# Automated Data Analysis Techniques for a Modern Silicon Debug Environment

Yu-Shen Yang[1], Andreas Veneris[2], Nicola Nicolici[3], Masahiro Fujita[4]

***Abstract*—With the growing size of modern designs and more strict time-to-market constraints, design errors unavoidably escape pre-silicon verification and reside in silicon prototypes. As a result, silicon debug has become a necessary step in the digital integrated circuit design flow. Although embedded hardware blocks, such as scan chains and trace buffers, provide a means to acquire data of internal signals in real time for debugging, there is a relative shortage in methodologies to efficiently analyze this vast data to identify root-causes. This paper presents an automated software solution that attempts to fill-in the gap. The presented techniques automate the configuration process for trace-buffer based hardware in order to acquire helpful information for debugging the failure, and detect suspects of the failure in both the spatial and temporal domain.**

## I. Introduction

In the pre-silicon stages of the integrated circuit development cycle, engineers verify designs with sophisticated simulation [1], emulation [2] and formal verification [3], [4] tools to check the correctness of the RTL model against its functional specification. However, due to the growing complexity of functionality and the size of designs, it becomes infeasible to achieve 100% verification coverage within the strict time-to-market constraints. Inevitably, functional bugs may not be captured by any pre-silicon verification techniques and only be discovered during in-system silicon validation where the design is exercised at speed. Consequently, silicon prototypes are rarely bug-free and multiple re-spins are often necessary [5]. Each re-spin dramatically increases project costs and the time-to-market. Therefore, it is important to develop a silicon debug flow that provides short turn-around time when a silicon prototype fails.

A typical silicon debug process consists of several iterative sessions, referred to as *debug sessions*. Each session can be divided into two stages: *data acquisition* and *data analysis*. In the data acquisition stage, test engineers set up the environment to obtain appropriate data from the chip under test while it is operated in real-time. Unlike pre-silicon verification, where values of any signals can be obtained through simulation, observability of internal signals in the silicon prototype is restricted. Several *Design-for-Debug (DfD)* hardware components, such as scan chains or trace buffers, are used to access the internal signals. Nevertheless, the amount of acquired data is limited by the integrated DfD hardware components. These limits greatly inhibit accurate and effective debugging analysis. During data analysis, the vast amount of data acquired during the test is analyzed to prune the error candidates and to set up the data acquisition environment for the next debug session. Despite the use of dedicated data-collection hardware mechanisms embedded directly into the silicon, there exists little in automated software solutions to help the validation engineer identify the root cause of the failure with the data acquired. This time-consuming and labor-intensive cycle continues until the root cause of the failure is determined.

This paper presents an automated software-based debug methodology that complements current data acquisition hardware solutions. The methodology is designed as a post-processing step after the data has been acquired. It identifies the potential locations of the error in a hierarchical manner, and estimates the time interval where the error is excited. The result allows the engineer to concentrate the manual investigation on a smaller set of locations within a more concise windows of cycles. In addition, this methodology sets up the data acquisition environment for the next debug session and uses the new data in the subsequent automated data analysis cycle to eventually determine the root cause.

Case studies on OpenCores.org circuits are presented. Results show that the methodology successfully determines the locations of the error and it also specifies with accuracy the time interval in which the error is excited.

The remainder of the paper is organized as follows. Section II summarizes work on hardware and software solutions for silicon debug, as well as the background material. Section III presents the software solution to silicon debug. Finally, the case studies and conclusion are given in Sections IV and V, respectively.

## II. Background

In this section, two data acquisition hardware components used to enhance the observability of internal signals in chips are discussed. Next, several automated data analysis algorithms are reviewed. Finally, we summarize background material for the presented methodology.

### A. Design for Debug Hardware Solutions

The behavior of internal signals in the chip can only be observed if they are routed to external pins. Since numbers of available pins on the chip are limited, this approach may

[1]Vennsa Technologies. Toronto, ON. Canada (terry@vennsa.com)

[2]Department of Electrical and Computer Engineering and with the Department of Computer Science, University of Toronto, Toronto, ON. Canada (veneris@eecg.utoronto.ca)

[3]Department of Electrical and Computer Engineering, McMaster University, Hamilton, ON, Canada (nicola@ece.mcmaster.ca)

[4]VLSI Design and Education Center, University of Tokyo, Japan (fujita@ee.t.u-tokyo.ac.jp)

not provide sufficient information to perform debugging. To improve observability of internal signals, two ad-hoc DfD solutions are mainly used in practice: scan chains and trace buffers.

