High-Level Design Verification of Microprocessors via Error Modeling

DAVID VAN CAMPENHOUT, HUSSAIN AL-ASAAD,
JOHN P. HAYES, TREVOR MUDGE, and RICHARD B. BROWN
University of Michigan, Ann Arbor, Michigan

A design verification methodology for microprocessor hardware based on modeling design errors and generating simulation vectors for the modeled errors via physical fault testing techniques is presented. We have systematically collected design error data from a number of microprocessor design projects. The error data is used to derive error models suitable for design verification testing. A class of basic error models is identified and shown to yield tests that provide good coverage of common error types. To improve coverage for more complex errors, a new class of conditional error models is introduced. An experiment to evaluate the effectiveness of our methodology is presented. Single actual design errors are injected into a correct design, and it is determined if the methodology will generate a test that detects the actual errors. The experiment has been conducted for two microprocessor designs and the results indicate that very high coverage of actual design errors can be obtained with test sets that are complete for a small number of synthetic error models.

Categories and Subject Descriptors: B.0 [**Hardware**] General —*Design Aids;* B.5.2 [**Hardware**] Register-transfer-level implementation —*Design Aids.*

General Terms: Verification, microprocessors.

Additional Key Words and Phrases: Design verification, design errors, error modeling.

1. INTRODUCTION

It is well known that about a third of the cost of developing a new microprocessor is devoted to hardware debugging and testing [25]. The inadequacy of existing hardware verification methods is graphically illustrated by the Pentium's FDIV error, which cost its manufacturer an estimated $500 million. The development of practical verification methodologies for hardware verification has long been handicapped by two related problems: (1) the

A preliminary version of this paper was presented in [4] at the 1997 IEEE International High Level Design Validation and Test Workshop, Oakland, California, November 14-15, 1997.

lack of published data on the nature, frequency, and severity of the design errors occurring in large-scale design projects; and (2) the absence of a verification methodology whose effectiveness can readily be quantified.

There are two broad approaches to hardware design verification: formal and simulation-based. Formal methods try to verify the correctness of a system by using mathematical proofs [32]. Such methods implicitly consider all possible behavior 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.

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

Hand-written test cases form the first line of defense against bugs, focusing on basic functionality and important corner (exceptional) cases. These tests are very effective in the beginning of the debug phase, but lose their usefulness later. Recently, tools have been developed to assist in the generation of focused tests [13,20]. Although these tools can significantly increase design productivity, they are far from being fully automated.

The most widely used method to generate verification tests automatically is random test generation. It provides a cheap way to take advantage of the billion-cycles-a-day simulation capacity of networked workstations available in many big design organizations. Sophisticated systems have been developed that are biased towards corner cases, thus improving the quality of the tests significantly [2]. Advances in simulator and emulator technology have enabled the use of very large sets as test stimuli such as existing application and system software. Successfully booting the operating system has become a basic quality requirement [17,25].

Common to all the test generation techniques mentioned above is that they are not targeted at specific design errors. This poses the problem of quantifying the effectiveness of a test set, such as the number of errors covered. Various coverage metrics have been proposed to address this problem. These include code coverage metrics from software testing [2,7,11], finite state machine coverage [20,22,28], architectural event coverage [22], and observability-based metrics [16]. A shortcoming of all these metrics is that the relationship between the metric and the detection of classes of design errors is not well understood.

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, as illustrated by Figure 1. For example, Al-Asaad and Hayes [3] 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 the area of software testing. Mutation testing [15] considers programs, termed mutants, that differ from the program under test by a single small error, such as changing the operator from add to subtract. The rationale for the approach is supported by two hypotheses: 1) programmers write programs that are close to

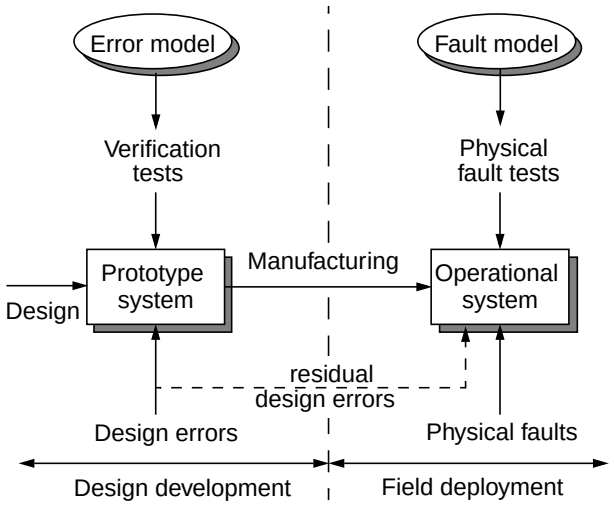


FIGURE 1. Correspondence between design verification and physical fault testing.

