A Distributed AXI-based Platform for Post-Silicon Validation

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: *With a significant increase in the design complexity of cores and associated communication among them, post-silicon validation has become a demanding task in System on Chips (SoCs) design. To ensure that final products are fault-free and ready for market, the post-silicon validation goal is to catch bugs and pinpoint the root causes of errors that could escape from pre-silicon verification tools. Post-silicon validation involves running a hardware prototype in an environment that is similar to its final platform with its expected workload. As new SoCs tend to have many cores, the interactions among these cores are becoming so complex that post-silicon debug techniques should address not only validation of the functional aspects of a design but such techniques have to "bulletproof" the communication and synchronization among cores inside an SoC. In this paper, we propose an AXI based environment for post-silicon validation. The proposed environment involves Local Debugging Unit (LDU) and Shared Debugging Unit (SDU). LDU monitors trace of transactions issued by the hardware prototype and detect undesired conditions on bus. SDU combines debug traces from different LDUs. We embed the proposed SDU inside an AXI configurable interconnect. Major benefits of using our proposed debug platform over traditional techniques for silicon validation are as follows: 1) it detects and bypasses real time severe faulty conditions such as deadlocks resulting from design errors or electrical faults 2) there is no need for internal trace memory because SDU can communicate to the external memory through slave ports 3) it enables online monitoring of the trace buffer.*

I. INTRODUCTION

With the advances in technology and CAD tools, and driven by reckless demands of consumers for new functionality, new SoCs require to have many cores. To carry out their demanding functions, SoCs typically require substantial number of communication links among its embedded cores.

Verification tools have to ensure that such a complex system is error-free and the final products meet the strict time-to-market deadline. Despite new enhancements in pre-silicon verification techniques, bugs slip to the first hardware prototype in significant percentage. In general, once a first-silicon is placed in its intended target environment and the actual workload (software) is exercised, errors arise in a hardware prototype. *Design errors* as well as the *electrical errors* are two major source of failure in first-silicon.

A design error results from designer mistakes in implementing the high level specifications. To manifest these errors, we need to exercise corner cases of a design. Such errors may cause failures in intercommunications among cores; consequently, monitoring transactions among cores (maters/slaves) in a bus-based system contributes to their detection.

An electrical error emanates from transient errors inside storage elements of a system. Such increases in rate of electrical errors are due to crosstalk, low voltage levels, high frequency and small noise margins. As we will illustrate later on monitoring bus transaction with our proposed unit, we can not only detect and bypass design errors but also mitigate the severe consequences of electrical errors.

Conventional debug methods and tools tend to focus more on the computational part of a system, e. g. the processor and its interaction with main memory. However, many SOCs contain processing cores, e.g., Digital Signal Processors (DSPs) and interfaces and a considerable complexity resides in the interactions between these processors and other system components. Hence, a platform for post-silicon validation should enable not only functional validation of a prototype but it also has to guarantee that the prototype can realize the required compatibility with other blocks. Fig. 1 illustrates some of the components that are usually part of a post silicon validation test board [2]. The prototype under validation has to plug into this platform. Other peripherals expected to have interconnection with such prototype should also get connected through the proper bus. For instance, in the realm of microprocessors, a platform for post-silicon validation must enable the processor to start executing an application that resides in memory. Moreover, to emulate a real working condition, commercially available peripherals must be able to interrupt a processor to communicate with it. As Fig. 1 shows, a programmable interrupt generator can play the role of a peripheral by interrupting a processor in a suitable manner.

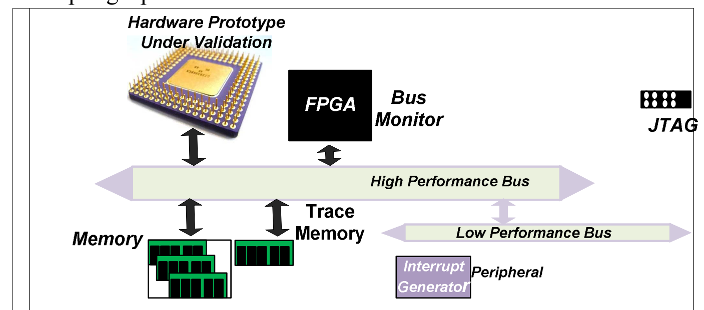

Fig. 1 Post-silicon validation environment

In this paper, we propose a distributed Advanced Extensible Interface (AXI) based platform for post silicon validation. A Local Debugging Unit (LDU) and Shared Debugging Unit (SDU) are the main components of our platform. An LDU is placed inside the AXI Master Interface that connects the hardware prototype and other masters to the AXI Bus.

