Extraction Error Modeling and Automated Model Debugging in High-Performance Custom Designs

Yu-Shen Yang, *Student Member, IEEE*, Andreas Veneris, *Senior Member, IEEE*, Paul Thadikaran, *Member, IEEE*, and Srikanth Venkataraman, *Member, IEEE*

***Abstract*—In the design cycle of high-performance integrated circuits, it is common that certain components are designed directly at the transistor level. This level of design representation may not be appropriate for test generation tools that usually require a model expressed at the gate level. Logic extraction is a key step in test model generation to produce a gate-level netlist from the transistor-level representation. This is a semi-automated process which is error-prone. Once a test model is found to be erroneous, manual debugging is required, which is a resource-intensive and time-consuming process. This paper presents an in-depth analysis of typical sets of extraction errors found in the test model representations of the pipelines in high-performance designs today. It also develops an automated debugging solution for single extraction errors for pipelines with no state equivalence information. A suite of experiments on circuits with similar architecture to that found in the industry confirms the fitness and practicality of the solution.**

***Index Terms*—Debugging, errors, extraction, test model, VLSI.**

I. Introduction

LARGE AND complex VLSI designs such as microprocessors, systems-on-chip (SoCs), and application specific integrated circuits (ASICs) often require high-performance low-power custom logic blocks designed at the transistor level [2], [3], [5], [12], [17], [21], [22]. Since a transistor-level representation cannot be used directly to generate production tests, a gate (logic)-level representation of these blocks, also known as a *test model*, is extracted from the transistor schematic and used instead. *Logic extraction* is a semi-automated process which is error-prone [16], [21], [22]. Since the amount of verification on the test model before test generation can be limited, extraction errors may carry forward in test generation and discovered during test validation. At that point, manual debugging is performed, which is a time-consuming and resource-intensive process.

Traditionally, generation of scan tests for high-volume manufacturing of complex VLSI circuits with custom-made components requires a design flow similar to the one shown in Fig. 1 [12], [17], [21], [22]. The test model is a logic-level representation that conforms to the functionality of the extracted gate-level netlist, additional synthesized logic, and external test vector constraints.

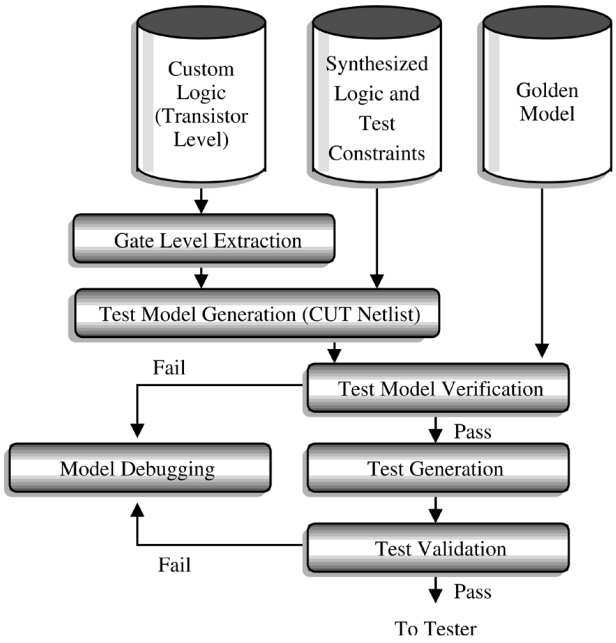


Fig. 1. Test generation flow.

To obtain a gate-level netlist for the transistor-level component, logic extraction is performed for library cells in the synthesized logic and the custom design. Logic extraction is usually an automated process for combinational logic and most sequential circuits [4]–[6], [10], [13], except certain complex cells such as clock generators and scan sequentials that are handled by manual generation of gate-level models. The level of abstraction in the libraries used during extraction, bugs in computer-aided design (CAD) tools, and the human factor may introduce errors in the extracted gate-level model [16], [21], [22].

For this reason, the logical netlist is usually verified using formal techniques and simulation prior to test generation (see Fig. 1). Both verification approaches may not be complete and/or exact due to several reasons. For example, formal methods prove equivalence in the domain of logic values $\{0,\ 1,\ X\}$, but they may not prove equivalence in the $\{0,\ 1,\ X,\ Z\}$ space where the design is exercised during test generation when high impedance values may also be included. Additionally, formal equivalence- and simulation-based approaches can require excessive runtimes due to a large number of validation tests. This limits the amount of validation/verification allowed at this stage. Consequently, some extraction errors may be carried forward to test generation.

Manuscript received August 5, 2005; revised December 19, 2005.

Y.-S. Yang is with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada (e-mail: yangy@eecg.toronto.edu).

A. Veneris is with the Department of Electrical and Computer Engineering and the Department of Computer Science, University of Toronto, Toronto, ON M5S 3G4, Canada (e-mail: veneris@eecg.toronto.edu).

P. Thadikaran and S. Venkataraman are with Intel Corporation, Hillsboro, OR 97124 USA (e-mail: paul.thadikarana@intel.com; srikanth.venkataraman@intel.com).

Digital Object Identifier 10.1109/TVLSI.2006.878346

Following test generation, the generated tests are again verified against the golden model to ensure test correctness, which is a process known as *test (vector) validation*. Tests may fail in this step due to errors introduced in model generation but not identified during model verification. When the test model fails, whether this is during verification or afterwards during test validation, manual analysis of the failures is the current industrial practice, also shown in Fig. 1. This is a daunting task for the test engineer that may delay test delivery to the factories and/or may reduce their quality.

In this study, we describe a set of extraction errors that are typical in test model generation for the pipeline component(s) of high-performance designs that have high frequencies (≥ 1 GHz), consume little power, and meet strict reliability and performance constraints [2], [3], [7], [8], [11]. Such errors may occur because of erroneous module mapping during extraction and because of library specification inaccuracies when complex hardware such as memory elements, multiple clock domains, and tri-state devices are integrated together. The manual interference of the designer/test engineer with the design may also introduce errors.

We also develop an automated simulation-based debugging solution that handles single extraction errors in the pipeline of test models. The proposed solution does not assume that there is any correspondence of primitives between the transistor-level schematic and the test model. It also works on extracted gate-level descriptions with no (and/or partial) state equivalence information to the transistor-level representation. Lack of state/primitive equivalence information between the test model and the transistor-level schematic increases the problem complexity and the debugging effort by the designer. It should be noted that the nature and the effects of extraction errors are radically different from the well-examined topic of design errors presented in [1]. For example, unlike the error types described here, the error model proposed by Abadir *et al.* [1] does not include primitives such as flip-flops and tri-state devices, and it does not apply to functional errors that arise from erroneous timing of the circuit. Therefore, design-error debugging techniques [18], [19] may not be applicable/efficient when applied to this problem.

An extensive suite of experiments on designs with similar architecture to that found in industry demonstrates the practicality of the approach. Methods such as the one presented here benefit the testing of microprocessors, ASICs, and SoCs. They help in identifying model inaccuracies at early stages of the design cycle, reducing test delivery turnaround time and improving test model generation.

The remaining paper is structured as follows. Section II presents the circuit architecture and gives more details about the intricacies of the particular problem. Section III contains a study of different types of extraction errors that are typical in a modern industrial environment for high-performance integrated circuits. Section IV describes the debugging methodology, and Section V presents the experiments. Section VI concludes this work.

II. Preliminaries

Extraction builds a logic representation, known as the test model, from a custom transistor-level block [4]–[6], [10], [13]. This representation is later used for test generation. This work assumes that the extracted netlist contains (N)AND, (N)OR, NOT, and tri-state buffer and flip-flop primitives. For simplicity, we assume that the reset event for all flip-flop memory elements with a reset line is positive-edge-triggered [15]. Flip-flops with level-sensitive reset are not implemented, but the proposed debugging algorithm is not affected, as explained later in the paper. Circuits are simulated under a five-logic value set $\{0,\ 1,\ X,\ X^*,\ Z\}$, where logic X represents the unknown value and X^* denotes a different type of unknown value utilized in debugging. A logic Z represents a *high-impedance* state tri-state devices generation. However, for simulation purposes, a logic Z is replaced with an unknown X. For example, if a tri-state buffer feeds a NOT gate, the output of that gate will be an unknown X.

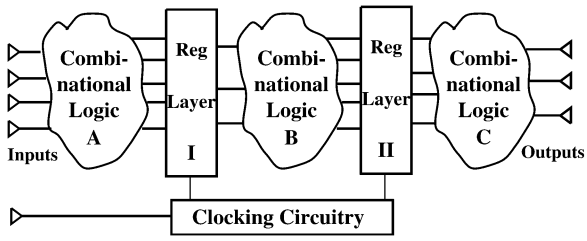


Fig. 2. Benchmark architecture.

In implementation, a zero delay simulator is used for all combinational circuitry. The proposed debugging algorithm will work for most other delay models under a simple assumption as discussed in Section IV. For the primitives involved in the extracted netlist, we define the controlling value of an (N)AND ((N)OR) to be a logic 0(1). We say that a line is *sensitized* to an error by input vector v if its logic value is changed in the presence of the error f for the input vector. A path from a line to a primary output or a register composed only by sensitized lines is called a *sensitized path* [9].

