

DesignCon 2002

Compliance verification for SoC and IP interfaces

Vigyan Singhal and Joseph Higgins

Tempus Fugit, Inc.

Abstract

We describe a new methodology to completely verify compliance of RTL designs with standard interfaces prevalent in complex systems-on-chips (SoCs) and IP blocks. As SoCs have grown in complexity, designs in every industry segment are built around standard protocols and interfaces. Storage systems rely on interfaces such as PCI, PCI-X, HyperTransport, Arapahoe, InfiniBand; communication networks rely on protocols such as ATM-Phy Utopia, SPI 4.2, OC-48 framing, etc. Standards for memory controllers, such as SDRAM, DDR SDRAM, SigmaRAM are ubiquitous. The design IP industry is based on standard interfaces such as VCI, ARM Amba AHB or IBM CoreConnect. The new verification methodology is based on formal methods and can find RTL bugs that are almost impossible to find with any simulation or emulation. We show examples from actual in-production systems that have verified and corner-case bugs found that were missed by many months of simulation. We also describe the methodology to create the requirements and constraints for such verification. Specifications are often ambiguous so that designers of one side of the interface have to assume all possible interpretations of the ambiguities from the other side. We will describe such scenarios for both bus-based interfaces (e.g. PCI) and point-to-point interfaces (e.g. Utopia).

Vigyan Singhal

Vigyan Singhal is the President of Tempus Fugit, Inc, a company that provides independent third-part verification solutions for RTL designs. Vigyan has published over 50 research papers in technical conferences and journals. He has a PhD in design verification from the University of California at Berkeley.

Joseph Higgins

Joe Higgins is the Vice President of Engineering at Tempus Fugit, Inc. Previously, Joe was at Cadence Design Systems for 10 years, where he architected many different EDA products. Joe has a PhD in control systems from the University of California at Berkeley.

1. Introduction

Most complex ASICs being designed today work around standard interfaces. We address the problem of verifying that these designs are compliant with the requirements of the standard interfaces. It is important that this compliance be achieved during the design at the Register Transfer Level (RTL). Compliance is critical because, frequently, during the design stage, it is not known which specific systems the ASIC will be interacting with. In fact, the purpose of using standard interfaces is facilitate this choice much after the ASIC has actually been fabricated.

Although the problem of compliance is critical, few solutions exist to solve this problem satisfactorily. One popular solution for standard and mature interfaces is to buy IP designs from third-party vendors. However, often enough, even off-the-shelf IP designs have corner-case compliance bugs (even for very mature interfaces, such as PCI); we show an example in this paper.

Whether the interface designs are crafted in-house or they are bought from third-party IP vendors, the problem of verifying compliance with the standards is critical. It is important to relying on robust bus functional models (BFMs). In addition, the models must have enough flexibility to try various random and directed sequences, especially around corner-cases of the RTL designs as well as the standard specifications.

Very often, it is possible that different designers reading the same English specification may interpret various scenarios differently. It is, therefore, important that designers (or verification engineers) try all possible interpretations of the other side of the interface. We will present examples of cases where ambiguous readings of the specifications can cause system errors when two ASICs are assembled together, each interface designed in a different company, according to a different interpretation of the specification.

In this paper we also discuss the benefits of independent design verification done by third-parties that are not part of the regular design and verification groups. Such verification has shown to uncover corner-case bugs that have been missed by months and years of simulation. The value of independent verification is that frequently it provides a completely new perspective on the requirements of the designs, without being biased by the usual test scenarios that the internal design and verification groups may have been exposed to.

This paper begins by justifying the need to exhaustively check compliance at the RTL stage. In Section 3, we describe the two example interfaces we use in this paper, PCI and ATM-Phy UTOPIA. Next, we describe the existing RTL verification solutions. Section 5 covers examples describing the difficulty in achieving complete compliance. Sections 6 and 7 cover a new methodology for compliance verification based on formal methods, and the practice of such solutions in industrial settings.

2. Why Verify Compliance at the RTL Design Stage?

The focus of this paper is on evaluating compliance at the RTL design stage. There are numerous solutions for testing compliance after the design has been implemented in hardware. These solutions range from bus traffic generators (from Agilent Technologies, for example) to "plugfests" (such as the PCI-SIG PCI Plugfest) where the design can be tested against other implementations.