LDUs are distributed among AXI Master Interfaces that connect master devices to the bus. Armed with two levels of hardware checkers that are generated automatically using MBAC checker generator [14], [15] in a way described as in [16], LDUs trace the transactions and signals issued by the hardware prototype and other modules connected to the bus, and detect the undesired conditions. The concept of assertions is taken from Assertion-Based Verification (ABV), the well-accepted pre-silicon verification techniques. LDUs are connected to the central SDU and transmit their debug traces to that module. SDU combines debug traces from different LDUs, and it schedules properly the extracted debug data from several masters to an external trace memory. We implant the proposed SDU inside an AXI configurable interconnect. The proposed post-silicon validation provides mechanisms to detect and bypass severe faulty conditions such as a deadlock resulting from design error or electrical errors.

978-1-61284-656-9/11/$26.00 ©2011 IEEE

Moreover, such a platform enables other master devices to access trace memory in an online manner; therefore, online debugging is achievable. To the best of our knowledge this works is the first study that provides the comprehensive distributed debug environment based on the AXI protocol.

II. Related Work

The main goal of Design for Debug (DfD) methods is to increase real-time observability and controllability of internal signals during post-silicon validation [1]. We can categorize broadly previous works on the SoC debug into two groups: 1) techniques that enhance observability of signals inside a core and 2) DfDs that enable monitoring inter cores communication.

The previous studies belonging to the first group mainly strive to employ conventional test techniques and resources available, such as scan chains and Test Access Mechanism (TAM) to achieve the required real-time observability [13], [19]. Once the specific trigger or hardware checker fires, internal state elements of a system are captured in parallel using available scan chains; subsequently, the captured data is offloaded serially utilizing scan-out operation. Finally, post processing algorithms analyze offloaded data. Such consecutive stops and resumptions are no longer practical for debugging of a complex system [1]. Another DfD technique dedicated to improving real time observability of signals inside the core is Embedded Logic Analyzer (ELA); ELA utilizes on-chip trace buffers and trigger units to capture signals in real-time [19]. In [6] a use of Assertion checkers in the context of wireless systems has been considered; however, authors provide no mechanism for the placement of the checkers inside a system. In [4][5] authors provide a hybrid HW/SW mechanism to reduce the effect of electrical errors in the SoCs that use Real-Time Operating Systems (RTOS). Further, synthesis of assertions that provides debug functionality was considered in [7]. Such methods mainly focus on debugging the functionality of a SoC, without any emphasis on communication. However, as more cores with various communication protocols constitute modern SoCs it is required to apply debug beyond the functionality check of single core and create a new debug mechanism.

Regarding debugging of inter-core communication, we have to consider bus protocols, handshaking mechanism, blocking and no blocking transaction etc. In [8] authors review the benefits of debugging and diagnosing at transaction level; Authors propose a debug system for network-on-chip (NoC) communications in [9]. A distributed performance analysis mechanism in an AXI-based system is introduced in [20]; however, their proposed environment cannot be utilized for debug and validation. In [15], we investigate a method for clustering assertion checkers inside the bus; however, there was no mechanism to trace transaction inside bus and detect and bypass undesired condition during the debug.

In [4] authors propose a method to raise the debug abstraction level of communication to transaction level in post-silicon validation. They utilize an offline method to find the erroneous sequences of transactions. However, considering the fact that as soon as the first severe failure such as a deadlock occurs, the system cannot proceed further. Therefore, there is a need for an online monitoring unit that extracts and stores erroneous transactions, thereafter takes proper measures concerning the failures. As we will see later on, such failures resulting from design errors and electrical errors are typical in new complex buses. In [10] authors explore the effect of 1-bit transient error on AXI-based interconnect. Here, we consider effect of both design errors and transient errors. Our proposed post-silicon validation provides efficient mechanisms to detect, store and bypass severe faulty conditions such as deadlock and live lock result from design error or electrical errors. Moreover, we relax the requirements of having transaction level assertions by utilizing SystemVerilog Assertions (SVA) language and the MBAC checker generator [15].

III. Preliminaries

In this section, we briefly review a few concepts that will be used during the paper.

1.1 Assertions and Checkers

Assertion is a statement that indicates how a given circuit should behave under different circumstances. Assertions represent a complex range of behaviors. Assertion-Based Verification (ABV) is one of the most practical and efficient RTL verification techniques for pre-silicon verification. System designers are able to define both expected and unexpected behaviors of a design using the temporal logic and the extended regular expressions using assertion languages such as PSL (Property Specification Language, IEEE 1850 standard) and the SVA. 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 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-hand arguments, respectively. $rose(b) as a precondition evaluates to true in the case of any changes in “b” 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. Assertions once converted to RTL module usually called checker.

Here, we use MBAC tool to generate checkers [15]. First, MBAC generates an automaton with respect to each assertion statement; thereby, a set of automata for properties and sequences are generated. A generated automaton is a directed graph, where vertices are states, and edges among states shows the conditions for transitions among them. Fig. 2 shows the generated automata from the previous mentioned PSL assertion. It has been shown in [15] that the properties in PSL and SVA can be converted in a recursive manner to an equivalent finite automaton. Assertion violation is activated whenever an automaton representing an assertion reaches its final (failure) state.

