Are IEEE-1500-Compliant Cores Really Compliant to the Standard?

Alfredo Benso
Politecnico di Torino

Stefano Di Carlo and Paolo Prinetto
Politecnico di Torino

Alberto Bosio
LIRMM

Editor's note:
Functional verification of complex SoC designs is a challenging task, which fortunately is increasingly supported by automation. This article proposes a verification component for IEEE Std 1500, to be plugged into a commercial verification tool suite.

—Erik Jan Marinissen, IMEC

The continual scaling of semiconductor technology represents the basis for SoC integration. Consequently, the race to market high-quality products with an ever shorter design-to-manufacturing cycle compels designers and manufacturers to emphasize the reuse of IP-embedded cores into SoC design. Core reuse also implies the ability to reuse the core test in the SoC test. IEEE Std 1500 for Embedded Core Test is currently one of the most effective solutions for supporting SoC manufacturing test,[1] and it also allows easy interoperability among different cores.[2]

The standard defines a set of guidelines to build a scalable and standard test data interface (*core test wrapper*), completed with an information model describing the implemented test features. The IEEE 1500 information model, described using the IEEE Std 1450.6 Core Test Language (CTL),[3] expedites the transfer of test data from core providers to core integrators.

An entire set of challenges arises when we consider the heterogeneous nature of today's IP market. Core providers must provide IEEE-1500-compliant cores to integrators to facilitate customer test integration into system-level infrastructures. Core integrators, on the other hand, must ensure that IEEE-1500-ready IP blocks properly comply with the standard's required functionalities. The question for both designers and integrators is whether IEEE-1500-compliant cores really are compliant to the standard.

The design of an IEEE 1500 core test wrapper includes several steps whereby bugs might be introduced. To support a wide range of test applications, IEEE 1500 defines only a minimal set of mandatory hardware features, giving designers the freedom to extend the test infrastructure with virtually unlimited sets of registers and instructions.[2,4] A comprehensive approach to thoroughly verify the functionality of IEEE 1500 core test wrappers in a SoC environment is therefore mandatory.

The problem of verifying an IP core's compliance to IEEE 1500 has been poorly addressed in the literature. To solve the problem of verifying an IP core's compliance to IEEE 1500, Globetech Solutions (http://www.globetechsolutions.com) has proposed a commercial answer.[5-7] In part, for example, Diamantidis et al. addressed IEEE 1500 in proposing a unified DFT verification methodology to provide a complete, methodical, and automatic verification flow for SoC DFT infrastructures.[5] The authors showed how to build and manage a database of reusable verification components, targeting different DFT techniques. The database facilitates the implementation of a complete SoC verification plan. The authors only briefly mentioned IEEE 1500 verification as an example of a verification component.

Elsewhere, Diamantidis et al. and Oikonomou et al. introduced an IEEE 1500 compliancy verification

0740-7475/09/$25.00 © 2009 IEEE Copublished by the IEEE CS and the IEEE CASS

component based on dynamic functional verification.[6,7] The component performs verification by comparing the given design with a reference model. One of this solution's key advantages is the ability to consider the verification of both isolated and daisy-chained wrappers. Nevertheless, although the authors claim their verification flow is completely automated, they provide little information on how the verification component is actually configured, and their verification plan is predefined, which means the plan might be not optimal when looking at the overall SoC verification, and therefore it might prevent the reduction of overall SoC verification time. Moreover, they do not completely specify how the reference model is implemented, and it is not clear whether the model can systematically address every aspect of the standard. For example, it is not clear how that model verifies the IEEE 1500 information model, which is a relevant part of the standard. Finally, it is not clear how that model measures overall compliancy to the standard.

Recently, we presented a Unified Modeling Language (UML) abstraction of an IEEE 1500 verification framework.[8] The main contribution of that work was the definition of a rule-based approach for IEEE 1500 compliancy verification that systematically addressed each design rule imposed by the standard. In particular, besides focusing on the implementation alternatives, we proposed a design rule classification on the basis of the methodology required for their verification, and we defined a detailed abstract model of the framework.

In this article, we present a complete implementation of the abstract framework model proposed previously.[8] We show how to map most of the verification tasks required by that model into the functionalities of a commercial verification tool such as Verisity's Specman Elite (http://www.verisity.com/products/specman.html; Verisity is now part of Cadence). In particular, we will show how

- IEEE Std 1647 Functional Verification Language e (*e* for short) can be easily used to write reusable rule verification components;[9]
- the embedded interface of Specman Elite with a commercial simulator can be used to verify a critical class of design rules; and
- coverage metrics the tool provides can be used to measure the quality and completeness of the verification process.

