Exhaustive Equivalence Checking on AMD's Next-generation Microprocessor Core

Baosheng Wang, Brian McMinn, Borhan Roohipour, Ashok Venkatachar, Arun Chandra, Richard Bartolotti, and Lerzan Celikkanat

Advanced Micro Devices, Inc., 1 AMD Place, Sunnyvale CA 94085, USA

{FirstName.LastName}@amd.com

Abstract

There is an ever-increasing demand for higher performance microprocessors within a given power budget. Such a demand forces design choices – that were once seen only in high-speed custom blocks – to spread throughout the microprocessor core. Given these unique custom structures, traditional equivalence checking methods can no longer meet the requirements of achieving the optimal verification coverage and runtime tradeoff. More advanced equivalence checking methodologies are needed to provide 100% verification coverage while maintain a reasonable verification runtime.

In this paper, we present a framework of exhaustive equivalence checking strategies used at AMD on it's next generation high-performance microprocessor core. This framework consists of one mainline tape-out sign-off flow, two assisting flows and three supporting flows. This framework utilizes multiple state of the art equivalence checking techniques, such as logic equivalence checking, symbolic simulation, binary simulation, and model checking. To avoid unnecessarily coverage overlap among the three major equivalence checking flows, thorough coverage analysis is performed. By removing the unnecessary coverage overlap among these validation flows and coupling their applications at different design hierarchies, this framework is achieving efficient runtime without compromising the verification quality.

In addition, automation has been put in place to prevent pilot errors as much as possible.

***Keywords**: Equivalence Checking, Exhaustive, Symbolic Verification, Model Checking, Simulation, Automation.*

1. Introduction

AMD's next-generation "Bulldozer" microprocessor core [1] is a high-performance core that is significantly different than previous AMD cores [2-3]. It is a power-efficient, cluster-based and multi-threaded execution engine. It is designed to enable high-frequency operation while delivering superior throughput at a given power budget. The monolithic "core pair" is capable of executing two threads via a combination of shared and dedicated hardware resources. The core includes a 16 KB L1 cache, 2 MB L2 cache and an enhanced 128-bit floating point unit. The core design has more than 200 million transistors and it will be used in microprocessors produced at 32nm or smaller technology nodes.

Due to ever-increasing performance requirements, a significant portion of the core design includes high-speed dynamic logic and custom memory structures. Traditionally, thanks to its high efficiency, good scalability and proven records on static designs (i.e., conventional CMOS standard cell designs), Combinational Equivalence Checking (CEC) is widely used to ensure the design implementations do not alter the functional behaviors of the RTL [4-6]. While dynamic logic and custom blocks spread throughout the design, CEC can no longer be able to achieve exhaustive equivalency checking due to the following reasons: 1) standard cells are never the only building blocks of the microprocessors and custom cell verification is demanded to solidify the foundation of CEC; 2) gate-level logic optimization requires certain constraints to help CEC sign off. Without exhaustively verifying those CEC constraints, the CEC quality is in question; 3) because of the limited EDA technology, certain circuit topologies, e. g., dynamic logic, have low CEC coverage and hence spice-level verification on those topologies is a must; 4) the microprocessor is a serialized engine and there might be sequential-event-sensitive structures which are never the CEC targets.

Some could argue that Sequential Equivalency Checking (SEC) might alleviate pains of CEC, however, the size of microprocessor designs makes SEC an impractical option in terms of verification runtime.

In the rest of this paper, Section 2 explains the "Bulldozer" microprocessor building and analysis blocks. The proposed framework, including the signoff, assisting and supporting flows, are presented in detail in Section 3. Section 4 utilizes an execution logistical flowchart to illustrate the framework automation and demonstrate how the exhaustive equivalence checking is performed on AMD's next-generation microprocessors core, and Section 5 concludes the paper.

