On Signal Tracing in Post-Silicon Validation

Qiang Xu
CUhk REliable computing laboratory (CURE)
Department of Computer Science & Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
e-mail: qxu@cse.cuhk.edu.hk

Xiao Liu
CUhk REliable computing laboratory (CURE)
Department of Computer Science & Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
e-mail: xliu@cse.cuhk.edu.hk

ABSTRACT

It is increasingly difficult to guarantee the first silicon success for complex integrated circuit (IC) designs. Post-silicon validation has thus become an essential step in the IC design flow. Tracing internal signals during circuit's normal operation, being able to provide real-time visibility to the circuit under debug (CUD), is one of the most effective silicon debug techniques and has gained wide acceptance in industrial designs. Trace-based debug solution, however, involves non-trivial design for debug overhead. How to conduct signal tracing effectively for bug elimination is therefore a challenging task for IC designers. In this paper, we provide in-depth discussion for trace-based debug strategy and review recent advancements in this important area.

I. INTRODUCTION

Due to the high design complexity and the inaccurate abstracted models used in various design and verification phases, today's complex integrated circuits (ICs) usually need to go through one or more re-spins to become bug-free [7, 20], even though half of the system development effort is allocated to verification tasks [16]. Since time-to-market dictates the success of a chip, post-silicon validation techniques that help identifying bugs effectively and efficiently is of crucial importance. Debugging silicon, however, is an extremely complex problem and cannot be tackled without effectively observing the operations of the design's internal nodes.

A widely-adopted post-silicon validation technique utilized by the industry is to reuse the IEEE Std. 1149.1 (JTAG) test access port and existing design for test (DfT) structures in the circuit (e.g., scan chains) to run, halt and step the circuit under debug (CUD) to find bugs [22]. This technique is quite effective in identifying those easy-to-find bugs that leave "evidences" when the circuit halts, but fails to find those tricky bugs that manifest themselves only after a long period of operational time. In addition, the behavior of many bugs is hard to repeat, making diagnosis with this run/stop debug methodology even more difficult. To mitigate the above problem, designers can add shadow flip-flops (FFs) to the CUD to increase its visibility during normal operation [8]. However, this method can only sample a few snapshots of the circuit's operational states and it also involves nontrivial design for debug (DfD) overhead.

To be able to root-cause design bugs, post-silicon validation requires to increase controllability and observability of the CUD's internal behavior to a much higher level than what manufacturing test generally needs [7]. A more effective silicon debug technique is to selectively monitor and trace internal signals of the circuit continuously during its normal operation. The traced data can then be either stored in an on-chip trace buffer or transferred out of the chip via a trace port for later analysis.

A huge volume of trace data, however, is difficult to analyze and results in high DfD overhead. Therefore, how to conduct signal tracing effectively for bug elimination while keeping the associated hardware cost manageable (usually required to be less than 10% of the original design) is a challenging task for IC designers. In this paper, we provide in-depth discussion for trace-based debug strategy and review recent advancements in this important area. In particular, we discuss the following issues in trace-based debug solution:

- Out of the large amount of state elements in the circuit, which signals should we choose to monitor and trace to provide high visibility to the CUD?
- How do we design the trace data transfer module to provide enough observation flexibility while keeping the associated DfD overhead low?
- How can we compress the large volume of trace data effectively so as to make efficient use of the limited trace bandwidth provided by trace buffers and/or trace ports?
- How do we control the signal tracing effectively to obtain highly-qualified trace data?

The remainder of this paper is organized as follows. Section II provides an overview for trace-based silicon debug strategy. In Section III and Section IV, we discuss automated trace signal selection methodologies and interconnection fabric design for trace data transfer, respectively. Section V illustrates trace data compression techniques. The control mechanisms for signal tracing is described in Section VI. Finally, Section VII concludes this paper and points out some future work in this direction.

II. OVERVIEW OF TRACE-BASED DEBUG METHODOLOGY

The hardware infrastructure to facilitate trace-based silicon debug is shown in Fig. 1, wherein various DfD modules are introduced at design stage of the CUD for later debug purpose.