We also present an application of the proposed prototype verification framework to a benchmark SoC's validation. The outcome of this article is a verification component that designers and integrators can insert into a global SoC verification strategy such as that described by Diamantidis et al.[5]

Verification flow

Figure 1 introduces the basic steps of our proposed verification flow. The goal is to see if an IEEE-1500-compliant core fully respects (observes) the design rules defined by the standard. By "IEEE-1500-compliant core" (*core* for short), we mean an IP core that incorporates an IEEE 1500 core test wrapper,

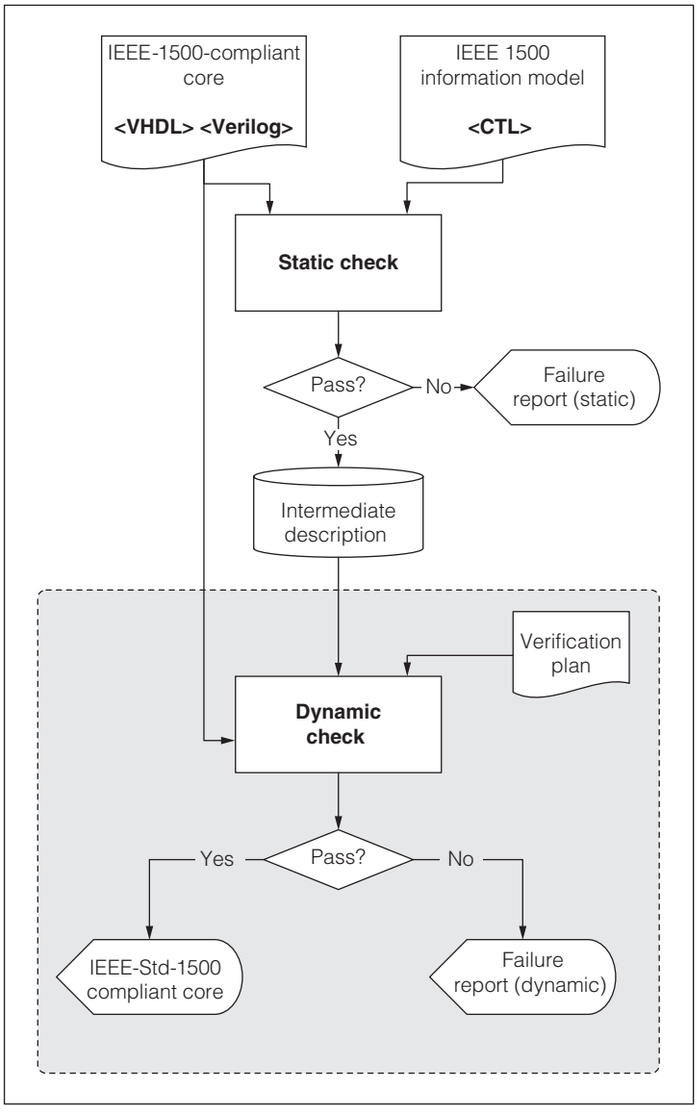

Figure 1. IEEE 1500 verification flow.

and comes with a corresponding CTL file (which is the IEEE 1500 information model) that describes the core test knowledge, including how to operate the wrapper at its external terminals. We consider cores provided using hardware description languages (HDLs) such as Verilog, VHDL, and so on.

The IEEE 1500 verification process consists of two distinct phases: static and dynamic check. The static check is a preliminary verification task to verify the syntax of the core description and the related CTL file. This check is particularly important for the CTL file. Although the core's HDL description is usually automatically synthesized and should not contain syntax errors, CTL files generation is usually not automated. The core designer should therefore identify any incorrect CTL files, which may contain errors or CTL dialects that are not fully compliant with the standard, before moving on to the next verification steps. The syntax analysis can be efficiently performed using special programs called lexers and parsers[10] automatically built over a formal grammar—for example, Vern Paxson's JFlex (http://jflex.de) and Scott Hudson's CUP (http://www2.cs.tum.edu/projects/cup).

Besides verifying the syntax analysis, another goal of the static check is to collect IEEE-1500-related information and convert it to a suitable format for the next verification steps (the "intermediate description" in Figure 1 refers to this process). Information collected during this phase includes the wrapper's signals and register names, implemented instructions, and so on. Because the static check phase is not complex, we don't provide additional details on its implementation but focus instead on the more interesting, dynamic aspects of verification.

