



WHITE PAPER

Using Formal Verification Across a Spectrum of Design Applications

Chip designers worldwide have told us that Jasper is fundamentally different in how we approach their technical and business problems by delivering a high ROI (return on investment) through the application of advanced formal verification techniques. Our tools address a spectrum of key verification challenges – from getting the architecture unambiguously right, to putting more power in the hands of designers, to promoting design reuse, to verifying critical functionality, to reducing process bottleneck, and even silicon debug.

Our ActiveDesign and JasperGold formal verification products deliver compelling benefits throughout the entire SoC design flow. Designers and verification engineers can deploy advanced formal verification technology for their competitive advantage, including architectural verification, RTL design and debug, proofs of functionality, design verification, post-silicon debug and design reuse. Applying these techniques in a targeted manner reduces risks from fatal bugs and silicon respins; helps prudently manage resources by reducing engineering costs; streamlines design and IP reuse; and accelerates schedules and time to market.

Jasper’s ActiveDesign accelerates design development and reuse for important internal design blocks, as well as commercial IP comprehension and deployment. 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.

JasperGold employs state-of-the-art formal verification technology to provide complete systematic verification of design behavior, ensuring correctness in designs where it matters most. JasperGold delivers a competitive advantage across the spectrum of SoC design applications, from

architectural analysis, to RTL design and debug, to verification, and low-power analysis, to silicon debug and software programmers' modeling. Recent improvements include 50% higher performance and capacity, and industry-leading technology to handle today's toughest formal verification tasks such as X-propagation, multi-cycle path analysis, clock domain crossing (CDC), and certification of the latest protocols such as DFI and AMBA 4 (enabled by designer-friendly Jasper Proof Kits).

A recent Jasper customer survey illustrates how formal solutions provide significant ROI across multiple applications, including architectural verification, RTL block verification, RTL development, protocol certification, design and IP leverage, low-power verification, SoC integration, and post-silicon debug:

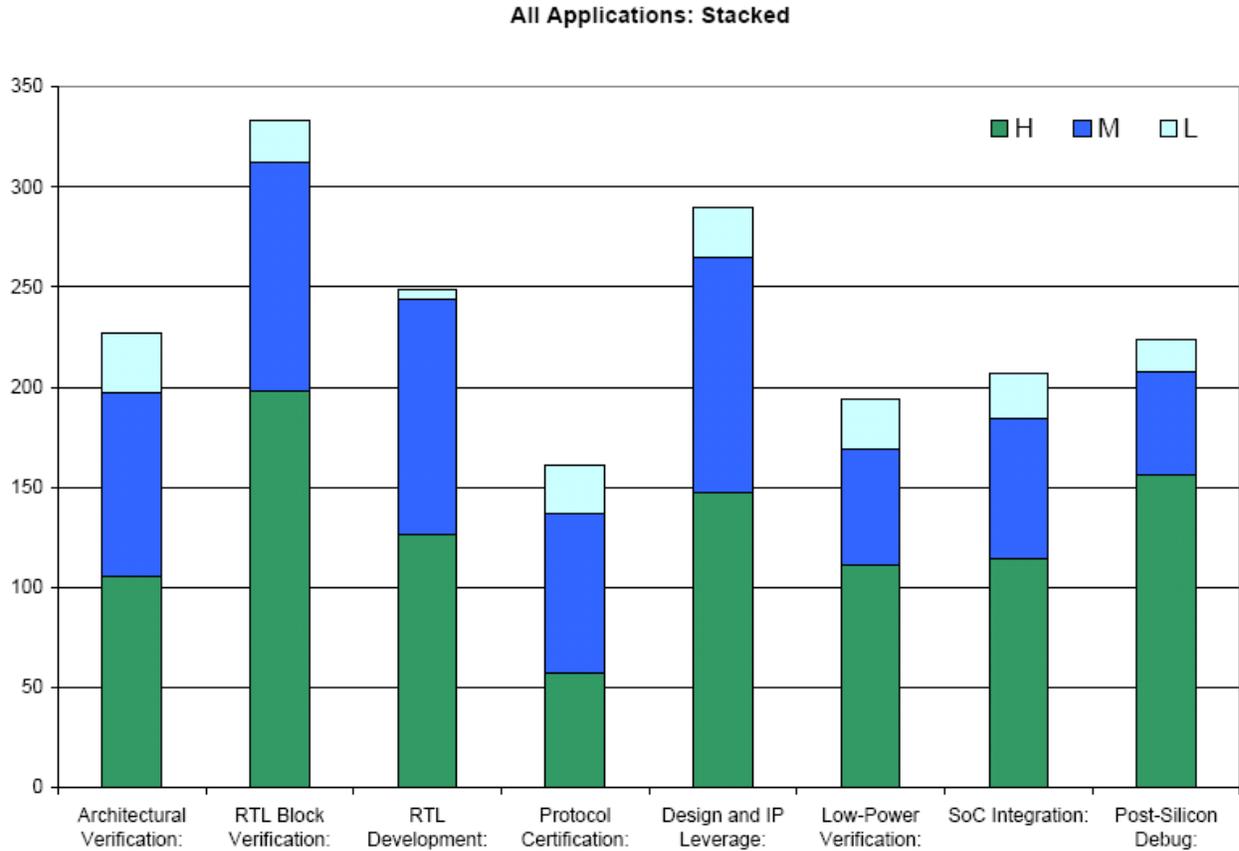


Figure 1: Jasper Customer Survey of Applications and Adoption

Examining each of these applications in detail demonstrates how Jasper formal solutions specifically address each area, beginning with architectural verification.

Architectural Verification

Architectural formal verification involves verifying the design architecture described in the System Verilog language against the set of specifications. Typically, this takes place prior to any RTL coding stage. Uncovering architecture-related problems early prevents costly downstream re-designs. For example, it's practically infeasible to get sufficient coverage at the design level for cache-coherence protocol checking. There are four main categories of architectural verification:

1. **Communication protocol:** A communications protocol is the set of standard rules for data representation, signaling, authentication and error detection required to send information over a communications channel. Formal property language provides a precise medium for capturing these rules, and formal verification can verify the soundness and completeness of the rules, and expose subtle synchronization flaws in the protocol.
2. **Cache coherency protocol:** A cache-coherence protocol contains complex interactions with parallelism and multi-threading. To verify a cache-coherence protocol, a tool must consider a range of traces that are both wide (in terms of starting and branching points) and deep (with long sequences of events). That's hard enough with an architectural model and infeasible with a design-level model. With complex protocols, it becomes extremely difficult to determine which corner cases are possible and how they might be manifested. Formal verification is well suited to discovering these corner cases, and can provide architects with an executable specification that can be queried.
3. **Architectural liveness:** Architectural verification is on critical path because the consequences of an undetected architectural bug are so severe. RTL verification won't catch such a bug; it will simply show that the RTL code matches the architectural spec. An architectural specification bug may well remain undetected until silicon, causing time-to-market delays, redesign, and possibly a silicon respin. Furthermore, liveness verification, i.e. checking that something defined as "good" eventually happens according to the architecture, is also extremely important.
4. **Power architecture:** Low-power designs often utilize CPF or UPF to specify the desired power architecture, providing an easy-to-use, easy-to-modify specification of power intent throughout the design, verification, and implementation flow. Such information, plus temporal behavior controlled by the power management unit, can be converted into appropriate properties for formal verification, allowing automation and exhaustive analysis, and debug of the power architecture.

RTL Development

The following three designer-oriented tasks improve baseline RTL block quality by establishing basic functional sanity; identifying sources of "X" and eliminating undesired propagation; and verifying register operations. Together, they reduce overall engineering effort by allowing RTL designers to debug their own code, reducing downstream verification effort.

