{i-1}^{t-1}, u^{t-1}, x^{t-1}), \text{ where } i = 3, \dots, n-1$$

$$u^t = \rho(\gamma_1(u^{t-1}, x^{t-1}), \dots, s_{n-1}^t, x^t)$$

$$y^t = \lambda(\gamma_1(u^{t-1}, x^{t-1}), \dots, s_{n-1}^t, x^t)$$

Finally we obtain:

$$u^t = \psi(u^{t-1}, \dots, u^{t-n+1}, x^t, \dots, x^{t-n+1}) \tag{4.1}$$

$$y^t = \kappa(u^t, \dots, u^{t-n+1}, x^t, \dots, x^{t-n+1}) \tag{4.2}$$

We define an extended pipestate $\bar{w}^t = (u^t, \dots, u^{t-n+1}, x^t, \dots, x^{t-n+1})$ as a cube in the space $U^n \times I^n$. Let $\bar{w}_R^t = (u_R^t, \dots, u_R^{t-n+1}, x_R^t, \dots, x_R^{t-n+1})$ be the reset pipestate. The pipestate justification problem is to find an input sequence that brings the machine from the reset state into the desired pipestate.

Equation 4.1 is illustrated in Figure 4.14b for the three-stage pipelined circuit shown in Figure 4.14a. The co-domains of the transition functions for tertiary signals (Equation 4.1) span multiple timeframes. To justify assignment to signals in timeframe t, assignments to primary signals in timeframes $t, \dots, t-n+1$ and to tertiary signals in timeframes $t-1, \dots, t-n+1$ may be required.

Algorithm

Pseudo-code for the overall algorithm TG is presented in Figure 4.15. TG is built on top of the directed search CTRLJUST. In fact, what is left of TG after dropping steps 2, 3.3, and 3.4 is CTRLJUST. DPTRACE selects justification and propagation paths in the datapath for activating and exposing the error, and thereby produces path objectives, which are partial assignments to the CTRL lines. These path objectives are used to guide the search performed by TG. DPRELAX uses discrete relaxation to determine appropriate data values.

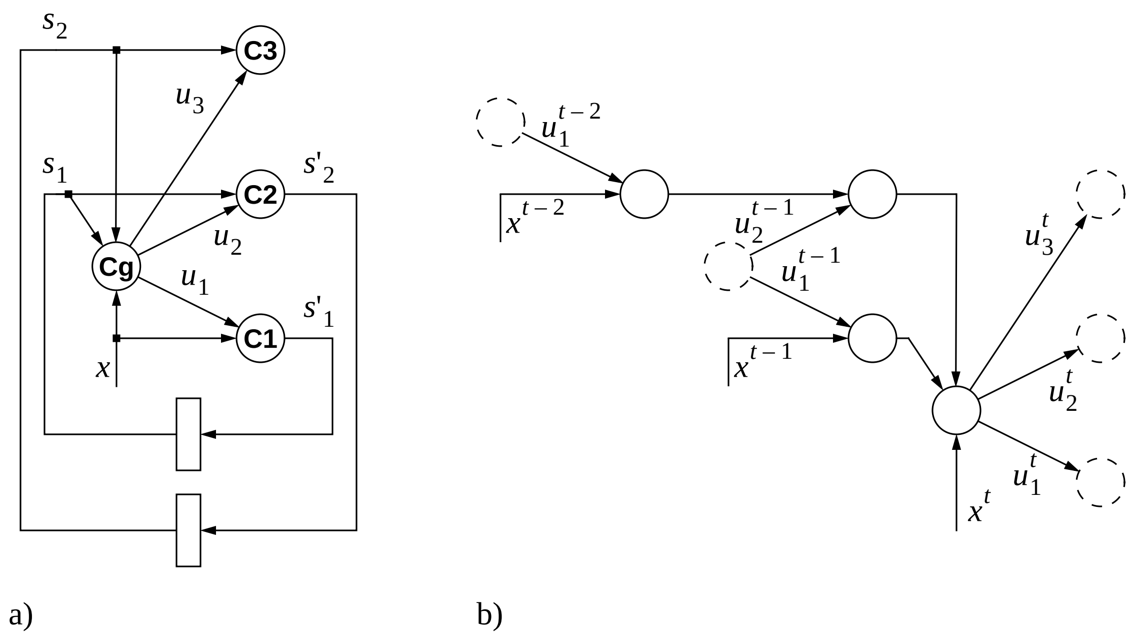


Figure 4.14: Pipelined controller: a) circuit b) pipeframe-based transition system

DPTRACE computes an initial path selection and corresponding set of path objectives (step 2). CTRLJUST makes decisions on CPI, CTS and STS signals (step 3.8.9.2), guided by the objectives (step 3.8.9.1). These decisions are implied on three fronts. First, they are implied in the controller (step 3.2) where they affect the CPO, CTO and CTRL signals. Second, DPTRACE checks whether the updated CTRL signals are consistent with the current set of justification and propagation paths in the datapath (step 3.3). If there is consistency, no further action is required. Otherwise DPTRACE computes a new set of justification and propagation paths, taking into account the current state of the CTRL lines. The objectives on the CTRL lines are updated accordingly. Only if DPTRACE fails to derive a set of justification and propagation paths will DPTRACE cause TG to backtrack. The third aspect of implication involves invoking DPRELAX (step 3.4) to compute data values. Failure to converge will cause TG to backtrack. Actual relaxation is only performed if the path objectives are completely satisfied, that is, all required CTRL assignments have been justified in terms of CPI and CTS signals. Otherwise, DPRELAX returns with an undetermined status (neither success nor failure). Steps 3.6.1-3.6.4 are the

```
            TG

1.          status ← UNDETERMINED
2.          DPTRACE /* derive initial path objectives */
3.          while ( status = UNDETERMINED ) {
3.1             /* imply */
3.2             ctrlStatus ← CTRLJUST:Imply()
3.3             pathStatus ← DPTRACE()
3.4             valueStatus ← DPRELAX()
3.5             status ← Status(ctrlStatus, pathStatus, valueStatus)
3.6             if (status = CONFLICT) {
3.6.1               UndoImplications(currentDecision)
3.6.2               status ← UNDETERMINED
3.6.3               while (NoUntriedValuesLeft(currentDecision)) {
3.6.3.1                 UndoImplications(currentDecision)
3.6.3.2                 if (DecisionStackEmpty) {
3.6.3.2.1                   status ← FAILURE
3.6.3.2.2                   break /* out of inner while */
3.6.3.3                 }
3.6.3.4                 else {
3.6.3.4.1                   currentDecision ← Pop(decisionStack)
3.6.3.5                 }
3.6.4               }
3.6.5               if (status = UNDETERMINED) {
3.6.5.1                 SelectNextUntried(currentDecision )
3.6.6               }
3.7             }
3.8             else { /* status = UNDETERMINED */
3.8.7               if (reset state reached and all objectives satisfied) {
3.8.7.1                 status ← SUCCESS
3.8.8               }
3.8.9               else {
3.8.9.1                 currentObjective ← SelectObjective()
3.8.9.2                 currentDecision ← BackTrace(currentObjective)
3.8.9.3                 Push(currentDecision, decisionStack)
3.8.10              }
3.9             }
4.          }
```

Figure 4.15: Overall test generation algorithm

usual actions for backtracking. Step 3.8.7 checks whether the reset state has been reached. If so, and if all objectives are satisfied, we return successfully with a test.

To prevent the generation of cycles during justification, we check that the pipestate is not covered by the pipestates encountered in the current sequence.

4.8 Experiments

Test generator implementation

We have built a prototype implementation of the proposed test generation algorithm. The inputs to the test generator are 1) a high-level structural description of the datapath, 2) a synthesizable Verilog description of the controller, 3) an attribute file that identifies signals in the controller according to our pipeframe model, 4) a list of bus SSL errors in the datapath to target, 5) a BDD variable order.

