



WHITE PAPER

Post-Silicon Validation Using Formal Analysis

Verifying the current generation of complex SoCs requires the best methodology and tools, including the application of high-capacity formal verification technologies throughout the design flow, from architectural exploration to post-silicon debug. We see this last area, post-silicon debug, as an important value delivered by formal technology for design and verification teams who have not employed formal earlier in the process to get the design right the first time. As the case studies presented in this white paper demonstrate, the use of formal to find, fix, and *verify* the fix adds tremendous value in the post-silicon lab.

To understand the stakes involved with post-silicon-bugs, consider these widely acknowledged consequences: Finding bugs in model testing is the least expensive option, as the cost of a bug goes up 10X when it's detected in component test, 10X more in system test, and at least 10X *more* if it gets into the field, the worst possible outcome for everyone involved of course.

Post-silicon debug can be very stressful. Effort is spent trying to reproduce bugs seen in the lab using directed random simulation and emulation, but often these traditional approaches run out of steam and cannot root-cause the bug fast enough. As the case studies presented in this white paper demonstrate, formal verification can eliminate debug stress as it uncovers the root causes of bugs and exhaustively validate fixes when other approaches have failed.

Before delving specifically into how formal is becoming a valuable asset in the post-silicon environment, it's useful to also point out just how Jasper's high-performance offerings benefit designers throughout the entire SoC design cycle. For example, our JasperGold formal verification system can have significant impact from early architectural exploration, down to final debug, the subject of this white paper.

Jasper's ActiveDesign with Behavioral Indexing lets users design, concurrently modify, and verify their RTL code, then store it in a persistent database containing both the RTL itself

and an “index” of its elastic behaviors. This information is shared downstream with the JasperGold verification team, facilitating increased collaboration between groups. Benefits are unity among multiple design groups and verification teams, a reduction in information demand on designers, acceleration of verification, and increased IP reuse since design behaviors are now archived and easily accessible.

Production-proven ActiveDesign and JasperGold provide rapid bug detection and debug as well as end-to-end full proofs of expected design behavior. Jasper’s Proof Accelerators – formal modeling methods that significantly reduce the state-space of a design through optimized modeling (abstraction) of common design functions – speed up formal proofs to significantly reduce verification complexity, and combined with Design Tunneling can consistently perform full proofs on properties where other formal tools fail to converge, with an average 10x proof capacity advantage over competitors. Recent enhancements include formal Proof Kits for rapid protocol certification, faster proof and visualization flows, proof engine enhancements, and high-performance design-traversal or “smart” algorithms to speed debugging.

In addition, JasperGold now incorporates QuietTrace™, a new feature shared by Jasper’s design and reuse solution, ActiveDesign™. QuietTrace is a visualization and debugging capability for RTL development that reduces iterations by allowing the user to focus only on the most relevant issues impacting the design. QuietTrace works with Jasper’s Visualize™, which automatically generates and manipulates waveforms without a testbench, answering “what-if” design questions and providing visual confirmation of design functionality which is especially useful for RTL development and debug. Visualize in JasperGold lets designers use formal technology more easily and efficiently, without assertions.

The following chart illustrates the “spectrum of applications” addressed by Jasper formal solutions:

Application	Sub-Applications
Architectural Verification:	<ul style="list-style-type: none"> • Communication Protocol • Cache Coherency Protocol • Liveness Checking
RTL Block Verification:	<ul style="list-style-type: none"> • Verification of Critical Functionality • Protocol Certification • Token Leakage Verification • Data Integrity Checking • Block-Level Simulation Replacement
RTL Block Development:	<ul style="list-style-type: none"> • Incremental RTL Dev. & Verif.

	<ul style="list-style-type: none"> • Register Verification • X-Propagation Detection
Protocol Certification:	<ul style="list-style-type: none"> • Master/Slave Configuration • Standard Protocols (OCP, AMBA, AXI...) • Proprietary Protocols
Design and IP Leverage:	<ul style="list-style-type: none"> • Design/IP Exploration and Comprehension • Targeted Configuration Analysis • Efficient Design/IP Modification Reuse, Leverage and Deployment
Low-Power Verification:	<ul style="list-style-type: none"> • Power Architecture Verification • Power Domain, Model Operation and State/Sequencing Checks • Data Integrity Checks • Frequency Phase Jitter Analysis for Multiple Asynchronous Clock Domains
SoC Integration:	<ul style="list-style-type: none"> • Chip-Level Connectivity Checks and Debug Automation • Automated Pad-ring Verification • Multi-cycle Path Verification
Post-Silicon Debug:	<ul style="list-style-type: none"> • Isolate Root Cause of Silicon Bugs • Validate Fixes

This white paper discusses the basics for employing formal to root out functional bugs in the post-silicon lab, and offers examples of how these techniques have been successfully employed in three different projects.

