

Extraction Error Diagnosis and Correction in High-Performance Designs

Yu-Shen Yang¹

J. Brandon Liu¹

Paul Thadikaran³

Andreas Veneris^{1,2}

Abstract

Test model generation is crucial in the test generation process of a high-performance design targeted for large volume production. A key process in test model generation requires the extraction of a gate-level (logic) model from the transistor level representation of the circuit under test. Logic extraction is an error prone process due to extraction tool limitations and due to the human interference. Errors introduced by extraction require manual debugging, a resource intensive and time consuming task. This paper presents a set of extraction errors typical in an industrial environment. It also proposes an automated solution to extraction error diagnosis and correction. Experiments on circuits with similar architecture to that of high speed custom-made industrial blocks are conducted to confirm the fitness of the approach.

1 Introduction

Large and complex VLSI designs such as microprocessors, SOCs and ASICs often require high-performance custom-made logic blocks designed at the transistor level. This type of circuit representation cannot be used directly to generate production tests. For this reason, a gate-level (logic) representation of the custom-made blocks is extracted and used instead.

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 in Figure 1 [6]. In this flow, the first step is *test model generation* that returns a gate-level model that ac-

curately represents the logical behavior of the Circuit Under Test (CUT). The behavior of the CUT is commonly determined by the functionality of the custom-made transistor level component, the functionality of additional synthesized logic and various test constraints.

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 transistor level design. This process enables generation of gate-level views of transistor level designs. Logic extraction is a completely automated process for combinational logic and most sequential circuits except certain complex cells such as clock generators and scan sequentials. These exceptions are handled by manual generation of gate-level models. The level of abstraction in the libraries used during extraction, bugs in CAD tools and the human factor may introduce errors in the extracted gate-level model.

For this reason, the logical netlist is usually verified prior to test generation (Figure 1). Verification is carried out with formal equivalence-based methods for combinational circuitry and for small pieces of sequential logic. Simulation is also used for large complex sequential circuits. Both verification approaches may not be complete and/or exact due to several reasons. For example, formal methods prove equivalence in the boolean domain of well defined logic values $\{0, 1\}$ but they do not prove equivalence in the $\{0, 1, X\}$ space where the design is exercised during test generation. Additionally, formal equivalence- and simulation-based approaches can require excessive runtimes due to 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 the integration stage that assembles the complete netlist for the CUT.

The next step of test generation is *model verification* where the assembled gate-level netlist is validated against a golden model such as a detailed switch-level model of the CUT. This verification step usually generates a few sample scan tests with a broad coverage and it simulates these

¹University of Toronto, Department of Electrical and Computer Engineering, Toronto, ON M5S 3G4 ({yangy, liuji, veneris}@eecg.toronto.edu)

²University of Toronto, Department of Computer Science, Toronto, ON M5S 3G4

³Intel Corporation, Architecture Group, Hillsboro, OR 97124 (paul.thadikaran@intel.com)

tests against a golden model. On successful completion of model verification, the model is used for test generation. Following test generation, the generated tests are again verified against the golden model to ensure test correctness, a process known as *test/vector validation*. Tests may fail in this step due to extraction errors introduced in model generation but not identified during model verification. In this case, manual analysis of failures is required, a time-consuming and resource intensive step that may delay test delivery to the factories.

In this paper, we describe a set of extraction errors that are typical in an industrial setting and cause scan test failures. Such errors occur because of erroneous module extraction. Another source of these errors are module specification (constraint) mismatches between different libraries (physical design library, logic library, simulation library, ATPG library etc) when complex hardware such as memory elements, multiple clock domains and tri-state devices are integrated together.

We also propose an automated approach to debug these errors. The approach identifies and corrects single extraction errors in two stages: *model-free diagnosis* and *model enumeration*. Model-free diagnosis uses the logic unknown(s) to identify error locations. This method suits many characteristics of the problem. Automated debugging approaches such as the one described here benefits the testing of microprocessors and other complex components. They aid in identifying and correcting contributors of model inaccuracies at early stages of the design cycle and help reduce the turnaround time of test delivery.

The paper is organized as follows. Section 2 describes the nature and different types of errors typical to the test model extraction process. Section 3 presents the proposed extraction error diagnosis and correction algorithm. Simulation results and future work are presented in Section 4. Section 5 concludes the paper.