correct ones, and 2) a test set that distinguishes a program from all its mutants is also sensitive to more complex errors. Although considered too costly for wide-scale industrial use, mutation testing is one of the few approaches that has yielded an automatic test generation system for software testing, as well as a quantitative measure of error coverage (mutation score) [24]. Recently, Al Hayek and Robach [5] have successfully applied mutation testing to hardware design verification in the case of small VHDL modules.

This paper addresses design verification via error modeling and test generation for complex high-level designs such as microprocessors. A block diagram summarizing our methodology is shown in Figure 2. An implementation to be verified and its specification are given. For microprocessors, the specification is typically the instruction set architecture (ISA), and the implementation is a description of the new design in a hardware description language (HDL) such as VHDL or Verilog. 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 specification. A discrepancy between the two simulation outcomes indicates an error, either in the implementation or in the specification.

Section 2 describes our method for design error collection and presents some preliminary design error statistics that we have collected. Section 3 discusses design error modeling and illustrates test generation with these models. An experimental evaluation of our methodology and of the error models is presented in Section 4. Section 5 discusses the results and gives some concluding remarks.

2. DESIGN ERROR COLLECTION

Hardware design verification and physical fault testing are closely related at the conceptual level [3]. The basic task of physical fault testing (hardware design verification) is to generate tests that distinguish the correct circuit from faulty (erroneous) ones. The class of faulty

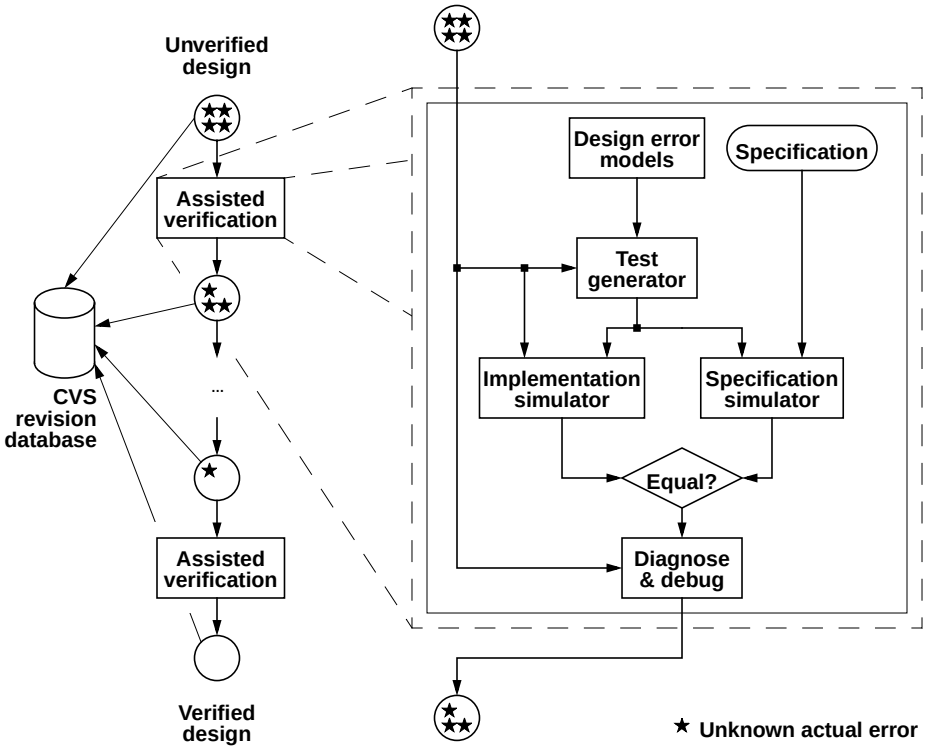


FIGURE 2. Deployment of proposed design verification methodology

circuits to be considered is defined by a logical fault model. Logical fault models represent the effect of physical faults on the behavior of the system, and free us from having to deal with the plethora of physical fault types directly. The most widely used logical fault model, the SSL model, combines simplicity with the fact that it forces each line in the circuit to be exercised. Typical hardware design methodologies employ hardware description languages as their input medium and use previously designed high-level modules. To capture the richness of this design environment, the SSL model needs to be supplemented with additional error models.