A. Circuit Architecture

The debugging methodologies developed in Section IV operate on *strictly pipelined* sequential circuitry that it is found in most cores of custom high-performance core modern designs. This architecture is shown in Fig. 2 for a two-stage pipeline, where combinational logic A, B, and C are completely separated by layers of registers I and II that contain memory elements. We refer to these parts of the design as the *core circuitry*. Since we assume that there is no feedback in the circuit, all test vectors used in experiments are p-clock-cycle-long input test patterns where p is the number of registers in the particular circuit. As we explain later in the paper, the method may or may not work in the presence of the feedback.

The *clocking circuitry* of the design is also shown in that figure. The difficulty in implementing designs operating at high frequencies is that not only do high frequencies cause high clock skew and jitter, but they also increase the power consumption. To maintain performance, minimize power, reduce noise, and lower clock skew and jitter, contemporary clock systems distribute global clocks at lower frequencies, generate faster local clocks, and enable multilevel clock gating [2], [3], [7], [8], [11]. Therefore, different types of clock manipulator components are

used to generate local clocks with required frequencies, and manual extraction of such complex components is a process which is particularly prone to errors.

B. Problem Characteristics

Extraction errors occur frequently in test model generation of high-performance designs in contemporary nanometer $(< 100$ nm$)$ technologies where many libraries that contain various instantiations of an element with different characteristics are utilized. Real-life information about the frequency of extraction errors in a typical industrial design cycle shows that, for every library that contains a few hundred modules, the extraction process is very likely to introduce an error in the test model for the 180-, 90-, and 65-nm technologies. Once the model is verified to be erroneous, the impact of these errors in the design cycle varies from a few labor days to more than one labor week of manual debugging by the test engineer. Clearly, this is a resource-intensive process that increases the costs, and it may jeopardize prompt test delivery to the factories.

There are several sources of extraction errors during test model generation [21], [22]. Logic extraction may introduce errors in the final flattened netlist due to *erroneous module mappings* by the designer or because of bugs in the automated CAD tools. Additional sources of errors are *functional mismatches* in the definition of module operation constraints between different libraries [e.g., simulation library, synthesis library, physical design library, and automatic test pattern generation (ATPG) library]. In this case, the mapping is correct, but corner-case specifications about the operation of the module(s) are interpreted differently by various libraries. In both cases, these errors may change the functionality of the extracted test model.

Notice that, due to the nature of the extraction process, a single error contained in a module definition may produce multiple erroneous *instantiations* in the final netlist. Modern design methodologies are traditionally based on a hierarchical composition to simplify the design efforts [15]. Every circuit is a collection of blocks, and each block contains a varying number of more basic components which are replicated instances of simpler *modules*. In this manner, a single module is usually instantiated into many copies in the final netlist.

This hierarchical information is useful during extraction. Instead of working on individual instances of a module piece by piece, the process extracts the gate-level model of a module and then it applies the extracted model to all instances of that module according to an *instantiation module mapping* information using pattern matching, ruled-based or symbolic approaches [4]–[6], [10], [13]. This also implies that an error during the extraction on a single module may result into multiple errors in the final flattened netlist. This situation is depicted in Fig. 3, where a single error in the mapping for Module 1 "translates" to two errors in the final netlist. Due to the presence of multiple error effects, diagnosis may become a challenging task because the solution space grows exponentially [18]. However, the instantiation module mapping information is usually known to the engineer. In other words, recognizing one line that is a source of error is sufficient to identify the remaining lines and rectify the design by replacing *all* erroneous modules in the netlist.

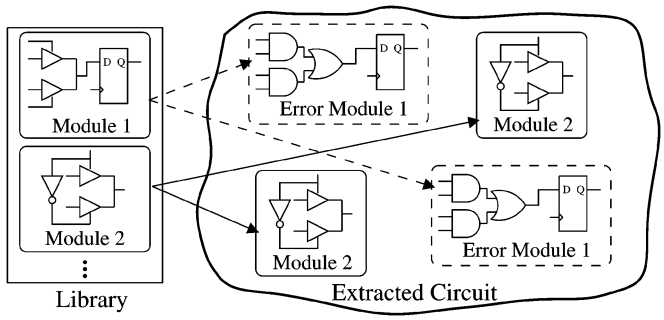


Fig. 3. Multiple error instantiation.

An additional challenge when debugging extraction errors is the fact that *state equivalence* information between the transistor level and the test model may not be fully available to the engineer. Extraction tools often locally convert the transistor components into logic components without recording information about this mapping process. Hence, when designers receive the extracted netlist, they may not have a one-to-one structural correspondence between registers of the test model and the transistor-level schematic. Full state equivalence between the schematic and the test model is usually observed in blocks designed with standard logic synthesis tools. Partial state equivalence (i.e., equivalence only at design block interfaces) and/or no state equivalence is a common characteristic of custom/manual designed register files and datapaths. On the average, our experience indicates that 40%–60% of the equivalence information remains available.

Due to the lack of this information, the complexity of the diagnosis is increased both in space and in time [19]. For example, registers cannot serve as pseudoinputs/outputs, and only responses at primary outputs can be used to verify the correctness of the implementation. Furthermore, without access to the registers, it may take several clock cycles to activate the error and propagate the effects to the outputs. Consequently, diagnosis involves analysis on the states of the circuit in several different time frames that adds to the complexity of debugging.

III. Extraction Error Types

Here, we study sets of extraction error types that are common in high-performance cores, for example, for microprocessors, SoCs, and ASICs, in the industry today [2], [3], [7], [8], [11], [12], [17]. With respect to the terminology and circuit architecture from Section II, we divide these error types into two categories, core error types and clocking circuitry error types, and we present them accordingly.

A. Core Error Types

1) Reset Synchronicity Error: A large portion of the silicon area in contemporary designs is dedicated to storage [15]. Memory elements such as flip-flops commonly have either a synchronous or an asynchronous reset, shown in Fig. 4. As illustrated in Fig. 4(b), a D-flip-flop (DFF) with asynchronous reset is evaluated as soon as an event arrives at its reset port, whereas a flip-flop with synchronous reset cannot change its value until a clock-edge occurs. During test model extraction, a library element that contains a flip-flop with an asynchronous

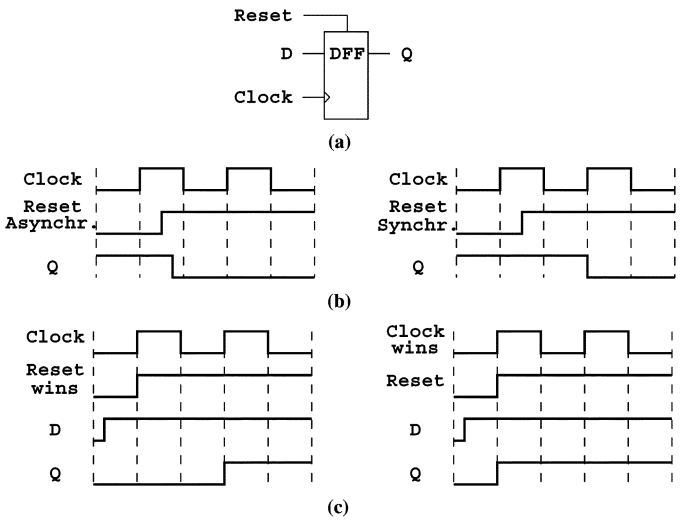


Fig. 4. (a) D-flip-flop. (b) Reset synchronicity. (c) Reset-clock contest.

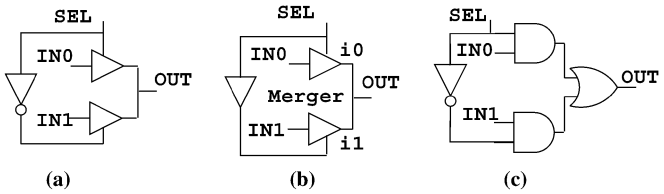


Fig. 5. MUX implementations.

TABLE I
MERGER TRUTH TABLE

i_0	i_1	Merger	i_0	i_1	Merger
0	0	0	1	0	X
0	1	X	1	1	1
0	X	X	1	X	X
0	Z	0	1	Z	1
X	0	X	Z	0	0
X	1	X	Z	1	1
X	X	X	Z	X	X
X	Z	X	Z	Z	Z

TABLE II
MUX ERROR WHEN SEL $= X$

		OUT	
IN0	IN1	Tri-state	AND/OR
0	0	X	0
0	1	X	X
1	0	X	X
1	1	X	X

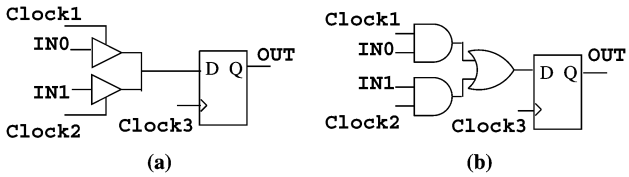


Fig. 6. MUX-latch implementations.

reset may be mapped to an element that contains a flip-flop with a synchronous one and *vice versa*, resulting in a functional error, that is, both directions of the error in Fig. 4(b) may occur. Clearly, the waveforms from Fig. 4(b) indicate that, when such a mismapping occurs, it may result in a functional error in the test model.

2) Reset-Clock Race Error: Reset-clock race happens due to module specification mismatch in the design libraries used during test model generation. Different libraries may make different assumptions about the winner of the contest, which may result in an erroneous implementation, shown in Fig. 4(c). The right-hand side of that figure depicts the situation where Q is initially 0 and the clock prevails, whereas in the left part of the figure the reset line wins the contest. This type of error can arise from the contest of other lines (e.g., set or hold) as well, and it may occur in both directions as in reset synchronicity.

