



Debug Dilemma

Simulate or Emulate?

What goes in software, and what goes in hardware? In most complex digital designs, the answer to this key question will determine success or failure of the architecture of the system. Put the wrong piece in software and performance suffers from overloading the processor. Put the wrong piece in hardware and your cost rises from the additional gates, static power consumption goes up, and flexibility and maintainability of the system drop significantly.

It turns out that the software versus hardware battle is going on in parallel in your design environment as well. With modern FPGA development boards, embedded logic analyzers and debuggers, and purpose-built emulation systems, hardware in the loop (HIL) debugging and verification has rocketed to prominence. HIL is not the panacea of programmable logic debug, however. As our friends in ASIC design can tell us, software simulation, formal verification, and yes, HIL emulation all have a significant contribution to make to our design process. As in system architecture, the key decision is what

part of your design and verification should be conducted in software versus hardware.

Debug and verification account for the lion's share of time and effort in any significant digital system project, whether targeted to FPGA, structured ASIC, or cell-based ASIC technology. For ASIC design teams, verification is a life-or-death proposition where the success or failure of the project depends on getting the design right the first (or more likely second) time. For FPGA-in-production teams, the penalty of verification failure is much lower, but there still is a sinister swamp of stagnation awaiting those who go too quickly into hardware and never emerge from the quagmire of interrelated bugs.

If we look at the design process stage-by-stage, it becomes apparent that certain techniques shine at each step. Thinking carefully through the flow, we can develop a methodology that leverages the strengths of transaction-level simulation, RTL simulation, formal verification, HIL debug, and full-blown

emulation. Each of these methodologies can pay for itself tenfold if we use it strategically and judiciously.

Chaos control

In the beginning of any design project, there is generally a great deal of disorder. A rough specification suggests a starting architecture that directs the acquisition of certain IP and the dusting-off of old design modules from previous projects. The first order of business is to get something together that can execute the entire system-level algorithm at the highest possible level of abstraction. We want to establish a framework from which we can validate our top-level system design and develop an appropriate architecture for our eventual implementation. Leading ASIC design teams turn to high-level language modeling and transaction-level simulation for this task. This approach provides the fastest possible execution of a software system model with the shortest startup time. It also requires the least detailed understanding of the underlying architecture of the system components.

At the transaction-level modeling stage, many teams make their first-order decisions on system partitioning and hardware/software tradeoff. They also begin the equivalent of a make-versus-buy decision on many of the larger system blocks. In general, the more of your system you can build out of reliable, proven, ready-made IP, the shorter and more predictable your design cycle will be. Verification will be simpler, the scope of debugging will be reduced, and design complexity for your team will drop dramatically. As a result, your time-to-market savings almost always offset the cost of reasonably licensed IP.

While transaction-level simulation can give you a reasonable understanding of the functionality and viability of your overall system design and can help you make and evaluate the architectural decisions that will guide the rest of your design process, there are alternative HIL techniques in use by FPGA design teams that are proving similarly effective. Using soft-core embedded processors, they initially model their systems purely in C code and run the code as a set of communicating processes on the embedded soft-cores. This has the advantage of getting the design up quickly in hardware, but with little or no actual hardware design completed. The partitioning of the software model lends itself to performance analysis using embedded debugging and performance monitoring tools. The modules that prove to be performance bottlenecks as software are obvious targets for hardware acceleration.

The advantages of the HIL techniques at the system level are extremely fast execution of the model and the possibility to evolve the model directly into working a hardware/software system as software modules are gradually replaced with hardware accelerators. The disadvantages include fairly lengthy cycle time, comparatively limited visibility, reduced controllability of the model (compared with a software simulation), and a fair amount of home-brew methodology development.

Finding focus

Once our system is proven from an overall algorithmic point of view, partitioned into basic modules, and ready for implementation, the task turns to integration. For ready-made IP, the problem is usually debugging interfaces at the periphery. For embedded software, the challenge is finding a suitable platform for development that reasonably reflects the target hardware environment under development, and for original hardware modules, the problem is detailed debug at the register-transfer level.

For the RTL debug portion, traditional HDL simulation shines. If you're a long-time FPGA (or ASIC) designer, this should be your comfort zone. For the majority of FPGA designers, Mentor Graphics's ModelSim or Aldec's ActiveHDL are the simulators of choice, and they've both been recently upgraded with extended capabilities that shine in this integration stage. When you're debugging original HDL code in the context of a design that might have higher-level language modules, mixed HDLs, and encrypted IP, you need a very sophisticated simulation environment to execute your design and give you the visibility and controllability that you need to find and fix the bugs that you've introduced in your original code.

