Whitebox Approach for Verifying PCIe Link Training and Status State Machine

Colm McSweeney
Joe McCann
Pinal Patel
Gaurang Chitroda

Synopsys Inc.,
Dublin, Ireland
www.synopsys.com

eInfochips,
Sunnyvale, USA
www.einfochips.com

ABSTRACT

Serial communication protocols like PCI Express and USB have evolved to enable high operating speeds. This evolution has resulted in their PHY Layer protocol growing in complexity, especially the Link Training and Status State Machines' (LTSSM's) logic. The top-level DUT behaviour does not reveal if the LTSSM functionality was correct and whether it hit the expected state transitions properly.

A traditional testbench often concentrates on higher level functionality of LTSSM such as achieving a working link, link speed and width updates, among other things. This paper talks about a new approach that reduces verification time and verifies the micro-level details of LTSSM functionality. Using this approach, we found more than 60 LTSSM RTL bugs in the DUT. Also, number of tests required were reduced to around 50 as compared to 500 tests in a legacy testbench. This approach is not just limited to the LTSSM, but can be re-purposed to verify any other complex state machine.

Table of Contents

Table of Figures

1. Introduction

The serial protocols like PCI Express and USB have evolved over the years to provide very high operating speeds and throughput. This evolution has resulted in their physical layer protocol becoming very complex. One of the most essential processes at physical layer is link initialization and training process. In the PCI Express devices, this process establishes many important tasks such as link width negotiation, link data rate negotiation, bit lock per lane, symbol lock/block alignment per lane, etc. All these functions are accomplished by Link Training & Status State Machine (LTSSM), which observes the stimulus from remote link partner as well as the current state of the link, and responds accordingly.

The PCI Express link training state machine has many states, which are further classified into multiple sub-states. Each LTSSM sub-state performs a set of well-defined operations and makes a next state transitions based on meeting required conditions. Even after the link is up, the device may need to change these working parameters of the link at runtime. Sometimes, the device may need to re-establish the link or it can be directed to go into a low-power state.

However, it is not always straightforward to take the LTSSM through the required state transitions by controlling or manipulating its inputs and predict its behaviour. We use a PCI Express LTSSM whitebox reference model, which is a part of the bigger UVM-based testbench environment. The LTSSM reference model observes the same physical layer traffic as the DUT, behaves as per the PCI Express Base Specification and also predicts the possible state transitions. As opposed to the Black Box tetsbench which has no idea about the state of DUT's internal blocks, this model is aware of DUT's LTSSM state and values of useful LTSSM parameters. This is done by monitoring DUT's internal signals from the LTSSM design block. As this model probes inside the DUT and is well-aware of the state of the DUT LTSSM all the time, we call it an "LTSSM Whitebox Model".

The PCI Express defines the state behaviour and and relevant state transitions so that there can be multiple state transitions conditions to transition to the same next state. For some of the sub-states, there are multiple state transition paths that lead to different next states. To trigger all the required state transitions and transition conditions, we use a mixture of directed and constrained random stimulus generation. As each and every statement in the PCIe Base Specification description of LTSSM requires attention, we create a detailed coverage for all sub-states that includes all state transition paths, transition reasons/conditions, transmit rules, stimulus of interest, etc. As our DUT is a configurable IP, we use a configurable HVP and Verification planner as the final sign-off for all checks and coverage.

2. LTSSM Testbench Architecture

The LTSSM whitebox testbench is a sub-environment of a bigger top level UVM compliant layered testbench. As we are verifying the MAC part of the physical layer, we use PIPE-2-PIPE connection to connect two PCIe devices. The device facing the DUT device can either be the VIP or another DUT. We monitor the stimulus from the remote link partner over the PIPE interface.

Note: In the following sections, we will refer to whitebox as an acronym WB.

