Towards an IEEE P1500 Verification Infrastructure
A Comprehensive Approach

Iraklis Diamantidis
iraklis@globetechsolutions.com

Thanasis Oikonomou
poisson@globetechsolutions.com

Stylianos Diamantidis
stelix@globetechsolutions.com

Globetech Solutions, Thessaloniki, Greece

Keywords: Embedded Cores, IEEE P1500, Verification, Infrastructure IP, Core Test

Abstract

Core-based design has quickly become today's de-facto approach to building increasingly complex Systems-on-Chip (SoC). With the IEEE 1500 Proposal for a Standard for Embedded Core Test effectively addressing the important issues of reuse and interoperability with respect to testing core-based SoCs, as well as providing the infrastructure for building and operating testability features within cores from different suppliers, it has also become imperative to thoroughly verify the functionality of the complete test infrastructure within a certain SoC. In this paper we take a comprehensive approach to designing such a verification infrastructure based on a dynamic, constrained-random, coverage-driven verification methodology, which can be part of the overall chip-level validation strategy. We also present a powerful implementation of such an environment using a contemporary Hardware Verification Language, along with experiences from application on a typical verification scenario.

1 Introduction

Reuse is a key element to designing high-complexity Systems-On-Chip (SoC) within the time and performance limitations imposed by demanding market conditions. In this content reuse refers to the ability of introducing complete functions into an existing design by means of integrating pre-designed, pre-verified, blocks of logic, most often termed to as *embedded cores*. According to Gartner Dataquest, an independent research firm, increasingly more and more designs will consume more and more embedded cores so that by 2007 some SoCs will have surpassed 40 integrated cores and the *Intellectual Property (IP) Core* market will have doubled.

Conversely, testing such large core-based designs has quickly become a major consideration in any chip-based product ecosystem. In a typical SoC environment, embedded cores are supplied by multiple, external, IP vendors, each applying different development and quality standards. Blocks procured from such suppliers could deploy anywhere from none to a wide range of testability features, summing up to a completely heterogeneous and, in some cases, unusable test infrastructure at the system level. The need for a standard test infrastructure has led to the development of a variety of efforts by both industrial and professional organizations. The IEEE 1500 Standard Embedded Core Test proposal, currently in the final stages of the IEEE standardization processes, comprises a comprehensive set of guidelines for building such an infrastructure, including the hardware architecture, information model (implemented in the IEEE P1450.6 *Core Test Language* proposal) and definitions of levels of compliance.

In related previous work, a variety of publications [1] [2] have been presented on IEEE P1500 architecture and applications. The authors of [3] describe an approach of verifying 1149.1 (JTAG) logic using a combination of simulation of black-box checks and tracing. In addition, several others [4] [5] describe the work done on SoCs built with IEEE P1500 testability features.

In this paper we proceed to first understand in more detail the underlying motivations for our work, claiming that thorough functional verification of IEEE P1500 wrappers and wrapper cores in an SoC environment is absolutely necessary. We then set forth our aims for what an environment used to verify this standard should support.

2 Motivation

In our work we represent that the growing demand for test infrastructure in cores and SoCs is quickly becoming subject to challenges faced elsewhere in the electronic design industry. One of the most significant challenges, the *verification bottleneck* arises from the combination of a variety of emerging conditions; design heterogeneity, large chip area, increasing complexity and poor interoperability are only a few.

A complete set of challenges arise when considering the heterogeneous nature of today's IP market. For the *core developer*, it is important to provide IEEE P1500- compliant wrapped or unwrapped (i.e. CTL-only) designs to facilitate customer integration into system-level test infrastructure. C*ore integrators* on the other hand, need to ensure that IEEE P1500-ready IP properly complies with the required functionality both at the standalone and system levels. Obviously there are several points in this process where design bugs can be introduced,

ranging from insufficient *functional* exercise of a wrapper by the IP vendor to cell library issues in the gate level model.

