# A Comprehensive Approach to Assessing and Analyzing 1149.1 Test Logic

Kevin Melocco Hina Arora Paul Setlak

Cadence Design Systems - Test Design Automation - Endicott, NY, 13760, USA
email: {meloccok, arorah, setlakpj}@cadence.com

Gary Kunselman

IBM Microelectronics - DFTS Development and Methodology - Burlington, VT, 05452, USA
email: gkunselm@us.ibm.com

Shazia Mardhani

Sun Microsystems - High End Server Engineering - Burlington, MA, 01803, USA
email: Shazia.Mardhani@sun.com

## Abstract

*In this paper we introduce a tool which is capable of verifying an 1149.1 test logic implementation and its compliance to the IEEE 1149.1 Standard [1][2] while providing a precise list of errors as well as good debug and diagnostic information using graphical analysis. The paper provides a review of the methods used to perform the logic verification. We introduce an efficient technique for verifying the correspondence of chip I/O with the boundary scan register and for verifying large scan registers. The tool is independent of how the test logic is instantiated. The tool requires only the design netlist, cell library definition, and its BSDL [2] identifying what 1149.1 test logic has been implemented. Results on current large ASIC designs is included [10].*

## 1 Introduction

It is estimated that a high percentage of today's ASICs (~80% or more) contain support for IEEE 1149.1 Boundary Scan. It is also estimated that while a large percentage of these designs have 1149.1 test logic inserted by test synthesis tools, there is still much room for error introduced by the test synthesis tool and by the human element. Presently, test synthesis tools do an excellent job of inserting the well defined test logic (e.g. Test Access Port (TAP) controller and instruction register/decode [2]). This limits scope the of 1149.1 errors to the design specific aspects of an ASIC such as I/Os, cores and user defined test logic. The ability to quickly detect, report, and debug test logic related errors is significant to ASIC designers and design centers since they are not typically test tool experts (and unfamiliar with the test logic) nor experts with the Standard. Additionally, the compliance verification and debugging process must be efficient to minimize the impact to the Floorplan, Preliminary and Production net list design phases.

Previous work [3] shows how simulation of black box checks based on Unique Input Output (UIO) sequences can be used to verify some of the mandatory aspects of an 1149.1 implementation. Work done in [6] shows how a hardware based approach can be used to verify an 1149.1 implementation. A simulation based approach is also used [7]. More recently, [5] uses a symbolic simulation approach to identify the test logic and determine its compliance.

In the tool presented here, we use simulation of black box checks based on [3] and a combination of simulation and logic tracing to identify 1149.1 test logic. Together, these two approaches provide the ability to perform efficient compliance verification, debug and analysis of compliance errors, and the ability to produce a BSDL description (See Section 4). The test logic identification

 0-7803-8106-8/03 $17.00 Copyright 2003 IEEE

approach is similar to that which is used by [5]. However, we use a top-down and bottoms-up approach to assessing compliance as opposed to a bottoms-up used in [5]. In particular, we use BSDL to drive the verification process providing an independent verification as opposed to using the test logic implementation to drive the verification process. The verification and analysis tool presented here is known as 1149.1 BSV (Boundary Scan Verification), or BSV. It is one of the tools in a Cadence's (formerly IBM's) DFT and Test Generation tool suite known as TestBench [10], and [8][9][11]. Together, these tools provide a complete solution for producing 1149.1 compliant designs.

BSV's verification and debug capabilities are independent of how the test logic has been inserted. This paper presents the verification and analysis of 1149.1 test logic from an ASIC design perspective, however note that the verification tool is equally suitable for verification and analysis of custom logic designs.

The remainder of this paper is organized as follows. In Section 2 we review why verification of 1149.1 implementations is required given current designs and design methodologies. In Section 3, we explain where verification is used in the design methodology and how the tool can be used early in the design cycle. Section 4 describes verification techniques while Section 5 provides a review of the tool's diagnostic approach and capabilities. The paper closes with current results and conclusions.

## 2 Motivation

The majority of the IEEE 1149.1 test logic in today's designs is produced by DFT tools. IEEE 1149.1 is an established test architecture with widespread use and acceptance. If mature DFT tools produce correct-by-construction test logic within the boundaries of a well defined test architecture, why is there any need for test logic verification? The answer is that today's designs include more than just a standard TAP controller, associated test data registers, and vanilla I/O's. A typical design may incorporate IP cores (hereafter "cores") with embedded I/O's and boundary scan, complex or custom I/O cells, and/or custom test data registers. Each of these entities may require human intervention in the connection process, thereby increasing the opportunity for error. Use of any or all of these constructs necessitates the verification of 1149.1 test logic as illustrated below.

With the growing popularity of high speed interfaces, incorporating the I/O function (and often the boundary scan logic) within an IP core macro is fast becoming a requirement. Many of today's high speed I/O interfaces require control of a number of core macro settings such as slew rate, drive strength, impedance, and mode of operation (e.g. single-ended or low-voltage differential). For example, the core developer may state a requirement that the drive strength setting be equal to or greater than a particular value to guarantee proper 1149.1 operation. If the chip designer fails to control the drive strength during the EXTEST instruction, the driver value captured on the receiving end may be indeterminate (i.e. an indeterminate state appears as a logic 'X', an unknown logic value in a four value simulation).