Bring on the Bugs

When the post-silicon debug team detects a problem in the chip under test, it is initially very abstract and not well understood: the chip hangs, not responding, dropping packets, sending out wrong output, etc. The first step is to know what is happening inside the chip.

Many chips today have some kind of on-chip trace extraction capability, e.g., controls to freeze the chip when certain events are identified; on-chip logic analyzers that allow a selected group of signals to be muxed to external pins; the ability to save the value of certain signals, N cycles before freeze event into some internal memory in the chip; or scan chains to scan out all the flops. Given this, the post-silicon debug team can extract a failure trace, capturing a limited number of signals for a number of cycles before (and maybe after) a problem is detected.

The next step is to isolate and root-cause the post-silicon bug. At this point it is known that the chip is exhibiting illegal behavior as it can be seen in the trace, but the trace represents the last N cycles of the run and it is not known how this state was reached. Typically, there will be a limited number of signals in the trace and it is difficult to choose the right set of signals to show the problem.

Failing Scenario Identified, but Where Is the Bug?

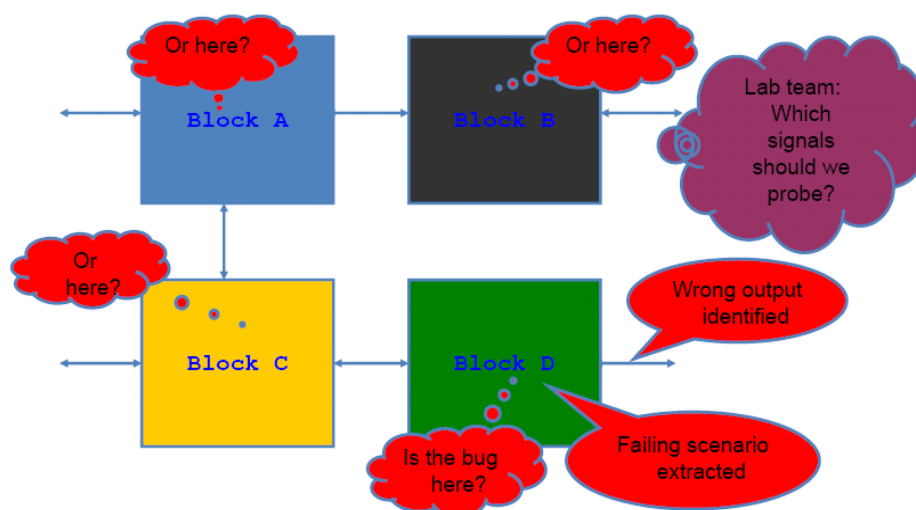


Figure 1: Bug Hunt

Figure 1 represents the dilemma in the lab: The last few cycles of the failing scenario can be observed, but how can root-cause of the problem be found? How can the designer know in which block the bug is located?

Traditionally, the directed-random-simulation team is called to isolate the bug: Can they discover how this state was reached using simulation? It is not clear where the bug is happening, but it is known that it is causing block D to act incorrectly. The bug happens after a 3-4 hour run in the lab when a certain kind of traffic is injected (e.g., only for read transactions on bus X).

In reality, finding the root cause with directed-random-simulation will be in most cases like “Mission Impossible.” If it took four hours of real time with random traffic to hit the bug, how long will it take to reproduce it when simulation time is 1000x slower? 4,000 hours of simulation? And can that be done in a week? Meanwhile, upper management is asking for the root-cause of the bug every other day!

Formal to the Rescue

Naturally, the right formal verification tool, with advanced capabilities, is required. These technologies include:

- High capacity for end-to-end proofs, making formal practical for very large designs of hundreds of thousands of blocks
- Ability to prove the integrity of data transfers across a design, as offered in Jasper Formal Scoreboard™ and Proof Accelerators. Multiple input and output ports make it possible to verify many different types of data transfers, such as transfers relying on byte enables and bus-width conversion, or serial-to-parallel conversion. Proof Accelerators for many common blocks increase performance and proof convergence
- Visualization for easy debug; the ability to easily refine the scenarios to match the lab
- Asynchronous clock handling

One of the key advantages of using proven formal verification technologies in post-silicon debug is its ability to find bugs fast. Usually, finding counter-examples (CEX) with formal is much faster than reaching full proofs on the same property, and this technique lends itself very well to bug hunting. A capable formal verification tool lets the user freeze a specific state in a specific cycle, and then continues the analysis from just that point, so it is unnecessary to analyze the entire design at once.