*a) Scan chains:* provide a means to take a snapshot of the registers at a specific cycle. This operation is referred to as *scan dump*. However, the scan dump operation interrupts the execution of the chip because the values stored in the registers are destroyed. In order to resume the execution from the same point, the environment must be reset and restarted from the beginning of the test vector [6].

*b) Trace buffers:* record internal signals in an on-chip memory, which typically ranges from 8Kb to 256Kb, in real-time. Trigger logic of a trace buffer monitors circuit behavior and records the logic values of selected signals when the trigger condition is asserted. Subsequently, the recorded data is read via a low-bandwidth interface, such as a boundary scan. Trace buffers can collect values of signals for consecutive cycles, but only a small set of pre-selected signals can be traced due to the limited size of the embedded memory. Those pre-selected signals are divided into groups and connected to the on-chip memory through a multiplexer. During execution, only one group can be traced at a time. The traceable signals are typically manually selected by the designer. Recently, several algorithms have been developed to automate the selection process [7], [8], [9]. Those works try to determine a small set of signals such that their values have a higher chance of restoring a significant amount of untraceable states.

### B. Work on Automated Data Analysis

Although DfD hardware enhancement increases the observability of internal signals, there is a lack of techniques that automate the data analysis process on the acquired data. Recently, there has been an effort to develop methodologies to aid the engineer in this part of the silicon debug process as summarized in the following.

The method proposed in [10] relies on scan dumps collected at multiple consecutive cycles to determine failing registers at each time frame. Next, it conducts back-tracing from those failing registers to identify the fault propagation paths and suspect registers at each cycle. Finally, it performs a forward-tracing from the suspect registers to further narrow down the root cause candidates.

Yen et al. [11] propose an approach that first isolates the critical cycles using a binary search paradigm based on the comparison between the observed data and the simulation results. A *critical cycle* is the first cycle in which the state elements show a discrepancy between the expected responses and the actual ones. Next, this method identifies suspect registers with a simple path-tracing method [12] from the unmatched registers. Finally, it simulates the golden model with faulty values injected at each suspect candidates. The candidate is included in the final suspect list if the response matches the behavior of the faulty chip.

*Backspace*, a formal approach that restores state values of a design in a failing trace is proposed in [13]. It starts from the crash state and computes backward in time. Signatures, computed with additional hardware structures, are captured during the chip execution and stored in the trace buffer. Later, those signatures are used to determine a unique or a small set of possible predecessor states that lead to the crash state. Because this technique only analyzes one timeframe each time, it can be still memory efficient when dealing with long test traces. This is a great advantage since a couple seconds of silicon execution can translate to thousands of simulation cycles.

### C. Boolean Satisfiability and UNSAT Cores

The backbone of the presented methodology is based on SAT-based diagnosis [14]. It models the debugging problem into a Boolean Satisfiability (SAT) instance and utilizes the use of UNSAT cores extracted from it to guide the silicon debug data acquisition setup. A brief overview of these methods are given in this section.

SAT proves or disproves whether a Boolean formula $\Phi$ has a satisfiable assignment, i.e., the formula is evaluated to `true`. If such an assignment exists, the formula $\Phi$ is said to be satisfiable; otherwise, it is unsatisfiable. For most modern SAT solvers, Boolean formula is presented in *Conjunctive Normal Form* (CNF), which consists of a conjunction of *clauses* where each clause is a disjunction of literals. A *literal* is an instance of the variable or its negation.