An obvious drawback to hardware testing is that the design hardware must be available. Deferring compliance testing until post-tapeout is often not a realistic option because today's economic environment with respin costs of the order of \$1M and ensuing delays of 3-6 months. In addition, when bugs are found in hardware, they are considerably more difficult and time consuming to debug.

Another drawback to testing compliance at the hardware level is that bus exercisers and "plugfests" are only available for widely used interfaces. This approach is not feasible for new or proprietary interfaces, and is essentially impossible for interfaces on SoC based designs.

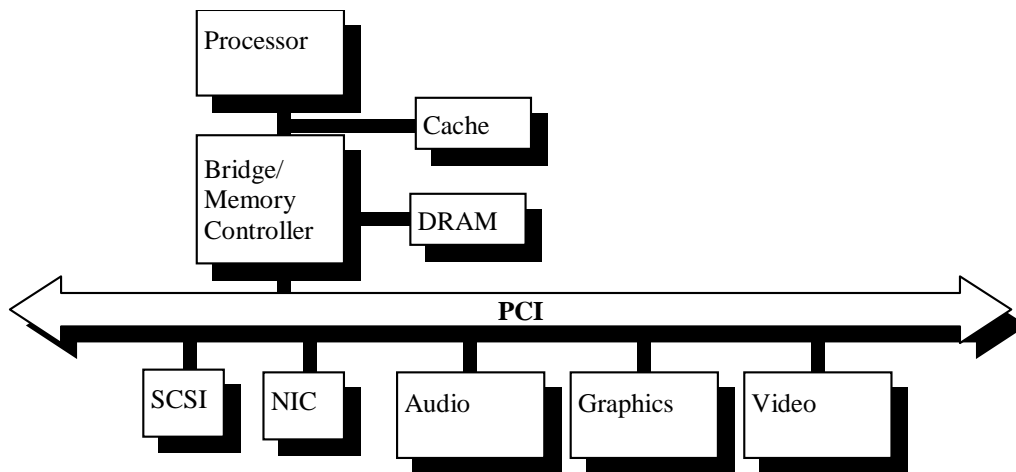
While hardware level compliance checking can cover many more test scenarios than simulation, it is still unlikely to consider many corner cases. Bus exercisers and "plugfests" have no internal knowledge of the design under test, and so cannot possibly consider design dependent bug-prone test scenarios. Such scenarios could include, for example, internal timer expiry and internal reset issues.

3. Interface examples

We will use two different standard interfaces, PCI and ATM-Phy Utopia, to highlight the issues.

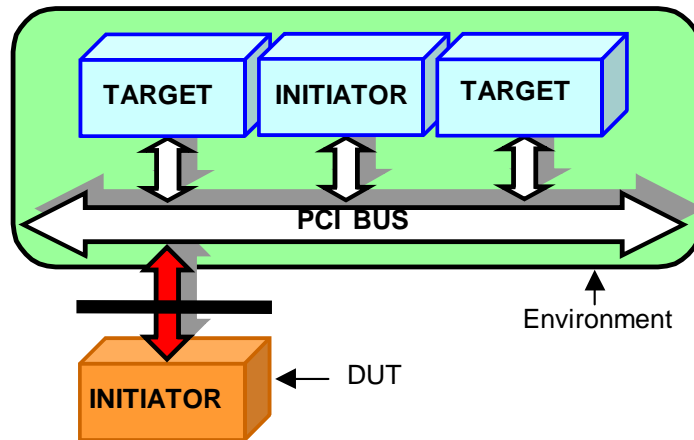
3.1 PCI

The PCI Local Bus[1], comes from the computing industry. PCI (Peripheral Component Interconnect) was developed by Intel and introduced in 1993, as a high-performance, synchronous bus architecture for 32-bit and 64-bit data transfers. Today, other faster standards have replaced PCI in many applications, for example, PCI-X, AGP, HyperTransport. However, PCI is still used widely in existing designs. In fact, in the area of bus interfaces, PCI probably has the largest market in third-party IP design market.



