Provably Correct Design

A new paradigm in design and verification

Harry D. Foster
Chief Methodologist
Jasper Design Automation

Version 1.0
Friday, June 04, 2004

Jasper Design Automation, Inc.
100 View St.
Suite 101
Mountain View, CA 94042
Tel: (650) 966-0200
Fax: (650) 625-9840
www.jasper-da.com

Introduction

"We have learned to live in a world of mistakes and defective products as if they were necessary to life. It is time to adopt a new philosophy." –W. Edwards Deming

At the end of World War II, Dr. W. Edwards Deming was invited to Japan by industrial leaders and engineers to help change the world perception that Japan produced substandard, cheap imitations. They wanted to promote a new perception that Japan is the leading producer of innovative high-quality products. As a result of his work, Deming is recognized as the father of the Japanese post-war industrial revival and the leading guru on quality. His business philosophy is summarized in what is known as Deming's *14 Points* for quality management, which has inspired significant changes among a number of leading companies striving to compete in the world's increasingly competitive environment.[1] One of Deming's fundamental principles is that we should cease inspection for the purpose of identifying defects since inspection is costly, unreliable, and ineffective. Instead, we should *design quality in* from the start.

In this paper, we introduce a new methodology in the spirit of Deming's *14 Points*, which we call Provably Correct Design. This methodology enables the engineer to *design quality in* from the start by formally proving fundamental high-level requirements (derived from the micro-architecture specification) during the ASIC and SoC development process. When using the Provably Correct Design methodology, the engineer receives early feedback about the consequences of design decisions and specific evidence of how the design can fail, thus improving the inherent design quality.

We realize that it is not always easy to abandon traditional approaches and try something new. Therefore, after introducing our ideal approach to adopting a Provably Correct Design methodology, we ease into this paradigm shift, by discussing a pragmatic, incremental and targeted approach to introducing it into your flow.

Why can't designers get it right?

There is a myth in the industry today that designers can write any kind of code—black-box it and then throw it at lots of simulation to find all bugs. However, the reality is that big companies with virtually infinite CPUs and engineers have shown that simulation is not enough to find all bugs. Functional bugs still escape to silicon in an alarming number of ASIC and SoC designs.

Functional bugs are not created by nature—designers create bugs. Prior to RTL synthesis, engineers were intimately familiar with every gate placed on their schematics. This meant that they were usually familiar with the consequences of their design decisions. By the mid-1990's, design productivity underwent a huge gain with the adoption of RTL synthesis into the development flow, which enabled engineers to create very complex systems. However, the increase in design productivity meant that engineers could no longer comprehend all the consequences of their complicated design decisions. The result? Bugs.

[1] W. Edward Deming, *Out of the Crisis*, 02 February, 1982 SPC PRESS

Yet, what truly matters is the steps we take after we identify a bug. The problem with using simulation to identify bugs (or as Deming states: “inspecting for design defects”) is that complex bugs are generally identified late in the design process when multiple RTL functional blocks are integrated into the system-level simulation environment. During system-level simulation, the root cause of a bug is not always evident. Hence, at this stage in the project, when there are too many interdependencies between fragments of RTL code, there is a tendency by the engineer to quickly patch the RTL based on the observed symptom of the bug versus re-examining the consequences of the original design decisions in order to fix the root cause of the bug. The result? The engineer has now taken what was a *buggy* design and created a *bad* design (see Figure 1).