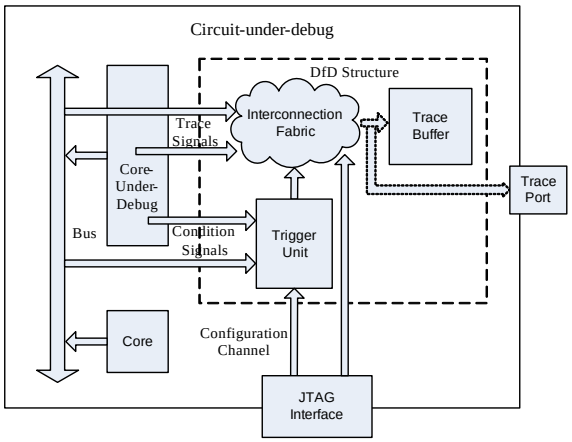


Fig. 1. Trace-Based Debug Infrastructure.

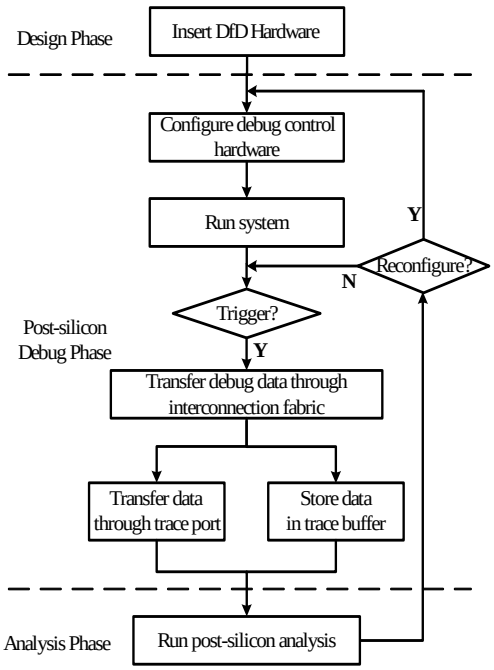


Fig. 2. Trace-Based Silicon Debug Flow.

Generally speaking, designers select to tap a number of signals in the CUD (typically thousands of signals in million-gate industrial designs [1]). However, only a subset of the tapped signals are traced concurrently during debug phase due to trace bandwidth limitation. This is achieved by an "interconnection fabric" (e.g., a MUX tree) that link the tapped signals to trace buffers or trace ports. In addition, trigger units are typically used to determine when to start and stop signal tracing so as to further reduce trace bandwidth requirement. In most cases, designers reuse JTAG test access port as the control interface for the debug phase.

When the first silicon is back with some bugs, in each debug run (see Fig. 2), designers first configure the DfD module in the CUD by selecting the to-be-traced signals from the tapped ones and determining the trigger conditions for signal tracing, and then put the CUD into normal operational mode. If the predetermined trace condition is met, the traced data is transferred through the interconnection fabric to on-chip trace buffers or off-chip trace ports. The collected data are then analyzed to root-cause the possible design bugs. The above process iterates a number of debug rounds or even a few re-spins (when unlucky), until all the bugs are eliminated.

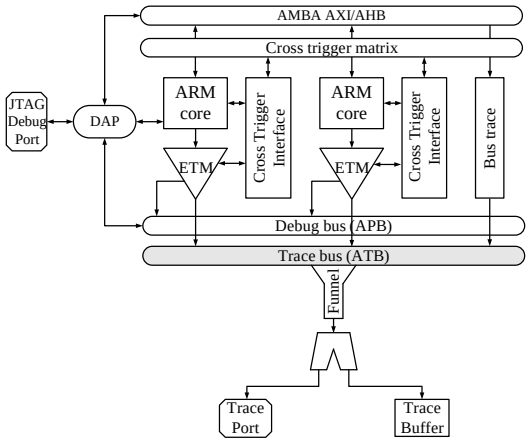


Fig. 3. ARM CoreSight Multi-Core Debug Architecture [3]

Fig. 3 depicts the ARM *CoreSight* trace-based debug solution [3], wherein each ARM core is equipped with an embedded trace macrocell (ETM) for capturing the processor's states. It also contains a cross trigger interface so that trigger events can be transferred between different ETMs to facilitate multi-core debug.