To understand the tasks to be performed during the dynamic check phase, we must first recall the characteristics of IEEE 1500 as Benso et al. modeled them.[8] The standard, a collection of design rules for a core test wrapper realization,[1] identifies two classes of wrapper aspects that must be verified: semantic aspects and behavioral aspects.

Semantic aspects concern the CTL IEEE 1500 information model and do not depend on the core behavior. Verifying semantic aspects involves identifying the correct definition of structures such as scan chains, macros, and environments in the information model. These can be easily verified by analyzing the CTL file provided with the core. Although this is a fast and powerful way to verify part of IEEE 1500 compliancy (because it does not require any simulation of the core itself), it is not enough to guarantee the core's complete verification. Semantic aspects are only a relatively small subset of the whole set of rules to verify. Moreover, semantic analysis is performed on information model data supplied by the core provider, so there is no guarantee that the data perfectly matches the actual hardware implementation. An efficient way to implement semantic aspects verification is to store the intermediate description in Figure 1 into a relational database. The verification can then be easily translated into a set of queries performed on the database.

Behavioral aspects target communication protocols and core behavior, and they are the most difficult to verify. Their verification requires core simulation. In fact, simulation is the only effective approach to verify timing, protocols, signal connections, and correct implementation of instructions. ATPG tools perform a similar task, for example, during design rule checking. Nevertheless, these tools must deal with well-known test structures inserted into the circuit, thus reducing the verification activity's complexity. With respect to IEEE 1500, the way the core test wrapper is implemented relates strictly to the target design, which makes verification more complex. Accordingly, in this article we exploit dynamic functional verification for this task.[11]

Finally, an overview of the verification flow would be incomplete without a few considerations on how both semantic and behavioral aspects can be derived from the standard. IEEE 1500 design rules are provided in a natural-language format. By thoroughly analyzing the standard, we can identify the following information for each rule:

- *Rule type.* Does the rule identify a behavioral aspect, a semantic aspect, or both?
- *Involved units.* Which wrapper elements does the rule involve (wrapper instruction register, wrapper bypass register, and so forth)?
- *Dependence.* What dependence does this rule have with other rules?
- *Observation points.* This concerns the wrapper's elements—that is, signals, and units, where a potential violation of the rule can be observed.
- *Actions.* A natural-language description of the actions required to verify the rule.

Starting from this structured description, we have translated each action into the appropriate

formalism—for example, the *e* language for behavioral aspects—to build the verification framework.

Dynamic functional verification

By *functional verification* we mean the task of demonstrating that the intent of a design is preserved in the design's implementation.[11] The most widespread method of functional verification is *dynamic functional verification*. It is called dynamic because input patterns are generated and applied to the design by simulation, and the corresponding results are collected and compared against a reference model for checking their conformance with the specifications.

Verifying all possible behaviors under every possible combination of inputs is, in most cases, unfeasible because the test space is too large to be fully covered in a reasonable amount of time. To overcome this problem, one of the best solutions is to apply *constrained-random* pattern generation. Verification patterns are randomly generated under a set of constraints, limiting the set of legal values on the input signals that drive the design. Moreover, we could also apply *coverage-driven* verification. Functional coverage metrics are automatically stored in real time to ascertain whether, and how effectively, a particular test verifies a given feature. This information can then be fed back into the generation process to drive additional verification effort toward the required goal. Coverage metrics are evaluated on coverage monitoring points defined by the user and specified in the verification plan.

The market offers various tools to support dynamic functional verification—for example, Verisity's Specman Elite and Synopsys' Vera. Specman Elite (Specman for short), which is our reference verification environment, uses the *e* language to capture behaviors defined in the specifications and to automatically generate tests. Designers use the *e* language to write *e* verification cores (eVCs)—that is, software modules modeling the functional behavior of the environment surrounding the system under verification.

IEEE 1500 behavioral-aspects verification

The verification of IEEE 1500 behavioral aspects starts from the definition of an eVC able to fully stress the target core and to compare the obtained results with a core test wrapper reference model. Besides modeling a generic reference IEEE 1500 core test wrapper (which is almost impossible, due to the number of customizations allowed by the standard), we accordingly consider all IEEE 1500 design rules and determine whether the target design actually respects all rules.[8] This will in turn require us to

- translate rule design constraints into portions of *e* code that can generate verification patterns and verify the rules' correctness; and
- identify, for each rule, the related coverage points—that is, wrapper signals or registers used to evaluate the coverage reached during the verification process.