2 Extraction Error Types

The extraction process builds a logic representation from a custom-made logic block. This gate-level representation is a hierarchical structural netlist of combinational logic primitives and various memory elements. This representation is later used for scan test generation.

The extraction process may introduce errors in the final flattened netlist due to *erroneous module mappings*. Since a single module may have multiple netlist instantiations, a single error contained in a module definition may map to multiple errors in the netlist. An additional source of

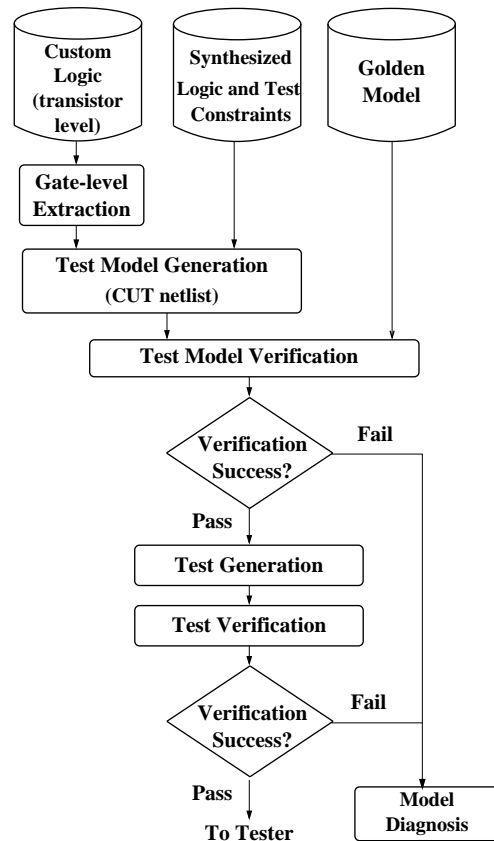


Figure 1. Test Generation Flow

extraction errors may be *functional mismatches* in the definition of module operation constraints between different libraries (simulation library, synthesis library, physical design library, ATPG library etc). In this case, the mapping is correct but the operation of the module is interpreted different by various libraries. This mismatch(es) may produce functional errors in the final extracted netlist.

Notice that both the erroneous module mapping(s) and the functional library mismatches may result in multiple error sites in the final extracted netlist. This is shown in Figure 2 where a single error for Module I “translates” to three errors in the final netlist. Therefore, the number of error sites can become large even though only a single error is introduced during extraction. Due to the presence of *multiple error effects*, diagnosis becomes a challenging task as the solution space becomes prohibitively large [9].

This work considers single extraction errors for sequential circuits with full-scan memory elements and different clock domains. It also assumes that all input test vectors are full-scan single clock cycle pat-

terns. These circuits contain primitives (N)AND, (N)OR, NOT, tri-state buffer and D flip flops, each with an intrinsic delay of 1 time unit. Circuits used in this work are *two-stage strictly pipelined* to resemble the architecture of custom-made high-performance components found in industry. This architecture is shown in Figure 8 (Section 4) where combinational logic A, B and C is completely separated by layers of (full-scan) memory elements I and II. In the future, we plan to develop automated multiple error diagnosis and correction tools for partial-scan and non-scan components.

The remaining section outlines various types of extraction errors that are typical to the process.

2.1 Reset Synchronicity

A large portion of the silicon area in contemporary designs is dedicated to storage [8]. Memory elements such as flip-flops commonly have either a synchronous or an asynchronous reset. As illustrated in Figure 3(a), 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, an asynchronous reset can be mapped to a synchronous one and vice versa. That is, both directions of the error in Figure 3(a) may occur.

2.2 Reset-Clock Contest

Reset-Clock contest happens due to module specification mismatch in the design libraries used during test model generation. Different libraries may make different assumptions on the winner of the contest, as shown in Figure 3(b). The right part of that figure depicts the situation

