Microelectronics Reliability xxx (2012) xxx–xxx

Contents lists available at SciVerse ScienceDirect

Microelectronics Reliability

journal homepage: www.elsevier.com/locate/microrel

Review paper

An infrastructure for debug using clusters of assertion-checkers

M.H. Neishaburi *, Zeljko Zilic

McGill University, Department of Electrical Engineering, Montreal, Quebec, Canada

ARTICLE INFO

Article history:
Received 5 August 2011
Received in revised form 19 April 2012
Accepted 20 April 2012
Available online xxxx

ABSTRACT

It has become indispensable to locate circuit defects and find the root-cause of errors as soon as the prototype of a system (first-silicon) gets ready. Various Design-for-Debug (DfD) solutions have been introduced as a means to increase the observability and controllability of internal signals, resulting to a speed-up in debugging process and a decrease in the time-to-market of new products. Assertion Based Verification (ABV) is one of the instrumental pre-silicon verification techniques. Once assertions are converted to hardware modules and incorporated into a debug infrastructure, the post-silicon debug can benefit from the additional observability provided by such assertions. In this paper, we first propose a new algorithm that generates clusters of assertion-checkers; in our proposed clustering algorithm, we resort to a graph partitioning algorithm to find the assertion-checkers that can be placed inside a cluster. The proposed method generates the clusters of assertion-checkers by means of exploring the logic-cones set of each assertion-checker. Moreover, coverage metrics for different configurations of clusters are defined. Then, we introduce several mechanisms through which the clusters of assertion-checkers can be incorporated into the DfD infrastructures. In our experiments, several case studies such as AXI bus, PCI bus protocol and a memory controller are considered; thereafter, the proposed debug infrastructure containing clusters of assertion-checkers is embedded into such case studies. It turns out that contrary to a non-clustering approach of placing assertion-checkers into a design the clustering algorithm along with the proposed method for incorporating assertion-checker clusters into a debug infrastructure lead to the better results in terms of the energy consumption and design coverage.

Contents

* Corresponding author. Tel.: +1 5146793344.
E-mail addresses: mh.neishabouri@mail.mcgill.ca (M.H. Neishaburi), zeljko.zilic@mcgill.ca (Z. Zilic).

0026-2714/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.microrel.2012.04.016

1. Introduction

With the rapid development of semiconductor technology, increasingly complex systems are being integrated into a single chip. Driven by high demands for a large set of new features, the design errors and bugs have become prevalent and difficult to track. The increase in the time-to-market of new products as a consequence of unpredictable bugs may cause a significant loss of market share, or even complete loss of revenue [1]. Hence, to ensure that new products can meet the strict time-to-market deadline, finding these defects and bugs in a timely and cost-effective manner is a must. Pre-silicon verification techniques which broadly belong to functional (dynamic) or formal (static) methods have been around for decades; however, such techniques cannot nowadays ensure that a post-fabricated IC, usually referred to as "first-silicon", works perfectly.

Almost two-thirds of newly manufactured SoC products suffer from the undetected defects and bugs in the first-silicon [1]. Factors such as the incorrect interpretation of specifications, human mistakes, design misinterpretations and errors in CAD tools can be designated as potential reasons for the failure in verification and possible defects in silicon. Plus, the issues such as the lack of accurate models for a complex design, the "electrical" bugs caused by crosstalk or power drops, and design marginalities make a through design validation and debugging much more difficult in the pre-silicon than in the post-silicon phase. For instance, due to the complexity of full-chip simulation, bugs may escape from simulation-based verification as many corner cases could be missed. Therefore, once the first-silicon becomes available, it is required to identify any bug resulting from either design errors, electrical faults or the issues related to Process–Voltage–Temperature (PVT) corners. It has been observed that close to 50% of the total development cycles for a new product is spent on validating the system behaviors after the availability of the first silicon [3].

The post-silicon validation as a means to identify and localize design errors and bugs has gained a lot of attention in industry. Post-silicon validation is the process of applying input stimulus to the design, and it can be performed at the system operational speed. The so-called "deep states" and corner cases would more likely be exercised and thus there will be a better chance to catch hard-to-detect bugs. Although post-silicon validation mechanisms can offer a raw performance in terms of the execution speed of test cases, they need to be improved in order to increase the real-time observability of the signals. Therefore, there is a huge demand for new methods that enable faster and more accurate debugging.

Assertion-Based Verification (ABV) is one of the instrumental pre-silicon verification techniques. Armed with temporal logic and extended regular expressions, PSL (Property Specification Language, IEEE 1850 standard) [22] and SVA (System Verilog Assertions) [23] are the modern verification languages to describe the expected behaviors of a design. Any deviations from the expected behaviors are captured by means of placing sufficient assertion inside a CUD (Circuit under Debug); thus increasing the visibility within the CUD, and enabling accurate debugging.

To expand the functionality of assertions beyond pre-silicon verification, a checker generator tool must be employed to convert assertions to hardware modules. Consequently, such modules must be incorporated efficiently in a debug infrastructure.

In the context of post-silicon debugging, assertions must be synthesized before one can integrate them inside a design. An individual assertion once converted into a circuitry is referred to as an ***"assertion-checker"*** or a checker. In the remaining of this paper, we use the term ***"assertion-checker"*** to refer to hardware-based assertions. Here, we have used the MBAC checker generator which can produce assertion-checkers from either PSL or SVA assertions [8] .

Post-silicon validation involves three major activities: (1) detecting errors through embedded DfD (Design-for-Debug) infrastructures by means of applying a proper stimulus, (2) localizing and identifying the root cause of problems, (3) correcting or bypassing errors. The post-silicon bug localization step involves identifying the location-time pair of bugs and is the most time-consuming step.

For incorporating assertion-checkers and capturing their violation signals, a debug module inside a CUD must be equipped with a suitable debug infrastructure [3,18]. As system complexity increases, more assertions are needed to ensure that corner cases of a design can be covered. In general, the more assertion checkers embedded inside a CUD, the higher the hardware overhead and energy consumption related to the debug infrastructure [8].

In this work, we have discovered that by grouping assertion-checkers and placing them inside clusters, the integration of assertions inside a circuit becomes easier. Plus, having clusters of assertion-checkers, and controlling each cluster selectively during the debug and normal operational mode causes lower energy consumption. Moreover, the time-consuming process of identifying the root-causes of failures will be significantly reduced by selectively offloading the related information of the clusters that contain fired assertions. In this paper, we extend the concepts and definitions explored in [6] and provide the implementation details and comprehensive comparison with the previous work. The prior work on post-silicon debugging that centers around the use of assertion-checkers is described in Section 2. Section 3 provides the definitions and concepts required throughout the paper. The proposed assertion-checkers clustering algorithm will be discussed in Section 4. A discussion on the generalities of the proposed clustering algorithm and how to employ it is presented in Section 5. The integration mechanism of assertion-checker clusters into different debug infrastructures along with experimental results is provided in Section 6. Finally, Section 7 concludes the paper.

1.1. Contributions

The unique contributions of this paper towards the efficient assertion-based debug are following:

- Introduction of a general assertion-checker clustering algorithm.
- Integration of the assertion-checker cluster into different debug infrastructures.
- Introduction of Shared Debug Unit (SDU) as a new debug infrastructure suited for SoCs debugging.

2. Related work

Post-silicon debugging can be performed using two major schemes: (1) real-time trace-based methods, (2) run-stop scan-based techniques. Previous studies have considered a wide range of different implementations for such infrastructures [4,5,10,13].

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016

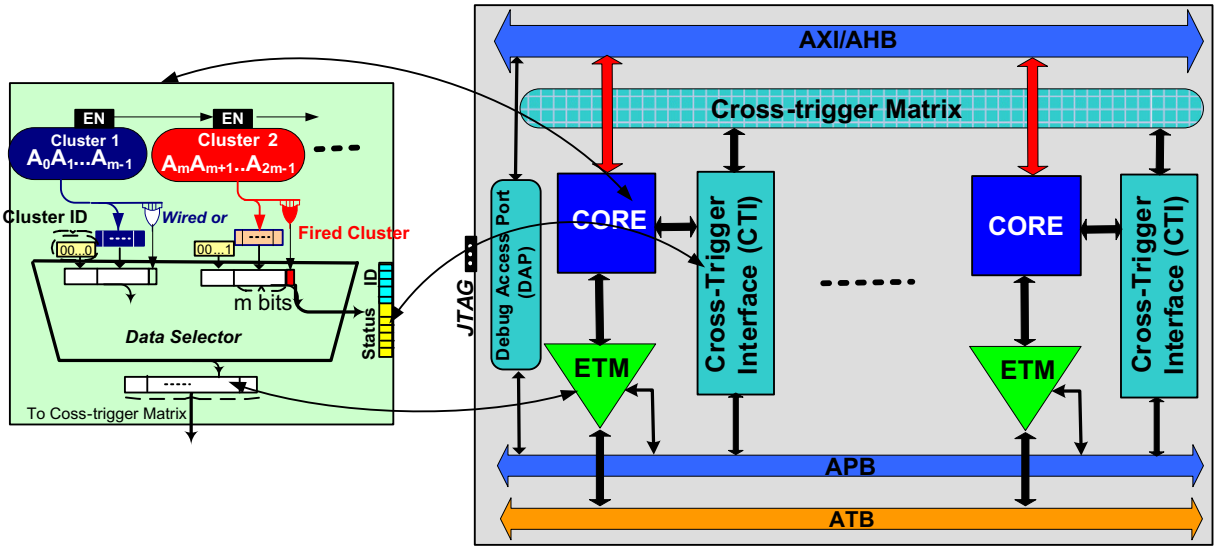

Fig. 1. Possible incorporation of the proposed infrastructure inside ARM CoreSight [24].

The primary goal in a scan-based debug approach is to reuse the internal scan chains that are being used during the manufacturing test. Whenever a specific programmable trigger or breakpoint module fires, all the internal states and signals are captured by means of available scan chains; thereafter, the captured data are offloaded using the 'scan-out' operation. Finally, to find out the exact causes of failures, a post-processing algorithm is applied to the offloaded data [15]. Due to the consecutive stops and resumptions, the scan-based debug technique cannot provide the required debug information in a real-time fashion [5]. Plus, this debugging scheme is slow and intrusive [5,24].

A trace buffer serves as a temporary space to keep the snapshot of a system under debug, including its signals and states, whenever a particular event occurs [17]. Trace buffers have been widely used in legacy debug and logic analysis systems [18,24]. For instance, as a multi-core debug solution for an AMBA based SoC, ARM presented CoreSight [24]. CoreSight uses Embedded Trace Microcell (ETM) as a debug core supporting modules and probe AMBA bus directly. As shown in Fig. 1, the Cross Trigger Interface (CTI) broadcasts trigger requests among embedded cores by means of the Cross Trigger Matrix (CTM). The registers inside the CTI and CTM blocks indicate the trigger conditions as well as trigger mapping, and they are programmed through IEEE 1149.1 (JTAG).

The proposed debug infrastructure in this work is orthogonal to the ARM CoreSight debug scheme. Triggers and breakpoint module inside an embedded core need to transfer their signals to the CTI unit. Once an assertion-checker detects an illegal sequence of events, it also raises an output signal. Therefore, an assertion-checker can be treated as a trigger unit. The only difference between assertion-checkers and regular hardware triggers and breakpoints is that hardware based triggers are programmable by means of a debugger tool, whereas assertion-checkers are usually hardcoded. During the validation of a complex system which includes multiple-cores, we need to trace the status of assertion-checkers placed inside cores. Therefore, a debug infrastructure must be equipped with an enhanced debugging module which makes the output of assertion-checkers transparent to a debugger tool. As illustrated in Fig. 1, once the proposed infrastructure is interfaced with the CTI and ETM, it can be used as a trigger unit inside a system.

A trigger generator tool, ZiMH, is presented in [7]. This tool exploits the hierarchical properties of a system as a means to generate a trigger circuit. The generated circuit provides instrumental trace information for root cause analysis. However, this study does not provide a solution for incorporating this trigger unit inside a system. A so-called assertion processor, along with synthesized assertions, is incorporated on a chip in [13,14]. These studies neither provide coverage metrics nor an automated method for integrating assertion-checkers inside a design. The authors in [16] exploit the fact that it is not necessary to observe the error-free state. Instead, they have introduced the "suspect window" and presented a method for determining its boundaries.

