

**AS THE COMPLEXITY OF ASIC DESIGNS GROWS AND DEVELOPMENT SCHEDULES SHRINK, DESIGN ENGINEERS MUST SEEK WAYS TO IMPROVE THEIR PRODUCTIVITY. THE BEST APPROACH FOR THEM IS TO ADOPT NEW DESIGN METHODS.**

# Design verification and debugging FPGA implementations

**V**ERIFICATION TAKES as much as 70% of an ASIC's development time and resources. With growing ASIC complexity, verification problems are growing exponentially. Given the high cost of ASIC mask sets, the financial impact of a silicon respin is substantial. As a result, ASIC designers tend to prototype their designs on FPGA platforms before tape-out.

FPGA prototyping can offer substantial performance gains, resulting in a 1000- to 100,000-times throughput improvement over HDL simulation. The recent advancements in FPGA densities have enabled designers to build FPGA prototypes of large ASICs without worrying about FPGA logic-resource usage and changes that may result in higher gate counts. Today's FPGAs with integrated high-performance microprocessors have millions of gates, and compete directly with low-range to midrange ASICs. Although these complex systems offer many advantages, they also require significant verification with superior debugging capabilities.

## VERIFICATION METHODS

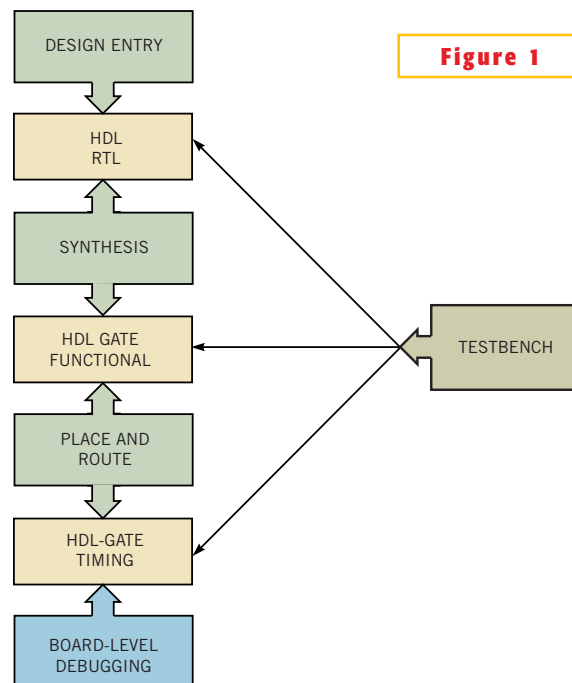
Verification is usually necessary in the early design stages to establish an implementation's correctness with respect to its logical specifications. In addition to verification at the manufacturing and programming stages, FPGAs need verification before implementation at the functional level and postsynthesis level and after implementation, which includes the timing of the routings and pinout delays in the FPGA.

Design verification, including functional and timing verification, takes the major part of this cycle. An increase in design complexity causes a dramatic increase in the simulation time. **Figure 1** shows the design flow for an FPGA implementation. You can use

Testbench, a stimulus file used for verification, at each stage to verify that the design matches the RTL.

## SIMULATION

Simulation is a widely used verification tool. It is easy to use, supports large designs, and has many debugging tools with a large base of trained users. Logic simulation is essentially a reactive tool: The designer needs to speculate about where he or she may find potential design errors, write a testbench that exercises that area of the design, and hope that the



**You can use Testbench at each stage for design verification.**

simulation results will expose any problems. This hit-and-miss approach is time-consuming, both in writing the testbenches and the use of workstation resources to run the regression tests as they get larger. The difficulty comes from engineers using simulation as a “white-box” verification tool—that is, when engineers use simulation with directed tests that exercise particular corner-case scenarios in large designs. It is difficult to conceive and write simulations that stress the implementation corner cases. Additionally, even after a verification engineer creates a test, it is difficult to reuse it when the design changes or a new design integrates the block. In many cases, the verification engineer must tweak the test before its reuse.

However, relying solely on simulation for logic verification is unsatisfactory because it allows you to test only a small fraction of a large circuit’s functions in a reasonable amount of time and because faults and errors often result from unexpected situations, which you are unlikely to have fully checked in advance. Code coverage is a part of a coverage-directed verification method. You can make faster, measurable progress as coverage feedback guides you on what areas require additional verification.

### **FORMAL AND SEMIFORMAL VERIFICATION**

Formal verification is a relatively new verification technology that uses mathematical algorithms to verify a post-placement-and-routing netlist that provides the same function as a pre-placement-and-routing netlist. Formal-verification tools do not require the user to create any test vectors and can significantly accelerate verification efforts for large designs. Formal verification provides a systematic technique. The most common problem is that these tools fail when they hit a capacity limit in time, memory, or both. This situation happens with almost any reasonably sized design.

Another subtle problem is that you need to carefully constrain the environment of the design for the formal tool. If you allow the tools to use any input, they will often find false errors, which could not happen with the design operating in its intended context.