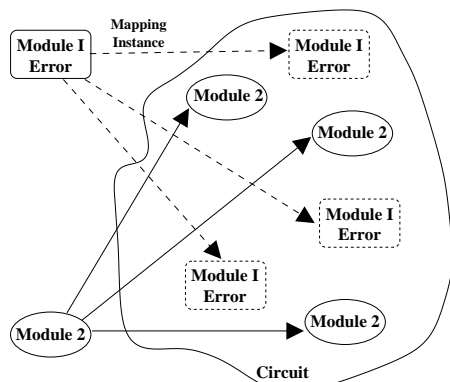


Figure 2. Multiple Instance of Single Error

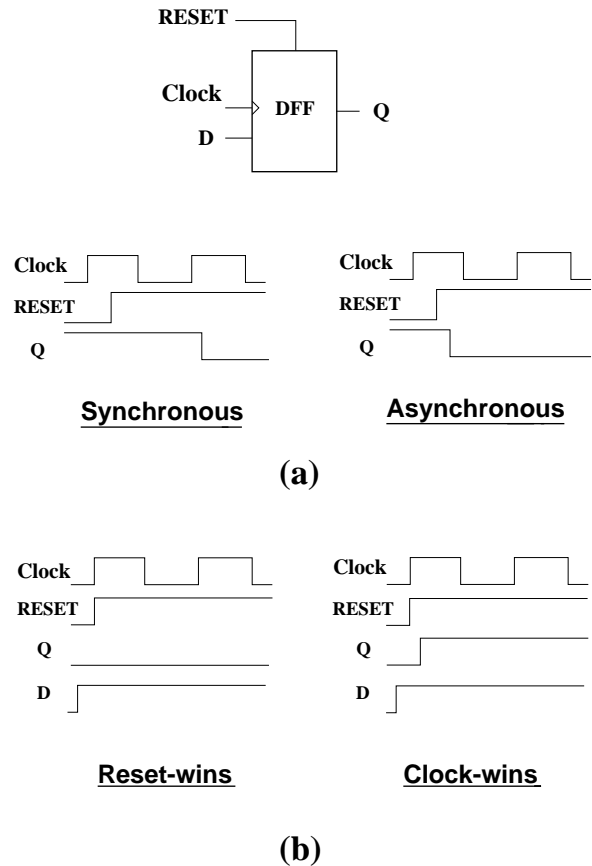


Figure 3. (a) Reset Synchronicity Error (b) Reset-Clock Contest Error

where Q is initially 0 and Clock prevails. In the left part of the figure, the RESET line wins the contest for this initial state configuration. This type of error can arise from contest of other lines (set, hold, etc) as well. The error may occur in both directions as in Reset Synchronicity.

2.3 Multiplexer Implementation

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 Figure 4 (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. In the first one, the inverter is replaced by a buffer (Figure 4 (b)). When SEL goes high, both tri-state buffers are enabled, causing a *merger* at the output OUT. Table 1 sum-

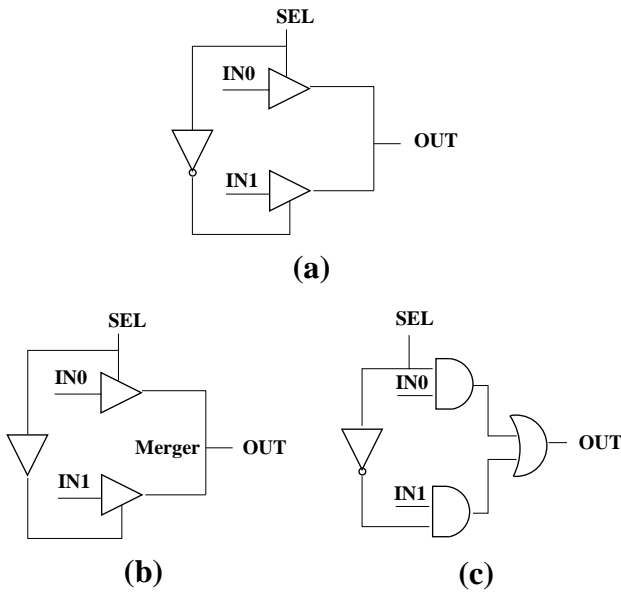


Figure 4. MUX Error

marizes the truth table for merger. Entries which are not bold 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 second case, instead of tri-state buffers, the multiplexer is erroneously mapped to an implementation with logic AND and OR gates (Figure 4 (c)). 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 bold entry in Table 2, the implementation with AND and OR gates drives 0 on OUT, while the tri-state buffer implementation drives X.