The integration of assertion-checkers in a scan-based run-stop debug infrastructure and in a debug trace infrastructure has been investigated in [3]. One conclusion of that work is that grouping assertion-checkers together and controlling each group through a single debug register results in a reduction in hardware overhead of debugging infrastructure, involved in transferring the violation signals of assertion-checkers to the trace-buffer. This study, however, provides no applicable solution for the clustering of related assertion-checkers. The work in [20] applies the time-multiplexing to a set of assertion-checkers in the debug infrastructure, which is related to the clustering in the sense that the checkers are grouped together in each time instance, but the clustering approach is not the focus of that work.

The integration of the proposed method in this paper in a form of hybrid HW/SW mechanism, such as a mechanism proposed in [31] can be instrumental as a means to reduce the effect of electrical errors in the SoCs, running Real-Time Operating Systems (RTOS). Moreover, Network on Chips (NoCs) routers should also be aware of the importance of design errors and faults [32–34], making the incorporation of debug infrastructures inside such routers an instrumental technique [29,30].

In this work, we present a mechanism to group assertion-checkers and place them inside clusters. The assertion-checkers can be efficiently integrated into a debug circuitry by means of our proposed mechanisms. Plus, the proposed debug environment in this paper addresses the reusability needs of SoC debugging.

3. Preliminaries

3.1. Assertions

An assertion is a statement that indicates how a given circuit should behave under different circumstances. Assertion-Based

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016

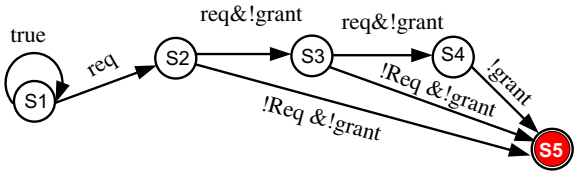

Fig. 2. Generated automaton from the SVA assertion.

Verification (ABV) has become one of the most important and efficient RTL verification techniques, and has gained a lot of attention in industry for pre-silicon verification [2].

Assertions represent a complex range of behaviors. System designers are able to define both expected and prohibited behaviors of a design using a wide range of Boolean expressions along with extended regular expressions and a large set of temporal operators. Verification languages such as PSL (Property Specification Language, IEEE 1850 standard) [22] and SVA (System Verilog Assertions) support assertions and enable Assertion Based Verification [23].

Assertion languages have a complex set of syntax and semantics that are beyond the scope of this paper. To demonstrate how an assertion works, an example of an Assertion (A1) written in PSL is:

$$A1 = \text{assert always}(\$\text{rose}(\text{req}) \mid\Rightarrow \text{req}[^{*}0:3]; \text{req\&grant}); \quad (1)$$

This assertion monitors the functionality of an arbiter. It states that the arbiter is expected to grant the bus to the client whose request signal 'req' is active within three clock cycles. The client must also keep its request signal active until it receives the 'grant' signal. The 'grant' signal indicates that access to the bus is given to the client. This assertion will fire if either the client or the arbiter cannot satisfy one of the previously mentioned conditions. The operator '|⇒' is a temporal implication, with pre and post conditions appearing as the antecedent and consequent, respectively. The function 'rose(b)' becomes **true** in case of any changes in the signal 'b' that make it '1'. In this example, the post-condition involves two sequences that are concatenated by means of a temporal concatenation ';'. The first sequence is a repetition range, whereas the second sequence is a Boolean expression.

3.2. Checker generator

Checker generator is a tool for producing assertion-checkers. An assertion-checker is a synthesized form of assertion(s). Assertion-checkers are permanent circuits that can be added to the design in order to perform on-line silicon monitoring, self-test and diagnosis assistance during the lifespan of the IC [8,12]. Here, we use the tool MBAC for checker generation [8]. The MBAC checker generator matches each assertion statement with its related automaton, either by a direct optimized production, or by applying a set of rewrite rules [9]. Thereby, various automata for properties and sequences are generated using this checker generator.

A generated automaton is a directed graph, in which vertices are the states and edges among states hold conditions for transitions among the states. Fig. 2 shows the generated automaton from the SVA assertion in Eq. (1). Transitions are labeled with Boolean expressions, built of combination of signals involved in the property. It has been shown in [8] that every property in PSL and SVA can be converted to an equivalent finite automaton in a recursive manner. An assertion violation signals triggers whenever an automaton representing an assertion reaches its final state. For instance, the violation signal of our sample assertion is triggered, once its automaton in Fig. 2 reaches the final state 'S5'.

In the pre-silicon verification, employing a large number of assertions is not a big issue. But, when it comes to the post-silicon verification, the situation is utterly changed. Given the fact that assertions are synthesized to hardware units during the post-silicon verification, the related hardware overhead and energy consumption should be acceptable.

3.3. Netlist graph

Due to an abundant use of memory elements such as flip-flops in industrial circuits, once a bug occurs errors will be recorded in some flip-flops [16]. Therefore, to capture a bug, it is sufficient to monitor flip-flop outputs during the debug. Fig. 3 shows a sample circuit and its corresponding ***netlist*** *graph*. Let *CUDG* = (*V*,*E*) be a

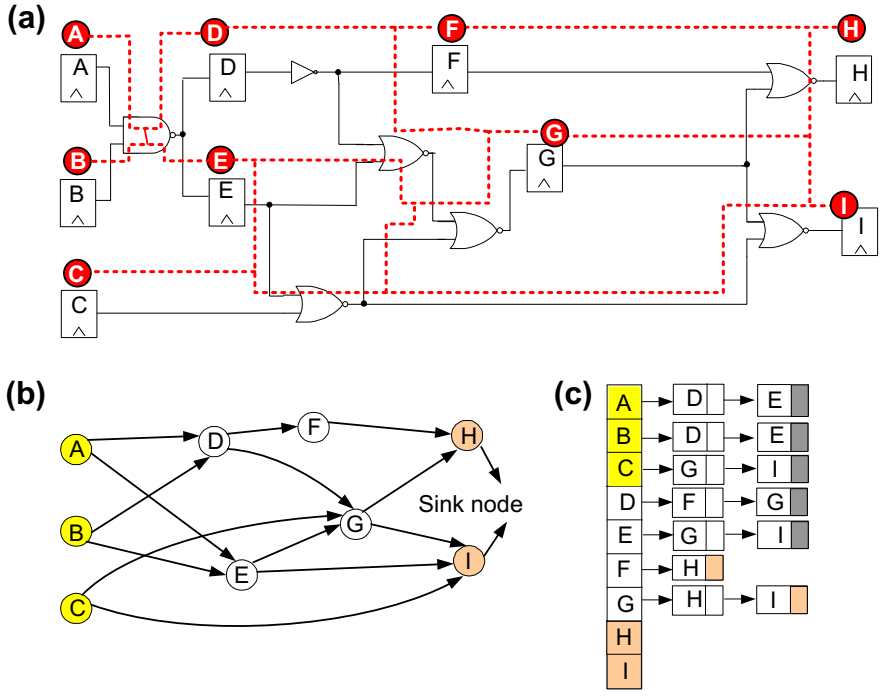

Fig. 3. Creating a graph from a given circuit under debug; (a) gate-level netlist, (b) generated graph, (c) adjacency list.

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016

directed graph associated with the given circuit netlist. Every vertex $v_i \in V$ in this graph is related to a flip-flop in the circuit netlist. The combinational parts of a circuit among storage units are represented by edges. For instance, there is an edge among vertices D, G and F in Fig. 3a. The vertices associated to primary outputs $\{H, I\}$ are marked as ***"sink node"***.

3.4. Definitions

Definition 1. Let $G = (V, E)$ be the "Fan-in Cone Graph" of a primary output. This graph is directed and each vertex $v_i \in V$ represents a storage element (Flip-flop) inside a given circuit. Let e_{ij} be a directed edge from the vertex v_i to v_j in this graph, any changes to the storage element that corresponds to v_i can modify the storage element related to v_j in the next cycle. In this graph, the node associated with the primary output is called the **"sink"** node. To extract the "**Fan-in Cone Graph"** of a particular output, the given netlist graph is traversed starting from its **"sink"** node using "Depth-First-Search" **(DFS)** algorithm.

Definition 2. Let $G^*_{o_i} = (V, E)$ be a "Weighted Fan-in Cone Graph" of a primary output o_i. This graph is a weighted directed graph generated from the "Fan-in Cone Graph" of the primary output o_i. The weight of each directed edge is placed on its head node. As shown in Fig. 4, the weight of $v_i \in V$ denoted by $w(v_i)$ shows the number of paths from v_i to the "sink" which is the vertex related to the o_i. The set of vertices adjacent to the $v_i \in V$ is denoted by adjacent-set (v_i). The number of edges that leaves the given vertex $v_i \in V$ is denoted by out-degree (v_i). As shown in Eq. (1), the weight of the sink node is equal to "1"; the weight of other vertices is computed by means of Eq. (2).

$$w(sink) = 1 \tag{2}$$

$$w(v_i) = \max(out - degree(v_i) + \sum_{vj \in adjacent-set(v_i)} w(v_j)) \tag{3}$$

Definition 3. We define the concept of "**Fan-in cone coverage of a primary output with respect to a vertex"** denoted by $Cov(G^*_{o_i}|v_i)$, where $v_i \in V$ is a vertex in the "Weighted Fan-in Cone Graph" of the primary output o_i. As Eq. (3) shows, this term denotes the number of paths which are covered by monitoring the particular vertex v_i over all available paths to the sink node (o_i).

$$Cov(G^*_{o_i}|v_i) = \frac{w(v_i)}{\sigma_{v_k \in V} w(v_k)} \tag{4}$$

Definition 4. A Finite Automaton (FA) associated with an assertion-checker is a tuple $FA = (Q, \Sigma, \delta, I, F)$, where '$Q$' is a nonempty finite set of states, 'Σ' is a set of symbols that represent Booleans expressions and signals such as primary inputs, outputs and the intermediate signals inside a CUD. In this FA, $\delta \subseteq Q \times \Sigma \times Q$ is a transition function consisting of a subset of triples from $\{(s, \sigma, d) \mid s \in Q, \sigma \in \Sigma, d \in Q\}$. As explained in Section 3.2, the MBAC checker generator synthesizes assertions by means of assigning an FA to them [8]. It is important to note that, in order to find the **"fan-in cone set"** of each assertion-checker, the fan-in cone graph of each primary outputs should be explored. As explained in Definition 1, given the fact that $G_{o_i}(V, E)$ be the "Fan-in Cone Graph" of a primary output o_i, where, each $e_{ij} \in E$ represents a directed edge from the vertex v_i to v_j. This directed edge denotes the existence of a combinational unit among the storage elements associated with v_i and v_j. It was shown in Definition 4 that transitions from different states inside an assertion-checker occur due to a change in the signals that are elements of the set Σ. Such a set consists of the signals and Boolean expressions.

Definition 5. Let $CH_{(i|o_j)}$ be the fan-in cone set of assertion-checker$_i$ with respect to the primary output o_j, where $CH_{(i|o_j)} \subseteq G^*_{o_j}(V)$. The set of vertices inside the weighted fan-in cone of the primary output o_i is denoted by $G^*_{o_j}(V)$. The vertices in this set that may cause changes in the state of the FA associated with the particular assertion-checker$_i$ are placed inside its fan-in cone set w.r.t the primary output o_j. As shown in Eq. (4), the union of $CH_{(i|o_j)}$ over all primary outputs is the "**Fan-in cone set of the assertion-checker$_i$**" and is denoted by CH_i.

$$CH_i = \bigcup_{o_j \in primary\_outputs} CH_{(i|o_j)} \tag{5}$$

Definition 6. The **Maximum Coverage** of the assertion-checker$_i$ whose "Fan-in cone set" is CH_i is denoted by $Cov(CH_i)$. To compute the "Maximum Coverage of the assertion-checker$_i$", we resort to the "**Fan-in cone coverage of a primary output with respect to a vertex**", explained in Definition 3 and denoted by $Cov(G^*_{o_i}|v_i)$. The $Cov(CH_i)$ can be computed using Eq. (5). We can also use Eq. (6) to find the "Maximum coverage of an assertion checker with respect to a particular primary output".

$$Cov(CH_i) = \sum_{v_i \in CH_i} [\max[Cov(G^*_{o_i}|v_i)], \quad \forall o_j \in Primary\_outputs] \tag{6}$$

$$Cov(CH_{(i|o_j)}) = \sum [Cov(G^*_{o_i}]|v_i) \forall v_i \in CH_{(i|o_j)} \tag{7}$$

Definition 7. The **CM = (V, E)** is the "**Checker Map Graph**". This graph is undirected and weighted. There is a vertex $v_i \in V$ associated with each assertion-checker. The existence of common elements in the "fan-in cone set" of any pair of assertion-checkers is denoted by an edge between the corresponding vertices; the weight of this edge indicates the number of common elements in the "Fan-in cone set" of those two assertion-checkers.

