Automating Trace Buffer Post-Silicon Debug

Fredy Alves, Danilo D. Almeida, Vitor Hugo Pereira,
Ana Cláudia Costa and José Augusto M. Nacif
Instituto de Ciências Exatas e Tecnológicas, Campus UFV-Florestal, Universidade Federal de Viçosa, Brazil
{fredy.alves, danilo.damiao, vitor.h.pereira, ana.c.paraiso, jnacif}@ufv.br

***Abstract*—One of the most important stages of an integrated circuit development cycle is the verification phase. Before the integrated digital circuit large scale production, the pre-silicon verification captures and fixes the project functional errors using simulation and formal techniques. Unfortunately, some design errors escape to first silicon and need to be identified and corrected. This new phase is called post-silicon verification and imposes signal observability restrictions preventing designer from capturing and evaluating the signal values against time. We propose PSi, the first open-source framework for post-silicon debug algorithm experimentation. PSi features include: Verilog RTL parsing, post-silicon debug infrastructure inclusion, and online monitoring and validation. The customizable post-silicon debug infrastructure is composed by interconnection network, trace buffer, assertion triggering, and signal selection. These modules can be extended in order to evaluate new verification algorithms and architectures. We present area overhead of circuits in which we have used PSi to generate post-silicon debug infrastructure.**

I. Introduction

Technological innovations in the manufacturing process of digital integrated circuits result in transistor size reduction allowing the inclusion of new features. At the same time, the effort required to check and correct design errors is increased due to the higher complexity of modern chips [15]. This stage is called Verification being one of the most challenging during integrated circuit development cycle. When a design error escapes to silicon, the project is delayed and losses can reach up to 93% [16].

Before the circuit manufacturing process, a set of methods is used to capture and correct design errors. This pre-silicon step consists of simulations, formal tests and emulations. The most usual technique is the simulation [14], which allows the analysis of circuit signal behavior over time. However, this option may be impractical for all states due to the exponential number of input stimulus generated in some situations [12], [17]. In formal verification, mathematical proofs are used [8] but, unfortunately, this technique is also impractical to be used in complex systems which may contain billions of transistors, such as the third generation of Intel processors Core.

In order to identify the root cause of design errors that escaped to silicon, we use post-silicon debug techniques. In this process, the operation of the integrated circuit is evaluated using applications and real environments running at full clock frequency. This process involves four steps [1]: a) detecting the problem; b) reducing its location to a smaller region; c) identifying the cause; d) fixing or ignoring it [10]. This procedure has become essential and consumes on average, 35% of the circuit development cycle. In this case, the limited observability is a challenging problem, preventing the designer to capture and analyze the values of all circuit signals. In order to overcome this limitation, post-silicon debug techniques are used to make possible the observation of Circuit Under Debug (CUD) internal signals behaviour. Post-silicon debug techniques include scan chains and trace buffers [9].

Scan chains use a series of interconnected registers, flip-flops or sequential elements. In this technique signal values are serially extracted, and then the circuit error state is uncovered and analyzed. However, the integrated circuit must be halted, preventing signal continuous monitoring. In industry, the most popular post-silicon debug technique is based on a trace buffer to store internal signal values during the integrated circuit real-time execution.

Although post-silicon debug techniques are broadly used both in academia and industry, there is no open-source system available to automate and implement post-silicon new algorithms and architectures. Developing open-source systems is a widely disseminated habit in Electronic Design Automation (EDA) research community. Some examples of EDA open-source systems are SIS (logic synthesis) [13], VIS (formal verification) [4], and ABC [3] (sequential synthesis and verification).

In the paper we present PSi, the first open-source framework for post-silicon debug algorithm experimentation. The framework is composed by basic features such as Verilog RTL parsing, post-silicon infrastructure inclusion, and online monitoring and validation. In terms of automatic post-silicon infrastructure generation PSi offers a set of pre-defined algorithms and architectures that are easily extensible. These pre-defined architectures and algorithms include MuxTree/Omega interconnection network generation, arbitrary size trace buffer generation, assertion triggering, and manual signal selection.