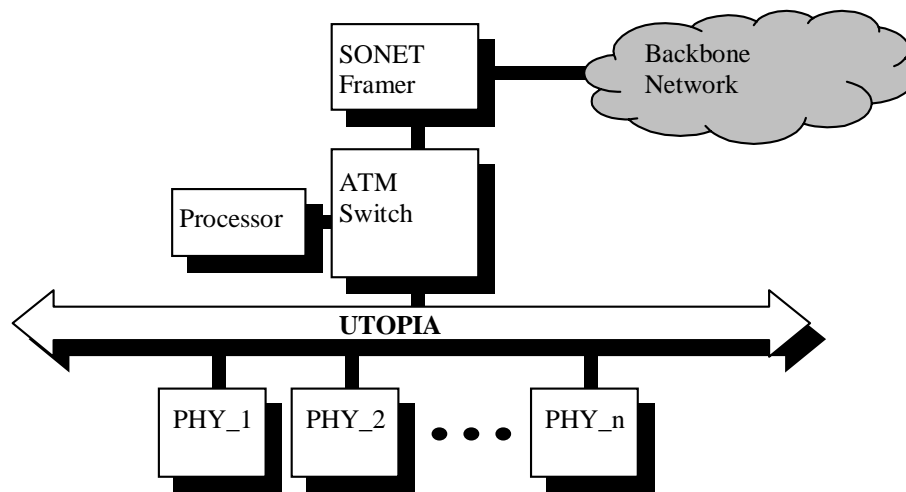
The PCI bus allows multiple initiators (or masters) and multiple targets (or slaves) to be hooked on to the bus. At any time, only one initiator may be transferring data to or from one target. The bus also requires one arbiter that arbitrates between all the initiators and makes sure that at most one of them is allowed to start a transaction on a given cycle. Later, in this paper, we will cover examples of PCI initiator and target implementations, and examples of corner-case bugs in them.

Designs containing initiators or targets for PCI-based systems must be verified for arbitrary environments with arbitrary number of other initiators and targets on the PCI bus (see figure).



3.2 ATM-Phy Utopia

The Universal Test & Operations PHY Interface for ATM (UTOPIA) [2,3] is an interface between ATM layer and Physical layer devices. Utopia Level 2 is an interface for data transfers up to 800Mb/s whereas Utopia Level 3 extends the interface to work up to 3.2Gb/s speeds. The interface signals are used for handshaking, data transfer and flow control.



A designer of a Physical Layer designs that complies with the Utopia interface has to design with all possible implementations of an ATM Layer device, as long as the latter complies with the specification. However, as we shall discuss later in this paper, requiring compliance with all possible implementations is not a trivial task.

4. Current RTL Stage Verification solutions

We discuss the role of various solutions for ensuring interface compliance, from BFM's to interface monitors to third-party independent verification checks.

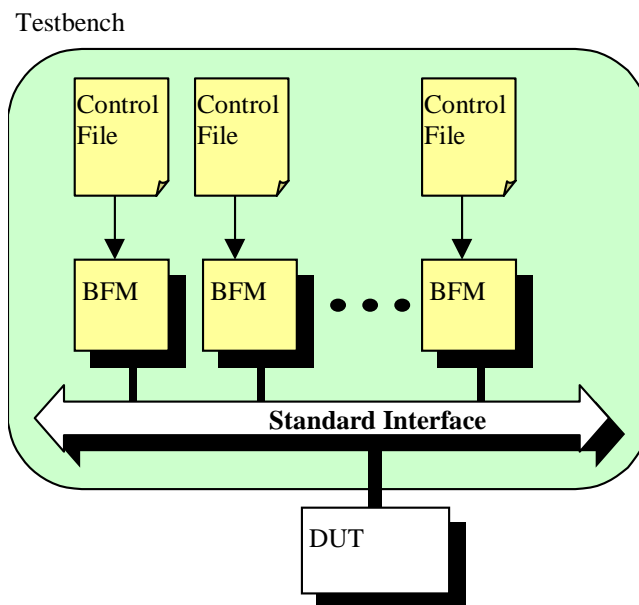
4.1 Tools

Existing approaches to verifying interface compliance at the RTL level are based on simulation. Simulation requires a testbench to drive both the design under test (DUT) and to provide an accurate working model of the interface being tested. The testbench must provide legal stimuli to the interface and must also provide a way of checking that the interface protocol is observed. Typical test environments take days to run and require considerable creativity on the part of the testbench author.

Tools such as VERA [4] or *e* [5] can be used to simplify and accelerate the creation of tests. Semi-formal tools (such as 0-In Search from 0-In [18]) attempt to leverage existing tests by examining scenarios close to existing simulation traces. There are no formal solutions available to verify interface compliance.

4.2 Bus Functional Models

For any design crafted to work around standard interfaces, bus functional models (BFMs) are essential for creating stimuli both for the interface design as well as for the entire system. The following figure illustrates how BFMs are a part of the testbenches used for verification.