The lack of published data on the nature, frequency, and severity of the design errors occurring in large-scale projects is a serious obstacle to the development of error models for hardware design verification. Although bug reports are collected and analyzed internally in industrial design projects the results are rarely published. Examples of user-oriented bug lists can be found in [21,26]. Some insight into what can go wrong in a large processor design project is provided in [14].

The above considerations have led us to implement a systematic method for collecting design errors. Our method uses the CVS revision management tool [12] and targets ongoing design projects at the University of Michigan, including the PUMA high-performance microprocessor project [9] and various class projects in computer architecture and VLSI

```
(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.

_ verilog syntax error

_ conceptual error

X combinational logic:

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

_ sequential logic:

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

_ statement:

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

_ expression (RHS of assignment):

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

_ buses:

   _ wrong bus width
   _ wrong bit order

_ new category (describe below)

BUG DESCRIPTION:

Used wrong field from instruction
```

FIGURE 3. Sample error report.

design, all of which employ Verilog as the hardware description medium. Designers are asked to archive a new revision via CVS whenever a design error is corrected or whenever the design process is interrupted, making it possible to isolate single design errors. We have augmented CVS so that each time a design change is entered, the designer is prompted to fill out a standardized multiple-choice questionnaire, which attempts to gather four key pieces of information: (1) the motivation for revising the design; (2) the method by which a bug was detected; (3) a generic design-error class to which the bug belongs, and (4) a short narrative description of the bug. A uniform reporting method such as this greatly simplifies the analysis of the errors. A sample error report using our standard questionnaire is shown in Figure 3. The error classification shown in the report form is the result of the analysis of error data from several earlier design projects.

Design error data has been collected so far from four VLSI design class projects that involve implementing the DLX microprocessor [19], from the implementation of the LC-2 microprocessor [29] which is described later, and from preliminary designs of PUMA's fixed-point and floating-point units [9]. The distributions found for the various representative design errors are summarized in Table 1. Error types that occurred with very low frequency are combined in the "others" category in the table.

TABLE 1. Actual error distributions from three groups of design projects.

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

3. ERROR MODELING

Standard simulation and logic synthesis tools have the side effect of detecting some design error categories of Table 1, and hence there is no need to develop models for those particular errors. For example a simulator such as Verilog-XL [10] flags all Verilog syntax errors (category 9), declaration statement errors (category 12), and incorrect port lists of modules (category 16). Also, logic synthesis tools, such as those of Synopsys, usually flag all wrong bus width errors (category 10) and sensitivity-list errors in the *always* statement (category 13).

To be useful for design verification, error models should satisfy three requirements: (1) tests (simulation vectors) that provide complete coverage of the modeled errors should also provide very high coverage of actual design errors; (2) the modeled errors should be amenable to automated test generation; (3) the number of modeled errors should be relatively small. In practice, the third requirement means that error models that define a number of error instances linear, or at most quadratic in the size of the circuit are preferred. The error models need not mimic actual design bugs precisely, but the tests derived from complete coverage of modeled errors should provide very good coverage of actual design bugs.

3.1 Basic error models

A set of error models that satisfy the requirements for the restricted case of gate-level logic circuits was developed in [3]. Several of these models appear useful for the higher-level (RTL) designs found in Verilog descriptions as well. From the actual error data in Table 1, we derive the following set of five basic error models:

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

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

3.2 Test generation examples

Because of their relative simplicity, the foregoing error models allow tests to be generated and error coverage evaluated for RTL circuits of moderate size. We analyzed the test requirements of two representative combinational circuits: a carry-lookahead adder and an ALU. Since suitable RTL tools are not available, test generation was done manually, but in a systematic manner that could readily be automated. Three basic error models are considered: BOEs, MSEs, and BSEs. Test generation for SSLs is discussed in [1,6] and no tests are needed for BDEs, since the circuits under consideration do not have tristate buses.

Example 1: The 74283 adder

An RTL model [18] of the 74283 4-bit fast adder [30] appears in Figure 4. It consists of a carry-lookahead generator (CLG) and a few word gates. We show how to generate tests for some design error models in the adder and then we discuss the overall coverage of the targeted error models.