Consider the core macro circuit (IP_CORE) shown in Figure 1 which contains an embedded multi-mode bidirectional I/O cell (many such I/O cells are embedded within the core; only one I/O is shown for clarity). In this example there are several factors which must be considered for a proper 1149.1 configuration. In addition to the drive strength control, a mode control signal (Mode_Ctrl), determines whether differential or single-ended receiver operation is enabled. If the mode control pin is not properly controlled for 1149.1 testing, 1149.1 compliance errors will result. Note that standalone I/O cells may have an equal degree of complexity and are subject to many of the same types of errors that are seen in cores with embedded I/Os.

Use of custom test data registers (TDRs) is becoming more prevalent; these too may be a source of compliance issues. At a minimum, the designer must specify register length, TDI and TDO ports, and the instruction(s) associated with the TDR. Depending upon the degree of automation offered by the DFT tool, TDI or TDO may need to be manually connected by the designer, or the TDO muxing logic may need to be manually inserted in conjunction with the TDR select signal from the TAP controller. Compliance errors will result if any of these connections is missing or incorrect and hence provides additional motivation for verification of 1149.1 test logic implementations.

Whether the design incorporates cores with embedded I/Os, complex I/O designs, or custom TDRs, verification of the IEEE 1149.1 test logic is a necessity.

## 3 Verification Methodology

Due to the increasing complexity of designs and heightened focus on design turn-around-time, IBM ASIC design centers have adopted the practice of running 1149.1 verification early and often in the design cycle. There are three netlist key phases in the design cycle: Floorplan, Preliminary, and Production.

Normally, the Floorplan netlist phase will have the system I/O's defined and large macros placed, but the system (functional) logic may be less than fifty percent complete. DFT tools such as [9] provide the capability to generate a top-level netlist (I/O cells and 1149.1 test logic) without requiring the system logic; this is referred to as a "coreless" design. Since the coreless netlist is relatively small in size, 1149.1 verification is quickly performed; all that is required is the ASIC design source,

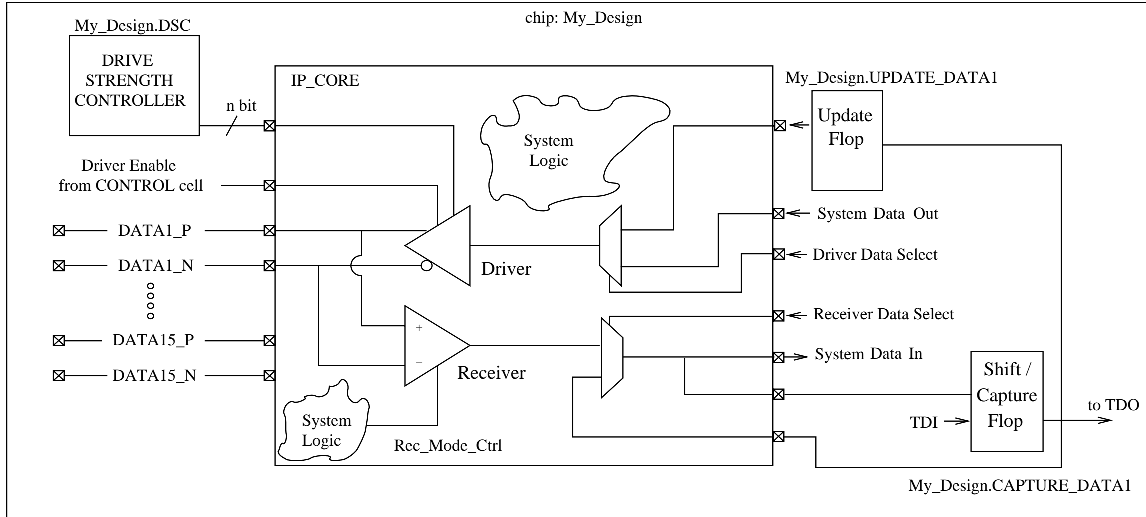


Figure 1. IP core with embedded I/O

cell library, and the BSDL (Boundary Scan Description Language) file describing the test logic functions to be implemented in the design. Errors are corrected in the DFT design files, and the coreless netlist is regenerated and re-verified. IBM ASIC design centers and customers have capitalized on the coreless capability, allowing them to spend more time in the latter design phases, focusing on debugging functional or timing errors in the system logic, while spending less time trying to squeeze in test logic without impacting the system. Designs using IP core macros with embedded I/O and/or boundary scan logic and custom test data registers must be verified when the user core logic is available later in the design cycle, usually no later than the Preliminary netlist phase. Assuming that the majority of the test logic debug was performed in the Floorplan phase, the time spent on 1149.1 verification and debug in the Preliminary and Production phases will be minimal thereby reducing turn-around-time when verification is in the critical path of the design methodology. Using a coreless approach to verification helps to limit the number of design errors seen in the Preliminary and Production net list phases. Typically, errors seen in the latter design phases are due to changes to I/O logic or (small) design changes which "break" a previously valid test structure.