The following diagram shows the basic block diagram of the LTSSM testbench:

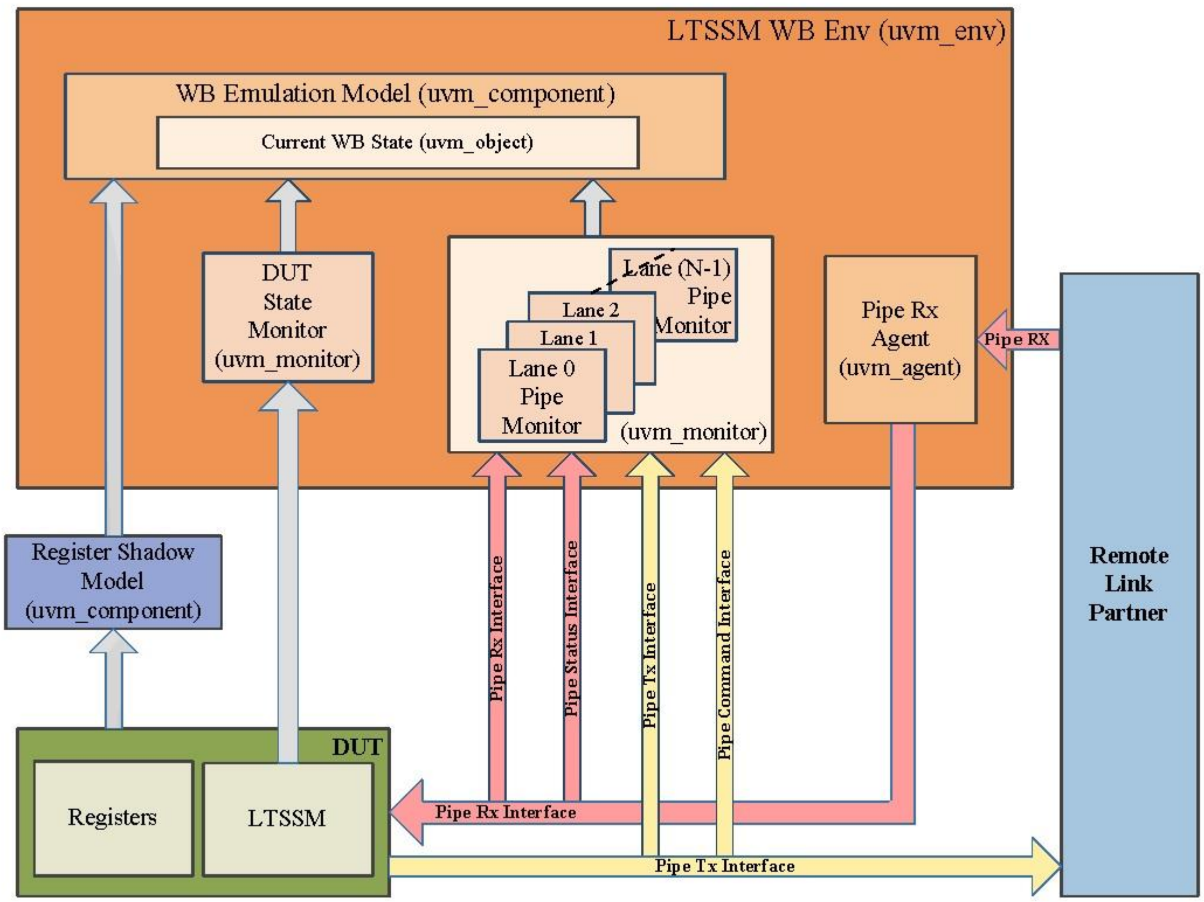


Figure 1Basic Block Diagram for LTSSM WB Testbench

There are 4 main components of the testbench:

1. PIPE Monitor (per lane)
2. DUT State Monitor
3. LTSSM Emulation & Checking Model
4. Pipe Rx Agent

1. PIPE Monitor:

The PIPE interface signal widths are scaled according to number of lanes. We take a PIPE Monitor instance per lane, which would observe the physical layer packets or OSs(Ordered Sets) and Command/Status information for a particular lane. These packets are sent to the LTSSM WB Emulation model, which in turn responds to the observed stimulus. It contains the following types of analysis ports:

- uvm_analysis_port#(os_item) for both transmitted and received ordered sets
- uvm_analysis_port#(pipe_status_if) for received PIPE status updates from the PHY
- uvm_analysis_port#(pipe_cmd_if) for PIPE commands sent from the core to the PHY
- uvm_analysis_port#(pipe_eq_event) for equalization PIPE events to/from the core

2. DUT State Monitor:

The DUT state monitor tracks the current LTSSM state of the DUT by observing its internal signal and sends this information to the model.

