On Post-silicon Root-Cause Analysis and Debug Using Enhanced Hierarchical Triggers

M. H. Neishaburi, Zeljko Zilic
McGill University, 3480 University Street, Montreal, Quebec Canada H3A-2A7
Mh.neishabouri@mail.mcgill.ca, zeljko.zilic@mcgill.ca

Abstract:
Post-silicon debugging process is aimed at locating errors that concealed themselves during the process of pre-silicon verification. Although in the post-silicon validation engineers can exploit the high speed of hardware prototype to exercise huge amount of test vectors, low level of real-time observability and controllability of signals inside the prototype is too big an issue. Various Design for Debug (DFD) techniques aim to improve the observability of signals and expedite the root cause analysis of errors. Typical practical DFD approaches are based on the Embedded Logic Analysis (ELA), using a trigger unit that can effectively control when to acquire the debug data. In this paper, we propose ZiMH a hierarchical trigger generator that builds a trigger unit. Additionally, it provides resourceful and compact trace information for root cause analysis. Major advantages over traditional trigger units are: 1) by keeping the trace of interactions that leads to the failure, it facilitates the process of failure localization and root-cause analysis 2) it can be tuned for the specific location of a design to avoid the huge cost related to interfacing with trace signals 3) it can get parameterized to generate several units that can be placed inside the limited area in multiple debug rounds using a time-multiplex fashion.

Keywords
Post-silicon Debugging, Assertion checker

1. Introduction

Due to the rapid increases in the design complexity of modern Microprocessors and SoCs, existing pre-silicon verification techniques can no longer satisfy the unquenchable demands of customers for faster, cheaper and more reliable hardware devices that materialize many of features. Verification process is nowadays divided into two separate tasks: pre-silicon verification and post-silicon validation. Although comprehensive pre-silicon verification gets applied to a hardware model prior to the silicon fabrication, the first prototypes of a system with ever increasing complexity are increasingly not fully functional.

Failures in first hardware prototype mostly emanate from Design errors (Errata) and Electrical errors [1][2]. Design errors are associated with designer's mistake in interpreting or implementing high level expected behaviors of a design to RTL. Electrical errors, however, are partially related to the transient errors inside storage elements of a system [1].

Several factors, including crosstalk, low voltage levels, high frequency and small noise margins contribute to the increases in electrical bugs. An electrical error is a hard-to-catch error during pre-silicon verification [20]. One task of a verification team is to ensure that a product can be released to the market on time without any unexpected errors.

Pre-silicon verification techniques are formal or dynamic (simulation-based). Formal methods suffer from scalability limitations, and are not feasible for the full-chip verification of complex SoC designs. However, in practice, few parts of a design such as the floating-point units of processors are verified using formal methods [2].

Functional verification techniques are not only subject to missing corner cases of a complex design, but also they have to keep up with the limited speed of software-oriented RTL simulators. Therefore, the demanding guarantees that the first-silicon is fault free and works perfectly is not achievable using pre-silicon methods.

Post-silicon validation promises to complement the task of pre-silicon verification teams. Once an SoC design passes all the checks within pre-silicon verification, post-silicon validation begins its mission on a fabricated prototype of a system. Because post-silicon validation caries out on the actual hardware, larger number of functional tests can be applied at speed. Moreover, hard-to-reach states and corner cases would more likely be exercised and thus there will be a better chance to catch hard-to-detect bugs. In general, post-silicon validation involves four steps: failure detection, failure localization, root cause analysis and fixing or bypassing the problem by patching [20].

Directed or random generated test vectors are applied to a hardware prototype during the failure detection phase. Specifically, during failure detection of processors, prototype validation engineers usually boot up a different Operating System (OS) and try to experience various OS features [7]. However, once a failure is observed, the process of localizing the problem to a small region and then identifying the root cause of the failure is time-consuming, easily accounting for about 35% of the chip development cycle [1]. In fact, although the post-silicon validation techniques offer raw performance in terms of the execution speed of test cases, they need to get improved when it comes to real-time observability. To facilitate failure detection and root cause analysis of a hardware prototype, various Design-for-Debug (DFD) techniques have emerged [1][6][18].

Embedded Logic Analysis (ELA) that adopts a trigger unit and trace buffers to capture debug data in real-time has been considered in several studies [5][6]. However, ELA has limitation in terms of the amount of data that can be acquired in a debug experiment. Since trace memory itself is subject to electrical bugs, we need to deal with this resource carefully. In this paper, we propose ZiMH, a trigger generator that builds an RTL model of the trigger unit. The generated circuit provides compact trace information for root cause analysis and error localization. Plus, it has fine control over the signals to capture. To the best of our knowledge, this work is the first study that exploits the hierarchical properties of the system to generate a trigger unit.

