

**OPTIMUM CO-VERIFICATION TECHNIQUES GIVE DESIGNERS
A BETTER UNDERSTANDING OF SYSTEM PERFORMANCE AND
A SHORTER TIME TO MARKET.**

Co-verification speeds SOC design

AS DESIGNS BECOME MORE complex, standard co-verification techniques are running out of steam. At the same time, many complex SOC (system-on-chip) designs are still using nothing more than a full-functional simulation model of a microprocessor and waveforms to debug complex hardware and software. A full-functional model of a microprocessor fetching and executing code in a logic simulator is not co-verification if the only means of debugging software are waveforms and assembly-language traces. You can realize the difficulty of the problem by looking at the vast array of alternatives available to verify both hardware and software before first silicon is available.

The ultimate goal of co-verification is to get the entire system—hardware and software—working before the prototyping stage by providing better visibility into its behavior (see sidebar “What is co-verification?”). You must debug using familiar development tools, as **Figure 1** shows. Co-verification achieves its goals by providing two primary benefits. First, software engineers have much earlier access to the hardware design, which allows software designers to develop code and test it concurrently with hardware design and verification. Performing these activities in parallel shaves more time from the project schedule than the serial method of waiting for the prototype to begin software testing. Moreover, the early involvement of the software team results in a much better understanding of the underlying hardware operation. Second, co-verification provides additional stimulus for the hardware design. In fact, it can provide the true stimulus that will occur in the embedded system, providing better hardware verification than a contrived testbench that may not represent real system conditions. The increased

confidence in the hardware design is invaluable.

By running hardware/software co-verification, you can find and fix a range of problems prior to silicon, such as register-map discrepancies, problems in the boot code, errors in DMA-controller programming, RTOS boot and configuration errors, bus-pipelining problems, and cache-coherency mishaps. Some of the errors will be software problems, and some will be hardware-related. You must attack this laundry list of example problems using a logical and well-defined verification strategy. The keys to a successful project are applying the correct tools to the problem, understanding the types of problems you are looking for, and comprehending the parts of the system you assume to be working at each stage of the project.

OVERVIEW OF SOC DEVELOPMENT

The traditional notion of the hardware design as the device under test and the testbench you use to stimulate the device under test is outdated. The device under test now includes software, and the test-

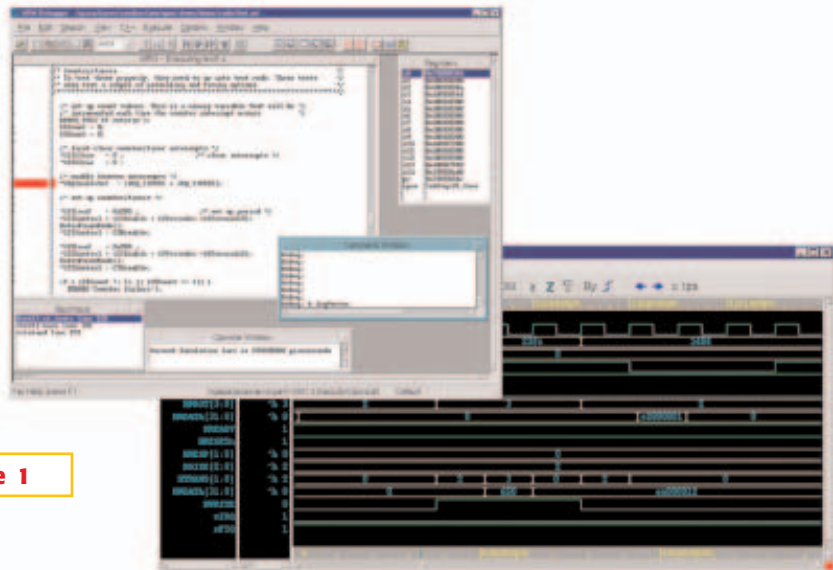


Figure 1

Efficient hardware/software co-verification employs standard development tools for debugging.

bench must verify how hardware and software operate together. The software-design process proceeds through the five stages—system-initialization software and HAL (hardware abstraction layer), hardware diagnostic test suite, RTOS, RTOS device drivers, and application software. The software content (lines of code) increases with each step.