If the formula is `UNSAT`, any subset of clauses in the instance that is also unsatisfiable is referred to as an *UNSAT core*. Modern SAT solvers [15], [16], [17] can produce UNSAT cores as a result of proving unsatisfiability. An example of an unsatisfied CNF formula and its UNSAT core is shown as follows.

$$\Phi = (a+b)\cdot(a+c)\cdot(\overline{b}+\overline{c})\cdot(\overline{a})\cdot(\overline{c})$$

$$\text{UNSAT core} = \{(a+c),(\overline{a}),(\overline{c})\}$$

An unsatisfiable SAT instance can have multiple UNSAT cores. Each core represents a situation where the instance is unsatisfied. Additional UNSAT cores can be obtained by eliminating a previously found UNSAT core, as described in [18].

Modelling the problem of logic and fault diagnosis in SAT is first presented in [14]. Given a circuit and a set of test traces that cause the design to fail, the problem is formulated in a CNF instance such that the SAT solver returns solutions that correspond to error location(s). In summary, this is achieved by inserting a multiplexer at every gate (and primary input) such that when the select line ($s$) of the multiplexer is inactive, the original design is maintained; otherwise, a new unconstrained primary input variable ($w$) drives the output of the multiplexer.

The SAT-based diagnosis algorithm from [14] performs *model-free* diagnosis [19]. That is, it does not make any assumption on the behavior of the fault/error. This is a desirable fit to silicon debug since silicon prototypes can fail test for various reasons. The engineer can utilize the values of the unconstrained variables $w$ to determine the type of the fault that has occurred. Readers can refer to [14] for more information on this SAT-based debugging methodology.

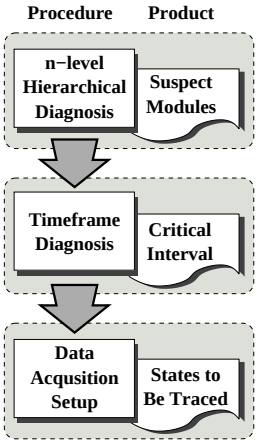
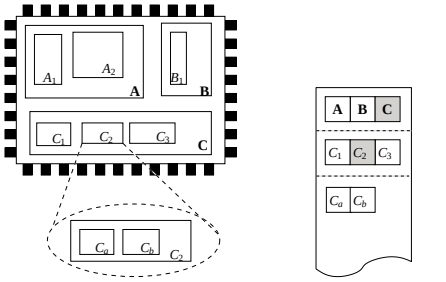



Fig. 1. A single debug analysis session




(a) Hierarchical design

(b) History of hierarchical diagnosis

Fig. 2. Hierarchical diagnosis

## III. Automated Data Analysis

A silicon debug process is different from an RTL debug process in many important aspects including the ones below. First, silicon debug needs to utilize the DfD hardware components in the design to acquire values of internal signals, whereas, in RTL diagnosis, values of internal signals can be obtained through simulation. Second, due to the vast complexity of the silicon debug problem, a software solution should be designed appropriately to take advantage of the debug hardware available to the engineer to reduce the iterations of the process. Finally, because silicon prototypes are operated at-speed during test, the test trace for debugging can be orders of magnitude longer compared to the one usually available during RTL diagnosis. As such, it is important to identify the segment of the trace that really matters to aid the future iterations of the debug process and simplify the analysis.

To accomplish the aforementioned objectives, a silicon debug flow is summarized in Figure 1. It consists of three steps. The first step is *n*-level hierarchy diagnosis, which identifies the suspect modules that contain the error in a hierarchical manner. The second step is timeframe diagnosis. This step finds the critical interval of the error. A *critical interval* is a window of cycles in which the critical cycle locates. The last step of the flow is data acquisition setup, which finds the registers that may provide useful information about the error. The above information feeds back to the proposed analysis flow which iterates the three steps in Figure 1 in the next debug session to aid in further root cause analysis. Each step is further discussed in the next sections.

### A. n-level Hierarchy Diagnosis

Hierarchical diagnosis is first proposed by Ali et. al. [20]. It is an extension of the SAT-based diagnosis to improve the performance and resolution of logic debugging. A hierarchical diagnosis process consists of several iterations. In each iteration, only modules in the same hierarchical level are considered. The procedure starts from the top-level of the design and goes deeper into the design hierarchy. Suspect candidates for debugging in each iteration are sub-modules of the modules that are determined to be suspects in the previous iteration. The presented flow would repeat the procedure for at most $n$ hierarchy levels from the level ended in the last session during each debug session. This is referred to as *n-level hierarchical diagnosis*.