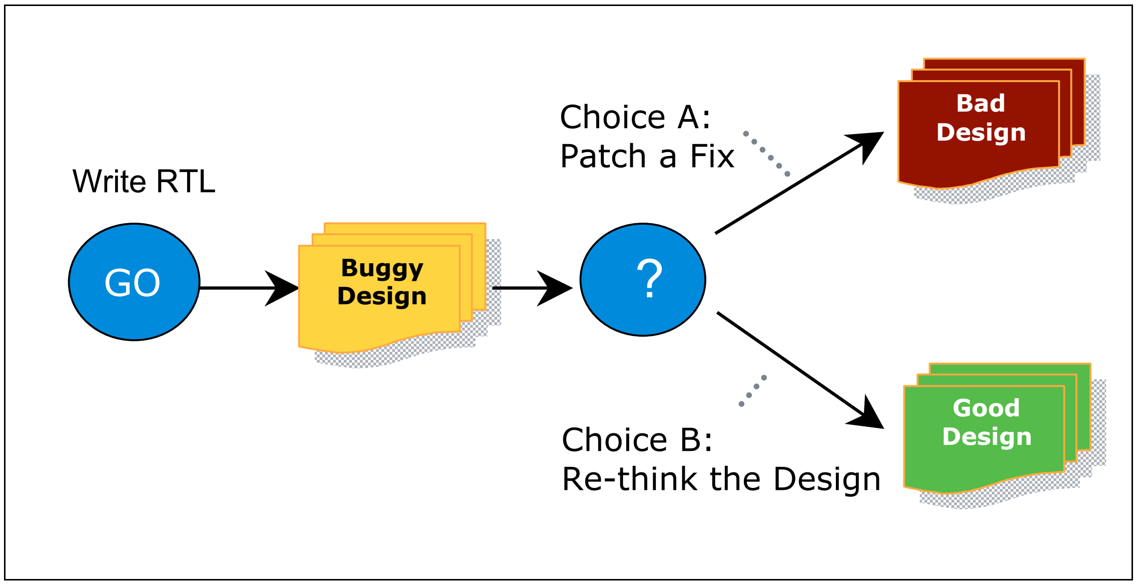

Figure 1. Creating a bad design from a buggy design by patching the symptom

This bad design might eventually be transferred to a new designer, who will spend unnecessary hours trying to unravel the *spaghetti* code resulting from the late-stage patch. Often, even the original designer won’t be able to reverse-engineer the intent as a result of the hasty RTL changes. Ultimately, new bugs will lurk in the bad design, patiently awaiting new modes of operation or error conditions to manifest themselves. Obviously this method of *verify-after-the-fact* is inherently flawed.

What is needed is a solution that exposes bugs created by complicated design decisions while the design is still flexible—before RTL check-in.

Why can’t verifiers get it right?

The previous section identifies problems associated with today’s design methodologies that use a *verify-after-the-fact* approach. The reality is, it is not that designers are bad designers, it is that today’s verification approaches have failed them. In traditional simulation approaches, design concurrency and complexity are typically explored late in a project’s design cycle when the full chip is integrated into a system simulation environment (see Figure 2).

At this late design stage there are typically massive design interdependencies in the fully developed RTL. Hence, any late-stage bug can have significant collateral damage across multiple RTL files.

Furthermore, debugging complex bugs from system-simulation is unpredictable and adversely impacts a project's schedule. Finally, a simulation approach is incomplete—thus, the end result is a *possibly correct design*—you never know when you are done. This description of today's verification challenges supports Deming's claim that inspecting for defects after the fact is "costly, unreliable, and ineffective."