Despite these problems, in practice, verification engineers—usually, teams of

specialists—have successfully applied formal verification to selected properties on selected parts of a design. These techniques do not address the problems of finding bugs during verification purgatory. This problem leads to the concept of semiformal verification. The idea is to combine the strengths of simulation—namely, ease of use and the ability to handle large designs—with the thoroughness of formal verification.

### **CONSTRAINED-RANDOM METHOD**

The constrained random method is a powerful verification approach that simplifies preventing a random test from generating illegal operations or illegal sequences of operations. By constraining the design, verification engineers tell the tool about the real-world environment in which the chip needs to work. This information can be about the limitations on the inputs, address lines, or other parts of the design. By imposing constraints, engineers give the other tools more knowledge about the size of stimulus that makes sense for the design. Instead of specifying each event to exercise the design, the engineer specifies ranges, within which the testbench exercises the target design. This approach limits waste-simulation cycles, because illegal logic never propagates through the design. Using constrained-random transactions more realistically tests the system. Constrained-random stimulus generation reduces the burden on verification engineers, allowing them to quickly and efficiently create testbenches. This method also simplifies the job of verification engineers and enables them to quickly complete the verification task. As a result, they can spend more time in simulation.

### **BLOCK-LEVEL VERIFICATION**

Verification at the block level is a more tractable problem than is system verification. The design unit is smaller, and a designer can more easily observe and control its function. This approach yields significantly shorter simulation, and when you apply thorough verification at this level at relatively low cost, you’ll later reap considerable savings.

Extensive module or block testing allows you to instantiate the design in a system and confidently regard the result as a “black box.” This method significantly

reduces the verification problem and minimizes the risk of finding a bug within the module. Code coverage at the module level catches the problems when the cost of debugging is lowest. By creating better designs in the first place, your verification improves because you have fewer bugs to deal with.

A good method is to do as much block-level verification as possible to find more bugs and to validate more functions before handing off the design blocks to full chip-level simulations. This strategy allows you to more easily find low-level functional bugs. It also enhances debugging efficiency because block-level simulations are faster. Because the HDL code is still fresh in a designer’s mind, he or she can more easily debug these blocks. Once the block matures, the designer can combine it into full-chip simulations.

### **USING LOGIC BLOCKS AND VENDOR CORES**

Make a library of the logic blocks that you’ve verified and used in previous designs and use them in your future designs. You can also use ready-made cores that you can purchase from a third party, which designed and optimized them for a particular implementation technology. These cores speed development by reusing designs and verification code.

As circuit designers exploit the increasingly large silicon resources available to them, they rely more and more on cores to quickly and affordably develop systems. One of the key values of a core to a customer is that the core vendor has subjected it to verification.

Xilinx and Altera, two key players in the FPGA market provide IP (intellectual-property) cores for many applications. Xilinx’s CoreGen tool allows a user to specify various parameters of a core and then automatically produce a circuit for a particular FPGA. The designs are available in low-level circuit-netlist formats, such as XNF (Xilinx Netlist Format) or in EDIF (Electronic Data Interchange Format).

### **MORE HINTS ON FPGA IMPLEMENTATION**

Use the right FPGA density and speed grade to achieve the best timing result. Using more than 80% of FPGA resources in a design might reduce its overall performance. In these situations, use denser

FPGAs. Implementing a small design in a dense FPGA does not improve timing and might have adverse effects. You must partition large designs across multiple FPGAs; tools such as Certify from Synplicity and SpeedGate from Mentor can help.

Improve your coding style. Bad coding causes mismatches between functional and postsynthesis simulations. For synthesis, use products from Synplicity or use Precision from Mentor Graphics.

Overconstraining your design can lead to unwanted results. For example, if your design operates at 50 MHz, do not synthesize it to more than 10% of the maximum operating frequency of the design—in this case, 55 MHz. This approach gives you the best results, and increasing the synthesis frequency adversely affects the circuit performance. By unrealistically increasing the synthesis frequency, the synthesis tool tries to overconstrain your design by using high-speed buffers and generating parallel logic that leads to an increase in power consumption and uses more logic resources.

Check the user-constrained file for possible constraint errors. Using the wrong pin for a specific signal, such as a clock signal that should use a dedicated clock pin, is a common mistake. Review the pad report and make sure that you use the right pins for all I/O signals.

If your target is FPGA Spartan or Virtex from Xilinx, you can use ChipScope as a logic analyzer. The ChipScope tools integrate key logic-analyzer hardware components with the target design inside the Xilinx FPGAs. The ChipScope tools communicate with these components and provide designers with a complete logic analyzer without cumbersome probes or expensive test equipment. Always dedicate some unused pins of the FPGA for future debugging and testing. Allocate these pins while you are designing your prototype FPGA board and wire them to a connector on your prototype board. □

---

#### AUTHOR'S BIOGRAPHY

*Hossain Hajimowlana, PhD, is a staff engineer at Analog Devices' DSP Tools Group, where he designs new products and high-performance controllers. He has a doctorate in electronics engineering from the University of Windsor (Canada), a master's in electronics engineering from KNT university (Tehran, Iran), and a bachelor's in electronic engineering from Polytechnic University (Tehran, Iran).*