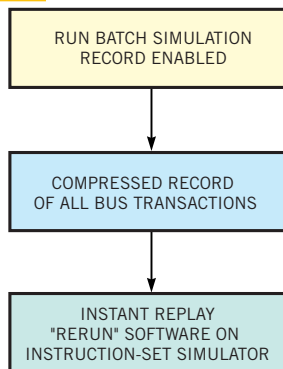
The abstraction of the hardware, along with stability and function assumptions, also increases with each stage of software development. Likewise, four methods are common for executing the hardware-design description in SOC design—logic simulation, simulation acceleration, hardware emulation, and hardware prototyping.

Each of these hardware-execution methods has specific debugging techniques for both the hardware and software associated with it, and each has its own set of benefits and limitations. These techniques range from the slowest execution method, with the most thorough debugging, to the fastest, with less debugging. The hardware-simulation detail also varies among hardware-execution methods. Certain phases of the project require more simulation detail (usually hardware-verification tasks), and others benefit from less simulation detail (usually high-level software applications).

SYSTEM INITIALIZATION

The first embedded-coding task is to write and test the processor-initialization software. This code includes configuring the operating modes and peripherals, such as cache configuration, memory-protection-unit or memory-management-unit programming, interrupt-controller configuration, timer setup, and DRAM initialization. This coding task is tedious using assembly and C languages, but it is important, because all other software re-

Figure 2



Instant-relay features allow software engineers to replay simulation results from a recorded file instead of repeating the hardware simulation.

lies on it, including the final product.

The HAL is the next layer of software that works with the initialization code to provide a common interface for higher level software to use for hardware-specific functions after the system is initialized. The HAL abstracts the underlying hardware architecture and the platform to a level sufficient for you to port the RTOS kernel onto the platform. For example, consider a system that has an interrupt controller with a prioritized interrupt scheme. The RTOS-porting engineer would like to change the priority using a simple C function and a priority level, such as `ipl(5)`, to set the priority level to 5. The detail of disabling interrupts and using an atomic read-modify-write sequence to change the interrupt-controller registers and then enabling interrupts again is uninteresting and unnecessary.

Many complex SOC projects use nothing more than a full-functional model of the microprocessor core in a logic simulator to write and debug this code. Soft-

ware debugging with waveforms requires a guru who understands hardware and software and can disassemble instructions in his head using instruction fetches on the data bus.

For the ARM SOC example, the ideal debugging option for early development of system initialization and HAL code is one based on a cycle-accurate instruction-set-simulation model tightly coupled to a logic simulator containing the SOC hardware design. This method provides interactive, graphical software debugging for the software engineer to single step through the code and verify register and memory contents with excellent control. Simulation performance is less important, because you must verify the code line by line, and the number of lines of code is relatively small. A standard Verilog simulator is more than adequate to execute the hardware design and provides good visibility into the hardware. Interactive software debugging by a graphical debugger interface is the ideal way to control software execution and software operation. It is common to encounter many hardware bugs during this phase, because it generally occurs soon after you've integrated the blocks of the SOC into a full-chip simulation and is the first time all of the subsystems are working together using the defined hierarchy of shared buses.

DIAGNOSTIC SUITE

Once the initialization software and HAL are stable, the next phase of software development consists of developing a detailed test suite for the hardware design. In the past, this phase usually took the form of a hardware testbench. Although testbenches are still necessary to provide stimulus for external interfaces, such as networking protocols, the software now serves as the testbench for the CPU core bus. You should develop a comprehensive set of diagnostic tests to verify each subsystem and peripheral. These tests start with the memory subsystem, progress to interrupt testing, and then move to other IP blocks, such as timers, DMA controllers, video controllers, MPEG decoders, and other specialty hardware. Most of these tests do not see their way into the final product, but they are very important, because they build the case for a solid hardware design.

WHAT IS CO-VERIFICATION?

The term "hardware-software co-verification" was coined in the mid-1990s to describe a process that allows you to execute embedded-system software on a simulated representation of the hardware design. The simulated representation of the hardware usually means a Verilog or VHDL de-

scription running in a logic simulator. Designers have since extended the co-verification concept to allow for other representations of the hardware that are not the final implementation, including logic-emulation systems and other prototyping methods.

