Automated Data Analysis Solutions to Silicon Debug

Yu-Shen Yang
Dept. of ECE
University of Toronto
Toronto, M5S 3G4
yangy@eecg.utronto.ca

Nicola Nicolici
Dept. of ECE
McMaster University
Hamilton, L8S 4K1
nicola@ece.mcmaster.ca

Andreas Veneris
Dept. of ECE & CS
University of Toronto
Toronto, M5S 3G4
veneris@eecg.utronto.ca

ABSTRACT

Since pre-silicon functional verification is insufficient to detect all design errors, re-spins are often needed due to malfunctions that escape into the silicon. This paper presents an automated software solution to analyze the data collected during silicon debug. The proposed methodology analyzes the test sequences to detect suspects in both the spatial and the temporal domain. A set of software debug techniques are proposed to analyze the acquired data from the hardware testing and provide suggestions for the setup of the test environment in the next debug session. A comprehensive set of experiments demonstrate its effectiveness in terms of run-time and resolution.

1. Introduction

During the integrated circuit development cycle, designs often go through several verification steps (e.g., functional, timing, power) before a silicon prototype is manufactured. Pre-silicon verification uses formal methods [1, 2] or simulation approaches [3] to check the functionality of the Register-Transfer Level (RTL) model against its specification (e.g. a behavior model). As the design size increases and the intellectual property (IP) blocks developed by different vendors are integrated together, silicon prototypes are rarely bug-free. There are several reasons for this to happen. Due to time-to-market constraints, 100% *verification coverage* at the RTL level remains an elusive task. As such, functional bugs may escape pre-silicon verification only to be discovered during in-system silicon validation where the design is exercised at speed. In addition, parasitics and the unmodeled process variation effects during the fabrication also take their toll.

With the above observations at hand, it comes as no surprise that more than 60% of design tape-outs require a re-spin. More than half of the failures are not due to power or timing defects but due to logical or functional errors not discovered during verification [4]. These silicon re-spins increase costs and the time-to-market margins dramatically. 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.

The challenge for automated software solutions for analyzing data in silicon debug is multifold. Unlike RTL verification, the error/defect behavior can be either *deterministic* or *non-deterministic*. Deterministic errors replicate their behavior with the same set of test vectors. This is the case if the circuit is debugged on the tester or on an application board where the inputs are controlled synchronously. On the other hand, if the sources of the board have non-deterministic behavior (e.g., interrupts from peripherals or timing of refresh cycles for dynamic memories [5]), the error can be triggered by an event that cannot be replicated deterministically. Another difference between RTL verification and silicon debug is that in simulation, traces can be collected for as many signals and as many clock cycles as the verification engineers decides to probe. In contrast, silicon debug observability is restricted in both space and time. Although *Design-for-Debug (DfD)* techniques (e.g. scan chains, trace buffers, etc) improve the observability of the internal signals, the amount of data extracted from a chip is limited by these techniques.

Once silicon fails test, a typical debug process consists of several iterative sessions to find the error. In each debug session, shown in Figure 1, test engineers setup the environment to obtain appropriate data from a certain subset of nets at pre-determined operational cycles. This data is analyzed to prune the candidate causes and to determine the best-fit environment for the next debug session. A series of debug sessions is iterated until the root cause is determined.

This paper proposes an automated software-based debug methodology to aid the engineer in discovering the root cause (*i.e.,* location) of chip failure. This methodology acts complementary to current silicon debug hardware and software solutions used for *data acquisition*. The major contribution of this work is that it automates the *data analysis* step in Figure 1 to help identify the location of the suspect(s) in a hierarchical manner. It also provides suggestions for the setup of the test environment in the next debug session by giving a better estimate for the *window (time interval)* of cycles the engineer should concentrate to catch the error. This data is added to the subsequent automated data analysis cycle to eventually determine the root cause.