The challenge stems from the definition of rule verifiers and rule coverage points independent of the specific wrapper design. This definition strongly depends on the core internal structure's level of controllability and observability. In a white-box design, which is customarily what core designers develop, the core internal structure is completely accessible. Internal core signals can be fully controlled and observed; therefore, designers can fully verify all IEEE 1500 rules. In a black-box design—the usual situation for core integrators, who buy cores from different vendors—the only available information for IP protection is the core I/O interface. This strongly impacts Specman's ability to generate verification patterns and evaluate the verification coverage. The only rules that can be fully verified are those that do not require direct controllability or observability of core or wrapper internal signals. Full IEEE 1500 compliancy verification can thus be achieved only for white- or black-box designs that implement the standard's basic functionalities.

IEEE 1500 eVC architecture

Here, we highlight the IEEE 1500 eVC architecture used to verify IEEE 1500 behavioral aspects. The full verification is based on a single verification component, according to the recommendations of the *e* Reuse Methodology (eRM; http://www.verisity.com/products/erm.html). We refer to IEEE 1500 rules with their corresponding rule number.[1]

Figure 2 sketches the architecture of the proposed eVC, highlighting for each block the related *e* files. The `IEEE_1500_env` (IEEE 1500 environment) built over the predefined `any_env` unit (a default environment used as the starting point to build eVCs), according to eRM, represents the overall eVC

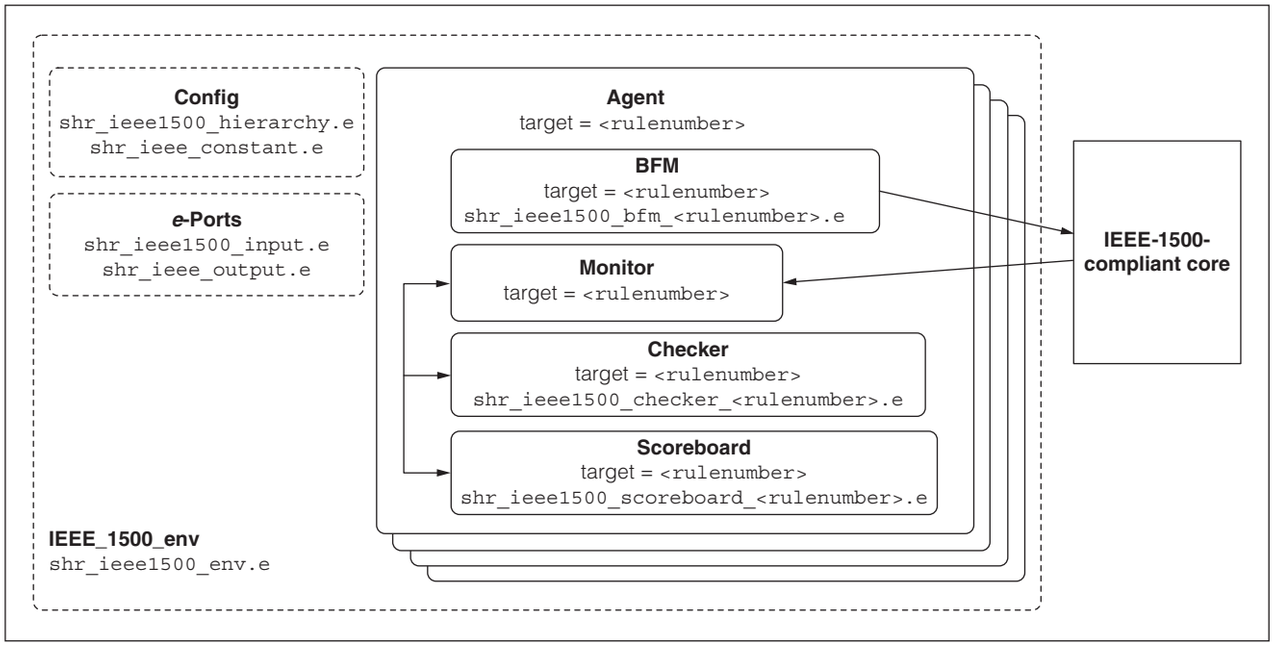

Figure 2. Structure of the IEEE Std 1647 Functional Verification Language *e* verification core (eVC).

environment. The environment defines the verification events required to perform the verification. For each design rule under verification, the environment defines a start event to begin the rule verification and an end event to control the verification result. Figure 3a shows an example of `shr_ieee1500_env.e`, including a checkpoint flag to stop the verification if one of the defined rules is not observed.