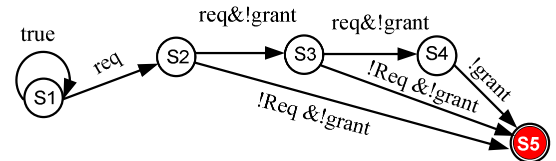

Fig. 2 Generated automaton from the PSL statement

In this paper, we model a transaction level assertion with two assertion statements. The first assertion represents the sequences of events that maintain non-periodic trigger signals for activation of the second assertion. The second assertion follows the sequences of transactions that cause erroneous status such as deadlock or livelock.

1.2 AXI Protocol

The AXI bus protocol is an enhanced bus protocol of the existing Advanced High-performance Bus (AHB). AXI is targeted at high-performance, high-frequency system designs. AXI protocol has five independent unidirectional channels that carry the address/control and data. Each channel uses a two-way valid and ready handshake mechanism. The five independents channels are the Address Read (AR) channel, Address Write (AW) channel, Read Data (RD) channel, Write Data (WD) channel, and write response channel (B). AW and AR channel convey the address and control of write and read transactions. The control signals of such channels describe the nature of the read and write transactions.

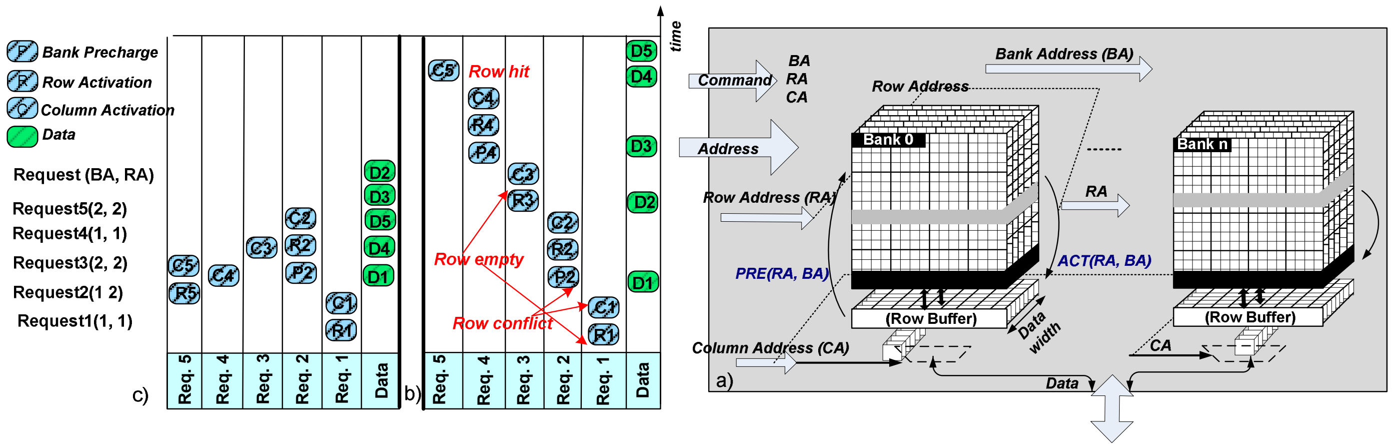

Fig. 3 SDRAM: a) scheduling and b) architecture

A transaction can be a burst of a different length, or it can be atomic. A burst is composed of a number of transfers whose length is defined. The data is transferred between master and slave, and vice versa using WD and RD channels respectively. Write response channel (B) allows a salve to signal completion of the write transaction or an error. One of the features of AXI is a burst transaction with only the start address issued. The split transaction AXI protocol enables out-of-order transaction completion; it provides a "transaction ID" field assigned to each transaction. Transactions from the same master IP core, but with different IDs have no ordering restriction while transactions with the same ID must be completed in order. Out-of-order transaction transactions completion improves system performance in that it allows a complex slave like memory return data out-of order. For instance, a data item of an earlier access might be available from an internal buffer sooner that of later access (temporal locality).

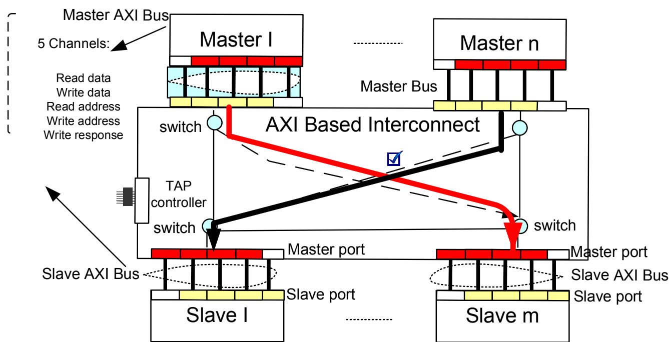

Fig. 4 An AXI-based system

Out of order execution and interleaving are the two main features of AXI bus that provide high throughput, but also increase the susceptibility of a system to bugs.