2. Building and analysis blocks of the Core

AMD's "Bulldozer" microprocessor core is a hierarchical design. It consists of three hierarchies (as shown in Figure 1): component, block-level-module (BLM) and macro.

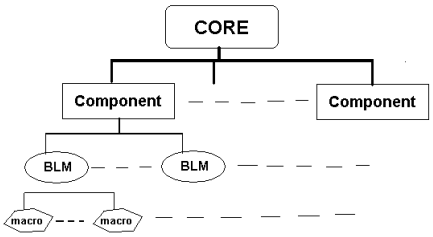

Figure 1. The hierarchies of "Bulldozer" core

Macro-level blocks are the most complex designs in the microprocessors. Numerous transistor-level design styles, such as dynamic logic, timing-adjusting circuits, are invented to meet timing, area and power requirements. No doubt, such blocks require the most extensive equivalency checking practices.

BLMs are usually designed to provide glue logic for custom macros and therefore most of them can be RTL-based. However, this does not mean their equivalency verification is well fitting with typical CEC flow. Certain structures, e.g., Build-in Constrain-Resolution (BICR) and one-hot structures [7], requires extra CEC enhancements to achieve exhaustive equivalency coverage.

Components consist of multiple BLMs, where unfortunately interconnects between BLMs are not always static. Those non-static interconnects actually introduce multiple equivalency checking challenges (see details in the following sections). Finally, several components are constructed together becoming the monolithic core.

AMD's Bulldozer microprocessor core utilizes templates as the fundamental blocks for electrical analysis purposes, such as analysis on noise, IR, electrical migration, and so on [8]. A template is a collection of a specific circuit topology originated from a standard cell, a custom cell or both. Besides, it carries plenty of circuit topology information, e. g., electrical properties, transistor-level connectivity graph, gate-level models, etc. Once all known templates are built, all macros, BLMs and components can be interpreted with multiple templates connected in certain ways. Such interpretation mechanics is the in-house tool "classification". As a result, the template library delegates the full circuit topology sets of the "Bulldozer" microprocessor core. In other words, the exhaustive template equivalence verification is fundamental for all core analysis flows.

3. The Proposed Framework

The proposed framework consists of multiple modern equivalency checking methods (as shown in Figure 2): the mainline tape-out signoff equivalence checking flow, i.e., CEC, the two assisting equivalence checking flows – symbolic equivalency checking and functional gate-level simulation-based equivalency checking, three supporting equivalency checking flows – template-level equivalency checking, CEC constraint harvesting and gate-level constraint proof.

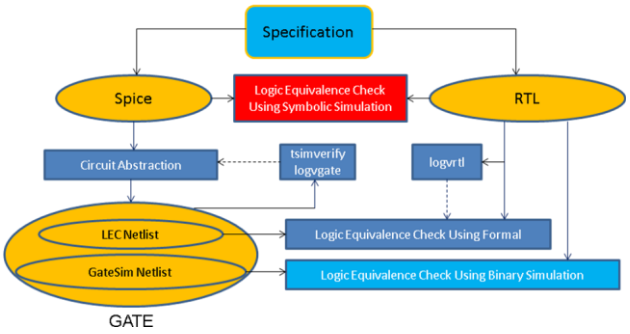

Figure 2. The proposed framework

From Figure 2, once the microprocessor design specification is created, its RTL-level coding and spice-level implementation can sometimes start in parallel. Whenever a CEC run is requested during the product development stages, the spice-level circuits have to be abstracted into gate-level formats in terms of basic building blocks, i.e., the templates for "Bulldozer" microprocessor core. Without question, template verification flow (i.e., tsimverify) has to be launched to ensure the template verilog models exactly matching its spice-level representation under certain conditions, i.e., input pin constraints. The supporting flow logvgate is designed to formally prove those template-level input pin constraints at various design hierarchies.

