High-Level Design Verification of Microprocessors via Error Modeling[1]

Hussain Al-Asaad, David Van Campenhout, John P. Hayes, Trevor Mudge, and Richard B. Brown

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor, MI 48109-2122
E-mail: {halasaad, davidvc, jhayes, tnm, brown}@eecs.umich.edu

Abstract

A project is under way at the University of Michigan to develop 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. We have developed a method to systematically collect design error data, and gathered concrete error data from a number of microprocessor design projects. The error data are being used to derive error models suitable for design verification testing. Design verification is done by simulating tests targeted at instances of the modeled errors. We are conducting experiments in which targeted tests are generated for modeled errors in circuits ranging from RTL combinational circuits to pipelined microprocessors. The experiments gauge the quality of the error models and explore test generation for these models. This paper describes our approach and presents some initial experimental results.

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. 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 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 methods and simulation-based methods. Formal methods try to verify the correctness of a system by means of mathematical proof [26]. Such methods consider all possible behaviors of the models representing the system and its specification, whereas simulation-based methods can only consider a subset of all behaviors. The accuracy and completeness of the system and specification models is a fundamental limitation for any formal method. The computational complexity of many methods makes their applicability dependent on the derivation of suitable abstractions (models).

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 cases. These tests are very effective in the beginning of the debug phase, but lose there usefulness later. Recently, tools have been developed to assist in the generation of focused tests [9,16]. Although these tools can significantly increase 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 [1]. Another source of test stimuli, thanks to advances in simulator and emulator technology, is existing application and system software. Successfully booting the operating system has become a common quality requirement [13,20].

Common to all the test generation techniques men-

1. This research 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.

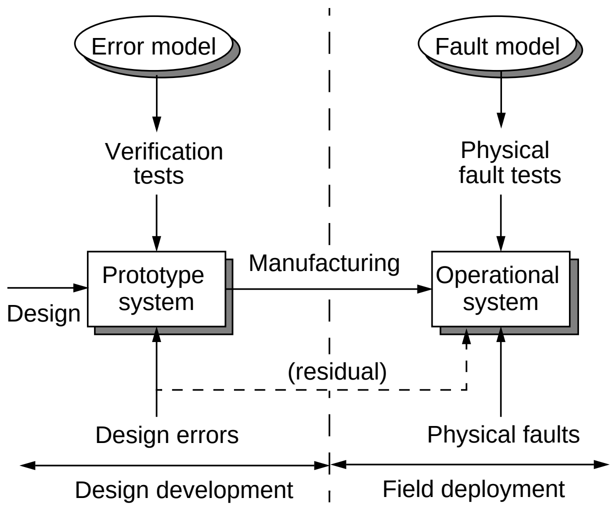

Figure 1 Similarity between design verification and physical fault testing.

tioned above is that they are not targeted at specific design errors. This poses the problem of quantifying the effectiveness of a test set. Various coverage metrics have been proposed to address this problem. These include code coverage metrics from software testing [1,4,7], finite state machine coverage [16,17,23], architectural event coverage [17], and observability-based metrics [12]. A shortcoming of all these metrics is that the relationship between the metric and detection of classes of design errors is not well understood.

A different approach is to use 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 [2] 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) errors that can be targeted with standard ATPG tools. This provides a method to generate tests with a provably high coverage for certain classes of modeled errors.

A second method originated from observing the similarities between software testing and hardware design verification. Mutation testing [11] considers programs, termed mutants, that differ from the program under test by a single simple error. The rationale for the approach is supported by two hypotheses: 1) programmers write programs that are close to correct ones, and 2) a test set that distinguishes a program from all its mutants is also sensitive to more complex errors. Although still considered too costly for industrial purposes, mutation testing is one of the only approaches that has yielded an automatic test generation system for software testing [19]. Recently, Al Hayek and Robach [3] have applied mutation testing to hardware

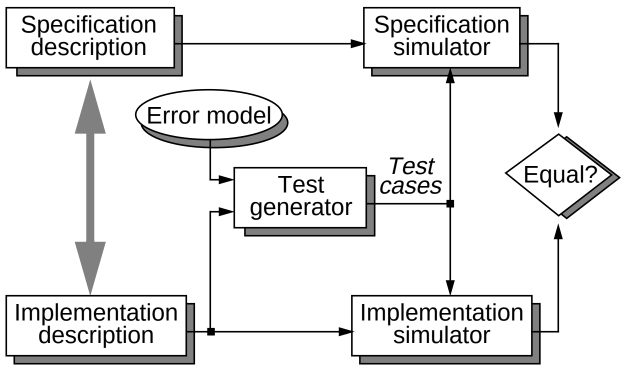

Figure 2 Block diagram of our validation method.

design verification. They demonstrate their approach with verification examples of small VHDL modules.