Finding the bug with formal follows almost the same process as in a pre-silicon formal verification flow. There are, however, some significant differences:

- Search is for one specific bug, one specific scenario
- Not looking for full proof or coverage completeness
- Just need to find the scenario that leads to the illegal behavior
- Can allow over-constraints to simplify the process (e.g., don't allow Write transactions because the bug happens with Read transactions only)

In a typical example, the team would start with what is already known. There is already a trace that shows the illegal scenario, and it is known that the problem happens when a read transaction is followed by another read transaction. All that needs be done then is define a property stating: **- not (illegal_scenario).**

In the process, it is unnecessary to constrain the design as in a normal formal verification run, thereby simplifying the process and allowing for additional shortcuts. For example, it is fine to over-constrain the design and add a constraint: “no_write_transaction_allowed” since it is known the bug happened in “read” and not in “write” transactions.

Then it is simply a matter of asking the formal engines to prove this property and they will compute the failure trace backward from the illegal scenario. Of course, after finding the bug, the designer can go back, fix their RTL and run the formal proof on the same property to confirm the illegal scenario does not happen again. Having the ability to visualize this process is a feature in our tools that automatically generates and manipulates waveforms without a testbench to answer “what-if” questions and provide visual confirmation of functionality.

Case Studies

To illustrate the power of formal in post-silicon debug, let’s examine a trio of examples from Jasper Design Automation customers who overcame big challenges using these techniques.

IP Bug Potential Disaster

The first describes debug of a memory controller violating a bus protocol (Figure 2). This large SoC with a processor and multiple peripherals was behaving badly in the field – hanging up under certain conditions – and was recalled by the manufacturer.

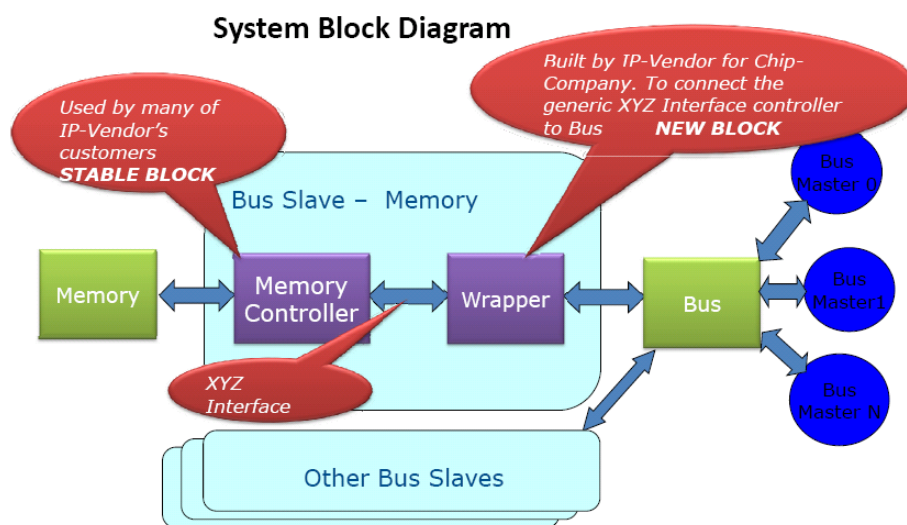


Figure 2: System block diagram of memory controller violating a bus protocol

The post-silicon debug team used directed random-simulation. They started with the little information available from the lab: The chip hung and the problem was isolated as coming from the memory controller when it performs a read transaction.

They worked more than three months until they were able to root-cause the bug. It was identified in the memory controller, sending its data to the wrapper block in a very specific pattern that activated a bug in the wrapper and caused a violation of the bus protocol (Figure 3). This very specific timing alignment of different events in the system was very hard to hit with

random simulation, hence it escaped as a silicon bug and was extremely difficult to detect in post-silicon simulations.

