electronic design

print | close

Implementing Functional Coverage with Hardware Emulation

Electronic Design
Lauro Rizzatti
Tue, 2015-10-20 09:12

The huge undertaking of verifying a system-on-chip (SoC) design has challenged engineers for more than 20 years –– the amount of time spent on it hasn't varied much from between 50-70% of the entire development cycle. That's because time-to-market pressures relentlessly increase, as do design sizes. However, verification budgets don't increase at the same rate, forcing management to use available resources more efficiently.

Innovative electronic-design-automation (EDA) suppliers work to develop compelling tools and methodologies, and they often succeed. One example is the testbench, the environment that exercises a design-under-test (DUT) pre-tapeout when it's typically in a register-transfer-level (RTL) code. Writing directed tests requires knowledge of the DUT's functionality, a time-consuming job that gets more complex with increasing DUT size.

Related

Emulation Fast-Tracks Networking Products to Market

Hardware Emulation: A Weapon of Mass Verification

Lauro Rizzatti Explains Hardware Emulation's Appeal

Automatically generating random tests removes the burden of manually creating them, but introduces a level of unpredictability in the testbench. To make random testing useful, the level of testing achieved by the testbench must be evaluated and the random algorithm must be maneuvered toward untested design areas.

Requirements-Driven Verification

Simply put, coverage measures the percentage of the verification objectives met by a testbench. If the objective calls for checking the execution of the design code, the verification engineering team uses code coverage, which can be automatically tracked.

The verification team applies functional coverage if the aim is to determine that the design is operating correctly and meeting specifications. Measuring functional coverage cannot be inferred from the design and is the responsibility of the verification engineering team.

Code and functional coverage complement each other, and are used within the framework of a test or verification plan. The process of developing such a plan means extracting design requirements from the design specification, and strategizing how to ensure these requirements are met. This is known as *requirements-driven verification*, where verification requirements are planned and attached to the design requirements so that the plan can guide the verification process to completion.

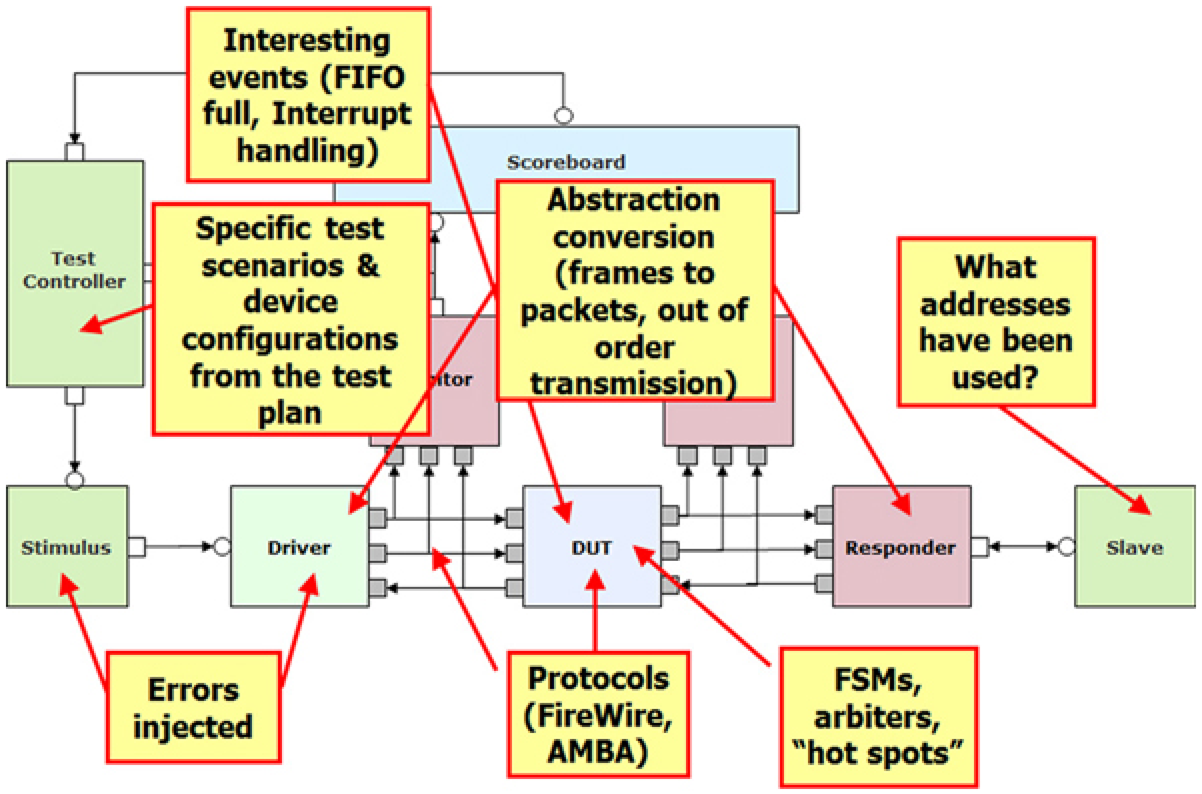
essential to plan in advance what to test and how to test it to manage the verification task. This is captured he verification plan. Three criteria drive the creation of the plan:

• What to cover

• How much data to cover

• When to sample

The first criterion is coded via SystemVerilog constructs, such as *cover*, *covergroup*, or *coverpoint*. Embedded into the testbench and the DUT, functional coverage code measures verification completeness against specifications, such as functional requirements, interface requirements, system specifications, and protocol specifications — both external and on-chip. Figure 1 shows examples of “what to cover.”


Interesting events (FIFO full, Interrupt handling)
Scoreboard
Test Controller
Specific test scenarios & device configurations from the test plan
Abstraction conversion (frames to packets, out of order transmission)
What addresses have been used?
itor
Stimulus
Driver
DUT
Responder
Slave
Errors injected
Protocols (FireWire, AMBA)
FSMs, arbiters, “hot spots”

The second criterion helps optimize coverage data. Collecting coverage data strains the verification engine, whether it’s a simulator, formal-verification software, or an emulator. As a result, it’s important to minimize the amount of collected data while maximizing its information contents. SystemVerilog *bins* contain this data, while *cross coverage* improves the quality of the data by value comparisons.

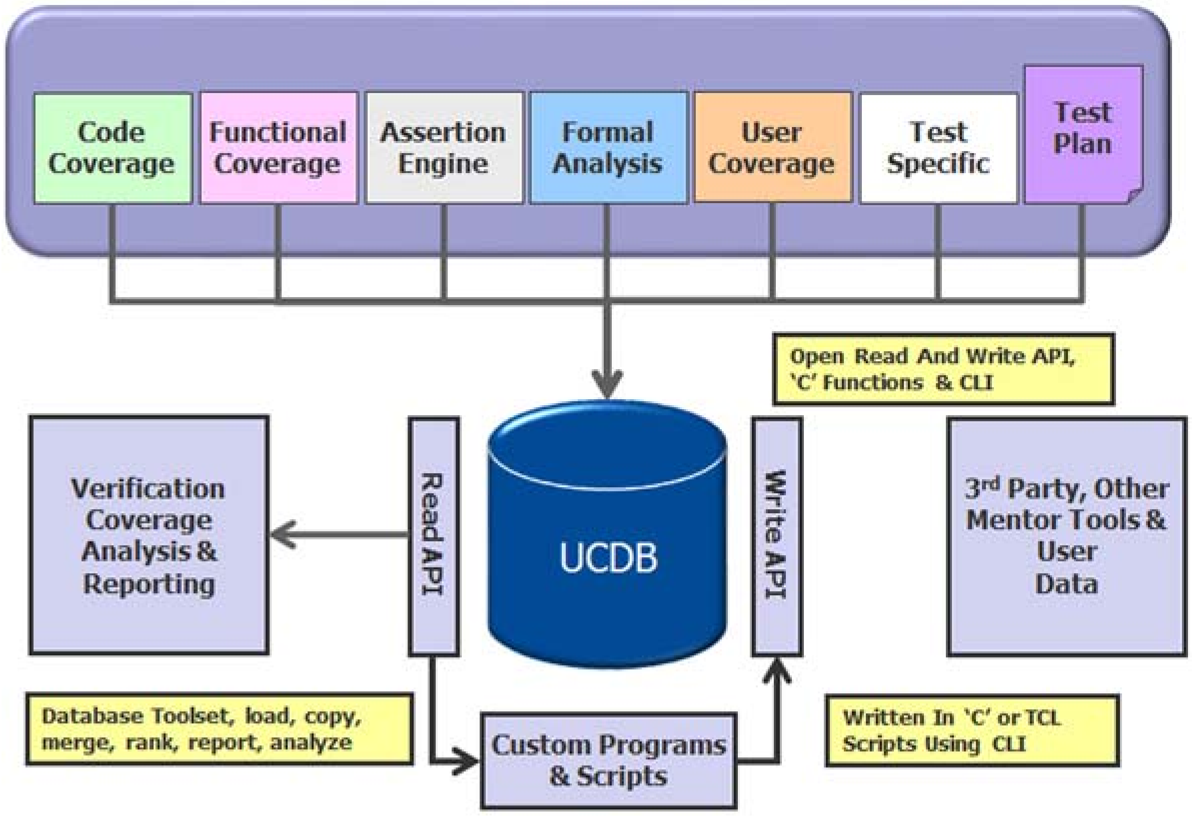
The third criterion perfects the second by recommending effective events to collect data. These include state transitions and transactions boundaries on clock edges only when appropriate (data changes far less frequently than clocks).