Even with both highly qualified RTL and templates, CEC still needs some helps from a bunch of RTL-level constraints to claim the RTL and the implementation are really logically equivalent. The reasons contributing to such a need on RTL-level constraints can be referred in the following sub-sections. Fortunately, we design a dedicate flow named as "logvrtl" to formally harvest those RTL-level constraint.

Lastly, to address certain possible weaknesses of the circuit abstraction/modeling process and CEC weakness on dynamic and sequential-even-sensitive circuits, we introduce 2 extra assisting flows, i.e., macro-level symbolic equivalence checking and functional simulation-based equivalence checking.

With careful arrangements and optimal coverage-productivity tradeoffs with those above flows, we can achieve exhaustive equivalency checking coverage on AMD's "Bulldozer" microprocessor core.

3.1 Tsimverify and Logvgate Flows

Once a template is formed through the "classification" process on the design implementations, the first step is to create its abstracted gate-level models. We can skip the template model abstraction topic since it is out of the focus of this paper. Once a template model is developed, we apply the "tsimverify" flow to compare the template

gate-level model and its spice representation making sure they are equivalent by feeding them with exhaustive binary vectors. The block diagram of the "tsimverify" flow is shown in Figure 3:

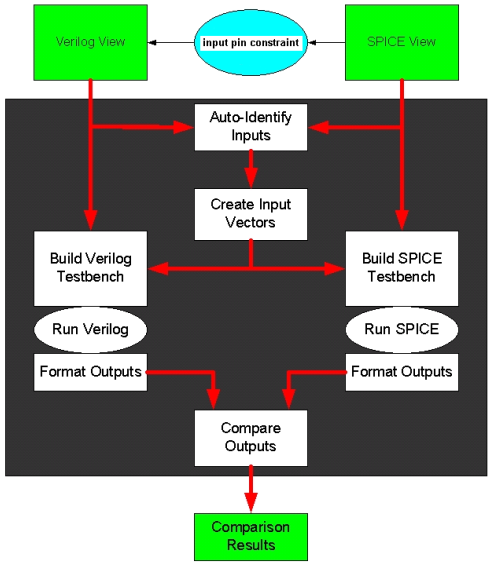

Figure 3. Block Diagram of "tsimverify" flow

As shown in Figure 3, the exhaustive binary vectors w/o violating the template input pin constraints are simulated on both types of template representations. Their output responses are compared. Once there is a mismatch, either the template gate model requires a fix or certain template input constraints are missing.

To avoid potential vector-order weaknesses of those binary vectors, symbolic vectors are selected on verifying templates which are sequential circuits, e.g., flops, latches, etc.

To ensure a template is used correctly, that is, its input pin constraints used at template-level verification are never violated when such a template is instantiated in the design, we develop a flow so called "logvgate" to formally prove those template input pin constraints. A block diagram of the logvgate flow is illustrated in Figure 4 (the flow details can be referred to [8]).

Once a flatten spice-level netlist is provided, the "classification" engine tries to identify every possible pre-defined templates used in the design and creates a flatten gate-level netlist in terms of templates. Meanwhile, the template-level input pin constraints are converted into OVL assertions, where each element in the equations is the driver net to those template input pins. With the helps of user-defined control files, the assertions plus the gate-level netlist are feeding into a model checking tool for constraint proof. If such assertions are all proven, it can be claimed that the input pin constraints used for template verification is true. In other words, all templates are used in the expected way.