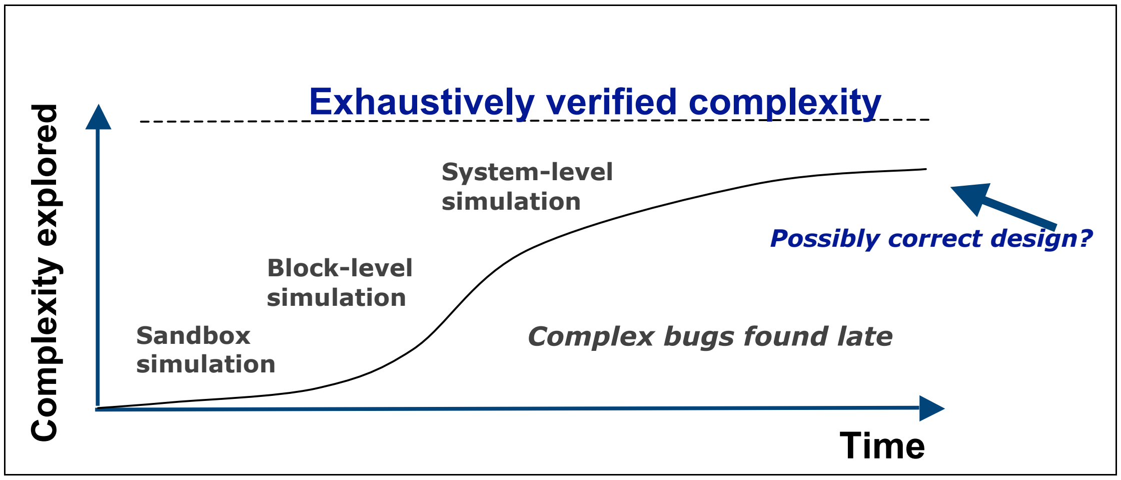

Figure 2. Simulation explores complexity due to concurrency late in the flow

How does the Provably Correct Design methodology help?

In comparison to the traditional *verify-after-the-fact* approach to design, the Provably Correct Design is a new paradigm that promotes a *prove-as-you-design* methodology. That is, the engineer incrementally proves that each aspect of design functionality works as intended *as the RTL code evolves*. Using this approach, engineers find bugs while the design is still fluid and the cost of bug fixes—as well as the risk of collateral damage associated with those bug fixes—is still low. Engineers can instantly explore massive design concurrency and complexity related to various high-level requirements of the design using functional formal verification (see Figure 3). Hence, the Provably Correct Design methodology allows engineers to *design quality in* from the start.

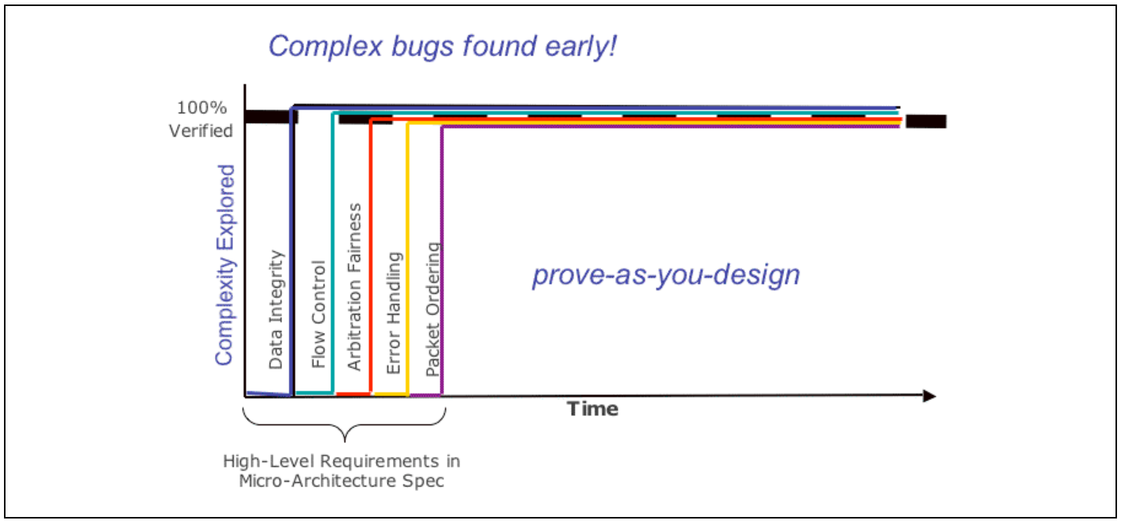

Figure 3. Formal verification explores massive complexity early in the flow

What is formal verification?