The eVC ports are defined through the `shr_ieee1500_input.e` and `shr_ieee1500_ouput.e` files. The core is then configured through the `shr_ieee1500_constant.e` and `shr_ieee1500_hierarchy.e` files. The `shr_ieee1500_constant.e` file provides the eVC enough information to apply verification patterns at the core inputs, monitor the corresponding outputs, and store core-specific information, such as wrapper signal names, register sizes, and so on. The generation of this configuration file can be easily automated, starting from the intermediate description of the target core (see Figure 1). This file provides the basis for guaranteeing the eVC's reusability.

The `shr_ieee1500_hierarchy.e` file is one of the eVC's most important elements. It defines the verification plan—that is, what must be verified and in which order. It describes the scope of the verification problem and serves as the functional specification for the verification environment, highlighting dependencies among results of different verification steps. Moreover, it lets us optimize the verification time, thus reducing overall verification costs. Figure 3b is a small verification plan example. Each time the verification of a given rule ends—that is, the `end_rulenumber` event occurs—the verification plan identifies the next rule to verify. When there is rules dependency, the result of a specific rule verification might modify the verification plan. For example, consider rule `7_4_1_a` in Figure 3b: its verification starts only if rule `10_3_1_j` is correctly verified. Moreover, rule `10_3_1_j` is verified after rule `10_3_1_a3`, regardless of whether the verification of rule `10_3_1_a3` was successful. Users can freely modify this file to build their own verification plan.

The actual verification is then performed by instantiating multiple eVC agents. The agent has a subtype target to identify the target IEEE 1500 design rule. Each agent instantiates a Specman bus functional model (BFM) and a monitor. The BFM generates and simulates the verification patterns for the target rule, and the monitor evaluates the simulation results to understand whether the rule is respected (by using the subtype target). The monitor includes a scoreboard to perform a static analysis of the simulation results (that is, it verifies whether or not the core output signals assume the proper sequence of values regardless of their actual timing), and a

checker to perform the timing analysis. Moreover, the BFM defines the set of coverage items used to evaluate the final verification coverage.

To better understand the proposed verification strategy, consider the following behavioral rule corresponding to Chapter 10.3.1, rule (e), of IEEE 1500[1] (WIR: wrapper instruction register; WRCK: wrapper clock; WRSTN: wrapper reset):

> *The WIR circuitry shall retain its current state (i.e., shift stage values and currently active modes) indefinitely while the WRCK signal is stopped (i.e., WRCK held at a fixed logic value of 1 or 0) and the signal connected to the WRSTN terminal is logic 1.*

The verification of this rule requires that we

1. Fetch one instruction.
2. Force WRCK to 0 (1).
3. Force WRSTN to 1.
4. Drive all the wrapper's input signals except WRCK and WRSTN.
5. Check that the WIR circuitry retains its state.

All operations are repeated for the 11 mandatory or optional instructions defined by the standard.[1] Item 4 resorts to random generation to efficiently drive a not-involved signal. Figure 3c shows the *e* code of the function in charge of randomly generating a vector and of driving it to the device under test. The vector drives all signals specified in `shr_ieee1500_constant.e`. In general, each rule determines a set of signals with a deterministic fixed value, while for the remaining signals, we apply random generation. Because of the proposed eVC's open architecture, users can extend the random generation function while considering the specific core functionality in order to cope with corner cases. Item 5 is finally performed by the checker; it ensures that the core respects the rule.

```
event start_13_1_1_a;
event end_13_1_1_a;
event start_7_2_1_c;
event end_7_2_1_c;
event start_7_2_1_b;
event end_7_2_1_b;
checkpoint: bit;
keep soft checkpoint == 0;
```

(a)

```
on end_10_3_1_a3 {
  emit start_10_3_1_j;
};
on end_10_3_1_j {
   if checkpoint == 1 {
     stop_run;
   } else {
     emit start_7_4_1_a;
   };
};
```

(b)

```
<'
gen_and_drive (sig_size: uint, sig_name: list of string, is_wire:
list of bit) is {

   var i: uint;

   gen input_vector keeping {

      -- Generates a random vector

      .size()==sig_size;
   };

   for i from 0 to sig_size - 1 do {

      -- Assigning generated values to sig_name list of signals

      drive_value(sig_name[i], is_wire[i], pack(packing.low,

                  input_vector[i:i]));
}; '>
```

(c)

Figure 3. eVC code snapshots: eVC environment (a), hierarchy (b), and rule verification example (c).