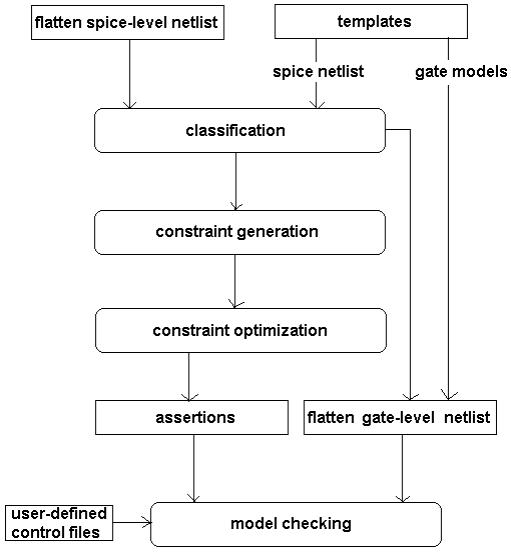

Figure 4. Block Diagram of "logvgate" flow

Two extra efforts are usually required during assertion failure debug:

1) Create helper assertions: due to gate optimization, sometimes there is a need to create help assertions as assumptions in order to prove certain template constraints. Usually, those assumptions can be found in the RTL and then this effort is to translate those RTL-level assertions into gate-level ones with the helps of CEC mapping files. If those assumptions do not exist in the RTL, negotiation with RTL designers is needed to create such valid helper assertions. One important note is that all elements of those RTL-level helper assertions have to be directly from outputs of state elements since CEC only provides state mappings.

2) Create initial vectors: our logvgate flow starts with random state values. However, sometimes, in order to constrain the model checking within a certain operation mode, e.g., mission mode, non-related states have to be initialized. For example, when running logvgate at mission mode, all scan control related fops are supposed to be in reset states.

Another engineering problem we are usually facing is the logvgate flow is running too long due to a large number of assertions. Our solution is to run logvgate hierarchically:

a) Run logvgate at macro-level w/o any helper assertions; report both passed and fired assertions;

b) Run constraint generation at BLM-level and report all generated assertions;

c) Prune the macro-level passed assertions from the BLM-level assertions based on test expression matching;

d) Run model checking on the pruned assertions at BLM-level.

This hierarchical logvgate flow is proven to be very efficient in runtime, especially when the same macros are instantiated in the BLMs multiple times.

3.2 CEC and Logvrtl Flows

The CEC flow used for AMD's "Bulldozer" microprocessor core is not much different from the one used for full-static designs. For better coverage-productivity tradeoffs, CEC is running at both functional and scan modes on macros while BLM-level CEC is running at functional mode only.

As mentioned in Section 2, there are quite a few BLMs which utilize BICR technology to prevent contentions on one-hot structures from random scan vectors [9] (see the diagram in Figure 5).

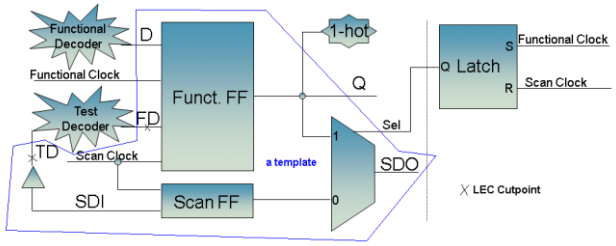

Figure 5. Enhance CEC for BICR Test Decoder Coverage

Except the functional decoder, the BICR equips test decoder logic to ensure any random scan input vectors can be decoded as one-hot values before feeding into the functional flops driving the 1-hot structures. The RS latch in the Figure 5 is used as a selection mechanism for observing scan or functional data paths on the scan chains.

Unfortunately, those BLMs are not the ones which can be created by synthesis due to performance requirements. As a result, the test decoder logic is also created manually and the creation of which is therefore error-prone. To overcome such a CEC coverage loss at scan mode, we enhance the typical CEC as follows:

1) Identify the inputs and outputs of test decoder and define LEC cut points on those nodes. Luckily, the whole BICR flop can be formed as a template and the LEC cutpoint identification becomes much easier since they can simply be the inputs and outs of BICR flops;

2). Based on BICR flop template naming conventions, all BICR flop instances can be found and those test decoder logic are fully validated using typical CEC.