Figure 2 illustrates the concept of debugging using hierarchical information. Figure 2(a) shows the hierarchical structure of a design. A situation in which hierarchical diagnosis is applied to this design with two iterations is shown in Figure 2(b). Diagnosis starts with three top modules. In the first iteration, module $C$ (grey box) is diagnosed to be the suspect. Hence, diagnosis, in the second round, only considers the sub-modules of $C$, namely, $C_1$, $C_2$, and $C_3$, as candidates. At that round, $C_2$ is identified as the suspect. As a result, the suspect candidate list for the third round consists of $C_a$ and $C_b$ only.

With the hierarchical information of the design, diagnosis can start with a coarse-grain global analysis and the search can be directed to local areas after each iteration. Such a procedure reduces the runtime and memory requirement, since there are fewer candidates that need to be analyzed.

### B. Timeframe Diagnosis

Timeframe diagnosis is carried out to find a greater precision estimate for the window of clock cycles in which the error may be excited. In silicon debug, the depth of the trace buffer limits the number of samples that are acquired in one debug experiment. Once the buffer is full, the older data is overwritten by the new samples. Hence, if the cycle in which the error is exercised can be estimated, the buffer can be utilized more effectively. Timeframe diagnosis divides the trace into $k$ intervals. The suspects returned by the previous hierarchy diagnosis step in each cycle of the interval are collectively considered as a single suspect by timeframe diagnosis. Such a conceptual suspect module is referred as *timeframe module*. Consequently, timeframe diagnosis selects suspects from this new set of timeframe modules. The final critical interval is determined as the union of intervals wherein the selected timeframe module are defined.

The result of timeframe diagnosis can help to set up the next debug experiment, such that data acquisition starts at the right cycle(s), i.e., the one(s) as close to the critical cycle as possible. This interval can further reduce the time interval

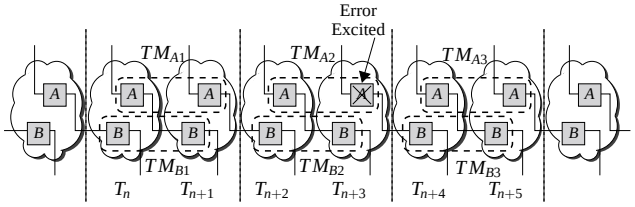



Fig. 3. Timeframe diagnosis

where the design needs to be analyzed in the next debug session. The trace can also be truncated to start at the same cycle as the begin of the returned interval. The idea is that the segment of the trace before the critical cycle can be safely removed for debugging analysis since it does not contain information related to the error observed (which is excited at the critical cycle). The value of state elements w.r.t. the truncated trace can be initialized with the value of scan dump at this new starting cycle.

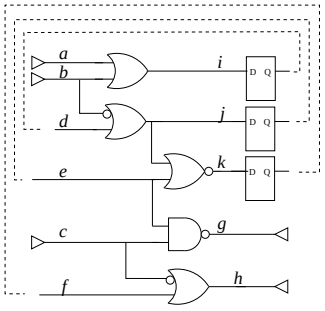
*Example 1:* Consider a test vector interval between cycles $T_n$ and $T_{n+5}$, as shown in Figure 3. From hierarchical diagnosis, it is known that modules *A* and *B*, shown in that figure as grey boxes, are suspects. To improve the estimate for the time interval where the error is excited, timeframe modules that consider two cycles at a time (i.e., $k = 2$) are created. These timeframe modules are shown in dotted rectangles (e.g., $TM_{A1}$ consists of $\{A^{T_n}, A^{T_{n+1}}\}$). Assume that the error is excited in module *A* at cycle $T_{n+3}$, that is, the grey box marked with an $\times$. As such, timeframe diagnosis returns solutions consisting of $TM_{A2}$ and $TM_{B3}$. Hence, timeframe diagnosis can deduce that the critical interval is $(T_{n+2}, T_{n+5})$ as defined by $TM_{A2}$ and $TM_{B3}$. Consequently, only cycles between $T_{n+2}$ and $T_{n+5}$ are analyzed in the next debug session.