Why the Bug Was Hard to Find with Coverage-Driven Random Simulations

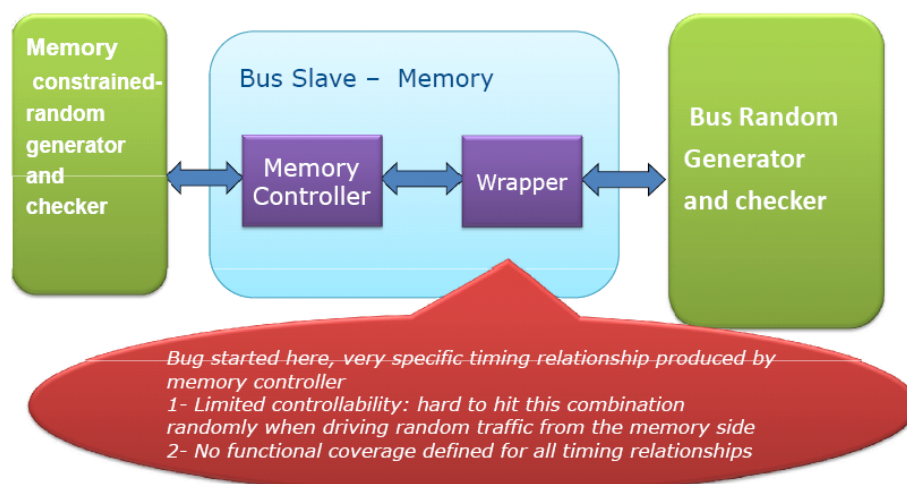


Figure 3: Bug source hard to find with simulation alone

Obviously, three months of simulations to root-cause the bug was a BIG problem for the chip manufacturer, its customers and the IP supplier; especially since products containing the chip had already reached the end customers, resulting in a recall.

Formal was called to help after the fact to see if it could root-cause the bug faster than simulation and the formal engineer was given exactly the same information as the simulation team started with. In this case, the bug was found after just 2.5 weeks, and much of that time was spent ramping up on the design and protocols involved. Actually, once the setup was finished, and properties written, the runtime to find the CEX was *less than a minute* – compared to weeks of simulations trying to hit the bug randomly!

These are realistic and expected results since formal works mathematically backward from the failing trace to generate its root immediately. With simulation, engineers build a checker to detect the bug WHEN it happens, direct random to do more reads, and hope to hit the bug eventually.

Afterward, formal was re-run on the fixed RTL code and uncovered two more new bugs that the post-silicon simulations missed, saving the chip manufacturer another re-spin.

Post-Silicon Partnership

A second case describes how the formal team worked in tandem with engineers in the post-silicon lab. Once the bug was discovered in the lab, formal was called in to help.

The chip included a series of four blocks where the bug occurred. Each block processes the data on its inputs and sends it to the next one. The chip had a built-in logic analyzer that can rout-out a group of signals to be observed on the chip output. However, for the lab team, selecting which signals to probe was always a challenge; it is simply hard to know which signals will show the problem. Since the failure trace was detected on the output of the chips, or the outputs of block D, the lab team put their focus on block D, hence the logic analyzer was probing this block.

The formal team initially wrote an end-to-end property (Figure 4), from inputs of B to outputs of D. In this case, block A was not relevant. The property was written based on the illegal trace captured in the post-silicon lab, and only inputs that caused the bug were allowed while others were constrained out.

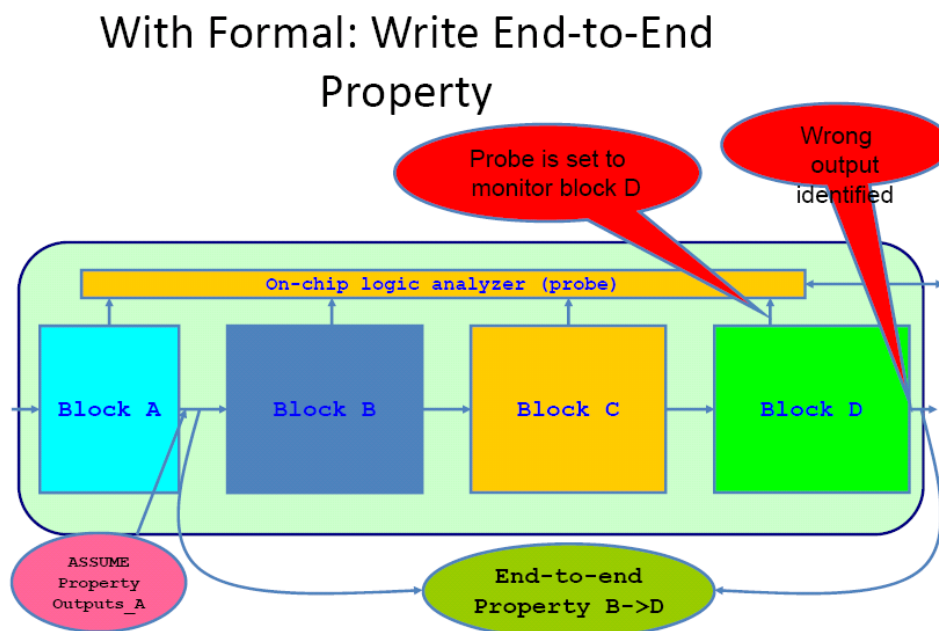


Figure 4: Write end-to-end property

Due to the size of the three blocks, the property did not converge and had to be broken into smaller properties. So a property was written from the inputs of block D to its output. For this property, it was necessary to add input constraints to define legal behavior on the inputs of D

(Figure 5). This property proved very fast. This means that **if** the inputs of D behaved according to the constraints, block D will never produce the illegal behavior.