This paper is outlined as follows. Section II presents general concepts related to trace buffer post-silicon debug. Section III discusses related work. In Section IV we present the methodology and, finally, Section V shows the results and Section VI, conclusion and future work.

II. Background

The most relevant post-silicon verification problem is internal signal observability limitation. Some on-chip strategies to overcome observability limitation include Design-for-Test (DFT) and Design-for-Debug (DFD). Although adressing the same problem, DFT and DFD have different objectives. While

DFT identifies manufacturing defects, DFD is focused on design error localization. The most popular technique for DFT is the scan chain [7]. The industry standard solution for DFD is storing and analyzing the signal values on a trace buffer.

In this paper we consider DFD as the infrastructure used for extracting and analyzing signal information for post-silicon debug being responsible for capturing and storing groups of signals in real-time at full clock speed. In remain of this Section we discuss general concepts related to DFD. Figure 1 presents a trace buffer based DFD architecture. The Interconnection Network module is responsible for selecting which signals will be stored in the trace buffer memory module. The Trigger Logic module monitors the circuit behavior and begins capturing data when preset conditions are met. We discuss those DFD modules in more detail in the next subsections.

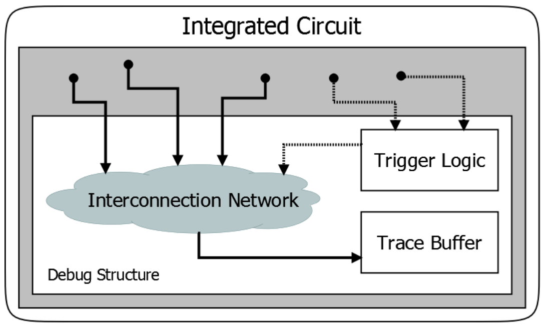

Fig. 1: Internal DFD Structure.

A. *Interconnection Network*

Interconnection networks are programmable systems used for communication between two components. In post-silicon debug, interconnection networks are used to select a subset of signals to be stored in the trace buffer being named asymmetric because they have more inputs than outputs. Interconnection networks and can be designed under diverse architecture types and the most relevant characteristics for post-silicon debug are blocking rate and area overhead. The blocking rate refers to the set of possible signal combinations that can be observed at the same time and area overhead is related to the integrated circuit area dedicated to implement the interconnection network.

III. Related Work

In this Section we present work related to post-silicon debug architectures, algorithms, and tools. Several post-silicon debug architectures have been proposed in recent years. Abramovici *et al.* [1] presents a reconfigurable infrastructure for SoCs to support post-silicon debug. This infrastructure is inserted at RTL providing a debug platform that can be configured using a JTAG port. The proposed system supports different debug structures such as assertion checkers, transaction identifiers, triggers, and event counters. This system is not publicly available.

Park *et al.* [11] use the trace buffer for bug localization. In this work on-chip recorders collect information about flows of instructions, and what the instructions executed as they passed through various design blocks. When a system failure occurs, the recorded information is scanned out and analyzed offline for bug localization. On [18], the authors use the trace buffer data for capturing errors in spatial and temporal domains. But even using trace buffer, the time of bug activation is a challenging problem. Thus, various methods for bugs and faults have been presented.

IV. Methodology

PSi is a system developed to help and speed up the post-silicon debug phase. The tool provides to the user the possibility of selecting wires from a design, in order to externalize them and generate the interconnection network used to capture the signals driven by these wires, and selecting assertions, to use for trace buffer trigger. PSi uses the tools Vericonn [6] and Veritrace [2] to generate the DFD. It uses Yosys in order to generate a list of wires in a Verilog design and their respective place on the project hierarchy and uses an assertion library to include the selected assertions on the design. In this Section we present PSi blocks: a) Front-end; b) Externalization process; c) Debug Infrastructure.

