In the field of design verification, the move to SystemVerilog has arguably been the most important change in the last few years. SystemVerilog offers a common language to express both design and verification intent. In doing so, it eclipses custom verification languages like “e” or Vera. At the same time as this migration to SystemVerilog, system-level validation—traditionally, the purview of hardware prototypes and in-circuit emulation (ICE)—has been going through a similar transformation. It has been adopting co-emulation and transaction-level methodologies and eliminating complex hardware targets in favor of flexible programmable testbenches. As these two trends gain momentum, it’s not surprising that SystemVerilog and hardware emulation are finally meeting. Moving forward, SystemVerilog holds the promise of a common language spanning simulation-based design verification as well as hardware-acceleration-based hardware-software validation.

System-Level Platforms

Building a system-level platform is the easiest way to deploy SystemVerilog in emulation. By using existing off-the-shelf transactors, design teams can quickly build a complete environment for most standard system-on-a-chip (SoC) applications.

Consider the example of a digital camera. Design teams would create a virtual platform to validate the image-processing design of a digital camera by mapping the design-under-test (DUT) into an emulator. The team would hook up four peripherals described via SystemVerilog and connected to the DUT via a set of off-the-shelf transactors. Typically, such peripherals would include a Universal Serial Bus (USB), liquid-crystal-display (LCD) panel, JTAG connector, and camera sensor (see Figure 1). By using off-the-shelf transactors, which are essentially reusable verification components, the task can be accomplished by writing less than 100 lines of SystemVerilog code. This represents a tremendous gain in productivity.

The platform is fast enough to run device drivers and see the application perform real tasks. For instance, the digital camera may take a picture that’s provided from a fake CCD sensor that takes its raw image from a reference file. It will then process the picture and download it via USB to visualize as a JPEG image. That end-to-end validation is powerful. It can include benchmarking the performance of the system, as all components are cycle accurate.

In this scenario, hardware waveforms will satisfy the needs of hardware verification engineers. For the software developers, satisfaction comes from the connection to a software debugger combined with emulation speed. Compared to a SystemC model of the entire platform including the design and test environment, which is popular with system-level engineers, the transaction-based emulation platform is both truly cycle-accurate and more familiar to hardware engineers.

Speed is what distinguishes transactors from other reusable verification intellectual property (IP) described via a hardware description language (HDL). Once in emulation, a design is capable of processing millions of requests per second—orders of magnitude more than what an HDL simulator can handle. That difference is not just quantitative. It has a direct impact on the nature of the verification that can be done. With simulation, one can verify that when the user presses the shutter of the camera, it generates an interrupt to the main processor. Another test can verify that writing to a certain device register triggers the CCD capture. With more speed, there’s no need to break the main user
function of the device (taking a picture) in small elements. It should just work.

The transactors that move data to and from the design need to be able to keep up with such high throughput. Instead of communicating with the design at the signal level, which is all that Verilog can do, transactors use SystemVerilog’s Direct Programming Interface (DPI) to exchange higher-level messages. Figure 2 shows a block diagram of a transactor.

![Block Diagram of a Transactor](image)

Figure 2: Here is a block diagram of a transactor.

A transactor includes two parts: a front end and a back end. The front end, which is written in SystemVerilog, exchanges commands/messages at a high level of abstraction with the testbench via a DPI interface. The back end, which is written in Verilog, is in essence a bus functional model (BFM) or state machine that implements the conversion of those high-level commands into bit-level signals. Rather than toggle bits individually, the testbench can stay focused on a higher level of abstraction. For instance, video frames from a digital camera in movie-capture mode can be sent without the concern for individual HSYNC/VSYNC signals, such as vertical and horizontal synchronization.

The actual signal wiggling is performed by a small state machine that’s synthesized. That state machine runs in emulation next to the design with the same speed as the design. This approach guarantees both performance and cycle accuracy.

**Structured Custom Testbenches**

System-level platforms are the best way to be sure that the final SoC will work with its associated software. Yet such platforms require fairly stable register-transfer-level (RTL) code. Conversely, emulation and SystemVerilog can be used much earlier in the design cycle—during the hardware verification stage, in fact—when design teams would use large testbenches and checkers that monitor proprietary internal buses and protocols. While transactors for those custom and proprietary interfaces aren’t readily available off the shelf, most application-specific-integrated-circuit (ASIC) design companies have developed internal IP that can be tweaked into custom transactors.

Previously, large testbenches have prevented the widespread deployment of emulation. They tended to be the bottleneck that would negate most of the acceleration benefits of emulation. It wasn’t uncommon for the testbench of a large networking chip to represent half of the simulation time. Any kind of hardware acceleration would be limited to a speedup of 2X—a non-starter in the world of verification.

