Melting Verification Pot: Integrating RVM/VMM and UVM, a Practical Guide and Lessons Learned.

Mark Azadpour

Introduction

- we share our experience with migration of a RVM testbench to SystemVerilog with a mixture of UVM and VMM methodologies
- Internally & Externally Developed IP's and VIP's
- Did not wanted to impact schedule and did not have control over the external VIP's.
- The DUT was recently ported to Verilog with Legacy mixed VHDL and Verilog.

Verification Env.

- Legacy RVM that was partially ported to VMM.
- Embedded system which required elaborate firmware programming to set limits via CSR as well as service IRQ and FIQ request as well as a sequencer that required special load to operate properly.
- Verification IP (VIP) for various IP's testbenches included RVM, VMM1.0, VMM1.1, VMM1.2 and UVM.
- With the goal of reuse, we intended to use as much of the VIP's as possible in our SoC environment.
- Our SoC testbench had to be able to instantiate all of the sub environments.

Choice of Top Level Methodology

- VMM on top with UVM sub-environments.
  - As a proof of concept we were able to develop a testcase with this configuration before diving completely into either methodology.
- UVM on top with VMM and others as the sub-components.
  - This alternative was selected since our ultimate goal was to port the whole test bench to UVM.
- In UVM, the top level object is the testcase
- In VMM it is the environment.

Migration Path

- Top Level UVM.
- Use Inter-operability library released for UVM1.0
  - Messaging between VMM and UVM
  - Phasing : If VMM1.2 was used, the phases were called automatically.
  - For explicit phasing, we wrapped the functionality in the “run” routine in a phase aware routine which was called automatically.
  - Those subenv. with no phasing in RVM, were extended to include phases added and called the appropriate routines.

Migration Path to UVM

- import uvm_pkg::*;
- import vmm_std_lib::*;
- import uvi_interop_pkg::*;
- `include "vmm_ral.sv" .
- Register Access Language (RAL) from both VMM and UVM were used in parallel with device memory map divided between the two.
  - We were able to call VMM RAL routines from the UVM side with no problems.

Instantiation Differences

- We used intermingled VMM and UVM code.
  - In VMM on new, handles are passed that “connects” the entity with other Config. And instances. Legacy code from RVM days.
  - In UVM, phases are utilized.
  - So the UVM objects did not necessarily know the status of VMM objects. This caused synchronization issues.
    - Solution: use the Universal DB as a registry mechanism so the UVM side knew about the VMM objects and if necessary instantiate the entities. This worked beautifully as it separate the producers and consumers.

UVM Migration Path

- We developed a number of UVM transactors which utilized RVM BFM's to drive the data into the DUT.
- These BFM were called via an openVera class that was extended in the UVM side. Therefore, we could call the base function that was implemented in Vera through an openVera wrapper.

UVM Migration Path

- We used TLM ports to pass transactions from UVM to VMM and back.
- This worked beautifully as both Methodology support TLM constructs.

Conclusion

- If you cannot afford the time and resources to convert you testbench to UVM in one-shot, consider migration in layers that allows you to move to UVM now and start enjoying the benefits of the methodology for any newly developed code while supporting the legacy sub-environments.
- The Synopsys Interoperability library can be used to ease the transition pain. Although it provides most of capabilities, there is a large amount of manual intervention is required so planning a strategy is key to success.