Another enhancement to ensure CEC robustness is the utilization of logvrtl flow [10], which is formally proving the CEC constraints at RTL level. The details of this flow principle can be referred in [10]. Currently, we are running logvrtl at BLM and component levels only for two reasons: a) since quite significant glue logic exists outside of macros, running logvrtl at macro-level will have lots of false negatives; b) Tool capacity issues prevent us running logvrtl at the full core level.

For better coverage-productivity tradeoff, we further enhance this flow in three ways:

1) Initialize all non-array states. It is very important to start the logvrtl process from a known operation mode, e.g., mission mode, to avoid unnecessary false negatives. We implement this enhancement by launching the reset vectors dumped from functional verification.

2) Create helper assertions as many as possible at BLM-level for either clearing the fires or improving logvrtl runtime. As stated in Section 2, lots of interconnects between BLMs are not always static. As a result, it is expected that there would be BLM-level fired assertions which can be fixed using helper assertions. Due to intended design loops, logvrtl sometimes is running very slow and appropriate helper assertions can greatly shorten the tool analysis time on those loops. Moreover, creating helper assertions can also help us to understand the root-cause of fires and therefore increase the robustness of the lower-level design at an early stage. Some may be concerned with the correctness of those helper assertions. Luckily, those helper assertions are blessed by massive functional verification vectors before becoming inline assertions and there is little chance when incorrect helper assertions are checked in.

3) Prove all CEC constraints plus helper assertions at component level. With all correlated logic in place for all BLMs, it is very unusual that helper assertions are needed, except the architectural-level assumptions and the ones that are not guaranteed by hardware but through firmware/software.

3.3 Symbolic-based Equivalence Checking

Typical CEC flow compares logic cones from one state point to another. The state point clocking mechanism must be activated for the cone logic in order to make it reachable. Unfortunately, this is not always true for dynamic logic and glitch latches [11] because their data paths also involve clocks. As a result, CEC coverage on dynamic logic and glitch latches are not exhaustive.

Another CEC flow drawback is that the keeper functionality can not be modeled accordingly. Basically, a keeper keeps the previous value of the circuit. Keepers are also called rail-pullers. Keepers are like half latches and have loops and introduce sequentiality. In CEC, the keeper circuitry is simply modeled using a weak resistive pull-up. If such a keeper transistor is not actually resistive, an error can be masked [12].

The last concern with gate-level equivalence checking is the dependency on the circuit abstraction accuracy. Although the templates are exhaustively verified, there might be some user-preferred modeling tricks but they are actually biased from the real circuit behaviors.

To resolve the three coverage/robustness concerns above, we introduce the symbolic-simulation-based equivalency checking flow for our "Bulldozer" microprocessor core [13]. This flow simply compares CEC-clean spice-level netlist with the corresponding RTL and hence no gate-level modeling is required, as shown in Figure 2. Since most of the highly-customized circuits reside in macros, this flow is only run at macro level.

The most annoying problem we encountered during symbolic equivalence checking is the symbolic space explosion. We resolve this issue in the following three ways:

1) Constrain the symbolic equivalence checking at function mode only. The design usually becomes simple enough when it operates at scan mode and typical CEC can cover them very well.

2) Do divide & conquer on complex macros. The common practice is to do either protocol-based or data-integrity-based divide & conquer. Sometimes, sub data-integrity-based divide & conquer is needed because of wide address or data buses.

3) Do coverage-driven divide & conquer on super complex macros. If the regular divide & conquer mentioned in 2) is not effective in terms of runtime, we do coverage-driven divide & conquer by further reducing overlapped coverage from CEC and only targeting on the necessary coverage area, e.g., the dynamic logic, glitch latches, the keepers, etc. By comparing the reports from template-based topology identifications and the ones from symbolic equivalence checking coverage, all CEC coverage loss can be compensated.