An effort from within the electronic-design-automation (EDA) industry to put some order into the development and deployment of testbenches is the Verification Methodology Manual (VMM). This effort, which was introduced by Synopsys Inc., seeks to create high-performance functional-verification platforms that favor reuse and an easy-to-maintain and consistent code structure. Initially, coding a transactor for any given protocol may be more complicated than implementing a free-form spaghetti testbench. Yet it has advantages in the long run. Reusability is more likely to apply when the boundaries between what may change and what stays the same are firm. The object-oriented features of SystemVerilog provide a means for modifying and reusing transactors.

VMM is a methodology that specifies different sets of layers that constitute a testbench. It also dictates how the communication between those layers is supposed to happen. At the highest level of abstraction, a testbench consists of a series of tests to execute. What those tests actually represent and what pseudo-random data they use is implemented in a lower layer. All of the layers of the VMM are abstracted from signal level except the lower layer. There, both monitors and checkers are implemented.

This structure is a natural fit for transaction-level emulation. By simply replacing the lower-level monitors and checkers by their synthesizable transactor equivalent, designers can keep most of the testbench code unchanged. They also can execute the same tests much faster by targeting emulation. In addition, VMM has built-in, generic, pseudo-random generation. As a result, much longer and more interesting tests can be created painlessly.

It’s typical for tests targeted at simulation to be too short to be worth running in emulation. For instance, a 50,000-cycle test will take only a fraction of a second to emulate, whereas it would take several minutes if not even hours in simulation.

Increasing that test to 500,000 cycles wouldn’t significantly change its duration in emulation. But it would quickly make this test impractical for simulation. Using the example of
a network chip, those kinds of cycle counts represent the difference between sending three packets to the chip—hoping to trigger a problem—and sending 1000 packets and finding some deep side effects. It’s just not possible to create deep scenarios with too small a number of cycles in a simulation budget.

Another benefit of VMM is the reproducibility. When the chip is getting close to tapeout, bugs are becoming rarer and rarer. It’s not unusual for the bug rate to go below one new bug per day and the point where hardware managers start to consider the chip ready for tapeout. By allowing deep tests, emulation helps to ensure that those last remaining bugs will be caught before the chip is sent out to be fabricated. Once such a bug is found, there must be an easy way to reproduce it. Otherwise, the stress level of the design team increases with those transient bugs that only appear once—never to be triggered again.

In VMM, the designer may change a few numbers in the testbench code and easily change the size of the test sequence while maintaining a reproducible behavior.

Efficiency during those last weeks before tapeout is critical. By operating at the transaction level and offering reproducible pseudo-random generation, VMM helps to guarantee that the bug found in emulation can be reproduced and analyzed in simulation as well.

VMM’s hardware abstraction layer (HAL) consists of a toolkit of classes and methods to implement emulation-friendly transactors. What does a VMM HAL transactor look like? Remember that the external interface or application programming interface (API) of the transactor is the same as the pure simulation version. Only the guts of the monitor or checker change to accommodate faster throughput and message passing for emulation.

The SystemVerilog part of the transactor mostly consists of forwarding messages to the emulator using the predefined objects of VMM HAL. Those objects are called VMM hardware interface objects (vmm_hwh_if). Messages are transmitted with high speed and low latency to the hardware emulator. There, they are decoded by the back end of the transactor.

The transactor’s back end must be written in synthesizable RTL Verilog code. Each SystemVerilog object has a corresponding Verilog macro module. Every time a stream is created in SystemVerilog, ports are added in the Verilog code to transport the actual data. The transport and translation between the messages in software and the ports in hardware are transparent and irrelevant to the designer implementing the transactor logic. What remains to be designed is the control logic or state machine. It processes the messages present on the message ports to convert them into activity on the design ports and internal signals. Typically, these are built using internally available IP.

Hardware emulators are a welcome tool for verification engineers who need to accelerate verification testing. Through the use of transaction-level modeling, emulation has been tightly integrated with the Verification Methodology Manual for SystemVerilog to create a high-performance functional-verification platform. This is possible through the VMM hardware-abstraction-layer application, which raises the level of abstraction for testbench developers. It also supports the interaction between a VMM testbench and emulation. Previously, performance gained with this approach was constrained by the speed mismatch between the software simulation of the testbench and the hardware-accelerated design under test.

---

Alain Raynaud, technical director of EVE of San Jose, Calif. and Palaiseau, France, has been involved with high technology for more than 15 years. Raynaud holds a master of science degree in computer science from the University of Illinois at Urbana-Champaign and from Supelec, France.