1. **Incremental RTL development and verification:** RTL is written incrementally, and ideally design changes could be effectively tracked and verified incrementally, guaranteeing against unintended consequences of any changes. The cost of uncovering a design bug increases non-linearly down the design flow. Formal verification can start with partial RTL, before

testbenches are available, can aid RTL development productivity, and can effectively and incrementally verify design changes.

2. Automatic register verification: Control and status registers (CSR) are the interface between software and hardware. Register verification is extremely important because a poorly implemented CSR always breaks downstream functionality, and an implementation bug can make debugging a functional failure difficult. Formal technology converts the CSR specification into a set of properties capturing the conditions in which the control registers can be configured, the conditions in which the control registers should maintain stable values, and the conditions in which the status registers should receive a specified value. Formal verification can then verify that these conditions are met in all cases, and simplify the debugging process of other failures.
3. X-propagation detection: Certain coding styles may require designers to assign X (unknown) in RTL, and yet, if not treated properly, RTL code and gate-level simulation can potentially mismatch. Unlike traditional simulation tests, which do not guarantee coverage for Xs, formal verification can exhaustively verify X propagation, proving that X's are not propagated to critical areas, and are blocked by appropriate blocking conditions.

RTL Block Verification

RTL block verification requires full proofs of critical functionality as the entire block or selected function is exhaustively verified. Doing so eliminates duplication by cutting down on block-level and chip-level simulation efforts, and completely eliminates any corner-case issues for critical design functionality. There are five sub-applications:

1. Verifying critical functionality: Formal verification enables exhaustive verification of critical functionality, removing uncertainty on corner cases that traditional simulation with directed and constrained random tests may miss.
2. Protocol certification: Formal property languages can capture protocol rules precisely in an executable format, enabling protocol certification to confirm that a piece of RTL obeys the protocol rules in all cases.
3. Verifying token leakage: Token leakage is hard to detect in simulation, since its effect may not be apparent, manifesting only in performance degradation. By specifying token leakage as an explicit specification-level property for formal verification, corner cases that result in token leakage can be discovered and fixed.
4. Verifying packet integrity: Packet integrity is a common specification for data transport design, and although it requires great capacity in a formal verification solution, this reduces the need to verify other implementation-level properties, and achieves a higher level of ROI for the verification effort.
5. Block-level simulation replacement: Simulation is quite time-consuming, creating block-level test benches for simulation is tedious, and maintaining them is even harder as blocks evolve. Formal verification is much faster than simulation, obviates the overhead of creating and maintaining block-level testbenches, and evolves with RTL changes, so it is a natural choice to replace simulation.

Protocol Certification

Protocol certification ensures that individual components on a chip-level bus can function correctly. System-level busses linking multiple components display an enormous number of possible configurations and it's infeasible to adequately verify them using system-level simulation alone. This method is a divide-and-conquer approach to verify components individually yet exhaustively.

The general area of protocol certification encompasses:

1. Both master / slave configuration and verification: Formal property languages can capture protocol rules precisely in an executable format. Using the assume-guarantee approach of arranging the resulting properties, the same description can be applied to both masters and slaves of a specific protocol. Formal verification of both masters and slaves using the same set of protocol properties ensures the masters and slaves interact with one another properly under all legal operating conditions.
2. Verifying standard protocols such as AHP, AXI, etc.: Formal property verification can capture protocol rules precisely in an executable format and confirm that a block of RTL obeys the protocol rules in all cases. For standard protocols, it provides an especially high ROI, since standard protocol properties previously applied to numerous RTL development projects may be commercially available.
3. Verifying proprietary protocols that may be involved in the design: As with standard protocols, for proprietary protocols, formal verification is a great way to ensure correctness, document and maintain consistency among different projects, and avoid the potential confusion and imprecision of a paper document. Project teams can rapidly learn a new protocol and compare changes between revisions.

Low-Power Verification

Low-power verification requires designers to exhaustively verify power up/down sequences, proper state-saving and restoring steps, as well as data integrity during state changes. This prevents power problems both structural (demanding respin) and temporal (violating the power spec and possibly requiring respin).