3) Multiplexer Implementation Error: A multiplexer circuit allows more than one signal to be placed on the same line via time-division multiplexing. A typical two-input multiplexer is implemented with two tri-state buffers and a logic NOT gate, as shown in Fig. 5(a). The inverter between the two select ports (SEL) of the tri-state buffers ensures that line OUT is driven by only one input signal at any time.

Two erroneous mappings may occur during extraction, as shown in Fig. 5(b) and (c). In the first one, the inverter is replaced by a buffer. In that case, when SEL goes high, both tri-state buffers are enabled, causing a *merger* at the output OUT. Table I summarizes the truth table for merger in Fig. 5(b). Entries which are not bold underlined may cause a discrepancy in the output logic. Furthermore, when one of IN0 and IN1 is driven by logic 1 and the other by 0, a hazardous path is created between power and ground.

In the other case [see Fig. 5(c)], instead of tri-state buffers, the multiplexer is erroneously mapped to an implementation with logic AND and OR gates. While the logic function of the two implementations agree for most input combinations, they differ when SEL is assigned to X. As shown by the first row in Table II, the implementation with AND and OR gates drives 0 on OUT, while the tri-state buffer implementation drives X. This difference in the simulation result may lead ATPG to generate wrong test vectors. Clearly, any mismapping between these three implementations will result in an erroneous test model.

4) Multiplexer-Latch Implementation Error: Multiplexer-latch (MUX-latch) design allows the sampling of signals from two nonoverlapping clock domains. As shown in Fig. 6(a), data from clock domains Clock1 and Clock2 are multiplexed into the D-flip-flop, which is clocked by Clock3, the frequency of which is the sum of the frequencies of Clock1 and Clock2. The multiplexer in a MUX-latch design can be implemented with tri-state devices or AND/OR gates, as illustrated in Fig. 6(b). These two implementations have different logic functions when Clock1 and Clock2 overlap, a situation which may happen during scan test. In the AND/OR implementation, the flip-flop latches IN0 OR IN1. In contrast, X can be latched in the tri-state buffer implementation.

B. Clocking Circuitry Error Types

1) Gated Clock Error: Modern devices impose strict power consumption and reliability requirements. Since not all design components may always need to operate simultaneously, the gated clock scheme of Fig. 7(a) is commonly used to disable

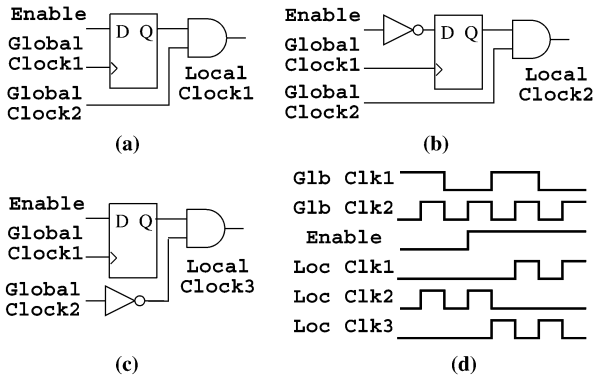


Fig. 7. Gated clock implementations.

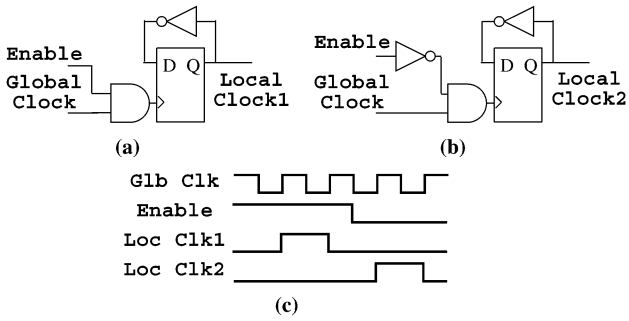


Fig. 8. Frequency-divider implementations.

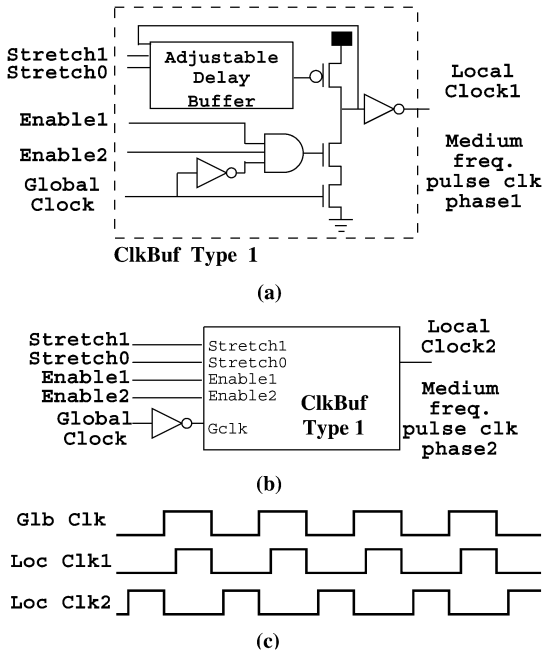


Fig. 9. Clock pulsed buffer implementations.

clocking of temporarily inactive components and save power [2], [7], [8], [11]. The hardware in that figure is built in such a way so that `Local Clock 1` operates in the frequency of `Global Clock 2` as long as `Enable` is at logic 1.

Different implementations of the gated clock are shown in Fig. 7(b) and (c). As can be seen, these implementations differ on the position (if any) of a `NOT` gate. An extraction process may erroneously replace one implementation with another during module mapping. The waveforms for all implementations are found in Fig. 7(d). We observe that, for the same input, they all produce different `Local Clock` results.

2) Local Clock Frequency-Divider Error: Frequency dividers are used in approximating domain global clocks to generate integral local clock frequencies that drive various design blocks at different speeds [3], [7], [11]. Fig. 8(a) and (b) contains common hardware to implement such dividers. If the extraction process maps erroneously between these two frequency-divider module implementations, the new `local clock` generated will be enabled at a complementary phase [see Fig. 8(c)]. This may result in a circuit malfunction because the memory elements of the core will lock/propagate different logic values.

3) Local Clock Pulsed Buffers Error: To achieve high performance, reduce power, and improve reliability, critical design blocks [e.g., arithmetic logic unit (ALU)] may be required to operate at higher (nonintegral) frequencies than this of the global clock, while noncritical blocks may operate at slower frequencies. Local clock buffers are used to generate clocks of a desired frequency for various design blocks [3], [7], [11]. These buffers are driven from global clocks through delay-matched taps, and they are available as pulsed and nonpulsed buffers. Nonpulsed drivers simply buffer the input global clock, and they usually present no problem to the extraction process.

Fig. 9(a) shows a medium pulsed clock driver. At the rise of the global clock, the pull-down path is asserted to generate the rising edge of the local clock. At the same time, the self-reset pull-up path is asserted to generate the falling edge of the clock. The delay buffer is adjustable to permit different types of duty cycle for the output local clock. Variations of the hardware in Fig. 9(a) allow for pulsed buffers that generate slow- and fast-frequency local clocks from global clocks. We omit these hardware descriptions that can be found in [3], [7], and [11]. Additionally, Fig. 9(b) shows the schematic for another medium pulsed clock driver with a phase complementary to that of Fig. 9(a).

During extraction, erroneous mapping or mismatches in library specifications may utilize a different pulsed buffer in place of the other. For example, a medium frequency buffer may be accidentally replaced with a slow-frequency one or with one with inverted phase due to human error. From real life experience, it is unlikely that a fast-frequency buffer will be replaced by a slower one, although the debugging method, presented next, can handle this case. When a pulsed buffer replacement error occurs, the difference in the operating waveforms [see Fig. 9(c)] may change the functionality of the test model, and the input/output vectors collected in test generation may give faulty output responses during vector validation.

IV. Debugging Extraction Errors

When an extracted netlist fails verification or the scan test (see Fig. 1), current practice requires a manual analysis of the failure, which is an expensive process in terms of both time and resources. This section describes an automated simulation-based

model-free [14] debugging procedure in the effect–cause direction [9] that rectifies single extraction errors in designs with full, partial, or no state equivalence. Model-free diagnosis utilizes the unknown logic value X to guide the search for candidate error lines, which is a desired characteristic in view of the complex error behavior seen in Section III.

Automated debugging involves two steps, namely, *diagnosis* and *correction*. Diagnosis proceeds in two phases, path-trace and X^*-simulation, to identify lines in the circuit where error effects may originate. Once done, correction proposes a module modification (replacement) on these lines from the set of errors presented earlier to rectify the design. Since module mapping information is known to the engineer, it suffices for diagnosis to identify a single line driven by an error module. All other lines with erroneous modules can be discovered using the mapping information.

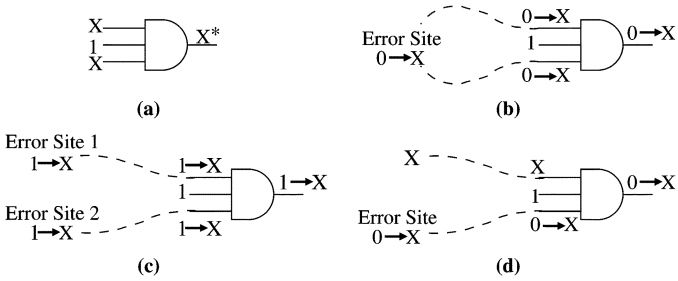


Fig. 10. Path-trace with unknowns.