Unified Coverage Database (UCDB)

The Unified Coverage Database (UCDB), developed by Mentor Graphics, is a storage system to unify coverage data shared by all verification engines. Designed to be platform-independent and efficient, the UCDB infrastructure has been optimized for capacity and performance to enable quick storage and analysis of large coverage models.

ːrs can perform verification-coverage analysis and reporting through a graphical user or command-line ːrface that provides easy access and intuitive understanding. The database can import test or verification plans from external documents, like Word and Excel, to trace the progress of coverage closure based on the plan. Coverage is tracked for every test, confirming the analysis of the test's effectiveness. The UCDB architecture enables the engineer to control the coverage and implement exclusions when applicable.

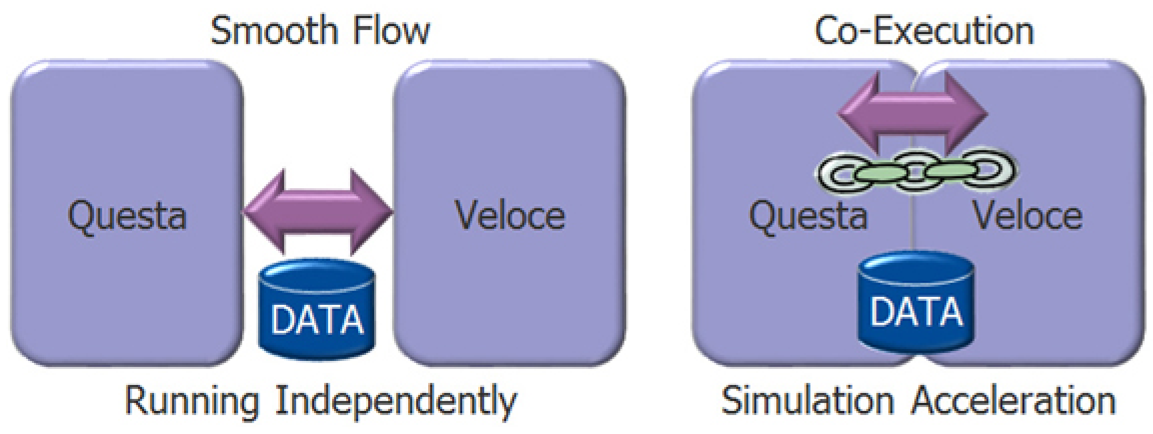
Through an open application programming interface (API), almost all verification tools can access the UCDB and write and read data into it, which lets UCDB acquire multiple metrics. Furthermore, different verification engines can read the UCDB and take actions aimed at improving overall verification coverage *(Fig. 2)*.


Code Coverage
Functional Coverage
Assertion Engine
Formal Analysis
User Coverage
Test Specific
Test Plan
Open Read And Write API, 'C' Functions & CLI
Verification Coverage Analysis & Reporting
Read API
UCDB
Write API
3rd Party, Other Mentor Tools & User Data
Database Toolset, load, copy, merge, rank, report, analyze
Custom Programs & Scripts
Written In 'C' or TCL Scripts Using CLI

The UCDB API was donated to Accellera's Unified Coverage Interoperability Standard (UCIS) technical working group, which accepted it as a basis for the development of standard coverage models and an API to access them. Version 1.0 of the UCIS API was released as a standard in 2012. Users can access Mentor's UCDB with either the UCDB API or the standard UCIS API.