A complete analysis of IEEE 1500 enabled us to identify a set of 165 mandatory rules and 27 optional rules. Our current prototype eVC component implements the verification of 138 mandatory rules summarized as follows: 40 semantic rules, 25 behavioral rules, and 73 mixed rules, including both semantic and behavioral aspects. Moreover, we can readily extend eVC to address new rules for optional properties of the standard by adding additional agent instantiations for the new rules and by inserting their verification in the overall verification plan.

Experimental results

We have tested the effectiveness of our proposed verification flow on a benchmark SoC consisting of an 8080 8-bit microprocessor, and an 8-bit programmable interrupt controller (PIC) from OpenCores (http://www.opencores.it), complete with a 256 bit × 8 bit

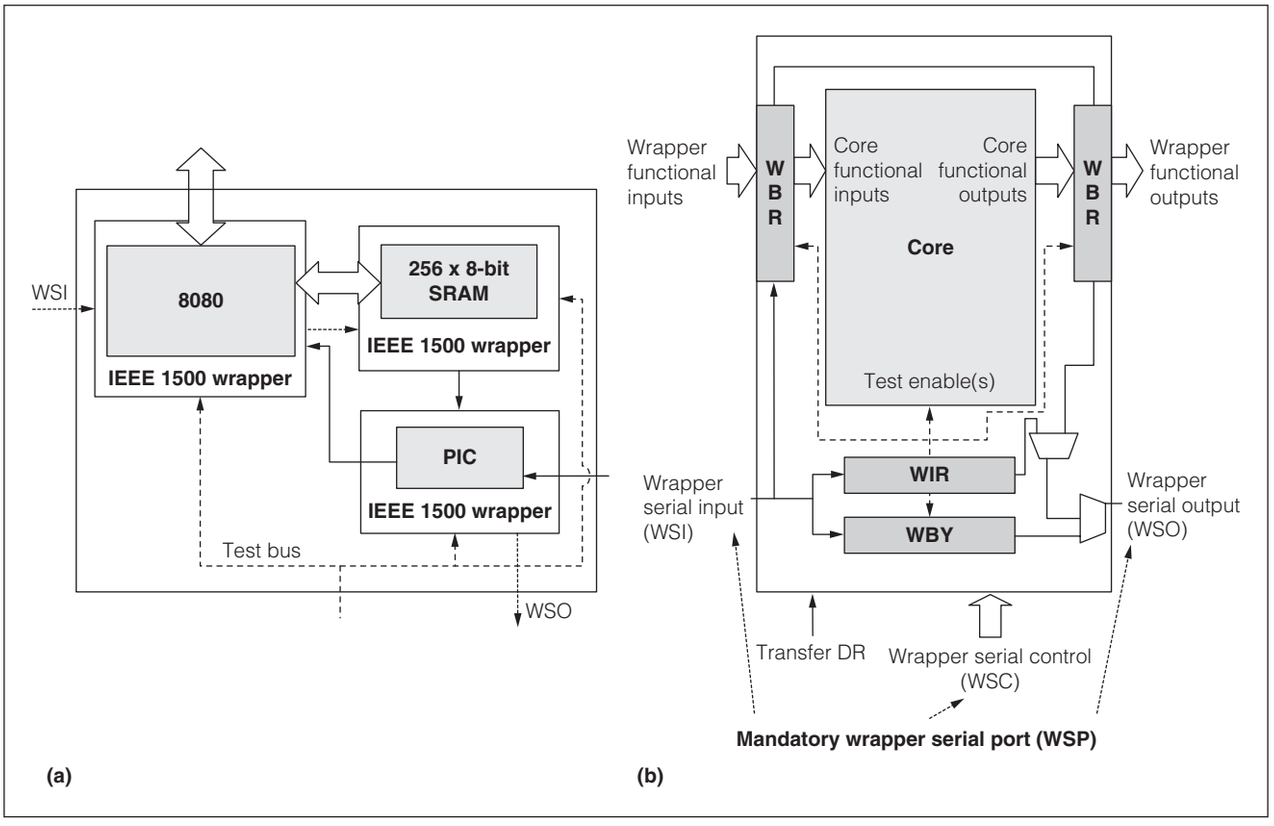

Figure 4. Test case general architecture: SoC architecture (a) and wrapper architecture (b).