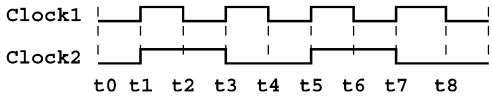


Fig. 11. Example clock waveform.

A. Generalized Path-Trace

A basic ingredient of diagnosis is a *generalized path-trace* procedure. Path-trace is a line-marking algorithm originally proposed by Venkataraman and Fuchs [20] and later redefined by Liu and Veneris [14] to aid fault diagnosis. Path-trace simulates input test vectors with faulty responses, and it marks lines starting from a failing primary output. Here, we use an extension to the procedure in [14] that handles three-value logic $\{0, 1, X\}$ in the view of multiple instances of a single error. It has the following rules.

1) If the output of a gate is marked and all of its inputs have noncontrolling values, path-trace marks all inputs of the gate.
2) If the output of a gate is marked and its inputs have one or more controlling values, path-trace randomly marks one such input.
3) If a branch is marked, then the stem of the branch is marked.
4) If the output of a gate is marked and its inputs have logic unknown and/or noncontrolling values, path-trace marks all inputs that have a logic X.
5) If the gate is a memory element, its input, reset, and clock lines are always marked.

Rules 1)–3) are taken directly from [14] and [20], and they guarantee to mark at least one line which is a source of error. Rule 5) is novel to this study, and it extends the ability of path-trace to mark lines in a circuit with memory elements. Finally, rule 4) is a new rule added to handle the case of single errors instantiated multiple times, as explained in the example below.

Example 1: Consider the situation depicted in Fig. 10(a), where the output of the gate is marked with an "*" by path-trace. There are three separate cases path-trace may continue marking as it moves towards the primary inputs. In the first case depicted in Fig. 10(b), both unknown values X at the gate inputs are a result of the same error site. Randomly marking one of them will eventually lead to the same error site later. In the second case [see Fig. 10(c)], the inputs are driven by two error sites which are instances of the same extraction error. In this case, randomly selecting one of the inputs will mark at least one of the error sites which is sufficient to diagnosis. In the last case shown in Fig. 10(d), randomly marking an input, as required by [14] and [20], does not work. In this case, there is only one error site. When path-trace inspects this gate, it will not be able to know which input with a logic X is caused by the error and which is not. Hence, it may miss the error site, but rule 4) helps avoid this situation. ◇

Due to the pessimistic nature of path-trace [20], it can be shown that it *always* marks at least one line driven by an error module. We omit the proof of this claim which is similar to the one found in [14] that contains another three-valued variation of path-trace. However, when state equivalence information is not fully available, additional care needs to be taken once path-trace marks the output of some memory elements (registers). Although the procedure correctly marks (among other lines) the input of a flip-flop, the logic values in the circuitry of the fan-in cone of this flip-flop may be obsolete. This is because of the different clock domains that trigger independently and may change the values of the lines in the fan-in cone of the marked flip-flop.

To elaborate further, consider the architecture in Fig. 2. For the sake of simplicity, in the remaining section, we explain the solution in terms of a two-stage pipeline since deeper pipelines can be handled similarly. Assume that the memory elements in register layers I and II are triggered by two independent clock domains, as shown in Fig. 11. Furthermore, assume that some flip-flop F in layer II driven by `Clock1` is marked by path-trace for an error observed at the primary outputs at time frame t_7. In other words, path-trace started from an erroneous output marking lines in combinational circuitry C and reaches the output of F.

The reader may verify that any error effect that propagates to F did so using logic values in combinational circuitry B that corresponds to time-frame t_5 (or before), that is, prior to the trigger of `Clock1`. This is because local clocks (including `Clock1`) may be shared with register layer I and overwrite the values in circuitry B past time frame t_5 when `Clock1` triggers to store values in F. Since these values are now lost, we need restore them so that path-trace continues marking lines at the input of F correctly. Equivalently, if logic values in circuitry B are updated correctly and a memory element in register layer I is marked, we need to restore all values of the gates in combinational logic A at time frame t_3.

The above discussion indicates that, for a two-stage pipeline, one needs to store the logic values of each logic primitive of the

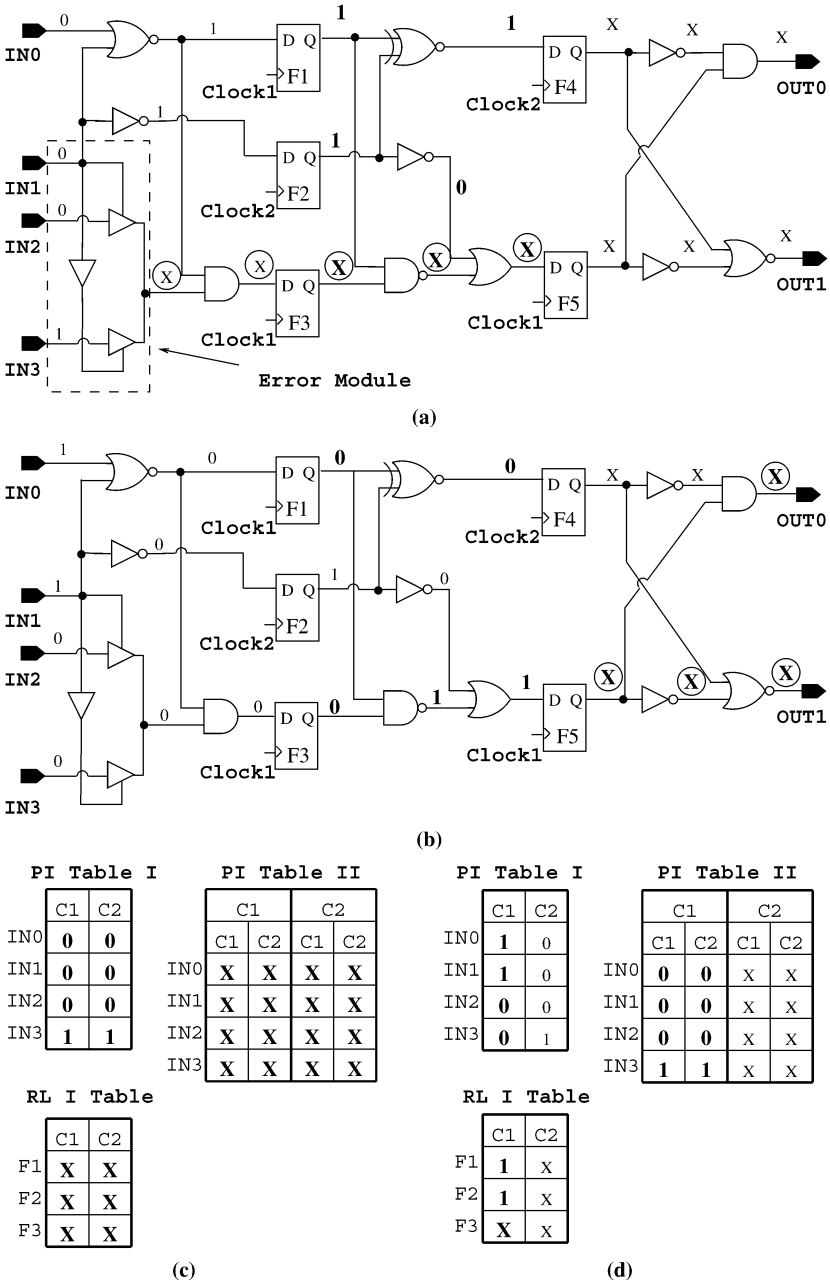


Fig. 12. Circuit and storage tables for example 2: (a) first vector, (b) second vector, (c) storage tables at t_1, and (d) storage tables at t_3.

core for all pipeline stages for one or two clock cycles before the time frame the error is observed to aid path-trace. In the proposed implementation, we store the appropriate logic values in the registry and the primary input of the circuit using problem-specific *storage tables* updated appropriately throughout simulation of the circuit. Once an erroneous primary output is observed, the storage table values are simulated in the combinational block on which they fan out to restore the logic values on the primitives at correct time-frames so that path-trace can continue marking. We illustrate the handling and use of these tables with the following example.

Example 2: Consider the circuit in Fig. 12(a) with register layers $RL\ I = \{F_1, F_2, F_3)$ and $RL\ II = \{F_4, F_5\}$ and combinational cores A, B, and C that contain a single error with a single instantiation. This is the multiplexer implementation, shown within the dotted box, that needs to be replaced with the one shown in Fig. 5(a) that contains an inverter. Assume that the two clocks of the design have the timing characteristics from Fig. 11.

In the same figure, the storage tables are shown. Observe that a pair of tables is dedicated to the primary input (PI) and a single table to register layer $RL\ I$. In detail, the column marked $C1$ $(C2)$ in `PI Table I` holds the value of the primary inputs before the last trigger of `Clock1` (`Clock2`). Equivalently, the column marked $C1(C2)$ in `RL I Table` holds the value of flip-flops in $RL\ I$ before `Clock1` (`Clock2`) triggers. On the other hand, the column marked $C1-C1$ in `PI Table II` holds the value of the primary input before two consecutive `Clock1`

triggers. Intuitively, this is necessary because, if error effects propagate to primary outputs after the trigger of Clock1 at time t_7 (Fig. 11), for example, path-trace will require the value of the primary inputs at time frame t_3 to simulate and restore logic values correctly in combinational core A.

