Verifying Xilinx FPGAs the Modern Way: SystemVerilog

You can improve design quality and time-to-market by investing upfront in currently available verification tools and methodologies.

by Stacey Secatch
Senior Staff Design Engineer
Xilinx, Inc.
stacey.secatch@xilinx.com

Bryan Ramirez
Senior Design Engineer
Xilinx, Inc.
bryan.ramirez@xilinx.com

Running an "is it alive?" test and then instantly downloading an FPGA design to a board is no longer sufficient for system development. Due to the increasing complexity of modern FPGAs, designs now require the same level of functional verification that engineers once reserved solely for ASSPs and ASICs. The good news is that advanced verification techniques are ready for use in FPGA development and will improve the quality of your design right now.

While it's true that FPGA design entails no expensive masks to turn if you uncover a bug during system bring-up, ensuring that your design is correct before going to hardware is still critical to the success of your project. Finding and squashing bugs before heading to the lab will speed your overall design cycle and increase the likelihood of releasing your product on time, saving you and your customers money and frustration.

Copyright © 2008 Xilinx, Inc. All rights reserved. XILINX, the Xilinx logo, and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners.

The Xilinx design team recently developed a new methodology and SystemVerilog infrastructure for verifying the Serial RapidIO™ LogiCORE™ (SRIO). In our latest release of the core, a new intelligent buffer automatically reorders and manages transaction priorities should your system need to retransmit packets. For this verification project in particular, our engineers used SystemVerilog (simulated using the Mentor Graphics® Questa™ tool) on top of standard AVM base classes (provided by Mentor Graphics) to verify the interactions between our newly designed Buffer LogiCORE and our existing Logical LogiCORE, while ensuring compliance to the applicable layers of the RapidIO standard.

We think you'll find it easy to make use of some of the advanced techniques we employed in our verification project as you develop new, portable, robust test benches of your own. We'll also describe some assertion and functional coverage techniques we used to improve the quality of our design without even changing our test bench.

Abstract Test Bench Development

Transactions give us a way to track data movement and control events during simulation. One transaction class in the SRIO verification infrastructure represents RapidIO logical packets and contains member elements for each field. We also use transaction classes to represent other events and conditions within the core. For example, we use one scheduling class to indicate whether packets need to be replayed on the link interface, and another configuration class to represent read and write transactions on the host interface.

SystemVerilog interfaces abstract core signals in order to provide simple connections between the test bench and the device under test (DUT). The only classes that communicate with core interfaces are the drivers (which convert transactions to vectors) and the monitors (which convert vectors back into transactions). Figure 1 depicts vector-based interfaces with solid dark lines. All other connections between elements are shown as hashed gray lines and represent transaction-based communications. When processing the vector streams, the drivers and monitors use functions that are built into the transactions to translate between fields and data streams, as well as helper member functions such as comparisons.

Using abstracted transactions allows you to focus on what your DUT is supposed to be doing, rather than how it does it. For example, the transaction representing a RapidIO packet allowed our test bench to move and operate on a single entity without requiring knowledge about how the packet entered or left the DUT.

With transaction class definition complete, our next step was to specify how transactions travel through the core and how the test environment creates and consumes them. Individual test cases direct generators to create transactions and pass them along to drivers, as shown in Figure 1. Drivers and monitors contain protocol checkers to verify proper operation with external circuits. The main difference between the two classes is that drivers actively interface to the core, while monitors passively snoop what is happening. At the same time, functional coverage triggers measure what the test bench has exercised within the core.

At the end of a transaction sequence, the driver or monitor sends the transaction to the scoreboard, as shown in Figure 1. The scoreboard contains a reference model for the DUT to verify that it behaves as expected. Specifically, the reference model checks that the core correctly arbitrates between interfaces, constructs proper packets and retransmits packets correctly when required.

The SRIO core uses Local Link interfaces, not only for external communication, but also among all of its top-level modules. An instance of a common monitor class checks each Local Link interface in order to catch problems as they occur within the core, rather than waiting for them to become visible on external interfaces. Using standard interfaces in your design detects potential problems in your circuit early, allowing for quicker debug. Flagging an error immediately eliminates the need to track the problem back through a sequence of events.