This paper addresses design verification for high-level designs such as microprocessors via error modeling and test generation. A block diagram summarizing our methodology is shown in Figure 2. Section 2 describes our method for design error collection and presents some design error statistics. Section 3 discusses design error modeling. Coverage experiments are described in section 4, followed by concluding remarks.

2 Design Error Collection

Motivation

Hardware design verification and physical fault testing are closely related at the conceptual level [2]. The basic task of physical fault testing (hardware design verification) is to generate tests that distinguish the correct circuit from faulty (erroneous) ones. The class of faulty circuits to be considered is defined by a logical fault model. Logical fault models represent the effect of physical faults on the behavior of the system, and free us from having to deal with the plethora of physical fault types directly. The most widely used logical fault model, the SSL model, combines simplicity with the fact that it forces each line in the circuit to be exercised.

Typical hardware design methodologies employ hardware description languages as their input medium and use previously designed high-level modules. To capture the richness of this design environment, the SSL model will have to be supplemented with new 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. An example of a user-oriented bug list can be found in [21]. Some insight into what can go wrong in a large microprocessor design project is provided in [10].

Collection method and statistics

The considerations above have led us to implement a systematic method for collecting design errors. Our method uses the CVS revision management tool [8] and targets ongoing design projects at the University of Michigan, including the PUMA high-performance microprocessor project [6] and various class projects, all of which employ Verilog as the hardware description medium. Designers are asked to archive a new revision 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 new revision is entered, the designer is prompted to fill out a standardized multiple-choice questionnaire which gathers four key pieces of information: (1) the motivation for revising the design; (2) the method by which a bug was detected; (3) the class to which the bug belongs, and (4) a short narrative description of the bug. This uniform reporting method greatly simplifies the analysis of the errors. A sample error report is shown in Figure 3. The error classification shown in the report is the result of the analysis of error data from several earlier design projects.

Design error data has been collected so far from four VLSI design class projects that involve implementing the DLX microprocessor [15], and from the design of PUMA's fixed point and floating point units [6]. The distributions found for the various representative design errors are summarized in Table 1. Error categories that occurred with very low frequency were combined in the "others" category in the table.

3 Error Modeling

We have begun the development of generic design error models based on the data we are collecting. The requirements for these models are threefold: (1) tests (simulation vectors) that provide complete coverage of the modeled errors should also provide very high coverage of actual design errors; (2) the modeled errors should be amenable to automated test generation; (3) the number of modeled errors should be relatively small. The error models need not mimic actual design bugs precisely, but the tests derived from complete coverage of modeled errors should also provide very good coverage of actual design bugs.

Standard simulation and synthesis tools have a side effect of detecting some design error categories of Table 1 and hence there is no need to develop error models for those error categories. For example, the Verilog compiler

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

MOTIVATION:

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

BUG DETECTED BY:

_ inspection
_ compilation
X simulation
_ synthesis

BUG CLASSIFICATION:

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

X combinational logic:

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

_ sequential logic:

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

_ statement:

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

_ expression (RHS of assignment):

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

_ buses:

     _ wrong bus width
     _ wrong bit order

_ verilog syntax error

_ conceptual error

_ new category (describe below)

BUG DESCRIPTION:

Used wrong field from instruction
```

Figure 3 Sample error report.

Table 1 Design error distributions.

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

flags all Verilog syntax errors (category 9), declaration statement errors (category 12), and incorrect port list of modules (category 16). Also, synthesis tools flag all wrong bus width errors (category 10) and sensitivity list errors in the *always* statement (category 13).

A set of error models that satisfy the requirements for the restricted case of gate-level logic circuits was developed in [2]. Several of these models appear useful for the higher-level (RTL) designs found in Verilog descriptions as well. For example, we can determine the following set of error models from the RTL errors in Table 1:

- *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.
- *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.
- *Bus order error (BOE)*: This refers to using an incorrect order of bits in a bus (category 16). Bus flipping is the most common form of BOE.
- *Bus count error (BCE):* This corresponds to using a module with more or fewer input buses than required (categories 4 and 8).
- *Bus source error (BSE):* This error corresponds to connecting a module input to a wrong source (category 1).
- *Bus driver error (BDE)*: This error refers to mistakenly driving a bus with two sources (category 16). This error model is useful in the validation of bus-oriented designs.

For behavioral Verilog descriptions, additional error models are needed as follows:

- *Label count error (LCE)*: This error corresponds to incorrectly adding or removing 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 FSM with an extra or missing state (category 14).
- *Next state error (NSE)*: This error corresponds to incorrect next state function in an FSM (category 14).

Conceptual errors (category 2) cannot be easily modeled due to their very high-level nature. Correcting such an error usually requires a major revision to the design. However, a conceptual error often manifests itself as a combination of the error models described above.

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

4 Coverage Experiments

We are presently conducting a set of experiments whose goal is twofold: 1) investigate the relationship between coverage of modeled design errors and coverage of (more complex) actual errors; and 2) explore test generation with these candidate error models. At this stage of our work we have yet to automate test generation. Consequently, we are limited in the size of circuits and the variety of error models that we can handle. We describe here three experiments for small but representative circuits and for a limited set of error models. In the first two experiments, we show how to generate tests for certain error models in a carry-lookahead adder and an ALU, and we further quantify the coverage of these error models. In a last experiment, we generate tests for two error models and for actual design errors in a small pipelined microprocessor and evaluate the coverage obtained.