3. LTSSM Emulation & Checking Model:

This model is the heart of the LTSSM Verification. It emulates every sub-state as mentioned by PCIe Base Specification. This model consists of various components like,

- the state timer that increments on required time scale and implements any possible timeout transitions
- separate Tx and Rx queues of type os_item to store transmitted and received Ordered Sets(Physical Layer packets)
- counters to keep track of any received/transmitted Ordered Sets
- methods that keep looking for possible state transitions whenever any of the following occurs:
  - ❖ a new OS is received/transmitted, which results in required condition (all/any lane conditions) getting satisfied.
  - ❖ some register field(s) update is observed.
  - ❖ some timeout condition occurs.
- checkers to keep track of unexpected state transitions, non-transition threshold, etc. Some examples are:
  - ❖ number of DUT transitions seen while WB did not move ahead exceeded allowed threshold.
  - ❖ number of ordered sets sent or received after DUT state transition (while WB is still unmoved) exceed the allowed threshold value.
  - ❖ WB moving to unexpected state transition as compared to the DUT transition.

The LTSSM emulation model makes a state transition only when all of the following conditions are satisfied:

(i) DUT state transition has occurred,
(ii) WB state transition condition has occurred,
(iii) Both the WB and DUT are moving to the same next state

4. PIPE Rx Agent:

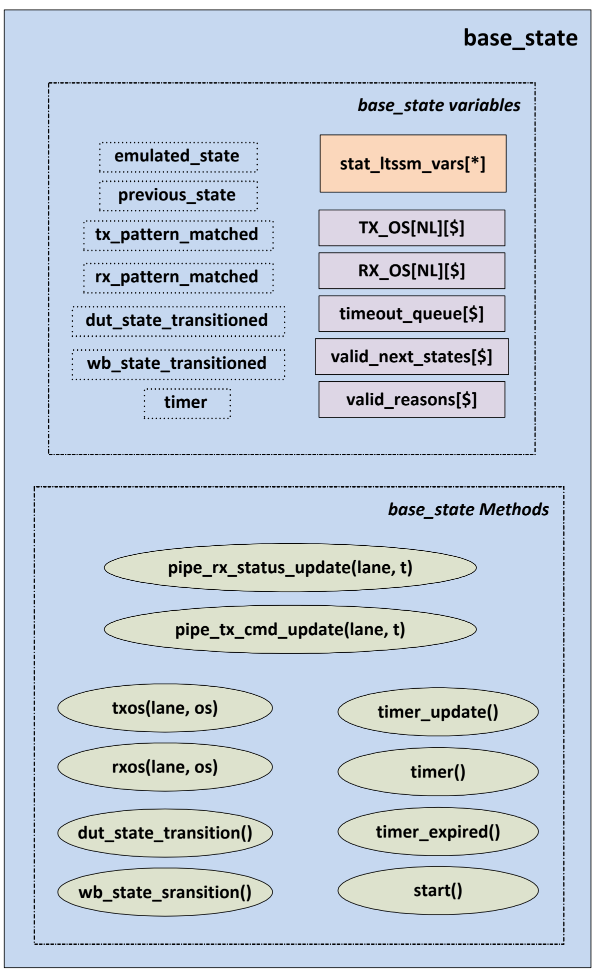
The Pipe Rx Agent is used for injecting erroneous LTSSM stimulus into the DUT. It consists of 3 components: pipe_rx_agent_monitor, pipe_rx_agent_queue and pipe_rx_agent_driver. By default, this agent works in a passive mode, which means it does not touch the data on the PIPE. While working in active mode, pipe_rx_agent_monitor monitors the PIPE data and stores monitored Ordered Sets within pipe_rx_agent_queue. Then the callback is used to inject errors in these stored OSs. The pipe_rx_agent_driver drives them back on the DUT PIPE interface.