The remainder of this paper is organized as follow. Section 2 reviews the related work in this domain. Some terminologies and basics will be introduced in section 3. Section 4 explains our proposed method for hierarchical trigger generation. The experimental results will be shown in Section 5. Finally, Section 6 concludes the paper.

2. Prior Work and Motivation

Achieving real-time observability of internal signals during post-silicon validation is difficult. DFD techniques come to address this problem. One of the traditional DFD techniques is the scan-based debug [19]. The primary goal in scan-based debug is to reuse resources that were used for the manufacturing test. In general, once a specific trigger or hardware checker fires the internal state elements of a system using the available scan chains are captured in parallel. Afterwards, the captured data will be offloaded serially using scan-out operation. Finally, post-processing algorithms analysis offloaded data. Due to consecutive stops and resumption during scan dump, there is a need to investigate better DFD approaches [3]. Integration of assertion checkers is appealing in scan-based run-stop debug infrastructure. The authors in [10] investigate a method for clustering assertion checkers inside the design; however, they have not integrated these clusters of checkers inside a trigger unit. Our trigger generator can be used to solve the issues related to integration of assertion checkers inside trigger units.

One of the newer DFD methods is embedded logic analysis (ELA), which utilizes on-chip trace buffers and trigger unit to capture debug data in real-time[5]. The amount of data that one can acquire in a debug experiment is limited. As a result, a number of solutions have been investigated to use the on-chip buffers more efficiently. For example, trace compression techniques reduce the amount of data stored for each sample [5]. Another approach to better utilize the on-chip trace buffers is to have a fine control over when to acquire the debug data. In [6] authors suggested an ELA with a programmable trigger unit. However, their approach has limitation in detection of complex sequences; plus, their proposed trigger unit provides no accurate details to facilitate root cause analysis. Since our proposed trigger unit keeps track of the sequences that lead to activation of a trigger, as we will show later on the root-cause analysis will be expedited.

In [15] authors presented a Time-Multiplexed Assertion Checking (TMAC) scheme for post-silicon bug detection. In their method, assertion checkers are instantiated in an on-chip reconfigurable block in a time-multiplex fashion and post-silicon debugging are carried out by only utilizing assertion checkers. However, in some cases the root cause analysis is not achievable unless we have access to internal signals of a design. ELA utilizes a trigger unit to capture internal signal inside the design. Here, the proposed tool generates a hierarchical trigger unit that can be used inside an ELA. ELA can exploit the hierarchical trace information generated by the proposed trigger unit to facilitate root cause analysis.

3. Preliminaries

3.1 Assertions

Assertion is a statement that indicates how a given circuit should behave under different circumstances. Assertion-Based Verification (ABV) is one of the most important and efficient RTL verification techniques for pre-silicon verification. Assertions represent a complex range of behaviors. System designers are able to define both expected and unexpected behaviors of a design using the temporal logic and the extended regular expressions that assertion languages such as PSL (Property Specification Language, IEEE 1850 standard) and the SVA (System Verilog Assertions). For example the PSL assertion below specifies that once the request signal goes high, the arbiter is expected to grant the bus to the client within three clock cycles. The client must also keep its request signal active until it receives control of the bus which is indicated by the 'grant' signal. This assertion will fire if any of these conditions not happen.

assert always ({$rose(req)} |=>{req[*0:3] ; req&grant});

The |=> operator is a temporal implication, with pre and post conditions appearing as left and right arguments respectively. rose(b) is an operator that evaluates to true in case of any changes from false to true. In this example, the post-condition is a regular expression consisting of a temporal concatenation ";" of two sub-expressions, the left of which contains a repetition range and the right expression is a Boolean expression.

3.2 Checker generator

Checker generator is in charge of producing assertion checkers. An assertion checker is synthesized from assertion(s). Here, we use MBAC for checker generation [8][11]. At the first step, a checker generator tries to match each assertion statement with its related automaton. Thereby, various automata for properties and sequences are generated using a checker generator. A generated automaton is a directed graph, where vertices are states, and edges among states shows the conditions for transitions among them. Fig. 1 shows the generated automata from the previous mentioned PSL assertion. Transitions are labeled with the Boolean expressions built over combination of signals involved in the property. It has been shown in [12] that every property in PSL and SVA can be converted to an equivalent finite automaton in a recursive manner [5]. Assertion violation has been activated whenever an automaton representing an assertion reaches its final state (S5).

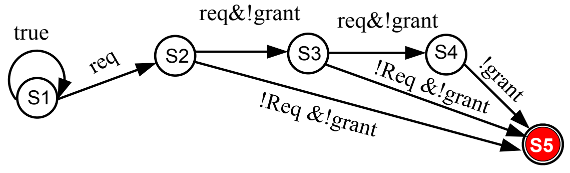

Figure 1. Generated automata from the PSL statement

3.3 Parallel Hierarchical Finite State Machine (PHFSM)

Hierarchical Graph Scheme (HGS) associated with HFSM[16] has been used to describe the behavior of digital control units in [12]. The automatic synthesis algorithm from an HGS to HFSM also has been established in this study [16]. To represent existing parallelisms and dependencies among assertion checkers, we use particular version of HGS, Parallel Hierarchical Graph Scheme (PHGS)[17]. In fact, to pinpoint the cause of an error, a mechanism to quickly determine which signals and parameters influence the assertion output is an important aid. As we show later on PHGS will be utilized to expedite root-cause analysis.