1.3 SDRAM

Design errors and bugs in a hardware prototype may lead to deadlock, livelock or other undesired conditions inside a bus. To understand how these failures take place, we explain the rescheduling mechanisms of the SDRAM in the sequel. SDRAM as one of the common complex slaves provides high bandwidth by executing memory requests in parallel. Memory requests are served by memory controller. It issues the required commands corresponding to each request and schedules them on SDRAM buses. By providing request reordering, the scheduler can provide higher bandwidth throughout.

SDRAM has a 3-D structure that involves banks, rows, and columns. Having multiple independent banks in such a 3-D structure enables memory scheduler to service serial requests in parallel; moreover, commands to different banks can be pipelined. Address bus inside SDRAM is divided into three parts: Bank Address (BA), Row Address (RA) and Column Address (CA)[12][3]. BA specifies one of the banks inside an SDRAM, while RA and CA points to the particular row and column on that bank. SDRAM controller works with three commands: Activate (ACT), Read/Write(R/W) and Precharge (PRE). Taking the RA and BA, the ACT command activates the particular row (RA) inside the bank (BA) and places that row on the row buffer of that bank after tRCD. The row buffer serves as a cache to reduce the latency of subsequent accesses to that row. PRE command gets the BA and after tPR copy the content of row buffer to its related row in the bank, then it makes that bank idle. The R/W commands can be executed only after a bank is activated and the row buffer contains the particular row that they want to access. After either read latency called column access strobe (CAS) latency (CL) or write latency (WL), successive data go from or to SDRAM. In this paper, we assume timing constraint of DDR2-512MB which is 2-2-2 (tRP-tRCD-tCL) [22]. Latency of a memory request depends on whether the requested row is in the row buffer of the bank or not, a memory request could be a row hit, row conflict or row empty with different latencies. Memory performance suffers from bank conflict and data contention. To improve its performance and decrease bank conflict, memory scheduler should reschedule the requests.

As Fig. 4 (b) and (c) illustrates scheduling affects the performance. We consider five memory requests. As it was shown in Fig. 4 (b), Request 1 and 3 are row empties, and request 1 and 2 are row conflicts, and request 5 and 4 are row hit.

If the memory controller schedules these memory requests in order, it will take 24 memory cycles to complete them Fig. 4 (b). However, in Fig. 4(c) the same five requests are scheduled out of order. As can be seen, request 5 is scheduled before request 2, 3 and 4. In addition, request 4 is pipelined after request 5, called bank interleaving, since it has the different bank address from the bank address of request 4. As a result, only 14 memory cycles are needed to complete the four requests. Therefore, out of order memory scheduling provides better memory utilization 5/14= 35% over 5(data)/24(cycle) = 20%. However, as we will see out of order memory scheduling leads to increase in susceptibility of the memory controller to design errors and bug.

IV. FAILURES IN AXI BUS

In this section, we introduce common errors that may happen during the validation of the hardware prototype. These problems that target communications among cores are mainly result of a design or electrical errors in bus interfaces (masters/slaves), memory controller and microprocessors. It is responsibility of LDU and SDU to detect these errors and apply a proper measure to relax their destructive effects and expedite root cause analysis of errors. Such errors fall into four categories: race condition, deadlock, livelock and data inconsistency.

a. Deadlock

Both design errors and electrical errors might lead to a deadlock in AXI. It is obvious that a debug unit should detect the deadlock, dismiss it, and let a system proceed with its operation; otherwise, the system under debug should get restarted. As it is illustrated in Fig. 6, supposing that at time 4, master1 issues transaction A1_2, which is the burst write. Meanwhile, due to a temporary error, the value of "LEN", which indicates a number of transfers in write transaction, gets changed. As a consequence of such an error, once slave2 starts receiving data, it expects either more or less than the number of actual data transfer. As soon as slave2 receives the expected number of data ("LEN"), it sends an acknowledgement on the B channel.

However, in the first scenario illustrated in Fig. 6, where a slave expects more data transfer than the actual one, not only slave2 gets block at time 9, while waiting to receive more data, but also master1 which is waiting to receive an acknowledgement on its B channel can no longer proceed.

The second scenario which slave2 expects less data transfers than the actual one causes Data-inconsistency which we will explain in section D. Deadlocks not only emanate from intermittent errors but it might also results from design errors. For instance, as it was shown in Fig.6, master 2 has issued following in order transactions: B2, C2 and D2 at time 2, 3 and 4. Assuming there is a design error inside the reordering unit of Mem 1. Consequently, Mem1 at time 7 provide data related to read transaction D2 sooner than that of C2. However, Master2 while expecting to receive C2 stops data D2 at time 7 that has the same transaction ID as C2. Thereafter data C2 cannot advance due to the blocked D2, which is the case of deadlock.