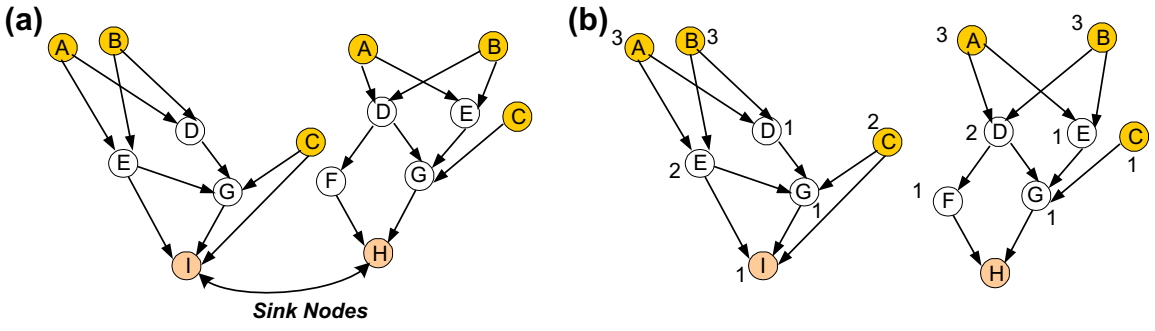

Fig. 4. (a) Fan-in cone graphs of primary outputs, (b) weighted fan-in cone graph of primary outputs.

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016

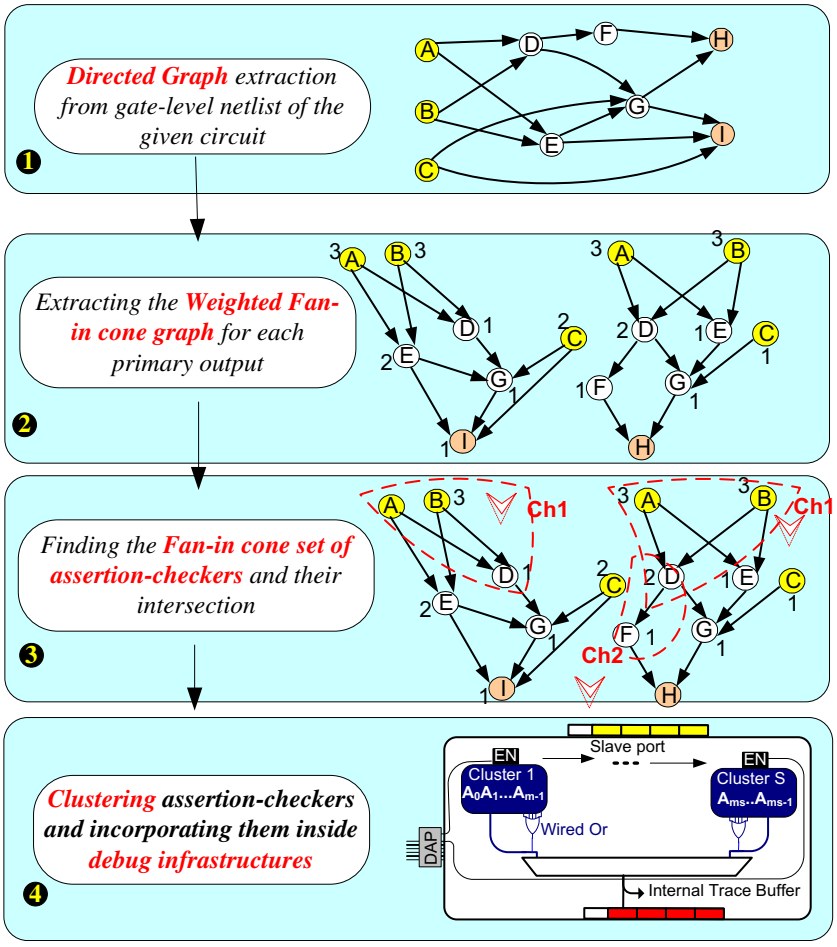

Fig. 5. Process of assertion-checkers clustering.

4. The proposed assertion-checker clustering algorithm

The proposed assertion-checkers clustering method, as shown in Fig. 5, consists of four steps. At the first step, a directed graph from the net-list of a circuit is created. As explained in Section 3.4, each vertex in this graph represents a storage element (Flip-Flop) inside the CUD. A directed edge between two vertices indicates that there exists a combinational logic or wire between the storage elements. The **Weighted Fan-in Cone Graph** for each primary output, Definition 2, is extracted in the second step in Fig. 5.

The Weighted Fan-in Cone Graph is generated from the "Fan-in Cone Graph" of a primary output. Inside the graph associated with a particular primary output, the weight of a vertex indicates the number of paths from that vertex to the primary output. The weighted fan-in cone graph of the primary outputs in the sample circuit in Fig. 3 is shown in Fig. 6.

As the left graph in Fig. 6 demonstrates, if a bug happens in the storage element related to the vertex "*A*", such a bug appears at the output "*I*" through three different paths. Likewise, the output "*H*" can be affected by two different paths once a bug occurs inside the storage element related to the vertex "*D*". Assume, for example, that the right graph in Fig. 6 corresponds to an arbiter, expected to provide the 'grant' signal. This 'grant' signal is connected to the combinational circuit among the vertices "*A*" and "*B*". In the right graph in Fig. 6, there are two different paths, *P1* and *P2*, in which a bug can reach the output.

The next step in the clustering finds the Fan-in cone set of assertion-checkers. Having produced the Weighted Fan-in cone graph of

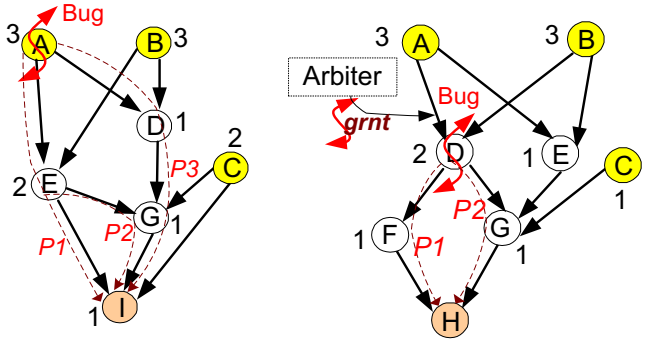

Fig. 6. Weighted fan-in cone graph of primary outputs.

primary outputs, we can obtain the Fan-in cone set. Fig. 7 illustrates the fan-in cone graphs of primary outputs and assertion-checkers inside the example CUD. Every superimposed dashed area in this figure represents an assertion-checker. Dashed lines bound the storage elements that may impact assertion-checkers output. As per Definition 4, the vertices in the "Fan-in cone graph" of primary outputs that lead to a transition to the states corresponding to an assertion-checker are placed in its Fan-in cone set. An assertion-checker can be influenced from the vertices placed in different "Fan-in cone graphs". For that, we make use of the "Fan-in cone set of an assertion-checker with respect to a primary output".

For instance, the assertion-checker 1 "Ch1" in Fig. 7 can trigger due to the changes in the storage elements associated with vertices

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016

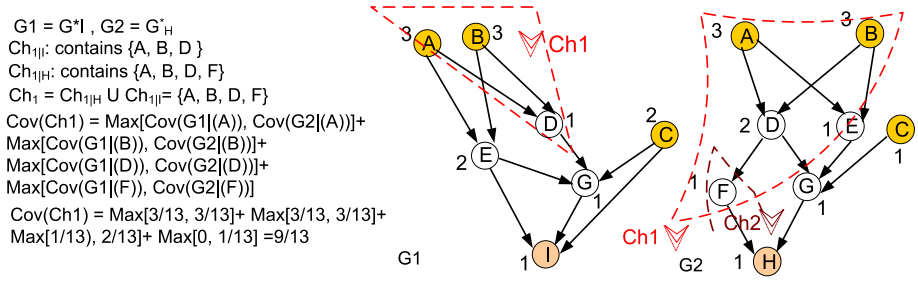

Fig. 7. Fan-in cone set of assertion-checkers and their maximum coverage.

```
/* Inputs: Maximum number of clusters "Max_Cluster", maximum
number of checkers inside each cluster "Max_Checker", and the
checker map graph "CM" created in the previous step */

//The maximum number of checkers inside each cluster
// CM (V, E) = CheckerMap (V,E)
//CheckerMap is a weighted graph

Cluster-Generator(Max_Cluster, Max_Checker, CM)
 //Proposed Cluster Generator algorithm
// CM is the Checker Map Graph
1. Cl_Count := |CM(V)|;
//|CM(V)| is the number of vertices in CM graph
2. While (Cl_Count ≥ Max_Cluster)
{3. Find the heaviest e_i ∈ CM(E), where (e_i.visited = false)
  // that edge must have not been visited yet
 4. if ((e_i.v_L.cur_checkers+ e_i.v_R. cur_checkers) ≤ Max_Checker)
  /*Check whether by merging the vertices connected to e_i the
number of checkers exceeds the maximum number of permissible
assertion-checkers in a cluster*/
        {5. Merge_Update(e_i.v_L, e_i.v_R, CM(V,E) );
        //merge the vertices connected to e_i
          6. Cl_Count--; }
   7. e_i.visited = true;
}
```

```
/* Inputs: A modified Checker map graph "CM"
Output:Updated CM graph */

1. CM ( V, E) = Modified CheckerMap (V,E)
//CheckerMap is a weighted graph

Merge_Update( v_L, v_R, CM(V,E) )
{

2. Add v_new to CM(V)
// Add a new Node to the CM graph
3. For all edge_i ∈ CM( E )
  {4. If v_L or v_R is connected to edge_i
   5. Disconnect edge_i from v_L or v_R
   6. Connect edge_i to v_new }
7. If more than one edge connect two vertices
   {8. find the maximum weight among these edges
    9. replace them with one edge
    10. assign the maximum weight to the new edge}
11. Remove v_L, v_R from CM(V)
}
```

Fig. 8. Cluster generator algorithms.

{A, B, D} and {A,B,D,F} located in the "Weighted fan-in cone graphs" of the primary output "*I*" and "*H*", respectively. Hence, the Fan-in cone set of this assertion-checker with respect to "*I*"/"*H*" denoted by $Ch_{1|I}/Ch_{1|H}$ is {A,B,D}, {A,B,D,F}, respectively. As per Fig. 7, the Fan-in cone of the assertion-checker1 denoted by Ch1 is the union of $Ch_{1|I}$ and $Ch_{1|H}$, i.e., {A,B,D,F}. The maximum coverage $Cov(Ch1)$ is computed using Eq. (5). Likewise, the $Cov(Ch2)$ is obtained as $Max[Cov(G1|'F'), Cov(G2|'F')] = Max\ [0,1/13] = 1/13$.

Having specified the "Fan-in cone" set of assertion-checkers, in the next step, we place such assertion-checkers into clusters using a graph partitioning algorithm. Here, we make use of the **CM** (Checker Map) graph presented in Definition 7. As it was explained in Section 3.4, this graph is a weighted graph. The weight of the edge e_{ij}, connecting the vertices v_i and v_j, indicates the number of common elements in the "Fan-in cone set" of the assertion-checkers corresponding to the v_i and v_j, respectively. For instance, the CM graph for the circuit in Fig. 7 has two nodes {a1,a2} that are connected using an edge with the weight "1". Fig. 8 shows our proposed algorithms to create clusters of assertion-checkers based upon a Checker Map graph. The Cluster-Generator needs to continuously update the given CM graph. The update procedure is shown on the right-hand side of Fig. 8.

This algorithm takes as inputs the CM graph, the maximum number of clusters allowed to be placed inside a debug infrastructure, denoted by "Max_Cluster", and the maximum number of assertion-checkers that can be placed inside a cluster, marked by "Max_Checker". In other words, the number of clusters that this algorithm can produce cannot exceed the "Max_Cluster". This algorithm should also consider "Max_Checker" as the maximum number of assertion-checkers allowed to be placed inside each cluster.