IN0	IN1	Merger	IN0	IN1	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 1. Merger Truth Table

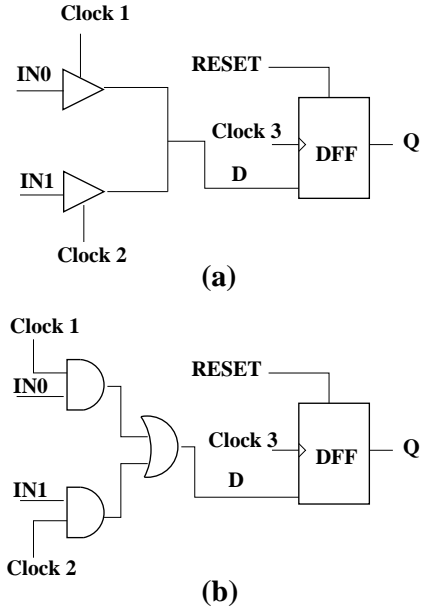


Figure 5. MUX-Latch Implementations

2.4 Multiplexer-Latch Implementation

Multiplexer-latch (MUX-latch) design allows the sampling of signals from two non-overlapping clock domains. In Figure 5, 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. As shown in Figure 5, the multiplexer in a MUX-latch design can be implemented with tri-state devices or AND/OR gates. These two implementations have different logic functions when CLOCK1 and CLOCK2 overlap. In the AND/OR implementation, the flip-flop latches IN0 OR IN1. In contrast, X is always latched in the tri-state buffer implementation.

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

Table 2. MUX Error when SEL=X

3 Debugging Extraction Errors

When an extracted netlist fails the scan test, current practice requires a manual analysis of the failure, which is an expensive process in terms of both time and resources. This section describes a simulation-based diagnosis and correction tool that rectifies single extraction errors in full-scan hierarchical digital circuit designs.

The *golden model* is a correct specification for the custom block. In our implementation, it is represented by a logic netlist we can simulate, however its internal structure is unavailable to diagnosis. The *erroneous netlist* is the circuit obtained by extraction when applied to the custom block. It is represented by a hierarchical gate-level logic implementation of the specification with a single extraction error on a module randomly injected into it. Recall from Section 2, a single error may be translated into multiple erroneous instances in the final netlist.

The overall debugging flow is shown in Figure 6. The golden model, the erroneous netlist and a set of test vectors with failing responses for the netlist are input to the algorithm. These vectors are collected via simulation of the random input full-scan test patterns and of a pre-computed input test patterns with high test coverage [4]. Test vector generation for extraction errors and design verification following debugging are beyond the scope of this work.

The objective is to identify and correct all instances of the injected error. This is done in two phases: *model-free diagnosis* and *model enumeration (correction)*. The proposed method for debugging extraction errors is similar to the model-free fault diagnosis approach presented in [7]. In the following paragraphs, we summarize the steps of the approach but we omit proofs of claims that can be found in [7].

Model-free diagnosis simulates a logic unknown value (represented by X) on a candidate fault line to capture all possible paths for error propagation [3]. Recall, X propagates through a gate if all other lines input of the gate have non-controlling logic values [1]. If simulating X on a line propagates to an Erroneous Primary Output or Register (EPOR) for a set of vectors, the EPOR may be sensitized to a potential error on the line for these vectors.

This observation is the basis of model-free diagnosis. When compared to model-based diagnosis approaches such as [2, 5, 9, 10], it has the advantage that it performs a single simulation step (by forcing logic unknown X on the line) for each candidate error line. On the other hand, in model-based diagnosis, one logic simulation step has to be performed for each error model and for each line. This

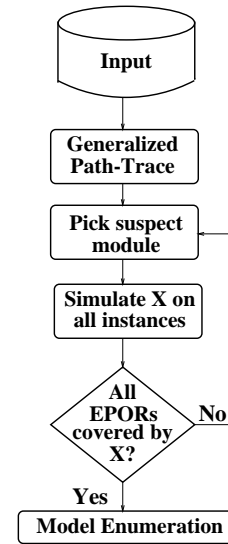


Figure 6. Model-free Diagnosis

may considerably increase the total computational cost of the procedure. Additionally, model-free diagnosis is a suitable method for this problem since many extraction error types involve logic unknown X values.

