What is formal verification?

By Alok Sanghavi
Technical Marketing Manager
Jasper Design Automation

Functional verification is a critical element in the development of today's complex digital designs. Hardware complexity growth continues to follow Moore's Law, but verification complexity is even more challenging. In fact, it theoretically rises exponentially with hardware complexity doubling exponentially with time. It is widely acknowledged as the major bottleneck in design methodology: Up to 70 percent of the design development time and resources are spent on functional verification. Even with such a significant amount of effort and resources being applied to verification, functional bugs are still the number one cause of silicon re-spins as pointed out by Collett International Research.

Corner-case bugs are artifacts of simulation since they are not detected due to the non-exhaustive nature of simulation-based verification. The reality is that no matter how long you simulate and no matter how intelligent your testbench and generator are, validating the design intent through simulation is inherently incomplete for all but the smallest circuits. The fundamental artifacts of simulation can be classified into three categories: exhaustivity, controllability and observability (**Table 1**).

	SIMULATION	FORMAL VERIFICATION
EXHAUSTIVITY (measurement of the ability to thoroughly observe all possible input scenarios)	▪ Not possible to simulate ALL possible states in a design even with 100s of CPUs and months of simulation ▪ Focus on scenarios and assertions to break the design	▪ Explores all possible states ▪ Results in high quality RTL ▪ Shifts focus on intent – correct functional behaviour
CONTROLLABILITY (measurement of the ability to activate, stimulate, or sensitize a specific point within the design)	▪ Must conceive of vectors, scenarios to "adequately" simulate design ▪ Likely to miss corner case scenarios	▪ No stimulus required ▪ Start early in the design cycle ▪ All corner case bugs caught
OBSERVABILITY (measurement of the ability to observe the effects of a specific, internal, stimulated point within the design)	▪ Must propagate bugs to output pins or insert local assertions to expose bugs and aid debug	▪ Automatically isolate root cause of bugs ▪ Visualize incorrect behaviors and fix them

Table 1: Simulation and formal verification are compared according to the three categories.

Formal vs. simulation

Formal verification is a systematic process that uses mathematical reasoning to verify that design intent (spec) is preserved in implementation (RTL). With formal verification, one can overcome all three simulation challenges (Table 1) since formal verification algorithmically and exhaustively explores all possible input values over time. In other words, it is unnecessary to figure out how to stimulate the design or create multiple scenarios to achieve high observability.

While in theory, a simulation testbench has high controllability of its input ports for the design under verification (DUV), testbenches generally have poor controllability over internal points. To identify a design error using a simulation-based approach, the following conditions must hold:

- Proper input stimulus must be generated to activate (that is, sensitize) a bug at some point in the design.
- Proper input stimulus must be generated to propagate all effects resulting from the bug to an output port.

With simulation-based verification, one needs to plan what to verify in the design:

- Define various input scenarios to test.
- Create a functional coverage model (determine if you have simulated enough).
- Build the testbench (checkers, stubs, generators etc.).
- Create specific direct tests.
- Simulate for months and months.

In reality, the engineer usually iterates on running tests, debugging failures, re-simulating the regression suite, observing various coverage metrics and adjusting the stimulus (for example, steering the input generator) to hit previously uncovered aspects of the design.

Let's consider the elasticity buffer example (**Figure 1**).