As shown in Fig. 9, the edge with the heaviest weight will be selected at each step. The salient property of this scheme is that the larger the weight of an edge, the higher the probability of the violation in the related assertion-checkers and the chance of extracting the required debugging details to spatially isolate the candidate error sites.

Once an edge with the heaviest weight is found, two nodes connected by this edge are chosen as a candidate to merge. Thereafter, the partitioning procedure checks whether by merging related nodes the maximum number of assertion-checkers exceeds. For example, since the weight of the edge between a1 and a2 is larger than that of the others in Fig. 9B, these two nodes will be merged together. To combine these nodes, we have to ensure that the number of elements in the new cluster {a1,a2} is smaller than the maximum

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016

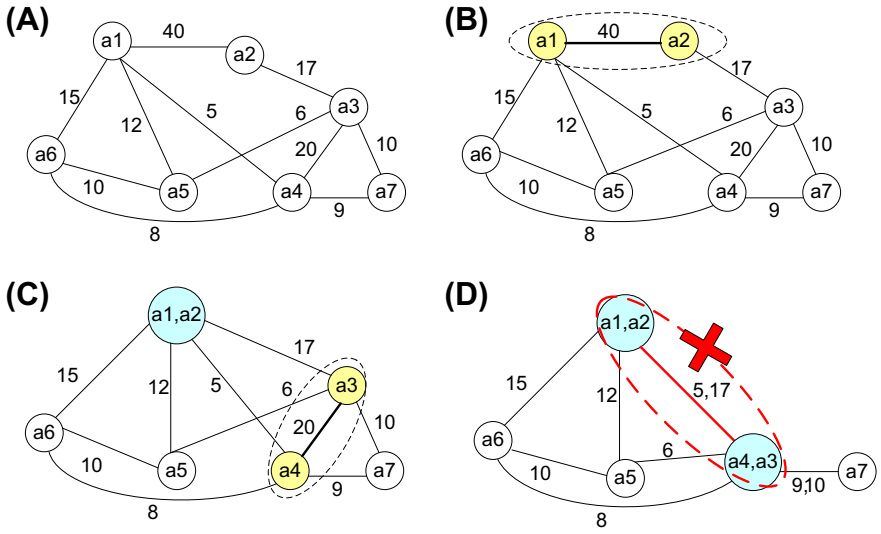

Fig. 9. Cluster generation on the sample CM (Checker Map) graph.

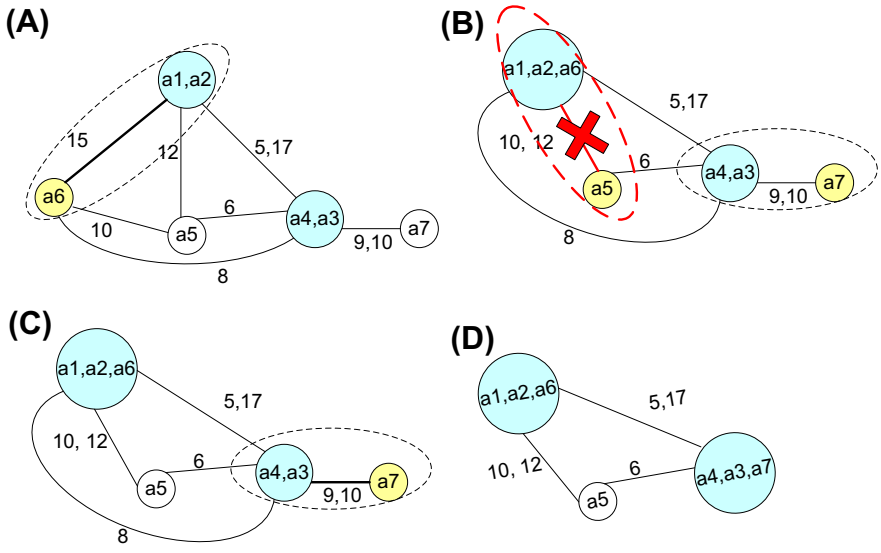

Fig. 10. Cluster generation on the sample CM.

number of allowable elements in each cluster. After merging these nodes, the algorithm should update the CM graph. To update the CM graph, any edge connected to the vertices "a1" or "a2", should go to the new composite node or cluster {a1,a2}. Having updated the CM graph, the iterative partition algorithm continues by merging the node "a3" and "a4" as in Fig. 9C. In the next iteration, Fig. 9D, the edge with the largest weight is selected again. However, since after merging two concerning clusters {a1,a2}, {a3,a4} the number of elements in the new cluster exceeds the maximum number of allowable elements, the "Cluster-Generator" algorithm refuses to merge these two clusters. Consequently, the next largest edge is selected as shown in Fig. 10A. The partitioning algorithm based on the merge and update procedure continues until it creates the demanded number of clusters. The final clusters obtained by applying the iterative partitioning algorithm is shown in Fig. 10D, where there clusters of assertions-checkers are created. After obtaining clusters of assertion-checkers, we have to incorporate them into the debug infrastructure inside a CUD.

5. How to use the clustering algorithm and how to obtain coverage metrics for clusters

To allow corrections of silicon bugs or to bypass faulty modules, reconfigurable elements or programmable-logic fabric are increasingly being placed into ASICs [20,21]. Such reconfigurable units can be used to implement debugging circuitry. Example of an SoC containing reconfigurable elements is shown in Fig. 11. As this figure demonstrates connections to the reconfigurable fabric are not shared uniformly among cores. In other words, in a typical SoC design, various Intellectual Property (IP) cores have different levels of trust. For instance, IP cores provided by third vendors with the prior successful tape-outs are considered more trustable than a new developed IP core [11]. Therefore, reconfigurable resources as a means to correct and bypass errors are dedicated in a non-uniform fashion among cores.

For example, Core3 and Core4 shown in Fig. 11 might have been used previously or taken from a third vendor; thus, a limited

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016

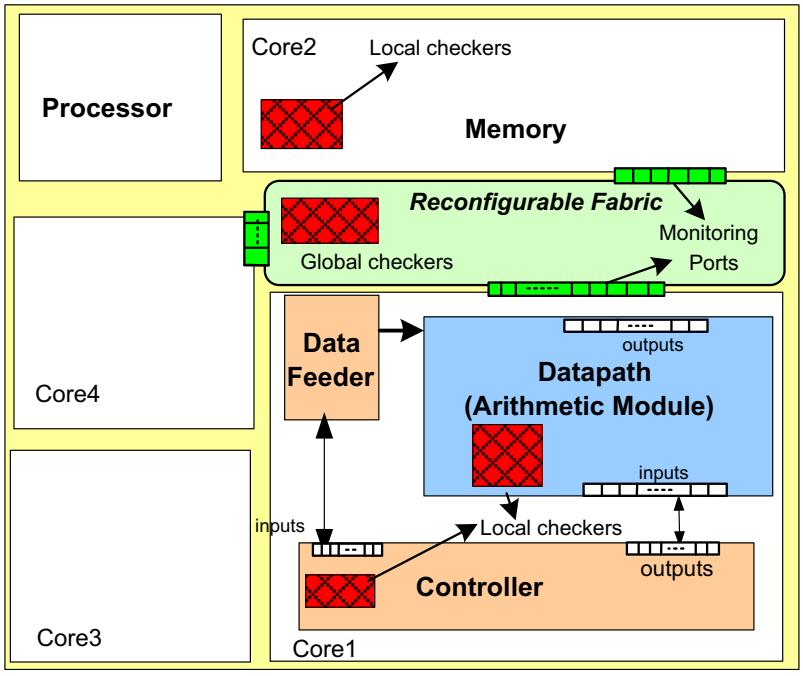

Fig. 11. Typical soc floorplan containing reconfigurable fabrics.

// this algorithm returns the coverage of a given cluster
// The inputs to this algorithm
//inputs: Cluster$_I$, the Maximum number of monitoring points: Max_MointorPoints
Cluster_Coverage (Cluster$_I$, Max_MointorPoints) {

1. *Avail_MonitorPoints = Max_MointorPoints;*
2. *Current_Coneset = ∅;*
3. *ClusterCoverage = 0;*
4. *While Avail_MonitorPoints >0 {*

4.1. Select a checker ch_i that has the Maximum Coverage among all the checkers in ClusterI

4.2. Remove Ch_i from ClusterI

4.3. Current_Coneset = Current_Coneset ∪ $Fanin\_ConeSet(Ch_i)$

4.4. If ($|Fanin\_ConeSet(Ch_i)| < Avail\_MonitorPoints$){

4.4.1. Avail_MonitorPoints = Avail_MonitorPoint – $|Fanin\_ConeSet(Ch_i)|$

4.4.2. ClusterCoverage = ClusterCoverage+$Cov(Ch_i)$

4.4.3 For all checker $Ch_k \in$ ClusterI

4.4.3.1 If ($Fanin\_ConeSet(Ch_k) \subseteq Curent\_Coneset$) {

4.4.3.1.1. ClusterCoverage = ClusterCoverage+$Cove(Ch_k)$

4.4.3.1.2. Remove Ch_k from Cluster

}//4.4.3.1

}// 4.4

Fig. 12. "Cluster_Coverage" algorithm: compute the maximum coverage of a cluster.

number of monitoring points are shared with the reconfigurable fabric, whereas a larger number of monitoring points are assigned to Core1 and Core2 which are new developed IPs. A debug circuitry built into a reconfigurable fabric can communicate with a CUD by means of monitoring points.

Although the main purpose of embedding programmable logic cores on SoCs is to provide post-fabrication flexibility for the design, such programmable cores are the best candidates to host assertion-checkers. However, when it comes to incorporating assertion-checkers into programmable modules, we have to be

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016

aware of the silicon area constraints. It is important to note that the "Cluster-Generator" algorithm shown in Fig. 8 can be easily modified to consider the area constraints. In particular, the area constrain should replace the "Max_Checker" in the "Cluster_Generator" algorithm shown in the left hand side of Fig. 8.

A wide range of assertion-checkers in IP cores are typically utilized to monitor the local properties. Such assertion-checkers, as shown in Fig. 11, are typically laid inside the cores. Global assertion-checkers of an SoC as a means to monitor interaction among cores are built into the reconfigurable fabrics.

It is important to consider that to cluster local assertion-checkers inside a core using the proposed "Cluster_Generator" algorithm in Section 4, the input and outputs of that particular module should be considered as primary input and outputs. For example, to cluster the local assertion-checkers inside the "Arithmetic" module in Core 1, shown in Fig. 11, the netlist graph among the inputs and outputs of this module should be generated.

The fan-in cone set and the maximum coverage of each assertion-checker are explained in Definitions 5 and 6, respectively. Once assertion-checkers are placed inside different clusters and a list of available monitoring points is specified, we can find the maximum coverage of each cluster. A monitoring point is a place which can be observed by a debug circuitry through a monitoring port. The maximum coverage of each cluster based on the coverage of assertion-checkers integrated into that cluster and the maximum number of monitoring points can be computed using the algorithm presented in Fig. 12.

6. Integration of assertion-checkers in a circuit under debug and experimental results

To integrate clusters of assertion-checkers into a debug infrastructure, two key issues should be resolved. First, the way that such clusters can be accessed needs to be defined; secondly, a mechanism through which the violation signals of these clusters can be transferred outside to a debugger tool must be established. Existing on-chip debug solutions, such as a scan-based run-stop debug and a debug trace infrastructure must be equipped with clusters of assertion-checkers. By incorporating clusters of assertion-checkers in existing debug infrastructure, we can ensure compatibility and reduce the impact on the debug tool support. In this section, we will show how clusters of assertions-checkers can be incorporated in a scan-based run-stop and a trace-based debug infrastructure.

6.1. Integration of clusters in a scan-based run-stop debug infrastructure

Having partitioned assertion-checkers based on their fan-in cone sets, we have to incorporate them inside a debug infrastructure. Fig. 13 illustrates how such clusters can be integrated into a scan-based debug infrastructure. The TAP controller is compliant with JTAG (Joint Test Action Group) IEEE Specification 1149.1. The controller manages the debug environment via instructions and data transfers to the on-chip registers from an external debug host. Moreover, it provides control to all user-defined debug circuits.

In our method, once an assertion-checker inside a cluster triggers, that cluster informs the TAP controller by raising an interrupt signal. The CUD stops working and switches to the debug mode; consequently, an external debugger connected to the system via the TAP port can scan-out the chain of debug status registers and check the state of the corresponding clusters. A Cluster Status Register (CST) is associated with each cluster. This register is in charge of holding the status of the assertion-checkers. As shown in Fig. 13, the size of this register is equal to the number of assertion-checkers inside a cluster. The violation signals of the assertion-checkers placed inside a cluster must be stayed active to make sure that an external debug tool can access them.