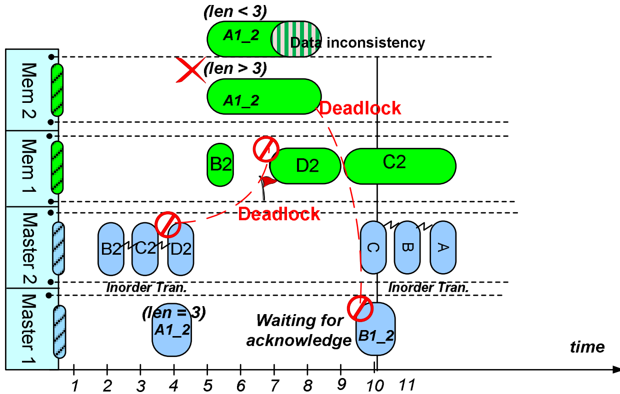

Fig. 5 Deadlock condition

b. Race Condition

Race condition usually occurs when multiple masters are writing to the same place with overlapped or consecutive transactions, or a master is to process two in-order write transactions (not necessarily consecutive) that has the same address. Any undesirable changes in execution order of such transactions may cause inconsistency in the content of memory (memory corruption). For example temporarily errors on the transaction ID field of one of the sequence of in-order write transactions makes memory controller consider that transaction out-of order; subsequently, random execution orders of these transaction defines the data consistency. As it was shown in Fig.7, due to the temporarily error in transaction ID of C2 or D2, memory controller at time 11might schedule them differently. Their execution order may change the result of the read transaction D2 at time 11 in that it can get the newer value of D2 provided that write transaction D2 on Mem1 execute at time 12. Design errors might also create race condition. For example, if master side interface without considering the dependency mistakenly assigns transaction ID to the read or write transactions the arbitrary scheduling of memory scheduler determine whether read or write transactions executed properly.

c. Livelock

Livelock is a situation that a system performs continuously the same sequence of operations without any changes in the status of that system. Livelock is different from deadlock in that once deadlock occurs in a system, such system no longer can proceed. For example, in Fig. 7 Master1 issues the A1_2 write transaction at time 3 and starts sending data. Once it finalizes its data transmission at time 7, it waits to receive an acknowledgement from its Channel B.

However, because of a design error in the memory controller, Mem2 is unable to detect completion of data transfer; consequently, it sends an error signal instead of an acknowledgement. Such a circumstance causes Mester1 initiates again its previous A1_2 write request at time 8 and the same scenario occurs continually.

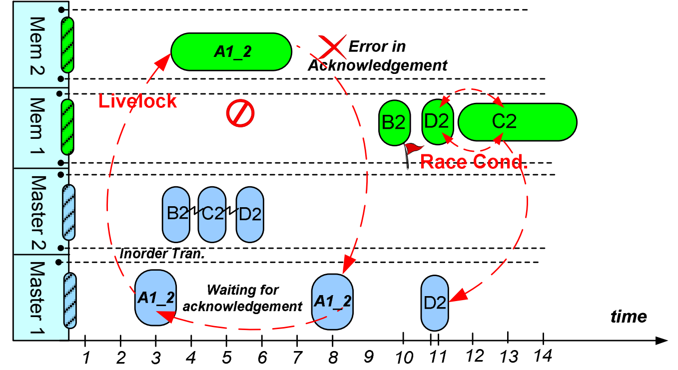

Fig. 6 Livelock and Race condition

d. Data Inconsistncy

Data inconsistency could cause failure later on, if the data is used at some point. Considering the previous example, which deals with the temporarily error on the value of "LEN", As fig.6 illustrates, If Mem2 samples value of "LEN" lower than the actual value at time 5, the memory controller of Meme2 will have problem generating memory address to store the data coming from Master 2. The data either will be restored in the same place or they will be restored in other parts of the memory. Design errors might also cause data inconsistency. For example, design errors that lead to race condition in Fig.7 might cause data inconsistency for the read transaction D2 at time 11.

V. Proposed Debug Platform

Our proposed AXI based debug environment consists of two main units: Local Debug Unit (LDU) and Shared Debug Unit (SDU). In the sequel, we will explain the architecture of LDU an SDU.

a. Local Debuging Unit (LDU)

Architecture of the proposed AXI master interface is illustrated in Fig. 7. A master that initiates several in order requests expect to receive requested data concerning to a read transaction or an acknowledgment to a write transaction in order.

However, such in-order transactions might get served out of order by slaves e.g. memories. For instance, as we explained before, SDRAM controller as a typical slave module performs transactions reordering to gain high performance and lower latency.

An AXI master relies on the AXI-based memory controller to receive its in-order requests properly. However, in case of bugs inside the memory controllers or other slave interfaces, a master no longer can communicate with its memory or other slaves due to the incident of the one of the failures, which is mentioned in previous section.