Another set of challenges arise from the nature of the IEEE P1500 standard itself. The need to support as wide a range of embedded core test applications as possible has led to a very flexible and applicable solution. As described in [2], although a mandatory minimal set of hardware support is defined, a designer can extend the test infrastructure by creating virtually unlimited sets of register and instruction extensions [6]; such extensions can include *Core Defined Registers (CDRs)*, i.e. chains that are embedded in the wrapped core and can be invisible to the integrator or *Wrapper Defined Registers (WDRs)*, i.e. chains that are part of the Wrapper itself. Functionality provided by IP vendors in the form of CDRs and WDRs should be described in a CTL information model. However it is also subject to logic bugs due to improper and/or insufficient verification on the side of the provider.

Finally, one can also claim that DFT generation tools do leave room for error at both the structural and functional ends of the design space spectra. Implementations of SoCs containing P1500 are bound to suffer from potential protocol deviations, most likely due to human error. Such deviations could render an embedded core or even, in the case of a chain of wrapped cores, a complete set of cores untestable.

It is hence obvious that if an SoC includes IEEE P1500-compliant cores, whether they are created in-house or sourced externally, wrapped or unwrapped, manually designed or generated, including minimal or extended IEEE P1500 features, complete and methodical verification of the IEEE P1500 test logic is a necessity.

To better understand the methodology and environment deployed for creating such a verification tool, it is important to first understand what some of the core features that the environment needs to provide are, in order to properly embrace the IEEE P1500 standard:

- **Abstraction** - Specifying vector stimuli at different layers of abstraction
- **Coreless Operation** - The ability to verify a standalone wrapper
- **Layered Monitoring -** Observing behavior in environments ranging from white-box to black-box
- **Functional Coverage Assessment -** Measuring the extent of functional coverage that has been exercised in the system
- **Extensibility** - Providing as much support for CDR/WDR extensions as possible without user input
- **Configurability -** Ensuring that all configuration options within the standard can be satisfied
- **Reusability** - Being able to apply the environment across providers, projects and abstraction levels

3 Methodology and Environment

All challenges arising from the requirements set in the previous section can be addressed with *dynamic, constrained-random, coverage-driven,* functional verification methodology. By *dynamic functional* verification, input patterns are generated and applied over a number of clock cycles to the design and the corresponding result is collected and compared against a reference model for compliance with the specification. *Functional coverage* metrics quantify the functional space that has been covered by a test suite. *Random dynamic* simulation provides random stimulus to the design, maximizing the functional space that can be covered. Stimulus may also be constrained on demand to narrow the range of possible values making the tests more directed where needed (*constrained-random* stimulus).

3.1 Implementation technology

For implementing the dynamic, constrained-random, coverage-driven functional verification methodology we deployed Verisity's ***e*** Hardware Verification Language[1]. ***e*** has built-in constructs for easily implementing dynamic constrained-random stimulus generation, describing functional coverage goals and defining certain protocol and data checks to reveal possible bugs. It has also provided the basis for the proposal of upcoming IEEE P1647 functional verification language standard.

The implemented verification environment has been structured as a single ***e***VC™ (***e*** *Verification Component*) architecture under the recommendations of ***e***RM™ (***e*** *Reuse Methodology*) [7]. An ***e***VC can be seen as a plug-n-play, configurable verification environment, typically focusing on a specific protocol (e.g. P1500) that encompasses all necessary facilities for generation, checking and coverage analysis. The experimental simulations were carried out using SpeXsim™, which is a direct-kernel integrated verification platform and HDL simulator.

3.2 Verification Environment

Due to the nature of the P1500 protocol, the ***e***VC is designed to be flexible and extensible from the beginning of development. This flexibility allows for the user to add user-defined registers, instructions, checks and coverage items as well as enables future work. By design, the ***e***VC is able to achieve those goals with the least amount of effort and the highest degree of reuse.

3.2.1 *e*VC Modules

The main verification module for the P1500 ***e***VC is the *Agent*. An Agent performs Coverage Driven Verification (CDV) on a single P1500 wrapper, by driving input stimuli to the P1500 wrapper, performing checks on its outputs and collecting coverage information based on those outputs.

The components of the agent that realize CDV are discussed below (see Figure 1).