A. *Front-end*

The front-end consists of a visual interface used by the designer in order to specify the design top level file, select the desired wires for analysis and configure DFD module. The configuration consists on the type and features (such as input and output size) of the interconnection network as well as the trace buffer dimensions and the assertions to be used as trigger of trace buffer. After configured, PSi starts the process of wire parsing, creating a new Verilog design with the assertions and all selected wires externalized in the top module, and with the DFD connected to the wires. Figure 2 shows the new design generated by PSi. With the new Verilog design, is possible to analyze the system behavior during execution through the serial interface. Figure 2 depicts the process of communication between the design generated by the PSi and the computer.

B. *Wire Extraction and DFD infrastructure generation example*

In order to extract and externalize the wires that compose a design we use a Verilog lexical analyzer from the Yosys synthesis tool. This process consists of five steps:

1) Yosys performs the Verilog code analysis and returns a list of wires with their respective attributes and where they are on the design hierarchy;
2) The tool generates the design modules hierarchy with their respective wires;
3) After wire selection, the tool performs a recursive modification of the modules interfaces on the hierarchy in order to create a path to the top module interface;
4) A new design containing all modules necessary for the circuit debug is generated, including assertions.
5) A XML file containing all information about the trace buffer structure is generated for the data extraction process.

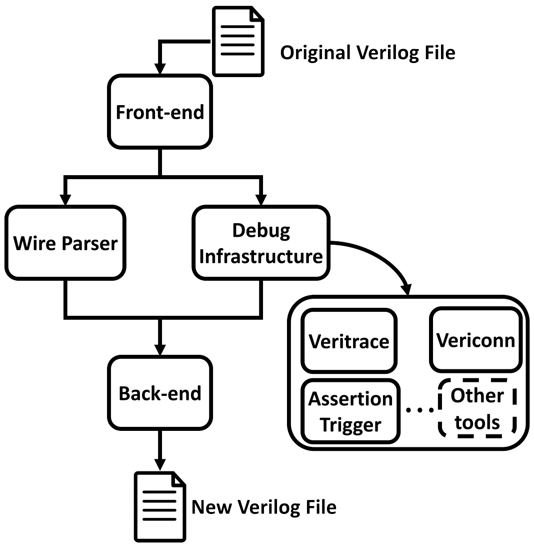

Fig. 2: PSi Execution Flow.

In order to externalize wires in deep modules, PSi creates a path between the selected wire module and the top module by modifying the intermediate modules interfaces. This process starts in the module that is deeper in the hierarchy. PSi generates an output for this wire connecting the selected wire to this output, creating an new interface in the module. This process is repeated in the upper modules finishing at the top level module, when the internal wires can be accessed by the DFD module.

C. Debug infrastructure

The debug structure is responsible to detect and store the errors occurrence during the post-silicon debug. This module consists in three parts:

1) Interconnection Network;
2) Trigger Logic;
3) Trace Buffer.

The interconnection network is responsible for selecting a subgroup of wires from a larger set and connect them to a trace buffer. The use of interconnection networks is necessary in order to maximize the number of captured signals from a design. The main reason for the use of these networks is the fact that the trace buffer width will always be smaller than the number of observed signals due to the area overhead caused by the trace buffer.

The trigger logic is responsible for detecting errors during debug process. This module uses assertions to monitor circuit behavior. All selected assertions are connected on the trigger logic. When an assertion is violated, the process of capturing signals from interconnection network and storing at the trace buffer starts. Figure 3 illustrates this process.

The trace buffer is a circular buffer memory responsible for storing all errors occurred during the debug process.