Note that the tool is normally run on a logic model which reflects the functional view of the circuit. This ensures that the "test" aspects of the logic are transparent in functional mode. For example, IBM ASICs require that every non-test I/O cell to be bidirectional regardless of the functional direction of the pin to allow wafer I/O Wrap testing as described in [12]. Here the tool verifies that bidirectional I/Os are in fact, unidirectional I/Os in functional mode.

# 4 Verification

From a compliance verification perspective, BSV has three objectives which are to verify that:

1. The BSDL is compliant with the IEEE 1149.1 Standard
2. The circuit's 1149.1 test logic is functional and complies with the IEEE 1149.1 Standard.
3. The BSDL for the circuit is an accurate description of the test logic implementation.

A verification suite is a collection of checks which are used to assess both the compliance and operability of the test logic. Here we define the verification suite as having three types of checks:

1. BSDL syntax and semantics checks
2. Test logic structure checks
3. Simulation based "black box" checks

The BSDL is used to drive the verification process since it:

1. Reflects what test logic should have been implemented in the design
2. Provides a verification which is independent of how the BSDL was created for the design

Note that the tool can also produce BSDL based on the results of its test logic structure identification. This

capability is particularly important for ASICs or processor design flows which do not use DFT synthesis tools for 1149.1 test logic instantiation and as such typically do not have a BSDL file available[1].

The IEEE 1149.1 Standard contains over 150 rules and this tool does not, nor could not, practically check each of these rules. Users of the tool are provided with a list of what rules are checked. Here we give a brief summary of the scope of checking performed:

- BSDL compliance
- All instructions defined by the Standard[2]
- User defined instructions for which a test data register length is known
- Asynchronous and synchronous reset operation
- Boundary register and I/O functionality (e.g. Boundary register I/O correspondence, Proper timing of I/O state transitions)
- TAP Finite State Machine and the instruction register operations

Experience has shown that the checking which is performed is sufficient to ensure that high quality ASICs are being manufactured.

The remainder of this section shows how BSV checks are used to satisfy its compliance verification objectives.

### 4.1 BSDL Checking

The first of BSV's checks verifies that both the syntax and semantic rules of the Standard have been adhered to. Each deviation from the standard is reported via a numbered error message.

The BSDL description may be incomplete for verification purposes. Allowing partial BSDL descriptions is useful since a complete description may be unavailable early in the design cycle. For example, the physical pin information may be unavailable or the design of certain I/Os may be incomplete. In these cases, the tool reports the error and continues the verification process allowing the user to verify those aspects of the design which are complete. The test logic identification and logic verification phases can begin provided the BSDL contains a minimum of mandatory information such as the TAP port description and compliance enable state.

### 4.2 Test Logic Structure Checking

The tool uses logic tracing and simulation to identify the TAP FSM (Finite State Machine) memory elements (latches/flops) and 1149.1 scan registers including the instruction register, boundary register, bypass and user defined registers. Structural identification is possible since the behavior (rules) for these logic constructs and the scan protocol under which they operate are well defined by the IEEE 1149.1 Standard [2]. The primary reasons for identification of test logic is to:

- Identify erroneous test logic implementations (especially data registers)
- Provide a significant improvement in diagnostic capabilities
- Enable efficient boundary register I/O pin mapping checking (correspondence between the I/O pin and the boundary register latch/flop)

The tool uses a simulation based approach similar to [5] to identify the TAP sixteen state Finite State Machine (FSM) latches/flops. In [5], symbolic simulation is used to identify the TAP's FSM, while this tool uses a four value (0,1,X,Z) simulation when identifying the TAP FSM. Here we define an FSM latch/flop as one which assumes a consistent logic state after application of a homing sequence. In the case of the TAP, the homing sequence is the synchronous reset sequence (i.e. pulse TCK five times with TMS=1). The FSM latches/flops are identified as follows:

First, a list of candidate FSM latches/flops are identified via a topological trace of TCK and TMS. The pool of latches/flops which are fed from TCK and TMS are considered candidates. With the circuit in a compliance enable state[3], each candidate FSM latch/flop is seeded with a random known state[4] followed by a simulation of a synchronous reset of the TAP FSM. This process repeats, each time taking note of the final state of the candidate FSM latches/flops. Latches/flops are evaluated after each seeding and reset operation and those which fail to go to a consistent known state (0,1) are dropped as candidates and are set to 'X' for subsequent iterations. The latch/flop pruning process (seeding and reset sequence simulation) continues while there are candidates which continue to change state. The pruning process is repeated a minimum of sixteen times (since the TAP is a sixteen state FSM) to help ensure that the final candidates are in fact, FSM latches/flops [5]. The final FSM latch/flop logic states are considered the home state

[1] The BSDL file is typically a byproduct of DFT synthesis.

[2] Note that while the tool can verify instruction opcodes of arbitrary length, the tool does not verify that each unspecified instruction opcode selects the bypass register.

