Taming a Complex UVM Environment

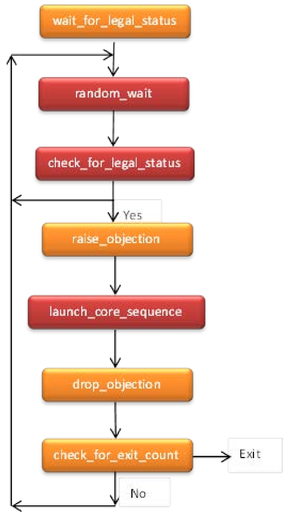
Manjunath Shetty, Ramamurthy Gorti
Broadcom Corporation

Introduction

Setting up an effective UVM-based verification environment can be challenging due to various reasons. This paper discusses an approach that can be used to improve controllability and scalability of tests while maximizing the effect of constrained randomness and reusability in any complex UVM-based verification environment. This paper also discusses the advantages and challenges of using this approach in the verification of an xHCI-based USB 3.0 host controller

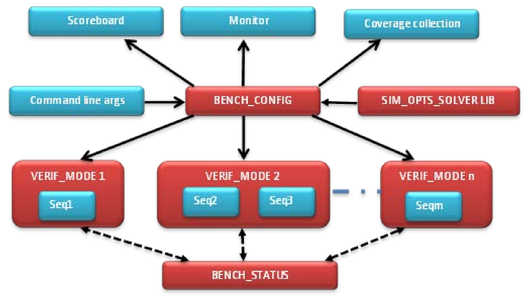
The proposed approach is to have multiple runtime switches, termed as “verif_modes”, corresponding to every design feature. Each verif_mode enables a sequence or a set of sequences to create the scenarios required to verify the design feature and also optionally enables checks related to the feature. The verification environment uses verif_modes to hierarchichally manage the real work sequences.The verif_modes can be enabled to test directed scenarios or concurrently with other verif_modes to create interleaved constrained random scenarios

General VERIF_MODE Flow Diagram



The core_sequence, which exercises the minimal iteration of the identified scenario, needs to be implemented. The verif_mode sequence following the above generic flow diagram needs to be implemented. Using bench_status from the mointor, every verif_mode sequence waits for some legal state during which it can exercise the scenario. The sequence waits for a random time that helps in creating corner cases. The sequence then again checks for a legal condition for exercising the verif_mode. The verif_mode sequence then launches the core_sequence. The verif_mode sequence could, optionally, raise/drop an objection before/after running the core_sequence. After running the core_sequence, the verif_mode sequence could either repeat the iteration depending on the intended number of times the verif_mode needs to be exercised in the test.

Verification Environment Block Diagram using VERIF_MODES



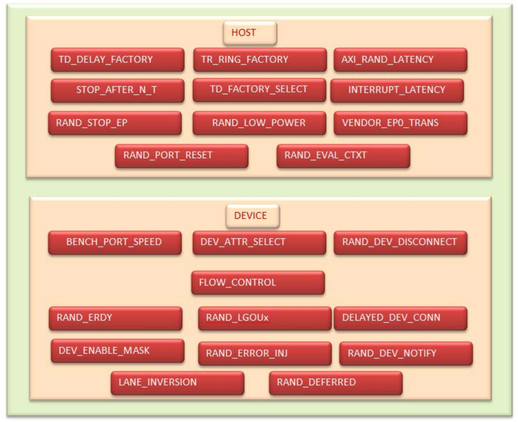
A singleton class called “bench_config” is used to control verif_modes. The bench_config class provides a centralized way to control all of the simulation options and their usage throughout the environment. Each verif_mode in bench_config class can be configured using command line options or using “sim_opts_solver” class. The uvm_cmdline_processor class can be used to set each verif_mode to desired values from simulation options provided by the user. For an automated way of controlling the verif_modes, the “sim_opts_solver” class is used. The sim_opts_solver class consists of constraints to configure all the verif_modes in different combinations to create interesting legal corner cases and to take care of any spatial and/or temporal dependencies. One can have a library of such sim_opts_solvers targeting single/multiple features in the design while enabling or disabling other background verif_modes. The sim_opts_solver also provides a way to deal with mutually exclusive and dependent verif_modes