BFMs allow the user to drive the simulation stimuli using high-level transactions, without worrying about the details of low-level signaling. This has the advantage the user is not burdened by low-level signaling, to satisfy the interface specification. For example, for PCI specification, a BFM may allow the test writer to initiate a read transaction of a specified length to a specified address, without having to specify the signaling of PCI control signals such as FRAME# and IRDY#.

BFMs can be written in the native design languages, Verilog or VHDL. Test authoring in higher level testbench languages, such as Vera [4] or *e* [5] has been becoming mainstream in recent times. BFMs are sometimes developed in-house; alternatively, BFMs can be purchased from third-party providers, such as InSilicon, Synopsys, or Verisity.

BFMs are effective for writing test situations that test normal behavior. However, the fact the BFMs usually hide low-level signaling details is a two-edged sword. While this makes the task of test writing much more productive, it is often impossible to try all possible corner-case legal combinations (of low-

level signaling), all resulting in the same high-level transaction. We elaborate on this further in the “Third-Party Verification” section.

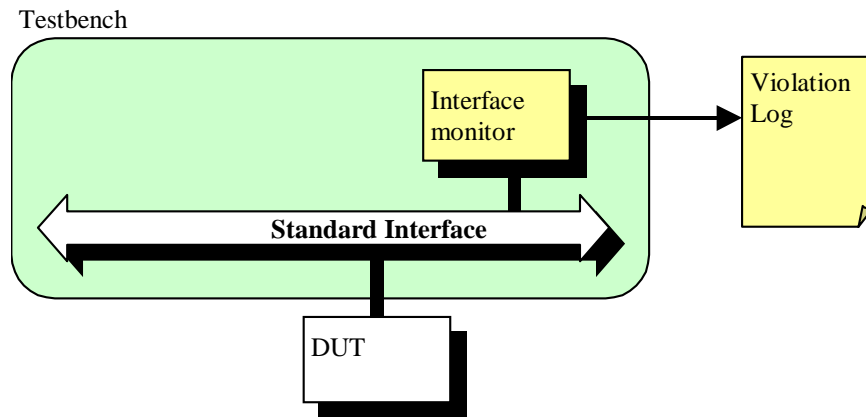
While BFM's increase test creation productivity, effective test creation still relies on the test author's creativity in coming up with scenarios that expose protocol violations. This creativity is limited both by available time and the test author's understanding of the design and protocol.

4.3 Interface Monitors

Interface monitors, often also called bus watchers, are simulation models that simply observe the transactions on the bus or the interface and flag protocol violations.

Interface monitors check whether the transactions occurring on the bus or the interface are respecting all the rules and requirements of the interface specification. These can be low-level signaling rules, for example, for a PCI 2.2 interface, IRDY# is asserted in the clock cycle FRAME# is deasserted. The monitors could also be observing higher-level transaction requirements; for example, a HyperTransport Tx unit never sends a request packet if it has no more remaining credits, per the flow control requirements.

As with BFM's, interface monitors can be developed in house or purchased from third-party vendors such as Synopsys (DesignWare), Verisity (eVCs) and 0-In (CheckerWare).



Interface monitors provide passive compliance testing in that they are only as good as the test scenarios provided by the simulation environment. A monitor cannot flag a compliance issue if the simulation trace doesn't provoke a corner-case problem. Interface monitors are an excellent verification tool if attention is paid to the warnings and errors produced. One customer missed a corner case bug that a monitor had found because the warning was buried among hundreds of other more innocuous warnings.

4.4 Third Party Verification

Independent verification of designs can often uncover bugs that will otherwise be missed. Third-party service providers can come in before tapeout and independently verify compliance for standards-based designs. Our experience has shown that in all cases, corner-case bugs will be uncovered that were missed by months or years of simulation. If designs are based on standard interfaces or requirements, such as PCI, PCI-X, AGP, HyperTransport, Utopia, InfiniBand, SPI4.2 or SPI5, DDR SDRAM interfaces, it is possible for third parties to verify designs independently. Proprietary interfaces can also be effectively verified, but in these cases the service provider must understand the functional details of the interface.

One of the most important factors in deciding whether such service is useful is the impact of such independent verification on current project schedule. If such third-party verification requires in-house assistance, whether in explaining design details or requirements, the benefits may not be so appealing. Further, it should be clear that the third parties have expertise and experience to ensure that new bugs will be uncovered that were missed by the in-house verification methods.