All entries of PI Table I are initialized to a logic unknown X. Assume that at time t_0 the first input vector 0001 is applied at the input. This vector excites the error, and error values propagate to $RL\,I$. At time t_1, both clocks trigger and all of the tables are updated as shown in Fig. 12(b) to reflect logic values of the memory elements and the PI *prior* to that clock trigger.

RL I Table is updated first with the original values of the flip-flops in $RL\ I$. In this case, all memory elements are initialized to X. Since both clocks trigger, all columns of the PI Table II are overwritten by two copies of PI Table I. As explained earlier, the first (last) two columns of PI Table II hold the value of the primary input prior to the last Clock1 (Clock2) trigger. Finally, both columns of PI Table I are updated with the input vector. After the tables are updated, the values are latched in $RL\ I$ and simulated in the circuit for time frame t_1. In that figure, the faulty values at the fan-out cone of the error site are shown in circles.

At time t_2, assume that a second input vector 1100 is applied, as shown in Fig. 12(c). At time t_3, only Clock1 triggers, and portions of all of the tables are updated as shown in Fig. 12(d). Since only Clock1 triggers, only respective columns of the three tables are updated as explained earlier and they are shown in bold. Observe that column $C1$ of RL I Table memorizes the logic values stored at the $RL\ I$ flip-flops at time frame t_1 [circuitry in Fig. 12(a)]. It can be seen that, after the tables are updated and the values are latched/simulated, error effects are observed at the primary output of the circuit.

Once the errors are observed, path-trace is performed on combinational core C, and it marks F_5. Since this flip-flop is controlled by Clock 1, the primitives in register layer $RL\ I$ are updated with the values in the C1 column of the RL I Table in Fig. 12(d). Next, combinational logic B is simulated to restore values and path-trace continues in B to stop at F_3. Since F_3 is controlled by Clock 1, PI values are restored from the $(\mathrm{C1}, \mathrm{C1})$ column of PI Table II. Once these input values are simulated in block A, path-trace continues and marks the error location. ◇

Algorithm 1 Table Update and Enhanced Path-Trace

1: T_{PI}_1 := PI Table I

2: T_{PI}_2 := PI Table II

3: T_{RL} := RL I Table

4: C := Clocks triggered at time t

5: RLI := the set of registers in Register Layer I

6: PI := the set of primary inputs

7: **procedure** UPDATE_TABLE(T_{PI}_1, T_{PI}_2, T_{RL}, C, RLI, PI)

8: **for all** $c \in C$ **do**

9: **for all** $f \in RLI$ **do**

10: $T_{RL}(c, f) \leftarrow$ the state of f

11: **end for**

12: $T_{PI}\_2(c) \leftarrow T_{PI}\_1$

13: **end for**

14: **for all** $c \in C$ **do**

15: **for all** $i \in PI$ **do**

16: $T_{PI}\_1(c, i) \leftarrow$ the value of i

17: **end for**

18: **end for**

19: **end procedure**

20: **procedure** ENHANCED_PATH_TRACE(T_{PI}_1, T_{PI}_2, T_{RL}, C, RLI, PI)

21: path-trace on comb. circuitry C

22: $RL2List \leftarrow$ marked registers in $RL2$

23: **for all** $r_1 \in RL2List$ **do**

24: $c_1 \leftarrow$ clock of r_1

25: read and place values in $T_{RL}(c_1)$ on registers in $RL1$

26: simulate and path-trace comb. circuitry B

27: $RL1List \leftarrow$ marked registers in $RL1$

28: **for all** $r_2 \in RL1List$ **do**

29: $c_2 \leftarrow$ clock of r_2

30: read and place values in $T_{PI}\_2(c_1, c_2)$ on PIs

31: simulate and path-trace on comb. circuitry A

32: **end for**

33: **end for**

34: **end procedure**

Example 3: The diagnosis effort for the circuit in Fig. 12 is significantly simplified if full state equivalence information is provided. In that case, after simulating the first test vector at time t_0, the error value propagates and it is observed at flip-flop F_3. Since this flip-flop serves as a pseudo primary output, path-trace can start at F_3, which marks the erroneous module, and there is no need for the second test vector. ◇

Algorithm 1 contains pseudo code for the procedure UPDATE_TABLE that updates the storage tables at every clock trigger during simulation for a two-stage pipelined test model. Longer pipelines require additional tables and they are handled similarly. The same figure contains the pseudo code for ENHANCED_PATH_TRACE that uses these values to assist the five rules of path-trace in diagnosis as defined earlier.

UPDATE_TABLE works as follows. For every clock $c \in C$ triggered at time t, the RL I Table is updated first with the current values of the flip-flops in register layer I (line 10). Next,

the PI Table II is updated by copying PI Table I to the column which is main-indexed with c (line 12). Intuitively, the columns of PI Table II under the same main index contain the values of the PIs prior to the last indexed clock triggered. Finally, the PI Table I is updated with the input vector (line 16). In our implementation, we use a parallel bitwise implementation for both the storage tables and for the simulation retrieving as many test vectors as the length of the computer word [9].

The values stored in the tables are used appropriately by ENHANCED_PATH_TRACE to restore logic values in the core circuitry and assist path-trace during diagnosis. When path-trace marks the registers in register layer II, it reads the proper entry in RL I Table and retrieves the correct values in combinational circuitry B by simulating the circuitry with the values read from the table. Then, path-trace proceeds by marking in the combinational circuitry B (lines 24–26). When it reaches the registers in register layer I, it again reads the entry in PI Table II according to clocks controlling the registers on the path and restores the states of the lines in combinational circuitry A. Path-trace continues marking in core circuitry A and terminates when it reaches a primary input of the circuit, as explained earlier. These actions are taken in lines 29–31.

The above procedures are utilized when no state equivalence information is available. In the cases of partial and full state equivalence, path-trace can start marking from erroneous primary outputs as well as memory elements that propagate erroneous logic values and their state equivalence information is available. We omit this code which is a straightforward extension of the one presented earlier.

In the final paragraphs of this section, we analyze memory requirements for the various tables. In this analysis, we partition the circuit to allow the primary input to be at stage 0, the register layer I at stage 1, and so on. We also assume that, for stage k, there are f_k memory elements (primary inputs) and c_k local clocks.

Using the above notation, it can be shown that the *size* (i.e., number of table entries) of the largest table at stage k is $f_k \prod_{i=k+1}^{s} c_i$, where s is the maximum number of pipeline stages. Recall that each stage k maintains additional smaller tables for all subsequent stages. Therefore, the size of the *total* number of tables at stage k is found to be $\sum_{j=k+1}^{s} f_k \prod_{i=k+1}^{j} c_i$. Using this result, the total number of tables for the complete design is $\sum_{l=0}^{s-1} \sum_{j=l+1}^{s} f_l \prod_{i=l+1}^{j} c_i$.

To further analyze the memory requirements as the number of pipeline stages increases, for the sake of simplicity, we assume that the number of registers and the number of local clocks for each register layer in every pipeline stage is upper bounded by f and c, respectively. In other words, $f_i \leq f$ and $c_i \leq c, \forall\, i = 1 \ldots s$. Using this upper bound, the total number of table entries is $T(s) \leq \sum_{l=0}^{s-1} \sum_{j=l+1}^{s} fc^{j-l} = sfc + (s-1)fc^2 + (s-2)fc^3 + \cdots + 2fc^{s-1} + fc^s$. It can be seen that, for every increase in the number of pipeline stages s, the number of table entries increases roughly by factor of c. For example, $T(3)/T(2) = (3fc + 2fc^2 + fc^3)/(2fc + fc^2) \doteq fc^3/fc^2 = c$, and so on, or, in general, $T(s) \leq O(c^s)$.

It is seen that the size of the table entries relates to the number of clocks per pipeline stage and the total number of pipeline stages in an exponential manner, a fact that may require prohibitive amounts of memory for large industrial designs with very deep pipelines. This is expected since the algorithm handles a sequential model with no state equivalence information to the transistor level. However, in practice, deep pipelines are usually designed in blocks of smaller (pipelined) stages to ease test generation, verification etc [15]. Additionally, the number of local clocks and the number of pipeline stages is a small fraction to the total number of circuit lines. Finally, the trend today in high-performance designs is multithreading on small-sized pipelines due to power and reliability concerns. Therefore, in most practical cases, the debugging algorithm is expected to be memory efficient.

B. X^*-Simulation

In most model-dependent diagnosis algorithms, the effects of the faults/errors are explicitly specified [9], [14]. During diagnosis, a model-dependent algorithm usually performs one logic simulation step for every error model manifestation on each suspect line. Therefore, this type of diagnosis is more suitable for simple faults/errors where their effects are easy to enumerate [14]. For example, the effects of a stuck-at fault are modeled with only two values 0 and 1 and, for each suspect line l, the algorithm can perform two simulations, l stuck-at-0 and l stuck-at-1. to see the effects of the candidate fault. However, for complicated faults/errors, such as the extraction errors presented here, the cardinality of the error models may be large, and it may be computationally expensive to simulate each such effect during diagnosis. In this case, model-free diagnosis seems to serve better in tackling the problem [14].