1. Verifying power domains and modal operation: One of the most effective techniques in power management is power shut-off (PSO), which switches off power to parts of the chip when not in use. Formal verification can be used to specify the situations in which various domains should be switched off, to confirm that no activities are generated when switched off, and that the chip works correctly with every combination of PSO for various domains.
2. Verifying state and sequence interactions for power architectures: Similar to modal operation, allows formal verification to confirm that state retention of key state elements performs properly, and the system can recover after powering up and down the various domains.
3. Full frequency/phase jitter: Clock tree optimization and clock gating, with possible asynchronous clock domain crossing, are typical in low-power architectures, since clock trees are a large source of dynamic power. Because of the difficult timing to trigger frequency jitter and phase jitter, simulation typically cannot be relied upon for detection of

functional errors due to jitter. Formal technology can efficiently verify end-to-end properties in the presence of frequency jitter and phase jitter.

SoC Integration

In the area of SOC integration there are three key items:

1. Chip-level connectivity checking
2. Automated pad-ring verification
3. Multi-cycle path generation

A pre-SoC integration step establishes the connectivity of pins across subsystems and block levels. Connectivity issues are one of the primary factors in the delay in chip-level integration. Establishing connectivity helps the system-level verification but is time consuming with conventional methods. Formal technology can bring a great deal of automation in problem formation, analysis and debugging.

1. Chip-level connectivity checking: During SoC integration, establishing the connectivity of pins across sub-systems and blocks is a necessary task, and involves functional signals and busses, general purpose I/O (GPIO) pins and pads. Verification of connectivity should be done at an early stage of integration to avoid tedious debugging should connectivity problems cause functional verification to fail. Formal verification brings automation and exhaustiveness to the problem of connectivity formulation, analysis, and debugging.
2. Automated pad-ring verification: During SoC integration, the connectivity of pins across sub-systems and blocks includes configuration of the pad-ring, which is often modified as the configurations of the chip increase. By capturing the intended connection in each configuration and utilizing formal technology for pad-ring verification, new configurations can easily be added. Since formal verification exhaustively explores all possibilities, it will identify corner cases when a connection should not exist, while simulation often won't.
3. Multi-cycle path generation: During static timing analysis, RTL designers can specify multi-cycle paths to allow more time for particular paths. Any mistake in such commands can lead to a timing problem in silicon. Since these commands are mostly human-written, they are error-prone, and there is no good way to verify these commands with simulation. Formal technology with waveform generation helps to comprehend activities along specified multi-cycle paths, and also to verify that the specified cycle bounds for the multi-cycle paths are correct.

Post-Silicon Debug

During post-silicon debug, formal verification is viewed as important to rapidly validate fixes, as well as to isolate the root cause for silicon bugs, and prevent expensive serial bugs where the debug team is iterating and iterating in a loop without the proper root cause analysis.

Often, problems in silicon are hard to reproduce in simulation and extra effort is spent in trial and error methods of capturing the right input sequence. Formal technology can quickly eliminate incorrect hypotheses and is capable of locating the root cause of the bug. By

appropriately leveraging formal technology, along with simulation methods of post-silicon debugging, the entire process can be expedited. Once the bug is root caused and appropriate fix is made, formal technology can establish that the fix indeed eliminates the problem and has not introduced any new design error.

1. Isolate the root cause for silicon bugs: In many post-silicon debug situations, the team has some ability to extract a trace of what went wrong in the chip when it failed, but this trace is often severely limited. So the team is in the situation that they can identify some wrong behavior at the output of the chip, but do not understand the triggering event for this incorrect behavior. With formal technology, the process of isolating the root cause for the silicon bug is much more predictable than simulation, because of the capability to rapidly find counter-examples.
2. Prevent expensive serial bugs: Debug teams may iterate endlessly in a loop, without proper root-cause analysis. Validation of bug fixes is critical, since attempting to cure the symptom without understanding the root cause only leads to another, similar bug in the silicon respin. Formal verification, with its exhaustive nature, is well-suited to validate a fix and make sure no other corner case will trigger a similar failure.
3. Rapidly validate fixes: A silicon bug may be fixed in many ways, sometimes without a respin. Yet, it is tedious and difficult to explore the effect of a proposed fix in simulation. Using formal verification, a fix can be validated quickly, due to the exhaustiveness of the technology. For example, it may confirm that a software fix is possible by limiting the number of outstanding transactions to a smaller limit, regardless of the operation conditions. (Simulation can only indicate the lack of a failure for the current trace.)