3.4 Equivalence Checking with Functional Vectors

To fully validate the Bulldozer's microprocessor core, the last equivalence checking flow we developed is to do equivalence checking with functional vectors so that sequential-event-sensitive structures can be covered. It is intended to prove the equivalency of the RTL-level design and its gate-level implementation through clean functional vectors. As an assisting signoff flow of CEC, its cleanness is very important. It paves the foundation for other related flows, such as ATPG (Automatic Test Pattern Generation) based verification, power analysis, and so on. Also, it reduces debug/run efforts on SoC-level validations. Gate-level simulation is a very complex and time-consuming practice since it processes the target RTL, the corresponding CEC-clean gate-level netlist, the clean template gate-level models, CEC mapping files and the passed functional vectors.

To achieve the optimal throughput without losing too much coverage, we apply the following four approaches in order to make it cost-effective:

1) Apply it on BLMs only at an early design stage. For more mature designs, apply it on components and cores only since there are only a few components and cores.

2) Limit the stimulation/observation points at primary inputs/outputs only. Only when there is a need for debug, state points can be the stimulation/observation nodes.

3) The number of functional vectors is limited. Those selected vectors are usually targeting the critical functions in functional/DFX/power-saving areas.

4) Provide a mechanism for substituting the gate-level netlist with its RTL/behavior counterpart. This not only can speed up flow runtime for fast turnaround but also enable an early equivalency checking on entire designs when a small portion of gate-level designs is unavailable.

4. Automation

Without automation, it is impossible to execute such an exhaustive equivalence checking framework in a timely manner. Also, with extensive automation, potential pilot errors can be avoided at the entries/exists of each equivalence check flow. The third benefit of automating this framework is that it can be reusable for other projects.

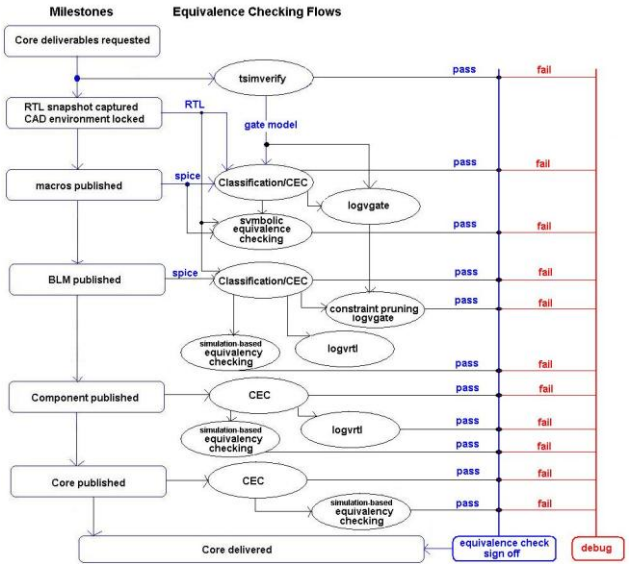

Figure 6. The Automation of the Proposed Framework

In Section 3, we have explained the flowchart of individual equivalence checking flow. This section mainly addresses the overall automation flowchart of the proposed framework (see the block diagram in Figure 6).

Since the template is the fundamental element for both microprocessor core analysis and equivalence checking flows, this library must be clean before starting corresponding downstream flows. As a result, the template information, e.g., the tsimverify status, the template version, is part of CAD environment for core deliverables.

To reduce the human communication efforts, we embed the CEC at all levels and other equivalence checking flows under component level into design publish processes. In other words, whenever a designer decides to publish his/her macros/BLMs, all necessary equivalence checking flows have to be run. Depending on different program milestones, some equivalence checking flows require to be passed for those macros/BLMs.

For example, as shown in Figure 6, the CEC flow is required to be run and passed at all design levels whenever a macro/BLM is published, which not only builds the foundation of the design quality but also reduces the engineering debug efforts on other equivalence checking flows. The macro-level symbolic equivalence checking and logvgate, BLM-level logvrtl, logvgate and simulation based equivalence checking flows are also parts of design publish processes to further improve the design quality of equivalence checking.