[1] more information at http://www.verisity.com

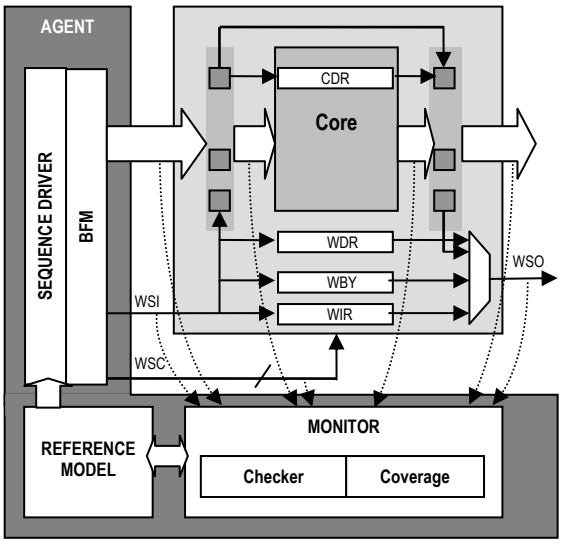

Figure 1: Verifying an IEEE P1500 Wrapped Core with Core Defined Registers

3.2.1.1 *The Sequence Driver*

This is the single user point-of-control for generating the input stimuli that will be passed to the verification environment. The sequence driver can generate input stimuli at three distinct levels of abstraction:

1. **P1500 event level** - Includes IEEE P1500 events {SHIFT_WIR, CAPTURE_WIR, UPDATE_WIR, SHIFT_DR, CAPTURE_DR, UPDATE_DR}
2. **Transaction level –** Includes complete transaction sequences, e.g. LOAD_INSTRUCTION
3. **Test level –** Includes sequences of transactions, such as W_EX_TEST_S_SHIFT_TEST (for a discussion, please refer to Section 4.1.2)

These abstraction levels and their relation to the signal level are depicted in Figure 2. Constrained-random generation is applicable on all three abstraction levels. This allows the verification engineer to identify corner cases using random generation, and then explore those corner cases using more constrained (directed) generation.

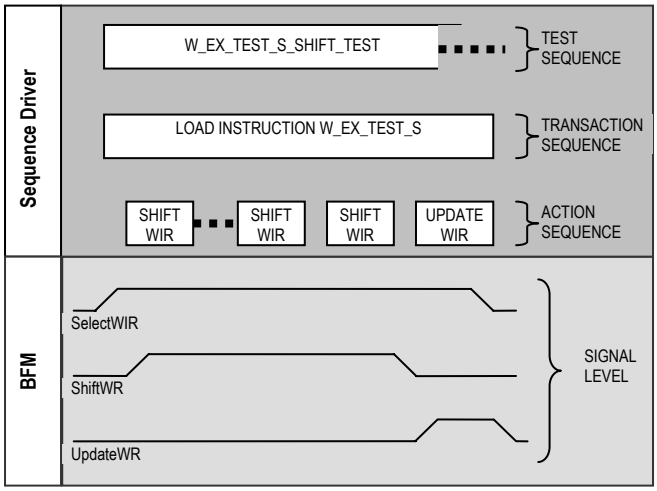

Figure 2: Vector generation at different levels of abstraction

The Sequence Driver also generates serial data on P1500 shift events and parallel data on P1500 capture events, which is part of the input stimuli and then passed to the BFM (explained below) to feed a wrapper's serial and parallel inputs.

3.2.1.2 *The BFM*

This component acts as a Bus-Functional Model and is responsible for driving all required signals to any P1500 wrapper using the input stimuli and associated data generated by the sequence driver.

3.2.1.3 *The Monitor*

This component is responsible for monitoring an individual P1500 wrapper's input and output signals, thus identifying the low-level P1500 events, which it then forwards to its sub-modules to perform checking and coverage collection as part of the CDV methodology.

- **Checker sub-module**: Used to check the wrapper's output signals. A number of checks are defined in the ***e***VC that can be divided into two main categories:
  - *Data Checks:* Used for integrity checking
  - *Protocol Checks:* Used to confirm protocol adherence

  The ***e***VC's extensible model allows for checks to be inherited in user-defined registers, to minimize effort and maximize reuse.
- **Coverage sub-module**: Used to extract coverage information based on a wrapper's reaction to input stimuli. This information is based on coverage metrics, explained in detail under Section 4.2 below.