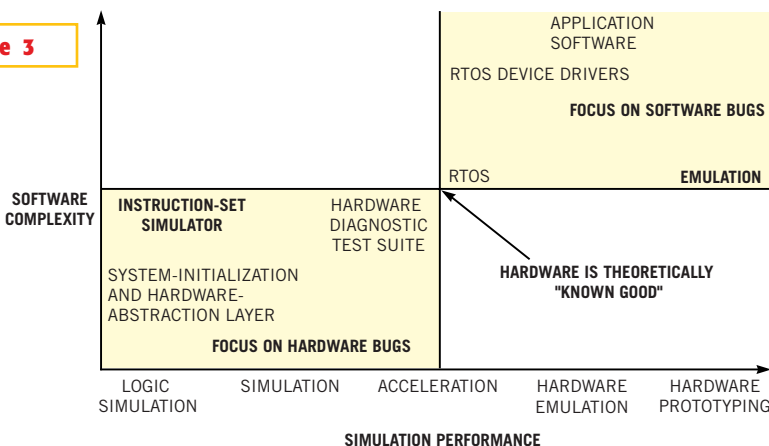
Creating the programs gives software engineers a good understanding of the hardware and serves as a chance to learn about the hardware specifics in a more secluded environment.

During this development of diagnostic tests, the logic simulator begins to become the bottleneck of the verification environment. As tests run longer and the number of tests increases, it becomes increasingly difficult to both verify the entire hardware design and continue to run the old tests as you fix hardware and software errors. This phase is also the most crucial, because it is where you find most of the hardware bugs. Debugging tools for both software and hardware at this stage are important. The ideal option uses simulation acceleration to increase the performance of logic simulation from what is possible using an ordinary software simulator. A simulation environment running at 10 to 100 Hz is too slow for software engineers to run today's longer tests. Moreover, the memory-optimization techniques common in co-verification are not useful, because the main purpose of the diagnostics is hardware verification. To abstract the hardware using time-domain techniques and to stray from a cycle-accurate simulation go against the purpose of the tests. A simulation-acceleration system that runs at 1 to 10 kHz is the ideal platform for simulation performance and debugging.

During this phase of the project, debugging is equally as important as speed. Traditional hardware-debugging techniques, such as waveform dumping, are far too inefficient. A value-change dump-on-demand technique allows engineers to extract all node-history values from any point in simulation without resimulating from time zero and is another reason to deploy simulation acceleration during this phase. The combination of simulation acceleration and co-verification for software debugging is the best way to create and verify the hardware diagnostic suite.

As the test suite grows, many tests are run in parallel, probably overnight in a batch environment. If specific tests fail, engineers must analyze the results. Although hardware engineers are familiar with postprocessing debugging methods, software engineers are not. A debugging

Figure 3



A complete methodology for debugging SOC designs concentrates on hardware during the early stages and then switches to software bugs.

method that allows software engineers to inspect the test results without rerunning the test is ideal.

INSTANT REPLAY

During batch simulation, you create a compressed file containing bus transactions at the processor interface. The file “records” the memory transactions, including an address, data, and simulation time stamp, along with interrupt information. After the simulation is complete, software engineers can start the instruction-set simulator and software debugger and “rerun” the software-execution sequence. However, this time, instead of interacting with hardware simulation, the program reads the results from the recorded file. This playback of the bus interface replicates the exact sequence of software execution. **Figure 2** depicts a diagram of instant replay.

Because the simulation now runs at megahertz speeds, software engineers can rerun the software as many times as necessary to find the problem—without looking at any simulation waveforms. The simulation time stamp helps correlate the software and hardware execution. This record-and-playback methodology provides an ideal way to debug long simulation tests that make interactive debugging unproductive.

One of the great benefits of this methodology and the diagnostic test suite is that the hardware, initialization sequence, and HAL are assumed to be good (pretty close, anyway) when the RTOS

work begins. The first assumption of the RTOS engineer is that the hardware is stable. The initial RTOS work consists of just getting it to boot on the platform. How big a task the RTOS is depends on how standard the platform is and how well the HAL was designed. During the initial porting phase, device drivers for application-specific hardware may be missing, but the RTOS can still boot.