Randomization and Coverage

A solid test bench infrastructure is important, but it doesn't do you any good without transactions that really exercise your

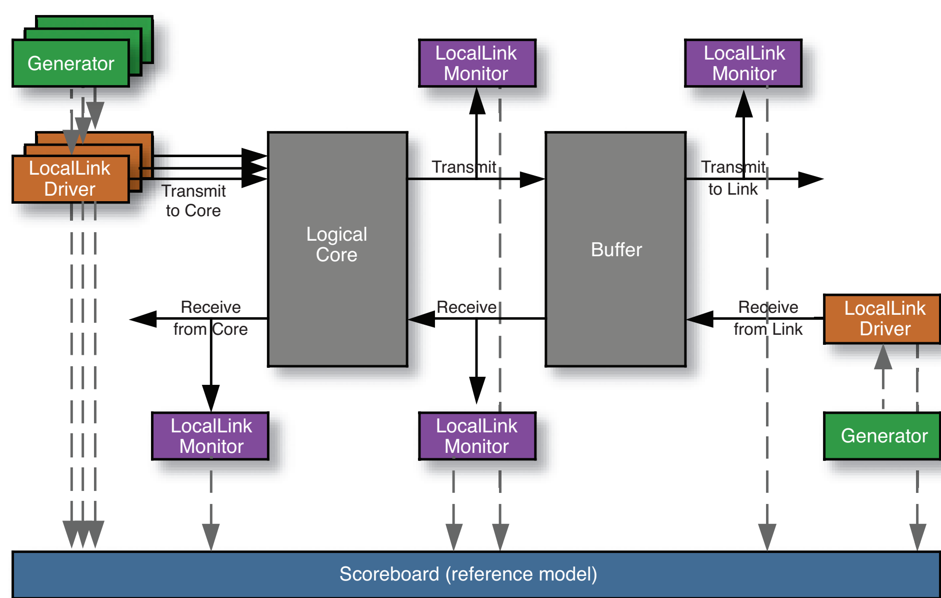

Figure 1- Simplified diagram of the SRIO cores surrounded by the test bench. Solid dark lines represent vector-based interfaces, and hashed light lines represent transaction-based interfaces.

design. The old style would be to create a test plan and write a directed test to cover the functionality you care about. Unfortunately, advanced verification still requires a test plan. You can cover a lot of the functionality automatically by using the built-in constrained randomization of SystemVerilog. One benefit of randomization is the ability to hit hidden test points that you haven't actually defined. After finding these new points, you can add them to your test plan to ensure the simulation environment exercises them.

We used the randomization engine to automatically create valid packets with varying content, and to issue them against the core with any valid timing. This guarantees that we tested the core against any kind of packet, in any sequence, with any other kind of packet already active in the core. This methodology very quickly proved the core data path was able to handle any possible combination of packets and for the core control to handle any possible state.

Defining Test Stimulus

Completely random vectors will exercise the DUT, but won't often provide challenging vectors for it to process. What's needed is useful test stimulus. We recommend starting with very simple tests to check the basic functionality of your design. To illustrate, our very first test case sent only one packet in each direction. Sending a single packet verified that the communication between the logical and physical layers was correct, and that the test bench itself was working properly.

Once that basic test worked, we turned on randomization. This changed packet content and size, as well as timing on all of the interfaces—a simple step to help us test more thoroughly. We then incrementally turned on increasingly complex tests to fully verify the core. This stair-step approach also allowed us to turn off some of the simpler tests and save simulation cycles as we progressed, as the more-advanced tests had already covered basic functionality.

To quickly get to tests that were interesting, we set up test cases that constrained portions of randomization space. For example, to test interface arbitration within the transmit side of the Logical core, we used fork/join constructs to direct the generators in parallel. Each generator created transactions that we constrained to each of the logical transmit interfaces at or near the same time in simulation. This shortened the testing time, because we could simulate this functionality directly, without requiring the constraint solver to bring us there as part of random simulations.

Once regressions were up and running, we examined our functional and code coverage data. As expected, there were coverage gaps. We wrote directed random tests targeting the areas that randomization didn't adequately address. For example, we added a test case that changed data throttling on the user and link interfaces to simulate the buffer while mostly empty and mostly full. When closing coverage gaps, a hybrid approach using directed random test stimulus tends to be the most effective. By focusing the random vectors toward the area of interest, the stimulus is more useful and the benefits of randomness are maintained.

Immediate Adoption: Assertions and Coverage

The SystemVerilog language is divided into verification and development components. Binding is a verification construct that connects one module to another. It allows verification code to directly observe register-transfer-level (RTL) synthesizable code without making any changes to the RTL source files.

Binding enables you to start adding more-advanced verification features without having to fundamentally change your test bench. All you need do is create a new module with the verification constructs and "bind" it to your code. This white-box testing method allows immediate identification of bug events, in addition to determining that desired testing events have occurred.