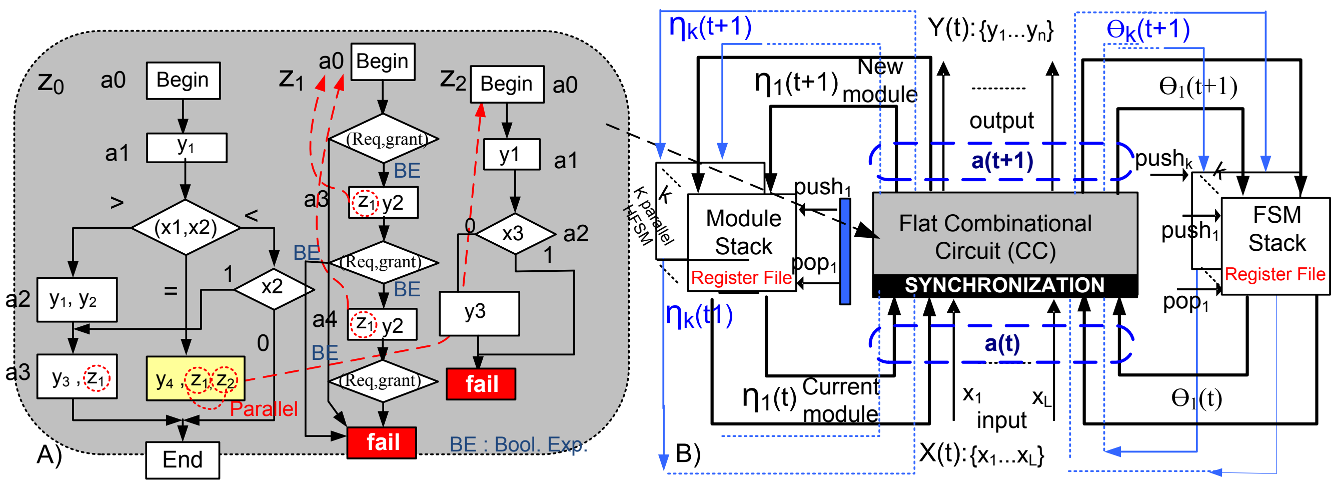

Figure 2. PHGS and its implementation in PHFMS system

Each macro operation is described by another PHGS of a lower level. For instance, z_0, z_1 and z_2 represent another PHGS graph in Fig. 2 (A).

The proposed trigger generator maps the automaton of each assertion checker to one of the macro-operations (z_i). For example, as Figure 2 illustrates, the automaton of the previously mentioned assertion is mapped to the z_1. Rhomboidal node contains one element from the set X, where X = $\{x_1,\ldots x_n\}$ is the set of logic conditions that represents the a trace signals. Inside PHGS multiple Graph Scheme (GS) can be active at the same time plus hierarchical calls are allowed inside a GS. As shown in Figure 2, the level of parallelism in PHFSM is defined by k, and PHFSM incorporates two stacks (Module-Stack and FSM-Stack) for each level.

PHFSM is defined as a six-tuple $S=(A,X,Y,\Psi,\Phi,a_0)$, where $A=\{a_0,a_1,\ldots a_M\}$ is a finite set of states, $a_0\in A$ is an initial state, $X=\{X_1,X_2,\ldots X_n\}$ is a finite set of input vectors, where $X_i=\{x_1,x_2,\ldots x_L\}$, $x_i\in\{1,0,-\}$, $Y=\{Y_1,Y_2,\ldots\}$ is a finite set of output vectors, $Y_i=\{y_1,y_2,\ldots y_n\}$, $y_i\in\{1,0,-\}$, the transition function Ψ which is defined as $\Psi: A\times X\leftrightarrow A$ that maps $A\times X$ to a subset of A. Based on this function, the next state $a(t+1)\in A$ depends on the current state $a(t)\in A$, and the input vector $X(t)\in X$. $a(t+1)=\Psi(a(t),X(t))$. The output function Φ defines output vector $Y(t)$ from the set Y.

An Item from Module-stack and FSM-stack is subset of Π and Θ respectively, where $\Theta=\{\Theta_1,\Theta_2,\ldots\Theta_k\}$ and $\Pi=\{\Pi_1,\Pi_2,\ldots\Pi_k\}$ and k is equal to the maximum parallel modules. We could state that $a(t+1)=\Psi(a(t),X(t),\eta(t),\Theta(t))$; however, as it was shown in Fig. 2 (B), $\eta(t)$ and $\Theta(t)$ are calculated from the a(t). Therefore, $a(t+1)=\Psi(a(t),X(t),\eta(t),\Theta(t))$ is equivalent to $a(t+1)=\Psi(a(t),X(t))$.