III. Trace Signal Selection

Ideally, we wish to "see any signal at any time" during post-silicon validation. This is clearly not achievable with the large amount of internal signals deeply embedded in the fabricated chip. As indicated in Fig. 1, with the help of constrained DfD resources, we can only afford to tap a few internal state elements and use them to help designers root-cause the abnormal behaviors of the CUD. Our objective is therefore to select those *essential* signals in the CUD so that bugs have a high chance to leave "evidences" on them, and the effectiveness of trace-based debug strategy highly relies on which signals are selected to be traced.

To debug errors on microprocessors and software running on them, naturally it is beneficial to observe the execution of the instructions. In [15], the authors proposed to trace the behavior of every execution stage of instructions to obtain more detailed information on how the microprocessor operates. In addition, several methods have been presented to monitor either the communication interface of the processor (data channel, address channel and control channel) [3, 11, 17], or the memory contents that store the execution results [10, 24].

For the increasingly popular network-on-chip (NoC) based designs, more visibilities are required to the communication among multiple cores, especially at transaction level to provide a globally consistent view of the system. Several trace selection methods were proposed for such type of designs in the literature to tackle this problem [6, 23, 18]. As an example, [6] proposed to attach dedicated monitoring probes on routers and provide transaction level observability of the NoC.

The above techniques are quite effective for the targeted types of circuits, but we are still facing the trace signal selection problem for general logic circuitries. In current practice,

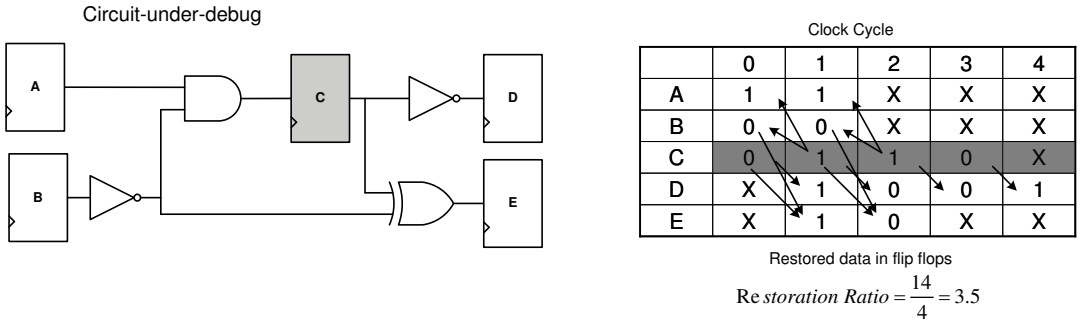
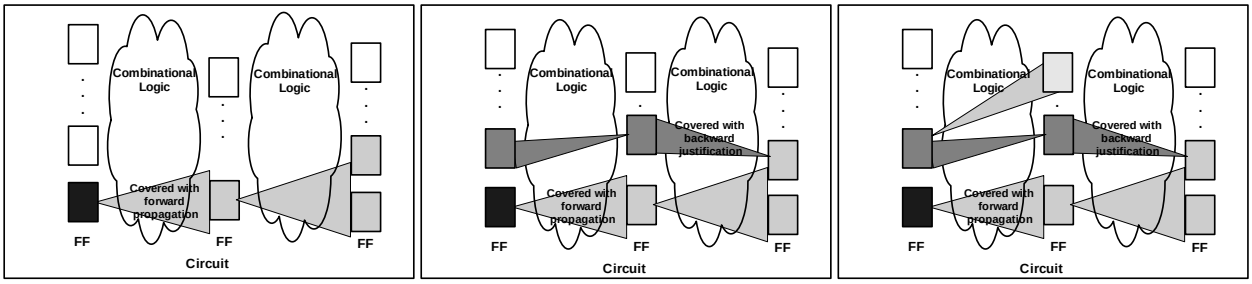


Fig. 4. State Restoration Example in [9].