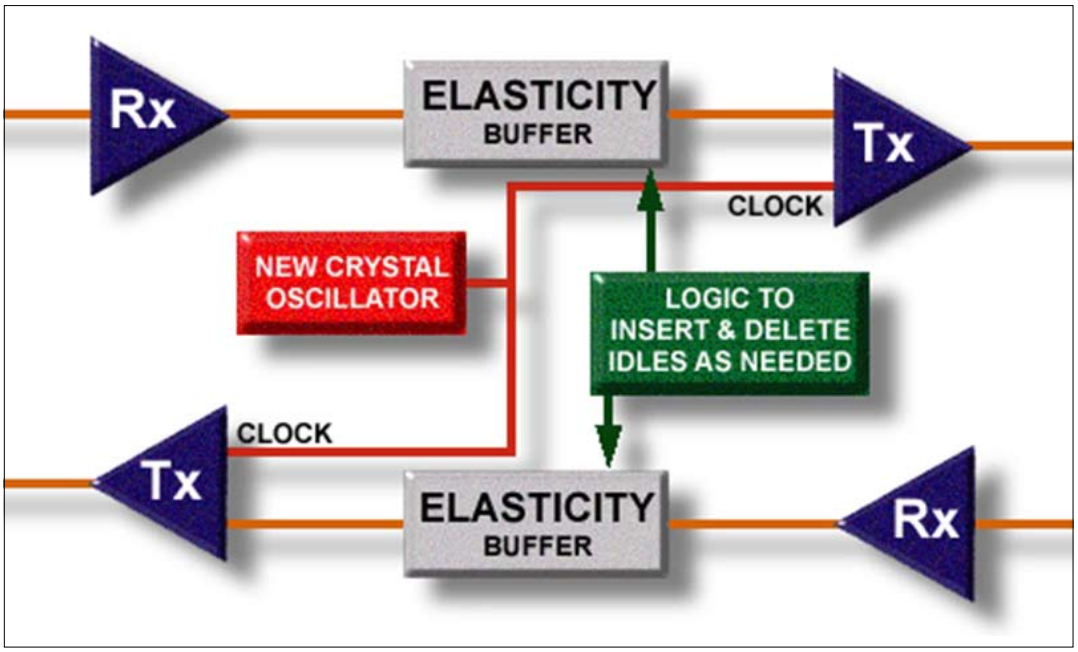

Figure 1: Shown is the elasticity buffer example.

Data can change between clock domains, with ability to adjust to the phase and frequency drift between the two clocks. Data must be transferred through the elasticity buffer without corruption even when the design allows clocks that are not fully synchronized. An example of a functional bug in this case could be buffer overflow due to data changing when clocks' active edges are aligned. It may require enormous amounts of simulation and consideration of all possible input scenarios to model and simulate this faulty behavior.