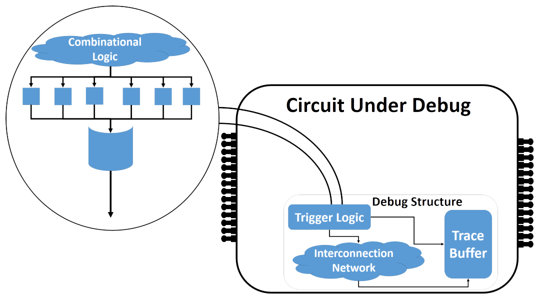

Fig. 3: Assertion Trigger Logic.

V. Results

To study the efficiency of the tool we generate a DFD to a simplified MIPS processor. MIPS (Microprocessor without Interlocked Pipeline Stages) is a RISC (Reduced Instruction Set Computer) architecture that only do arithmetic and logic operations between registers, requiring load/store instructions to access data from memory. The version used allows logic arithmetic, branch and load/store instructions.

To trigger logic we have selected 3 assertions, presented in Table I:

1) Assertion 0 tracks if the control signal `memRead` is different from signal `memWrite`, in other words there is not memory read and memory write at the same time;
2) Assertion 1 tracks if the part of instruction corresponding to the operator is 0 (R-Type instruction) and the destination register is from the field `rd` from instruction i.e. the control signal is generated correctly;
3) Assertion 3 tracks if one of the inputs of `ALU`, or both, are greater than 0. The operator is `ADD` ($ALUop == 2$) and the result of ALU is 0, testing if an error occured in an add operation.

When one, or more, assertions fire, the trace buffer will start storing data from the interconnection network.

There are three principal types of design bugs: logic, algorithm and synchronization bugs [5]. Logic bugs consists on incorrect logic in combinational circuits. Algorithm bugs occur when an algorithm was not correctly implemented in the design. Synchronization bugs are errors in the timing behavior of a design. Using the first or the second assertion of the Table I we can trigger a logic bug on signal control generation. With the last assertion we have an arithmetic bug detection.

TABLE I: Assertions used in the design example.

ID	Expression
0	assert_always(memRead != memWrite)
1	assert_always(instruction[31:26] == 0 && regDst)
2	assert_always((op1ALU>0 || op2ALU>0) && ALUop==2 && !zero)

In the Figure 4 we present the area of the new design for two different interconnection networks (Mux Tree and Omega) in three different sizes (32x8, 128x16 and 256x32) with a trace buffer of fixed depth of 32 and width of 8, 16 and 32 respectively. The area is presented as the number of NAND equivalent gates.

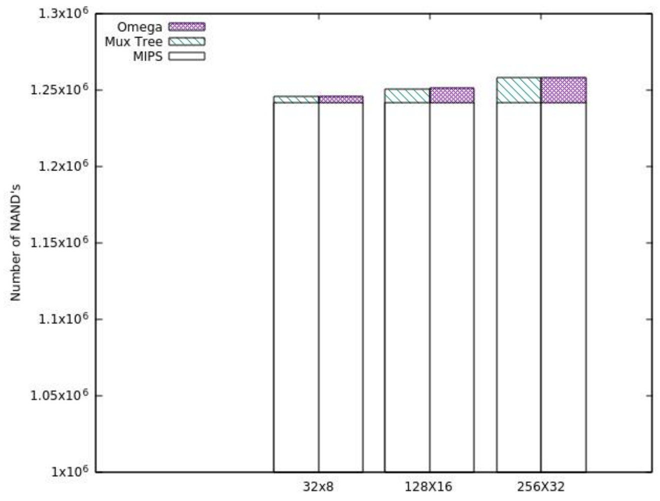

Fig. 4: Final Overhead caused by DfD.

The y-axis of Figure 4 represents the number of NAND equivalent gates. The white bar represents the size of the MIPS processor and the bars with patterns represent the size including the DFD infrastructure.

Table II presents the new size of the MIPS processor with the DFD and the respective overhead. All modules generated presents overhead below 2%.