BOE on A bus: A possible bus value that activates the error is $A_g = (0XX1)$, where X denotes an unknown value. The erroneous value of A is thus $A_f = (1XX0)$. Hence, we can represent the error by $A = (\bar{D}XXD)$, where D $(\bar{D})$ represents the error signal which is 1 (0) in the good circuit and 0 (1) in the erroneous circuit. One way to propagate this error through the AND gate G_1 is to set $B = (1XX1)$. Hence, we get $G_2 = (1XX1)$, $G_5 = (DXX\bar{D})$, and $G_3 = (DXX\bar{D})$. Now for the module CLG we have $P = (1XX1)$, $G = (\bar{D}XXD)$, and $C_0 = X$. The resulting outputs are $C = (XXXX)$ and $C_4 = X$. This implies that $S = (XXXX)$ and hence the error is not detected at the primary outputs. We need to assign more input values to propagate the error. If we set $C_0 = 0$, we get $C = (XXD0)$, $C_4 = X$, and $S = (XXX\bar{D})$.

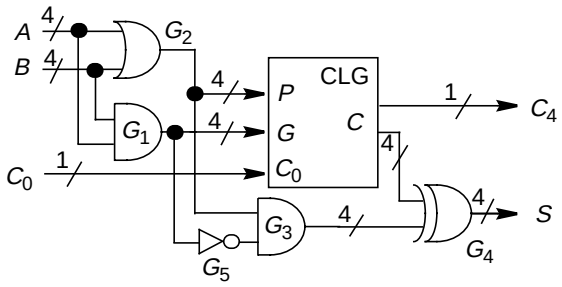


FIGURE 4. High-level model of the 74283 carry-lookahead adder [18].

Hence, the error is propagated to S and the complete test vector is (A, B, C_0) = (0XX11XX10).

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

Example 2: The c880 ALU

In this example, we try to generate tests for some modeled design errors in the c880 ALU, a member of the ISCAS-85 benchmark suite [8]. A high-level model based on a Verilog description of the ALU [23] is shown in Figure 5; it is composed of six modules: an adder, two multiplexers, a parity unit, and two control units. The circuit has 60 inputs and 26 outputs. The gate-level implementation of the ALU has 383 gates.

The design error models to be considered in the c880 are again BOEs, BSEs, and MSEs (inversion errors on 1-bit signals). We next generate tests for these error models.

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

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

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

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

All signals not mentioned in the above test have don't care values. We found that just 10 tests detect all 22 detectable BOEs in the c880 and serve to prove that another 2 BOEs are redundant.

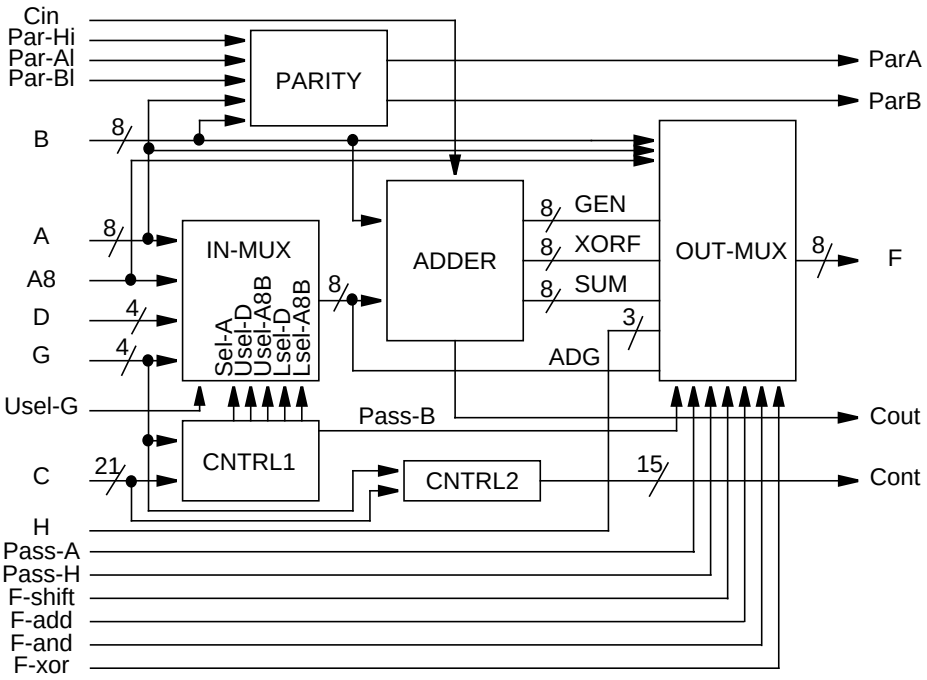


FIGURE 5. High-level model of the c880 ALU.