Model-free diagnosis (Figure 6) first performs a generalized path-trace routine [7]. This is similar to the linear-time effect-cause direction line-marking routine in [10] but appropriately modified to handle the unknown value X . Although many erroneous module instances may exist, generalized path-trace guarantees to mark at least one of these instances. Further, if the number of error instances is N and the number of failing input test vectors is V , one instance has to be marked at least $\frac{V}{N}$ times to qualify diagnosis. Path-trace is performed for multiple vectors and all instances that satisfy the above property are collected. Subsequently, the module definitions of these instances are extracted from the netlist hierarchy and they are re-compiled into a list of *suspect modules*.

In the next step, the suspect modules are simulated one at a time with a logic unknown X value forced at the output for *all* instances of the suspect module. This is possible because module mapping information is available to the test engineer. If X is propagated to all the EPORs, the module is returned by the model-free diagnosis as a potential location for extraction error. Detailed description and efficient implementations for this step are found in [7]. As shown by the experiments, the list of suspect modules is reduced by a factor of 2 on average.

In the *second stage* of debugging, appropriate extrac-

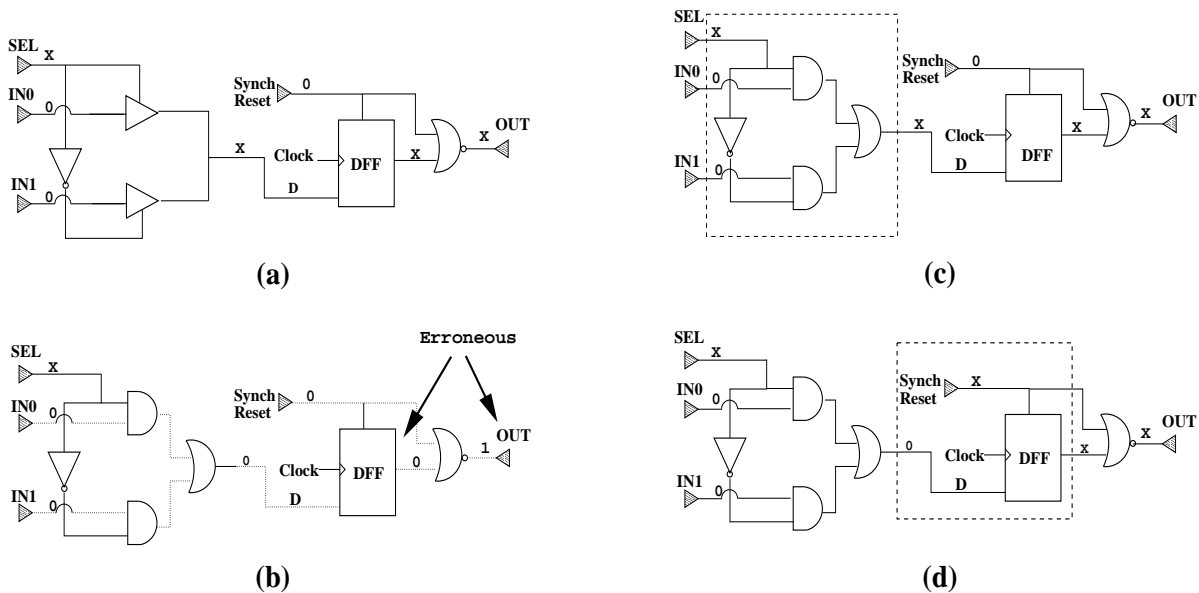


Figure 7. Example circuit

tion correction (error) types from Section 2 are injected exhaustively to each suspect module returned by the first phase. These corrections are injected to all the instances of the module in the flattened netlist and they are simulated for all the erroneous vectors. All modules for which the simulation results agree with that of the golden model are returned as a solution to the problem.

Experiments show that the list of suspect modules is usually reduced to one or two modules. The solution list may contain more than one candidate solution due to error equivalence [9]. The example that follows illustrates the approach.

Example. An example of the presented algorithm is given in Figure 7. The golden model is illustrated in Figure 7(a) and an erroneous version in Figure 7(b). In the erroneous model, the 2-input MUX implemented with tri-state buffers is replaced by one with AND/OR gates. Simulated values for a single vector are shown above the circuit lines. After one clock cycle, the golden model has an X value on its DFF and OUT but the erroneous extracted model has logic 0 on DFF and logic 1 on OUT.