Not only validation process might get stuck because of bug in slave side, but bugs inside the master modules also hamper the validation process.

Our proposed debug mechanism (LDU and SDU) takes advantages of the observability of transactions inside the AXI

Interface. LDU monitors follow of transactions inside the wrapper. By placing LDU inside the AXI Interface, we maintain resource efficiency in that many components are shared between the units.

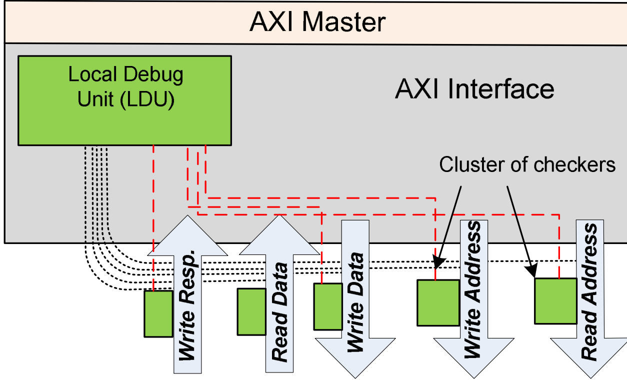

Fig. 7 AXI Interface Master side and LDU

The proposed LDU consists of three main units. 1) Transaction Register (TR) 2) Transaction Table 3) Cluster of checkers. TR keeps track of all the in-order transactions. The master shows its intention for having several in-order requests (transactions) by assigning the same transaction ID to them. It is slave responsibility to ensure such transactions which may complete out of order are issued in order to the master. Any deviation from such rules must be detected by LDU. To carry out its task, LDU assigns different Tag-IDs to these in-order transactions. TR is a 32-bit register, where each bit corresponds to one of the AXI transaction IDs. To record states of the outstanding transactions, LDU adopts a table called Transaction-Table [16]. Each row in this table corresponds to a request with the same transaction ID. Every row involves valid tag (V), Transaction ID (T-ID), Number of outstanding transactions (N-T) and Expecting Tag number (E-T). As Fig. 8 shows a bit at the specific location inside the TR register has a direct correspondence with a request that master has already issued.

Once a master initiates a request on AXI read address, LDU compares T-ID of this new request with the concerning bit at TR register. If that bit was set before, LDU realizes that either one or more than one request with the same transaction ID have been issued before, otherwise LDU updates TR register by assigning 1 to that particular bit. In the first case, a new Tag_ID is produced by LDU and once request wants to get buffered at the next level, Tag_ID will be attached to it. Tag_ID indicates the order of the request within the transactions with the same transaction ID.

On the other hand, once a master receives either data concerning its read transaction or acknowledgment with respect to its write transaction, LDU again explores and updates the TR register and Transaction table. In fact it is a critical moment for LDU that can detect deadlock or other erroneous condition.

As Fig. 8 illustrates LDU first investigates whether the new served transaction has been issued before or not by looking at the TR register. Then if such a transaction has been initiated before it will be considered whether its Tag-ID matches with the Expected-tag-ID from the Transaction Table. If there is no such match, LDU will detect the deadlock, interrupt the processor and will send trace of data contain information of transaction-table as well as TR-register.

As Fig. 8 illustrates, once a newly served transaction passes all the checks without any inconsistencies, SDU updates Expected-Tag-ID field of the entry in the transaction-table that is associate with the Transaction-ID of the served request. Meanwhile, if the Expected-Tag-Id becomes zero, that means that all the in-order transactions with that particular Transaction-ID were served properly and the corresponding entry in the transaction table will be freed.

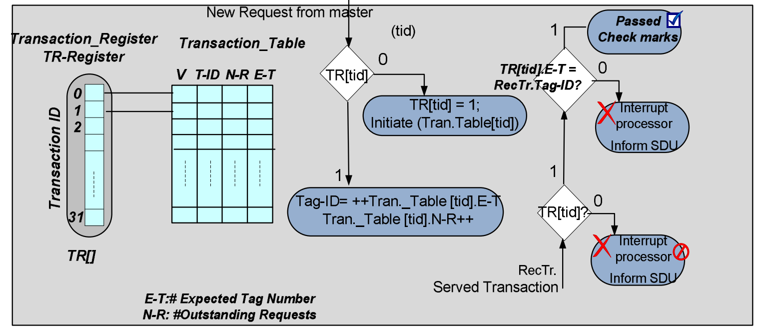

Fig. 8 LDU transaction control

Not only an LDU monitors the connected hardware prototype at transaction level, but it also continuously checks actions of the attached hardware prototype such as handshaking and X-propagations by means of hardware checkers. We have used the method in [16] to cluster these checkers; however, here we placed each checker cluster as close as possible to the source of signals that need to get monitor. These assertions are selected among 83 assertion statements available in [21]. We have used MBAC [15] to convert these assertion to hardware checkers and then we synthesize them using Xilinx ISE 9. These cluster of checkers are connected to LDU, as soon as one of the checker inside each cluster get fired, LDU interrupts the processor and send the debug trace which involves both cluster-ID and checker-ID to SDU.

b. Shared Debugging Unit (SDU)

Architecture of general SDU is illustrated in Fig. 9. SDU combines trace information from LDUs and schedules them to the trace memory. SDU traces failure patterns that might lead to erroneous conditions on bus. Since SDU is connected directly to the trace memory, in Fig. 9, we pictured both units. In fact SDU can be programmed to perform DMA based operation. In our future work, we plan to extend SDU with new functionalities that will enhance the proposed post-silicon validation environment.