It has been proven in [17] that every control algorithm defined by PHGS can be synthesized to the PHFSM which behaves in accordance with the given description. There are two ways of implementing PHFSMs[16] [17]:

1) Flat combination-circuit such as the circuit in Fig. 2 (B) where all sub-algorithms are implemented inside one combinatorial block

2) Bounded combinational circuit where synthesis for each sub-module is performed independently. As a result, the combinatorial unit will be divided into autonomous segments in such a way that each segment implements only one sub-algorithm. Although first approach is easier to realize, we are subject to losing the modularity of the specification during a circuit implementation. In the second approach, sufficient hierarchy and modularity are maintained during the implementation. However, additional components are required to select and activate the correct combinatorial sub-circuit. Inherent characteristic of PHFSM that we exploit later on are:

1) Ability to trace back system: in order to trace the history of events and transitions that took place in a system we can investigate current state of PHFSM as well as its k level FSM-Stack and Module-Stack.

2) Recursive calls: PHFSM allows recursive calls. Therefore, we can recognize overlapped sequences and their root-causes in checkers automata [11][8].

4. Proposed Method

In our proposed method, using MBAC checker generator[12], we generate synthesizable FSM automata for the checkers that we want to embed in the specific location of our design. Thereafter, we feed these automata to our proposed trigger generator ZiMH.

ZiMH then, based on the maximum allowable parallelism inside each checker generates a set of synthesizable PHFMS for the trigger unit. All the k-level FSM-Stack and Module Stack of all these PHFMS are chain together.

4.1 Post-Silicon Trigger Generator

Fig. 3 shows the required steps to build our hierarchical trigger unit as a core of ELA. First, for each SVA or PSL assertion statement, MBAC tool runs in generator mode to create Finite Automata (FA). The generated FAs can be tuned to either detect failure or acceptance. In failure mode the generated FAs can discover the sequences of input signals leading to assertion failures, while in acceptance mode FAs can find the sequences that cause an assertion get complete. In our case, we select the failure mode detection of MBAC.

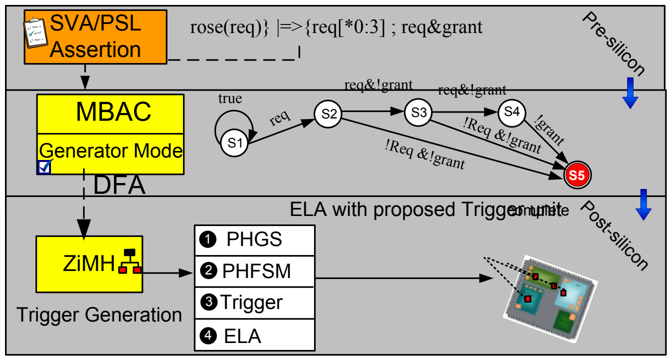

Figure 3. The proposed trigger generator

Our proposed post-silicon trigger generator, ZiMH, takes the set of failure detection FAs which have been generated by MBAC. The first task of ZiMH is to generate PHGS from these FAs. To reach this goal, ZiMH first tries to discover the common failure patterns among these FAs. For example, assuming that we have a bus that supports 4 masters and salves, properties that control request-acknowledge sequences and handshaking among masters and slaves mostly have a lot of common failure patterns. Having discovered these common failure patterns, ZiMH starts generation of PHGS from a set of automata.

During PHGS generation, ZiMH actually prepares an intermediate representation of a group of automat that will be converted to the PHFSM-based trigger; therefore, ZiMH should also be aware of the locality of the checkers automata that can be clustered. Due to the overhead related to the interfacing of trace signals to the central trigger unit, it is not always feasible to place all FAs inside one trigger unit. For example, assuming again bus example, a local trigger unit that monitors master transactions has lower overhead, in terms of wiring and interfacing, than one central ELA that monitors all transactions inside a Bus.

Two parameters that ZiMH should consider during the process of PHGS generation are the maximum allowable parallelism (p) for each checker and maximum level of permissible hierarchy (h). Architectural limitation of the final PHFSM controller dictates higher limits of these two parameters.

As Figure 2 (B) shows, 'k' represents the numbers of FSM-Stacks inside PHFSM. However, if PHGS cannot fit into k levels of parallelism, ZiMH will increase the level of pipelining to deal with the speed issues.

On the other hand, stack size represents the depth of hierarchy that PHFSM can support. For example, by using a register file that has 32 registers for FSM-Stack, we can store 32 levels of hierarchies.