It is noted that macro-level logvgate and BLM-level logvrtl flows are run for improving productivity. In other words, no debug efforts are required when there are failures coming out from both flows. After all failures are fully debugged and fixed/waived, we officially sign off on equivalence checking to core deliver team.

5. Conclusion

AMD's next-generation "Bulldozer" microprocessor core is a high-performance, a power-efficient and multi-threaded execution engine. It is designed to deliver superior throughput at a given power budget with the helps of highly-customized designs. Such design choices create lots of equivalency checking challenges when we are trying to ensure those customized circuits work as expected, i.e., not altering the functionalities defined at RTL level. The proposed exhaustive equivalence checking framework greatly resolves the concerns of silicon bugs due to circuit designer errors without delaying the products. The automated execution logistical flow chart demonstrates such a framework is very cost-effective vehicle on ensuring the transistors perform what we describe in RTL.

6. Acknowledgements

We thank anonymous reviewers at Advanced Micro Devices, Inc. for their insightful comments and suggestions on this work.

7. References

[1] M. Butler, "Bulldozer – a new approach to multi-thread compute performance", *the IEEE 22nd HotChips conference – a symposium on high performance chips*, Session 7.2, August 22 – 24, 2010

[2] T. Wood, "Test and debug features of the AMD-K7™ microprocessor", in *Proceedings of IEEE International Test Conference (ITC)*, 1999, pp. 130 -136

[3] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, R. Kumar, "An Integrated Quad-Core Opteron Processor", the *IEEE International Solid-State Circuits Conference (ISSCC)*, 2007, pp. 102-103

[4] X. Feng, "Formal equivalence checking of software specifications vs. hardware implementations", PhD thesis, University of British Columbia, January 2007

[5] A. Kuehlmann and Cornelis A. J. van Eijk, "Combinational and sequential equivalence checking", in Tsutomu Saso, Soha Hassoun, editor, Logic Synthesis and Verification, pp. 343–372. Kluwer Academic Publishers, 2002. ISBN: 0-7923-7606-4

[6] F. Somenzi and A. Kuehlmann, "Equivalence checking", in Louis Scheffer, Luciano Lavagno, and Grant Martin, editor, Electronic Design Automation For Integrated Circuits Handbook. CRC Press, 2006. ISBN: 0-8493-3096-3

[7] G. Giles, J. Irby, D. Toneva, and K.-H. Tasi, "Built-in constraint resolution", in *Proceedings of IEEE International Test Conference (ITC)*, 2005, pp. 696-706

[8] R. Bartolotti, T. Burd, B. McMinn and A. Chandra, "Constraint management and checking in template-based circuit designs", *the 10th IEEE International Workshop on Microprocessor Test and Verification (MTV)*, 2009, pp. 107-113

[9] M. Yilmaz, B. Wang, J. Rajarman, T. Olsen, K. Sobti, D. Elvey, J. Fitzgerald, G. Giles and W-Y Chen, "The scan-DFT features of AMD's next-generation microprocessor core", in *Proceedings of IEEE International Test Conference (ITC)*, 2010, pp. 2.1.1-2.1.10

[10] X. Feng, J. Gutierrez, M. Pratt, M. Eslinger and N. Farkash, "Using model checking to prove constrains of combinational equivalence checking", in *the Design & Verification Conference & Exhibition (DVCON)*, 2010, pp. 8.2.1-8.2.7

[11] A. Chandrakasan, W. Bowhill, and F. Fox, Design of high-performance microprocessor circuit, New York, NY: Wiley-IEEE Press, 2001

[12] A. Chandra, L. C. Wang, and M. Abadir, "Practical considerations in formal equivalence checking of PowerPC microprocessors", in *Proceedings of the 8th IEEE Great Lakes Symposium on VLSI*, 1998, pp. 362-367

[13] Synopsys, ESP-CV User Guide, Production Version 2010.06, June 2010