### C. Data Acquisition Setup

Trace buffers provide the engineer great flexibility in the choice of traced signals. However, the buffers can only trace a limited subset of signals. In most real-world designs, only a small set of hard-wired signals can be traced during the execution.

Among all traceable registers, the engineer wants to select ones that are related to the error source or provide valuable information to aid in pruning suspects. A simple approach to identify those registers is using X-simulation [19], which simulates the design with logic unknown at the output of the suspects to capture all possible paths for error propagation. Then, any registers that store logic unknown are the candidates for tracing. Because X-simulation is a pessimistic process, it may return too many registers to make the information useful.

To improve resolution and accuracy, an alternative selection algorithm is presented in [21], which utilizes the proof of unsatisfiability generated by SAT solvers. Given an erroneous circuit, the input vector sequence, and the correct output response, the CNF formula of the ILA representation of the circuit is unsatisfiable due to the contradiction between the erroneous output response and the correct output response. Intuitively, the contradiction can occur at any signals along the paths from the actual fault location to the output where discrepancies are observed. As discussed in Section II-C, an UNSAT core of an unsatisfiable SAT problem is a subset of clauses that is also unsatisfiable. Therefore, signals associated with clauses in the UNSAT core can be potential locations for tracing and provide information about the behavior of the failure. Note that each UNSAT core can potentially represent different error propagation paths. To ensure that all paths are considered, the union of all UNSAT cores should be considered.




(a) Erroneous Circuit

| Cycle | Vector $\{abc\}$ $(v)$ | Response $\{g, h\}$ Correct $(y_{corr})$ | Erron. $(y_{err})$ |
|---|---|---|---|
| 1 | 100 | 11 | 11 |
| 2 | 011 | 01 | 01 |
| 3 | 110 | 11 | 11 |
| 4 | 111 | 11 | 00 |

(b) Test vector sequence and response. The initial value of {`d, e, f`} is 000

Fig. 4. Example erroneous circuit. The correct implementation of gate `i=OR(a,b)` is `i=AND(a,b)`

*Example 2:* Consider the circuit shown in Figure 4(a). Assume the error is at `i`, where the correct implementation is `i=AND(a,b)`. The test vector and the correct/erroneous response are shown in Figure 4(b). Since the circuit is erroneous, the CNF formula, $\Phi = \cup_{i=1}^{4}(C^i \cdot v^i \cdot y^i)$, is unsatisfiable. Due to the space limitation, the formulation of $\Phi$ is not shown. However, the construction can be done in linear time as shown in [22]. Given $\Phi$ to the SAT solver (e.g., MiniSAT [17]), an UNSAT core of the instance can be extracted from the proof of unsatisfiability provided by the solver as shown below.

$$\{(\overline{j^3} + e^4) \cdot (\overline{c^4} + \overline{e^4} + \overline{g^4}) \cdot (\overline{d^3} + j^3) \cdot (\overline{i^2} + d^3) \cdot (\overline{b^2} + i^2) \cdot (c^4) \cdot (g^4) \cdot (b^2)\}$$

By examining the UNSAT core, variables that represent registers can be extracted: $d^3$ ( from the clause $(\overline{i^2} + d^3)$) and $e^4$ (from the clause $(\overline{j^3} + e^4)$). Therefore, signals that should be traced are `d` at cycle 3 and `e` at cycle 4.

As mentioned in Section II-A, there are two ways to obtain the values of those registers. One way is through the use of scan dumps, if they are scannable. This can be time inefficient since tests need to be reset and started over again after each dump. The second approach is tracing these registers with trace buffers. However, the issue here is that only a limited set of registers can be traced in practice. To alleviate this issue, a searching technique is proposed in [21] to obtain the value of non-traceable registers indirectly by implication using other traceable registers.