Experiment 1: The 74283 adder

An RTL model [14] of the 74283 4-bit fast adder [24] is shown in Figure 4. It consists of a carry-lookahead generator (CLG) and a few word gates. The 74283 adder is an SSL-irredundant circuit, i.e. all SSL faults are detectable. We next show how to generate tests for some design error models in the adder and then we discuss our 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 faulty 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 faulty 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 $S = (XXX\bar{D})$. Hence, the error is propagated to S and the complete test vector is $(A, B, C_0) = (0XX11XX10)$.

BSE of the P bus (The correct source is G_3): To activate the error we need to set opposite values on at least one bit of the P and G_3 buses. If we start with $P_f = (XXX0)$ and $G_3 = P_g = (XXX1)$, we reach to a conflict through implications. If we try $P_f = (XXX1)$ and $G_3 = P_g = (XXX0)$, we obtain $P = (XXX\bar{D})$, $A = (XXX1)$, and $B = (XXX1)$. However, no error is propagated through the CLG module since $G = (XXX1)$. After all the activation conditions are explored, we can easily conclude that the error is redundant (undetectable).

MSE G_3/XNOR: To distinguish the word AND gate G_3 from an XNOR gate, we need to apply the all-0 pattern on one of the gates forming G_3. So, we start with the values $G_5 = (0XXX)$ and $G_2 = (0XXX)$. By making implications, we find that there is a conflict when selecting the values of A and B. We then change to another set of activation condition $G_5 = (X0XX)$ and $G_2 = (X0XX)$. This also leads to a conflict. After trying the other possible combinations, we conclude that no test exists, and hence the error is redundant.

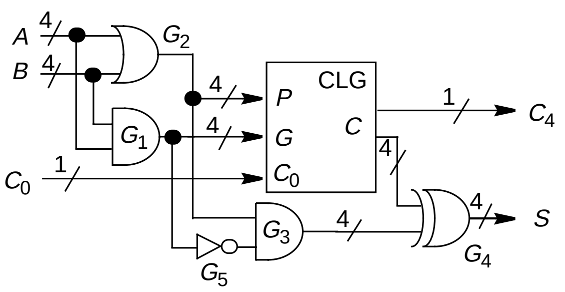

Figure 4 High-level model of the 74283 carry-lookahead adder [14].

We generated tests for all BSEs in the adder and we found that just 2 tests detect all 33 detectable BSEs and proved that 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.

Experiment 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 [5]. A high-level model based on a Verilog description of the ALU [18] is shown in Figure 5, and is composed of six modules: an adder, two multiplexing units, a parity unit, and two control units. It 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: 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 across IN-MUX and then OUT-MUX is selected. The signal values needed to activate this path are:

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

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

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

All signals not mentioned in the above test have don't care values.

We generated tests for all BOEs in the c880. We found that just 10 tests detect all 22 detectable BOEs and serve to prove that another 2 BOEs are redundant.