A set of comprehensive experiments on OpenCores circuits is conducted. Results show that our methodology successfully determines the location of the error and it also specifies with accuracy the time interval that it is excited. As such, the proposed methodology adds value to any silicon debug environment.

In the remaining paper, Section 2 summarizes the known DfD techniques. Section 3 illustrates the new methodology. Section 4 contains experiments and Section 5 concludes the work.

2. Background

The silicon debug process entails different hardware and software components. The former refers to DfD techniques that improve signal observability. The later includes debugging software and the overall environment setup to integrate the different tools that collect and analyze the data from the tester. In the following subsections, we briefly review some of this background material.

2.1 Design for Debug Hardware Solutions

There are two main DfD techniques used in practice: scan chains and trace buffers. *Scan chains* are commonly employed in manufacturing test as a Design-for-Test (DfT) technique. This hardware can be reused during silicon debug [6]. During the test mode, the state for all the scanned registers can be extracted by performing a scan dump. Unless each scanned register has two elements, which leads to excessive area investment, after each scan dump the test environment needs to be restarted. Even if two state elements are present in each scanned register, a new state capture cannot occur until the previous scan dump has been completed [7].

Another DfD technique uses *trace buffers* [8, 9]. A trace buffer is based on an on-chip memory that records internal signals. It contains control logic, called trigger logic (e.g., embedded hardware assertions),

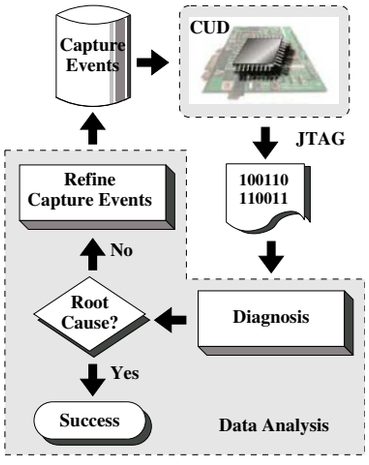


Figure 1: A typical silicon debug flow

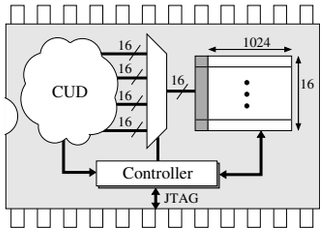


Figure 2: Trace buffer configuration

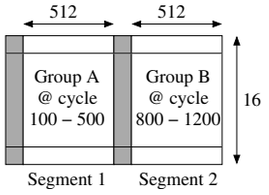


Figure 3: Trace buffer segmentation

employed for on-line monitoring of circuit behavior. Once the trigger condition is asserted the on-chip memory can start/stop recording the logic values of the selected signals. Subsequently, the recorder data can be read via a low-bandwidth interface, such as boundary scan. Typical sizes for trace buffers range from 8K to 256K. Clearly, due to the size limitation for this on-chip memory, only a subset of pre-selected signals can be traced in each debug session.

2.2 Related Work on Debug Data Analysis

In this section we examine some of the relevant data analysis solutions. The method proposed by Caty et. al. [10] identifies the fault propagation paths by back tracing from the failing registers for each timeframe. Then it performs a forward tracing from the registers to further narrow down the root cause candidates. Their analysis relies on scan dumps for multiple consecutive cycles.

Yen et. al. [11] propose a similar approach. Their methodology 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. After the cycle is identified, the suspect list is pruned using both a path-tracing method [12] and simulating the faulty value of the suspect in the golden model.

For previously described methods to work, the complete golden reference has to be available, a pre-requisite which is not always true. For instance, in the case of functional errors, the golden model can be a behavior model (e.g. a software program in MATLAB, C/C++). This model can be simulated to obtain the expected responses at the primary outputs, but there may not exist one-to-one signal mapping from all registers in the actual design to variables in the behavioral model. Hence, there may be only a *partial state equivalence* between the implementation and the specification. In this case, only logic values for the registers that have a reference mapping in the golden model can be checked, a fact that may deteriorate the final resolution, as explained in [13].