Given a set of traceable registers and a target untraceable register, this technique finds a subset of traceable registers

TABLE I
TESTCASE CHARACTERISTICS

| Ckt. name | Ckt. description | # of gates | # of states | # of modules |
|---|---|---|---|---|
| divider | 16-bits divider | 5276 | 510 | 31 |
| spi | spi core | 1889 | 162 | 79 |
| wb | WISHBONE Conmax IP core | 2253 | 110 | 94 |
| rsdecoder | Reed-Solomon Decoder | 10265 | 521 | 481 |

such that their values can be used to restore the value of the target register through the means of forward implications and backward justifications. The technique formulates the problem into a SAT instances and identifies possible implications within a bounded timeframe window. At the end, engineers can trace those alternative registers for further debugging.

## IV. CASE STUDIES

This section presents the study of the presented methodology on OpenCores.org designs. Minisat [17] is used as the underlying SAT-solver. Experiments are conducted on Core 2 Duo 2.4GHz process with 4 GB of memory. All runtimes are reported in seconds. In each testcase, a single random functional error (e.g., wrong assignment, incorrect case state, etc) is inserted into the RTL code. Test vectors are extracted from the test-bench provided by OpenCores.org. The trace length is between 100 to 300 time frames. Finally, to fully take advantage of hierarchical diagnosis, building blocks of HDL code, such as a case statement or an if-statement, are parsed as a module.

The first set of experiments examined the two parameters that can affect the performance of the presented diagnosis methodology. These two parameters are the level of hierarchy that the hierarchical diagnosis examines at each session ($n$), and the timeframe module interval sizes used in the timeframe diagnosis ($k$). The size of the trace buffer is assumed to be 16*128 bits. It is assumed that 80% of registers in each design are traceable and they are divided into groups of at most 16. In each debug session, the buffer can store values of one group for at most 128 cycles or two groups for at most 64 cycles.

Table I summarizes the characteristics of designs used in this study. The name and a short description of the design are given in the first two columns. The third and fourth columns reports the number of primitive gates and the number of registers for each design. The fourth column shows the number of the modules at the lowest level of hierarchy. This is also the number of suspects one needs to examine in a brute-force manual silicon approach.

Figure 5 shows the total numbers of modules returned by each hierarchical diagnosis round when various numbers of hierarchy levels are examined in one debug session. In general, the numbers are increased as hierarchical diagnosis runs more rounds in one debug session. This is because fewer state values are available and the diagnosis algorithm cannot distinguish some of the suspects. Nevertheless, comparing the result to the number of modules shown in the fourth column of Table I, we can see that in all cases the presented methodology can significantly prune the modules that one needs to examine.

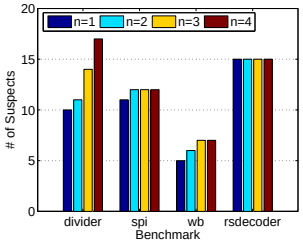



Fig. 5. Impact of depth $n$ in Performance

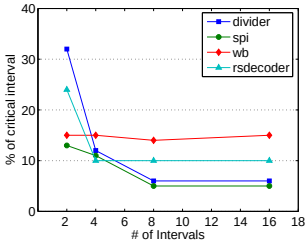



Fig. 6. Impact of number of intervals used in timeframe diagnosis

TABLE II
TRACEABLE REGISTER GROUP INFORMATION

| Circ. | Gate count | Total reg. | # of groups | # of reg./group | Perc. |
|---|---|---|---|---|---|
| spi | 2832 | 162 | 8 | 8 | 40% |
| hpdmc | 20536 | 453 | 16 | 8 | 28% |
| usb | 39179 | 2054 | 32 | 16 | 25% |

Figure 6 shows the ratio of the size of the critical interval after the last debug session compared to the original trace length when various numbers of interval are used in timeframe diagnosis. Four cases are considered: 2, 4, 8 and 16 intervals. As expected, greater reductions are achieved with finer-grain intervals. The only exception is `wb` in the case where the interval size is 16. In this case, the error happens to be excited across two intervals, which results in a wider range. In all cases, over 50% of reduction is achieved.

The second set of experiments demonstrates the effectiveness of the UNSAT-core register selection, as well as this of the searching algorithm. To emulate the real trace buffer hardware structure, a subset of registers of each design is selected randomly as traceable by the trace buffer. These registers are divided into groups and the grouping configuration is summarized in Table II. The first column lists the designs used in this experiment; the size of the designs in terms of the number of primitive gates is reported in the second column. The third column of the table shows the total number of registers in each design. The fourth and fifth columns have the number of the register groups and the number of registers in each group, respectively. The sixth column shows the percentage of total registers that can be traced.