5. Shortcomings of Current Solutions

Two major shortcomings of current solutions are in generating testcases and dealing with ambiguities and misinterpretations of the interface specification. Corner case problems are difficult to isolate and are potentially serious issues because the design must cope with all possible implementations of an interface protocol. This is illustrated by an example of a real bug in which a misinterpretation bug in a device that caused another perfectly compliant device to lock up the bus. The difficulty of generating good testcases is illustrated by a bug found in an IP core. Isolating the bug depended on the coincidence of many design specific events.

5.1 Testcase Generation

Generating good testcases to check interface compliance is a difficult, thankless task that requires great creativity, detailed knowledge of the design, detailed knowledge of the interface protocol and some element of a perverse nature.

The number of testcases that can be generated is limited both by the time it takes to create a test and by the time taken to simulate each test. The testbench authoring tools mentioned previously can ameliorate the first drawback by providing means of parameterizing and generating variations (possibly random) of a given base test. The simulation time is governed by the fact that a substantial portion of the design must be assembled in order to test the interface. Furthermore, since the number of simulations required to verify the system is often exponential in the design size, simulation can only provide limited testing.

Testcases must be based on knowledge of the design internals, otherwise it is very unlikely that corner case scenarios will be created. The testcases must simultaneously explore all legal behaviors of the environment that the design must interact with. In most customer situations, the testcases used reflect expected bus behavior or the behaviors encountered by the test author.

As a consequence, typical test suites fail to either design specific corner cases or to test the robustness of the design in the face of legal but unanticipated behavior.

5.2 Specification Interpretations

English specifications of standard interfaces can often lead to multiple interpretations by different designers. This can cause dangerous system problems. In this section, we give two examples of scenarios that cause such problems.

Complex standard interface specifications, usually written as a textual document with a limited number of diagrams, make it difficult for the designer to be fully cognizant of all the subtleties of allowable protocol behavior. In addition to understanding the interface, the designer should ideally account for all ambiguities and reasonable misinterpretations of the specification. This means that the designer should take the most conservative interpretation for the system under design, but consider the most liberal

(worst case) interpretation with respect to bus behavior. The inevitable ambiguities introduced by the use of a textual document are compounded by tapeout deadlines, making the designer's task a difficult one.

Another unfortunate reality is that if a bug is discovered late in the design cycle, the designer must often make a judgement call to determine the downstream impact of a repair. The above ambiguities compound this difficulty, and it is easy to underestimate the consequences of an apparently innocuous bug.

The first example illustrates an example of an apparently innocuous bug in the design of a PCI initiator that can result in a problem on the PCI bus. Under certain target abort conditions, the initiator failed to deassert FRAME# in cycle 5, prior to deasserting IRDY#. The designer, who was very experienced, considered the bug harmless, because the target designs she had designed and seen previously would not be affected by the late FRAME# deassertion.



However, in one legal target design that we have worked with, the target waited for FRAME# to be deasserted while IRDY# was still asserted before transitioning to an idle state. Consequently, under these conditions, if this target was hooked on to the above initiator, the target will continue to drive STOP# while waiting for the appropriate initiator transition, thereby hanging the bus. This bug was never uncovered in the initiator design simply because the BFM for the target did not encode this particular implementation of the target (which is legal, but disastrous in conjunction with the buggy master).

The second example illustrates how a reasonable interpretation of the UTOPIA Level 2 interface [2] that may cause the UTOPIA interface to lock up. The discussion of back to back cell receipt in the Section 4.2.2 of the Receive Interface Specification [2] describes the intended behavior by means of some text



and timing diagrams. Each PHY device uses the RxClav signal to indicate to the ATM device if it has a cell available. The PHY device responds to the address sent by the ATM on the RxAddr signals in the previous cycle. In the case of back to back cell receipt from the same PHY (address N in this example),

all relevant timing diagrams show RxAddr always being driven with address N on exactly the same cycle as data P48 (cycle 4 in the figure) prior to receiving a back to back cell from PHY N. However, this is not required by the UTOPIA Level 2 interface. The designer of the PHY device interpreted the diagrams as a requirement, rather than an illustration, and consequently depended on address N being driven the cycle before sending the next cell. The designer's interpretation was reasonable if one takes the receipt of non back to back cells into consideration (see [2] for details).