In the worst-case condition, if ZiMH cannot restructure PHGS to need at most 32 function calls. Content of filled FSM-Stack and its related module-stack will be stored inside the trace memory in a specific segment. Consequently, when stack pointer of the FSM-Stack points to the location inside the trace buffer, trigger unit will be updated by previously stored data.

In the last step, before generating PHFSM and its related trigger unit, PHGS is annotated with failure information. In other words, each micro operation from the set Y = {y1, y2, ..yn} will be assigned to one specific action.

During the post-processing and error root-cause analysis, we use this information to find the root cause of errors immediately. Also, trace buffer controller can be programmed to capture data once it detects a specific vector of Y, providing fine control over the time to capture internal signals.

As noticed,[13] the main burden in synthesizing checkers for debug purposes is the detection of overlapped patterns. Assertion threading has been proposed in as a technique to deal with such issue [14]. Figure 4 illustrates how assertion threading works for the property mentioned previously. To utilize assertion threading to synthesis checkers with the ability to detect overlapped failure patterns, at first step a sequence automaton will be separated from its precondition automaton [8]. As it was shown in Fig. 4, the precondition automaton is a simple two state automaton.

Due to the fact that the precondition automaton has a self-loop with the true condition in the initial state, it continuously triggers until it sees the req signal. The activated token from the precondition signal should be redirect to multiple sequences.

Dispatcher works in a round robin manner and redirects this token to multiple sequences. Implementation details of assertion threading can be found in [8]. The disadvantages of this technique for synthesizing checkers that detect overlapped failure patterns are 1) the related hardware overhead for dispatcher unit as well as the recurrent flip-flops and combination circuits for each sequence 2) the synthesized checkers using this technique cannot work with the circuit frequency.

Since our proposed hierarchical trigger generator uses PHFSM as a building block for synthesizing assertion checkers, the synthesis checkers using our mechanism can detect overlapped failure patterns at a circuit frequency with lower hardware overhead.

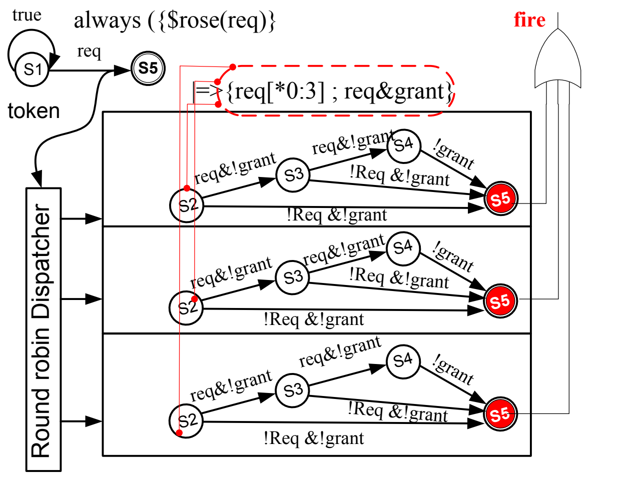

Figure 4. Assertion threading

Figure 5, illustrates the process of failure root cause analysis inside the generated trigger unit. We assume that we have two modules inside the PHGS (z0 and z1). Z1 generated based on the automaton of the previously mentioned property. Inside this property we might have 3 overleaped failure patterns. Therefore, the required parallelism for this property is 3.

As it was shown in this Figure 5, module and FSM stacks are set aside for this specific module (z1). Z0 is assumed to be the first active module inside this simple trigger.

Two overlapped failure patterns which activate failure status in z1 automaton are considered in this figure. As it was shown debug traces which involve the status of module and FSM stacks are sufficient to root cause failure inside the trigger.

When z1 is active any call to z1 (Figure 5 (a)) causes recursive and parallel activation of the same module utilizing the previously allocated module and FSM stacks (Figure 5 (b)).

As Figure 5 depicts, once any parallel version of z1 reaches to its final state, assertion violation will be triggered and the root cause analysis will be started using the debug traced already inside the module and FSM stack.

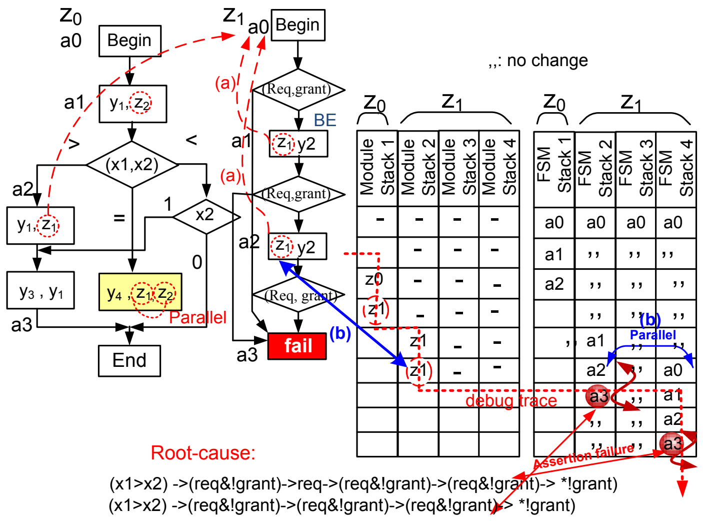

Figure 5. Proposed root cause analysis

4.2 Proposed Embedded Logic Analyzer (ELA)

Fig. 4 illustrates the proposed embedded logic analyzer that involves our generated trigger unit. In an ELA, trigger signals need to be monitored and the debug-data will be captured under the control of the trigger unit.