During preprocessing of the controller, the next state and output functions are derived. These are internally represented by ordered binary decision diagrams (BDD's) [Brya86]. We use CUBDD, a BDD package from Colorado University [Some97] that is included in the VIS distribution [Bray96]. Subsequently, the secondary variables are eliminated and expressions for the CTRL and CTO signals in terms of CPI, CTS and STS signals are derived (See Equations 4.1 and 4.2).

Test vehicle

We use a version of the DLX microprocessor [Henn90] as a test vehicle. This implementation was studied earlier in Chapter 2 (DLX1 in Table 2.1 and Table 2.4), and Chapter 3, Section 3.8. This design implements 44 instructions, has a five-stage pipeline, static and dynamic branch prediction logic, and consists of 1552 lines of structural Verilog code, excluding the models for library modules such as adders and register-files, and not counting blank and comment lines. For the purpose of test generation we disabled the dynamic branch predication. A simplified schematic showing the data-control dichotomy is presented in Figure 4.16. Architectural state elements, such as register file (RF), instruction memory (IMEM), data memory (DMEM), interrupt address register (IAR), are modeled as primary inputs and outputs. Registers that are clocked each cycle are represented by lightly shaded boxes; those that have a hold-function are represented by

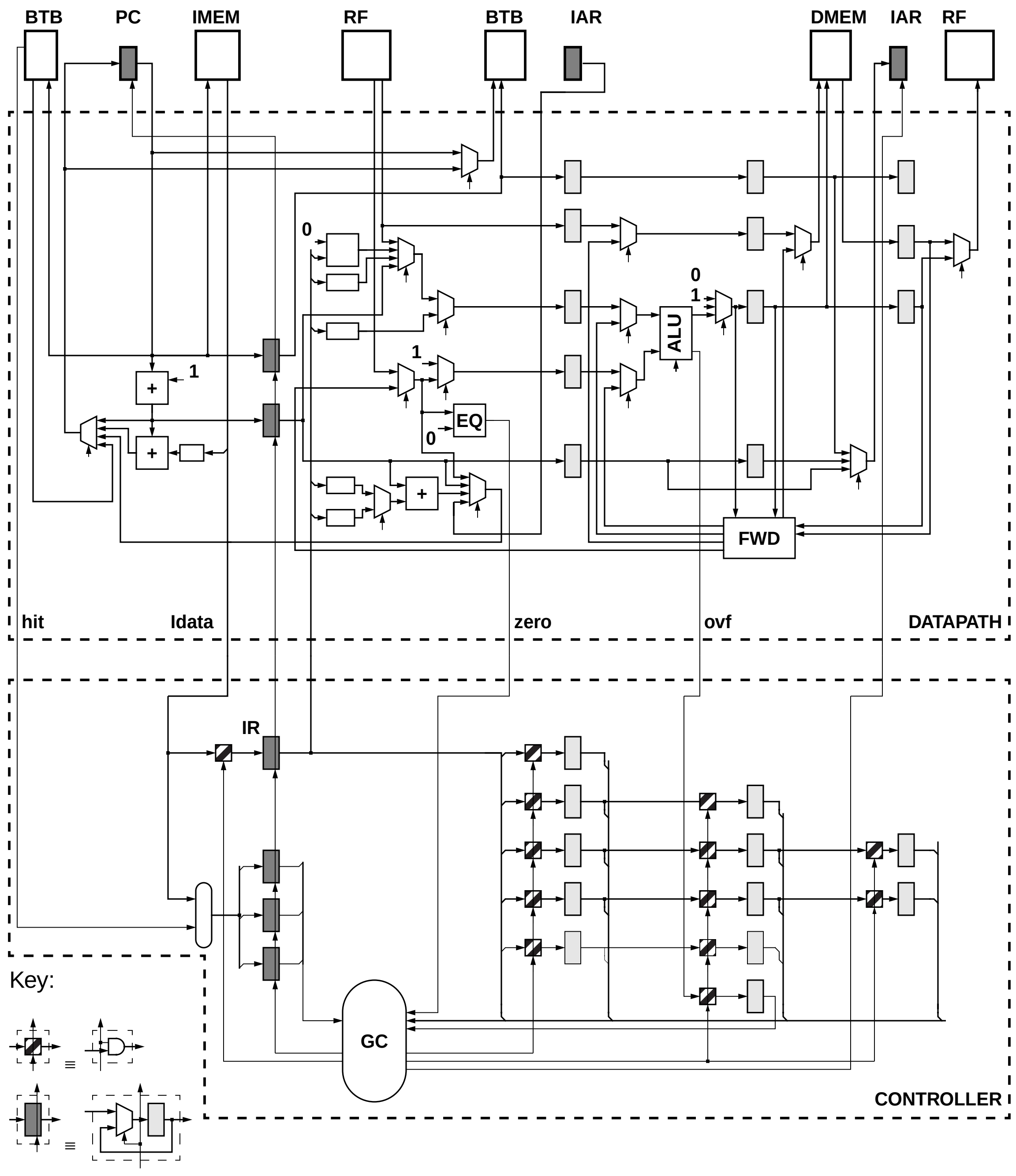


Figure 4.16: Simplified schematic of DLX implementation

darker boxes. Squashing logic is represented by hashed boxes; in this design the actual logic consists of an AND gate. The diagram also exposes the tertiary signals in the controller. These are the stall signals (to registers with a hold function), the squash signals (see squash logic), and the registers with a hold function. This design has three STS signals, which are also shown in the figure (*hit*, *zero*, *ovf*); the generation of the CTRL signals is not shown. The controller has 95 bits of state; the number of tertiary signals in the controller is 43. The pipeframe organization reduces the number of decision variables that need justification from 95 to 43 compared to the conventional timeframe organization. The datapath has 512 bits of state, not including those in the register file. The high-level model of the datapath consists of 100 combinational modules. Counts for each of the signal types defined by our model are given in Table 4.3. Note that most data signals (Dxx) have a bit-width greater than one, whereas control signals are single-bit signals.

We targeted our test generation system at all bus single stuck line (bus SSL) errors [Bhat85] in the decode, execute, memory and write-back stages of the datapath. Short-cuts in the implementation of the test generator have resulted in the inability to handle errors related to the program counter PC. These errors are mainly located in the fetch stage and are not considered in the experiments. Although our test generation algorithm can be used in conjunction with other error models, such as CSSL1, the bus SSL model was chosen for these initial experiments because it defines a number of error instances linear in the size of the circuit. The results are summarized in Table 4.4. A total of 316 errors were targeted; test generation succeeded for 87% of these errors. Typical sequences consist of slightly more than 11 cycles: 6 cycles to reset the machine followed by one or more non-trivial instructions, followed by 4 cycles. The overall algorithm performed only 50 backtracks for the successfully detected errors. Analysis of the 42 aborted errors showed that 8 of them are undetectable, 2 failed because the maximum number of backtracks in DPTRACE was exceeded, 14 errors require a non-sequential instruction stream (branches). The remaining 18 errors require error propagation through STS signals, which is not yet supported. The current implementation does not use error simulation, and much re-use of work by the algorithm has not yet been exploited. Therefore, we can expect that run times will significantly improve as these issues are addressed.