As a means to control clusters, we equipped them with an enable register. The TAP controller in Fig. 13 activates each cluster through the chain of EN registers. It provides the required flexibility to enable or disable a particular assertion-checker cluster. Plus, clusters are able to transform their violation signals by means of CST registers, which are daisy-chained together.

The first disadvantage of incorporating assertion-checker clusters into a scan-based run-stop debug infrastructure is that scanning out the serial debug status register is slow. JTAG is not a fast serial interface (the upper limit of transfers is typically less than 100 MHz) and was not designed to support data transfers for any real-time analysis, so it provides limited bandwidth [3,19]. The inability to transfer vast amounts of trace data off-chip without significant slow-down impedes the debugging of a design.

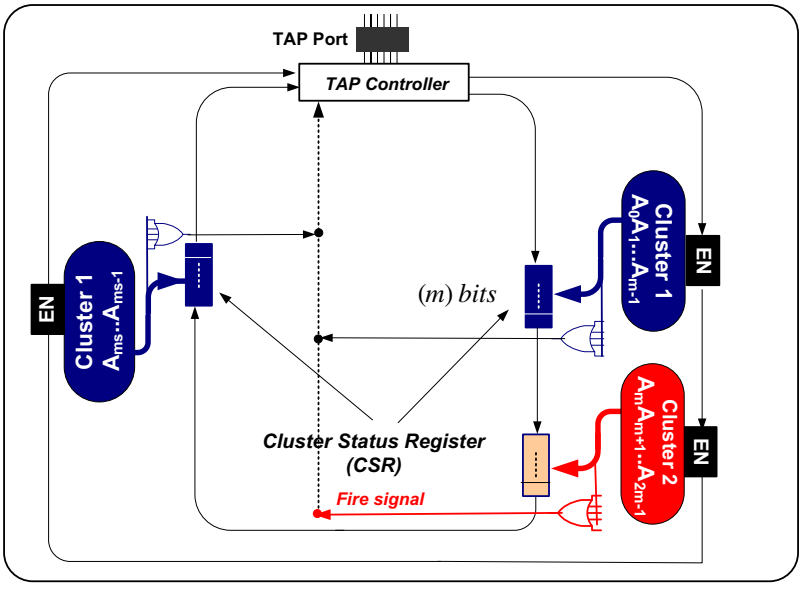

Fig. 13. Integration of the assertion-checker clusters inside a scan-based debug infrastructure.

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016

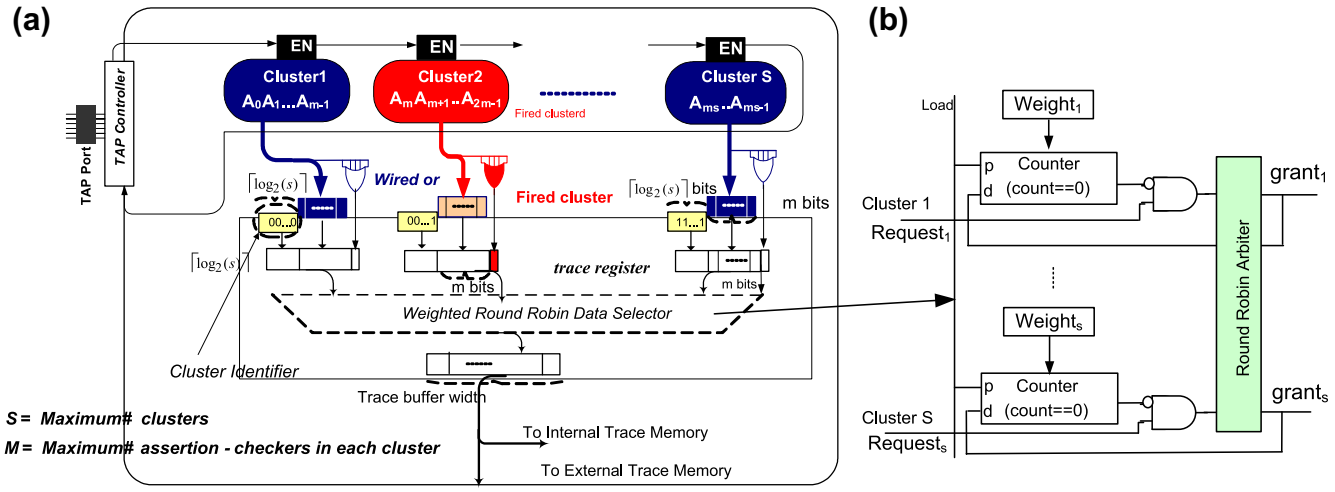

Fig. 14. (a) Integration of the assertion-checker clusters into a real-time trace-based debug infrastructure, (b) weighted Round Robin (WRR) arbiter.

Secondly, once a debug infrastructure switches to debug mode no longer can serve to other clusters. Since a debug session may take up to thousands of clock cycles, the cluster containing the assertion-checkers related to the other parts of a design stay idle for a large period of time; furthermore, an assertion-checker must keep its violation signal active until it gets captured, leading to an inability to detect multiple failures in the same assertion-checkers.

6.2. Integration of clusters in real-time trace-based debug infrastructures

In a real-time trace-based debugging scheme, currently being used in commercially available ICs such as ARM family [24], embedded memories are used as a means to record and trace signals. That leads to the higher observability in designs and allows embedded software to execute at-speed while transparently logging debug events.

As mentioned before, one limitation of incorporating assertion-checkers cluster into a scan-based run stop debug infrastructure is that an assertion-checkers, placed inside a cluster, should keep its violation signals active until it become processed by an external debugger; hence, multiple violations of the same assertion-checker are not detectable. For instance, since the overlapped sequences of events that lead to a failure cause consecutive violations in the assertion-checker controlling such sequences, it is not possible to detect such failures by means of the chain of clusters ruled by a TAP controller.

Debug trace infrastructure can be used effectively as a means to log more accurately assertion-checker clusters status. In other words, embedding violating assertions into the debug trace makes it possible to trace the status of assertion-checkers per clock cycle. Therefore, such a debugging scheme allows logging of multiple violations of the same assertion-checkers. However, due to the limited width of the debug trace channel, we have to provide a mechanism to effectively store clusters information. Fig. 14a shows our method to integrate clusters into real-time debug trace infrastructure. The "*Cluster-Generator*" algorithm from Section 4 determines which assertion-checker belongs to which cluster. It is important to note that the value of '*S*' is the maximum number of affordable clusters, and '*M*' is the maximum number of assertion-checkers that can be placed in a cluster.

A unique cluster identifier has been assigned to each cluster. Once an assertion-checker inside a cluster gets fire the debug infrastructure should transfer related detail to a trace buffer. As Fig. 14 shows, the "wired-or" signal of a cluster gets triggered as soon as the assertion-checker(s) hosted by it fire; thereafter, the status of all assertion-checkers inside that particular cluster will be copied to the trace register. The data that needs to be transferred to the trace buffer is the *Cluster's Identifier* and the *Cluster Status Register* (CSR) containing violation signals of the assertion-checkers placed inside the fired cluster; the former requires bandwidth of *M=Max_Checker* bits, while the latter needs $S = [\log_2(Max\_Cluster)]$ bits.

For example, as shown in Fig. 14a, once one of the assertion-checkers in the cluster 2 triggers, the related cluster identifier along with the violation information of that cluster are placed inside the trace-register to be stored into the embedded trace memory. When a trace buffer width is larger than the number of clusters, multiple CSRs can be stored at the same cycle on the debug trace. In the inequality given in Eq. (7), N is the total number of assertion-checkers and M is the maximum number of assertion-checkers that can be placed inside one cluster and C is a number of trace registers that can be placed at the same cycle into the debug trace data.

$$Trace\ Buffer\ Width \geqslant C \times \left\lceil \log_2\left(\frac{N}{M}\right)\right\rceil + C \times M \tag{8}$$

When more than one cluster wants to place its information inside the trace register, the Weighted Round Robin data selector assigns the trace registers to clusters based upon a fixed priority. Because of the Round Robin data selection scheme, once a cluster reports its information and unique cluster ID its priority decreases. This data selection scheme reduces the delay between the time that an assertion-checker gets fired and the time that information of that particular cluster is reported. Thereby, it becomes easier to distinguish the root cause of an error during the offline processing of trace data. As Fig. 14a shows, the TAP controller can be effectively used to control each cluster through the enable registers that are chained together.

6.3. Weighted Round Robin (WRR) arbitration mechanism

While a part of a design is under debug, the assertion-checkers responsible to monitor that particular module are expected to be exercised more. In addition, the larger the number of assertion-checkers inside a cluster, the more grants signals that cluster requires. Therefore, arbitration mechanism among clusters should perform unfairly. Fig. 14b shows a weighted round-robin arbiter used to carry out arbitration among clusters. A weight w_i is assigned to each cluster "i" that indicates the maximum fraction

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016

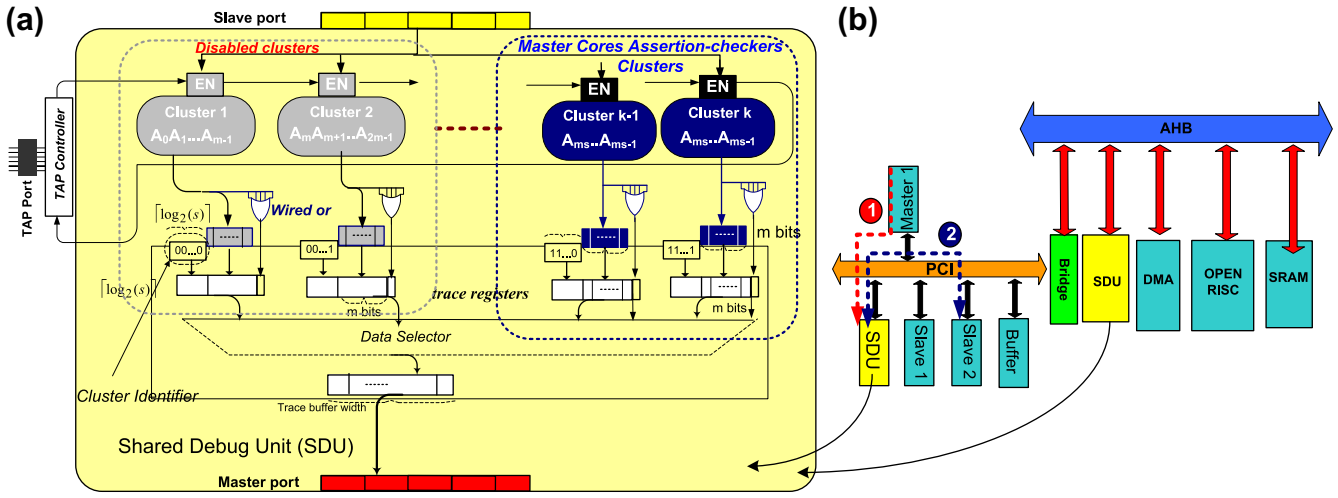

Fig. 15. (a) Shared Debug Unit (SDU): a debug environment suited for SoCs, (b) integration of SDU into a SoC based platform.

"f_i of grants that cluster i is supposed to receive according to $f_i = \frac{w_i}{W}$, where $W = \sum_1^S w_i$.

The higher the number of assertion-checkers inside a cluster the larger is its fraction of grants. As shown in Fig. 14b, each time a cluster receives a grant, the counter is decremented. As soon as the counter associated to a particular cluster reaches to "zero", that cluster no longer is able to activate its request line. The load line will be activated periodically in every W cycle. The counter associated to each cluster is loaded with the previously assigned weight when the load line is asserted.

6.4. Integration of clusters in a Shared Debug Unit (SDU)

Modern SoCs include many IP blocks and the interconnection networks have become one of the important components inside SoCs. As SoCs are getting more complicated, it has become more critical to monitor the interaction between multiple master and slave devices. However, conventional debug methods and tools tend to focus on the computational parts of a system, e. g. the processor and its interaction with the main memory. As different master and slave modules inside modern SoCs are connected by complex protocols, every module should be compliant with a list of rules specific to that protocol.