The trigger control unit monitors the switching between serialized data of the trigger unit and the trace buffer. In some failure cases there is no need to transfer the data inside trace buffer. Throughout the debug phase, current status of trigger unit, which involves data inside Module-Stack and FSM-Stack along with current state of trigger unit are serialized and transferred through JTAG.

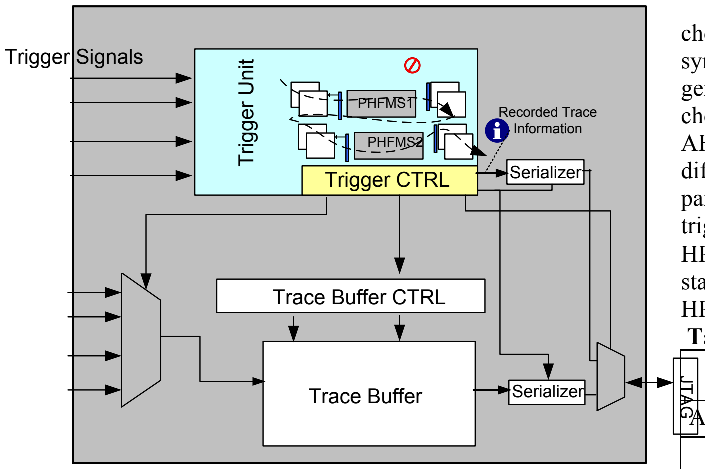

Figure 6. Embedded logic analyzer (ELA)

As shown earlier in Fig.4, trigger unit may have more than one PHFMS controller because in some cases ZiMH is not able to fit the PHGS into just one centralize PHFMS. In this case as it was shown in Fig. 4, the scan-out line of the first PHMS feeds the scan-in of the next one. It is also possible to compress the debug data.

Because our main contribution is on the trigger unit, in our experimental part we focus on the area over head of the trigger unit.

4.3 Debug Scheduling

As Fig. 5 illustrates, during failure root-cause analysis, it is not always mandatory to transfer big chunks of data stored inside the trace buffer. Internal status of the proposed trigger unit includes data inside FSM and Module Stacks. Because the data stored in trace buffer is significantly larger in size than internal status of trigger unit, as we will show in the experimental result, the generated trigger unit reduces trace data that need to be transferred during the post-silicon failure root cause analysis. Plus, another benefit of our approach is that the post processing software can use the generated HPGS of ZiMH to facilitate and improve the failure root-cause analysis.

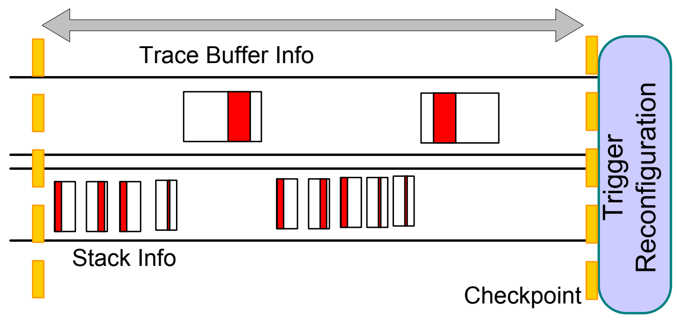

Figure 7. Root cause analysis

As Fig. 5 shows, assuming that we have reconfigurable block inside our SoC, we can generate trigger units that can get fit inside that reconfigurable unit. Thereafter, we can perform the process of post-silicon validation in a time-multiplex manner [15]. ZiMH generates trigger units based on two parameters, stack-size (h) and maximum allowable parallelism (p); by forcing ZiMH to generate several PHFSMS that can get fit inside the reconfigurable blocks, we can achieve the required fault coverage in several steps.

5. Experimental Results

To verify effectiveness of our method, we consider two case studies, and using ZiMH, we build automatically trigger units for two case studies. In our first experiment, trigger units for AMBA-slave and AMBA-AHB interface has been generated; Details of assertions that have been used by ZiMH can be found in [22] pages 182-190.