(a) 1st forward propagation (b) 1st backward justification (c) 2nd forward propagation

Fig. 5. Example of Circuit Level Restoration Calculation.

designers usually manually select those signals that are considered to be vulnerable to bugs or important for analysis to trace, based on their own design experience. This ad-hoc method, however, cannot guarantee the quality of the selected trace signals. More importantly, bugs often occur in unexpected scenarios and it is very difficult, if not impossible, to predict which signals will be related to them during the design phase. Therefore, we need to have at least some trace signals that are selected in an automated manner without designers' intervention.

The work presented in [9] is the first attempt to address the above problem. In this paper, the authors found that it is possible to expand the logic states on the few traced signals to "restore" many missing states on untraced ones with logic implication, as the example shown in Fig. 4. The restoration can be either forwardly propagated (e.g., from *FF_C* to *FF_D*, referred as *forward propagation*) or backwardly justified (e.g., from *FF_C* to *FF_A*, referred as *backward justification*. A so-called *restoration ratio* was defined and calculated as $R = \frac{N_{traced}+N_{restored}}{N_{traced}}$, serving as the evaluation metric to measure the quality of the selected trace signals. N_{traced} and $N_{restored}$ represent the number of traced states and that of restored states, respectively. Since the actual states of the internal signals are unknown before runtime, such restoration ratio can only be estimated at design stage. Therefore, [9] also defined several gate-level *restorabilities*, which are utilized to estimate restoration ratio. With the above, the objective for the automated trace signal selection problem is to maximize the *restoration ratio* for the selected trace signals, which is resolved by a greedy heuristic in [9].

In [14], we showed that the gate-level restorability definitions in [9] are lack of theoretical basis and hence are not accurate in many cases. Consequently, the quality of the selected trace signals guided by such inaccurate estimation is not very effective. To tackle this problem, in [14], we redefined the gate-level restorabilities in a theoretically-precise manner to obtain the *visibility*[1] as the evaluation metric for trace signal selection. Then, we conduct circuit-level propagation of visibilities from traced signals to untraced ones carefully, which tries to avoid both overestimation and underestimation of the restorability by imitating the logic implication behavior as described in Fig. 5. Experimental results show that under constrained random simulation, with the more accurate visibility estimation, our method is able to achieve up to 133.6% higher restoration ratio than [9] for ISCAS'89 benchmark circuits, and the actual restoration ratio varies from 5x to 64x.

Recently, [25] tried to address the trace selection problem from a different angle. In this work, they proposed to select trace signals in such way that the erroneous states on other internal signals will take fewer cycles to expose themselves on the traced ones.

IV. Trace Data Transfer

In this section, we discuss the DfD module that facilitates trace data transfer from the tapped signals to trace buffer/port, including block-level interconnection fabric design for signal tracing and system-level trace data transfer infrastructure for multi-core debug.

[1]The probability that a logic value '1'/'0' is **actually** observed on a circuit node.

A. Interconnection Fabric Design for Signal Tracing

As designers are not knowledgeable about which part of the design may contain bugs, a relatively large number of signals are selected to be traceable in the circuit, typically in the thousand range for million-gate designs [1]. Due to the associated DfD area cost and debug bandwidth requirement, however, it is impossible to *concurrently* monitor and trace all the tapped signals. Instead, only a small number of internal signals can be real-time observed together, and it is up to the designers to determine which signals to trace at a specific debug run, according to the system's erroneous behavior. These signals are then transferred to on-chip trace buffers and/or off-chip trace ports for diagnosis.

To reduce the DfD cost, industrial designs typically use MUX trees to select a subset of the tapped signals to trace in each debug run, in which the control signals to the multiplexers are configured through the JTAG interface (e.g., [1, 21]). To meet timing constraint for the tracing logic, the MUX trees can be pipelined. In addition, when the tapped signals come from multiple clock domains, first-in first-out (FIFO) buffers and/or flip-flop chains can be used to ensure data safety [1]. The above design methodology, however, limits this flexibility of observing any combinations of related tapped signals and reduces the visibility to the CUD, as any signals going through the same multiplexer cannot be traced concurrently. This problem can be easily solved by introducing non-blocking concentration network, which is able to select any m signals out of n inputs ($m \leq n$) and output them to the trace buffers/ports, but such design is with prohibitive DfD cost.