Formal verification, which is a key component of the Provably Correct Design methodology, uses mathematical techniques to exhaustively prove that an RTL model satisfies its high-level requirements. Unlike simulation, formal verification does not depend on testbenches, simulation vectors, or any form of input stimulus to verify the design. This independence from traditional simulation-based methodologies comes about because the formal verification tool automatically compiles the RTL model and the high-level requirements into a mathematical representation, and then algorithmically and exhaustively proves that the implementation is valid with respect to its specification—for all possible input sequences. *Note that when a high-level requirement proves true using formal verification, there is no legal set of input vectors for which the design could fail.* However, if the high-level requirement proves false, then the formal verification tool, unlike simulation, isolates the root cause of the bug, which dramatically improves a project's design productivity.

One key aspect of formal verification is that it does not require a completed RTL model to start proving its high-level requirements. That is, even for partially completed designs, all possible input sequences for unconnected wires and input ports are explored using mathematical techniques. When necessary, the user can add constraints on unconnected wires or ports to limit the formal search to legal behavior. Hence, formal verification enables a Provably Correct Design methodology to be applied early in the design cycle and thus permits the engineer to explore massive design concurrency and complexity early in the flow.

What can you prove?

As we previously stated, the Provably Correct Design methodology allows engineers to incrementally guarantee that the design meets its fundamental high-level requirements as the RTL code evolves. So, what are *high-level requirements*? High-level requirements describe the design intent of a block, which is its end-to-end behavior from a black-box perspective. These high-level requirements are generally derived directly from the micro-architectural specification.[2] Examples of high-level requirements include:

- Data is always transmitted correctly through a block, without ever being dropped, duplicated, or corrupted.
- Flow control credits do not leak from the system under any corner-case scenario.
- No sequencing of abnormal, infrequent error conditions can ever corrupt the control in the design.

Unlike white-box implementation assertions, these high-level requirements provide much higher coverage of the design's functionality when proven formally, and they are often reusable across various design implementations and multiple projects. Figure 4 illustrates, from an abstraction level, the difference between high-level requirements and implementation assertions. The Y-axis represents the level of abstraction while the X-axis represents the amount of design behavior covered by a particular assertion or requirement. The dotted line through the middle of this illustration is referred to as the *line of intent* (that is, the point at which our focus shifts from *what* we want to design and verify to *how* we plan to design or verify). You will notice that requirements are significantly above the line of intent, while RTL implementation assertions are below the line the line of intent. In fact,

[2] H. Foster, et al., "Formal Verification of Block-Level Requirements", *DesignCon*, 2004

the high-level requirements are often derived directly from your micro-architecture specification and are independent of the actual RTL implementation. Furthermore, notice that as we move higher up the Y-axis, more RTL design space is covered by the high-level requirement. Hence, with only a few high-level requirements, we are able to cover a significant amount of design space. In sharp contrast, using many lower-level assertions does not produce coverage that is as significant.

By formally verifying a block's set of high-level requirements during the design process using the Provably Correct Design methodology, we are able to achieve significant improvement in design quality, which ultimately translates into productivity gains in a project's flow. These gains cannot be matched by contemporary assertion-based verification flows that combine traditional simulation with semi-formal verification to verify localized RTL implementation-specific assertions.