To find area overhead of trigger units that involves packs of checkers as well as the control unit; first, we build checkers and synthesize them with Xilinx ISE 9. After that we run MBAC in generator mode and feed ZiMH with FAs of these individual checkers. Thereby, ZiMH generates two separate trigger units for AHB-Interface and AMBA-slave. We run ZiMH(h, p) four times with different parameters related to stack-size (h) and maximum allowable parallelism (p). Table 1 represents the area overhead of the generated trigger units with different parameters. #T shows the numbers of HPFSM generated by ZiMH. For example, forcing ZiMH to have stack size (h = 4) and parallelism (p= 2) leads to two separated HPFSM.

Table 1. Trigger unit area overhead

ZiMH	(10,4)			(8, 3)			(16,3)			(4, 2)		
	FF	LUT	#T	FF	LUT	#T	FF	LUT	#T	FF	LUT	#T
AHB-Interface	197	285	1	133	290	1	229	294	1	138	310	2
AHB-Slave	190	202	1	126	203	1	222	201	1	140	312	2

The general rule about the trigger unit generated by ZiMH is that the higher the level of parallelism (p), the larger is the area overhead and the higher the frequency. On average, the generated trigger unit with its controller has 18% area overhead in comparisons to the central unit that packs all the checkers. For example, in the case of ZiMH (10,4), we have 30% area overhead in comparison to the case that we have one central monitor involves all the checkers. Another important consideration is that since the generated trigger unit has more hierarchical information there is no need to feed ZiMH with all the checkers' FAs. However, increasing the level of parallelism leads to more area overhead. When we have limited area aside for the trigger units, we can have the slow trigger unit with lower area overhead. Considering the fact that the major duty of the trigger unit is to discover error and provide root-cause analysis, we can easily deal with slow trigger unit by adding pipeline stages.

In the second experiment, we applied ZiMH trigger generator to create trigger unit for AXI Bus protocol from ARM[21].

The AMBA AXI is targeted at high-performance, high-frequency system designs. AXI protocol has five independent channels, and each channel uses a two-way valid and ready handshake mechanism. Each channel has particular set of assertions that monitors its related transactions.

The most distinctive feature of AXI protocol in contrary to AHB protocol signals is that multiple master devices can access to different slave devices simultaneously. In AHB system only one master can occupy the bus at one time. There are 85 assertions in the AMBA 3 AXI protocol; we generate two trigger units Master-side and Slave-side. In this case, it turns out that ZiMH could find a lot of explicit hierarchical overlaps during the process of PHGS generation. As a matter of fact, assuming two level of parallelisms (p=2) and sixteen level of hierarchies (h=16) inside trigger unit master-side trigger unit has 5% lower area overhead than central monitor unit which packs all the checkers [10].

Fig. 6 shows our experimental fault injection environment. We place the ELA that has our generated trigger unit as a core inside this experimental environment to detect fault and perform root cause analysis.

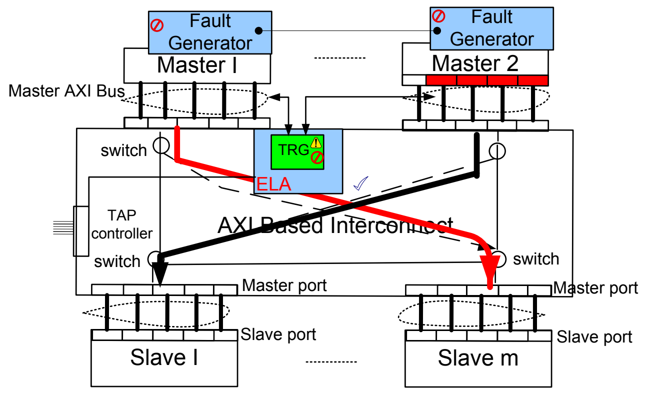

Figure 8. Fault Injections Environment

The fault generator units shown in Fig. 6 inject incorrect transactions that boil down to failure such as hand-shaking error, X-propagation and functional errors related to the AXI interface. The trigger unit as the core of ELA is in charge of discovering these faults. We also assume "Dependent Fault", scenarios such as Master 1 will issue an incorrect transaction provided that Master 2 in previous transactions had issued an specific transaction.

The proposed trigger unit manages to detect all the design faults (100% fault detection). In total, 65% of the time using its hierarchical information that has been stored in its stacks the process of root-cause analysis has done immediately. In other words, it was not necessary to transfer the content of trace buffer serially through the JTAG thanks to its internal stacks of data; our ELA in contrary to the central monitor unit in [10] performs the root-cause analysis of all the "Dependent Faults". We do not consider electrical bugs in our environment, but X propagations can be detected using ELA. Plus, we can easily write properties to detect electrical faults and feed ZiMH with the FAs related to those properties.