A wide range of assertion-checkers are needed to monitor the global properties of an SoC, such as hand-shaking protocols between master and slave cores, timing constraints for memory access, fair arbitration mechanisms among cores and others. A similar set of rules applies to devices that support the specific interface. Therefore, one of the primary concerns for the verification environment in charge of testing these standard protocols is reusability. Fig. 15a shows our proposed Shared Debug Unit (SDU) which is suited for compliance checking of standard protocols. The clusters inside SDU involve assertion-checkers related to different devices. For example, *Cluster k-1* and *Cluster k* in Fig. 15a are dedicated to a master core which is now being tested; alternatively, *Cluster 1* and *Cluster 2* involve the assertion-checkers of a device that is not under debug.

The SDU infrastructure should be equipped to control selectively each cluster of assertion-checkers, and it should be supplied by a mechanism to capture the violation signals of each cluster. The SDU can be configured by means of the slave port. In other words, the masters or slave devices sitting on the bus can reconfigure the SDU. Actions such as activating or deactivating particular assertion-checker clusters and changing the destination of trace buffer can be performed on the SDU by devices connected to the bus. Additionally, the SDU can benefit from the available observability on the main system bus for protocol checking and complaint testing. For example, to overcome the limited on-chip memory capacity, the SDU can be configured to serve as a new master and send its debug information through the master port to an external trace memory.

As shown in Fig. 15b, the SDU can be controlled either by the TAP controller or through the slave port by one of the Master devices in the system. By disabling the clusters containing the assertion-checkers of the devices that are not being tested, we can efficiently make use of the limited bandwidth of trace buffer.

Example of SDU's reusability is shown in Fig. 15b; in scenario 1, the SDU is configured by the Master1 to start debugging the Slave2. The clusters responsible to monitor the transactions related to the Slave2 will be enabled, while other clusters are disabled. Thereafter, the Master1 can start generating the transactions destined to the Slave2, which is currently the device under debug. In the second scenario, the Slave2 coordinates the SDU and configures it to debug the Master2.

6.5. Case studies

To verify the effectiveness of the proposed clustering algorithm, we have considered three case studies. We applied our proposed algorithm to cluster the assertion-checkers inside the case studies. In the following those case studies and their features will be discussed. Thereafter, we show how resorting to the clustering technique and the proposed method for incorporating such clusters inside debug infrastructures can be beneficial in terms of energy consumptions and the design coverage.

One of the major challenges in SoC designs has become compliance testing. It is very common for designs to support certain standard protocols. Therefore, we have considered the following standards to present the application of our method. We consider the following designs as our test cases:

1. AMBA 3 AXI bus protocol checkers adopted from ARM [25].
2. The PCI bus protocol checkers adopted from [26].
3. Memory Controller.

6.5.1. AMBA 3 AXI bus protocol checkers

The AXI bus protocol is an enhancement of the existing Advanced High-performance Bus (AHB) that is being used in high-performance systems [25]. AXI protocol has five independent unidirectional channels that carry the address/control and data.

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016

Table 1
AXI configuration settings.

Parameter	Value	Specification
DATA_WIDTH	64	Data bus width
ID_WIDTH	4	The number of channel ID bits required
MAXBURST	16	Size of Content Accessible Memory (CAM) for storing outstanding read burst
MAXWBURST	16	Size of Content Accessible Memory (CAM) for storing outstanding write burst
MAXWAITS	16	Maximum number of cycles between VALID → READY before a warning is generated

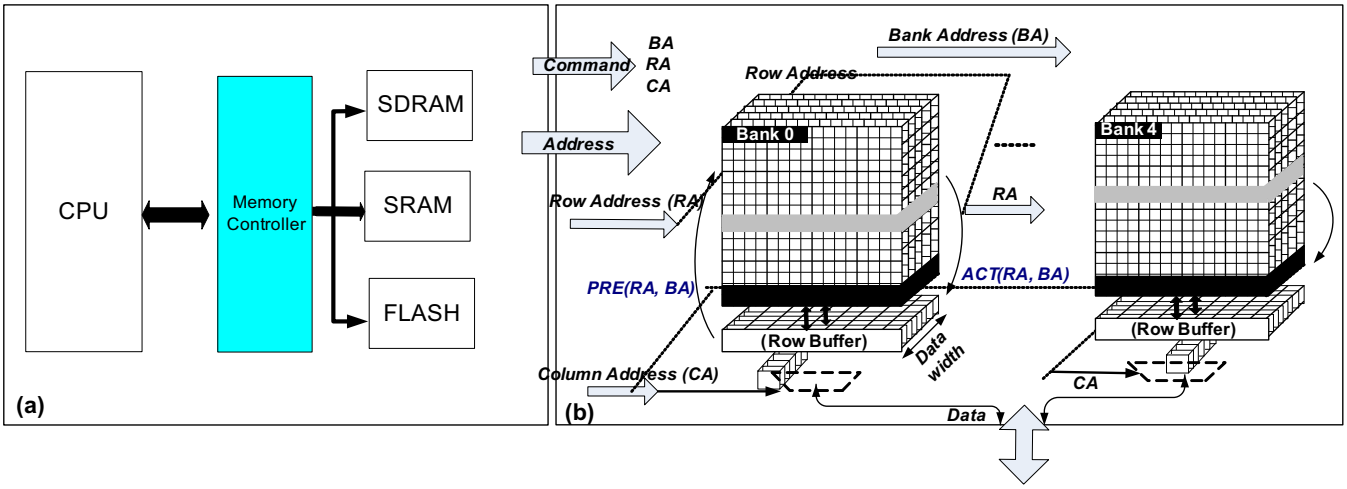

Fig. 16. (a) Memory controller, (b) SDRAM structure.

Each channel uses a two-way valid and ready handshake mechanism. The five independent channels are the Address-Read (AR) channel, Address-Write (AW) channel, Read-Data (RD) channel, Write-Data (WD) channel, and Write Response channel. The AW and AR channels convey the address and control for write and read transactions. Control signals describe the nature of transactions.

A transaction can be a burst of a different length, or it can be atomic. A burst is composed of a number of data transfers whose length is defined before. Masters and slaves communicate through the WD and RD channels. Write response channel (B) allows a slave to signal completion of a write transaction or an error. A support for the burst transaction with only the start address issued and split transactions that enable the out-of-order transaction completion are among other features of AXI. As AXI assigns an ID to each transaction, those with the same ID must be completed in order, otherwise the order is irrelevant. Out-of-order transactions improve system performance. A data item of an earlier access might be available from an internal buffer sooner than that of a later access (temporal locality). In experiments, we considered 154 assertion-checkers for AXI bus protocol taken from [25]. The configurable AXI settings include different data-bus widths and support for a varying number of outstanding transactions. In our experiments, we make use of the particular settings listed in Table 1.

6.5.2. PCI bus protocol checkers

The Peripheral Component Interconnect (PCI) bus is being used as an interconnection among high-performance peripherals such as network cards, sound cards, modems, extra ports such as USB or serial and other add-in boards. Although developed by Intel, it is not tied to any particular family of microprocessors [27]. The PCI local bus is a 32-bit or 64-bit bus with multiplexed address and data lines [27] that run at clock speeds of 33 or 66 MHz. For instance, the PCI bus can yield throughput rate of 264 MBps at 64 bits and 33 MHz. Although PCI bus is being replaced by PCI Express, most motherboards are still made with one or more PCI slots, which are sufficient for many uses. In our experiments, we have considered 40 assertion-checkers from [26] that monitor the properties of the PCI bus protocol and perform compliance testing for the devices connected to the bus.

6.5.3. SDRAM controller

There are a lot of timing parameters for SDRAM device and assertion based verification can be used effectively to verify these timing requirements. Fig. 16a shows a memory controller through which the processor communicates with SDRAM, SRAM and Flash memory. SDRAM, as one of the common complex slaves, provides high bandwidth by executing memory requests in parallel. As shown in Fig. 16b, SDRAM has a 3-D structure that involves banks, rows, and columns. Having multiple independent banks in a 3-D structure, enables memory scheduler to service serial requests in parallel; moreover, commands to different banks can be pipelined. The Address bus is divided into three parts: Bank Address (BA), Row Address (RA) and Column Address (CA). The BA specifies one of the banks inside an SDRAM, while the RA and CA points to a particular row and column on that bank. SDRAM controller accepts commands such as Activate (ACT), Read/Write(R/W) and Precharge (PRE). Different combinations of the SDRAM interface signals "sel", "ras", "cas" and "we" constitute the different commands. Taking the RA and BA, the ACT command activates a particular row (RA) inside the bank (BA) and transfers it to the row buffer of the bank after tRCD. The row buffer serves as a cache to reduce the latency of subsequent accesses. The PRE command gets the BA and after tPR copies the row buffer contents to its related row in the bank, and then makes the bank idle. The R/W command is executed only after a bank is activated and the row buffer contains the given row. After either the column access strobe latency (CL) or write latency (WL), the data goes from or to SDRAM. We consider a memory controller module adopted from *Gaisler IP-Cores* [28] and 38 assertion-checkers adopted from [25]. The 512 Mb SDRAM under verification is a quad bank SDRAM with a synchronous interface. Each bank is

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016

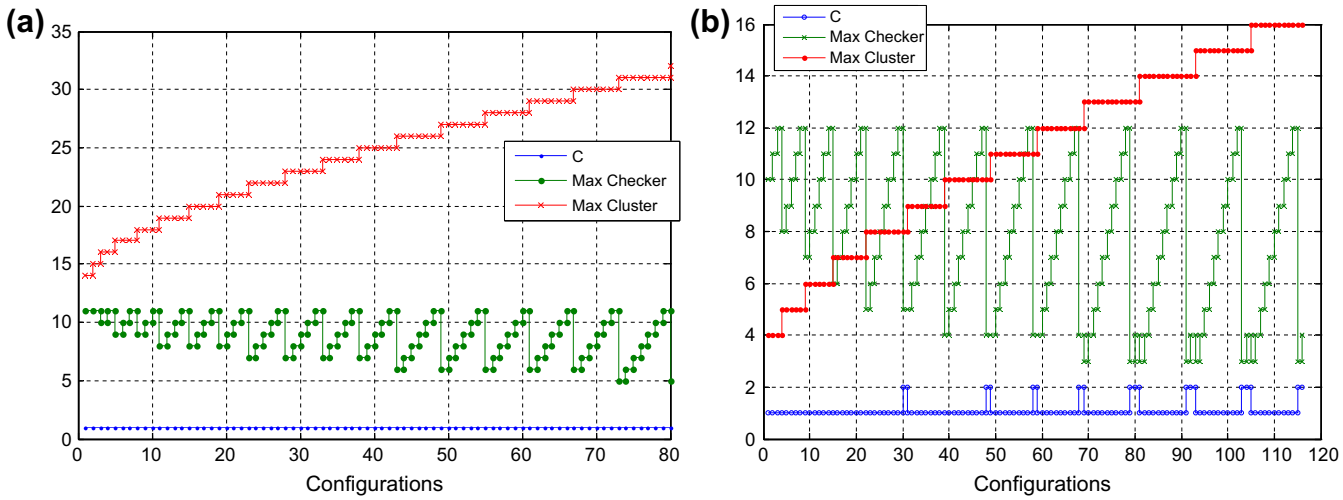

Fig. 17. Different arrangements for assertion-checkers related to (a) AXI bus protocol checkers, (b) SDRAM controller.

Table 2
Implementation results of incorporating assertion-checkers into our test-cases using a non-clustering method.

Test Cases	Number of assertions	Gate equivalent	Number of ports	Design area (μm^2)	Number of cells used from TSMC 65 nm library	Total power (mW)
AXI Bus protocol checker	154	7431	1290	10699.22	2763	2.46
SDRAM controller assertions	38	2705	47	3895.08	645	0.68
PCI bus assertions	40	6780	75	9762.16	1805	1.287

organized as 8192 rows $*$ 1024 columns $*$ 16 bits. Read and write access to the SDRAM is burst oriented.

6.6. Cost analysis of integrating assertion-checkers into test-cases

We have used *Synopsys Design Compiler* to synthesize our test cases and generate the gate level netlist. This tool first is employed to extract the netlist graph of our test cases. Then, the MBAC [8,12] was used to create synthesizable Verilog RTL modules from SVA assertions; consequently, such modules have been synthesized using *Synopsys Design Compiler*. In the next step, the CM graph is created by considering the assertion-checkers and designs' netlist graphs. The proposed clustering algorithm is invoked with the obtained CM graph along with the maximum number of clusters allowed to be built into a debug infrastructure denoted by "*Max_Cluster*", and the maximum number of assertion-checkers that can be placed inside a cluster marked by "*Max_Checker*".