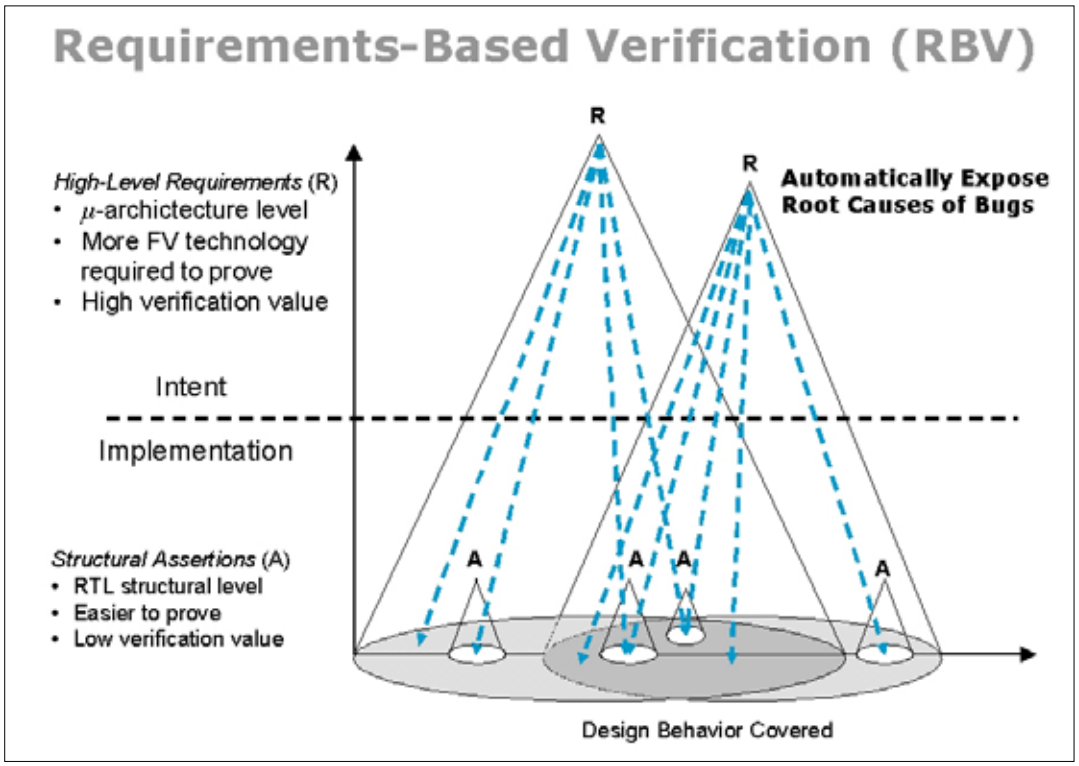

Figure 2: Here's a closer look at the requirements and RTL assertions.

High-level requirements

Many companies have adopted an assertion-based verification (ABV) methodology to reduce the time spent in verification while improving their overall verification effort. However, basic adoption of ABV is typically focused on localized RTL implementation-specific assertions, which are used in simulation. The aggregate of all internal assertions will not characterize or fully specify a block's end-to-end behavior as defined by the micro-architecture. Furthermore, these localized assertions are not reusable when the design implementation changes. In other words, by specifying a block's required black-box behaviors through end-to-end properties, high-level assertions (which we refer to as high-level requirements in this article) provide a much higher coverage of the design's functionality, and they are often reusable across various design implementations and multiple projects. More importantly, by formally verifying a block's set of high-level requirements, one can achieve significant gains in verification completeness and productivity. Hence, high-level formal verification eliminates the need for block-level simulation and dramatically shortens system-level verification. Let's take a closer look at high-level requirements, as shown in **Figure 2**.

The Y axis represents level of abstraction while the X axis represents the amount of design covered by a particular assertion or requirement. The higher up the Y axis we move, the more design space is covered by the high-level requirement. There is high value in proving these high-level requirements for a number of reasons:

- High-level requirements correlate to the requirements in micro-architecture.
- High-level requirements correlate to the set of output checkers in a testbench.
- High-level requirements cover the same design space as hundreds of lower-level assertions that the engineer thought to write.
- High-level requirements cover the design space that was not covered by missing lower-level assertions that the engineer neglected to add.

As an example of the last point, assume the design contains a FIFO, and the engineer neglected to write an assertion to check that the FIFO never underflows. This safety violation would be identified by a high-level requirement. However, by formally verifying the high-level requirement, the root cause of the high-level requirement violation can be traced. For example, if the FIFO were contained within the cone-of-influence for the high-level requirement, the underflow condition that caused the high-level requirement to fail would be detected.

An ideal formal verification tool requires capacity in order to exhaustively explore all possible input scenarios as well as controllability and visibility at any arbitrary point within the design (Table 1). JasperGold, for example, employs high-performance and high-capacity formal verification technology to exhaustively verify that blocks meet high-level requirements derived from the micro-architecture. It uses mathematical algorithms so no simulation testbench or stimulus is required.

Conclusion

Formal verification requires you to think differently. For example, simulation is empirical i.e. you use trial and error to try to uncover bugs that can take an intractable amount of time to try all possible combinations. Hence, it is never complete. Furthermore, since engineers have to define and generate a significant number of input scenarios, they are focusing their effort on how to break the design not on what the design is supposed to do. Formal verification, on the other hand, is mathematical, exhaustive and allows the engineer to focus solely on intent or "What is the design's correct behavior?"

The effort spent in verification implementation involves defining multiple input scenarios as part of the test plan, creating a functional coverage model, developing a testbench, creating input stimulus generators, writing directed tests, as well as the effort to run the tests, analyze the coverage metrics, adjust the stimulus generators to target unverified portions of the design, and then iterate on this process. In contrast, a pure formal verification methodology that focuses on proving a block's end-to-end, high-level requirements directly corresponding to the micro-architecture specification, enables users to dramatically increase a project's design and verification productivity while ensuring correctness.