However, there is no requirement that the ATM send address N in this circumstance. It is reasonable for an ATM implementation to avoid polling address N again until it has received the back to back cell, since it knows at least one cell is available from prior polling (for example, in cycle 2, in the figure). If this occurs (address N is not driven in cycle 4 and after that), the PHY device will lock up waiting for the ATM to send address N on the bus at the same time as asserting RxEnb*. At the same time, this implementation of the ATM device will lock up waiting for the PHY device to send the back to back cell (since ATM already knows that PHY has an available back to back cell). This is illustrated in the following diagram.



Since the ATM designer's interpretation is not unreasonable, a comprehensive compliance check of an ATM device will check if the ATM does broadcast address N on cycle P48 for a back to back cell transfer, even though the specification does not strictly require this. Not implementing this may expose the ATM if it is ever attached to a PHY that is designed only based on the figures in the specification, and assumes a few more requirements on an ATM device, than are necessary.

In terms of interface compliance, the impact of the specification ambiguities and unanticipated design dependence on specification is that (1) it is best if an implementation of an interface is compliant with the most liberal interpretation of the specification, and (2) a designer should strive to design so that the design works with most reasonable interpretations of the other side of the interface. Effective interface compliance verification must echo these considerations. Such an objective can only be achieved if the compliance solution is able to explore all reasonable interpretations of the specifications, something that is hard to achieve by usual BFM based verification.

5.3 Compliance Checklists

Often the committees responsible for the interface standards also come up with compliance checklists, for example, the PCI 2.2 Compliance Checklist [10]; similar checklists are available for PCI-X [11] as well as InfiniBand Physical and Link Layer implementations [12].

Even though these checklists may come from the same source as the standard itself, it is important to remember that whenever the checklists conflict with the standard, the standard definition overrules. There are cases when the checklist requirements may be incorrect. An example is PCI target check

numbered TP22 in the PCI 2.2 Compliance Checklist [10]: “IUT always deasserts STOP# the cycle immediately following FRAME# being deasserted [sic]”. One of the regular PCI behaviors, for Master Complete Termination, Fig 3-11 in the PCI Local Bus Specification [1] directly contradicts this “requirement”.

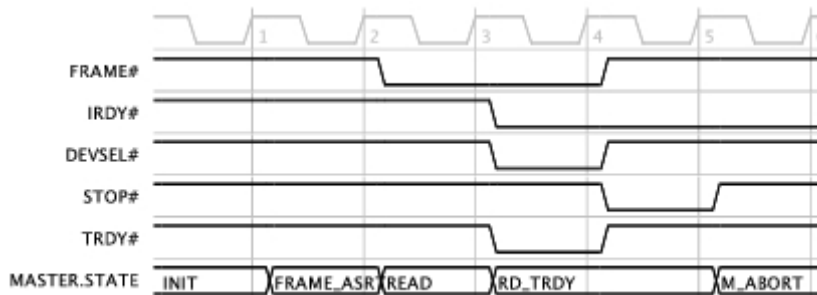
In addition, relying on these checklists to be complete can be dangerous. We have found many important PCI checks missing from the official checklist. For example, a PCI target requirement such as the target should never assert DEVSEL# when the bus is idle. A system we have seen would have seriously misbehaved had another PCI target on the bus not satisfied this requirement.

Later, in Section 6, we will discuss the only way compliance can be effectively guaranteed on real designs.

5.4 IP Core Bug Missed by Months of Simulation

An example of a corner-case bug found in a PCI implementation is shown below. What is most surprising is that this bug was one of many uncovered in a PCI IP block sold in the RTL form by a popular IP vendor. Other customers of this IP vendor had previously used this IP block. The bug was missed by many months of simulation because it required a number of events to occur with just the right (or wrong!) timing.

The bug illustrated in the following figure is a corner-case bug caused by the following conditions happening. The bug is in the implementation of the PCI initiator. The Latency Timer inside the initiator times out just before the target decides to terminate the transaction by a Target Abort. Each of the conditions, the Latency Timer expiration and a Target Abort by a target, is a rare condition. The combination of the two makes the scenario much more unlikely to be uncovered in usual random or directed simulation environments. In hindsight, it is not surprising that this corner-case bug was missed during the verification process employed by the IP vendor.



When this bug is triggered, the PCI initiator state machine gets stuck, and keeps on asserting IRDY# forever. This causes the PCI Local Bus to freeze up, causing disastrous results. Clearly, it is preferable to avoid this situation.