MSEs: Tests for BOEs detect most, but not all, inversion errors on multibit buses. In the process of test generation for the c880 ALU, we noticed a case where a test for an inversion error on bus *A* can be found even though the BOE on *A* is redundant. This is the case when an *n*-bit bus (*n* odd) is fed into a parity function. Testing for inversion errors on 1-bit signals needs to be considered explicitly, since a BOE on a 1-bit bus is not possible. Most inversion errors on 1-bit signals in the c880 ALU are detected by the tests generated for BOEs and BSEs. This is especially true for the control signals to the multiplexers.

3.3 Conditional error model

The preceding examples, as well as prior work on SSL error detection [1,6], show that the basic error models can be used with RTL circuits, and that high, but not complete, error coverage can be achieved with small test sets. These results are further reinforced by our experiments on microprocessor verification (Section 4) which indicate that a large fraction of actual design errors (67% in one case and 75% in the other) is detected by complete test sets for the basic errors. To increase coverage of actual errors to the very high levels needed for design verification, additional error models are required to guide test generation. Many more complex error models can be derived directly from the actual data of Table 1 to supplement the basic error types, the following set being representative:

- *Bus count error (BCE):* This corresponds to defining a module with more or fewer input buses than required (categories 4 and 8).

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

Although, this extended set of error models increases the number of actual errors that can be modeled directly, we have found them to be too complex for practical use in manual or automated test generation. We observed that the more difficult actual 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. Modeling these difficult composite errors directly is impractical as the number of error instances to be considered is too large and such composite modeled errors are too complex for automated test generation. However, as noted earlier, a good error model does not necessarily need to mimic actual errors accurately. What is required is that the error model necessitates the generation of these more specific tests. To be practical, the complexity of the new error models should be comparable to that of the basic error models. Furthermore the (unavoidable) increase in the number of error instances should be controlled to allow trade-offs between test generation effort and verification confidence. We found that these requirements can be combined by augmenting the basic error models with a condition.

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

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

When $n = 0$, a conditional error (*C*,*E*) reduces to the basic error *E* from which it is derived. Higher-order conditional errors enable the generation of more specific tests, but lead to a greater test generation cost due to the larger number of error instances. For example, the number of CSSL*n* errors on a circuit with *N* signals is $\theta(2^n N^{n+1})$. Although the total set of all *N* signals we consider for each term in the condition can possibly be reduced, CSSL*n* errors where $n > 2$ are probably not practical.

For gate-level circuits (where all signals are 1-bit), it can be shown that CSSL1 errors

cover the following basic error models: MSEs (excluding XOR and XNOR gates), missing 2-input gate errors, BSEs, single BCEs (excluding XOR and XNOR gates), and bus driver errors. That CSSL1 errors cover missing two-input gate errors can be seen as follows. Consider a two-input AND gate Y=AND($X1$,$X2$) in the correct design; in the erroneous design, this gate is missing and net Y is identical to net $X1$. To expose this error we have to set $X1$ to 1, $X2$ to 0, and sensitize Y. Any test that detects the CSSL1 error, ($X2$=0, Y s-a-0) in the erroneous design, will also detect the missing gate error. The proof for other gate types is similar. Higher-order CSSLn errors improve coverage even further.

4. COVERAGE EVALUATION

To show the effectiveness of a verification methodology, one could apply it and a competing methodology to an unverified design. The methodology 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 we have demonstrated such test generation in Section 3.2, it has yet to be automated. We therefore designed a controlled experiment that approximates the conditions of the original experiment, while avoiding the need for automated 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 6. As design error models are used to guide test generation, the effectiveness is closely related to the synthetic error models used.

To evaluate our methodology, a circuit is chosen for which design errors are to be systematically recorded during its design. Let D_0 be the final, presumably correct design. From the CVS revision database, the actual errors are extracted and converted such that they can be injected in the final design D_0. In the evaluation phase, the design is restored to an (artificial) erroneous state D_1 by injecting a single actual error into the final design D_0. This set-up approximates a realistic on-the-fly design verification scenario. The experiment answers the question: 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, and determining 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 which defines the dominated modeled error. Such a dominated modeled error has the property that any test that detects the modeled error in D_2 will also detect 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. Error detection is determined as discussed earlier (see Section 1, Figure 2). Note that the concept of dominance in the context of design verification is slightly different than in physical fault testing. Unlike in 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