[3] The compliance enable state is the state of the circuit after simulation of the compliance enable pattern; A primary input pattern (vector) which enables compliance with the Standard.

[4] The random seeding of an FSM latch/flop is required since the TAP is a sequential FSM which may not have an asynchronous reset and therefore requires a known logic state to prime the FSM to a known state when using a four value simulation.

[5] As described, this algorithm may falsely identify FSM candidate latches/flops as part of the FSM when they home to a par-

(Test Logic Reset) of the TAP FSM. The latches/flops and their home states are recorded and subsequently used in conjunction with the compliance enable state to establish a circuit reset state for simulation purposes. Note that unlike [5], the tool will not terminate when fewer than four FSM latches/flops are identified (a warning is reported). This allows verification patterns to be generated so that they can be used to analyze an erroneous FSM implementation or compliance enable state.

The tool automatically identifies the TAP FSM and latches/flops which make up the boundary register since the I/O pin mapping checks[6] are dependent on the successful identification of this test logic. However, if the tool fails to identify the necessary test logic, it will report the error and provide information about what logic was successfully identified and continue the verification process for all other test logic constructs (e.g. TAP, BYPASS, user defined instructions etc.).

All other test data registers (except the boundary register) are topologically identified when: specifically requested by a user, a simulation based check fails, or are considered excessively large for simulation based verification. Verification of large registers is discussed in Section 4.4. As with the boundary register, the tool will report the error encountered in the register identification process and provide information about what logic was successfully identified. Failure to successfully identify a register does not preclude simulation based checking unless the checking is dependent on the register identification in the first place (i.e. as is the case with the boundary register I/O pin mapping checks). In all cases, the verification process continues so that additional diagnostic information may be generated. In particular, the tool continues generating the verification patterns (vectors) and captures specific debug circuit states which can be used for design debug purposes.

### 4.3 Black Box Checking

The black box checks are an ordered collection of predefined checks each of which contain simulation vectors to be applied to the circuit to determine whether the circuit behaves as expected. Unlike the approach used in [3], the checks are specifically designed to be independent of one another to allow a simplified debug process. One potential drawback to independent checks is that the checks are not organized in an optimal manner thereby increasing CPU time required for verification. However, we trade off optimizing verification time for improved debug time since it is a key requirement of the verification tool. Additionally, as described in Section 4.3.2 and Section 4.4, the tool uses an efficient technique to reduce the CPU time required for verifying boundary register I/O correspondence and large registers which dominate overall CPU time.

Each of the predefined checks can be viewed as implementation (and design) independent (neutral) checks which verify one or more of the 1149.1 rules. The checks are defined as a collection of templates which are customized based on the BSDL description, the logic model, and the test logic information provided when the tool is invoked. The test logic contains information such as the TAP FSM latches/flops and compliance enable state. The checking templates are an intrinsic part of the tool and are defined by the particular rule(s) they verify as well as the 1149.1 test architecture. As shown in Figure 2, an implementation neutral checking template is used to generate the simulation vectors for an implementation specific check prior to simulation. For example, one of the checks verifies the BSDL-defined instruction register capture value. This check is defined with checking templates which serially loads an N bit instruction register with the complement of its expected capture state, then performs an instruction register capture followed by a serial unload of the instruction register. The unload operation includes the expected simulation states.

The tool assumes that the BSDL is an accurate description of the test logic. This assumption allows the tool to meet its objective of verifying that the BSDL accurately reflects the test logic implementation based on the results of the verification suite. This approach also ensures that the test logic design objectives have been satisfied. Note that the tool provides an independent verification of the BSDL description. Previous work [5] uses the test logic identification process to verify compliance and ensure an accurate BSDL description by generating BSDL as a by-product of the test logic identification process. However, this BSDL must be subsequently reviewed to verify the tool was able to identify all the test logic implemented and that the implementation is what was intended.

The tool also combines each of the checks into independent groups of related checks. The grouping allows users to select what aspects of the standard should be verified when the tool is invoked. This simplifies the debugging process since it shows all of the errors related to a specific aspect of the standard together. For example, the IDCODE instruction can be verified and debugged independently of the EXTEST instruction.

ticular state via a synchronous reset. (e.g. the instruction register update latches). A further refinement of the pruning process eliminates the false FSM latches/flops. The algorithm may be incomplete since it relies on random seedings which may not cover each of the $2^N$ states when N exceeds a pratical limit (e.g. N$\geq$ 8) and N is the number of final FSM candidates.

[6] The I/O pin mapping checks verify boundary register I/O pin correspondence are described in Section4.3.2.