3. How does the LTSSM WB Emulation Model work?

The LTSSM WB emulation model is a state-machine component called ltssm_wb_comp, which is extended from uvm_component. The ltssm_wb_comp consists of an instance current_wb_state of a base class base_state. The instance of current_wb_state indicates the current state that is being emulated currently.

The start() method of ltssm_wb_comp has a free-running logic in forever that waits for both WB and DUT to transition and once it has seen both the WB and DUT states transition, it would change the emulated state to new WB state. However, there are multiple checks performed before the WB emulated state is changed. All these checks are part of ltssm_base_state class, which is the base class used for modelling all the LTSSM states.

The ltssm_base_state consists of many variables and methods that are employed to mimic the state functionality as per the specification. The most important variables and methods of ltssm_base_state are shown in the following block diagram:



emulated_state – The emulated WB ltssm state

previous_state – Previous state from which WB entered currently emulated state

tx_pattern_matched – Flag to indicate that Tx pattern of required Oss has been matched

rx_pattern_matched – Flag to indicate that Tx pattern of required Oss has been matched

dut_state_transitioned – Flag when 1 indicates that DUT state has changed already

wb_state_transitioned – Flag when 1 indicates that WB has hit the transition condition

timer – Integer that stores the timer value

TX_OS – Queue to store transmitted Ordered Sets per lane basis

RX_OS – Queue to store received Ordered Sets per lane basis

timeout_queue – Queue to store all possible timeouts for the emulated state

valid_next_states – Queue to store all possible valid state transitions for the emulated state

valid_reasons – Queue to store all possible valid state transition reasons for the emulated state

start() – This method configures the emulated state by populating valid_next_states[$], valid_reasons[$], timeout_queue[$].

timer() - Increments timer after every 1us

timer_update() - Called every time after timer is incremented

timer_expired() – Whenever any of the timeout is met, this method gets called, and optionally makes WB transition

txos() / rxos() – These methods are called whenever received/transmitted ordered set is seen by WB component from the pipe monitor.

***_state_transition()** – Called whenever WB or DUT has made state transitioned. These methods call some important checks for the validity of transitions.

Figure 2 base_state class and its important member used in state emulation

The base_state class consists of an instance of a static class **stat_ltssm_vars**, which is used to store the values of the LTSSM variables that are used throughout all the states. The WB model updates, resets and reads these variables from static class as and when required in an emulated state. Apart from this, base_state also implements multiple checks on WB and DUT state transitions. Every emulated state inherits the following checks from the base_state class:

(i) If the DUT transition occurs after WB transition, the WB method dut_state_transition() first checks if the new DUT state is a valid transition for currently emulated state. If not, the emulated state shouts an error.

(ii) If DUT state transition occurs before WB transition, the new DUT state is stored into a static queue. Now, there are two possibilities:

1. WB state transition occurs after some time, in which case the DUT state previously stored in a queue is popped out and we check if the WB transition matches the DUT transition.
2. Another DUT transition occurs after some time, in which the new DUT state is stored again in the queue. This way we allow maximum of 2 DUT state transitions till WB makes the required transition. However, if DUT makes third state transition before WB transition, we shout an error.

(iii) If WB takes longer time to transit after DUT has already moved, there is a possibility that the new DUT state may transmit/receive ordered sets as per the requirement of new state. We have a threshold of 3 Tx/Rx ordered sets after DUT state transition. If WB sees four or more Tx/Rx OS after DUT state transition, then it shouts an error. We can also change the value of threshold using plusarg.