In [13], we proposed a novel interconnection fabric design to satisfy necessary debugging flexibility while keeping the cost at acceptable level. As shown in Fig. 6, our design contains two parts (1). a *multiplexer network* that connects those mutually-exclusive tapped signals, which can be designated by designers and/or extracted automatically based on structural analysis. This stage outputs potentially-correlated signals. (2). a *non-blocking concentration network* that is able to transfer any signals (constrained with bandwidth) out of potentially-correlated inputs to the trace buffers and/or trace ports. One of the main objectives of the design is to minimize the number of potentially-correlated signals (the outputs of MUX network). This is because, accessing the same signals with MUX tree always results in smaller DfD cost when compared to concentrator, since the latter one requires more resources to achieve "any combination" transfer capability. We presented a structural analysis method to identify those mutually-exclusive signals. Then, an "uncorrelation graph" is built as shown in Fig. 7, in which each vertex denotes a tapped signal while an edge denotes that the connected two signals are not highly correlated. Hence, the signals in a *clique* represent they are mutually-exclusive and can be transferred with MUX tree. To minimize the output signals from the *multiplexer network*, it can be simply mapped to the "minimum clique cover problem" and we resort to a classical heuristic to solve it. Starting from existing concentration network, we also proposed several simplification rules to remove redundant hardware that provides unnecessary paths or permutations in the concentrator. Experimental results verify that our solution is able to significantly reduce DfD area cost while satisfying designer's debug flexibility requirement.

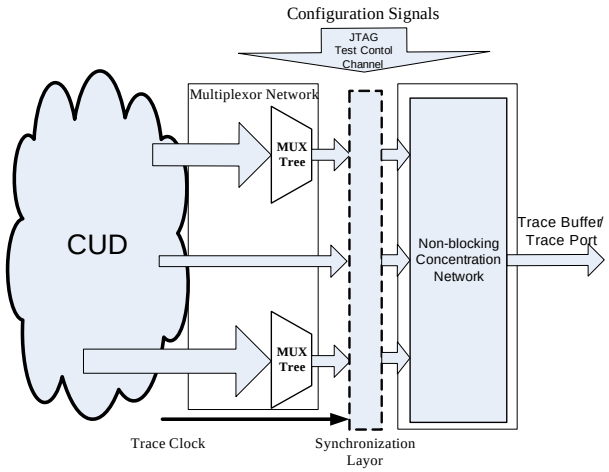


Fig. 6. Proposed Interconnection Fabric Design for Signal Tracing.

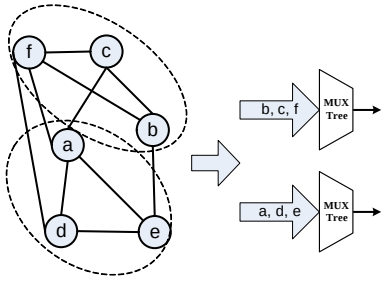


Fig. 7. An Example of Uncorrelation Graph.

B. System-Level Trace Data Transfer

In today's complex system-on-a-chip (SoC), embedded cores communicate with each other during normal operation. Hence, debugging one core at a time can be ineffective and sometimes misleading [17], especially for SoCs containing a number of processors. To tackle this problem, several multi-core debug solutions were proposed for bus-based SoCs by introducing various on-chip instrumentation (OCI) blocks customized for diverse processors, logic cores and embedded buses [3, 11, 17]. Dedicated debug buses are then used for system-level trace data transfer, which incurs non-trivial routing overhead to the CUD.