Any specific verif_mode could also be set from simulation options using uvm_cmdline_processor. The verif_mode configured using command line option takes precedence over randomized value from sim_opts_solver class. This also provides the option of disabling scenarios when needed, either due to a known design issue or when a verif_mode sequence itself has issues

Above figure also shows the usage of the “bench_status” class. This is a data structure maintained in monitor and is used by all verif_mode sequences to check and update the status of the bench and DUT at any point in time during the run_phase of the test. Each verif_mode uses this during check_for_legal_status and also updates it according the verif_mode during the running of core_sequence

Monitor can optionally maintain a data structure for each verif_mode indicating the number of times the expected scenario was hit. This can be used during the check_phase() of the test to make sure all the intended scenarios were hit during the test, at least once. It can also be used during the report_phase() to print statistics of various verif_modes enabled in the test and the number of times each scenario was hit during the test

xHCI-based USB 3.0 host controller



Template use cases of VERIF_MODES:

General verif_modes: These are the verif_modes that are needed for all tests. Most of these are used to pass a particular runtime argument to the bench.

- Controlling the speed: BENCH_PORT_SPEED and RANDOM_SPEED_SELECT_INCLUDE
- Controlling the device model configuration: DEVICE_ATTRIBUTE_LIBRARY_SELECT
- Controlling the quantity of traffic: DMVC_STOP_AFTER_N_TDS_PER_EP.
- Controlling the type of traffic: TD_FACTORY, TD_FACTORY_LIBRARY_SELECT, ENQUEUE_TD_DELAY_FACTORY and TR_RING_FACTORY_RANDOM_SELECT
- Controlling external latencies: INT_LATENCY_RANDOM_SELECT and AXI_SLAVE_CFG_RANDOM_SELECT

Special features verif_modes: These are the verif_modes identified from the specification targeting a specific scenario and needs a sequence implemented as per Figure 1.

- Device disconnect: NUM_DISCONNECT_CONNECT.
- Port Reset: NUM_PORT_RESET and PORT_RESET_TYPE
- Stopping the endpoints: ISSUE_RAND_STOP_EP_CMD.
- Flow control on the endpoints: FLOW_CONTROL_BUFFER.
- Low power modes: ENABLE_FAST_U1_ENTRY and ENABLE_FAST_U2_ENTRY.

.

Background activity verif_modes: These verif_modes are identified from the specification to create random activities in the test to help create legal corner-case scenarios. These are mostly light on sequence implementation and exercise simple design features, but when enabled together, are powerful in creating corner cases.

- Device issuing ERDYs: ENABLE_RAND_ERDY and ISSUE_MULTI_DUMMY_ERDY
- Device issuing LGO_Ux:. RAND_DEVICE_TX_LGOU1 and RAND_DEVICE_TX_LGOU2.
- Delayed device connection: RAND_DELAYED_DEVICE_CONNECT .
- Ports with no device connected: DEVICE_ENABLE_MASK.
- Vendor Endpoint0 transfers: ENABLE_BACKGROUND_EP0_TRAFFIC
- Control transfers with address=0:. GET_DESC_BEFORE_ADDR_DEV.
- Error injection: ENABLE_RANDOM_ERROR_INJECTION.
- LMP packets:. RAND_PORTPM_WRITE_BACK
- Device notification packets:. ISSUE_RAND_DEV_NOTIFY
- Evaluate context command:. ISSUE_RAND_EVALUATE_CONTEXT
- Lane Inversion: INVERT_LANE_POLARITY
- Deferred packets: ENABLE_DEFERRED_MODE

Conclusion

The proposed approach has been very effective in verifying an xHCI-based Host Controller. It facilitated modular verification of incremental design/test-bench enhancements such as Streams protocol, Broadcom PHY, and USB2.0 support. This approach has proven to be of very high quality and has managed to catch over 300 RTL and corner cases, many of which were obscure and would not easily have been hit with a simpler test bench

Contact information

Manjunath Shetty
Sr Staff Engineer
shetty@broadcom.com

Ramamurthy Gorti
Sr Manager
rgorti@broadcom.com