TABLE II: Area Overhead

Network	Trace	New Size	Overhead
Mux_Tree-32x8	8x32	1245427	0.34%
Mux_Tree-128x16	16x32	1250304	0.73%
Mux_Tree-256x32	32x32	1257885	1.34%
Omega-32x8	8x32	1245691	0.36%
Omega-128x16	16x32	1251124	0.79%
Omega-256x32	32x32	1258167	1.36%

VI. Conclusion

We have proposed the PSi post-silicon debug tool. It provides the features to select a group of wires from a design and connect them to a DFD module. We have also shown that PSi is extensible and new funtions can be easily included. The already implemented features are the generation of interconnection network and trace buffers in Verilog RTL code, assertion based event trigger, and manual signal selection.

Acknowledgments

We would like to thank CAPES, CNPq, and FAPEMIG for the financial support.

References

[1] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A reconfigurable design-for-debug infrastructure for socs. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 7–12, 2006.

[2] D. Almeida, F. A. M. Alves, and J. A. M. Nacif. Veritrace: A tool to generate trace buffers for post-silicon debug. In *2015 Student Forum (SFORUM)*, 2015.

[3] Robert Brayton and Alan Mishchenko. *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, chapter ABC: An Academic Industrial-Strength Verification Tool, pages 24–40. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[4] Robert K. Brayton, Gary D. Hachtel, Alberto Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Staple, Gitanjali Swamy, and Tiziano Villa. *Computer Aided Verification: 8th International Conference, CAV '96 New Brunswick, NJ, USA, July 31– August 3, 1996 Proceedings*, chapter VIS: A system for verification and synthesis, pages 428–432. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[5] Kypros Constantinides, Onur Mutlu, and Todd Austin. Online design bug detection: Rtl analysis, flexible mechanisms, and evaluation. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 282–293. IEEE Computer Society, 2008.

[6] A. B. M. Gomes, F. A. M. Alves, R. S. Ferreira, and J. A. M. Nacif. Vericonn: a tool to generate efficient interconnection networks for post-silicon debug. In *2015 16th Latin-American Test Symposium (LATS)*, pages 1–6, March 2015.

[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] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: A survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4(2):123–193, April 1999.

[9] H. F. Ko and N. Nicolici. Combining scan and trace buffers for enhancing real-time observability in post-silicon debugging. In *2010 15th IEEE European Test Symposium*, pages 62–67, May 2010.

[10] S. Mitra, S. A. Seshia, and N. Nicolici. Post-silicon validation opportunities, challenges and recent advances. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 12–17, June 2010.

[11] S. B. Park, T. Hong, and S. Mitra. Post-silicon bug localization in processors using instruction footprint recording and analysis (ifra). *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1545–1558, Oct 2009.

[12] G. J. Van Rootselaar and B. Vermeulen. Silicon debug: scan chains alone are not enough. In *Test Conference, 1999. Proceedings. International*, pages 892–902, 1999.

[13] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Computer Design: VLSI in Computers and Processors, 1992. ICCD '92. Proceedings, IEEE 1992 International Conference on*, pages 328–333, Oct 1992.

[14] M. Talupur. Hardware model checking: Status, challenges, and opportunities. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 154–154, Oct 2011.

[15] B. Vermeulen and S. K. Goel. Design for debug: catching design errors in digital chips. *IEEE Design Test of Computers*, 19(3):35–43, May 2002.

[16] Bart Vermeulen and Kees Goossens. *Debugging Systems-on-Chip: Communication-centric and Abstraction-based Techniques*. Elsevier, 2014.

[17] S. Yang, R. Wille, and R. Drechsler. Determining cases of scenarios to improve coverage in simulation-based verification. In *Integrated Circuits and Systems Design (SBCCI), 2014 27th Symposium on*, pages 1–7, Sept 2014.

[18] Y. S. Yang, N. Nicolici, and A. Veneris. Automated data analysis solutions to silicon debug. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 982–987, April 2009.