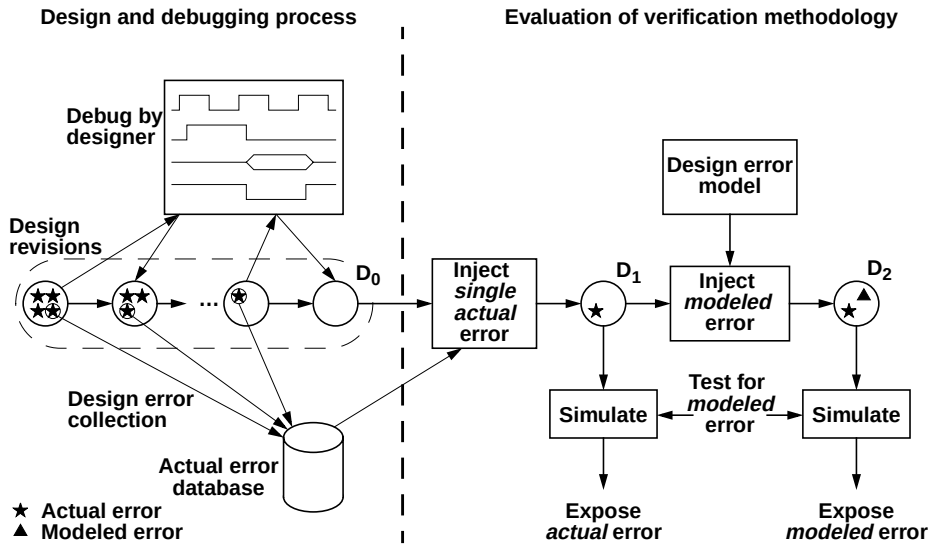


FIGURE 6. Experiment to evaluate the proposed design verification methodology

design error. To the extent that the design errors are independent, it does not matter if we consider a single or multiple design errors at a time. Furthermore, our results are independent of the order in which one applies the generated tests.

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.

4.1 A simple microprocessor

The Little Computer 2 (LC-2) [29] is a small microprocessor of conventional design used for teaching purposes at the University of Michigan. It has a representative set of 16 instructions which is a subset of the instruction sets of most current microprocessors. To serve as a test case for design verification, one of us designed behavioral and RTL synthesizable Verilog descriptions for the LC-2. The behavioral model (specification) 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. It 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 (Section 2).

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

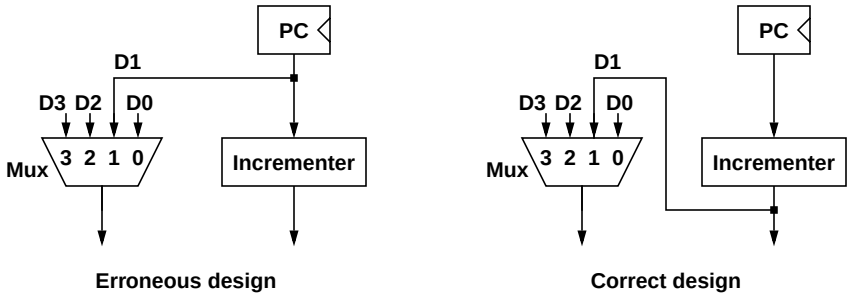


FIGURE 7. An example of an actual design error that is dominated by an SSL error.

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

CORRECT CODE:

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

ERRONEOUS CODE:

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

Design error

```
// Instruction sequence

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

Test sequence

FIGURE 8. An example of an actual design error for which no dominated modeled error was found, and an instruction sequence that detects the actual error.

can be found. An example where we were able to do that is shown in Figure 7. The error is a BSE on data input D_1 of the multiplexer attached to the program counter PC. Testing for D_1 stuck-at-1 will detect the BSE since the outputs of PC and its incrementer are always different, i.e., the error is always activated, so testing for this SSL error will propagate the signal on D_1 to a primary output of the microprocessor. A case where we were not able to find a modeled error dominated by the actual error is shown in Figure 8. The error occurs where a signal is assigned a value independent of any condition. However, the correct implementation requires an if-then-else construct to control the signal assignment. To activate this error, we need to set ir_out[15:12] == 4'b1101, ir_out[8:6] ≠ 3'b111, and RF[ir_out[8:6]] ≠ RF[3'b111], where RF[i] refers to the contents of the register i in the register file. An instruction sequence that detects this error is shown in Figure 8.

We analyzed the actual design errors in both the behavioral and RTL designs of the LC-2, and the results are summarized in Table 2. A total of 20 design errors were made during the design, of which four errors 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 numbers in parentheses refer to the corresponding category in Table 1. The columns in the table give the type of the simplest dominated modeled error corresponding to each actual error. For example, among the 4 remaining wrong-signal-source errors, 2 dom-

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

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

inate an SSL error and 2 dominate a BSE error.

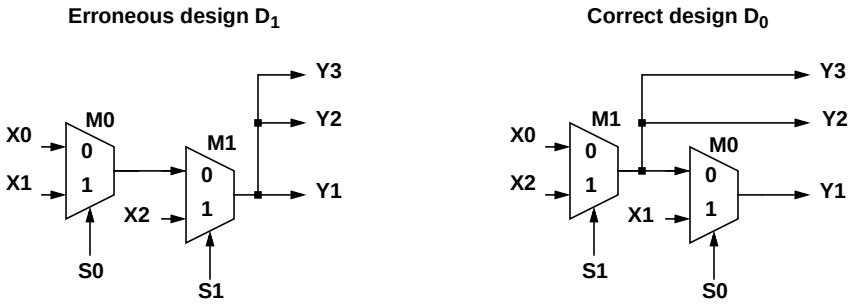
We can infer from Table 2 that most errors are detected by tests for SSL errors or BSEs. About 75% of the actual errors in the LC-2 design can be detected after simulation with tests for SSL errors and BSEs. The coverage increases to 90% if tests for CSSL1 are added.

4.2 A pipelined microprocessor

Our second design case study considers the well-known DLX microprocessor [19], which has more of the features found in contemporary microprocessors. The particular DLX version considered is a student-written design that implements 44 instructions, has a five-stage pipeline and branch prediction logic, and consists of 1552 lines of structural Verilog code, excluding the models for library modules such as adders, 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, and as such is not cycle-accurate: only 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 or not with respect to each reference model. Errors undetectable with respect to both reference models may arise from the following two reasons: (1) Designers sometimes make changes to don't care features, and log them as errors. This happens because designers can have a more detailed specification (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



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 9. Example of an actual design error in our DLX implementation

error dominated by the actual error, wherever possible. One actual error involved multiple signal source errors, and is shown in Figure 9. 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 via fanout $Y2$ 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. A total of 39 design errors were recorded by the designer. The actual design errors are grouped by category; the numbers in parentheses refer again to Table 1. The correspondence between the categories is imprecise, because of inconsistencies in the way in which different student designers classified their errors. Also, some errors in Table 3 are assigned to a more specific category than in Table 1, to highlight their correlation with the errors they dominate. ‘Missing module’ and ‘wrong signal source’ errors account for more than half of all errors. The 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 add up the number given

TABLE 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**	**Un-known**
Missing module (2)	8	2	14	0	2	0	6	1	0	1
Wrong singal source (1)	9	2	11	1	4	5	1	0	0	0
Complex (2)	3	0	3	0	3	0	0	0	0	0
Inversion (5)	1	2	3	3	0	0	0	0	0	0
Missing input (4)	1	0	3	0	0	0	1	0	0	0
Unconnected input (4)	3	0	3	3	0	0	0	0	0	0
Missing minterm (2)	1	0	1	0	0	0	0	0	1	0
Extra input (2)	1	0	1	0	1	0	0	0	0	0
Total	**27**	**6**	**39**	**7**	**10**	**5**	**8**	**1**	**1**	**1**

in ‘Total’; the difference corresponds to actual errors that are not undetectable 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 10 detectable ‘missing module(s)’ errors, 2 dominate an SSL error, 6 dominate a CSSL1 error, and one dominates a CBOE; for the remaining one, we were not able to find a 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 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.

5. DISCUSSION

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