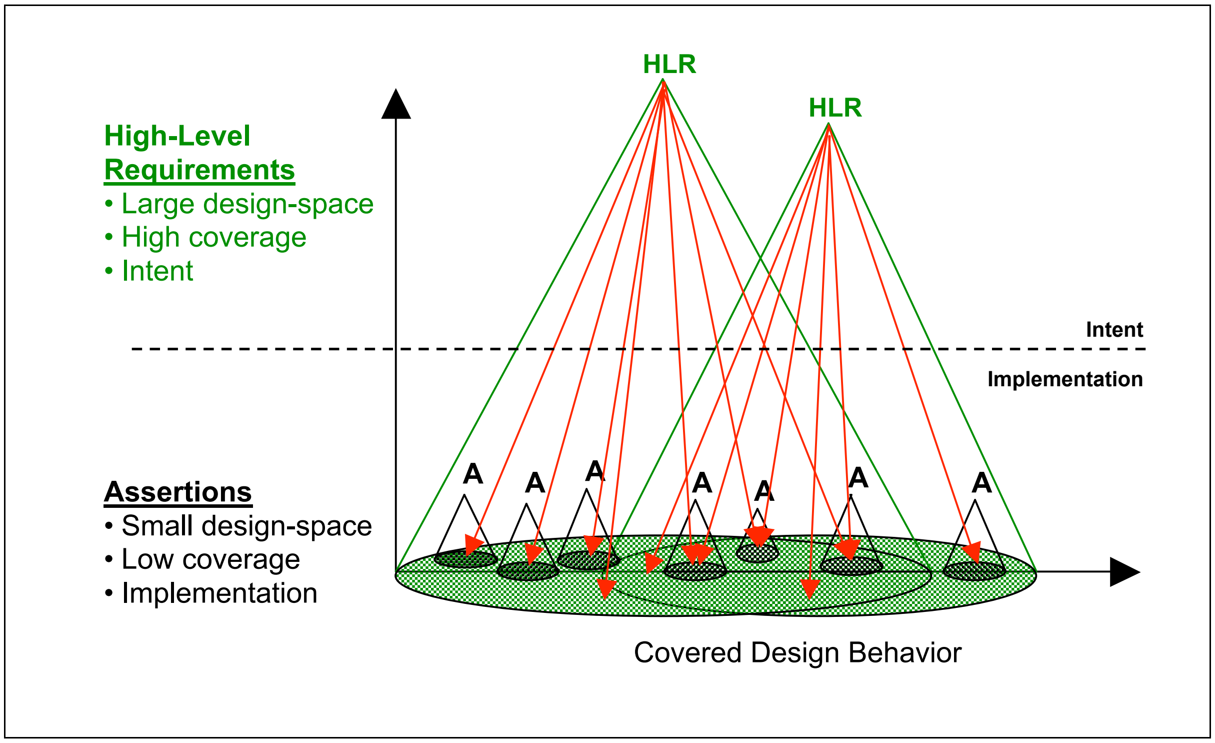

Figure 4. High-level requirements versus implementation assertions

What types of designs are applicable?

We have found that the Provably Correct Design methodology, which uses formal verification to exhaustively prove high-level requirements, is particularly ideal for control logic—as well as data transport blocks containing high concurrency.[3] The following list includes examples of blocks where we have applied our Provably Correct Design methodology:

- Arbiters of many different kinds
- On-chip bus bridge
- Power management unit
- DMA controller
- Host bus interface unit

[3] L.Loh. et al., "Overcoming the Verification Hurdle for PCI Express", *DesignCon,* 2004.

- Scheduler, implementing multiple virtual channels for QoS
- Clock disable unit (for mobile applications)
- Interrupt controller
- Memory controller
- Token generator
- Credit manager block
- Standard interface protocols (for example, PCI Express)
- DMA controllers

A common characteristic of all these blocks is that they are very hard to verify using simulation. For example, simulation typically misses corner-case bugs when it is applied to blocks that support multiple streams of data (as shown in Figure 5), because the set of random input vectors never excites the unique temporal combination or sequence of events required to expose a particular bug. However, formal verification uses mathematical techniques to exhaustively explore all possible input scenarios, which instantly explore massive concurrency and complexity related to the design's specified high-level requirement.

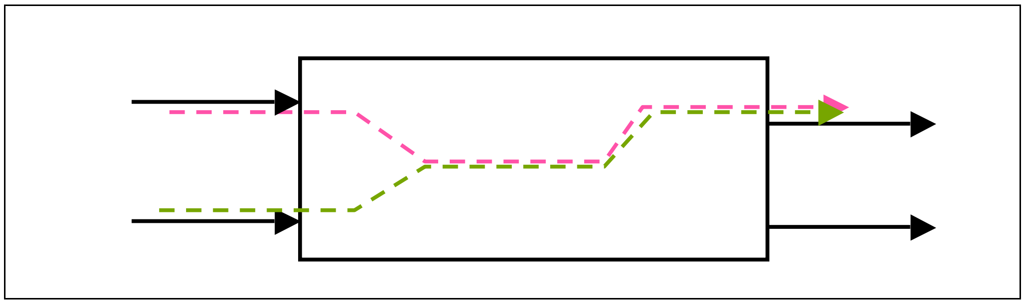

Figure 5. The Provably Correct Design is ideal for verifying concurrency

An example of a bug identified by the Provably Correct Design methodology using formal verification is the following:

> During the first three cycles of a *transaction start* from one side of the interface, a second *transaction start* unexpectedly came in on the other side of the interface and changed the configuration register.