3. Proposed Methodology

In this section, we describe the novel silicon debug data analysis methodology. We use the following assumptions:

- The erroneous silicon behavior is deterministic. Given the same input sequence, the responses are consistent in all debug sessions. This assumption is necessary to replicate experiments and obtain the values of multiple state elements at different cycles. It is also the fundamental underlying assumption of a silicon debug environment such as the one depicted in Figure 1.
- We assume that access to the values of internal states is available, because scan chains and trace buffers (Section 2.1) are utilized. In this scenario, the design is fully scanned and trace buffers can be programmed to capture the value of specific state elements (as explained later in the paper).
- In this paper we deal with functional errors (bugs) in the design. For examples of functional bugs that escape to silicon, we refer the reader to [14]. Malfunctions due to electrical and fabrication defects are not considered in this paper.
- The golden model during diagnosis is a high-level behavioral model. Hence, the proposed framework assumes a partial state equivalence. That is, there are state elements which cannot be mapped from the silicon implementation to the golden model.
- We assume that all discrepancies are due to a single error present in the silicon. Since most test vectors target specific functionalities of the design, it is realistic to ascertain that a test vector that fails is due to a single error [15].

Note, not every signal from the circuit can be probed into the trace-buffer (this will lead to unacceptable area of the debug module). Instead, we define groups of signals of width equal to the trace buffer width and we place a multiplexer at the data input of the trace buffer. This is illustrated in Figure 2 where four groups of 16-bit signals are fed into a trace buffer of depth 1024 and width 16. Another common trace buffer feature that is used is the segmentation. For example, in Figure 3, a trace buffer of depth 1024 has two segments: in the first segment we sample group A from clock cycles 100 to 500; in the second segment we sample group B from clock cycles 800 to 1200. This feature is relevant when the length of the test vector traces is reduced below the depth of the trace buffer. By exploiting segmentation, useful data can be collected from two different groups (required for two different suspects) at different times in the same debug session.

3.1 Methodology Overview

The complete flow of the methodology is summarized in Figure 4. An overview of the methodology is given in this subsection with the

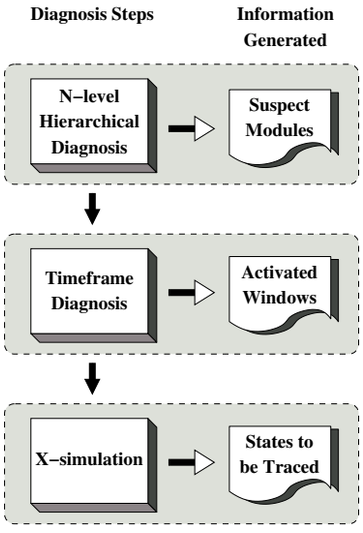


Figure 4: A single debug session

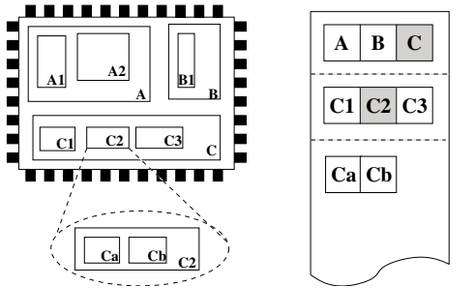
details of the implementation described in the remaining subsections. The objective of the proposed methodology has three main goals: to identify the suspect modules that contain the error, to find the critical interval of the error, and to find the state elements that may be on the error propagation paths. Note, a *critical interval* is a window of cycles that contains the critical cycle. Unlike for source code (or RTL debug), the above objective must be achieved with a conscious usage of the on-chip debug hardware resources. This objective is unique to silicon debug and it motivates the key contributions in this paper.