Using the inequalities given in Eq. (8) and Eq. (9) as well as considering the width of trace buffer, it is possible to obtain a range of valid configurations for "*Max_Cluster*" and "*Max_Checker*". In the inequality of Eq. (8), "*C*" is the number of trace-registers that can be embedded into trace data.

$$Trace\ Buffer\ Width \geqslant C \times [\log_2(Max_{Cluster})] + C \times Max_{Checker} \quad (9)$$

$$Max_{Checker} \times Max_{Cluster} \geqslant Number\ of\ (assertion - checkers) \quad (10)$$

A valid configuration is denoted by (*Max_Cluster*, *Max_Checker*, *C*). In our experiments, the width of the trace buffer is assumed as 16 bits. With such an assumption, the assertion-checkers inside the AXI bus protocol checkers can be configured based on the following arrangements: {(14,11,1), (15,11,1), (16,10…11,1), (17,9…11,1), (18,9…11,1), (19,8…11,1), (20,8…11,1), (21,8…11,1), (22,7…11,1), (23,7…11,1), (24,7…11,1), (25,7…11,1), (26,7…11,1), (27,6…11,1), (28,6…11,1), (29,6…11,1), (30,6…11,1), (31,5… 11,1), (32,5…11,1)}. Fig. 17a plots these configurations. The *x*-axis in this figure represents a configuration number. Similarly, the set of valid configurations for the SDRAM controller is plotted in Fig. 17b.

To compare the effectiveness of the proposed clustering algorithm with the non-clustering scheme proposed in [3], we synthesized a large set of the assertion-checkers using Synopsys Design Compiler and the TSMC 65 nm technology library at supply voltage 1.00 V. Table 2 lists the resulting silicon area, number of ports and energy consumptions. The area usage is also reported in terms of Gate Equivalents (GEs), which is the number of 2-input NAND gates.

As listed in Table 2, the module that involves AXI protocol checkers has 1290 ports; given the fact that there is an output port associated to each assertion-checker, plus the number of assertion-checkers in this module is 154, the required number of monitoring ports is (1290–154) = 1136. Such a large number of monitoring ports result to a huge wiring overhead as well as increases in energy consumptions. In fact, as Table 2 presents, the debug module containing AXI bus protocol checkers consumes more energy than two other modules.

The debug modules in the SDRAM controller and PCI Bus protocol checkers contain 9 and 35 monitoring ports, respectively. As explained in Section 5, the maximum coverage of a design can be obtained once the number of available monitoring ports is specified. By assuming that a particular number of monitoring ports is available, we perform the design coverage analysis on case studies with different configurations. In our experiments, we suppose 32 available monitoring ports to the debug circuitry.

Fig. 18 plots the maximum design coverage achievable by the debug unit, containing clusters of assertion-checkers, for the AXI bus protocol. The maximum obtainable design coverage of the SDRAM controller by means of the debug unit armed with the SDRAM controller's assertion-checkers is plotted in Fig. 19.

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016

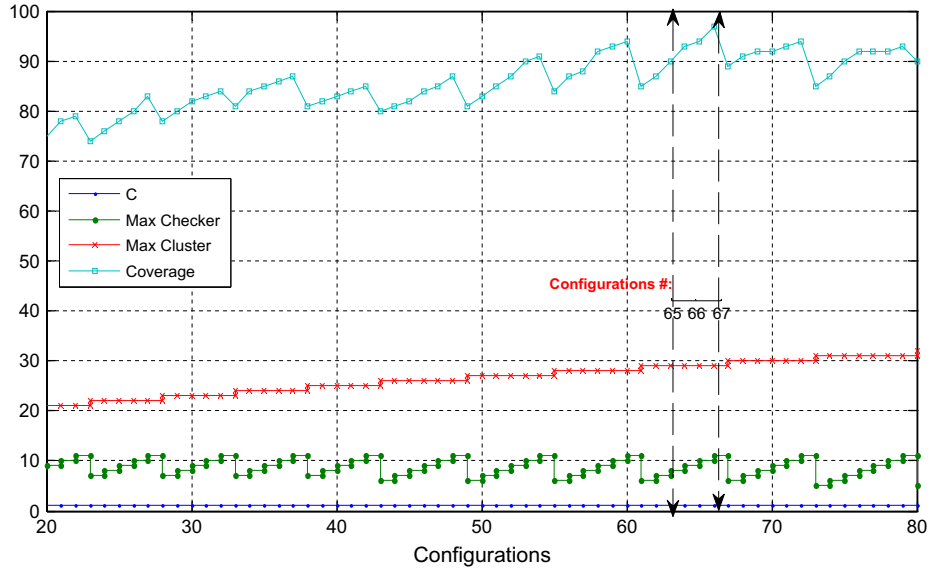

Fig. 18. Maximum design coverage of a device complaint with AXI bus protocol checkers in different configurations.

The analysis of these plots shows that by increasing the number of clusters the design coverage increases. Another important fact is that by increasing the limit on the maximum number of assertion-checkers placed inside a cluster, the design coverage grows too. Another important observation that can be extracted from these plots is that after reaching the certain cluster counts, the design coverage no longer increases. For example, for AXI protocol checker the maximum design coverage is achievable by means of this configuration (29,11,1); such a configuration for the PCI protocol checker and SDRAM controller is (14,12,1) and (8,11,1), respectively.

The important consideration here is that assuming the limited number of monitoring ports the clustering in general leads to a significant increase in the design coverage with respect to the non-clustering mechanism. Although the design coverage using non-clustering method is not reported in [3], assuming the limited number of monitoring points and using our mechanism, presented in Section 5, we computed its design coverage. It turned out that when the number of required monitoring ports is far more than that of available, the design coverage using the non-clustering approach is significantly low. For instance, assuming that the available monitoring ports is 32, the maximum design coverage for the AXI bus protocol checker achievable by the non-clustering scheme is 45%, which is less than that of clustering approach. On average the clustering scheme of placing assertion-checkers inside a debug circuitry results in 38% improvements in the design coverage of AXI protocol checkers. However, such improvements for the PCI bus protocol and SDRAM controller are 15% and 6%. Therefore, if a debug circuitry consisting of assertion-checkers is connected through a large set of wires (monitoring ports) to a design under debug, it is highly beneficial to resort to the clustering mechanism as a means to place such assertion-checkers into the debug module.

Fig. 20a plots the energy consumption of the debug module containing the assertion-checkers associated with the AXI bus protocol. The energy consumptions of the debug module containing SDRAM controller assertion-checkers with respect to different configurations is shown in Fig. 20b. As seen in these figures, the increases in the number of clusters result in higher energy consumptions. The important consideration here is that the clustering scheme in general leads to a drop in energy consumption in comparison to a non-clustering approach. One can simply associate such a decrease in the energy consumptions to the reduction in the required number of request lines in data selector module shown in Fig. 14b.

Fig. 21a and b represent hardware overhead of the debug module containing the assertion-checkers associated with the AXI bus protocol and SDRAM controller assertion-checkers, respectively. Plus, the area usage is also reported in terms of Gate Equivalents (GEs), which is the number of 2-input NAND gates.

As it can be seen, the increases in the number of clusters result to increases in the area overhead. Such an increase in the area overhead results from the increases in the required number of request lines in data selector module shown in Fig. 14b. However, the area overhead for the configurations that provide better design coverage are less than that of non-clustering method. Fig. 22a and b shows energy consumption and hardware overhead of the debug module associated with the PCI bus protocol, respectively. Fig. 22a represents a drop in energy consumption in comparison to a non-clustering approach.

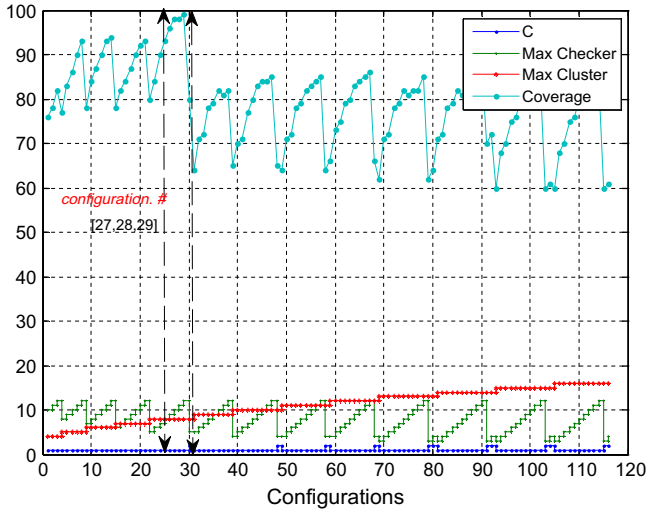

Fig. 19. Maximum design coverage of a SDRAM controller in different configurations.

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016

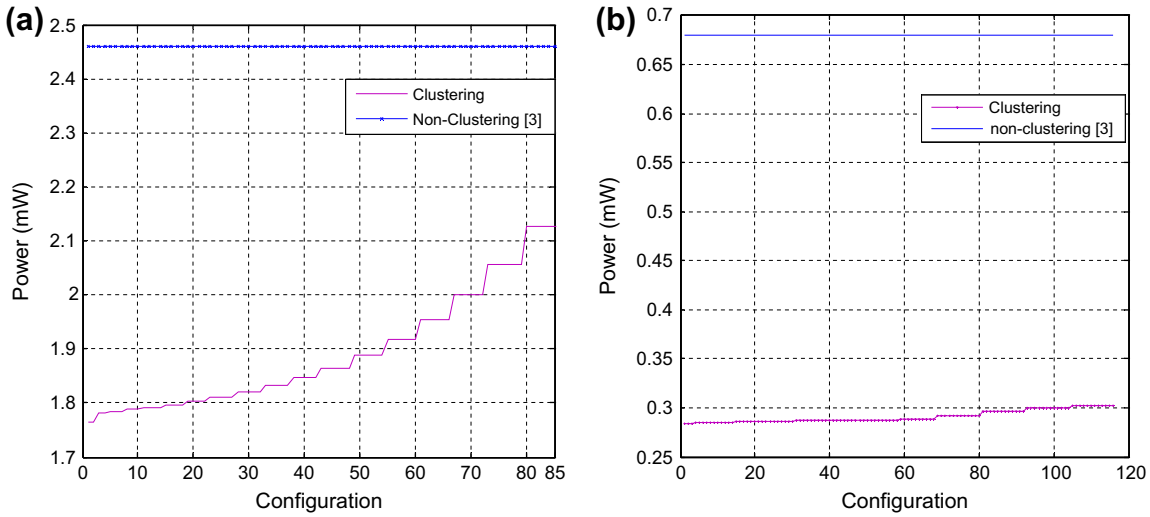

Fig. 20. Energy consumption of clustering scheme versus non-clustering mechanism (a) AXI bus protocol checkers, (b) SDRAM controller.

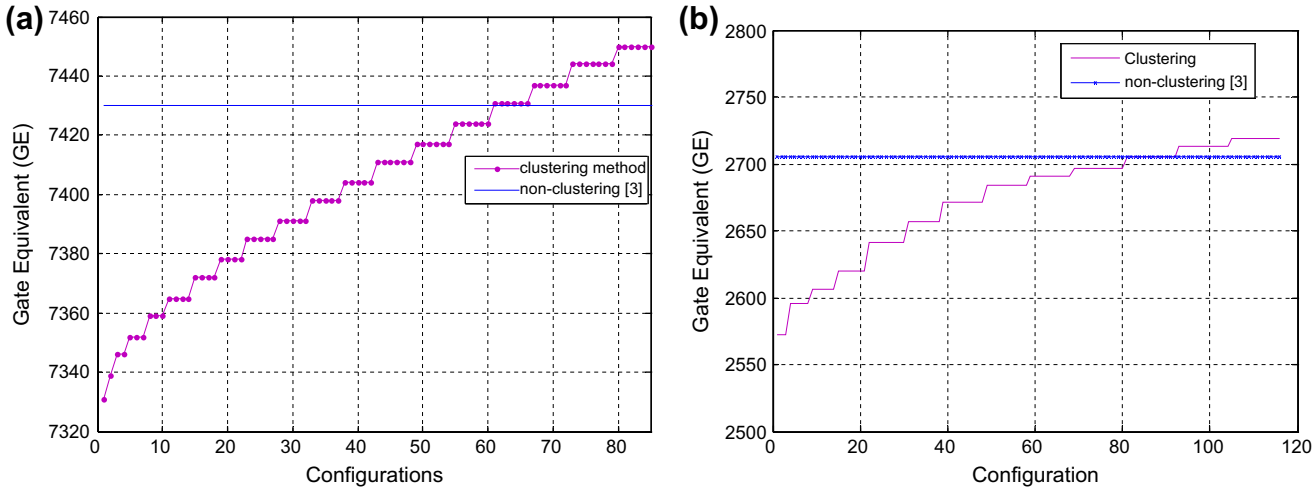

Fig. 21. Area overhead of clustering scheme versus non-clustering mechanism (a) AXI bus protocol checkers, (b) SDRAM controller.