Design and IP Leverage

Design and IP leverage is constituted by five different categories:

1. Design exploration and comprehension of designs: Traditional simulation is limited for exploration and comprehension of an unfamiliar design, since the user is required to determine how to trigger a scenario by manipulating the inputs. Formal verification applied to scenarios being investigated, along with visualization, advanced waveform generation and debugging features, provides an ideal environment to explore and comprehend a design.
2. Targeted configuration analysis: A reusable design is typically highly configurable, and yet, it is difficult to verify the design for all configurable options. Formal verification applied with visualization technology can generate and annotate complex waveforms with the exercised behaviors, answering questions like "How do I program the design into the target state?" and "What target state will the design get into if I execute this programming sequence?"
3. Modifying existing designs for design reuse efficiency: Leveraging an existing IP block often involves some changes and complete re-verification, reducing reuse efficiency. Focused re-verification only of the features related to the modification made to the RTL is desirable.
4. Promoting knowledge transfer and delivery: A static document is difficult to keep up-to-date as the design evolves, and waveforms captured in the screenshots may not reflect end user needs. Some formal-based solutions capture design knowledge in an executable

specification with the ability to regenerate, annotate, and customize waveforms from the latest RTL.

- Design and IP leverage and deployment: Designers' time is expensive, yet design verification and design reuses often relies on RTL knowledge known only by the original designer. Since the RTL is the authority of what it can and cannot do, tools can extract the answer to any question directly from the RTL, instead of letting the original designer answer the same questions over and over again for multiple design projects reusing the same block.

Conclusions

There are many key technologies necessary to furnish a solution that will enable all of these applications across the development process cycle. These technologies are unified in a flow and methodology supported by Jasper product

Capability	Feature	JasperGold	ActiveDesign
INTERACTIVE ANALYSIS AND DEBUGGING	Visualize: Quickly and easily explore design and debug "what-ifs," automatically generate and manipulate waveforms without a testbench	YES	YES
	QuietTrace: Simplify RTL development and debug with unique visualization and debugging capability that finds similar behaviors with fewer signal events	YES	YES
DESIGN INDEXING AND REUSE	Behavioral Indexing: Design, concurrently modify, and verify RTL code, then store it in a persistent database containing both the RTL itself and an "index" of its elastic behaviors	Use ActiveDesign	YES
	Implication Analysis: Analyze and compare information on multiple revisions of RTL	Use ActiveDesign	YES
INTERACTIVE FORMAL PROOFS	Design Space Tunneling: A natural debugging process to explore and control the amount of logic needed to complete proofs	YES	Use JasperGold
	State Space Tunneling: Identify unreachable states considered by the formal tool and write assertions to rule them out	YES	Use JasperGold
PROOF POWER	Proof Accelerators: Increase the power, capacity and performance of formal verification by significantly reducing the state-space of a design through optimized modeling of common design functions	YES	Use JasperGold
	ProofGrid: Advanced parallel computing capability for dynamic allocation of properties and engine race over a network	YES	Use JasperGold
	Proof Kits: Sets of properties, written in SystemVerilog, for verification of standard interface protocols	YES	Use JasperGold

The time and effort required for verification is a well-known bottleneck in the IC design process. As Jasper users have demonstrated, using advanced formal verification techniques throughout the flow from architectural exploration and RTL design, through silicon debug, not only provides the most complete end-to-end proofs, but dramatically decreases both engineering effort and time.

###

OCTOBER 2010

www.jasper-da.com

info@jasper-da.com