Table 4.3: Model parameters of DLX design

Parameter	Value
No. of DPI’s	6
No. of DPO’s	9
No. of DSI’s/DSO’s	16
No. of DTI’s/DTO’s	15
No. of CPI’s	32
No. of CPO’s	0
No. of CSI’s/CSO’s	95
No. of CTS’s/CTO’s	43
No. of STS signals	3
No. of CTRL signals	103
No. of comb. datapath modules	100

Table 4.4: High-level test generation for bus-SSL errors in DLX implementation

Parameter	Value
No. of errors	316
No. of errors detected	274
No. of errors aborted	42
Coverage	0.8671
No. of backtracks (detected errors only)	50
CPU time [minutes]	17

Table 4.5: Gate-level test generation for standard SSL errors using HITEC

Parameter	Value
No. of errors	385
No. of errors detected	278
No. of redundant errors	14
No. of errors aborted	93
Coverage	0.7221
Efficiency	0.7493
CPU time [minutes]	46

Table 4.6: Comparison of high-level and gate-level test generation for DLX

Parameter	HITEC	Our method	Manual
Total no. vectors	207	2,893	11,937
Total no. of errors	385	385	385
No. of errors detected	278	332	355
No. of errors not detected	93	53	30
Coverage	0.7221	0.8623	0.9221

To put our results in perspective, we also investigated gate-level test generation for this design. We synthesized a gate-level implementation, containing 10,117 gates and 640 flip-flops, for the same DLX version using Synopsys Design Compiler [Syn97]. For each bus SSL error in the high-level design, we selected SSL errors corresponding to bits 0, 15, 16 and 31. For some of the signals in the high-level design we were not able to identify corresponding signals in the gate-level design. We used HITEC [Nier91b] to generate tests; two passes with progressive time-out and abort-limits were run; the results are summarized in Table 4.5. The number of SSL errors in this table is after error collapsing, which is not the case for the number of bus SSL errors reported. Analysis of the gate-level test generation results revealed that HITEC has great difficulty generating tests for errors that require a sequence of instructions with register dependencies. Gate-level test generation succeeded for only one of a total of 14 forwarding paths, whereas our method generated tests for errors associated with 9 forwarding paths (2 paths are redundant, and 2 more require branches to exercise them). For further comparison, we error-simulated the test sequences obtained with our high-level test generator for the standard SSL errors. Note that these tests were targeted at bus SSL errors. The results are summarized in Table 4.6. As can be seen from the table, our method compares favorably with HITEC. Our test generator does not perform error simulation after generating a test sequence for an error; this explains why our method generated an order of magnitude more vectors than HIITEC. Finally, we also computed the SSL coverage obtained by a set of manually generated focussed tests. Again, these tests were not targeted at the SSL errors. Considering the 14 redundant errors that HITEC identified, the focussed tests detect 95.7% of the detectable errors. The lower coverage achieved by the test set generated using our method is primarily due to the absence of branch and jump instructions.

4.9 Conclusions

Test generation for synthetic errors is similar to test generation for SSL faults in a sequential circuit, which is known to be a very hard problem. To cope with this complexity, we focus on a limited, but important, class of pipelined microprocessors. Domain-specific knowledge can then be incorporated in the test generation algorithm. For this purpose, we have introduced a model that exposes high-level knowledge about pipeline structure. We have developed a high-level test generation algorithm that has the following features: 1) ‘pipeframe-based’ iterative organization, which we have shown to reduce the decision space and avoid many conflicts; 2) integration of high-level treatment of the datapath with fully detailed treatment of the controller; 3) separation of path and value selection.

Our test generation experiments with a DLX implementation show that our method can generate test sequences that achieve higher coverage than those produced by a powerful gate-level test generator.

The current implementation of our test generation method does not implement the following aspects:

- Error effects are not propagated through the controller. This results in failure to detect some errors.
- Only bus-SSL errors in the datapath are targeted. Extensions to different error models are straightforward; extensions for errors in the controller are not. This is also a limitation of Lee and Patel’s work.
- The current implementation does not perform error simulation. Including an error simulator will improve run times and result in smaller test sets.

The limitations of our method are as follows:

- The method is incomplete. It cannot prove errors redundant. It is common for high-level test generators to sacrifice completeness for sake of efficiency.
- No support is provided to efficiently handle memory arrays that are not part of the ISA. A branch target buffer is an example, and although it can be ignored during test generation by forcing it to always ‘miss,’ this is not desirable as it makes some errors undetectable.

CHAPTER 5
Conclusions

This chapter summarizes our contributions and suggests directions for future research.

5.1 Contributions

Functional design verification is one of the most serious bottlenecks in modern microprocessor design. Most present-day simulation-based methods use biased-random test generation in combination with a variety of coverage measures. A different, but previously little studied, approach is to construct test sets targeted at specific design errors. We have studied this approach, which involves modeling design errors and generating functional vectors for modeled errors using methods adapted from physical fault testing techniques.

The major contributions of this thesis are summarized below.

- A systematic method for collecting design error data.
- Design error data statistics collected from design projects at the university.
- An evaluation of the value to design verification of a number of design error models.
- A novel class of conditional error models.
- An efficient error simulation algorithm, called CESIM, for conditional errors.
- A high-level test generation method for a class of pipelined microprocessors. Its key features are its ‘pipeframe-based’ iterative organization, which exploits high-level knowledge about pipeline structure, and the integration high-level treatment of the datapath with fully detailed treatment of the controller.
- An experiment that compares our high-level test generation method with a gate-level method.

5.2 Future work

We suggest three main directions for future work; they concern 1) design error data, 2) error models, and 3) high-level test generation.

Design error data collection

Comparison of different methods for functional design verification is difficult. In an ideal situation, this could be accomplished by applying two competing methods to verify and debug the same design. For a given amount of time, the preferred method is the one that achieves the highest functional quality of the design. The latter itself is hard to quantify. A meaningful standard would be to run a large collection of real-life workloads on the debugged designs (obviously, running these workloads would have to require much more time than allowed to do debug the design); functional quality would then be defined as the fraction of correctly run workloads. It is clear that an ideal comparison requires a large effort. Furthermore, practical verification methods are typically specific to the design being verified.

The experiments we have described in Section 3.8 of Chapter 3 attempt to approximate the ideal comparison. They require recording the state of the design, and the uncovered errors, throughout the debug process of an actual design project. The competing verification methods can then be applied to the unverified design, and the functional quality can be more easily defined as the fraction of uncoverd actual design errors. One problem is that the set of actual design errors would have to be kept confidential.

We suggest that during future design projects undertaken at universities design errors are systematically recorded. Our methodology and experience with collecting design errors, reported in Chapter 2, together with industrial experience, can serve as a good starting point. The verification community would greatly benefit from such data.

Error modeling

We have shown that the CSSL1 error model is well suited for error-directed design verification. However, the quadratic (in the size of the circuit) number of error instances limits its practical use. One way to address this, is to use the design hierarchy to define restrictions on the basic error and condition components of CSSL1 errors. Further work is required to study the effect of such restrictions on coverage.

High-level test generation

High-level test generation is very useful, not just in the context of our error-directed verification approach. A test generator can also serve as a powerful debugging and diagnosis tool. For example, a designer may suspect that a certain local behavior of part of the design is incorrect, but he may not be sure if this behavior can actually be excited during normal operation. A test generator could be used to generate a sequence that excites the behavior and exposes it with respect to the ISA, if such a sequence exists.

This thesis has shown that domain-specific test generation is a promising way to handle larger designs than those that can be handled by general methods. We have studied a class of pipelined microprocessors. State-of-the-art microprocessors incorporate many complicated micro-architectural features, such as, multiple pipelines (superscalar), out-of-order completion, and register-renaming, to name just a few. A common element in these implementations is the presence of memory arrays that are not part of the ISA specification. Effective test generation for such designs will require the use of high-level models for these structures that hide the underlying individual memory elements but still allow the complete functionality to be exercised. The need for such models has also been realized by researchers in different areas. For example, Velev et al. have recently developed models for array memories of symbolic simulation [Vele97].