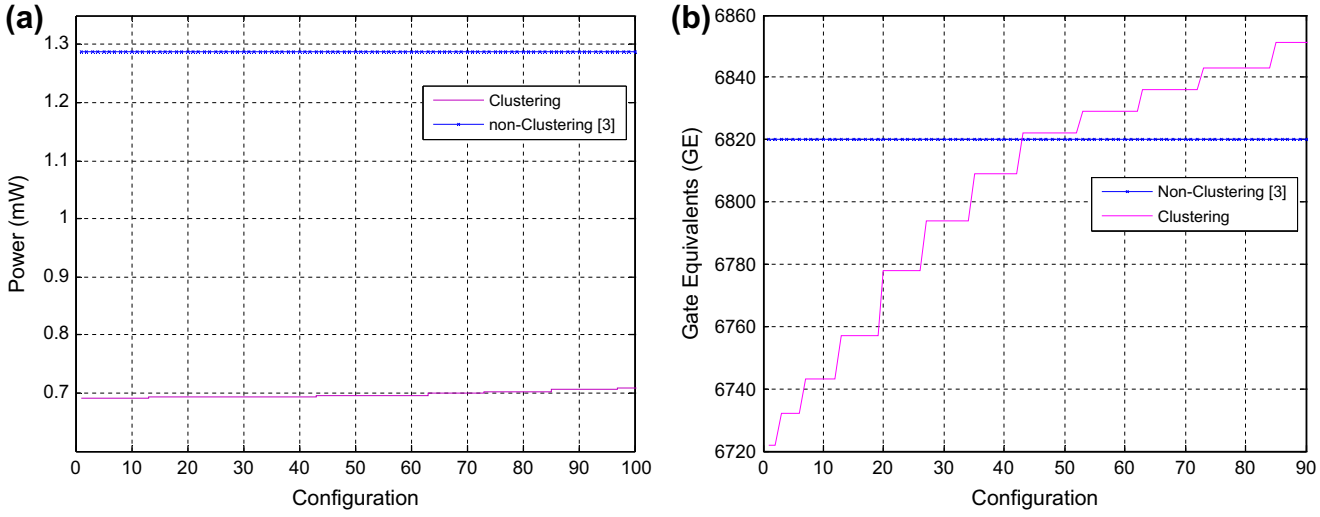

Fig. 22. PCI bus protocol checkers: (a) Energy consumption of clustering scheme versus non-clustering mechanism, (b) area overhead of clustering scheme versus non-clustering mechanism.

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016

Table 3
Features provided by our proposed method versus two related work.

	Partitioning	Coverage	Power	Incorporation in debugging Infrastructures	Port Count
Our proposal in this paper	Graph partitioning based on assertion-checkers fan-in cone set	Formally defined	Decrease in power consumptions	Support for Scan-based run-stop and trace-based debugging Enable interfacing with both Internal and External Memory Offer reusability through SDU	Reported
The proposed method in [11]	Subset Sum algorithm constrained with the available silicon area	N/A	N/A	N/A	N/A
The proposed method in [3]	N/A	N/A	N/A	Scan-based run-stop and trace-based debugging enable interfacing with internal memory	N/A

6.7. Comparison of features provided by our method with the related work

A summary of the features provided by our proposed method against related work in [3,11] is listed in Table 3. These features in particular are related to the integration of assertion-checkers in debug infrastructures. As it is listed in this table, the proposed method makes use of a graph partitioning to select assertion-checkers and place them inside a cluster, while the study in [11] uses a customized "Subset-Sum" algorithm constrained by the available silicon area as a means to partition assertion-checkers. It is important to note that the "Cluster-Generator" algorithm shown in Fig. 8 can be easily parameterized to consider the area constraints. Although the proposed method in [3] advocates clustering, it does not provide any mechanism for partitioning assertion-checkers. Neither the study in [3] nor [11] considers coverage metrics, while we formally defined the coverage metrics for assertion-checkers and their clusters.

The incorporation of assertion-checkers inside a scan-based debug infrastructure and a trace-memory based debug infrastructure has been addressed in our proposed method. We also investigate the integration of a set of assertion-checkers inside a shared debug unit (SDU) that can be treated as an independent salve module in the bus based SoCs.

As it is shown in Table 3, the method proposed in [11] has not addressed incorporation of assertion-checkers inside any debug infrastructure. Although the authors in [3] support incorporation of assertion-checkers inside a trace-memory based debug infrastructure, their scheme is bias toward the use of internal trace memory and cannot be generalized to support external trace memory. Neither the study in [3] nor [11] considers the important issue related to power consumption, while assertion-checkers are active, and they monitor the properties of a system. We leverage the fact that if there is a bug in a particular part of a system, the assertion-checkers monitoring the properties of that part of the system are more likely subject to failures. Therefore, by means of incorporating the related assertion-checkers into a cluster, we increase the chance of the root-cause extraction of errors. Plus, when an external debug tool generates test cases with a primary focus to exercise a precise part of the system, clusters involved in the validation of that particular module can be enabled selectively.

7. Conclusions

In this paper, we proposed a new algorithm to cluster assertion-checkers. Moreover, a mechanism to find the coverage of each cluster is also introduced. We also presented several mechanisms to incorporate clusters of assertion-checkers into the DfD infrastructure. The efficiency of the proposed methods is investigated using AXI bus, PCI bus protocol checkers and SDRAM memory controller checkers. The clustering algorithm, along with the proposed infrastructure lead to better results in terms of the energy consumption and design coverage compared to placing assertion-checkers without clustering.

References

[1] Abramovici M, Bradley P, Dwarakanath K, Levin P, Memmi G, Miller D. A reconfigurable design-for-debug infrastructure for SoC. In: Proceedings of design automation conference (DAC); 2006. p. 7–12.
[2] Foster H, Krolnik A, Lacey D. Assertion-based design. 2nd ed. Kluwer Academic Publishers; 2004.
[3] Geuzebroek J, Vermeulen B. Integration of hardware assertions in systems-on-chip. In: Proceedings of IEEE international test conference (ITC); 2008. p. 1–10.
[4] Vermeulen B, Goel SK. Design for debug: catching design errors in digital chips. IEEE Des Test Comput 2002;19:35–43.
[5] Van Rootselaar GJ, Vermeulen B. Silicon debug: scan chains alone are not enough. In: Proceedings of IEEE international test conference (ITC); 1999. p. 892–902.
[6] Neishaburi MH, Zilic Z. Enabling efficient post-silicon debug by clustering of hardware-assertions. In: Proceedings of IEEE design, automation & test in Europe conference (DATE); 2010. p. 985–8.
[7] Neishaburi MH, Zilic Z. Hierarchical trigger generation for post-silicon debugging. In: Proceedings of IEEE VLSI design, automation and test (VLSI-DAT); 2011. p. 1–4.
[8] Boule M, Zilic Z. Generating hardware assertion checkers: for hardware verification, emulation, post-fabrication debugging and on-line monitoring. Springer; 2008.
[9] Morin-Allory K, Boulé M, Borrione D, Zilic Z. Validating assertion language rewrite rules and semantics with automated theorem provers. IEEE Trans CAD Integr Circ Syst Sep. 2010;29(9):1436–48.
[10] Boule M, Zilic Z. Incorporating efficient assertion checkers into hardware emulation. In: Proceedings of international conference on computer design (ICCD); October 2005. p. 221–8.
[11] Boule M, Chenard J-S, Zilic Z. Assertion checkers in verification, silicon debug and in-field diagnosis. In: Proceedings of international symposium on quality electronic design (ISQED); 2007. p. 613–29.
[12] Boule M, Zilic Z. Efficient automata-based assertion checker synthesis of psl properties. In: Proceedings of IEEE high-level design validation and test workshop (HLDVT); November 2006. p. 69–76.
[13] Kakoee MR, Neishaburi MH, Daneshtalab M, Safari S, Navabi Z. On-chip verification of NoCs using assertion processors. In: Proceedings of IEEE digital system design (DSD); 2007. p. 535–8.
[14] Sica FC, Coelho Junio CN, Nacif JAM, Foster H, Fernandes AO. Exception handling in microprocessors using assertion libraries. In: Proceedings of IEEE integrated circuits and systems design (SBCCI); 2004. p 55–9.
[15] Caty O, Dahlgren P, Bayraktaroglu I. Microprocessor silicon debug based on failure propagation tracing. In: Proceedings of IEEE international test conference (ITC); 2005. p. 755–63.
[16] Gao J, Han Y, Li X. A new post-silicon debug based on suspect window. In: Proceedings IEEE VLSI test, symposium; 2009. p. 85–90.
[17] Yang J-S, Touba NA. Enhancing silicon debug via periodic monitoring. In: Proceedings of IEEE symposium on defect and fault tolerance (DFT); 2008. p. 125–33.
[18] Ko HF, Kinsman AB, Nicolici N. Distributed embedded logic analysis for post-silicon validation of SoC, In: Proceedings of IEEE international test conference (ITC); 2008. p. 1–10.
[19] Vermeulen B, Waayers T, Goel SK. Core-based scan architecture for silicon debug. In: Proceedings of IEEE international test conference (ITC); 2002. p. 638–47.
[20] Gao M, Cheng KT. A case study of time-multiplexed assertion checking for post-silicon debugging. In: Proceedings of IEEE high-level design validation and test workshop (HLDVT); 2010. p. 90–6.
[21] Quinton BR, Wilton SJE. Concentrator access networks for programmable logic cores on SoCs. In: Proceedings of IEEE symposium on circuits and systems (ISCAS); 2005. p. 45–8.
[22] Accellera, PSL Language Reference Manual, version 1.1. <http://www.eda.org/vfv/docs/PSL-v1.1.pdf>.
[23] SystemVerilog Assertions. <http://www.synopsys.com/products/simulation/assert_sverilog_wp.pdf>.

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016

[24] ARM Limited. Coresight on-chip debug and trace technology. <http://www.arm.com/products/solutiona/coresight.html>.
[25] ARM AMBA 3 specification and assertions. <http://www.arm.com/products/solutions/axi_spec.html>.
[26] Vijayaraghavan S, Ramanathan M. A practical guide for systemverilog assertions. Springer; 2005.
[27] http://www.webopedia.com/TERM/P/PCI.html.
[28] Gaisler IP Cores. <http://www.gaisler.com/products/grlib/> 2009.
[29] Neishaburi MH, Zilic Z. On failure rate assessment using an executable model of the system. In: Proceedings of digital system design (DSD); 2011. p. 29-36.
[30] Neishaburi MH, Zilic Z. Hierarchical embedded logic analyzer for accurate root-cause analysis. In: Proceedings of international symposium on defect and fault tolerance in VLSI and Nanotechnology Systems (DFT); 2011. p. 120-8.
[31] Neishaburi MH, Kakoee MR, Daneshtalab M, Safari S. HW/SW architecture for soft-error cancellation in real-time operating system. IEICE Electron Express 2007;4(23):755-61.
[32] Neishaburi MH, Zilic Z. Reliability aware NoC router architecture using input channel buffer sharing. In: Proceedings of great lake symposium on VLSI (GLSVLSI); 2009. p. 511-6.
[33] Neishaburi MH, Zilic Z. ERAVC: Enhanced reliability aware NoC router. In: Proceedings of international symposium on quality electronic design (ISQED); 2011. p. 591-6.
[34] Neishaburi MH, Zilic Z., A fault tolerant hierarchical network on chip router architecture. In: Proceedings of international symposium on defect and fault tolerance in VLSI and nanotechnology systems (DFT); 2011. p. 445-53.

Please cite this article in press as: Neishaburi MH, Zilic Z. An infrastructure for debug using clusters of assertion-checkers. Microelectron Reliab (2012), http://dx.doi.org/10.1016/j.microrel.2012.04.016