Model-free diagnosis simulates a logic unknown value X on candidate error lines to capture all possible paths for error propagation [14]. In the proposed debugging algorithm, after path-trace returns a set of candidate error locations, the algorithm performs this X^*-simulation step using failing input test vectors. Because of multiple error sites in the netlist, the true candidate error locations must be marked by path-trace at least $\lfloor |V|/N \rfloor$ times, where $|V|$ is the cardinality of failing input test vectors and N is the number of erroneous module instantiations. This claim is proved in terms of a simple theorem in [14] using the pigeon-hole principle, and we refer the reader to that paper for details. Furthermore, recall that these multiple error sites are instances of one single module. In other words, we can prune the size of the candidate list further using the following rule. *The sum of path-trace marks for all instantiations of a candidate error module must be greater or equal to* $|V|$. X^*-simulation is performed on modules that qualify this rule.

Since some error types involve the logic unknown X, we introduce a second unknown value X^* and simulate this value on the candidate error locations. Under this scenario, if a gate primitive has an input with controlling value, then this input prevails, otherwise value X^* prevails over any other input. For example, a two-input AND gate with fan-in values of 0 and X^* evaluates to 0 while one with values X and X^* or 1 and X^* evaluates to X^*. A module qualifies for correction if this simulation of the value X^* propagates to all erroneous primary outputs [14].

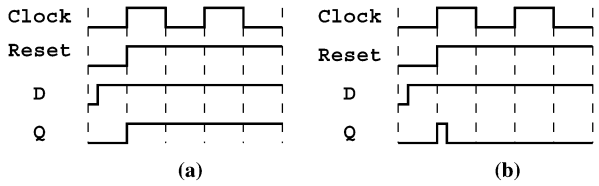


Fig. 13. (a) Edge-sensitive reset. (b) Level-sensitive reset.

C. Correction

After diagnosis returns a set of candidate error lines, debugging enters correction. Since we assume that the module mapping information is available, correction tries to rectify all instantiations of each candidate error line. In detail, for each set of module instantiations, all potential error modifications from Section III are exhaustively enumerated on these sites, and the circuit is resimulated for all input test vectors. Corrections that return correct primary output results for the test vectors qualify debugging. For example, once diagnosis returns the error site in dotted lines in Fig. 12(a) from Example 2, correction will enumerate candidate correction models from Fig. 5(a) and (c) to find that both modules qualify.

D. Level-Sensitive Reset

As mentioned in Section II, the reset signals of flip-flops can be either edge-sensitive or level-sensitive. Even though the edge-sensitive reset model is chosen to model the reset event in this study, the proposed debugging algorithm is not affected if level-sensitive reset is used. Nevertheless, the reset-clock race error behaves differently.

Fig. 13 shows the waveform of the flip-flop when an asynchronous reset event and a clock event trigger concurrently and the clock event prevails. For the edge-sensitive reset [see Fig. 13(a)], the effective duration of the reset event exists only at the moment of the edge transition. The flip-flop will not reset until the reset line has another positive-edge trigger. Comparing it with the level-sensitive reset, the reset event stays active until the value of reset goes low (assuming that reset is active high). As shown in Fig. 13(b), once passing the clock edge, the flip-flop will reset if reset is still active. Consequently, a glitch will be created. In this case, the winner of the contest does not affect the value of the flip-flop, so this error type cannot happen.

E. Other Delay Models

During simulation, we assume that all gate primitives have a zero-delay, that is, the output of a gate is evaluated as soon as the new values arrive at the inputs of the gate. There are several other delay models used for simulation, such as *unit-delay*, *multiple-unit-delay*, and *minmax-delay* [9], [15]. The proposed debugging procedure will function correctly for these models as long as the following condition holds: *the minimum clock period is no shorter than the longest delay path in any combinational circuitry block.*

This condition ensures that no new values arrive at the inputs of the combinational circuitry before all gates have been evaluated. If this condition fails, we cannot assure that the storage tables will maintain the correct values for a particular time frame. In a real-life design context, it is realistic to assume that this condition is valid or the behavior of the design will be hard to predict accurately [9], [15].

TABLE III
BENCHMARK CHARACTERISTICS

crt. name	prim. count	PI count	# of pipeline	total # FFs	total # libraries
A	2381	35	2	50	32
B	5644	50	2	54	32
C	8854	93	2	115	30
D	16467	106	2	169	32
E	23091	103	2	182	31
F	6404	50	3	86	32
G	13206	93	3	194	31
H	21770	106	3	249	31
I	10896	45	4	156	30
J	21129	40	4	140	32

F. Handling Feedback

In this paper, we assume that there is no feedback in the design. Depending on the state equivalence information availability, the feedback may introduce different degree of difficulty. For example, in the full state equivalence case, the proposed algorithm is not affected and an error(s) can be identified right away at the register layers and the primary outputs so that path-trace still marks the erroneous module(s).

For the cases of no and partial state equivalence, the debugging problem becomes more complex because an erroneous logic value may oscillate through the feedback several times before getting observed. For path-trace to operate correctly, the different logic values of the state elements at the beginning of each oscillation need to be recorded. In the current implementation, these values may not be available because the storage tables record state element values for as many clock cycles as the length of the pipeline. Moreover, path-trace needs to be redefined to trace along the feedback lines. Therefore, the proposed approach may not work in the presence of a feedback in a design with no or partial state equivalence information.

V. EXPERIMENTS

Tests are carried on two-, three-, and four-stage pipelined sequential designs with the architecture from Fig. 2 that resembles the one found in some high-performance circuits today. Recall that the number of stages indicates the number of register layers that separate the combinational circuitry of the core. These benchmarks do not contain feedback, and they are built by modifying, reusing, and padding circuitry from the ISCAS'85 and ITC'99 family of benchmarks. Their clocking circuitry consists of a scheme of four global clocks that drive 14–18 local clocks. The frequency of the global clock domains is an integral multiple of each other. The circuit name, primitive gate count, flip-flop count, and other characteristics of the designs are shown in Table III. Experiments are conducted on a Pentium 2.8-GHz processor with 2 GB of memory. All run times in this section are reported in seconds.

There are 11 different types of extraction errors presented in Section III. To emulate a real physical synthesis environment,

TABLE IV
FULL STATE EQUIVALENCE

crt. name	# of module			run time (CPU sec)				
	path-trace	X*-sim	corr.	path-trace	X*-sim	corr.	pro-posed	brute force
A	4.5	2.5	1.2	0.1	0.6	1.4	2.8	23.0
B	3.7	1.3	1.0	0.1	0.7	0.4	2.5	34.5
C	2.6	1.4	1.0	0.1	1.6	1.0	4.9	59.0
D	4.6	1.0	1.0	0.1	3.7	2.3	8.8	80.7
E	2.8	1.5	1.0	0.1	3.5	3.8	10.4	87.3
F	4.2	2.2	1.3	0.1	2.1	4.3	9.6	71.5
G	1.7	1.3	1.2	0.1	2.1	2.5	10.1	126.4
H	3.8	1.0	1.0	0.1	8.0	4.8	19.5	161.1
I	3.5	2.5	1.2	0.1	14.4	1.4	41.1	414.7
J	7.5	1.2	1.0	0.1	19.0	4.1	48.0	441.8

TABLE V
PARTIAL STATE (50%) EQUIVALENCE

crt. name	# of module			run time (CPU sec)				
	path-trace	X*-sim	corr.	path-trace	X*-sim	corr.	pro-posed	brute force
A	15.6	6.2	1.2	0.1	1.9	1.2	3.3	36.9
B	3.0	1.8	1.3	0.1	0.9	2.7	5.4	57.5
C	5.2	2.6	1.0	0.4	1.8	2.9	7.5	79.7
D	10.8	2.7	1.2	0.8	8.4	6.7	18.9	113.8
E	1.8	1.4	1.2	0.1	2.2	5.6	11.4	143.1
F	2.4	1.2	1.0	0.1	0.8	0.7	8.3	130.4
G	12.2	5.2	1.4	1.1	7.9	11.5	32.5	224.0
H	9.0	1.8	1.2	0.1	10.1	4.4	30.4	319.1
I	3.8	1.8	1.0	0.3	14.7	1.0	59.5	624.7
J	16.3	2.7	1.0	1.8	16.1	1.3	55.3	806.1

we realize two to three different module libraries for each error type for a maximum of 32 module types (see Table III). In practice, this indicates a set of modules with the same functionality but different physical characteristics. For each circuit, we perform three types of experiments using the debugging algorithm from the previous section where *all* information, *some (partial)* information, and *no* information of state equivalence is known to the algorithm. In the case of partial state equivalence, we randomly utilize 50% of the state equivalence information during debugging.

Each experiment contains averages of 10 runs. In each run, a module is selected at random and all of its instances are replaced by an another module type to change the test model functionality. Debugging results shown here are based on simulation of 700–2000 erroneous vectors with high fault coverage $(> 90\%)$ [9]. Test generation for extraction errors is not a topic of this study. To exhibit the effectiveness of the proposed debugging approach, we compare its performance with a *brute-force* method where the engineer debugs the test model by enumerating exhaustively all module libraries. The brute-force approach is the common manual debugging practice in the industry today when the test model fails.