The lines marked by path-trace are dotted in Figure 7(b). Because the NOR gate driving OUT has all non-controlling values on its input, both RESET of the DFF and the MUX are marked by path-trace to give a list of suspect modules. During model-free diagnosis, X is forced and simulated at the output of the suspect modules. Both modules propagate X to the erroneous primary output and

register, as shown in Figure 7(c) and (d) and both suspects are passed to the model enumeration step of the algorithm. In these figures, a dashed box is used to indicate the module tested.

The solution is found when the MUX in Figure 7(c) is replaced by a tri-state buffer implementation. A valid error model cannot be found for Figure 7(d) because the only applicable correction model is reset synchronicity which does not correct the circuit.

4 Experiments

Tests were carried on sequential designs built with ISCAS'85 and ITC'99 combinational circuitry. As illustrated in Figure 8, the designs are two-stage strictly pipelined. This architecture resembles some high-speed high-performance custom blocks seen in industry today.

Benchmarks are prepared by injecting two layers of D flip-flops (Register Layer I and II) each of which is randomly assigned one of four possible different clock domains. Combinational logic circuitry A, B and C are insulated from each other by the two register files. The frequency of the clock domains is an integral multiple of each other. An error-free scan-chain is also inserted so that all flip-flops are observable and controllable [1].

Characteristics of the benchmark circuits after flattening are presented in Table 3. The first column identifies the circuit name and the second column has the total number of logic primitives in the three combinational blocks. The

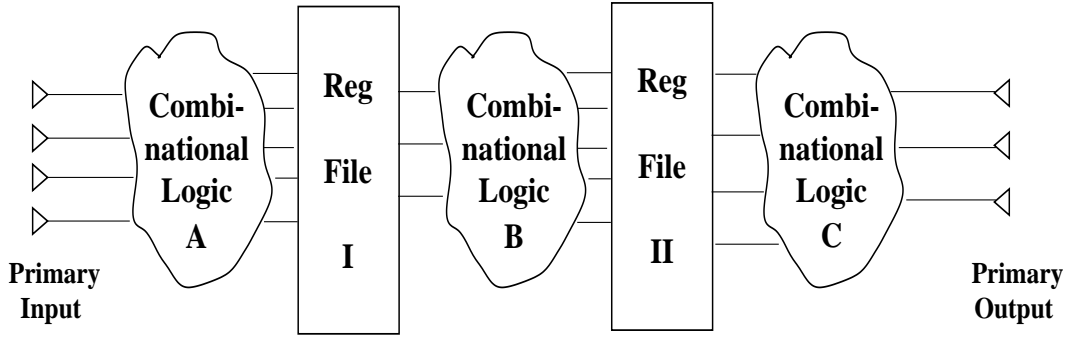


Figure 8. Benchmark Architecture

next three columns contain the number of registers in Register File I, II and total, respectively. The last column of the table has the maximum depth of logic for each combinational block. Tristate devices and buffers are introduced within combinational logic and latches randomly. We used a set of 5 libraries with different names for each error module case from Section 2. An extraction error can be mapped to 5-50 instances, if it occurs, in the final gate-level implementation.

Ten experiments are performed for every circuit. The original structural netlist is used as the specification. To prepare the erroneous netlist, an extraction error is randomly injected into the specification netlist. 3200 random full-scan vectors and vectors with high coverage [4] are generated and simulated with an event-driven parallel vector simulator. The average number of vectors that detect the error is shown in column 2 of Table 4. Errors that involve memory elements are easy to detect because we use a scan chain and this makes them highly observable.

circuit	gate count	register count			max logic depth
		RF I	RF II	total	
A	416	45	54	99	17
B	653	83	64	147	16
C	1251	90	180	270	27
D	1984	151	59	210	28
E	2979	475	92	567	28
F	4379	639	132	671	25
G	5226	974	614	1588	25
H	6684	1404	610	2014	30
I	7653	1433	828	2261	29
J	7716	1505	532	2037	27

Table 3. Circuit Characteristics

On the other hand, errors such as MUX-Latch implementation involve many conditions to excite the error and they can be harder to detect. Experiments show that 72% of random vectors (on the average) detect the errors at primary output whereas 88% of vector with high fault coverage detect them. This suggests that, in most cases, random vectors provide sufficient resolution to diagnosis.

Columns 3 to 5 of Table 4 show the average number of suspect modules returned by each step of the proposed algorithm. In detail, the third column has the number of suspect modules after generalized path-trace. In most cases, the number of candidates is reduced by one third or more. The column that follows contains the number of suspect modules after model-free simulation. Notice that model-free simulation disregards most invalid solutions.