3.2.1.4 *The Reference Model*

This component models the structure and behavior of a P1500 wrapper. It uses information passed on by the monitor to update its internal state and keep an up-to-date model representation of a P1500 wrapper. Developed with extensibility in mind, the reference model can support in addition to the mandatory spec as defined in the P1500 standard:

- Flexible register sizes
- Arbitrary number and size of instruction opcodes
- User defined, P1500 compliant instructions, registers, types of cells, cell isolation behavior and checks

The critical entity in the reference model is the cell. All actions depend on the cell's configuration and behavior and this is how the reference model is able to demonstrate such great extensibility, modeling any P1500 wrapper cell.

3.2.2 Supported Testing Scenarios

The aforementioned extensibility facilitates the support of arbitrary P1500 wrapper structure and behavior, which in turn, allows for a plethora of testing scenarios that are common on applications of the P1500 standard.

Such testing scenarios involve the use of a wrapped or unwrapped core, mandatory or user-extended instruction and register set. The ***e***VC also supports extended testing scenarios involving a chain of the aforementioned wrappers, or even a set of chains in system-level testing scenarios.

Finally, "legacy" support is also supported in the ***e***VC, as it can be configured to use JTAG (IEEE 1149.1) signals to drive control and data signals to a single, a chain, or a set of chains of P1500 wrappers with an embedded JTAG interface.

4 Verifying IEEE P1500 Wrappers

We now proceed to describe the process of verifying IEEE P1500-compliant wrappers based on CDV.

4.1 Test Plan

In this subsection we propose a test plan for verifying a chain of N P1500 wrappers ($N \geq 1$), connected in a daisy-chain, through their mandatory serial TAM. In the test plan, whenever a shift event is issued, the ***e***VC drives only the WSI of the first wrapper, since wrappers are connected in a chain. Whenever a P1500 event is issued, each reference model associated to a wrapper in the chain simulates the behavior of its wrapper by also "executing" the event. Thus, the checker of each wrapper is able to compare the values on WSO and register parallel outputs to the ones in the reference model and report potential miscompares. The test plan is realized by writing a set of tests (test suite) based on the environment presented in Section 3.2.

4.1.1 WIR integrity test

Since most tests require loading of instructions using the WIR of each wrapper it is good to first check the integrity of the WIR. There are two ways of writing a WIR: *shifting* in bits through its WSI and *capturing* bits through its optional parallel input. There is only one way of reading WIR contents: by *shifting* them out to WSO. Keeping in mind the above, we propose the following test sequences:

WIR_SHIFT_TEST. Shift bits in from WSI and check if they are shifted out to WSO through WIR correctly.[2]

WIR_CAPT_SHIFT_TEST. Have each WIR capture bits from its parallel input and then shift them out through WSO.

4.1.2 Data Register integrity test per instruction

Having verified WIR integrity for each wrapper we can proceed with testing data register integrity. As discussed in [1], each instruction is associated with a data register (i.e. the data register which connects WSI with WSO). However, each register (or part of it) can be used in more than one instructions, in each of which it may behave completely differently. For example, WBR is active for both W_EX_TEST_S and W_CORE_TEST_WS; however its cells face differently for each. Thus we'll verify the integrity of each register in the context of each instruction used.

The P1500 serial TAM is a one way path, beginning from a source and ending to a sink. There are no feedback loops. It is obvious that we can start by testing the instructions on the first wrapper. Once we verify its data registers' integrity, we can load it with an instruction that puts it in a preferred state and proceed to the testing of the next wrapper in the chain, using the first one as a path for the second's scan data. Continuing this way we test the instructions of every wrapper in the chain, after verifying its previous ones operate correctly.

For testing the *Mth* wrapper, $M \leq N$, we should decide the instruction with which to load its previous *M-1* wrappers. It's good if we use an instruction that only responds to SHIFT_DR events, ignoring CAPTURE_DR and UPDATE_DR events, because we want it just for passing scan data through the wrappers. Also, we'd like the instruction chosen to use the shift register with the smaller possible size so that we reduce the overhead bits added on the actual scan data thus reducing test simulation time. The mandatory instruction that has these two properties is W_BYPASS: responds to SHIFT_DR events only and uses WBY, the shift register with the smaller possible size[3].