The algorithm begins with reducing the length of the test trace by finding a new starting clock cycle for it, since the test trace can be long to manipulate. The idea is that vectors before the critical cycle can be safely removed for debugging analysis. This is because this portion of the sequence contains information unrelated to the error which is excited at the critical cycle. The initial state of state elements can be replaced with the value from a scan dump at the new starting cycle.

Next, the algorithm performs debugging in three steps. First, it diagnoses the circuit in a hierarchical manner to reduce the complexity of the analysis. It has been shown that using the design hierarchy information for searching between different components of a design is effective [16]. Then, timeframe diagnosis is carried out to find a greater precision estimate for the window of clock cycles in which the error may be excited. This interval can further reduce the length of the test vector trace needed to be analyzed in the next debug session. In addition, during the test, signals only need to be traced within the new reduced window. Finally, X-simulation [17], simulating the design with logic unknown at the output ports of the suspects, is performed to identify the state elements where the error effects propagate. The above information feeds back to the algorithm to drive the next debug session where the algorithm iterates the three steps in Figure 4. During each debug session, a single scan-dump is conducted to collect additional register values for one clock cycle at the begin and end of the critical interval.

3.2 Hierarchical Diagnosis

The backbone of our diagnosis algorithm is based on hierarchical diagnosis similar to that proposed by Ali et. al. [16]. In brief, the algorithm takes in the RTL code, failing input test vectors and the expected (*i.e.,* correct) output responses and builds a Boolean satisfiability instance. It then enters different rounds of hierarchical diagnosis as it goes deeper in the design hierarchy by considering only sub-modules



(a) Hierarchical design

(b) History of hierarchical diagnosis

Figure 5: Hierarchical diagnosis

Algorithm 1 Timeframe diagnosis

```
1: M_List := list of suspect modules
2: k := size of timeframe interval
3: T_b(T_e) := beginning(end) timeframe of the trace
4: procedure TIMEFRAMEDIAGNOSIS(M_List, k, T_b, T_e)
5:     TM_List := the new list containing timeframe modules
6:     TM_sol := the timeframe diagnosis solutions
7:     for all S ∈ M_List do
8:         for t = T_b to T_e incremented by k do
9:             TM_new := A new timeframe module consists of
                   {S_t ··· S_{t+k}}
10:            Add TM_new to TM_List
11:        end for
12:    end for
13:    TM_sol ← Debug with candidates from TM_List
14:    return TM_sol
15: end procedure
```

of the modules that are determined to be suspects previously. This is repeated until the lowest level of hierarchy is reached where it terminates and returns the suspects. The following example illustrates the concept of debugging using hierarchy information.

EXAMPLE 1. *Figure 5(a) shows a design and its hierarchical structure. Applying hierarchical diagnosis on this design for two rounds is shown in Figure 5(b). The design has three modules at the top level. After the first iteration, module C (shaded box) is diagnosed to be the solution. Therefore, in the second round, only the sub-modules of module C, namely, C_1, C_2, and C_3 are considered as suspects. At that round, C_2 is identified as the solution and the suspect candidate list for the third round contains only C_a and C_b.*

In the proposed setup, in every debug session the algorithm parses suspect modules from the previous session and performs hierarchical diagnosis for at most n levels from the level ended in the last session. The number n is the number of hierarchy levels that the algorithm would expand in each session. This process terminates when the method reaches the lowest level of design hierarchy. For example, if $n = 2$ and the maximum hierarchy depth of the design is 10, the algorithm will run at most five sessions. Each time, it goes deeper in the hierarchy by two levels and truncates the test vector trace (using the technique in the next sub-section) to increase performance and resolution.

3.3 Timeframe Diagnosis

In RTL debugging the length of the collected traces used in diagnosis is not limited by the amount of data that can be stored on the chip.