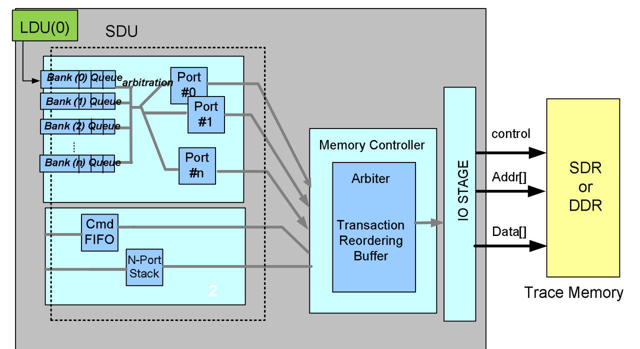

Fig. 9 SDU architecture

VI. Experimental Results

To evaluate our proposed post-silicon validation environment, we have implemented an environment illustrated in Fig. 10. The platform consists of two instances of 16-bit SAYEH processors [23] with their cache controllers. These two instances of processor are connected to our proposed post-silicon validation using an AXI bus. We utilized a memory controller module from Gaisler ip-cores[24].

The interrupt generator unit is periodically asserted during the test execution. Interrupt Service Routine that resides in memory will be executed once the interrupt pin of a processor gets activated by Interrupt generator unit. Once an interrupt pin is activated, the time stamp and its status is stored to the previously assigned memory location. LDUs that are placed inside the AXI interface all monitor

the local transactions. We intentionally inserted design errors inside the processors by changing the functionality of some parts of each core. We injected these faults inside SAYEH processor, memory controller and AXI interface. We assumed that LDU and SDU are fault-free. We consider design error at the RTL level. In other words, we injected errors in RTL specification of the processors and the memory controller. The errors are tuned to represent the corner cases. For example, inside the memory controller, we target memory scheduler and in the processor, we injected faults inside the forwarding unit, cache controller and interrupt controller such that the expected functionalities of these units get changed.

During the debug phase, the processors start running the program residing in the memory. Table 1 shows the result of synthesizing one SAYEH processor on Virtex IV family of Xilinx FPGAs.

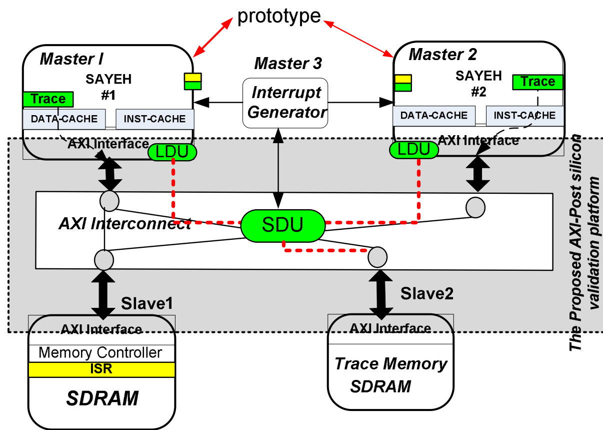

Fig. 10 our experimental environment

Table 1

32 bits Architecture	CLB SLICE	Utilization
SDU	290	4.70%
LDU (45 checkers)	929	7.57%
AXI Interface	271	4.43%
SAYEH with Cache and Interrupt	1162	18.94%

During the validation of faulty hardware prototype, deadlock never happened in our proposed AXI–based validation platform. It turned out that 94% of the errors inside the memory controller have been detected using the LDU; moreover, 87% of injected errors the AXI interface inside have been detected. Fig. 11 illustrates the patterns of failures inside the proposed validation platform system. As it was illustrated in this figure 40% and 35% of design errors lead to data inconsistency and deadlock respectively.