The conditional error models proved to be especially useful for detecting actual errors that involve missing logic. Most ‘missing module(s)’ and ‘missing input(s)’ in Table 3

cannot be covered when only the basic errors are targeted. However, all but one of them is covered when CSSL1 and CBOE errors are targeted as well. The same observation applies to the 'missing assignment(s)' errors in Table 2.

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

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

We envision the proposed methodology eventually being deployed as suggested in Figure 2. Given an unverified design and its specification, tests targeted at modeled design errors are automatically generated and applied to the specification and the implementation. When a discrepancy is encountered, the designer is informed and perhaps given guidance on diagnosing and fixing the error.

ACKNOWLEDGMENTS

We thank Steve Raasch and Jonathan Hauke for their help in the design error collection process. We further thank Matt Postiff for his helpful comments.

The research discussed in this paper is supported by DARPA under Contract No. DABT63-96-C-0074. The results presented herein do not necessarily reflect the position or the policy of the U.S. Government.

REFERENCES

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

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

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

[4] H. Al-Asaad, D. Van Campenhout, J. P. Hayes, T. Mudge, and R. B. Brown. "High-level design verification of microprocessors via error modeling," *Dig. IEEE High Level Design Validation and Test Workshop*, pp. 194–201, 1997.

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

[6] D. Bhattacharya and J. P. Hayes, "High-level test generation using bus faults," *Dig. 15th Int. Symp. on Fault-Tolerant Computing*, 1985, pp. 65–70.

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

[8] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran", *Proc. IEEE Int. Symp. on Circuits and Systems*, 1985, pp. 695-698.

[9] R. Brown et al., "Complementary GaAs technology for a GHz microprocessor", *Techn. Dig. of*

the GaAs IC Symp., 1996, pp. 313-316.
[10] Cadence Design Systems Inc., *Verilog-XL Reference Manual*, 1994.
[11] F. Casaubieilh et al., "Functional verification methodology of Chameleon processor", *Proc. Design Automation Conf.*, 1996, pp. 421–426.
[12] P. Cederqvist et al., *Version Management with CVS*, Signum Support AB, Linkoping, Sweden, 1992.
[13] A. K. Chandra et al., "AVPGEN - a test generator for architecture verification", *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, Vol. 3, pp. 188–200, June 1995.
[14] R. P. Colwell and R. A. Lethin, "Latent design faults in the development of the Multiflow TRACE/200", *IEEE Trans. on Reliability*, Vol. 43, No. 4, pp. 557–565, December 1994.
[15] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer", *IEEE Computer*, pp. 34–41, April 1978.
[16] S. Devadas, A. Ghosh, and K. Keutzer, "Observability-based code coverage metric for functional simulation", *Proc. Int. Conf. on Computer-Aided Design*, 1996, pp. 418–425.
[17] G. Ganapathy et al., "Hardware emulation for functional verification of K5", *Proc. Design Automation Conf.*, 1996, pp. 315–318.
[18] M. C. Hansen and J. P. Hayes, "High-level test generation using physically-induced faults", *Proc. VLSI Test Symp.*, 1995, pp. 20-28.
[19] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, San Francisco, 1990.
[20] A. Hosseini, D. Mavroidis, and P. Konas, "Code generation and analysis for the functional verification of microprocessors", *Proc. Design Automation Conf.*, 1996, pp. 305–310.
[21] Intel Corp., "Pentium Processor Specification Update," 1998, available from http://www.intel.com.
[22] M. Kantrowitz and L. M. Noack, "I'm done simulating; Now what? Verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor", *Proc. Design Automation Conf.*, 1996, pp. 325–330.
[23] H. Kim, "C880 high-level Verilog description", Internal report, University of Michigan, 1996.
[24] K. N. King and A. J. Offutt, "A Fortran language system for mutation-based software testing", *Software Practice and Experience*, Vol. 21, pp. 685–718, July 1991.
[25] J. Kumar, "Prototyping the M68060 for concurrent verification", *IEEE Design & Test*, Vol. 14, No. 1, pp. 34–41, 1997.
[26] MIPS Technologies Inc., *MIPS R4000PC/SC Errata, Processor Revision 2.2 and 3.0*, May 1994.
[27] A. J. Offutt et al., "An experimental determination of sufficient mutant operators", *ACM Trans. on Software Engineering & Methodology*, Vol. 5, pp. 99-118, April 1996.
[28] S. Palnitkar, P. Saggurti, and S.-H. Kuang, "Finite state machine trace analysis program", *Proc. Int. Verilog HDL Conf.*, 1994, pp. 52–57.
[29] M. Postiff, *LC-2 Programmer's Reference Manual*, Revision 3.1, University of Michigan, 1996.
[30] Texas Instruments, *The TTL Logic Data Book*, Dallas, 1988.
[31] M. R. Woodward, "Mutation testing – its origin and evolution", *Information & Software Technology*, Vol. 35, pp. 163–169, March 1993.
[32] M. Yoeli (ed.), *Formal Verification of Hardware Design*, IEEE Computer Society Press, Los Alamitos, Calif., 1990.