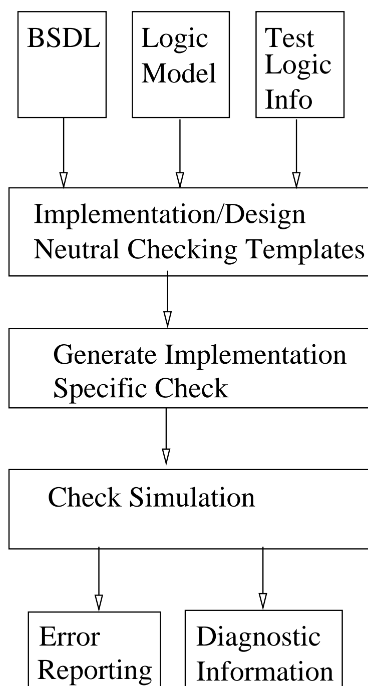


Figure 2. Customization and Execution of Black Box Checks

#### 4.3.1 Black Box Sequences

A sequence is a collection of simulation vectors. Each black box check consists of several sequences:

- Initialization Sequence
- Preconditioning Sequence
- Checking Sequence

The initialization sequence is used to take the circuit from an unknown state to the home state for the check. The home state is the same circuit state for all checks in a verification group. For example, the BYPASS checks use the Test Logic Reset (TLR) state as their home state. In this example, the initialization sequence includes the application of a compliance enable pattern followed by a synchronous or asynchronous reset of the TAP which by definition should bring the circuit to the TLR state.

The preconditioning sequence is used to set the circuit state for the particular check being performed. This might include an instruction register load to select the target register, or to precondition the target logic to a state which is opposite the state expected in the checking sequence. The checking sequence contains the simulation expects and is used to determine whether the check passes or fails.

#### 4.3.2 Preconditioning and Checking Sequences

Most every check involves one or more register loads and unloads in the preconditioning and checking sequences. The tool uses two approaches to loading and unloading registers. One approach is to perform the conventional serial loads and unloads of a register (e.g. clock N bit register N times). The other approach is to use parallel (or broadside) loads and unloads of the register.

The tool uses a five bit shift signature (10011) as proposed by [3] in the checking sequences using serial register access. The shift signature is always immediately preceded with the expected state of the register. So, each checking sequence has the form:

$$S_1,\ldots,S_n,1,0,0,1,1$$

where: $S_1,\ldots,S_n$ is the expected state of an n bit register just prior to a register unload and $S_1$ is the expected state of the bit nearest the scan out pin (TDO).

Every register defined in the BSDL is verified via the five bit shift signature.[7] This includes the instruction, bypass and boundary registers as well as user defined test data registers. The BSDL defines the expected register length and hence determines when the five bit shift signature should appear at the TDO scan out pin. The five bit shift signature has proven to be very valuable in detecting register breaks, clocking problems, length and inversion problems. Verification of user defined TDRs is possible when the BSDL associates an instruction with a user defined register via the BSDL REGISTER_ACCESS statement. The verification is independent of whether the register is part of an IP Core, top shell or simply embedded in the core logic.

The ability to perform parallel loads and unloads is predicated on the successful identification of the latches/flops which make up the shift register. The motivation for using parallel access to a register is the significant CPU time savings in simulation as shown in Table 2 and discussed in Section 6. Parallel register access is used in checks which verify boundary register I/O pin correspondence. We refer to these checks as the I/O pin mapping checks. The key I/O pin mapping checks verify:

- observability of each input pin
- controlability of each output pin
- controlability of output pins with shared enable
- proper timing of output pin state transitions (e.g. state changes on trailing edge of TCK)
- proper boundary register cell to I/O pin correspondence

[7] Except those which are considered a large register. See Section 4.4.

The boundary register functionality as defined by the BSDL boundary register definition is used to determine what I/O pin mapping checks are required. For example, each input capable cell (i.e. input, clock, bidir, observe_only) is checked to ensure that it can observe the state of its input pin. Similarly, each output capable cell is checked to ensure it can drive its active logic states and achieve its inactive logic states.

Marching patterns[8] are used to verify each individual pin:cell pair relationship defined in the BSDL boundary register definition is correct. Collectively, the checks march a one through a field of zeros when checking each of the input pins and similarly when checking each of the output pins. Each "marching pattern" targets one input or one output pin at a time. Assuming that the boundary register is about as long as the number of pins, the marching patterns check requires O($N^2$) shift cycles (where N is the number of input or output pins participating in the check) when the patterns are applied via serial scan operations. Applying the patterns via parallel loads/unloads allows N marching patterns to be simulated with O(N) simulation vectors. Assuming that each pin is bidirectional, the complete I/O correspondence check requires 2N marching pattern checks where N is the number of bidirectional pins. Figure 3 shows an example of the marching patterns produced when verifying input capable cells. Note that each flop/latch is preconditioned to a state which is opposite of its expected capture state. As shown, each input capable cell sees a state transition in each pattern/check.

The I/O pin mapping checks also include checks dedicated to verifying control cells which control more than one output driver (shared control cells). Each shared control cell is checked to ensure that it controls only the drivers it is supposed to control per the BSDL. Each shared control cell has a shared enable check whereby each driver controlled by the target control cell is set to drive an active state (A), while all other drivers are set to drive either their inactive state or the complement state ($\overline{A}$). Similarly, each shared control cell has a shared disable check whereby each driver controlled by the target control cell is set to drive its inactive state, while all other drivers are controlled to drive an active state.