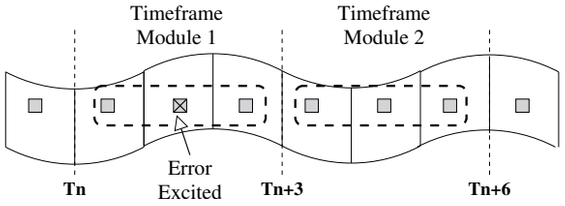


Figure 6: Timeframe diagnosis

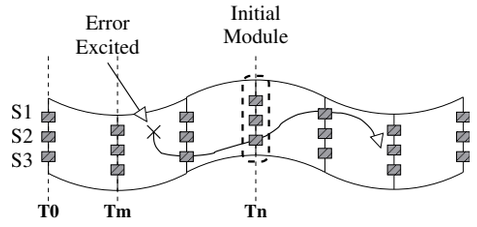


Figure 7: Test vector trace reduction

However, in silicon debug, the depth of the trace buffer limits the number of samples that are acquired in one debug experiment. *This unique constraint, motivates timeframe diagnosis*.

A timeframe diagnosis pass narrows down the critical interval and it helps 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. Note, the test still runs from the beginning of the test vector sequence. The trace buffer is programmed to begin the capture at a later cycle. In the following description, a module M at timeframe t is denoted as M_t. $INPUT(M)$ ($OUTPUT(M)$) indicates the input (output) nets of M.

DEFINITION 1. *A **timeframe module** TM for a design module M over a set of clock cycles $\{T_n \cdots T_{n+k}\}$ is an conceptual entity that contains the instances $M_{T_n} \cdots M_{T_{n+k}}$ of module M over this set of clock cycles such that $INPUT(TM) = \bigcup_{t=T_n}^{T_{n+k}} INPUT(M_t)$ and $OUTPUT(TM) = \bigcup_{t=T_n}^{T_{n+k}} OUTPUT(M_t)$*

Pseudo-code to identify the critical interval is described in Algorithm 1. Recall, hierarchical diagnosis returns a list of suspect modules. Timeframe diagnosis divides the trace into several intervals of width k and constructs a timeframe module for each interval. The timeframe module considers the suspect module in each cycle of the interval as a single suspect. This is shown in lines 8–11. Intuitively, instead of diagnosing single-cycle timeframe modules, we examine timeframe modules that are sets-of-cycles. Consequently, suspects are selected from this new set. If the timeframe module contains the critical cycle, or it is the interval between the critical cycle and the cycle in which the erroneous effects is observed, it will be selected as a solution. The resulting critical interval is the union of the timeframe modules in the solution.

EXAMPLE 2. *We continue from Example 1 in Figure 6. Assume we examine a test vector interval between cycles T_n and T_{n+6}. From hierarchical diagnosis, we know that Module C2, shown in a gray box through the different cycles, is a suspect. To improve the estimate where the error is excited, we do not look at a single cycle, but we consider timeframe modules three cycles at a time. The timeframe modules for $k = 3$ are shown in dotted rectangles. If the error is excited at cycle T_{n+1} (gray box marked with an X) and the values of registers at T_{n+3} are observable, timeframe diagnosis can deduce that the time interval defined by* `Timeframe Module 1` *is a critical interval.*

The algorithm guarantees that one of the selected timeframes modules is the critical interval. Hence, the subsequent analysis can focus on the trace within the begin/end cycles defined by the solution timeframe module. In Example 2, because `Timeframe Module 1` is the only timeframe module selected, we analyze the set of cycles between T_n and T_{n+3} in the next debug session. The value of k defines a tradeoff between performance and resolution. The more timeframe modules one has to examine, the more candidates need to be considered at every iteration of the algorithm. In early debugging sessions, a larger value for k may be more preferable for some coarse-grain analysis. Since failing test vectors can contain many cycles, short timeframe modules will introduce a lot of candidates that take more time to screen. On the other hand, having excessively long timeframe modules intervals may not always be a good practice at later stages.