SRAM. Each core has an IEEE 1500 core test wrapper with the following characteristics:

- a wrapper instruction register (WIR), 3-bit length;
- a wrapper bypass (WBY) register, 1-bit length;
- a wrapper boundary register (WBR);
- an optional *TransferDR* wrapper serial control; and
- four implemented instructions: `WS_BYPASS`, `WS_PRELOAD`, `WS_INTEST`, and `WS_EXTEST`.

Figure 4a shows the basic SoC architecture, and Figure 4b shows the wrapper architecture. Table 1 shows the SoC complexity in terms of gates.

We applied the complete verification flow to the three SoC cores using a Sun Blade 1000 workstation, and 100% of the implemented rules were correctly verified. Table 2 shows the verification time for each core in terms of clock cycles and CPU time. The simulation time is obviously strictly connected to the core's complexity.

In addition to providing the list of verified rules, Table 3 shows an example of verification coverage analysis on five rules. Verification coverage metrics are an efficient and easy way to assess the result of verification. For each IEEE 1500 rule and defined coverage items that we considered (in column 1), we

Table 1. Synthesized SoC characteristics.

Block type	Number of blocks
Combinational gate	11,292
Nonscan flip-flops	3,092
RAM blocks	1 (256 bits × 8 bits)

Table 2. Simulation time.

Core	Clock cycles	CPU time (s)
8080	207,453	840
SRAM	181,751	730
PIC	18,112	120

* PIC: programmable interrupt controller.

analyzed the actual number of measured hits for the three circuits—8080, SRAM, and PIC—with respect to the number of required hits (RH).

Coverage items and RH depend on the target rule. For example, consider rule 12.1.1.b:

> *The WBR shall have at least one configuration in response to the state of the WIR, allowing serial access to and from all WBR cells between TI and TO.*

This rule implies shifting data through the WBR cells, and requires a number of WBR shift events at least equal to the WBR length. This metric was respected in the three analyzed cores (see Table 3). Table 3 also shows that RH was always reached for all rules, ensuring the reliability of the verification framework.

Table 3. Coverage results.

Rule, coverage item	No. of hits (8080), RH	No. of hits (SRAM), RH	No. of hits (PIC), RH
10.1.1.c, instruction fetch	4, 4	7, 4	6, 4
12.1.1.b, WBR shift	24, 24	16, 16	8, 8
10.3.1.g, WRSTN transition	4, 4	4, 4	13, 4
13.1.1.c, WRSTN set	17, 4	4, 4	4, 4
11.1.1.b, WBY register select	1, 1	15, 1	1, 1

* WBR: wrapper boundary register; WBY: wrapper bypass; WRSTN: wrapper reset.

Finally, to carefully validate the verification capabilities of the framework, we designed a set of different wrappers, systematically violating different rules of IEEE 1500. Table 4 summarizes the verification results for each of these experiments. The first column reports the target rule number, the second column describes the injected error, and the third column reports the verification result. All violations

Table 4. Rules failure tests.

Rule	Violation	Detected?
7.2.1.c	No register selected during **`WS_PRELOAD`**	Yes
7.2.1.e	Modified signals configuration for the update WIR register	Yes
7.4.1.c	Inverted wrapper *reset*	Yes
7.4.1.d	WBR not in functional mode during **`WS_BYPASS`**	Yes
7.4.1.e	WBY register shift signal stuck at 0	Yes
10.1.1.b	Last flip-flop of WBR not connected to WSO	Yes
10.2.1.a	WBR register never selected	Yes
10.2.1.b	WIR mode signal stuck at 0	Yes
10.2.1.c	WRSTN connected to one of the core inputs	Yes
10.2.1.d	Unconnected wire in WIR	Yes
10.2.1.e	Inverted WSI during WIR shift	Yes, 10.3.1.h fails
10.2.1.f	Update operation on every shift	Yes, 10.3.1.a3 and 10.3.1.j fail
10.3.1.a	**`WS_BYPASS`** not selectable	Yes, 10.3.1.g, 13.1.1.b, 7.3.1.h, and 13.1.1.d fail
10.3.1.b	*updateWR* signal of WIR combined with one or more core inputs	Yes
10.3.1.d	*updateWR* signal of WIR combined with *captureWR*	Yes
10.3.1.e	WIR shifts without clock	Yes
10.3.1.f	*ShiftWR* signal of WIR connected to a core input	Yes
10.3.1.h	*ShiftWR* sampled on both the rise and fall clock transitions	Yes
10.3.1.i	WSI and WSO sampled at the wrong clock transition	Yes
10.3.1.j	*UpdateWR* sampled at both the rise and fall clock transitions	Yes
11.1.1.a	WBY duplicated	Yes
11.1.1.b	Modified WBY register select signal	Yes

* WBY: wrapper bypass; WIR: wrapper instruction register; WSI: wrapper serial input; WSO: wrapper serial output.

have been correctly detected. It is interesting that, in some cases, the violation of a single rule was propagated to other rules, generating multiple design errors.