The RTOS port is an ideal place to take advantage of memory optimizations common in co-verification, such as software-memory models. Software-memory models use a memory that tightly couples to the instruction-set simulator to retrieve instructions at a much faster rate than you could achieve using logic simulation. The result is less simulation detail about how the ARM SOC works but increased performance. Because the instruction-fetch path is well-verified, it makes sense to use the memory optimizations there, not back in the diagnostic test suite, as a workaround for a low-logic simulator. Co-verification includes important memory optimizations for software-memory models tightly coupled to the instruction-set simulator to increase software-execution performance.

DEVICE DRIVERS

The final stage of verifying the hardware design and system software is to develop and test the remainder of the device drivers for the RTOS. For the RTOS porting work and the device drivers, you should make the first attempt using sim-

ulation-acceleration mode. After you verify the drivers, you can use a faster hardware-execution method.

Once the RTOS boots and is stable, you can do future work using a faster execution method, such as an emulation system or a prototype. Here, debugging is a bit more difficult, but the number of hardware bugs is very small. The choice of emulation versus prototyping is a complex one involving several factors. The primary factors revolve around the number and type of required interfaces to exercise the SOC and the benefits you gain by running at prototyping speeds. You need to make a trade-off decision to assess the benefits of prototyping speed, such as 20 MHz versus emulation speeds of less than 1 MHz in exchange for the debugging difficulty. The decision also depends on the design confidence in the testbenches used to verify interfaces in simulation-acceleration mode. Emulation products provide interfaces to target systems used to verify real-world interfaces and increase design confidence. Emulation provides the necessary performance to execute the RTOS and device drivers. Combining emulation with a target board containing the ARM silicon is the ideal way to obtain the next level of performance, one to two orders of magnitude over that of acceleration mode.

APPLICATION SOFTWARE

Once the platform is complete, it is time to test application software. At the application level, the hardware is assumed to be correct and becomes more like workstation programming. If the application crashes, you can safely assume it is a software problem, not something wrong with the hardware. Projects that are too large for software-based logic simulation have long turned to logic-emulation systems to gain both speeds reaching 1 MHz and access to hardware interfaces to real-world stimuli. Recent systems have been run at speeds of 20 MHz or more, fueling the use of prototyping for higher speeds at the expense of visibility. These high-speed platforms are best for application development. Sometimes, waiting for silicon is best for applications, but whether you do so depends on your confidence in the hardware design and whether everything was verified.

Application developers usually want to interface to real network traffic, see things on the screen, and use the pointer or mouse. During application development, hardware and lower level software bugs are few and far between, and the software engineer focuses on providing robust applications with differentiating features for end users.

Figure 3 shows the complete methodology for debugging ARM SOC designs. As software complexity increases with each new layer of software, hardware-simulation performance also increases. The first phase of the project focuses on verifying hardware, and it is important to make sure you identify and fix most hardware bugs before you run more complex software, such as the RTOS. Later in the project, as the hardware stabilizes, you can achieve faster performance with less simulation detail using emulation. The best platform for all phases of the project combines logic simulation, simulation acceleration, and emulation to execute the hardware design with the appropriate ARM CPU models.

Today, more focus is on system-level design, viewing the total project as a complete system instead of the individual parts—chips, boards, chassis, and software. Managers are looking for ways to shorten the project schedule and increase confidence in the design. At the verification level, this method translates into selecting the best tools to build a productive environment for both hardware and software engineers. A single integrated system for logic simulation, simulation acceleration, and system emulation, along with the concurrent debugging capabilities of both hardware and software, provides the best platform to get the project done in the shortest amount of time. The ability to easily transition between these models and modes of operation provides the greatest flexibility and control over the entire verification process. □

AUTHOR'S BIOGRAPHY

Jason Andrews is a product manager at Axis Systems, where he develops software for verifying embedded-system designs. He holds a BSEE from The Citadel (Charleston, SC) and an MSEE from the University of Minnesota (Minneapolis).