6. Conclusions

In this paper, we proposed ZiMH, the trigger generator that builds a synthesizable trigger unit as the core of ELA. The generated trigger unit provides resourceful and efficient trace information for root cause analysis of error. By keeping the trace of interactions that lead to the failure, the proposed trigger unit facilitates the process of failure localization and root-cause analysis. Moreover the proposed trigger unit generator can be tuned for the specific location of a design to avoid the huge cost related to interfacing with trace signals and it can get parameterized to generate several trigger units that can be placed inside the limited area in a time-multiplex fashion and multiple debug rounds. Through injecting fault (considering design errors), 100% of injected errors have been detected among them 65% of detected error could get localized without transferring the content of trace buffer.

In our future work, we are planning to enhance ZiMH to ignore definition of some redundant FAs. Further, work such as [5] provides a technique for compressing debug data. Combination of these two approaches will be another interesting area for new research.

7. References:

[1] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoC," *Proc Design Automation Conference*, pp. 7-12, 2006.

[2] B. Bentley, "Validating the Intel® Pentium® 4 microprocessor," *Proc. Design Automation Conference*, pp. 224-228, Jun. 2001.

[3] G. J. Van Rootselaar and B. Vermeulen, "Silicon debug: scan chains alone are not enough," *Proc. IEEE Intl. Test Conference ITC*, 1999, pp. 892-902.

[4] B. Vermeulen, T. Waayers, and S. K. Goel, "Core-based scan architecture for silicon debug," *Proc. IEEE International Test Conference (ITC)*, pp. 638-647, Baltimore, MD, USA, October 2002.

[5] E. Anis and N. Nicolici, "On Using Lossless Compression of Debug Data in Embedded Logic Analysis," *Proceedings of the IEEE International Test Conference*, 2007, paper 18.3.

[6] H. F. Ko and N. Nicolici, "Resource-Efficient Programmable Trigger Units for Post-Silicon Validation," in *Proceedings European Test Symposium (ETS)*, pp. 17-22, 2009.

[7] I. Wagner and V. Bertacco, "Reversi: Post-silicon validation system for modern microprocessor," in Proceedings IEEE International Conference on Computer Design (ICCD), pp. 307-314, Oct. 2008.

[8] M. Boule, and Z. Zilic, "Generating Hardware Assertion Checkers: for Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring," Springer Publishing Company 2008.

[9] J. Geuzebroek and B. Vermeulen, "Integration of Hardware Assertions in Systems-on-Chip," in *Proceedings IEEE International Test Conference (ITC)*, 2008, 10 pages.

[10] M. H. Neishaburi and Z. Zilic, "Enabling efficient post-silicon debug by clustering of hardware-assertions," in *Proceedings IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 985 – 988, March 2010.

[11] M. Boule and Z. Zilic, "Incorporating efficient assertion checkers into hardware emulation," in *Proceedings International Conference on Computer Design (ICCD)*, pp. 221–228, 2005

[12] M. Boule and Z. Zilic. "Automata-based assertion checker synthesis of PSL properties," *ACM Transactions on Design Automation of Electronic Systems (TODAES),* Vol. 13, No. 1, 20 pages, Jan. 2008.

[13] M. Boulé, J-S. Chenard and Z. Zilic, "Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug", Proc. IEEE Intl. Conf. Computer Design, ICCD, pp. 294-299, 2006.

[14] M. Boulé, J-S. Chenard and Z. Zilic, "Debugging Enhancements in Assertion-Checker Generation", IET Computers and Digital Techniques,Vol. 1, No. 6, pp. 669-677, Nov. 2007

[15] G. Ming, T. Cheng, K. Ting. "A case study of Time-Multiplexed Assertion Checking for post-silicon debugging," *Proc. High Level Design Validation and Test Workshop (HLDVT),* June 2010

[16] V. Sklyarov, Hierarchical Finite State Machines and Their Use for Digital Control, *IEEE Transactions on VLSI*, Vol. 7, No 2, June, 1999, pp. 222-228

[17] V. Sklyarov, I. Skliarova. "Design and implementation of parallel hierarchical finite state machines," *Proc. Intl. Conf. Communications and Electronics* (ICCE) pp. 33-38, June 2008.

[18] B. Vermeulen and S. K. Goel, "Design for debug: catching design errors in digital chips," *IEEE Design & Test of Computers*, vol. 19, pp. 35-43, 2002.

[19] G. J. Van Rootselaar and B. Vermeulen, "Silicon debug: scan chains alone are not enough," in *Proceedings IEEE International Test Conference (ITC)*, 1999, pp. 892-902.

[20] Park, S., T. Hong and S. Mitra, "Post-Silicon Bug Localization in Processors using Instruction Footprint Recording and Analysis (IFRA)," *IEEE Trans. CAD*, Vol. 28, No. 10, pp. 1545-1558, 2009.

[21] ARM AMBA 3 specification and assertions. http://www.arm.com/products/solutions/axi_spec.html

[22] B. Cohen, S. Venkataramanan, and A. Kumari. *Using PSL/ Sugar for Formal and Dynamic Verification*. VhdlCohen Publishing, Los Angeles, California, 2004