In summary, the functional design verification approach that we have developed in this thesis has proven to be a valuable addition to the range of simulation-based methods already available.

APPENDICES

APPENDIX A
Relationship between CSSL1 errors and bridging faults

An important parameter of an IC manufacturing process is the minimal allowed spacing between metal lines. The minimal line spacing design rule guards against imperfections in the manufacturing process that may lead to shorts between normally unconnected signals. However, integration density is typically very sensitive to the minimal spacing rule. Consequently, the minimal line spacing is set very close to what is achievable by process control. Therefore, despite the minimal spacing rule, shorts are still a common manufacturing defect. A fault model specifically targeted at these defects is the *bridging fault* (BF) model [Abra90]. The BF model has some similarities with the CSSL1 error model, and in this appendix we study the relationship between them.

A BF between two lines x and y is denoted by (x, y), and causes the fanouts of x an y to assume the same value, denoted by $Z(x, y)$. $Z(x, y)$ is a function that has the property that $Z(a,a) = a$. Figure A.1 illustrates the concept. If there exists a combinational path between x and y, then a BF (x,y) is called a *feedback bridging fault* (FBF), otherwise (x,y) is called a *nonfeedback bridging fault* (NFBF). FBF’s transform a combinational circuit into a sequential one. In this appendix we only consider NFBF’s. This restriction is in concert with our definition of CE’s (see Chapter 3, Section 3.2) that stipulates that the signals

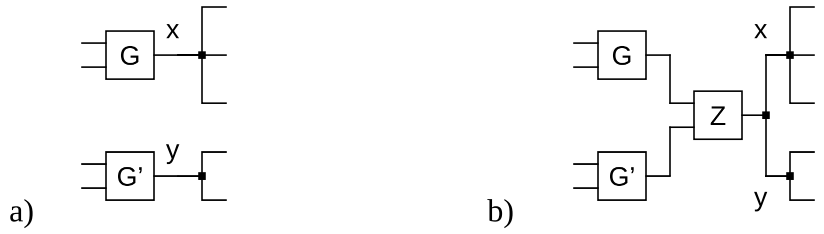


Figure A.1: BF (x, y): a) Fault-free circuit, b) faulty circuit

Table A.1: Bridging functions $Z(x, y)$

$x\,y$	Z_1	Z_2	Z_3	Z_4
0 0	0	0	0	0
0 1	0	1	0	1
1 0	0	1	1	0
1 1	1	1	1	1

appearing in the condition part are not to be part of the transitive combinational fanout of the basic error part.

If we restrict the domain and co-domain of $Z(x, y)$ to binary values, then there are four possible functions Z; they are listed in Table A.1. BF's with $Z = Z_1$, and $Z = Z_2$ are the most commonly used BF's; they are referred to as AND BF's, and OR BF's, respectively.

We derive the following results:

- (x, y) with $Z = Z_1$ is equivalent to the multiple CSSL1 error $(x = 0, y / 0)$, $(y = 0, x / 0)$.
- Similarly, (x, y) with $Z = Z_2$ is equivalent to the multiple CSSL1 error $(x = 1, y / 1)$, $(y = 1, x / 1)$.
- (x, y) with $Z = Z_3$ is equivalent to the signal source error y replaced by x, but this cannot be accurately modeled by (multiple) CSSL1 errors. However, (x, y) with $Z = Z_3$ dominates both $(x = 0, y / 0)$, and $(x = 1, y / 1)$. In other words, any test that detects $(x = 0, y / 0)$ or $(x = 1, y / 1)$ will also detect (x, y) with $Z = Z_3$. This can be seen as follows: To detect $(x = 0, y / 0)$, a test has to set x to 0, y to 1 and to sensitize y. Note that for input vectors that set x to 0 and y to 1, the faulty/erroneous circuits defined by the BF and the CSSL1 operate identically. Therefore, we conclude that the error detection conditions for $(x = 0, y / 0)$ are identical to those for the BF. A similar argument holds for $(x = 1, y / 1)$. The dominance relationship may not hold for sequential circuits.
- (x, y) with $Z = Z_4$ dominates both $(x = 0, y / 0)$, and $(x = 1, y / 1)$.

APPENDIX B
Conditional error simulation on ISCAS 89 benchmarks

This appendix describes an experiment, using the ISCAS’89 benchmarks, to measure the effectiveness of our error simulation algorithm CESIM, which is presented in Section 3.5.

We generated test sequences for SSL faults using HITEC [Nier90, Nier91a]. The parameters that determine when to abort a fault were set as follows: the backtrack limit was set to 10,000; the state backtrack limit was set to 10,000; the time limit per fault was set to 2 seconds. We separately fault simulated the obtained test sequences using PROOFS [Nier90, Nier91a]. Test generation and fault simulation were performed on a Fujitsu HALStation/300; the results are summarized in Table B.1. We did not try to improve fault coverage further by increasing the abort limits.

We error-simulated the same test sequences using CESIM for CSSL0 and CSSL1 errors. The error list for CSSL0 errors is identical to the collapsed SSL fault list from before. The CSSL1 error list was constructed as follows. For each CSSL0 error, we considered a maximum of 500 lines to condition the error. The smaller circuits have less than 500 lines, so every line in the circuit is considered as the condition line. This leads to a maximum of 1000 CSSL1 errors per CSSL0 error. However, some CSSL1 errors are rejected because their condition is part of the transitive fanout of the error site. The results of error simulation using CESIM are summarized in Table B.2. In the analysis that follows, we exclude benchmarks for which the test sequence achieves an SSL coverage less than 0.1.

Comparing the results of CSSL0 error simulation using CESIM in Table B.2 with the results of SSL fault simulation using PROOFS in Table B.1, we conclude the following. Fault/error coverages for corresponding circuits are identical, as expected. We can deduce that CESIM is on average 4.5 times slower than PROOFS. The actual slowdown varies between 0.3 and 7.7; for the largest benchmark it is 5.6. CESIM does not include any optimizations that are specific to CSSL0 errors.

The last two columns in Table B.2 compare error simulation for CSSL1 errors to error simulation for CSSL0 errors. We observe that the ratio of coverage of CSSL1 errors to coverage of CSSL0 varies between 0.58 and 0.94; its average is 0.76. We also computed the ratio of CPU time per error for CSSL1 errors to CPU time per error for CSSL0 errors. This ratio attempts to measure the speedup of CESIM for CSSL1 errors compared to a naive approach. We observe an average speedup varies between 2 and 78, its average is 34, and for the largest benchmark a speedup of 43 was obtained. This speedup can be displayed graphically by plotting the CPU time per test vector versus the number of errors, as in Figure B.1 (s27 was dropped because of its size). The figure also shows least-square fits for the data. The improved performance of our error simulation algorithm is achieved by exploiting the close relationship between CSSL1 errors derived from the same CSSL0 error. The execution time of CSSL0 error simulation with CESIM is dominated by event-driven simulation of faulty circuits. However, when simulating for CSSL1 errors, checking whether the condition of each CSSL1 error holds dominates the execution time. Least-square analysis shows that the CPU time per test vector is proportional to the number of CSSL0 errors to the power 1.33. The superlinear behavior reflects that those data points with a larger number of CSSL0 errors correspond to larger circuits, and hence the execution time of each event-driven simulation increases. For the CSSL1 execution time we find an exponent of 1.13. The almost linear behavior is a consequence of checking CSSL1 conditions, which is independent of the size of the circuit, dominating the execution time.