3.4 Test Vector Trace Reduction

In silicon debug, the length of the test vector trace determines how many debug experiments need to be carried out to collect the trace of interest. One way to reduce the length is by comparing scan dumps and the golden model simulation results, as described in [11]. If they match, we can assume that the error is excited in a later cycle. Therefore, this provides a conservative estimate for the new *initial cycle* for the test vector trace. However, this approach would require time-consuming scan dumps and it is limited when we have partial state equivalence. If the error effect propagates through the state elements with no golden model reference, the discrepancy will not be detected early.

To ensure that diagnosis fails when the error is excited at an earlier cycle, we introduce an additional coarse-grain module, called *initial module*, in the suspect list. This module considers all the state elements of the initial cycle as one candidate suspect. If the critical cycle is before the new truncated test vector trace, the initial module will be selected by the diagnosis algorithm to indicate that the error effects originate at a cycle before the initial state. In that case, the complete set of debug sessions is repeated with a new initial state at an earlier cycle.

EXAMPLE 3. *The design in Figure 7 contains three state elements, $\{S_1, S_2, S_3\}$. Assume that only S_1 and S_2 have a golden model reference, and that the error is excited at cycle T_m and propagated along the path shown in the figure. In this case, one may consider cycle T_n as the new initial starting cycle of the test vector trace during diagnosis, since S_1 and S_2 contain no discrepancy. This can result in an incorrect diagnosis result. Hence, by introducing an initial module (the dotted rectangle) that contains $\{S_1, S_2, S_3\}$ at the timeframe T_n, diagnosis will capture the error effects by returning the initial module as the solution. At that time, diagnosis has to be restarted with a new initial cycle estimate before T_n.*

4. Experiments

In this section, experiments on industrial designs are presented. We investigate two factors that impact the method performance and its resolution from Section 3, namely, the depth of the hierarchical rounds in each debug session (n) and the width of the interval in timeframe diagnosis (k). Experiments are carried out on OpenCores circuits and run on a Core 2 Duo 2.4 GHz processor with 4 GB of memory. All runtimes are reported in seconds.

Each experiment contains the average of five runs. In each run, a random functional error (wrong assignment, incorrect case statements, etc) is inserted into the RTL code. The test vector is extracted from the testbench provided by OpenCores. In experiments, unless mentioned otherwise, we use the following set of parameters:

- We randomly select 80% of the state elements to generate an environment with partial state equivalence.

Table 1: Benchmark characteristics

ckt. name	ckt. description	# of gates	# of state elements	# of trace modules	hier. depth max	hier. depth avg
divider	16-bits divider	5276	510	26	7	4.3
spi	spi core	1889	162	8	10	4.6
wb	WISHBONE Conmax IP core	2253	110	6	11	4.9
rsdecoder	Reed-Solomon Decoder	10265	521	27	12	5.4

Table 2: Performance of the methodology

ckt.	# of sessions	total time (sec)	total groups traced	# of final suspects	% of trace reduced
divider	4	123.1	7	11	88%
spi	4	351.5	6	8	89%
wb	3	101.4	3	6	86%
rsdecoder	4	162.2	5	15	90%

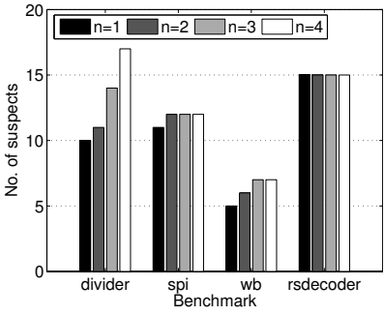
- During hierarchical diagnosis we set $n = 2$, that is, the algorithm goes two levels in the hierarchy deep at each debug session.
- At each session, timeframe diagnosis divides the test vector trace into four timeframe modules of an equal number of cycles each.

The size of the trace buffer is assumed to be 16x128 bits. We divide the state elements that can be traced during debugging into groups of 16. The buffer can be used to store one of the groups for at most 128 cycles or two groups for at most 64 cycles.