For NoC-based system, since the communication bandwidth is much higher than that of bus-based system and usually they are not fully occupied, several work advocated to reuse the NoC for trace data transfer so that we can avoid the routing overhead associated with dedicated debug buses (e.g., [6, 18]). At the same time, since a large volume of trace data can significantly affect the performance of the on-chip network, such reuse methods should be designed carefully. In [18], we proposed a novel NoC-based multi-core debug platform as shown in Fig. 8 to address this problem. With a system-level debug agent (DA) and several core-level debug probes between CUD and its network interface, the platform can facilitate designers to synchronize multiple CUD's debug operations and qualify trace data before transferring them through NoC, so that the required NoC traffic cost can be dramatically reduced.

SoC devices often contain dedicated test access mechanisms (TAMs) used to transfer test data between external testers and embedded cores. Since TAMs are left unused after manufactur-

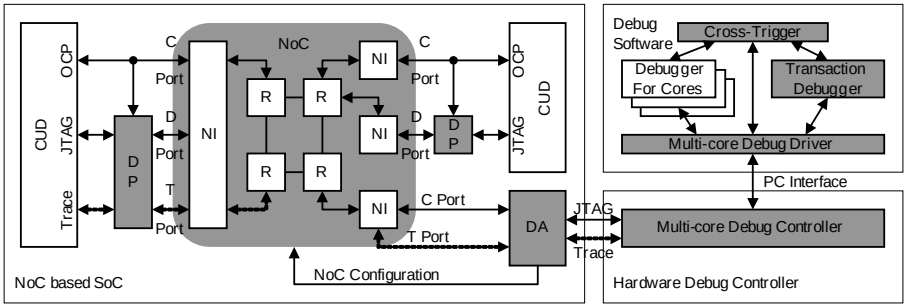


Fig. 8. Proposed Debug Platform for NoC-based Systems

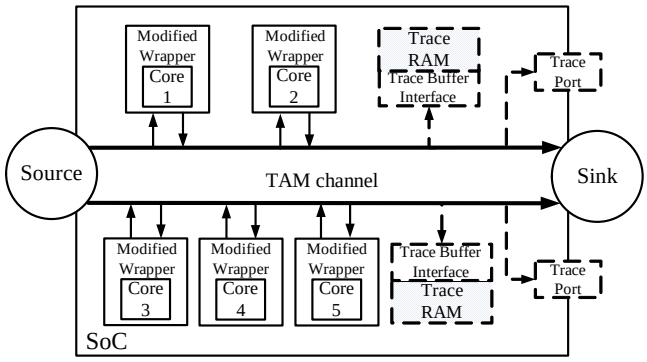


Fig. 9. Proposed Debug Data Transfer Framework with Bus-Based TAMs.

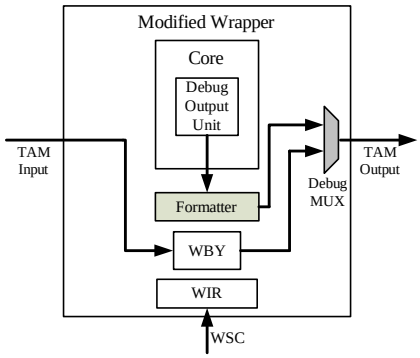


Fig. 10. Modified Testing Wrapper.

ing test, we proposed to use these valuable resources for trace data transfer in normal functional mode in [12], thus significantly reducing routing cost for trace data transfer. To achieve the above objective, several DfD structures are introduced, as shown in Fig. 9. As depicted in Fig. 10, the original test wrappers that enable test data flow from the internal signals of the wrapped core into TAMs are modified to facilitate real-time trace data transfer with the help of a formatter and a debug MUX. We also define the transfer data format and design the corresponding buffer interface to decode it. The method is further developed to avoid data corrupting during multi-core debugging. Experimental results show that the proposed method facilitates trace data transfer in various debug scenarios with negligible DfD cost.

V. Trace Data Compression

Due to the limited trace bandwidth provided by trace buffers and/or trace ports, storing the "raw" traced data is not quite economical. Various trace data compression methodologies were presented to tackle this problem.