HDL simulation can be so intoxicating that many teams stick with it too far into areas where other approaches would excel. The key is to monitor your bug discovery rate and move on to a different approach when the rate of finding new bugs drops in proportion to the time required to fix them. "Observability, controllability and turnaround time are the key advantages of simulation," says Steven White, General Manager of Mentor Graphics O-In Functional Verification Business Unit. "When your bug discovery rate drops below a certain level, it's time to move on to emulation."

Going for speed

When your design starts to seem stable through the window of simulation, it's time to speed up the process and expand your coverage. While simulation

is great for setting your design into known states and evaluating corner conditions, it is inherently performance-limited for broad-based exercise of your system in a wider variety of situations. For that, you need to move at least the key elements of your design into hardware. If you're lucky enough that your entire system can fit into a single FPGA platform, (which is becoming the case more often with today's larger FPGAs) you can be pretty effective just moving the design into your FPGA on a development board. Instrumented with the latest embedded logic analyzers, debuggers, and performance analysis tools, you can make rapid progress on final debug with actual hardware-based stimulus going through real-world physical interfaces.

If your design would require multiple FPGAs, however, or if you're planning to move on to a cost-reduced solution such as cell-based or structured ASIC, you should consider a purpose-built emulation solution. "As your design grows into multi-million gates," says Lauro Rizzatti, President of Eve USA, "the performance of the simulator drops to such a degree that it is impractical. The number of cycles required to verify a design varies on the order of the square of the number of gates. There are bugs where millions or billions of cycles must be executed in order to force the problem to show up at an I/O. Finding these bugs is one place where emulation comes into play."

Many design teams try to save money by creating their own FPGA-based environments for this step, but these efforts typically result in more time being spent on the emulation environment than on the design itself. Instead of developing your own emulator, it makes economic sense to use one of the ones already designed for you. Emulation companies have invested significant engineering effort in producing systems that will give you the maximum performance, visibility, and controllability with the shortest cycle times possible, and trying to re-invent their wheel as a by-product of your own design project just doesn't make sense. "Beyond eight FPGAs or so, the success rate for teams doing internal FPGA-based prototypes is very low," Rizatti continues. "Also, the lack of compiler support, limited visibility, and performance demands of the circuit board make such prototypes impractical compared with commercial emulation systems."

Another place where emulation comes into play is on the software side of the house. Software designers, desperate for a platform where they can develop and debug their portion of the design, are increasingly relying on emulation-based environments to give them a semi-stable platform that resembles the hardware architecture of the actual system. "Emulation can provide a significant, differential advantage when it comes to software development," continues Mentor's Steven White. "By making careful, controlled releases of

the hardware design to an emulation environment, development of hardware and software can be parallelized."

Matching up the parts

As you progress through the debug and verification of your design at various levels of abstraction from behavioral/transaction to gate and timing detail, it can be difficult to match up the work you did on each level with adjoining ones. From the behavioral level to the RTL level, the move from transaction to cycle-accurate tests often invalidates testbenches. The possible remedies are to develop a comprehensive strategy that insulates the testbench from the implementation details (which can also reduce test coverage) or to resolve to re-design many of the tests at each level of abstraction. Usually, the latter approach prevails, as the effort to develop a unified stimulus strategy often is greater than the value it returns.

When FPGAs are not the final technology target, emulation and prototyping run into another problem: how to maintain the fidelity of the FPGA-based model compared to the ASIC design while accounting for the technology differences between the two. Significant discrepancies can arise in areas like clock-trees, IP availability, and timing accuracy that make the programmable logic version deviate significantly from the eventual ASIC. In addition to the capabilities built into your favorite emulation package, there are commercial software solutions to this problem as well. Our contributed article, "Deliver Products On-Time with RTL Hardware Debug," goes into depth on one such solution from Synplicity.

Putting it all together

The key in getting the most from your verification and debug environment lies not in choosing a particular approach exclusively, but in understanding the relative strengths and weaknesses of the various techniques available and designing a methodology that leverages those to their maximum advantage. Transaction level simulation, RTL simulation, and hardware-based prototyping and emulation all bring capabilities to the party that justify their use in many complex digital system design projects. By understanding which one to use in each phase of your project, you can dramatically cut your design time and increase your chances of success. In today's highly competitive environment, the time-to-market that you gain can mean the difference between a product that is a home run and one that is an also-ran.

Kevin Morris, FPGA and Programmable Logic Journal

January 11, 2005