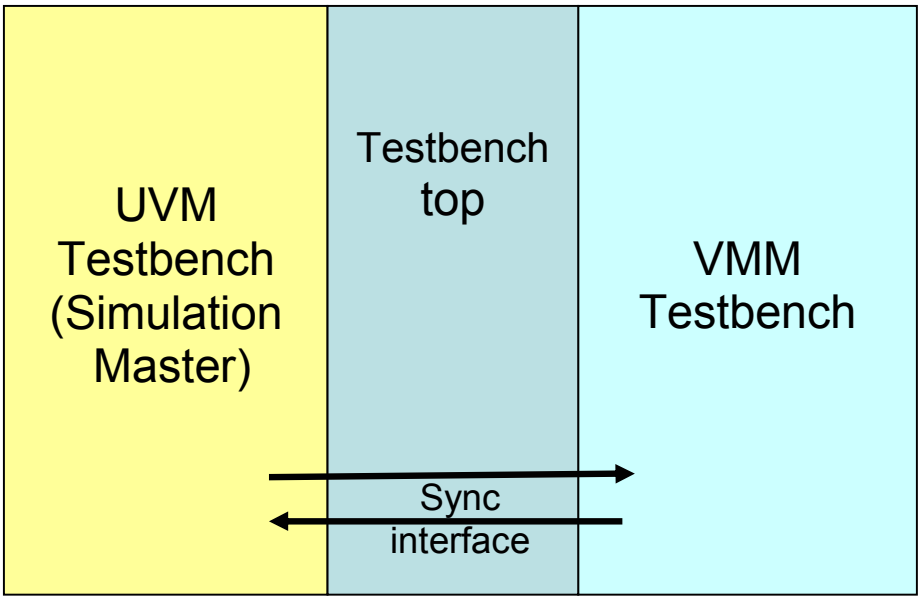
Integrating UVM and VMM

Bharath Kumar Yogeesh
Technology Architect
(bharathy@synapse-da.com)

In general in Design Verification, there is an increasing trend to reuse the legacy verification environments along with UVM adoption wherever feasible. This will help the teams to meet not only 'time to market' goals but also save verification effort significantly. We discuss how two different Verification environments were integrated and what techniques were used to resolve the issues during and after integration.

One of the verification environments is used to test Analog and Mixed signal design. It is a System Verilog 'program' based VMM verification component instantiated at the testbench top level. Here the verification Environment is created and instantiated in a 'program' and the program is in-turn instantiated in the testbench top. This facilitated the connection of 'virtual' interfaces to 'physical' interfaces since VMM config db, VMM_TEST were not used. Every test has the 'program' with a task overwriting a common task definition. Different transactors of the instantiated environment were accessed in every test. There were around 100 tests in this setup.



UVM-VMM Integrated Testbench

The other verification environment is a UVM based CortexM0-platform system level test bench. This uses the three main phases of UVM, UVM config db for 'set'/'get' interfaces and UVM_TEST is used to run each test. Module is used to set uvm config db which in-turn is useful for binding the interfaces from testbench-top. Cortex-sync is used for message passing and synchronizing 'program code' executed in Cortex-M0 and UVM.
The objectives for unified environment are

1. Facilitate interaction between two testbenches so that the dataflow/communication between different DUT blocks can be verified.
2. Two separate testbenches are difficult to maintain

Following is the method is a straight-forward approach for migrating, merging VMM testbench to UVM

1. Convert each program-based test (VMM) to sequence based test (port it to UVM)
2. Use Config db for set, get interfaces
3. Use UVM_TEST_NAME for running each test
4. Convert VMM specific phases, messages to equivalent of UVM.

However this may generate unknown issues in scoreboard and certain verification scenarios. Hence the following approach is taken:-

It was necessary to keep the VMM environment physically separate from UVM. This implies that there must be a synchronization scheme for passing messages between UVM and VMM. A physical interface is created at the test-bench top level to achieve the communication and synchronization between UVM and VMM. Separate classes vmm_sync, uvm_sync were added in UVM, VMM testbenches respectively. The report() phases are extended to pass the error counts from VMM to UVM.
Since UVM and VMM will be executing in parallel, UVM is made the master to resolve the simulation ownership (e.g. $finish) conflicts between UVM and VMM. POR (Power-on-reset) physical interface is controlled by VMM, hence all the UVM tests wait till the VMM is done with POR. This made the UVM tests to jump-start directly into UVM 'run' phase without worrying about the POR phase. After integration, to run a VMM test, it required to have separate compilation for each VMM test. This was accomplished using pre-compilation constructs (`ifdef) to select the particular VMM test. A single script (ucsim – reused from CortexM0 platform) is used to launch both UVM and VMM tests.
The main advantages of this scheme were

1. It relieved the Verification engineers from the intricacies of the UVM-VMM adapters and synchronization of UVM-VMM phases.
2. Simultaneous development of VMM and UVM testbench development is feasible.
3. It facilitated reuse the legacy testbench environments and test suite
4. Time and effort to taken to implement the scheme was relatively less compared to migrating VMM to UVM.