The algorithm is configured to perform one-level hierarchical diagnosis and the timeframe diagnosis divides the time interval into two timeframe modules. For the searching algorithm, the window size is set to be six time frames.

Table III summarizes the performance of debug analysis under two situations: debug without values of registers (columns 2 – 4) and debug with values of registers selected by the UNSAT core-based selection procedure (columns 5 – 11). Each row is one individual case that contains a different bug in the design. The sum of the number of modules returned at the end of each debug session is shown in the second and fifth columns. This is the total number of modules that the engineer needs to investigate. The ratio of these two columns is reported in the sixth column, which means the percentage reduction on the number of suspects. The third and seventh columns

TABLE III
PERFORMANCE OF DEBUGGING WITH PROPOSED TECHNIQUES

| Circ. | No state value used | | | With UNSAT-core-based register selection | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | # of susp. | # of sessions | Runtime (s) Diag. | # of susp. | % reduction | # of sessions | # of traced sig. | Runtime(s) Diag. | Search | Total increased |
| spi | 146 | 11 | 1990 | 73 | 50% | 11 | 24 | 828 | 1011 | 0.92 |
| | 144 | 11 | 179 | 76 | 48% | 9 | 32 | 101 | 94 | 1.09 |
| hpdmc | 213 | 17 | 3817 | 170 | 21% | 17 | 40 | 2323 | 15734 | 4.73 |
| | 167 | 16 | 2321 | 131 | 22% | 15 | 40 | 1963 | 14233 | 6.98 |
| usb | 103 | 15 | 3795 | 38 | 74% | 11 | 64 | 1609 | 9218 | 2.85 |
| | 224 | 14 | 7091 | 138 | 39% | 7 | 128 | 4245 | 18519 | 3.49 |

show the number of debug sessions performed. The number of registers traced by the trace buffer is shown in the eighth column. Finally, the runtime of the diagnosis procedure of both situations is reported in the fourth and ninth columns. In the case of the presented methodology, the additional runtime for searching the registers for tracing is recorded in the $10^{th}$ column and the total runtime is shown in the $11^{th}$ column.

As shown in the sixth column of the table, one can see that for all cases the algorithm can effectively eliminate more false candidates when it utilizes the values of registers. The reduction can be as high as 74% (i.e., case 1 of `usb`). The result also indicates that fewer debug sessions are required to find the root cause of the failure. All of those are achieved with tracing a small amount of registers. The benefit of the UNSAT-core-based technique is shown when one considers the reductions in both the number of suspects and the number of debug sessions. Furthermore, because of the reduction of suspects and debug sessions, the runtime for diagnosis is reduced in the case of the presented methodology. However, the presented methodology requires additional computation for the searching algorithm. As shown in the table, this computation can be significant in cases such as `hpdmc`. This is because the algorithm has a higher failing rate on finding the recommendation for the non-traceable registers in those cases. Since the number of the final suspects is reduced significantly, this additional runtime may be acceptable if there is a greater amount of time saved by manually inspecting fewer suspects.

## V. CONCLUSION

Automated software silicon debug solutions are a necessity today to ease the task of the test/design engineer during chip failure analysis. In this paper, we present a debugging methodology that comprises of multiple iterative debug sessions. At each session, the methodology uses the circuit hierarchy to debug the failure and also narrows down the window of cycles wherein the error is exercised. The methodology also contains techniques to aid in selection of traceable registers to be traced in the next debug session such that the diagnosis can benefit from the new data. Case studies are presented to confirm the effectiveness of the approach.

## REFERENCES


[1] A. Gupta, S. Malik, and P. Ashar, "Toward formalizing a validation methodology using simulation coverage," in *Design Automation Conf.*, June 1997, pp. 740–745.

[2] J. Kumar, N. Strader, J. Freeman, and M. Miller, "Emulation verification of the Motorola 68060," in *Int'l Conf. on Comp. Design*, Oct. 1995, pp. 150– 158.