In contrast, design blocks that generally do not lend themselves to the Provably Correct Design methodology tend to be sequential in nature (that is, a single-stream of data) and potentially involve some type of data transformation (see Figure 6).

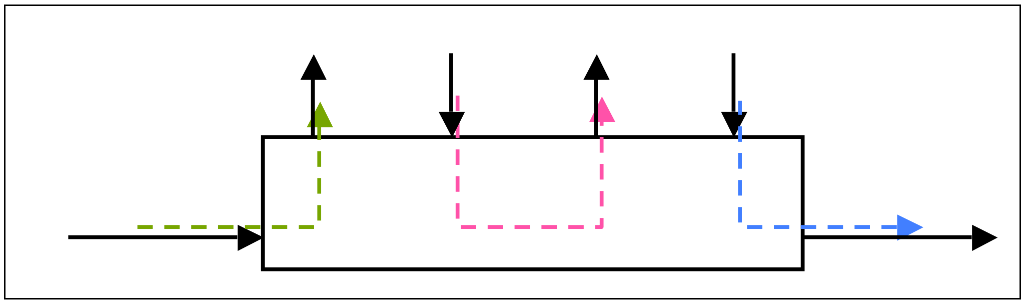

Figure 6. Blocks lacking concurrency are not ideal for formal verification

Examples of designs that perform mathematical functions or involve some type of data transformation include:

- Floating point unit
- Graphics shading unit
- Convolution unit in a DSP chip
- MPEG decoder

An example of a bug associated with this class of design includes the following:

The IFFT result is incorrect if both inputs are negative.

How can designers and verifiers get it right?

Deming once said, "*we should work on the process, not the outcome of the process*". In other words, we can spend a tremendous amount of resources attempting to optimize our simulation flow to identify defects after design (the outcome of the process). Yet, as multiple industry studies continue to indicate, traditional verification approaches are (as Deming says) costly, unreliable, and ineffective. A better solution is to "*work on the process*" that enables the engineer to *design quality in* from the start.

The Provably Correct Design methodology empowers the engineer with a *prove-as-you-design* approach. However, the Provably Correct Design methodology depends heavily on formal verification technology to succeed. Yet due to their capacity limitations, formal verification tools have *traditionally* been applied to problems involving relatively small portions of a design, which does not enable us to achieve the goals of the Provably Correct Design methodology.

In contrast, at Jasper Design Automation, Inc., we have developed a new generation JasperGold™ formal verification tool along with a formal verification infrastructure that delivers the capacity, performance, and supporting environment required to power the Provably Correct Design methodology.

Our infrastructure, shown in Figure 7, provides the following capabilities:

- A knowledge-based system called Jasper Formal Testplanner™ that offers a set of design-specific high-level requirement templates
  - o Simplifies the process of specifying high-level requirements
  - o Provides tips, a formal verification strategy, and example source code for high-level requirements
- A formal verification tool with the capacity to handle realistic designer-sized blocks
  - o Provides unbounded and exhaustive proofs of high-level requirements
- A productive proving environment
  - o Suggests next steps through a proof navigation system that guides the user to success
  - o Supports interactive specification or removal of constraints
  - o Provides progress metrics and complexity analysis
  - o Provides context-sensitive debugging by linking the debugging waveform to a specific point in time related to the source code, which isolates the root cause of each bug to the faulty line of RTL code.