The experiments demonstrate that error simulation with CSSL1 errors is practical for moderately sized circuits. However, as the run time of our algorithm is linear in the number of errors, for very large circuits the quadratic number of CSSL1 errors becomes prohibitive. For those circuits restrictions on the general CSSL1 model may be appropriate. For example, when deriving CSSL1 errors from a given CSSL0 error, one

Table B.1: Test generation and fault simulation of ISCAS’89 circuits using HITEC

Circuit	Detected faults	Redundant faults	Aborted faults	Vectors	Efficiency	Coverage	HITEC CPU[s]	PROOFS CPU[s]
s27	32	0	0	21	1.0000	1.0000	0.12	0.03
s208.1	18	146	53	12	0.7558	0.0829	164.07	0.07
s298	265	26	17	220	0.9448	0.8604	48.18	0.25
s344	321	9	12	89	0.9649	0.9386	32.38	0.15
s349	329	11	10	106	0.9714	0.9400	31.57	0.17
s382	281	2	116	881	0.7093	0.7043	267.70	1.45
s386	314	70	0	273	1.0000	0.8177	5.52	0.22
s400	331	9	86	1,228	0.7981	0.7770	215.72	1.20
s420.1	28	151	276	20	0.3934	0.0615	635.98	0.15
s444	254	16	204	316	0.5696	0.5359	777.90	0.88
s526	51	14	490	34	0.1171	0.0919	1,167.25	0.15
s526n	55	13	485	37	0.1230	0.0995	1,162.57	0.18
s641	404	63	0	203	1.0000	0.8651	3.97	0.35
s713	476	105	0	196	1.0000	0.8193	5.55	0.40
s820	811	29	10	940	0.9882	0.9541	96.17	1.52
s832	813	46	11	962	0.9874	0.9345	99.63	1.78
s838.1	48	515	368	28	0.6047	0.0516	849.52	0.40
s953	89	990	0	14	1.0000	0.0825	38.68	0.28
s1196	1,239	3	0	439	1.0000	0.9976	2.95	0.80
s1238	1,283	72	0	472	1.0000	0.9469	4.77	0.97
s1423	578	11	926	89	0.3888	0.3815	2,100.88	0.85
s1488	1,368	20	98	778	0.9341	0.9206	325.48	2.30
s1494	1,447	32	27	991	0.9821	0.9608	118.78	2.93
s5378	3,146	159	1,298	894	0.7180	0.6835	2,941.93	14.68
s9234	18	3,916	0	6	1.0000	0.0046	1.38	1.05
s9234.1	366	181	3,391	38	0.1389	0.0929	7,562.58	3.62
s13207	557	8,218	672	76	0.9289	0.0590	2,266.77	16.08
s13207.1	858	7,583	1,148	106	0.8803	0.0895	3,475.12	28.25
s15850	85	11,407	21	8	0.9982	0.0074	60.15	2.97
s15850.1	4,374	1,229	5,920	2,493	0.4862	0.3796	16,302.12	383.82
s35932	34,868	3,984	242	300	0.9938	0.8919	2,350.30	45.47
s38417	1,088	356	29,016	51	0.0474	0.0357	96,335.17	46.05
s38584	7,798	6,759	21,744	1,593	0.4010	0.2148	54,293.70	1,293.65
s38584.1	20,589	1,948	13,766	4,383	0.6208	0.5671	38,090.13	1,535.40

could restrict the condition signals to those signals appearing in the same hierarchical module as the CSSL0 error.

Table B.2: Error simulation of ISCAS’89 circuits using CESIM

Circuit	Vectors	CSSL0 No.	CSSL0 Coverage	CSSL0 CPU [s]	CSSL1 No.	CSSL1 Coverage	CSSL1 CPU [s]	$\frac{\text{cover.}_1}{\text{cover.}_0}$	$\frac{\left(\frac{\text{CPU}}{\text{no.}}\right)_0}{\left(\frac{\text{CPU}}{\text{no.}}\right)_1}$
s27	21	32	1.0000	0.01	664	0.7289	0.10	0.7289	2.08
s208.1	12	217	0.0829	0.11	44,492	0.0228	0.67	0.2750	33.66
s298	220	308	0.8604	0.63	75,468	0.7103	7.80	0.8255	19.79
s344	89	342	0.9386	0.39	114,294	0.7598	5.09	0.8095	25.61
s349	106	350	0.9400	0.32	117,484	0.7796	5.20	0.8294	20.66
s382	881	399	0.7043	5.61	130,656	0.5830	60.35	0.8278	30.44
s386	273	384	0.8177	1.30	93,060	0.6040	12.60	0.7387	25.00
s400	1,228	426	0.7770	7.77	144,628	0.6275	74.47	0.8076	35.42
s420.1	20	455	0.0615	0.28	199,616	0.0180	3.60	0.2927	34.12
s444	316	474	0.5359	3.44	175,616	0.4185	42.15	0.7809	30.24
s526	34	555	0.0919	0.65	218,816	0.0656	7.15	0.7138	35.84
s526n	37	553	0.0995	0.69	219,166	0.0704	7.54	0.7075	36.27
s641	203	467	0.8651	1.31	349,832	0.7219	24.87	0.8345	39.46
s713	196	581	0.8193	1.39	452,256	0.6742	32.78	0.8229	33.01
s820	940	850	0.9541	6.30	442,758	0.6456	160.27	0.6767	20.48
s832	962	870	0.9345	6.72	445,274	0.6355	165.86	0.6800	20.74
s838.1	28	931	0.0516	1.02	821,774	0.0158	17.72	0.3062	50.81
s953	14	1,079	0.0825	0.63	879,540	0.0321	12.95	0.3891	39.66
s1196	439	1,242	0.9976	5.75	1,058,844	0.7483	151.78	0.7501	32.30
s1238	472	1,355	0.9469	7.49	1,140,402	0.6969	186.76	0.7360	33.75
s1423	89	1,515	0.3815	5.44	1,287,036	0.2695	72.08	0.7064	64.12
s1488	778	1,486	0.9206	10.18	1,142,374	0.6615	272.48	0.7186	28.72
s1494	991	1,506	0.9608	10.74	1,151,208	0.6993	327.06	0.7278	25.10
s5378	894	4,603	0.6835	94.80	4,517,276	0.5418	1,753.33	0.7927	53.06
s9234	6	3,934	0.0046	2.11	3,850,188	0.0004	43.25	0.0870	47.75
s9234.1	38	3,938	0.0929	21.70	3,854,738	0.0490	147.79	0.5274	143.73
s13207	76	9,447	0.0590	77.66	9,325,456	0.0132	638.82	0.2237	120.00
s13207.1	106	9,589	0.0895	115.96	9,453,928	0.0363	1,167.80	0.4056	97.90
s15850	8	11,513	0.0074	11.92	11,197,872	0.0021	151.72	0.2838	76.42
s15850.1	2,493	11,523	0.3796	2735.14	11,186,886	0.2574	33,994.54	0.6781	78.11
s35932	300	39,094	0.8919	170.70	38,708,548	0.8400	4,541.98	0.9418	37.21
s38417	51	30,460	0.0357	249.88	30,222,794	0.0109	1,981.77	0.3053	125.11
s38584	1,593	36,301	0.2148	6233.05	36,124,976	0.1244	122,142.85	0.5791	50.78
s38584.1	4,383	36,303	0.5671	8,656.32	36,124,758	0.4423	199,346.07	0.7799	43.21