In [15], the authors utilized the locality feature of instruction sequence and redundant information in monitored data that can be easily identified with the executed instruction to store the execution states of microprocessor. With the technique, a small amount of *footprints* are enough to observe the whole operational behavior of the microprocessor under debug. Recently, several works [10, 24] were presented for tracing the contents in cache. They both utilize the data locality feature when accessing cache and adopt dictionary-based compression to further improve the compression ratio. Different from each other, [24] observed the following features to enhance compression ratio: the similarity in tag field caused by spacial locality of memory reference and the unusual usage of high order bits for integer value; [10], on the other hand, proposed to reuse cache to compress instructions by inserting supporting module into it. The method is able to restore full information with a small amount of traced data combined with the contents remaining in cache.

Recently, a lossy compression method based on multiple-input signature register (MISR) was presented in [2]. With the assumption that the CUD behaves repeatable in different debug iterations, the method consecutively zooms-in the sampling intervals with compressed failure signatures generated from MISR to localize the error.

VI. Trace-Based Debug Control

This section discusses the control mechanisms that determine the signal tracing behavior. These mechanisms can be used to start and stop tracing so that unnecessary data can be filtered to reduce trace data volume (known as *trace qualification*). More importantly, the designers' capability of controlling the CUD directly affects the debug effectiveness. This is because, when designers can easily control the CUD into suspicious state and obtain relevant information, it becomes much earlier for them to root-cause the possible errors.

In [22], the authors described several basic DfD modules that can be used for debug control, including comparator to check if the condition signals are met with pre-configured value, and counter to facilitate the trigger control with temporal information. Later, [4] proposed to synthesize more complex control

unit (i.e., assertion checker) to monitor complex behaviors of the CUD (e.g., ATB communication protocol). The unit is a state machine that can be generated from the description with formal languages for assertion-based verification. Later, the same authors [5] introduced several enhanced features to localize the errors that are buried as internal states in sophisticated assertions, in addition to reduce the associated hardware cost in unit generation in [4],.

Multi-core debug control for complex SoC devices is a challenging task since we need to be able to control related cores simultaneously [23]. This problem becomes particularly difficult when the data transfer among cores is not deterministic, in which case, it is rather ineffective to configure all the required trigger conditions before running the system. To tackle this problem, [19] proposed a so-called in-band cross-trigger event transmission infrastructure. By inserting the cross-trigger events into the messages, designers are able to trace the desired messages more easily.

VII. Conclusion and Future Work

Trace-based debug techniques have been successfully applied in the industry for some time and they are shown to be quite effective for post-silicon validation. In this paper, we provide in-depth discussion for state-of-the-art signal tracing techniques presented in the literature.

While trace-based debug solution has advanced a lot recently, the overall methodology is still more like an 'art' rather than a 'science'. Whether we can obtain more accurate evaluation metrics to measure the quality of trace-based solutions (e.g., similar to "fault coverage" in manufacturing test) remains to be an open question. In addition, most prior works in this area focused on debugging logic errors that are easy to be repeated. A more challenging problem in post-silicon validation is to debug those electrical errors that only manifest themselves in certain electrical environment, which has not been explored much so far. More innovations are required in the above areas to shorten the time-to-market for the increasingly complex IC designs in the future.

References

[1] M. Abramovici. In-System Silicon Validation and Debug. *IEEE Design & Test of Computers*, 25(3):216–223, May-June 2008.

[2] E. Anis and N. Nicolici. Low cost debug architecture using lossy compression for silicon debug. In *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 225–230, 2007.

[3] ARM Ltd. How CoreSight Technology Gets Higher Performance, More Reliable Product to Market Quicker. http://www.arm.com.

[4] M. Boule, J. Chenard, and Z. Zilic. Assertion Checkers in Verification, Silicon Debug and In-Field Diagnosis. In *Proceedings International Symposium on Quality of Electronic Design (ISQED)*, pages 613–620, 2007.

[5] M. Boule, J. Chenard, and Z. Zilic. Debug enhancements in assertion-checker generation. *IEEE Design & Test of Computers*, 1(6):669–677, December 2007.