There are two I/O pin mapping check requirements: (1) the test logic structure identification process must have successfully identified the TAP FSM latches/flops and a valid boundary scan shift register, and (2) the BSDL boundary register length must equal the topologically identified boundary register length. Given that these two requirements have been satisfied, the boundary register latches/flops are position-ally mapped to the boundary register cells (i.e. the scan latches/flop which make up the bit position nearest to TDO are mapped to BSDL boundary register bit position zero). Satisfying requirement (1) allows the verification tool to set (stimulate) and measure (observe) the latches/flops directly (like Primary Input or Output pins). Requirement (2) allows the tool to verify the boundary cell functionality which should be implemented.

[8]Here a pattern does not necessarily equate to a simulation pattern/vector.

### 4.4 Large Register Verification

A number of ASICs we have encountered contain a user defined instruction commonly referred to as the ALLSCAN instruction which accesses a large test data register. The TDR includes all scan latches/flops on the chip stitched together to form one large scan register for board and system test/debug purposes. For such large registers, the verification tool does not (by default) attempt to check the register via the five bit shift signature as described in Section 4.3.2. Instead, the tool relies on its structural checks to determine whether the instruction is compliant. From a verification perspective, the tool considers any user defined register over 30K scan bits as a "large" register [9]. Note that while the tool has the ability to use the simulation based approach to perform verification on these registers it will be costly from a CPU time perspective. For instance, a 3M gate ASIC defined an ALLSCAN instruction accessing a 195K bit register required 76 CPU hours to simulate one scan operation while the structural checking completed in less than 5 CPU minutes. Note that the structural checking does not currently verify the fixed state broad-side load capability which may be defined for a register (i.e. a fixed CaputureDR state) as would be verified if simulation were used.

## 5 Verification Results and Analysis

Problem resolution turn-around-time is very important. Many users of the tool are in the position of having to debug an 1149.1 problem with limited information about: the test logic which was instantiated by a test insertion tool, the Standard, and the verification tool.

The tool provides several debug options:

- Verification log file
- A simulation results file
- Simulation vectors for simulation based checks
- Graphical analysis of errors detected via logic structure identification or simulation

The tool uses a message based approach to reporting and debugging BSDL and design errors. Formal error

[9]The size of a large register is controlled via a tool parameter.

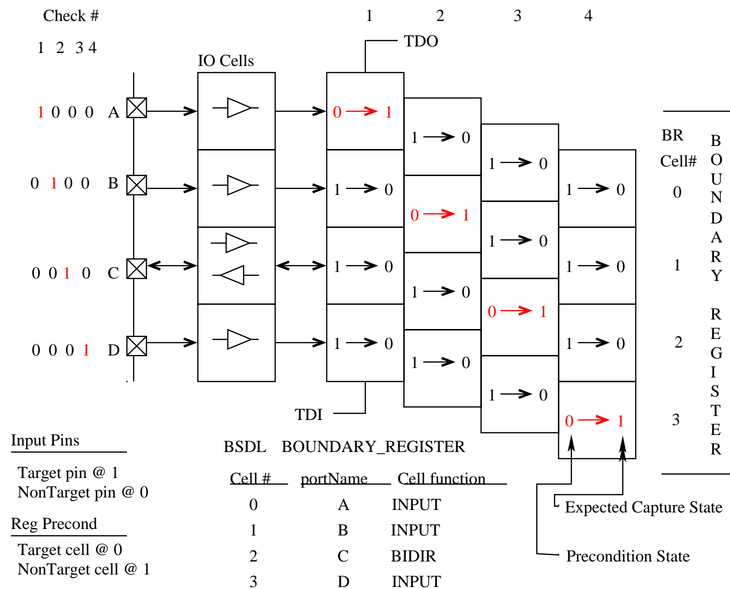


Figure 3. Example Input I/O Pin Mapping Checks

messages are used to report a compliance problem[10].

The tool's logfile summarizes all messages on a per checking group basis such as (BSDL errors, I/O Mapping errors, IDCODE etc). BSDL syntax and semantic errors are simply reported to the tool's log file and are accessible via the GUI. Test logic structure and simulation based checks and their error messages are also recorded in the tool's log file but carry additional information and debug capabilities. The simulation results file contains the details of all the simulation based checks. It is an annotated text form of a waveform display. It contains all the simulation vectors for each check including the stimulus, expected circuit responses and the actual simulation results. Each failing check identifies which circuit nodes failed (miscompared) and when they failed (results reported on a per simulation vector basis). I/O pin mapping check simulation results are reported along with their boundary latches/flops, I/O pins and BSDL information (e.g. cell function) to allow a broad side view of the I/Os and their boundary cells. The TAP FSM state transitions are reported for each of the verification sequences in a check. Detailed information about each check is provided via on-line documentation.

The tool also provides the ability to write its simulation based checks to a binary file to enable message (error) analysis in the tool's GUI environment. The verification patterns may be ported to another simulation environment (e.g. Verilog) which in turn provides the ability to verify that the logic model is an accurate representation of the functional model. Finally, since the tool generates functional tests, these test (patterns) can be used to gain fault coverage when evaluated by a fault simulator.