The fifth column summarizes the number of suspect modules after model enumeration. This is also the set of suspicious modules returned by the rectification algorithm. Note that, at this stage, not only an erroneous module is found but an extraction correction is also proposed. In all experiments, the original error is discovered. In the few cases where this column is greater than one, additional equivalent extraction errors are found due to error equivalence. These results demonstrate the accuracy and efficiency of the proposed algorithm. The number of suspect modules is gradually reduced through the different stages until the original error and its equivalent are returned in all ten experiments for each circuit.

The sixth column of that table has the average amount of CPU time required for all steps. By construction, most of the time of the algorithm is spent performing fault simulation. A compiled simulator implementation will improve the run-times of the approach. The last column shows the CPU time spent by a brute-force approach where all possible error types are enumerated. We observe, a significant speed up (x3 on the average) is achieved due to the ap-

circuit	avg fail vectors	# of suspect modules			CPU time (sec)	
		Path-trace	X sim	Model enum.	proposed	brute-force
A	1979.9	15.7	8.4	1.1	5.3	17.6
B	2123.9	12.2	5.7	1.1	3.87	4.58
C	2292.4	14.1	6.2	1	21.32	63.72
D	2116.6	12.9	6.7	1	17.16	72.12
E	1624.0	14.8	5.4	1.2	30.22	108.21
F	2536.2	16.1	4.6	1.1	38.7	142.1
G	2578.3	10.7	4.8	1	63.1	267.8
H	2598.4	12.8	5.1	1	124.7	436.73
I	2657.8	13.5	5.0	1.1	132.6	448.7
J	2447.1	13.5	4.3	1.3	119.6	356.1

Table 4. Diagnosis and Correction Results

proach.

In the future, we plan to investigate additional types of extraction mismatches and extend the rectification algorithm to handle them efficiently. We also plan to apply the approach to cases where the test model is corrupted by many errors. Finally, we intend to enhance these solutions for partial-scan and non-scan sequential circuits. These are challenging problems because the error space grows exponentially with increasing number of errors and error effects [9].

5 Conclusion

This paper investigates the problem of extraction error diagnosis and correction in high-performance custom-made designs. Such errors occur because of erroneous module mapping(s) and/or module specification mismatch between libraries. Different classes of extraction errors common to the industry are presented. A diagnosis algorithm for single extraction errors is also proposed. Experiments demonstrate the efficiency of the approach that helps improve and shorten the test delivery turnaround time for high-performance ICs.

References

[1] M. Abramovici, M. Breuer, and A. Friedman. *Digital Systems Testing and Testable Design*. IEEE Press, Piscataway, New Jersey, revised edition, 1994.

[2] R. C. Aitken. Modeling the unmodelable: Algorithmic fault diagnosis. *IEEE Design and Test of Computers*, pages 98–103, July-September 1997.

[3] V. Boppana and M. Fujita. Modeling the unknown!towards model-independent fault and error diagnosis. *Proc. IEEE ITC*, pages 1094–1101, 1998.

[4] I. Hamzaoglu and J. Patel. New techniques for deterministic test pattern generation. *Proc. IEEE VTS*, pages 138–148, 1998.

[5] S. Huang. Towards the logic defect diagnosis for partial-scan designs. *Proc. IEEE ASP-DAC*, pages 313–318, 2001.

[6] M. Kusko, B. Robbins, T. Snethen, P. Song, T. Foote, and W. Huott. Microprocessor test and test tool methodology for the 500mhz ibm s/390 g5 chip. *Proc. IEEE ITC*, 1998.

[7] J. Liu, A. Veneris, and H. Takahashi. Incremental diagnosis of multiple open-interconnects. *Proc. IEEE ITC*, pages 1085–1092, 2002.

[8] J. M. Rabaey. *Digital Integrated Circuits: a design perspective*. Prentice Hall, 1996.

[9] A. Veneris, J. Liu, M. Amiri, and M. Abadir. Incremental diagnosis and debugging of multiple faults and errors. *Proc. IEEE DATE*, pages 716–721, 2002.

[10] S. Venkataraman and W. Fuchs. A deductive technique for diagnosis of bridging faults. *Proc. IEEE ICCAD*, pages 562–567, 1997.