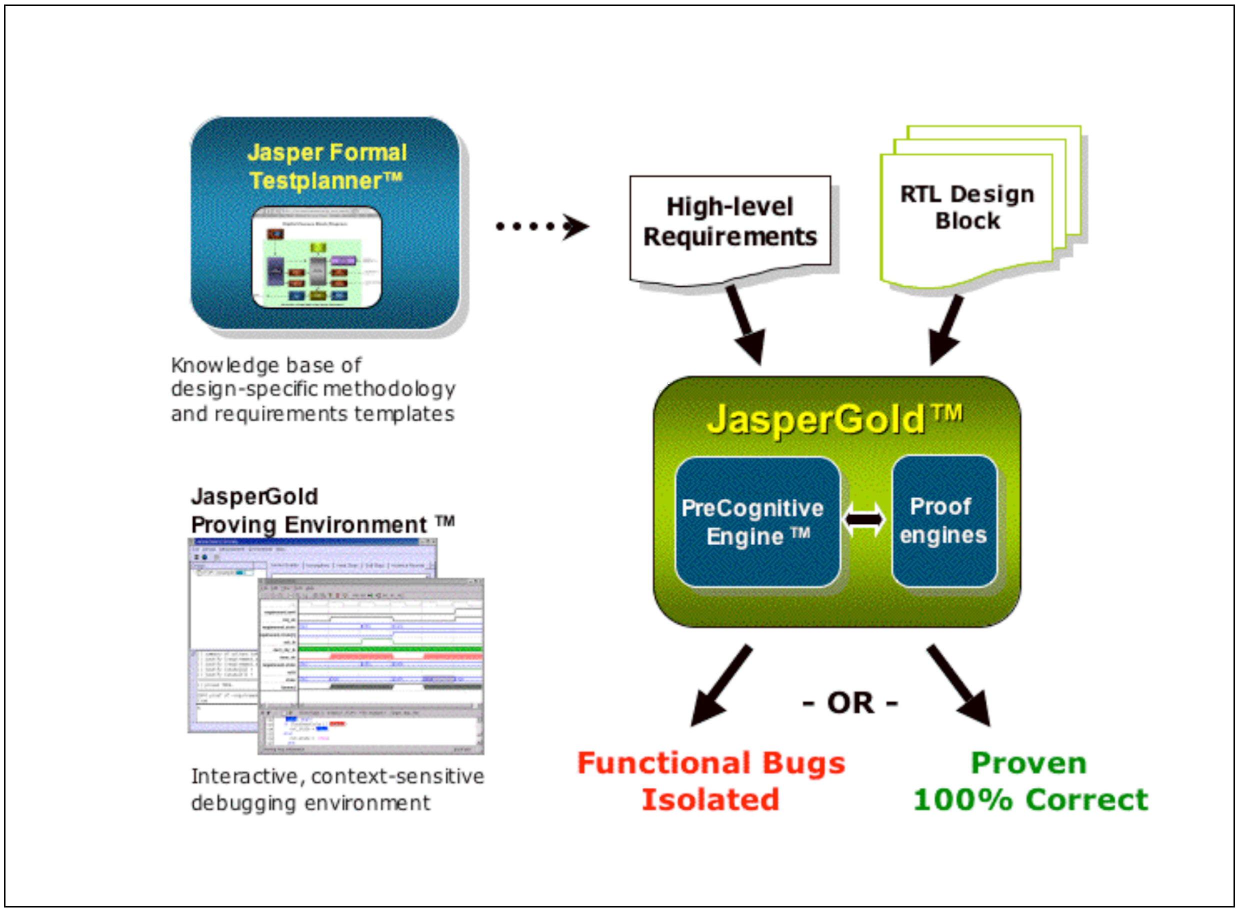
Jasper Formal Testplanner™
Knowledge base of design-specific methodology and requirements templates
JasperGold Proving Environment ™
Interactive, context-sensitive debugging environment
High-level Requirements
RTL Design Block
JasperGold™
PreCognitive Engine ™
Proof engines
- OR -
Functional Bugs Isolated
Proven 100% Correct

Figure 7. Provably Correct Design Infrastructure

One unique aspect of our Provably Correct Design methodology is its regression capabilities. Once JasperGold formally proves a high-level requirement, the engineer can automatically generate a regression static formal verification script that can be added to the design's regression suite. Note that this is not possible for non-static, semi-formal verification approaches that exploit randomness with a bounded proof engine.