[6] R. J. Cloutier, K. Goossens, T. Basten, A. Radulescu, and A. Boon. Transaction monitoring in networks on chip: The on-chip run-time perspective. In *Proc. Symposium on Industrial Embedded Systems*, pages 1–10, 2006.

[7] A. B. T. Hopkins and K. D. McDonald-Maier. Debug Support for Complex Systems on-Chip: A Review. *IEE Proceedings, Computers and Digital Techniques*, 153(4):197–207, July 2006.

[8] D. Josephson and B. Gottlieb. The Crazy Mixed up World of Silicon Debug. In *Proceedings IEEE Custom Integrated Circuits Conference (CICC)*, pages 665– 670, October 2004.

[9] H. F. Ko and N. Nicolici. Automated Trace Signals Identification and State Restoration for Improving Observability in Post-Silicon Validation. In *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 1298–1303, 2008.

[10] C. Lai, F. Yang, C. Kao, and I. Huang. A trace-capable instruction cache for cost efficient real-time program trace compression in SoC. In *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 136 –141, 2009.

[11] R. Leatherman and N. Stollon. An Embedded Debugging Architecture for SOCs. *IEEE Potentials*, 24(1):12–16, Feb-Mar 2005.

[12] X. Liu and Q. Xu. On reusing test access mechanisms for debug data transfer in soc post-silicon validation. In *Proceedings IEEE Asian Test Symposium (ATS)*, pages 303–308, 2008.

[13] X. Liu and Q. Xu. Interconnection fabric design for tracing signals in post-silicon validation. In *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 352–357, 2009.

[14] X. Liu and Q. Xu. Trace signal selection for visibility enhancement in post-silicon validation. In *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 1338–1343, 2009.

[15] S. B. Park and S. Mitra. IFRA: Instruction footprint recording and analysis for post-silicon bug localization in processors. In *Proceedings ACM/IEEE Design Automation Conference (DAC)*, pages 373–378, 2008.

[16] Semiconductor Industry Association (SIA). *The International Technology Roadmap for Semiconductors (ITRS): 2003 Edition*. http://public.itrs.net/Files/2003ITRS/Home2003.htm, 2003.

[17] N. Stollon, R. Leatherman, B. Ableidinger, and E. Edgar. Multi-Core Embedded Debug for Structured ASIC Systems. In *Proc. DesignCon*, 2004.

[18] S. Tang and Q. Xu. A Multi-Core Debug Platform for NoC-Based Systems. In *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 870–875, 2007.

[19] S. Tang and Q. Xu. In-band Cross-trigger Event Transmission for Transaction-based Debug. In *Proceedings Design, Automation, and Test in Europe (DATE)*, pages 414–419, April 2008.

[20] B. Vermeulen and S. K. Goel. Design for Debug: Catching Design Errors in Digital Chips. *IEEE Design & Test of Computers*, 19(3):37–45, May 2002.

[21] B. Vermeulen, S. Oostdijk, and F. Bouwman. Test and Debug Strategy of the PNX8525 Nexperia™ Digital Video Platform System Chip. In *Proceedings IEEE International Test Conference (ITC)*, pages 121–130, Baltimore, MD, Oct. 2001.

[22] B. Vermeulen, T. Waayers, and S. K. Goel. Core-Based Scan Architecture for Silicon Debug. In *Proceedings IEEE International Test Conference (ITC)*, pages 638– 647, October 2002.

[23] B. Vermeulen, T. Waayers, and S. K. Goel. Transaction-Based Communication-Centric Debug. In *ACM/IEEE Int. Symp. on Networks-on-Chip (NOCS)*, May 2007.

[24] A. Vishnoi, P. Panda, and M. Balakrishnan. Cache Aware Compression for Processor Debug Suppport. In *Proceedings Design, Automation, and Test in Europe (DATE)*, 2009.

[25] J. S. Yang and N. A. Touba. Automated Selection of Signals to Observe for Efficient Silicon Debug. In *Proceedings IEEE VLSI Test Symposium (VTS)*, pages 79–84, 2009.