The wrappers *after the Mth* one can be loaded with any instruction. This is because there are no feedback paths in the chain. But, for uniformity reasons we load them with the same instruction as the first *M-1* ones (i.e. W_BYPASS).

We now propose test sequences for some of the instructions defined by the IEEE 1500 Proposal.

4.1.2.1 *W_BYPASS integrity test*

When a wrapper is loaded with W_BYPASS, the data register connected between WSI and WSO is WBY which can only be shifted. So, we can define a test sequence like the first one for WIR described above:

W_BYPASS_SHIFT_TEST. Shift bits in from WSI and check if they are shifted out to WSO through WBY correctly.[1]

4.1.2.2 *W_EX_TEST_S integrity test*

When a wrapper is loaded with W_EX_TEST_S, WSI is connected to WSO through WBR. WBR can be shifted with SHIFT_DR events. A CAPTURE_DR causes the input cells of WBR to capture their parallel functional inputs. UPDATE_DR causes the output cells of WBR to output their shift/capture stage contents to their parallel functional outputs, provided they have an update stage. So, WBR can be written and read with shifts. The input cells of WBR can be written by issuing capture events too. Finally, the output cells of WBR can be read by issuing update events too. We can test integrity of WBR in W_EX_TEST_S using all combinations of WBR reads and writes:

W_EX_TEST_S_SHIFT_TEST. Shift bits in from WSI and check if they are shifted out to WSO through WBR correctly. [1]

W_EX_TEST_S_CAPT_SHIFT_TEST. Have WBR input cells capture bits from their parallel, functional inputs and then shift them out through WSO.

[2] Consecutive shifts should have random length each time to check WIR behavior to small size shifts and also make sure all WIR cell values have been output by having large size shifts. This is also true for shift test sequences proposed for data register later, i.e. W_BYPASS_SHIFT_TEST and W_EX_TEST_S_SHIFT_TEST.

[3] We expect that WBY will typically consist of 1 or 2 bits

W_EX_TEST_S_SHIFT_UPD_TEST. Shift bits in from WSI and get the respective ones out from the parallel, functional outputs of WBR output cells, by using the update event. If WBR output cells don't have update stage the update event is not necessary to be issued because shift/capture stage contents are always reflected to the cell outputs.

4.1.2.3 *P1500 optional and user-defined instructions integrity test*

The verification environment is capable of generating stimuli at three levels of abstraction: P1500 events, sequence of P1500 events, high-level tests. Any other instruction not described here, either proposed by the 1500 working group or user defined, can be tested using the data register approach we followed for the tests presented and the environment's powerful generation capabilities.

4.1.3 Test suite

We have created a test suite based on the test plan proposed. Tests have been implemented using a sequence library defined in the environment. The sequence library defines meaningful combinations of basic P1500 events to be sent to DUTs (e.g. the sequence of SHIFT_WIR's and UPDATE_WIR events that load instructions to all wrappers in a chain). We will now proceed in defining interesting functional cover metrics by which we will evaluate the test suite, hence the test plan proposed, through two real-life application scenarios.

4.2 Functional Coverage

Functional coverage provides a metric of progress in the verification approach. It allows us to determine if tests exercise different parts of the design functionality and avoid running tests that do not contribute to the verification progress. Hence, functional coverage is a quality judge of the test plan. After applying the test suite to the DUT and analyzing functional coverage results, corner cases may be revealed that can lead to the design of new tests using altered generation constraints.

We demonstrate a representative set of functional coverage metrics that can be used for measuring verification progress of black-box P1500 wrappers, i.e. wrappers for which we have no information on the way their cells and control logic have been designed and no observability of wrapper internal signals. It is obvious that the same metrics can be applied to white-box or partially white-box implementations. Of course, access to white-box wrapper internal structures can lead us to the definition of more coverage metrics giving a better insight of the functionality that has been exercised by the tests. The extensibility feature of our environment allows us to define new coverage metrics *a posteriori* with little effort. Finally, the set is suitable to measure coverage in a multi-wrapper scenario, in which all wrappers are connected through their mandatory serial TAM in a daisy-chain way. Results will be represented on a per wrapper basis for metrics that may vary among wrappers. We now proceed to define a set of functional coverage metrics; this set should not be considered exhaustive:

Instructions loaded **[per wrapper]**. The reference model is capable of discovering which instruction will be loaded to its wrapper upon each UPDATE_WIR event. In a multi-wrapper scenario, the instructions loaded vary among the wrappers so information is gathered on a per wrapper basis. This metric will help us discover if there are untested instructions in any wrapper.