BSEs: The buses in the ALU are grouped according to their size since the correct source of a bus must have the same size as the incorrect one. We targeted BSEs with bus widths 8 and 4 only. We found that by adding 3 tests to the ten tests generated for BOEs, we are able to detect all 27 BSEs affecting the c880's multibit buses. Since the multi-

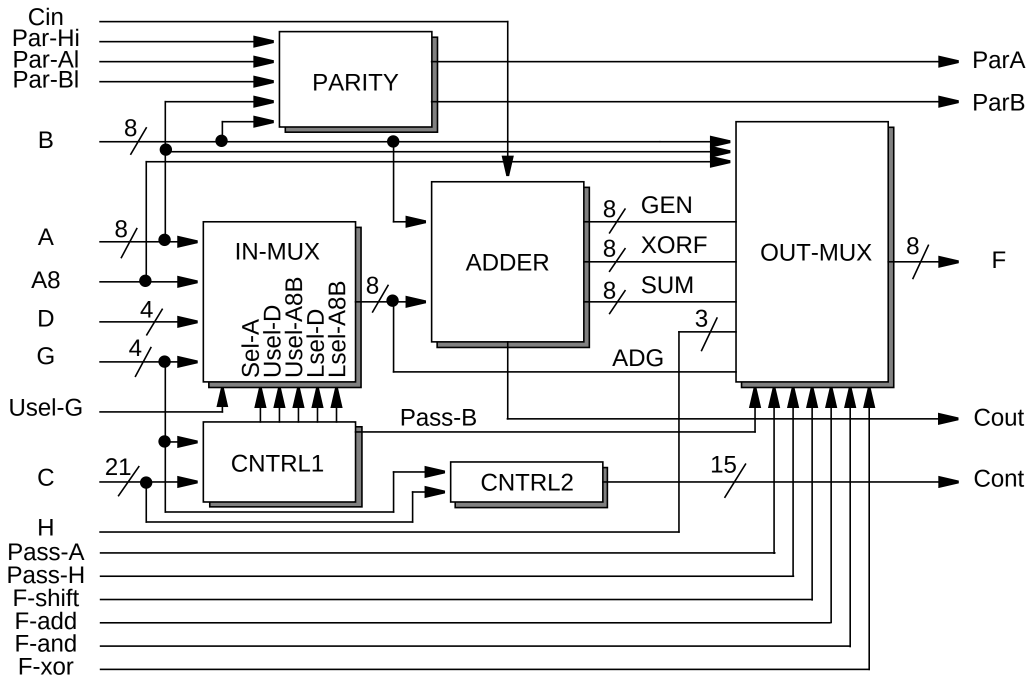

Figure 5 High-level model of the c880 ALU.

plexing units are not decoded, most BSEs on their control 1-bit signals are detected by the tests generated for BOEs. Further test generation is needed to get complete coverage of BSEs on the other 1-bit signals.

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

In summary, we have generated tests for a subset of error models using an RTL version of the ISCAS-85 c880 benchmark. This experiment demonstrates that good coverage of the modeled errors can be achieved with very small test sets.

Experiment 3: A pipelined microprocessor

This experiment considers the well-known DLX microprocessor [15]. The particular DLX version considered is a student-written design that implements 44 instructions, has a five-stage pipeline and branch prediction logic, and consists of 1552 lines of structural Verilog code, excluding the models for library modules such as adders, register-files, etc. A simplified block diagram of the design is shown in Figure 6. The design errors committed by the student during the design process were systematically recorded using our error collection system. An example of an actual error is shown in Figure 7. The differing code fragments of the good and faulty designs are shown with some context. The error is composed of several signal source errors. The bug data reported by the student for this error was shown earlier in Figure 3.

In this test generation experiment, we only consider two basic error models: the SSL fault model, and the inverter insertion (INV) model adopted from mutation testing [25]. Under the INV model, the faulty design differs from the fault-free one in that it has an extra inverter inserted in any one of the signal lines. A set of faulty designs was constructed by injecting an actual design error (Table 1) or a single modeled error into the final complete DLX design. For each such error, a test consisting of a short DLX assembly program, was constructed manually. An example of such a test is shown in Figure 8. This test was targeted at an SSL error in the branch prediction logic. To expose this particular error, a conditional branch has to cause a hit in the branch target buffer, and the branch prediction has to be wrong. The test code is a small loop where in the last iteration, these conditions are exhibited.

Three separate test sets targeted at the INV errors, the SSL errors, and the actual design errors were devised. As tests were constructed by hand, the size of this experiment was necessarily very limited. We considered only those

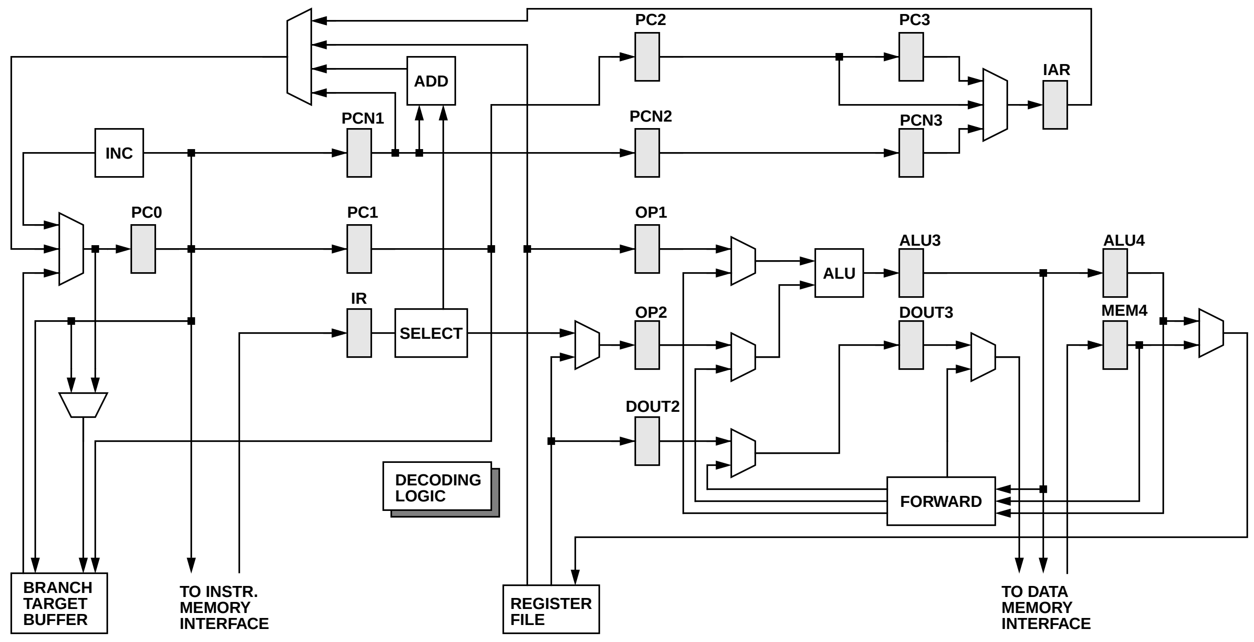

Figure 6 Block diagram of the student-designed DLX.