[10]Each message can be selected in GUI environment to determine which 1149.1 rule(s) have been violated via an HTML reference to the user documentation.

More recently, the tool was enhanced to provide graphical analysis of errors identified via logic structure identification or simulation [11]. In the GUI environment, a user selects a particular error message to be analyzed. In response, the tool populates a View Circuit window with a circuit display showing the logic where the error was detected. Additionally, the circuit display is "set" with the simulation state(s) associated with the error. For static or (vector-less) checks, there is one circuit state which is displayed by the tool. For vector based checks, the circuit display is populated with an initial circuit state resulting from the simulation vectors used in detecting the error. A second window is used to display the simulation vectors defined for the error being analyzed. A user can then view different circuit states by requesting simulation of different portions of the check. This capability allows a user to decide what circuit states are of interest in the debugging process. For example, a user can incrementally step through the simulation vectors of a check and see how the logic responds. Any part of the logic model may be viewed in conjunction with the "simulated circuit state". The simulated logic states appear in the View Circuit display. The problem of determining what logic to display when errors are detected

is covered in [11].

## 6 Results

Table 1 shows the total CPU time required to complete the verification for a variety of fully populated ASICs[11]. In Table 1, Number of nodes (# nodes) represents the number of logic signals and reflects circuit size. The number of input/output (nioc) cells represents the number of boundary register cells with either input or output capability (i.e. input, clock, observe_only, output2, output3, bidir). Note that a bidir cell contributes twice in this count, once as an input and once as an output cell. Std Instructions (SI CPU time) represents the CPU time taken to verify standard instructions and operations. It includes verification of the TAP FSM, synchronous/asynchrouous reset, the instruction register and each of the following instructions and their registers: idcode, bypass, highz, clamp, sample/preload and extest. Note that almost every ASIC included one or more of the optional instructions: idcode, highz and clamp. If any of the standard optional instructions are present on the ASIC, their verification time is included in the "Std Instructions" total time. The CPU time required for verifying user defined instructions is not included in Table 1. IOmapping (IOM CPU time) represents the time taken to verify the I/O boundary register correspondence via marching patterns as well boundary register and I/O functionality such as such as verification of control cells. Finally, the Total Time shows the CPU time required to perform the entire verification process for the checks included in Table 1.

Note that in most ASICs, the CPU time required to perform the I/O pin mapping checks contributes to over 80% of the overall CPU time. The CPU time requirement for the I/O pin mapping checks is largely determined by the number of input/output cells (nioc). However, there are exceptions, as in ASIC_QL (the second largest ASIC sampled) which has the fewest number of I/O cells yet ranks fourth in total CPU time.

Table 2 provides a closer look at the CPU time requirements for performing the I/O pin marching patterns checks on the same ASICs included in Table 1. In Table 2: Boundary register length represents the bit length of the register. The number of input/output (nioc) cells is the same as defined in Table 1. The single serial load/unload (sslu) is the CPU time required to perform a single (complete) serial load or unload of the boundary register. The parallel marching patterns time (tpmp) is the CPU time required to perform the marching patterns verification via boundary register parallel (broadside) loads/unloads. Since the time to load/unload the boundary shift register is essentially zero, tpmp reflects the time required to capture the states of primary input pins into the boundary register (input mapping checks) and the time to update the states of the primary output pins from the boundary register (output mapping checks). Note that the marching patterns check only includes the verification of the I/O correspondence; It does not include CPU time required to perform additional boundary register verification (e.g. shared disable checking). The serial marching patterns time is the CPU time required to perform the marching patterns verification via boundary register serial loads/unloads. Since the tool does not perform the marching patterns check via serial loads/unloads we computed the time for serial marching patterns as follows: the number of I/O cells which must be exercised (nioc) * the CPU time required to load/unload a marching pattern for exercising a single I/O cell (sslu) plus the CPU time required to apply the marching patterns (tpmp). The results show significant CPU time savings (as high as 14X) for marching pattern checking performed via parallel register access, particularly as circuit size and I/O counts increase.

Finally, although we do not present results for custom logic designs, we note that the tool has been used to verify and debug several custom logic designs including serveral microprocessors.

## 7 Conclusion

We described a production tool which employs a comprehensive approach to 1149.1 test logic verification and analysis. While no tool can be complete in verifying 100% compliance to the 1149.1 Standard, this tool has a robust set of checks for detecting and debugging errors introduced when inserting I/Os, cores and user defined registers. We show how the tool uses BSDL, logic structure, and simulation based checking to verify a wide variety of ASICs. The logic structure identification and checking enables an efficient approach to verifying boundary register cell I/O pin correspondence and large test data registers and is also used to provide robust diagnostics of compliance problems. Simulation based checking provides the ability to analyze compliance problems in the presence of logic errors. A well defined verification suite and the circuit's response to the verification patterns allows the tool to display the "erroneous" logic in an "erroneous" circuit state for evaluation by a user. In the tool's analytic mode, the verification patterns can be used to see the same circuit states the checking tool sees when checking is performed. We also note that using BSDL to drive the verification process has the distinct advantage of enabling an independent verification of what test logic should be implemented and its expected 1149.1 behavior. While the tool does not yet provide support for the 2001 level of the IEEE 1149.1 Standard [2], the verification