6. Formal Methods

The area of formal and semi-formal methods offers a promise to be able to verify “corner-case” scenarios that are so rare, that they would be almost impossible to uncover using traditional simulation, emulation or even post-silicon use of hardware analyzers. The most successful application of formal

verification has been in the area of checking RTL designs vs gate-level designs, or comparing two different versions of gate-level designs [6]; commercial tools such as Verplex Conformal, Synopsys Formality and Mentor Graphics FormalPro are popular. What is less common is the use of formal verification tools to show the correctness of RTL design itself; however, there have been reports of success in finding “corner-case” bugs in post-silicon designs [7].

The big question is whether formal methods can scale to be able to solve protocol compliance on real production-size designs. In fact, a previous presentation [8] correctly emphasizes that while “it would be nice to verify a PCI implementation [with formal verification tools]”, “it’s hard to capture constraints for a complicated protocol”, and “gate count pushes the limit of formal tools”.

Our experience has been that, using advanced formal tools, it is indeed possible to use formal methods for interface verification, such as compliance testing for PCI, USB2.0, ATM-Phy Utopia or SPI4-2 interfaces. However, it is very important that the environment constraints be modeled accurately and completely. In addition, the dependence on a versatile and powerful formal tool is imperative. Of course, it is equally important to completely and accurately represent the requirements of the compliance.

The only way to completely verify compliance of checklists, as in Section 5.3, is via the use of formal methods. Interestingly, vendors frequently advertise the complete compliance of their parts vis-à-vis the compliance checklists [13]; it would be interesting to know if this advertisement is done based on verification via simulation (which is necessarily incomplete), or with the use of formal tools.

The big advantage of formal tools is that dependence on user-generated stimuli is eliminated; the tool automatically generates all possible legal sequences and looks for violations of desired protocol requirements. The two important requirements for this methodology are (1) to make sure that the input sequences generated by the tool are legal, and (2) to have a list of requirements to check. We will call the former “constraints”, and the latter “requirements”.

6.1 Constraints

Constraints are an integral of a methodology based on formal tools. Constraints make sure the bugs uncovered by the tool represent legal scenarios. On the other hand, in a simulation based methodology, constraints are implicit in the testbench. While some may consider specifying constraints as a burden for a methodology based on formal tools, it is exactly the specification of constraints that provides the completeness that simulation can never achieve. The reason is that the process usually starts with very few or an incomplete set of constraints. Every scenario that has not been explicitly “constrained away” is a legal scenario. Compare this to a simulation based process that works the opposite way: we enrich the set of legal scenarios (or the set of tests in our testplan); this is a never-ending process that will never completely cover all possible corner-case scenarios.

Of course, until the list of constraints is sufficiently complete, the tool can generate scenarios that are illegal. An example of a constraint required for PCI initiator verification is that no PCI target on the bus asserts DEVSEL# while the bus is active (recall the example in Section 5.3).

6.2 Requirements

Protocol requirements ensure that a design obeys the protocol completely, and is able to interact with any legal environment, whether it is attached in a bus-based topology, or if it interacts with another device on a point-to-point interface. Often, compliance checklists, published by the standards organization itself, serve as a great starting point for a complete and exhaustive list of requirements. As

mentioned in Section 5.3, even with these published checklists, an expert evaluation may be necessary to make sure these are accurate as well as complete.

7. Tempus Fugit Compliance Verification

The Tempus Fugit verification service uses an effective methodology based on formal methods to verify interface compliance. By utilizing formal methods, the service isolates bugs that, in practice, will never be found using a simulation-based flow. Formal methods can quickly locate design specific corner case problems, and provide an effective means of understanding the design without requiring input from the design team.

A requirement of a third-party verification service, is that the third-party vendor be able to find RTL bugs without any guidance or training from the customer. It is rarely the case that the design groups have resources to spare that can guide in the application of such tools.

Customers have successfully use the Tempus Fugit verification service at stages of the design process ranging from unit level availability to near tapeout. The service provides an independent evaluation of compliance that is unbiased by the design team's perspectives and internal schedule pressure. A wide range of experience is brought to bear to determine compliance with the relevant protocol under the constraints imposed by the interface. Typically, compliance is determined for the most stringent interpretation of the specification with the least constraining interpretation of the specification. This allows the design team to make informed decisions about protocol violations.