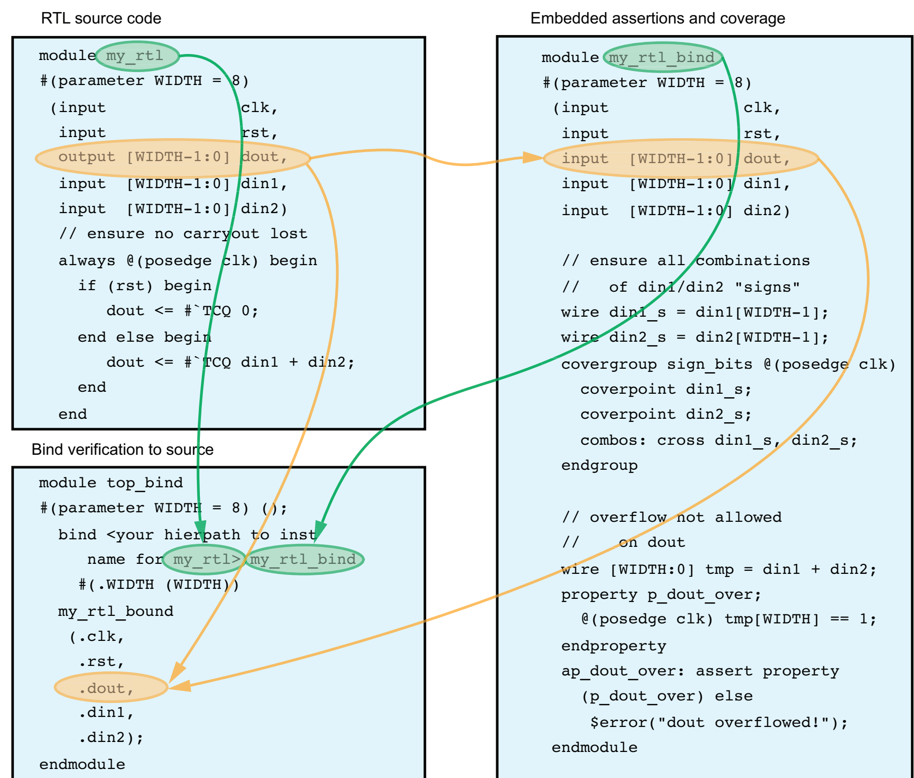

Figure 2- Binding embedded assertions and coverage example for a simple adder. At the top-level simulation, also include "top_bind" as a simulation target.

For each RTL module, we created an embedded verification module containing assertions and functional coverage. We first copied the port declaration for each RTL module, but changed output ports to input ports. Then, we added parameters to the verification module for any parameters or local parameters that we needed access to. Finally, we added input ports corresponding to any internal signals we wanted to probe.

The final step was a wrapper module to bind the verification modules to the existing RTL modules. The SystemVerilog bind keyword in combination with implicit connect-by-name made creating the wrapper module very simple. Note that SystemVerilog doesn't allow implicit connections for parameters, so they must be explicitly called out as ports at this level. Hierarchical references allowed access to signals and parameters residing in submodules.

This wrapper module becomes a secondary top-level module when simulating the design. Figure 2 shows this methodology at work on a simple adder example. You can also create any temporary signals you might need inside the verification module.

Embedded Assertions, Functional Coverage

We used SystemVerilog assertions to quickly detect any problems. Like internal monitors, assertions immediately find any bug that has been triggered, without requiring it to propagate to a port for analysis. SystemVerilog assertions are very flexible; we primarily focused on verifying correctly formed input (such as malformed packet demarcation and inconsistent control signals) and error detection (such as X detection and overflow conditions). The assert property in Figure 2 detects an overflow condition on the carry bit.

We found embedded assertions to be most useful for detecting initial failing events to sequences that cause functional failures much later in simulation. They were even more useful when the failure happened in logic not related to the source of the issue. Assertions allowed the failure to immediately present itself during simulation and be located in the incorrect source logic.

In addition, we used SystemVerilog covergroups and cover properties to ensure that the test coverage for the DUT was adequate. In general, covergroups worked best to check signal values or when we wanted to ensure that combinations of values were available using cross-coverage (such as a discontinued packet with every possible priority). Cover properties excelled when we wanted to detect sequences of events, such as a discontinued packet immediately followed by a valid one. The covergroup in Figure 2 provides coverage for all permutations of the high bit of each addend seen in simulation.

Embedded coverage was most useful in two situations. The first was to prove that internally generated sequences covered all possible values, since the test bench doesn't directly control those sequences. The second was to prove that buffers were in all possible states during simulation (empty, full and every possible one-from-full configuration). Moreover, embedded coverage lets you fully measure the testing of coverage points created as a result of implementation details rather than specification requirements. You can improve the quality of your design by ensuring that you cover these points during testing.

Looking forward, we'll continue to use this base methodology on our LogiCORE IP cores while finding ways to integrate new verification techniques such as OVM (a new base class library developed jointly by Mentor Graphics and Cadence™). There are a lot of helpful books and white papers available about high-level test benches. For usable examples, we like the *Writing Testbenches* series by Janick Bergeron and the *AVM Cookbook* from Mentor Graphics Corporation. We found the SystemVerilog language reference manual to be irreplaceable.

For details about the Serial RapidIO protocol, go to the trade association Web site at *www.rapidio.org*. For more information about the Xilinx Serial RapidIO LogiCORE and the new buffer design, go to *www.xilinx.com/rapidio*.