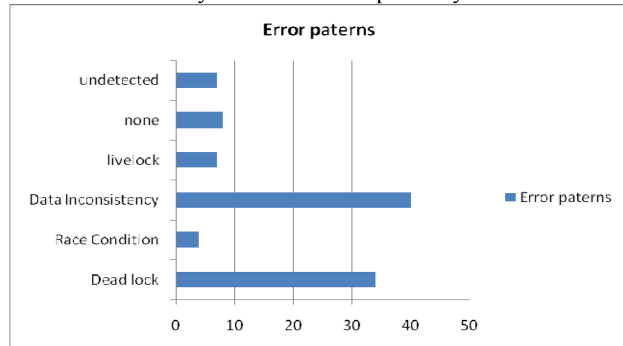

Fig. 11 error patterns

VII. CONCLUSIONS

We presented in this paper an AXI-based post-silicon validation platform. The proposed post-silicon validation provides mechanisms to detect and bypass severe faulty conditions such as deadlocks resulting from design error or electrical errors. Plus, our platform enabled online debugging by letting other master devices access trace memory in an online manner.

References:

[1] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoC," *Proc. 43rd Design Automation Conference* (43rd DAC), pp. 7-12, 2006.

[2] V. Bertacco, "Post-silicon debugging for multi-core designs," in *Proceedings of 15th Asia and South Pacific Design Automation Conference* (ASP-DAC), pp. 255 – 258, 2010.

[3] M. Daneshtalab, M. Ebrahimi, P. Liljeberg, J. Plosila, H. Tenhunen, "A Low-Latency and Memory-Efficient On-chip Network," *Proc. Intl. Symposium on Networks-on-Chip (NOCS)*, pp. 99 – 106, 2010.

[4] M. H. Neishaburi, M. R. Kakoee, M. Daneshtalab and S. Safari, "HW/SW architecture for soft-error cancellation in real-time operating system," *IEICE Electron. Express*, Vol. 4, No. 23, pp.755-761, (2007)

[5] M. H. Neishaburi, M.R. Kakoee, M. Daneshtalab, S. Safari, Z. Navabi, "A HW/SW Architecture to Reduce the Effects of Soft-Errors in Real-Time Operating System Services," *Proceedings of* DDECS 2007, pp. 247-250, 2007.

[6] V. Kallankara, M. H. Neishaburi, Z. Zilic and K. Radecka, "Using Assertions for Wireless System Monitoring and Debugging," *Proc. of IEEE International* NEWCAS Conference, pp. 401-404, 2010.

[7] M. Boulé, J-S. Chenard and Z. Zilic, "Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug", *Proceedings of IEEE International Conference on Computer Design*, ICCD 06, pp. 294-299, Oct. 2006.

[8] A.M. Gharehbaghi, M. Fujita, "Transaction-based debugging of system-on-chips with patterns." *Proceedings of IEEE International Conference on Computer Design* (ICCD), pp. 186-192, 2009.

[9] K. Goossens, B. Vermeulen, R. van Steeden and M. Bennebroek, "Transaction-Based Communication-Centric Debug," *Proceedings of Intl. Symposium on Networks on Chip*, NOCS'07, pp. 95-106, 2007.

[10] W. Kwon, S.Yoo, J. Um; S.Jeong, "In-network reorder buffer to improve overall NoC performance while resolving the in-order requirement problem," *Proceedings of Design, Automation & Test in Europe Conference* DATE '09, pp. 1058 - 1063, 2009.

[11] D. Graham, P. Strid, S. Roy, F. Rodriguez, "A low-tech solution to avoid the severe impact of transient errors on the IP interconnect," *in proceeding of IEEE/IFIP International Conference on Dependable Systems & Networks* (DSN), pp 478-483, 2009.

[12] W. Jang, D. Z Pan, "An SDRAM-Aware Router for Networks-on-Chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (TCAD), Volume: 29 , Issue: 10, pp. 1572 - 1585, 2010

[13] 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.

[14] M. Boulé and Z. Zilic, "Efficient Automata-Based Assertion-Checker Synthesis of SEREs for Hardware Emulation", *Proc. Asia South Pacific Design Automation Conference*, ASP-DAC2007, pp. 324-329, 2007.

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

[16] M. H Neishaburi, Z. Zilic, "Reliability aware NoC router architecture using input channel buffer sharing," *Proceedings of Great Lake Symposium on VLSI* (GLSVLSI), pp 511-516, 2009.

[17] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," *Proc. of ISCA '00*, pp. 128-138, US, 2000.

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

[19] H. F. Ko, A. B. Kinsman and N. Nicolici, "Distributed Embedded Logic Analysis for Post-Silicon Validation of SoC," *Proceedings of IEEE International Test Conference (ITC)*, 10 pages , 2008.

[20] H.-m. Kyung et al. Design and Implementation of Performance Analysis Unit (PAU) for AXI-based multi-core system on a chip (SoC), *Microprocess. Microsyst*. (2010), doi:10.1017/j.micpro.2010.03.001.

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

[22] Micron Technology, *Micron 512Mb: x4, x8, x16 DDR2 SDRAM Datasheet*, 2006.

[23] Z. Navabi, "*Digital Design and Implementation with Field Programmable Devices*," Springer, May 2004.

[24] *Gaisler IP Cores, http://www.gaisler.com/products/grlib/, 2009.*