From Simulation to Emulation

Let's turn to Mentor for an example of *simulation acceleration*. Its hardware description language (HDL) simulator and emulator can operate independently or in co-execution *(Fig. 3)*.



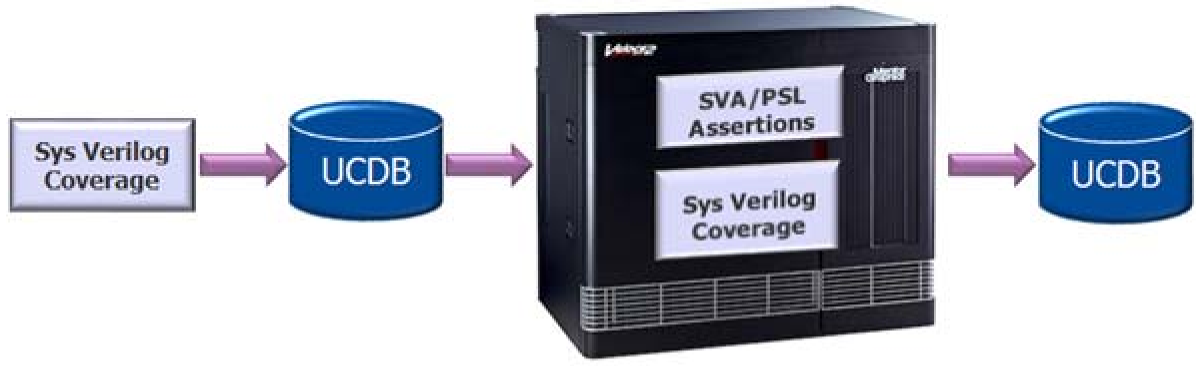
When run independently, the simulator can act as a pre-process for the emulator to secure smooth and fast bring-up of the DUT onto the emulator. This is possible because the emulator's components, which include memory models, accelerated verification intellectual property (VIP), Codelink loggers, and VirtuaLAB peripherals, run in the simulator. As for the stimulus, verification engineers needn't run the simulator to process the testbench. This can be done by using synthesizable testbenches or testbench acceleration with C or virtual devices.

One benefit of this mode is that the user can take advantage of the two verification engines –– the speed of emulation and the easy and fast debug of simulation. By saving and restoring the state of the DUT, the verification engineer can use emulation to quickly can get to the point where a bug is suspected to lurk, then switch to simulation for debugging.

Inside the emulator, the O/S software maps the same SystemVerilog functional coverage available in the simulator. The emulator supports the same SystemVerilog Assertions and Property Specification Language assert and cover directives available in the simulator.

Results are stored in the UCDB and shared between the verification engines. This means that the emulator's O/S synthesizes coverpoints and covergroups into the emulator. However, coverpoints in an emulator consume hardware that can reduce the number of designs run in an emulator or prevent a design from running at all.

Until now, without the knowledge of what was covered in simulation, verification engineers had to decide which coverpoints should be thrown away to preserve emulator capacity. The UCDB provides the infrastructure to integrate functional coverage across both simulation and emulation *(Fig. 4)*.



This shared database enables the emulator to know which coverpoints have been hit in simulation so that they don't need to be covered again. The emulator then targets only areas not already covered, increasing functional

erage in a highly effective and efficient manner, relieving verification engineers from having to make
leoffs between coverage and capacity.

When the simulator and emulator run in co-execution, testbench acceleration begins operating. It acts as a traffic cop that manages the communication between a more abstract testbench and the DUT mapped inside the emulator at speeds orders of magnitude faster than a lock-step communication between a Verilog testbench and the emulator. The differentiator is VIP. Assuring portability across simulation and emulation, the same transaction-based testbench can run in either of the verification engines.

Conclusion

By preserving capacity without sacrificing coverage, verification engineering teams get comprehensive functional verification with minimal incremental effort and without a hit on emulation capacity. Users get a high level of confidence that their SoCs are going to work at tapeout, and they achieve a much shorter time-to-coverage and much better chance of reaching their verification goals on schedule.

Source URL: http://electronicdesign.com/eda/implementing-functional-coverage-hardware-emulation