```
CONTEXT:

//
//  Instruction Decoding . . .
//
assign {c0b,c1b,c2b,c3b,c4b,c5b} = ~IR_IN[0:5];
//
//  Choose ‘opcode’
//
mux2 #(6) opcode_mux(.Y(opc), .S0(rtype_opcode),
          .IN0(IR_IN[0:5]),
.IN1(IR_IN[26:31]));
assign {f0, f1, f2, f3, f4, f5 } = opc[0:5];
assign {f0b,f1b,f2b,f3b,f4b,f5b} = ~opc[0:5];
//
//  Encode RDEST
//(All but: SW,J,JR,BEQZ,BNEQ,TRAP,RFE,HALT)
//
nor6 a15(a15_Y, a10_Y, a11_Y, a12_Y, a13_Y,
          a14_Y, 1’b0);
and2 a16(use_rdest, nop_opcode_bar, a15_Y);
//
// DISCREPANT CODE FOLLOWS HERE
//

CORRECT CIRCUIT:

and6 a10(a10_Y, c0, c1b, c2, c3b, c4, c5);
and6 a11(a11_Y, c0b, 1’b1, c2b, c3b, c4, c5b);
and6 a12(a12_Y, c0b, c1b, c2b, c3, c4b, 1’b1);
and6 a13(a13_Y, c0b, c1, c2b, c3b, c4b, 1’b1);
and6 a14(a14_Y, c0, c1, c2, c3, c4, c5);

ERRONEOUS CIRCUIT:

and6 a10(a10_Y, f0, f1b, f2, f3b, f4, f5);
and6 a11(a11_Y, f0b, 1’b1, f2b, f3b, f4, f5b);
and6 a12(a12_Y, f0b, f1b, f2b, f3, f4b, 1’b1);
and6 a13(a13_Y, f0b, f1, f2b, f3b, f4b, 1’b1);
and6 a14(a14_Y, f0, f1, f2, f3, f4, f5);
```

Figure 7 Example of an actual design error.

```
    // TARGET ERROR: dbp_a5.IN0 stuck-at-0
    //
    // or3 dbp_a5 (.Y(btb_squash),
    //    .IN0(hit_but_not_taken),
    //    .IN1(bw_not_taken),
    //    .IN2(fw_taken)) ;
    //
    @C00
    main:
         ADDUI r1, r0, #2
    loop:
         SUBI  r1, r1, #1
         BNEZ  r1, loop
         NOP
         HALT
```

Figure 8 Example of a test for the DLX design.

actual design errors that occurred in the instruction decode stage of the pipeline, which was observed to contain the most actual design errors. As to the modeled errors, we selected a random sample of 5.5% of total population in the same decode stage. Error detection was determined by checking the contents of the DLX’s register file and the contents of its program counter in every clock cycle.

We then calculated the coverage of each test set with respect to all three error types. The results are summarized in Table 2. Only one actual error was not detected by the test sets targeted at the modeled errors. This error, which was presented in Figures 3 and 7, can only be exposed by two particular instruction types: “shift left logical” (SLL) and “set greater than” (SGT). However, in the decode stage there does not appear to be any single INV or SSL error that can only be exposed by an SLL or SGT instruction. The execute pipeline stage contains logic specific to

Table 2 Coverage of modeled and actual design errors in the instruction decode stage of the DLX pipeline by various test sets.

Target errors of test set	Coverage (%) of each error type		
	INV	SSL	Actual
Inverter insertion (INV)	100	82	92
Single stuck-line (SSL)	100	100	92
Actual design errors	72	53	100

SLL and SGT instructions. Hence the undetected actual error in the decode stage would have been detected by the test set targeted at the modeled errors in the execute stage.

The coverage results suggest that test sets aimed at the modeled errors provide good coverage of actual design errors. This conclusion, although still very preliminary in view of the small scale of the experiment, tends to confirm the power of simple error models, which has been observed in several other design validation domains [2,22].

5 Conclusions

We are attempting to develop a practical microprocessor design verification methodology and supporting CAD tools, based on the synthesis of error models from actual design errors, and the adaptation of test technology for physical faults to detect these errors. Our experimental work to date suggests that high-level test generation is feasible, and that good coverage of both modeled and actual errors is possible with relatively small test sets. Supporting evidence for this approach is provided by experiments that compare the error coverage of verification tests with respect to such errors in some nontrivial designs.

References

[1] 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.

[2] H. Al-Asaad and J. P. Hayes, “Design verification via simulation and automatic test pattern generation”, *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 1995, pp. 174-180.

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

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

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

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

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

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

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

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

[11] 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.

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

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

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

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

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

[17] 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. ACM/IEEE Design Automation Conference*, 1996, pp. 325–330.

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

[19] 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.

[20] J. Kumar, “Prototyping the M68060 for concurrent verification”, *IEEE Design & Test of Computers*, Vol. 14, No. 1, pp. 34–41, 1997.

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

[22] A. J. Offutt et al., “An experimental determination of sufficient mutant operators”, *ACM Transactions on Software Engineering & Methodology*, Vol. 5, pp. 99-118, April 1996.

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

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

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

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