How can you adopt the Provably Correct Design methodology?

In this paper, we have presented a new and revolutionary way of doing design and verification. We realize, however, that it's not always easy to abandon traditional approaches and try something new. To ease into this paradigm shift, design teams can adopt a pragmatic, incremental and targeted approach to introducing a Provably Correct Design methodology. For example, it has been our experience that some design project teams decide to introduce formal verification at specific stages within the current flow as opposed to completely replacing their current flow with the Provably Correct Design methodology. One approach has been for a project to supplement traditional verification at a late stage by formally proving critical functionality that is a concern. Then on the next project, the team would initially target a set of blocks they want to verify as they are designed. Then, after formal verification has demonstrated its value both late and early in the flow, future projects embrace the Provably Correct Design methodology as mainstream.

Customer experience

The following anecdote describes one customer's positive experience with JasperGold in a critical post-silicon situation and illustrates the value of the Provably Correct Design methodology. The customer's new system-level test failed consistently and was incurring a very high cost-per-day product delay. Worse yet, the failing chip was actually in its second respin. It had been 20 weeks since RTL freeze and the failing chip had spent one month in post-silicon validation in the lab. The chip failure resulted in hanging the CPU, and the problem was compounded by limited visibility into the failing ASIC in the lab. The observed failing behavior could be described as:

- A single beat request from the CPU generated two beats of data valid, which resulted in the CPU getting corrupted data.

This scenario was difficult to simulate since it depended on a unique concurrent situation to occur from multiple sources. The designer decided to write a high-level requirement that checked the following:

- All transfers shall have the proper number of requested data valids

This general requirement captures a significant amount of end-to-end behavior, and turned out to be easier to write than a specific requirement for a single beat failure. Within three days, the team isolated the root cause of the complex bug that had been evading both traditional simulation and post-silicon verification for weeks and caused multiple respins. Furthermore, the team reused the formal environment that isolated the bug to validate the fix. Hence, having formally proved the high-level requirement on the new RTL code, the designer was assured that there would be no legal set of input vectors for which the design could fail. An insightful remark made by the engineer responsible for the defective block after his experience was that having not formally proved his block *to start with* ultimately cost him 10x the time.

Summary

In this paper, we discussed the inherent flaw of traditional verification methodologies that depend on a *verify-after-the-fact* approach. The problem with these simulation approaches is that they typically explore concurrency and complexity late in the design process—when multiple RTL functional blocks are integrated into the system-level simulation environment. At this late design stage there are generally massive design interdependencies in the fully developed RTL. Hence, any late stage bug can have significant collateral damage across multiple RTL files.

As an alternative to traditional *verify-after-the-fact* methodologies, we introduced a new paradigm that promotes a *prove-as-you-design* methodology. We refer to this new methodology as the Provably Correct Design. This methodology enables the engineer to *design quality in* from the start by formally proving the design's fundamental high-level requirements (derived from the micro-architecture specification) during the ASIC and SoC development process. When using the Provably Correct Design methodology, the engineer experiences early feedback about the consequences of design decisions and specific evidence of how the design can fail, thus improving the inherent design quality.

To support the Provably Correct Design methodology, we introduced JasperGold, which incorporates Jasper Formal Testplanner™, a knowledge base of design-specific methodology tips, formal verification strategies, and example source code for high-level requirements. Formal Testplanner greatly simplifies creating and managing high-level requirements, and jumpstarts the learning curve for the Provably Correct Design. The highly productive Proving Environment in JasperGold accelerates time to complete a proof with state-of-the-art new proof engines and the addition of features such as progress metrics, a design illumination mode, and design-specific next step guidance.