[3] J. Jan, A. Narayan, M. Fujita, and A. S. Vincentelli, "A survey of techniques for formal verification of combinational circuits," in *Int'l Conf. on Comp. Design*, Oct. 1997, pp. 445–454.

[4] G. Parthasarathy, M. K. Iyer, K. T. Cheng, and L. C. Wang, "Safety property verification using sequential SAT and bounded model checking," *IEEE Design & Test of Comp.*, vol. 21, no. 2, pp. 132–143, March 2004.

[5] J. Jaeger. (2007, Dec.) Virtually every ASIC ends up an FPGA. EETimes. [Online]. Available: http://www.eetimes.com/showArticle.jhtml;jsessionid=JRHN5OJ1CLD2SQSNDLP%SKH0CJUNN2JVN?articleID=204702700

[6] P. M. Rosinger, B. M. Al-Hashimi, and N. Nicolici, "Scan architecture with mutually exclusive scan segment activation for shift- and capture-power reduction," *IEEE Trans. on CAD*, vol. 23, no. 7, pp. 1142–1153, July 2004.

[7] H. F. Ko and N. Nicolici, "Algorithms for state restoration and trace-signal selection for data acquisition in silicon debug," *IEEE Trans. on CAD*, vol. 28, no. 2, pp. 285 – 297, Feb. 2009.

[8] J.-S. Yang and N. A. Touba, "Automated selection of signals to observe for efficient silicon debug," in *VLSI Test Symp.*, May 2009, pp. 79 – 84.

[9] X. Liu and Q. Xu, "Trace signal selection for visibility enhancement in post-silicon validation," in *Proc. of Design, Automation and Test in Europe*, 2009, pp. 1338 – 1343.

[10] O. Caty, P. Dahlgren, and I. Bayraktaroglu, "Microprocessor silicon debug based on failure propagation tracing," in *Proc. of Int'l Test Conf.*, Oct. 2005, pp. 284–293.

[11] C. C. Yen, T. Lin, H. Lin, K. Yang, T. Liu, and Y. C. Hsu, "Diagnosing silicon failures based on functional test patterns," in *Int'l Workshop on Microprocessor Test and Verification*, Dec. 2006, pp. 94–97.

[12] S. Venkataraman and W. K. Fuchs, "A deductive technique for diagnosis for bridging faults," in *Proc. of Int'l Conf. on CAD*, Nov. 1997, pp. 562–567.

[13] F. M. D. Paula, M. Gort, A. J. Hu, S. Wilton, and J. Yang, "BackSpace: Formal analysis for post-silicon debug," in *Int'l Conf. on Formal Methods in CAD*, 2008, pp. 1–10.

[14] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, Oct. 2005.

[15] M. W. Moskewicz, C. F. Madigan, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Design Automation Conf.*, 2001, pp. 530–535.

[16] J. P. Marques-Silva and K. A. Sakallah, "GRASP: a new search algorithm for satisfiability," *IEEE Trans. on Comp.*, vol. 48, no. 5, pp. 506–521, May 1999.

[17] N. Eén and N. Sörensson, "An extensible SAT-solver," in *SAT*, 2003, pp. 502–518. [Online]. Available: http://dblp.uni-trier.de/db/conf/sat/sat2003.html#EenS03

[18] Z. Fu and S. Malik, "On solving the partial max-sat problem," in *SAT*, 2006, pp. 252–265.

[19] V. Boppana and M. Fujita, "Modeling the unknown! towards model-independent fault and error diagnosis," in *Proc. of Int'l Test Conf.*, Oct. 1998, pp. 1094–1101.

[20] M. F. Ali, S. Safarpour, A. Veneris, M. S. Abadir, and R. Drechsler, "Post-verification debugging of hierarchical designs," in *Proc. of Int'l Conf. on CAD*, Nov. 2005, pp. 871–876.

[21] Y.-S. Yang, A. Veneris, and N. Nicolici, "Automating data analysis and acquisition setup in a silicon debug environment," *IEEE Trans. on VLSI Systems*, 2012.

[22] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Trans. on CAD*, vol. 11, no. 1, pp. 4–15, Jan. 1992.