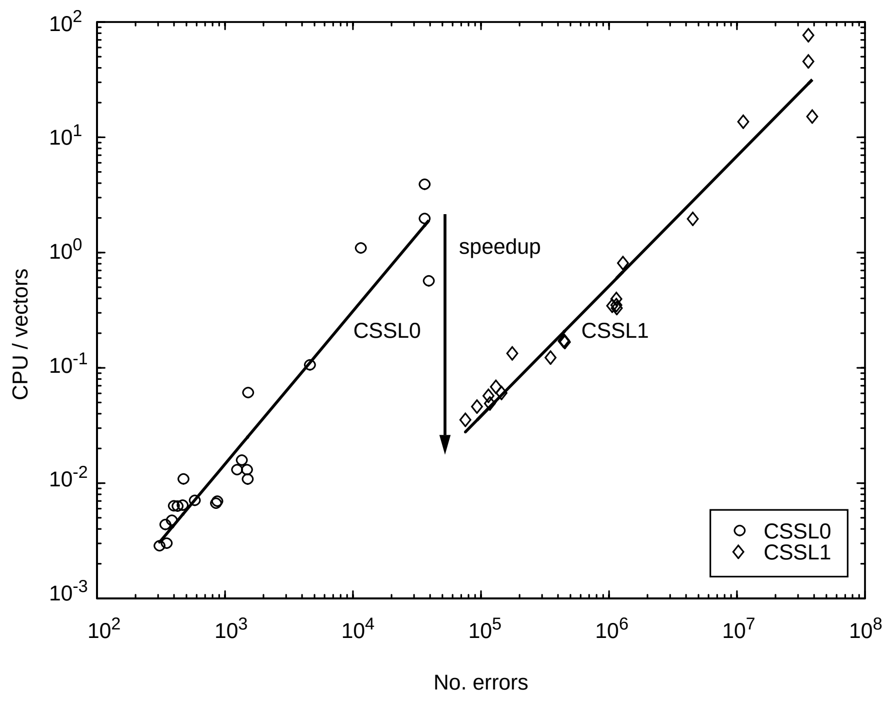


Figure B.1: Run time analysis of CESIM on ISCAS’89 benchmarks

BIBLIOGRAPHY

BIBLIOGRAPHY

[0In] 0-In Design Automation, http://www.0-in.com. *Methodology overview*.

[AA95] H. Al-Asaad and J. P. Hayes. Design verification via simulation and automatic test pattern generation. In *Proc. Int. Conf. Computer-Aided Design*, pages 174–180, 1995.

[AA98] H. S. Al-Asaad. *Lifetime validation of digital systems via fault modeling and test generation*. PhD thesis, University of Michigan, 1998.

[Abad88] M. S. Abadir, J. Ferguson, and T. E. Kirkland. Logic design verification via test generation. *IEEE Trans. Computer-Aided Design*, 7(1):138–148, January 1988.

[Abra90] M. Abramovici. *Digital systems testing and testable design*. Computer Science Press, New York, 1990.

[AH96] G. Al Hayek and C. Robach. From specification validation to hardware testing: A unified method. In *Proc. IEEE Int. Test Conf.*, pages 885–893, 1996.

[Ahar91] A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, and V. Schwartzburd. Verification of the IBM RISC System/6000 by dynamic biased pseudo-random test program generator. *IBM Systems Journal*, pages 527–538, 1991.

[Ahar95] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek. Test program generation for functional verification of PowerPC processors in IBM. In *Proc. Design Automation Conf.*, pages 279–285, 1995.

[Alex96] T. B. Alexander, K. A. Dickey, D. N. Goldberg, R. V. La Fetra, J. R. McGee, N. Noordeen, , and A. Prakash. Verification, characterization, and debugging of the HP PA 7200 processor. *Hewlett-Packard Journal*, pages 1–12, February 1996.

[Bass95] M. Bass, T. W. Blanchard, D. D. Josephson, D. Weir, and D. L. Halperin. Design methodologies for the PA 7100LC microprocessor. *Hewlett-Packard Journal*, pages 23–35, April 1995.

[Baum98] A. J. Baum and A. J. Smith. Hot chips – hot suff. *IEEE Micro*, pages 11–13, March/April 1998.

[Beer96] I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: An industry-oriented formal verification tool. In *Proc. Design Automation Conf.*, pages 655–660, 1996.

[Beiz90] B. Beizer. *Software testing techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.

[Beiz95] B. Beizer. The Pentium bug – an industry watershed. *Testing Techniques Newsletter (TTN), TTN Online Edition*, September 1995.

[Bent97] B. Bentley. Personal Communication, 1997.

[Bhag94] V. Bhagwati and S. Devadas. Automatic verification of pipelined microprocessors. In *Proc. Design Automation Conf.*, pages 603–608, 1994.

[Bhat85] D. Bhattacharya and J. P. Hayes. High-level test generation using bus faults. In *Digest of Papers - FTCS 15, Fifteenth Annual International Symposium on Fault-Tolerant Computing*, pages 65–70, 1985.

[Bisc97] G. P. Bischoff, K. S. Brace, S. Jain, and R. Razdan. Formal implementation verification of the bus interface unit for the Alpha 21264 microprocessor. In *Proc. Int. Conf. Computer Design*, pages 16–24, 1997.

[Bose98] P. Bose and T. M. Conte. Performance analysis and its impact on design. *Computer,* pages 41–49, 1998.

[Bray96] R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, T. Villa, R. Alur, and T.A. Henzinger. VIS: a system for verification and synthesis. In *Proc. Int. Conf. on Computer-Aided Verification*, pages 428–432, 1996.

[Brow96] R. B. Brown, T. D. Basso, P. N. Parakh, S. M. Gold, C. R. Gauthier, R. J. Lomax, and T. N. Mudge. Complementary GaAs technology for a GHz microprocessor. In *Proc. GaAs IC Symposium*, 1996.

[Brya86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, pages 677–691, August 1986.

[Burc94] J.R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification,* pages 68–80, June 1994.

[Burg97] D. Burger and T. M. Austin. The SimpleScalar tool set. *Computer Architecture News*, 25(3), June 1997.

[Cad94] Cadence Design Systems. *Verilog-XL Reference Manual*, December 1994.

[Casa96] F. Casaubieilh, A. McIsaac, M. Benjamin, M. Bartley, F. Pogodalla, F. Rocheteau, M. Belhadj, J. Eggleton, G. Mas, G. Barrett, and C. Berthet. Functional verification methodology of Chameleon processor. In *Proc. Design Automation Conf.*, pages 421–426, 1996.

[Cede93] P. Cederqvist et al. Version management with CVS. Signum Support AB, 1993.

[Chan94] A.K. Chandra, V.S. Iyengar, R.V. Jawalekar, M.P. Mullen, I. Nair, and B.K. Rosen. Architectural verification of processors using symbolic instruction graphs. In *IEEE Int. Conference on Computer Design VLSI in Computers and Processors,* pages 454–459, 1994.

[Chan95] A.K. Chandra, V.S. Iyengar, D. Jameson, R. Jawalekar, I. Nair, B.K. Rosen, M.P. Mullen, J. Yoon, R. Armoni, D. Geist, and Y. Wolfsthal. AVPGEN - a test generator for architecture verification. *IEEE Trans. on VLSI*, pages 188–200, 1995.

[Chen96] K.-T. Cheng. Gate-level test generation for sequential circuits. *ACM Trans. Design Automation of Electronic Systems*, 1(4):405–442, October 1996.

[Clar96] E. M. Clarke and R. P. Kurshan. Computer-aided verification. *IEEE Spectrum*, pages 61–67, June 1996.

[Clar98] P. Clarke. EDA software – code coverage for state machines. *EE Times*, (1012), 1998.

[Cohn87] A. Cohn. A proof of correctness of the VIPER microprocessor: the first level. In G. Birtwistle and P. A. Sybrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer, 1987.

[Cohn89] A. Cohn. Correctness properties of the VIPER block model: the second level. In G. Birtwistle and P. A. Sybrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 1–91. Springer Verlag, 1989.

[DeMi78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, pages 34–41, April 1978.

[Deva96] S. Devadas, A. Ghosh, and K. Keutzer. Observability-based code coverage metric for functional simulation. In *Proc. Int. Conf. Computer-Aided Design*, pages 418–425, 1996.