Instruction transitions **[per wrapper]**. The coverage collector of our environment is capable of recording previous and present values of the metrics defined. In the case of instructions, upon an UPDATE_WIR event, the coverage collector also remembers the previous instruction loaded and provides us with a list of instruction pairs that were loaded one after the other. This transition metric will help us answer questions like: have we loaded all possible instructions after W_BYPASS to a wrapper?

P1500 events applied. P1500 events are caught by the environment's monitor and coverage information is unique for all wrappers in the chain. This is because wrappers connected in a daisy-chain way, using the mandatory serial TAM, share the same control lines of the TAM. This metric helps us discover if we have applied all possible P1500 events to the wrappers.

P1500 event transitions. Using the coverage collector's capability of recording metric transitions we cover all possible transitions of P1500 events. Again, this metric is unique for all the wrappers in a chain. We can hence answer questions like: have we applied all P1500 events after a CAPTURE_DR?

Instructions × P1500 events **[per wrapper]**. Another feature of the environment's coverage collector is the ability to cross two or more metrics. We thus implemented the cross coverage metric of instructions and P1500 events per wrapper. Notice that in a multi-wrapper scenario this cross metric has to be presented on a per wrapper basis since the ***instructions*** metric varies per wrapper. This metric will help us discover if all P1500 events have been tested for each instruction loaded on every wrapper.

4.3 Application Testing Scenario

We will now present the results of application testing under a typical dual-wrapper chain configuration scenario. Our aim is to illustrate the effectiveness of CDV, by applying coverage results obtained in early simulations to tune our input stimuli.

The scenario includes two P1500 wrappers connected in a daisy-chain (see Table 1 for wrapper characteristics). The WBR topology for both wrappers is defined as WSI → WBR Input Cells → WBR Output Cells → WSO for simplicity. The instruction set used is {W_BYPASS, W_EX_TEST_S, W_CORE_TEST_WS, W_PRELOAD_S}. WSI of wrapper A is driven by the ***e***VC, while WSI of wrapper B is driven by WSO of wrapper A. The ***e***VC drives the wrappers' parallel inputs and also provides clock and reset signals.

Register	Wrapper A	Wrapper B
WIR	5 cells	4 cells
WBY	1 cell	2 cells
WBR	100 cells* 50 I/P - 50 O/P	70 cells* 40 I/P - 30 O/P

**All WBR cells have an update stage*

Table 1: Wrapper Configuration for Application Scenario

4.4 Coverage Results and Test Suite Evaluation

After running the test suite implementing test plan of Section 4.1 on the testing scenario earlier presented, we got functional coverage results and analyzed them. We discuss results on the metrics defined in Section 4.2 below.

The *"Instruction loaded per wrapper"* coverage metric (not presented here) has shown that all instructions have been loaded to both wrappers. Table 2 shows the results from coverage metric *"instruction transitions per wrapper"*. Hits per wrapper are presented in the two columns named **Hits (initial)**. Observe that not all instruction transitions have taken place, as shaded cells indicate. This is because W_BYPASS, UNSUPPORTED and only one of the rest instructions are loaded in a wrapper throughout a certain test. So, for example, we haven't been able to test the behavior of *wrapper A* when moving from W_EX_TEST_S to W_CORE_TEST_WS in the same test run.

The third metric, *"P1500 events"* (not shown here), indicated that all P1500 events have been issued. However, the fourth metric, *"P1500 event transitions"* (not shown here), revealed that not all P1500 events have been observed. This metric is closely related to the functionality of the *WIR decoder* which seems to have not been covered enough as the holes in the metrics reveal.