Some examples of “corner-case” interface bugs such a service can catch are the following. It has been our experience that is possible to, otherwise, these bugs until long after tapeout:

1. A DDR SDRAM memory controller [15] issues a read request to a column before the row was activated. This bug occurred only when the application (connected to the memory controller) issues a read request to an address in one bank, followed by a write request to an identical address in a different bank issued on exactly the same cycle that the read request terminated.
2. An OC-192 Link Layer device on a SPI-4 interface [9] does not update its local flow control information soon enough after receiving a transfer, and almost concurrently, the FIFO Status information on the RSTAT link goes out to the Physical Layer device indicating more buffers than what the Link Layer actually has.
3. A PCI initiator misses a parity error on the bus if the error occurred in the penultimate cycle of a long burst transaction, the initiator's write buffers were full in that cycle, and the target inserted a wait cycle in the previous cycle.
4. A USB 2.0 Host Controller [14] sends an IN data transaction that crosses the microframe boundary because the device responded exactly one clock before the timeout, and the data had the maximum bit stuffing.
5. An AGP [16] arbiter in corelogic pipelines a grant for read data, before PIPE# is seen asserted from the AGP graphics master. This happens when the read FIFOs get the return data exactly in the cycle that the grant for the PIPE# request was granted.

6. A HyperTransport [17] Tx module sends a control packet without corresponding data packet, due to a CPU interrupt on last cycle of burst.

8. Conclusion

Interface compliance checking adds a different twist to the verification problem because of ambiguities and misinterpretations of the protocol specification. The design must operate correctly under the most liberal interpretation, yet must present the most conservative behavior if it is to operate correctly in the widest set of environments. Verification of interface compliance by an independent third-party such as Tempus Fugit is an effective way of checking compliance in these situations, by finding design specific problems without time consuming testbench creation and simulation.

9. References

- [1] PCI Special Interest Group. PCI Local Bus Specification Rev 2.2. December, 1998. www.pcisig.com
- [2] ATM Forum Technical Committee. Utopia Level 2, Version 1.0, af-phy-0039.000. June 1995. www.atmforum.com
- [3] ATM Forum Technical Committee. Utopia 3 Physical Layer Interface, af-phy-0136.000. November 1999. www.atmforum.com
- [4] Marco Brunelli, Francesco Sforza, Luca Battu, and Andrea Castelnuovo. Functional Verification of a HW Block Using VERA. SNUG 2001, San Jose, CA, 2001. www.open-vera.com
- [5] Tommy Kuhn, Tobias Oppold, Markus Winterholer, Wolfgang Rosenstiel, Mark Edward, and Yaron Kashai. A Framework for Object Oriented Hardware Specification, Verification, and Synthesis. 38th Design Automation Conference (DAC), Las Vegas, NV, 2001. <http://www.sigda.org/Archives/ProceedingArchives/Dac/>
- [6] Gil Bassak. Focus Report: Formal Verification. ISD Magazine, February 1999. www.isdmag.com
- [7] Carl Pixley, Vigyan Singhal. Model Checking: a Hardware Design Perspective. International Journal on Software Tools and Technology Transfer, volume 2, number 3, 1999, pp 288-306. <http://sttt.cs.uni-dortmund.de/>
- [8] Thomas L. Anderson. Design and Verification IP for PCI and PCI-X. Applied Computing Conference, Santa Clara, CA, May 2000. www.0-in.com
- [9] Optical Internetworking Forum. OIF-SPI4-02.0 - System Packet Interface Level 4 (SPI-4) Phase 2: OC-192 System Interface for Physical and Link Layer Devices. www.oiforum.com
- [10] PCI Special Interest Group. PCI 2.2 Compliance Checklist. www.pcisig.com
- [11] PCI Special Interest Group. PCI-X 1.0a Compliance Checklist. www.pcisig.com

- [12] InfiniBand Trade Association. InfiniBand Architecture Specification Volumes 1 and 2. www.infinibandta.org
- [13] Xilinx, Inc. Xilinx PCI Data Book. www.xilinx.com
- [14] USB Implementors Forum. Universal Serial Bus Revision 2.0 Specification. www.usb.org
- [15] Micron Technology, Inc. General DDR SDRAM Functionality. www.micron.com
- [16] Intel Corporation. Accelerated Graphic Port Interface Specification, Revision 2.0. www.agpforum.org
- [17] HyperTransport Technology Consortium. HyperTransport I/O Link Specification. www.hypertransport.org
- [18] 0-In Design Automation, 0-In Search Data Sheet, www.0-in.com