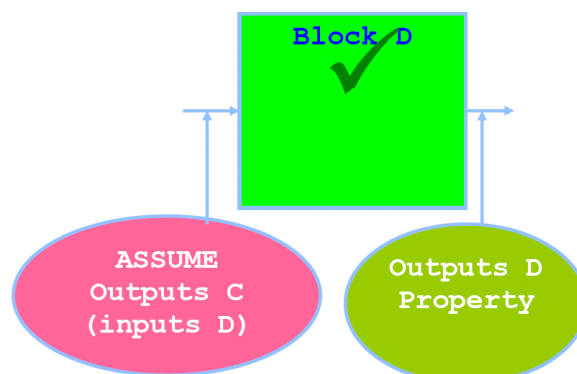


Figure 5: Proving the outputs of block-D with assumptions on its input

Now block D is cleared and it is determined that it does not have the bug. So, the post silicon-debug team moved its focus to block C, so the logic analyzer was set to capture signals from block C.

In parallel, the formal team worked on verifying block C: the assumes on D's inputs got converted to asserts on C's output. And new constraints were written to allow only legal input on Block C.

Within minutes, these properties were proven. Using the exhaustive answer-clearing block C from the formal team, the post-silicon-debug lab changed their focus and now block B became the suspect. The logic analyzer was set to capture signals from block B. With this, very quickly, the post-silicon-debug team was able to find the bug from the trace extracted from block B.

This case presents an excellent example of how the two teams can work hand-in-hand, each using their strengths and sharing information from the other team to isolate the bug fast.

Formal Finds Bugs Missed by Simulation/Emulation

A final real-world scenario demonstrates the value of this approach for a new customer that had not included formal as part of its initial verification strategy. The original verification strategy was simulation and emulation only, with no formal, but even after this exhaustive testing, a bug was found in the post-silicon lab. Inherent limitations with simulation and emulation caused the bug to be missed – it was a corner case and cycle-dependent.

An ECO was done, followed by a couple of weeks in simulation and emulation to confirm the fix was correct. Just in case, they brought in formal to double check the result, and within just a few hours it was determined the fix was broken! Without even having a test bench, the formal tool found another scenario that would cause the bug, and that the fix did not cover all cases.

A second fix, when reviewed with formal, led to the discovery that in a different part of the logic the same bug could occur, which led to a third fix that passed the formal test with no problem. Third time's the charm!

The takeaway from this experience was that formal review saved the company from multiple re-spins, because their simulation environment would not have uncovered the bugs after re-running the fixes. They now design with formal, and have done away with block-level testbenches.

Summary

The right formal verification tool delivers significant ROI in the post-silicon lab, potentially saving enormous amounts of time and money while protecting a company's reputation. Addressing post-silicon debug solely with sophisticated, expensive hardware to inject and capture signals provides an incomplete solution. Formal is delivering impressive results for users who cannot afford delay, or to ship flawed parts even once in today's competitive marketplace.

Ambiguity in the post-silicon debug process can have devastating effects. Verification ambiguity includes incomplete testbench setup, improper property specification, software bugs, and more. Each of these can be a potential bug source, so eliminating them early is key to isolating the root cause of the bug. The benefit of deploying formal verification in post-silicon debug is to remove this ambiguity. Because of high-capacity formal's exhaustive analysis behavior, conclusive answers can be generated as to whether or not a particular verification component is the source of the undesired behavior, enabling quick isolation of the root cause of the problem.

The real solution is to integrate formal as part of the design flow to root out bugs early. But it can also be used in concert with traditional post-silicon methodologies to insure production silicon is bug-free. Pairing formal verification with the capabilities of the post-silicon lab produces highly desirable outcomes, as it eliminates working in the dark. The exhaustiveness of

formal can rule out the existence of the bug in a given block and avoid multiple respins, as evidenced by the three case studies cited.

#

OCTOBER 2010
www.jasper-da.com
info@jasper-da.com