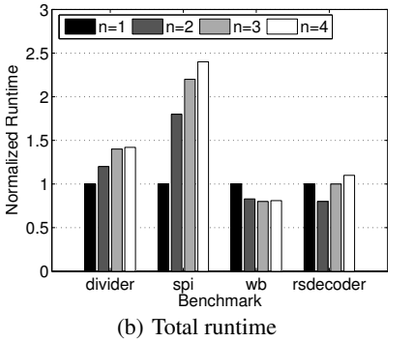
Experiments are conducted on four OpenCores circuits. Their characteristics are summarized in Table 1. A short description of the design is given in Column two. The next two columns show the number of gate and state elements of the circuits, respectively. The number of groups of state elements that can be traced is shown in Column five. Column six contains the number of the modules at the lowest level of hierarchy. This is also the total number of suspects one needs to examine in a brute-force manual silicon debug approach. The final two columns have the maximum and average hierarchical depth for each design.

Table 2 outlines general performance metrics for the methodology. The number of debug sessions and the total runtime for all sessions are shown in Column two and three, respectively. The next column shows the total number of groups of state elements that are traced during debug sessions. For example, seven groups are traced during debugging `divider`: one group is traced during the first session and two groups are traced during each of the remaining three sessions. One can see that in most cases two groups can be traced in one session. This is because that our methodology often reduces critical intervals to more than half in the first 1-2 sessions. As mentioned in Section 3. trace buffers can be divided into segments. Hence, when the width of the critical interval is smaller than the half of the width of trace buffers, two groups of signals can be traced in one hardware run. Column five has the number of final suspects in the lowest level of hierarchy need to be investigated by the engineers. Comparing this result to the total number of modules shown in Table 1, we observe more than a 90% improvement in resolution. The percentage of reduction in the length of the final test vector trace is shown in the last column. One can see that the length of the final test vector trace is 90% shorter in the best case and 88% shorter, on the average, than the original.

In the next set of experiments, we first examine the performance of the method for a varying value of n, the number of hierarchy level that hierarchical diagnosis examines at each session. Results for this experiment are depicted in Figure 8, which shows the total numbers of



(a) Final suspect percentage

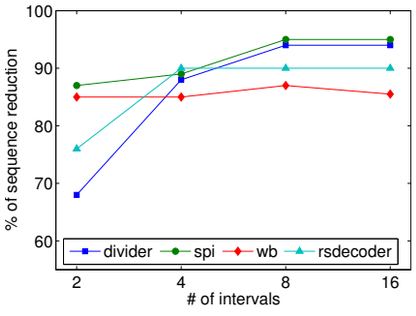


(b) Total runtime

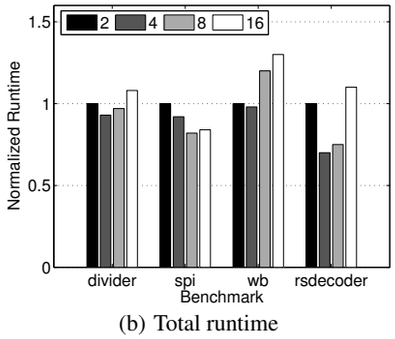
Figure 8: Impact of depth n in performance

modules returned by each hierarchical diagnosis round. In general, the number are increased as the hierarchical diagnosis runs more rounds in one debug session. This is because there are less state elements provided and the diagnosis algorithm cannot distinguish some of the suspects. However, in some cases, like circuit `rsdecoder`, the numbers of the modules are the same in all cases. The runtime is plotted in Figure 8(b) and is normalized by comparing it to the runtime of $n = 1$ in each benchmark. As shown, the runtime is increased as n increases. This is because the hierarchical diagnosis does not take the benefit from the trace reduction when it runs more iterations in one debug session. Recall that the timeframe diagnosis is carried out after the completion of n-level hierarchical diagnosis. Hence, with smaller values of n, diagnosis iterates less for the longer traces. One may notice that in some cases the best runtime happens when $n = 2$ and not when $n = 1$. This is because the timeframe diagnosis is carried out more frequently when $n = 1$. As the result, the overhead is greater than the time saved from the vector trace reduction.