Tables IV–VI contain results for the full, partial, and no state equivalence cases, respectively, presented in a similar manner as follows. The first column has the circuit name. The next two columns contain the average number of modules that qualify the two steps of diagnosis. The resolution of diagnosis is better appreciated when contrasted with the last column of Table III that contains the total number of modules. Intuitively, the values in that column in Table III are an indication of the effort of the brute-force approach that it exhaustively enumerates all possible

TABLE VI
NO STATE EQUIVALENCE

crt. name	# of module			run time (CPU sec)				
	path-trace	X*-sim	corr.	path-trace	X*-sim	corr.	pro-posed	brute force
A	30.8	3.5	1.2	3.7	6.1	1.7	12.6	42.4
B	24.0	16.0	1.5	1.7	5.0	9.4	18.2	85.8
C	22.7	7.3	1.2	6.9	10.6	12.0	32.2	104.8
D	19.2	2.4	1.2	19.7	13.6	8.0	45.4	171.4
E	23.3	5.8	1.3	36.2	17.7	17.3	75.5	262.6
F	31.5	29.5	1.2	73.3	22.2	54.5	174.5	292.5
G	26.5	6.8	1.0	43.1	14.5	11.3	81.2	245.2
H	31	7.7	1.0	152.5	29.0	18.0	216.1	357.1
I	23.5	5.8	1.8	215.2	23.1	6.1	297.7	621.1
J	29.0	8.5	1.1	877.8	36.0	20.5	984.8	798.9

module instantiations to fix the design. We observe that the proposed diagnosis has better resolution than the brute-force one as it eliminates more than 70% (on the average) of useless module enumerations. For example, in circuit A with full equivalence information available, the proposed debugging algorithm simulates only 2.5 modules while the brute-force method may simulate up to 32 modules.

Column four of the tables contains the number of modules that qualify correction. Since we assume that the instantiation module mapping information is available to the engineer, it can be used to find (and replace) all other erroneous modules. We observe that the solution returned may not be unique. This is true due to fault equivalence where more than one correction type may be available to synthesize a function and correct the design [18], [19].

The next five columns contain CPU times for the brute-force method and the proposed debugging algorithm. Columns five and six contain the time for diagnosis, column seven shows the time to do correction, and the next two columns present total times for the brute-force and the proposed method to debug and rectify a design for the set of test vectors used. From these columns, it can be seen that the proposed automated approach reduces the manual debugging effort by a factor of $\times 9.9$ and $\times 10.9$ when full and partial state equivalence is available, respectively, and by a factor of $\times 3.1$ when no state equivalence exists. The degrade in the speed up when no equivalence information is available versus the brute-force approach is caused by the pessimistic nature of path-trace. Specifically, as shown from columns two and five of Table VI, the procedure marks more modules and it takes more time as it moves deeper in the pipeline towards the primary inputs. However, with the exception of circuit J (see Table VI), where the information provided by this processing overhead does not pay off with a performance improvement, for all other circuits, the proposed approach shows a significant speed-up. On the other hand, in the case of partial state equivalence, the proposed methodology offers the maximum speed up against the brute-force one. This is because, in that case, the brute-force approach exhibits its worst performance as it enumerates all possible error scenarios and often simulates their effects through many pipeline stages.

Fig. 14 plots debugging run times for different cases of state equivalence information from Tables IV–VI for four circuits. In all cases, the CPU saving due to the state equivalence information is reflected in the final debugging effort. This is expected because state equivalence eases the task of path-trace. It reduces

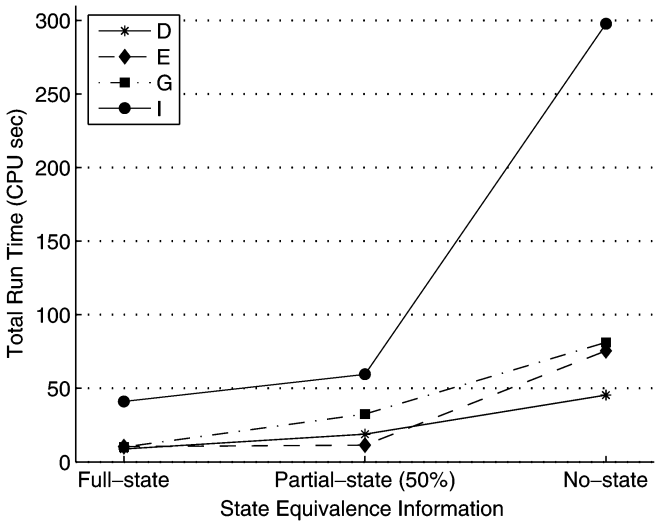


Fig. 14. Run time versus state information.

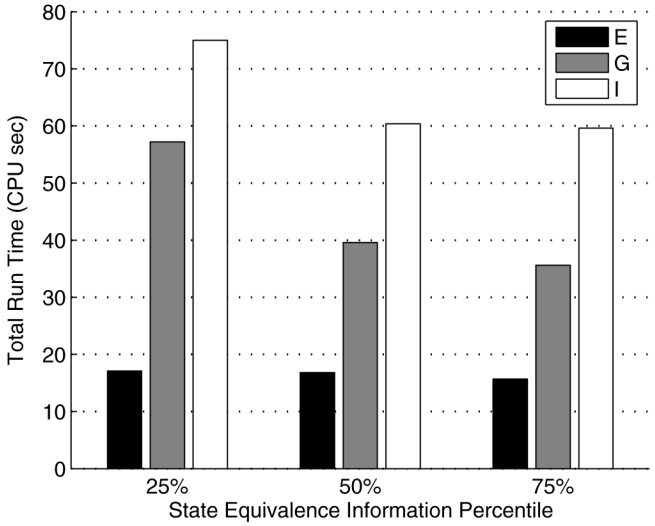


Fig. 16. Performance for partial state equivalence.

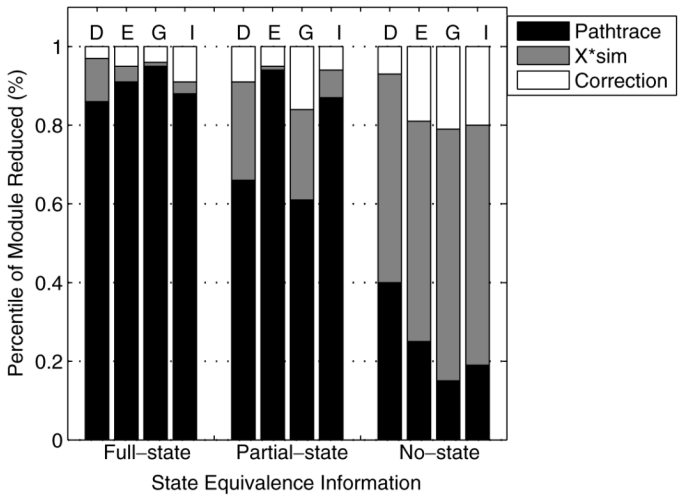


Fig. 15. Debugging effort versus state equivalence.

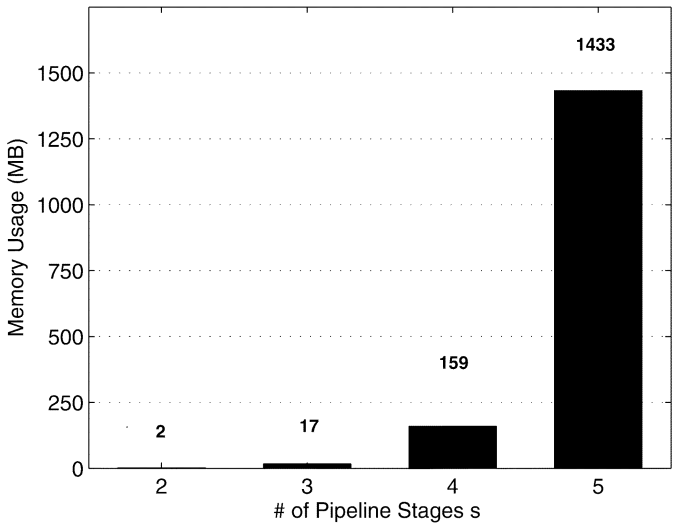


Fig. 17. Memory utilization.

the number of combinational circuitry it needs to traverse and less lines are usually marked. This is because path-trace can observe responses from memory elements with state equivalence since they act as pseudo primary outputs. Therefore, it can start marking directly from the ones that have an erroneous simulation value.

Next, in Fig. 15, we profile the different stages of the debugging algorithms in terms of the number of modules returned against the state equivalence information available. The graph confirms the trend described above, that is, the performance of path-trace diminishes as less state equivalence information becomes available. For example, in the full state equivalence case, path-trace can eliminate more than 80% of module candidates while this number drops to less than 40% when no state equivalence is available. Clearly, the pessimistic line marking ability of path-trace when less state equivalence information is available is complemented by X^*-simulation, as shown in Fig. 15, which is a more precise procedure at the expense of additional computational (simulation) effort.

The graph in Fig. 16 provides us with an intuition about the behavior of the method when 25%, 50%, and 75% of state equivalence information is available to the debugger for three benchmark circuits. The graph indicates that state equivalence information reduces the debugging effort. This reduction is sometimes drastic (such as in circuit G) and other times less important (circuits I and E). Again, experience says that the overall performance of debugging depends on the resolution of path-trace. Nevertheless, as already explained for Figs. 14 and 15, more state equivalence information eases the task of debugging, and this is also justified by the plot in Fig. 16.

Finally, Fig. 17 plots the memory utilization due to the storage tables for circuits with increasing pipeline stages. Each register file of those designs contains 34 registers and nine local clocks (i.e., $f = 34$ and $c = 9$). The core circuitry between each two register layers contains approximately 500 primitive elements. We observe that memory requirements increase by factors of 8.5, 9.3, and 9.0, respectively, as the value of s increases, which is an observation that matches the asymptotic theoretical analysis in Section IV-A. In all cases, the core circuitry contributes for less than 5% of the total memory requirements.