[Dohm98] N. Dohm, C. Ramey, D. Brown, S. Hildebrandt, J. Huggins, M. Quinn, and S. Taylor. Zen and the art of Alpha verification. In *Proc. Int. Conf. Computer Design*, pages 111–117, 1998.

[Eco96] A survey of the world economy: The hitchhiker's guide to cybernomics - the hitchhiker's guide to cybernomics. *The Economist*, September 28 1996.

[EET94] Chrysalis aims tools at commercial applications:DAC to see formal 'insight'. *EE Times*, (800), June 1994.

[Fall98a] F. Fallah, S. Devadas, and K. Keutzer. OCCOM: Efficient computation of observability-based code coverage metric for functional simulation. In *Proc. Design Automation Conf.*, pages 152–157, 1998.

[Fall98b] F. Fallah, S. Devadas, and K. Keutzer. OCCOM: Efficient computation of observability-based code coverage metric for functional simulation. In *Proc. Design Automation Conf.*, pages 152–157, 1998.

[Gana96] G. Ganapathy, R. Narayan, G. Jorden, D. Fernandez, M. Wang, and J. Nishimura. Hardware emulation for functional verification of K5. In *Proc. Design Automation Conf.*, pages 315–318, 1996.

[Geis96] D Geist, M Farkas, A Landver, Y Lichtenstein, S Ur, Y Wolfsthal, M Srivas, and A Camilleri. Coverage-directed test generation using symbolic techniques. In *Proc. Int. Conf. Formal methods in Computer-Aided Design*, pages 143–158, 1996.

[Ghos91] A. Ghosh, S. Devadas, and A. R. Newton. Test generation and verification for higly sequential circuits. *IEEE Trans. Computer-Aided Design*, 10(5):652–667, May 1991.

[Goel81] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Trans. Computers*, C-30(3):215–222, March 1981.

[Goer95] R. Goering. Chrysalis expands tool to prove equivalency – verification gets upgrade. *EE Times*, (873), November 1995.

[Goer97] R. Goering. Model checker leads Lucent's commercial-EDA push – Bell Labs goes formal with design verification. *EE Times*, (948), April 7 1997.

[Gold79] L. H. Goldstein. Controllability/observability analysis of digital circuits. *IEEE Trans. Circuits and Systems*, pages 685–693, September 1979.

[Gupt97] A. Gupta, S. Malik, and P. Ashar. Toward formalizing a validation methodology using simulation coverage. In *Proc. Design Automation Conf.*, pages 740–745, 1997.

[Ham94] Hamarsoft, Heerlen, The Netherlands. *Hamarsoft's 86BUGS list*, 4 edition, November 1994. Available from http://www.xs4all.nl/~feldmann.

[Hans95a] M. C. Hansen and J. P. Hayes. High-level test generation using physically-induced faults. In *Proc. IEEE VLSI Test Symp.*, pages 20–28, 1995.

[Hans95b] M. C. Hansen and J. P. Hayes. High-level test generation using symbolic scheduling. In *Proc. IEEE Int. Test Conf.*, pages 586–595, 1995.

[Hard96] R. H. Hardin, Z. Har'El, and R. P. Kurshan. Cospan. In *Proc. Int. Conf. on Computer-Aided Verification*, pages 423–427, July 1996.

[Henn90] J. Hennessy and D. Patterson. *Computer Architecture: A quantitative Approach*. Morgan Kaufman Publishers, San Mateo, Calif., 1990.

[Ho95] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill. Architecture validation for processors. In *Proc. Int. Symp. Computer Architecture*, pages 404–413, 1995.

[Ho96a] C.-M. R. Ho. *Validation tools for complex digital designs*. PhD thesis, Stanford University, 1996.

[Ho96b] R. C. Ho and M. A. Horowitz. Validation coverage analysis for complex digital designs. In *Proc. Int. Conf. Computer-Aided Design*, pages 146–151, 1996.

[Hosk95] Y. V. Hoskote. *Formal techniques for verification of synchronous sequential circuits*. PhD thesis, The University of Texas at Austin, 1995.

[Hoss96] A. Hosseini, D. Mavroidis, and P. Konas. Code generation and analysis for the functional verification of microprocessors. In *Proc. Design Automation Conf.*, pages 305–310, 1996.

[IEEE88] IEEE Standards Board. IEEE standard 1076-1987 VHDL language reference manual. New York, NY 10017, 1988.

[IEEE94] IEEE Standards Board. IEEE standard 1175-1994 standard reference model for computing system tool interconnections. New York, NY, 1994.

[IEEE96] IEEE Standards Board. IEEE standard 1364-1995 verilog hardware description language reference manual. New York, NY, 1996.

[Int89] Intel Corp. *8086/8088 User's manual. Programmer's and hardware reference manual*, 1989.

[Iwas94] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose. Automatic test program generation for pipelined processors. In *Proc. Int. Conf. Computer-Aided Design*, pages 580–583, 1994.

[John91] M. Johnson. *Superscalar microprocessor design*. Prentice Hall, Englewood Cliffs, N.J., 1991.

[Kang94] S. Kang and S. A. Szygenda. The simulation automation system SAS; concepts, implementations, and results. *IEEE Trans. on VLSI*, 1994.

[Kant96] M. Kantrowitz and L. M. Noack. I'm done simulating; now what? Verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor. In *Proc. Design Automation Conf.*, pages 325–330, 1996.

[Kels93] T. P. Kelsey, K. K. Saluja, and S. Y. Lee. An efficient algorithm for sequential circuit test generation. *IEEE Trans. Computers*, pages 1361–1371, November 1993.

[King91] K.N. King and A.J. Offutt. A Fortran language system for mutation-based software testing. *Software Practice and Experience*, 21(7):685–718, July 1991.

[Kogg77] P. M. Kogge. Microprogramming of pipelined processors. In *Proc. Ann. Symp. Comput. Archit.*, pages 63–69, 1977.

[Kuel97] A. Kuelmann and F. Krohm. Equivalence checking using cuts and heaps. In *Proc. Design Automation Conf.*, pages 263–268, June 1997.

[Kuma97] J. Kumar. Prototyping the M68060 for concurrent verification. *IEEE Design & Test of Computers*, pages 34–41, 1997.

[Kurs97] R.P. Kurshan. Formal verification in a commercial setting. In *Proc. Design Automation Conf.*, pages 258–262, June 1997.

[Kusk94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. Stanford FLASH multiprocessor. In *Proc. Int. Symp. Computer Architecture*, pages 302–313, 1994.

[Lee92a] J. Lee. *Architectural level test generation and fault simulation*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Lee92b] J. Lee and J. H. Patel. A signal-driven discrete relaxation technique for architectural level test generation. In *Proc. 1991 IEEE Int. Conf. on Computer-Aided Design - ICCAD-91*, pages 458–461, 1992.

[Lee94] J. Lee and J. H. Patel. Architectural level test generation for microprocessors. *IEEE Trans. Computer-Aided Design*, pages 1288–1300, 1994.

[Levi97] J. Levitt and K. Olukotun. Verifying correct pipeline implementation for microprocessors. In *Proc. Int. Conf. Computer-Aided Design*, pages 162–169, 1997.

[Lewi96] D. Lewin, D. Lorenz, and S. Ur. A methodology for processor implementation verification. In *Proc. Int. Conf. Formal methods in Computer-Aided Design*, pages 126–142, 1996.

[Malk98] Y. Malka and A. Ziv. Design reliability - estimation through statistical analysis of bug discovery data. In *Proc. Design Automation Conf.*, pages 644–649, 1998.

[Mall95] C. H. Malley and M. Dieudonne. Logic verification methodology for PowerPC microprocessors. In *Proc. Design Automation Conf.*, pages 234–240, 1995.