[11]These are proprietary ASIC designs, a number of which are from the telecommunications market.

| *Part Name* | *Number of Nodes (# nodes)* | *Total Number of Input/Output Cells (nioc)* | *Total Time for Std Instructions (SI CPU time)* | *Total Time for IOMapping (IOM CPU time)* | *Total Time (Tot CPU time)* |
|---|---|---|---|---|---|
| ASIC_AN | 7038794 | 1844 | 19.69 | 132.50 | 152.18 |
| ASIC_CO | 1621876 | 833 | 1.97 | 27.55 | 29.52 |
| ASIC_MO | 1291146 | 1041 | 2.86 | 16.54 | 19.41 |
| ASIC_QL | 1735827 | 255 | 1.54 | 7.39 | 8.93 |
| ASIC_PP | 324492 | 868 | 2.67 | 5.08 | 7.75 |
| ASIC_CR | 13787 | 572 | 0.54 | 2.28 | 2.82 |
| ASIC_MF | 400505 | 317 | 0.96 | 1.71 | 2.67 |
| ASIC_VU | 682536 | 276 | 0.33 | 1.57 | 1.90 |

Table 1. Boundary Scan Verification CPU times (in minutes).

| *Part Names* | *Boundary Register Length (bits)* | *Total Number of Input/Output Cells (nioc)* | *Time For Single Serial Load/Unload (sslu) (secs)* | *Time for Parallel Marching Patterns (tpmp)* | *Time for Serial Marching Patterns (nioc*sslu)+(tpmp)* |
|---|---|---|---|---|---|
| ASIC_AN | 1852 | 1844 | 30.62 | 66.58 | 941.05 + 66.58 |
| ASIC_CO | 735 | 833 | 1.65 | 8.61 | 22.91 + 8.61 |
| ASIC_MO | 1041 | 1041 | 3.25 | 8.77 | 56.39 + 8.77 |
| ASIC_QL | 253 | 255 | 0.35 | 2.36 | 1.49 + 2.36 |
| ASIC_PP | 832 | 868 | 2.84 | 3.12 | 41.09 + 3.12 |
| ASIC_CR | 455 | 572 | 0.76 | 0.85 | 7.25 + 0.85 |
| ASIC_MF | 331 | 317 | 0.65 | 0.95 | 3.43 + 0.95 |
| ASIC_VU | 251 | 276 | 0.30 | 0.98 | 1.38 + 0.98 |

Table 2. Marching Patterns Checking CPU time (in minutes): Serial vs Parallel Boundary Register Loads and Unloads.

and error analysis techniques presented in this paper are applicable to 2001 level of the Standard.

## References


[1] IEEE Standard Test Access Port and Boundary-Scan Architecture, IEEE Std 1149.1-1990 (including IEEE Std 11491.a-1993). published by IEEE.

[2] IEEE Standard Test Access Port and Boundary-Scan Architecture, IEEE Std 1149.1-2001. published by IEEE.

[3] A. Dahbura and C. Yau and M. Uyar, An optimal test sequence for JTAG boundary scan controller, *Proc. Int. Test Conf.*, pages 55–62, 1989.

[4] K. Parker, The Boundary-Scan Handbook, Kluwer Academic Press, 1992

[5] H. Singh and G. Patankar and J. Beausang, A Symbolic Simulation-Based ANSI/IEEE Std 1149.1 Compliance Checker and BSDL Generator, *Proc. Int. Test Conf.*, pages 256–264, 1997.

[6] D. Raymond et. al., Algorithmic Extraction of BSDL from 1149.1-compliant Sample ICs, *Proc. Int. Test Conf.*, pages 561–568, 1995.

[7] Bruce et. al., Method for Testing a Test Architecture within a Circuit, United States Patent 5,517,637, 1996.

[8] V. Chickermane and K. Zarrineh, Addressing Early Design-For-Test Synthesis in a Production Environment, *Proc. Int. Test Conf.*, pages 246–255, 1997.

[9] V. Chickermane et. al., Automated Chip-Level I/O and Test Insertion Using IBM Design-for-Test Synthesis, *IBM Micro News*, Vol6, No. 2, pages 18-22, Second Quarter 2000.

[10] P.S. Gillis and T.S.Guzowski and B.L. Keller and R.H. Kerr, Test methodologies and design automation for IBM ASICs. *IBM J. Res. Dev.*, pages 461-474, 1996

[11] M. Cogswell et. al., A Structured Graphical Tool for Analyzing Boundary Scan Violations, *Proc. Int. Test Conf.*, pages 755-762, 2002

[12] P. Gillis and K. McCauley and Ulrich Baur, Delay Test of Chip I/Os Using LSSD Boundary Scan, *Proc. Int. Test Conf.*, pages 83-90, 1998.