Instruction Transitions		Hits (initial)		Hits (new)	
Previous	*Next*	*Wr A*	*Wr B*	*Wr A*	*Wr B*
W_BYPASS	W_BYPASS	133	139	346	695
	W_EX_TEST_S	1	0	26	9
	W_CORE_TEST_WS	1	1	32	5
	W_PRELOAD_S	1	1	23	8
	UNSUPPORTED	139	152	166	168
W_EX_TEST_S	W_BYPASS	1	0	25	2
	W_EX_TEST_S	92	76	119	90
	W_CORE_TEST_WS	0	0	15	10
	W_PRELOAD_S	0	0	20	9
	UNSUPPORTED	0	0	21	4
W_CORE_TEST_WS	W_BYPASS	1	0	27	7
	W_EX_TEST_S	0	0	19	6
	W_CORE_TEST_WS	78	85	103	97
	W_PRELOAD_S	0	0	8	8
	UNSUPPORTED	0	0	21	3
W_PRELOAD_S	W_BYPASS	0	0	19	15
	W_EX_TEST_S	0	0	14	6
	W_CORE_TEST_WS	0	0	16	6
	W_PRELOAD_S	77	66	111	76
	UNSUPPORTED	1	0	21	6
UNSUPPORTED	W_BYPASS	138	150	174	161
	W_EX_TEST_S	0	1	22	6
	W_CORE_TEST_WS	0	0	12	4
	W_PRELOAD_S	0	0	19	9
	UNSUPPORTED	131	123	163	132

Table 2: Functional Coverage – Cumulative Instruction Transitions per Wrapper

In order to cover holes that the coverage analysis revealed and increase the number of random resets, we designed a new test which performs the following:

INSTR_REGRESSION_TEST. For every wrapper in the chain:

- it loads its previous and next wrappers with W_BYPASS and then repeatedly
  - ✓ loads current wrapper with a random instruction and
  - ✓ issues random number of sequences of random P1500 events
- also, at random intervals it resets the chain of wrappers.

After running this test and aggregating coverage reports from this and the initial test suite, we got the results shown in the **Hits (new)** column. Observe that we managed to hit every uncovered case that the initial test suite did not. Metrics not shown in tables have also increased their hits and in the case of *"P1500 transitions"* all uncovered combinations have been also hit.

5 Conclusions

In this paper, we presented the need for deploying advanced verification methodologies in contemporary SoCs. The IEEE P1500 Standard for Embedded Core Test provides a solid test infrastructure for such SoCs, enhancing reusability and interoperability of complete core.

In our approach, we described a comprehensive verification environment for IEEE P1500-based test infrastructures implemented as an ***e***VC under Verisity's SpeXsim platform. The environment employs techniques such as constrained-random vector generation, automated checking and coverage-driven verification to be able to fully verify such infrastructures under virtually any configuration. Combined with a well defined verification strategy and test plan, this approach offers clear advantages over traditional testbenches, illustrated by the functional coverage measurements obtained on a typical IEEE P1500 chain verification scenario. In addition, the environment can be reused across levels of design abstraction and project flows.

References

[1] E. J. Marinissen, R. Kapur, M. Lousberg, T. McLaurin, M.Ricchetti, and Y. Zorian, "On IEEE P1500's Standard for Embedded Core Test," *Journal of Electronic Testing*, vol. Theory and Applications, pp. 365-383, 2002.

[2] Y.Zorian, E.J.Marinissen, and S.Dey, "Testing embedded-core based system chips," presented at ITC - International Test Conference, 1998.

[3] K. Melocco, H. Arora, P. Setlak, G. Kunselman, and S. Mardhani, "A Comprehensive Approach to Assessing and Analyzing 1141.1 Test Logic," presented at ITC - International Test Conference, Charlotte, NC, USA, 2003.

[4] S. Picchiottino, M. Diaz-Nava, B. Foret, S. Engels, and R. Wilson, "Platform to Validate SoC Designs and Methodologies Targeting Nanometer CMOS Technologies," presented at IP/SoC, Grenoble, France, 2004.

[5] T. McLaurin and S. Chosh, "ETM10 Incorporates Hardware Segment of IEEE P1500," *IEEE Design & Test of Computers*, pp. 8-13, 2002.

[6] D. Appello, F. Corno, M. Giovinetto, M. Rebaudengo, and M. S. Reorda, "A P1500 Compliant BIST-Based Approach to Embedded RAM Diagnosis," presented at 10th Asian Test Symposium (ATS'01), Kyoto, Japan, 2001.

[7] *e Reuse Methodology Manual*: Verisity, 2003.