[Mang97] S. T. Mangelsdorf, R. P. Gratias, R. M. Blumberg, and R. Bhatia. Functional verification of the HP PA 8000 processor. *Hewlett-Packard Journal*, 48(4):22–31, August 1997.

[Marc96] T. E. Marchok, A. El-Maleh, W. Maly, and J. Rajski. Complexity analysis of sequential atpg. *IEEE Trans. Computer-Aided Design*, pages 1409–1422, November 1996.

[McFa93] M. C. McFarland. Formal verification of sequential hardware. a tutorial. *IEEE Trans. Computer-Aided Design*, pages 633–654, May 1993.

[McGe95] P. C. McGeer and A. Saldanha. Multivalued diagrams show the way. *EE Times*, (867), Sept. 25 1995.

[McMi93] K. L. McMillan. *Symbolic model checking*. Kluwer Academic, Boston, 1993.

[McMi94] K. L. McMillan. Fitting formal methods into the design cycle. In *Proc. Design Automation Conf.*, pages 314–319, 1994.

[Micz86] A. Miczo. *Digital logic testing and simulation*. Harper & Row, New York, 1986.

[MIP94] MIPS Technologies Inc. *MIPS R4000PC/SC Errata, Processor Revision 2.2 and 3.0*, May 1994.

[Mona96] J. Monaco, D. Holloway, and R. Raina. Functional verification methodology for the PowerPC 604(TM) microprocessor. In *Proc. Design Automation Conf.*, pages 319–324, 1996.

[Moun98] D. Moundanos, J. A. Abraham, and Y. V. Hoskote. Abstraction techniques for validation coverage analysis and test generation. *IEEE Trans. Computers*, 47(1):2–14, January 1998.

[Murr90] B. T. Murray and J. P. Hayes. Hierarchical test generation using precomputed tests for modules. *IEEE Trans. Computer-Aided Design*, pages 594–603, June 1990.

[Murr92] B. T. Murray and J. P. Hayes. Test propagation through modules and circuits. In *Proc. IEEE Int. Test Conf.*, pages 748–757, 1992.

[Nels97] K. L. Nelson, A. Jain, , and R. E. Bryant. Formal verification of a superscalar execution unit. In *Proc. Design Automation Conf.*, pages 161–166, 1997.

[Nier90] T. Niermann, W. T. Cheng, and J. H. Patel. PROOFS: A fast memorry efficient fault simulator for sequential circuits. In *Proc. Design Automation Conf.*, pages 190–196, 1990.

[Nier91a] T. Niermann. *Techniques for sequential circuit automatic test generation*. PhD thesis, University of Illinois, 1991.

[Nier91b] T. Niermann and J. H. Patel. HITEC: A test generation packaged for sequential circuits. In *Proc. European Design Automation Conf.*, pages 214–218, 1991.

[Offu96] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Software Engineering and Methodology*, pages 99–118, April 1996.

[Owre96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, M. Srivas, R. Alur, and T.A. Henzinger. PVS: Combining specification, proof checking and model checking. In *Proc. Int. Conf. on Computer-Aided Verification*, pages 411–414, 1996.

[Paln94] S. Palnitkar, P. Saggurti, and S.-H. Kuang. Finite state machine trace analysis program. In *Int. Verilog HDL Conf.*, pages 52–57, 1994.

[Paxs98] C. Paxson. Analyze the reliability of your software. *Java Report*, April 1998.

[Pope96] V. Popescu and B. McNamara. Innovative verification strategy reduces design cycle time for high-end SPARC processor. In *Proc. Design Automation Conf.*, pages 311–314, 1996.

[Post96a] Matt Postiff. *LC-2 Programmer's Reference Manual. Revision 3.1*. University of Michigan, 1996.

[Post96b] R. M. Poston. *Automatic Specification-Based Software Testing*. IEEE Computer Society Press, May 1996.

[Rat97] Rational Software. *Purify's user guide*, version 4.1 edition, 1997.

[Shep97] K. Shepard, S. Carey, E. Cho, B. Curran, R. Hatch, D. Hoffman, S. McCabe, G. Northrop, and R. Seigler. Design methodology for the s/390 parallel enterprise server g4 microprocessors. *IBM Journal of Research and Development*, pages 515–547, Jul.-Sep. 1997.

[Some97] F. Somenzi. CUDD: CU decision diagram package release 2.1.1. Technical report, Dept. of Electr. and Comp. Engineering, University of Colorado at Boulder, 1997.

[SP92] M. St. Pierre, S.-W. Yang, and D. Cassiday. Functional vlsi design verification methodology for the CM-5 massively parallel surpercomputer. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 430–435, 1992.

[SR] Software Research Inc., San Francisco, CA. *TestWorks/ Coverage for UNIX*. http://www.testworks.com.

[Syn97] Synopsys Inc. *Design Compiler Reference Manual: Fundamentals*, January 1997.

[Tayl98] S. Taylor, M. Quinn, D. Brown, N. Dohm, S. Hildebrandt, J. Huggins, and C. Ramey. Functional verification of a multiple-issue, out-of-order, superscalar Alpha processor - the DEC Alpha 21264 microprocessor. In *Proc. Design Automation Conf.*, pages 638–643, 1998.

[Upto94] M. Upton, T. Huff, T. Mudge, and R. Brown. Resource allocation in a high clock rate microprocessor. In *Proceedings sixth international conference on Architectural support for programming languages and operating systems*, pages 98–109, 1994.

[Upto97] M. Upton. *Architectural Trade-offs in a Latency Tolerant Gallium Arsenide Micrprocessor*. PhD thesis, University of Michigan, 1997.

[VC97] D. Van Campenhout and S. Raasch. Getting started with CVS. Internal Report, University of Michigan, 1997.

[VC98] D. Van Campenhout, H. Al-Asaad, J. P. Hayes, T. Mudge, and R. B. Brown. High-level design verification of microprocessors via error modeling. *ACM Trans. Design Automation of Electronic Systems*, 3(4):581–599, October 1998.

[Vele97] M. N. Velev, R. E. Bryant, and A. Jain. Efficient modeling of memory arrays in symbolic simulation. In O. Grumberg, editor, *Proc. Int. Conf. on Computer-Aided Verification*, LNCS 1254, pages 388–399. Springer-Verlag, June 1997.

[Weir97] D. Weir and P. G. Tobin. Verifying the correctness of the PA 7300LC processor. *Hewlett-Packard Journal*, 48(3):69–72, June 1997.

[Wile97] B. Wile, M. Mullen, C. Hanson, D. Bair, K. Lasko, P. Duffy, E. J. Kaminski, T. Gilbert, S. Licker, R. Sheldon, W. Wollyung, W. Lewis, and R. Adkins. Functional verification of the CMOS S/390 parallel enterprise server G4 system. *IBM Journal of Research and Development*, pages 549–566, July/September 1997.

[Wils99] R. Wilson and B. Fuller. Soaring mask costs roil fine-geometry ASICs. *EE Times*, March 26 1999.

[Wind95] P. J. Windley. Formal modeling and verification of microprocessors. *IEEE Trans. Computers*, pages 54–72, January 1995.

[Wolf98a] A. Wolfe. CPU clone-makers wrestle with X86. *EE Times*, 01/28 1998.

[Wolf98b] A. Wolfe. Merced grips Intel in verification vise. *EE Times*, (990), January 1998.

[Wood93] M.R. Woodward. Mutation testing–its origin and evolution. *Information-and-Software-Technology*, pages 163–9, 1993.

[X86] Intel’s secrets. x86 Monthly digest, http://www.x86.org.

[Yoel90] M. Yoeli, editor. *Formal Verification of Hardware Design*. IEEE Computer Society Press, Los Alamitos, Calif., 1990.