Next, we profile the performance of the method for different timeframe module interval sizes in timeframe diagnosis. Figure 9(a) shows the percentage of the reduction of the test vector traces in the final debug session. As expected, greater reductions are achieved with finer-grain intervals. The only exception is the case where the interval is 16 for the circuit `wb`. In this case, the error happens to be excited across two inter-



(a) Final test vector trace reduction



(b) Total runtime

Figure 9: Performance effects of interval size k

Table 3: Reduction of test vector trace

ckt.	# of intervals				
	2	4	8	16	32
divider	53%	77%	76%	77%	77%
spi	74%	88%	90%	91%	91%
wb	25%	60%	63%	66%	68%
rsdecoder	77%	94%	96%	96%	96%

vals, which results in a wide range. In all cases, over 50% of reduction in the trace length is achieved. Table 3 summarizes the percentage of the reduction after the first half of debug sessions. It can be seen that in most cases, at least 50% of the traces are truncated after the first few rounds of timeframe diagnosis, an observation that reinforces the fact that the process is most effective at the beginning of the debug cycle.

The normalized runtime of those experiments is depicted in Figure 9(b). In general, as discussed in Section 3.3, it requires more computation if smaller intervals are used, since timeframe diagnosis has more candidates to screen. However, using `spi` as example, its runtime is reduced as the number of intervals increases. This is because approximately 90% of the traces are truncated after the first few sessions when the number of intervals is over eight, as shown in Table 3. As a result, the diagnosis in the latter debug sessions has much smaller traces to analyze and requires less computation.

5. Conclusion

Automated software-based silicon debug solutions are a necessity today to ease the task of the test/design engineer during chip failure analysis. In this paper, we propose a novel debugging methodology that comprises of multiple iterative debug sessions. At each session, debugging is performed using the circuit hierarchy. Additionally, the length of the failing test vector traces is reduced to alleviate the problem complexity for the next session. An extensive suite of experiments confirm the robustness and effectiveness of the approach that can be used as a stand-alone methodology or it can complement contemporary silicon debug software and hardware practice.

6. References

[1] 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.

[2] 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.

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

[4] J. Jaeger, "Virtually every ASIC ends up an FPGA," EETimes.com, Dec. 7 2007.

[5] S. Sarangi, B. Greskamp, and J. Torrellas, "CADRE: Cycle-Accurate Deterministic Replay for Hardware Debugging," in *IEEE International Conference on Dependable Systems and Networks (IDSN)*, June 2006, pp. 301 – 312.

[6] B. Vermeulen, T. Waayers, and S. K. Goel, "Core-based scan architecture for silicon debug," in *Proc. of Int'l Test Conf.*, Oct. 2002, pp. 638–646.

[7] 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.

[8] A. Abramovici and Y.C.Hsu, "A new approach to silicon debug," in *IEEE International Silicon Debug and Diagnosis Workshop*, Nov. 2005.

[9] E. Anis and N. Nicolici, "Low cost debug architecture using lossy compression for silicon debug," in *Proc. of Design, Automation and Test in Europe*, April 2007, pp. 1–6.

[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. 1–10.

[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] Y. Yang, A. Veneris, P. Thadikaran, and S. Venkataraman, "Extraction error modeling and automated model debugging in high-performance custom designs," *IEEE Trans. on VLSI Systems*, vol. 14, no. 7, pp. 763–776, July 2006.

[14] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas, "Patching processor design errors with programmable hardware," *IEEE. Micro*, vol. 27, no. 1, pp. 12–25, Feb. 2007.

[15] L. Huisman, "Diagnosing arbitrary defects in logic designs using single location at a time (SLAT)," *IEEE Trans. on CAD*, vol. 23, no. 1, pp. 91–101, Jan. 2004.

[16] 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. 6–10.

[17] 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.