Our automated environment for IEEE 1500 compliancy verification targets different users, from core designers to core integrators. The environment can therefore guarantee various compliancy levels, depending on the amount of information about the internal core structure that is available to users. Soon, a verification environment such as this will help designers increase productivity, reduce design time, and optimize the test plan of very complex SoCs. Moreover, the same approach used to build the IEEE 1500 verification environment could be exploited to build verification environments for other types of standards such as IEEE 1149.1. ■

References

1. *IEEE Std 1500, Testability Method for Embedded Core-Based Integrated Circuits,* IEEE, 2005.
2. Y. Zorian and A. Yessayan, "IEEE 1500 Utilization in SoC Design and Test," *Proc. Int'l Test Conf.* (ITC 05), IEEE CS Press, 2005, pp. 543-552.
3. *IEEE Std 1450.6, Core Test Language,* IEEE, 2005.
4. D. Appello et al., "A P1500 Compliant BIST-Based Approach to Embedded RAM Diagnosis," *Proc. 10th IEEE Asian Test Symp.* (ATS 01), IEEE CS Press, 2001, pp. 97-102.
5. S. Diamantidis, I. Diamantidis, and T. Oikonomou, "A Unified DFT Verification Methodology," *Proc. IP-Based SoC Design* (IP/SOC 05), EE Times, 2005; http://www.us.design-reuse.com/ipsoc2005.
6. I. Diamantidis, T. Oikonomou, and S. Diamantidis, "Towards an IEEE P1500 Verification Infrastructure: A Comprehensive Approach," *Proc. 3rd IEEE Int'l Workshop on Infrastructure IP,* IEEE Press, 2005, pp. 22-30.
7. T. Oikonomou, I. Diamantidis, and S. Diamantidis, "Coverage Driven Verification of IEEE P1500-Compliant Embedded Core Test Infrastructures," Globetech Solutions, 2005; http://www.globetechsolutions.com/index.php?module=uploads&func=download&fileId=40.
8. A. Benso et al., "IEEE Std 1500 Compliance Verification for Embedded Cores," *IEEE Trans. VLSI Systems,* vol. 16, no. 4, 2008, pp. 397-407.
9. *IEEE Std 1647, Functional Verification Language 'e',* IEEE, 2006.
10. N. Wirth, *Compiler Construction,* Addison-Wesley, 1996.
11. A. Piziali, *Functional Verification Coverage Measurement and Analysis,* Springer, 2004.

Alfredo Benso is a tenured associate professor in computer engineering at Politecnico di Torino, Italy. His research interests include DFT techniques, BIST, and dependability. Benso has a PhD in information technologies from Politecnico di Torino. He is a Golden Core member of the IEEE Computer Society and a senior member of the IEEE.

Alberto Bosio is an associate professor at LIRMM (Laboratory of Informatics, Robotics, and Microelectronics in Montpellier), University of Montpellier, France. His research interests include dependability, diagnosis, test generation, and memory testing. Bosio has a PhD in information technologies from Politecnico di Torino. He is a member of the IEEE.

Stefano Di Carlo is an assistant professor in the Department of Control and Computer Engineering at Politecnico di Torino. His research interests include DFT techniques, SoC testing, BIST, memory testing, and reliability. Di Carlo has a PhD in information technologies from Politecnico di Torino. He is a Golden Core member of the IEEE Computer Society and member of the IEEE.

Paolo Prinetto is a full professor of computer engineering at Politecnico di Torino and a joint professor at the University of Illinois at Chicago. His research interests include testing, test generation, BIST, and dependability. Prinetto has an MS in electronic engineering from Politecnico di Torino. He is a Golden Core Member of the IEEE Computer Society.

■ Direct questions and comments about this article to Stefano Di Carlo, Department of Control and Computer Engineering, Politecnico di Torino, Corso Duca degli Abruzzi 24, I-10129 Torino, Italy; stefano.dicarlo@polito.it.