In the future, we plan to investigate additional types of extraction mismatches including errors that occur at the digital/analog tapping circuitry of custom designs. We will also extend the debugging methodology to handle multiple errors with multiple instantiations, and we will attempt to refine processes such as path-trace, because their resolution seems to have a strong effect on the overall debugging effort. Since the work in this paper deals with strictly pipelined circuit architectures, it is our intention to extend the debugging methodologies to operate on different non-pipelined circuitry as well as pipelines that include feedback. Finally, we plan to modify diagnosis and develop approximation heuristics that allow the approach to handle deeper pipelines with no state equivalence yet remain memory-/time-efficient.

VI. Conclusion

Logic extraction is a mandatory reverse engineering process to generate tests in custom high-performance designs. This paper investigates discrepancies during extraction and presents methodologies to improve test model generation. Different classes of extraction errors in the core and clocking circuitry of modern designs are presented, and their effects are analyzed in detail. A robust diagnosis algorithm for single extraction errors with multiple instantiations in gate-level implementations with full, partial, and no state equivalence with the transistor-level schematic is also proposed. A comprehensive suite of experiments on circuits with architecture similar to the one found in industry demonstrates its efficiency as it helps reduce the manual debugging effort by orders of magnitude. Investigating the nature of extraction errors and debugging techniques for these errors help improve test model generation and shorten test delivery time for high-performance low-power ICs.

Acknowledgment

The authors would like to acknowledge the technical contribution of J. Liu at early stages of this work. They would also like to thank the anonymous reviewers of this paper and the reviewers in earlier conference versions who helped improve its presentation and impact with their comments.

References

[1] M. S. Abadir, J. Ferguson, and T. E. Kirkland, "Logic verification via test generation," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 7, no. 1, pp. 138–148, Jan. 1988.

[2] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyamal, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama, "A 1.3-GHz fifth-generation sparc64 microprocessor," *IEEE J. Solid-State Circuits*, vol. 38, no. 11, pp. 1896–1905, Nov. 2003.

[3] D. Bearden, D. Caffo, P. Anderson, P. Rossbach, N. Iyengar, T. Petrsen, and J.-T. Yen, "A 780 MHz powerpc microprocessor with integrated l2 cache," in *Proc. IEEE ISSCC*, 2000, pp. 90–91.

[4] D. T. Blaauw, D. G. Saab, P. Banerjee, and J. A. Abraham, "Functional abstraction of logic gates for switch-level simulation," in *Proc. IEEE Eur. Conf. Design Autom.*, 1991, pp. 329–333.

[5] M. Boehner, "LOGEX—An automatic logic extractor from transistor to gate level for CMOS technology," in *Proc. Design Autom. Conf.*, 1988, pp. 517–521.

[6] R. E. Bryant, "Extraction of gate level models from transistor circuits by four-valued symbolic analysis," in *Proc. IEEE Int. Conf. Comput.-Aided Design*, 1991, pp. 350–353.

[7] D. Draper, M. Crowley, J. Holst, G. Favor, A. Schoy, J. trull, A. Ben-Meir, R. Khanna, D. Wendell, R. Krishna, J. Nolan, D. Mallick, H. Partovi, M. Roberts, M. Johnson, and T. Lee, "Circuit techniques in a 266-MHz MMX-enabled processor," *IEEE J. Solid-State Circuits*, vol. 32, no. 11, pp. 1650–1664, Nov. 1997.

[8] M. Gowan, L. Biro, and D. Jackson, "Power considerations in the design of the alpha 21264 microprocessor," in *Proc. IEEE/ACM Design Autom. Conf.*, 1998, pp. 726–731.

[9] N. Jha and S. Gupta, *Testing of Digital Systems*. Cambridge, U.K.: Cambridge Univ. Press, 2003.

[10] T. Kostelijk and B. D. Loore, "Automatic verification of library-based IC designs," *IEEE J. Solid-State Circuits*, vol. 26, no. 3, pp. 394–403, Mar. 1991.

[11] N. Kurd, J. Barkatullah, R. Dizon, and T. Fletcher, "A multigigahertz clocking scheme for the Pentium microprocessor," *IEEE J. Solid-State Circuits*, vol. 36, no. 11, pp. 1647–1653, Nov. 2001.

[12] M. Kusko, B. Robbins, T. Snethen, P. Song, T. Foote, and W. Huott, "Microprocessor test and test tool methodology for the 500 MHz IBM S/390 G5 chip," in *Proc. IEEE Int. Test Conf.*, 1998, pp. 717–726.

[13] S. Kundu, "Gatemaker: A transistor to gate level model extraction for simulation, automatic test pattern generation and verification," in *Proc. IEEE Int. Test Conf.*, 1998, pp. 372–381.

[14] J. B. Liu and A. Veneris, "Incremental diagnosis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 2, pp. 240–251, Feb. 2005.

[15] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective*. Upper Saddle River, NJ: Prentice-Hall, 1996.

[16] Needs Document in Test and Testability (Area 6) Semiconductor Research Corporation, 2003 [Online]. Available: http://www.src.org/fr/test_call_03.asp

[17] T. McDougall, A. Parashkevov, S. Jolly, J. Zhu, J. Zeng, C. Pyron, and M. S. Abadir, "An automated method for test model generation from switch level circuit," in *Proc. IEEE Asian-South Pacific Design Autom. Conf.*, 2003, pp. 769–774.

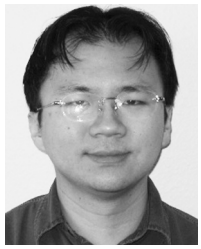
[18] A. Veneris and I. N. Hajj, "Design error diagnosis and correction via test vector simulation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 18, no. 12, pp. 1803–1816, Dec. 1999.

[19] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 10, pp. 1606–1621, Oct. 2005.

[20] S. Venkataraman and W. K. Fuchs, "A deductive technique for diagnosis of bridging faults," in *Proc. IEEE Int. Conf. Comput.-Aided Design*, 1997, pp. 562–567.

[21] Y. Yang, J. Liu, P. Thadikaran, and A. Veneris, "Extraction error diagnosis and correction in high-performance designs," in *Proc. IEEE Int. Test Conf.*, 2003, pp. 423–430.

[22] Y. Yang, A. Veneris, P. Thadikaran, nd, and S. Venkataraman, "Extraction error modeling and automated model debugging in high-performance low power custom designs," in *Proc. IEEE Design Test Europe*, 2005, pp. 996–1001.



Yu-Shen Yang (S'02) received the B.A.Sc. degree (with honors) and the M.A.Sc. degree from the University of Toronto, Toronto, ON, Canada, in 2002 and 2004, respectively, both in computer engineering. He is currently working toward the Ph.D. degree in computer engineering at the University of Toronto.

His research interests include VLSI circuit diagnosis and correction, design resynthesis, and design rewiring.

Andreas Veneris (S'96–M'99–SM'05) was born in Athens, Greece. He received the Diploma in computer engineering and informatics from the University of Patras, Patras, Greece, in 1991, the M.S. degree in computer science from the University of Southern California, Los Angeles, in 1992, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign (UIUC), Urbana, 1998.

He was a Vsiting Faculty Member with UIUC from 1998 to 1999. In 1999, he joined the University of Toronto, Toronto, ON, Canada, where he is currently an Associate Professor, cross-appointed with the Department of Electrical and Computer Engineering and the Department of Computer Science. His research interests include CAD for synthesis, diagnosis, and verification of digital circuits and systems, data structures, and combinatorics. He is the coauthor of one book.

Dr. Veneris is a member of the the Association for Computing Machinery, AAAS, the Technical Chamber of Greece, and the Planetary Society. He was corecipient of a Best Paper Award at ASP-DAC'01.

Paul Thadikaran (M'00) received the Ph.D. degree in computer science from the State University of New York (SUNY), Buffalo.

He is currently a Principal Engineer with the Enterprise Microprocessor Development Group, Intel Corporation, Hillsboro, OR. He is also an Adjunct Professor with the Oregon Health Sciences and Engineering University and SUNY Stonybrook. He has been involved with various aspects of design and test of previous three generations of Intel's IA-32 CPU. He has managed CAD tool development and standard cell library development targeted for CPU designs for the past six years. His areas of interest include CAD tools and algorithms for test generation, power estimation, functional verification, and diagnosis. He has published several papers in IEEE/Intel conferences and journals. He has also coauthored the book, *Introduction to I_{ddq} Testing* (Springer, 1997). He has served as a reviewer for several IEEE and ACM journals, such as the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS and *ACM Transactions on Design Automation in Electronic Systems*.

Srikanth Venkataraman (S'93–M'97) received the Ph.D. degree in electrical engineering from the University of Illinois at Urbana-Champaign, Urbana.

He is a Manager and Principal Engineer with Intel Corporation, Hillsboro, OR. He is involved in the areas of DFT, test tools and methodologies, diagnosis, and debug technologies in the Design and Technology Solutions Group. His research interests include VLSI test, CAD and software engineering. He has authored or coauthored over 50 publications, holds one patent, and has three patents pending, and he has presented tutorials at several conferences.

Dr. Venkataraman was the recipient of the Best Paper Award at IEEE Vehicular Technology Symposium (VTS) 2000, a Top 10 Papers at ITC 2000, and Best Panel at IEEE VTS 1999. Intel awards include the Intel Achievement Award (2005), the Divisional Recognition Awards (2000, 2002, and 2004), the Technical Recognition Award (2002), the Excellence Award (2001), and Best Paper Awards at the Design and Test Conference (2002 and 2003).