The following flowchart explains how the current emulated state implements its check based on new Ordered set sent/received, DUT state transition or WB state transition:

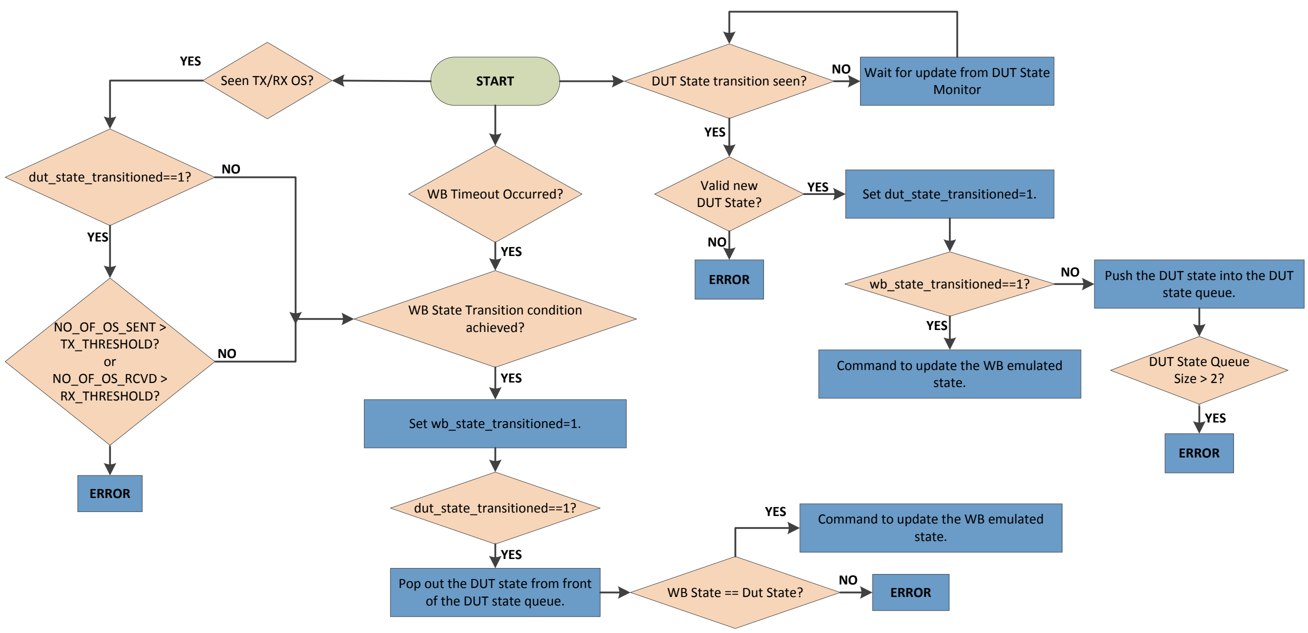


Figure 3 Flow-chart for WB Emulated State Checkers

The model uses SV polymorphism for modelling state machine. Whenever both DUT and WB make a valid and matching state transition, the ltssm_wb_comp overrides the base_state handle current_wb_state by the object, which is of type next state to be emulated. The following pseudo code explains this:

```
task start();
  forever
  begin
    // wait until both model and DUT has made a transition
    wait(current_wb_state.wb_state_transitioned==1 &&
        current_wb_state.dut_state_transitioned==1);
    next_state=current_wb_state.get_exp_state();
    prev_state=current_wb_state.get_state();
    `uvm_info(m_id, $psprintf("Changing from %0s to %0s",
              current_wb_state.get_state(), next_state), UVM_MEDIUM)
    l_string_name=get_state_class_name(next_state);
    if(! $cast(current_wb_state , l_factory.create_object_by_name( l_string_name  ) ))
      `uvm_fatal(m_id, $psprintf(" Failed to cast %0s ", l_string_name ))
    // configure the state, start it and add the callback.
    current_wb_state.configure_state(device_index);
    current_wb_state.set_srevious_state(prev_state);
    current_wb_state.start();
  end // forever
endtask : start
```

The active component of the WB emulation model is ltssm_wb_comp, which consists of instance current_wb_state of base_state to represent currently emulated state of the LTSSM. ltssm_wb_comp gets different updates like PIPE interface status and command updates, transmitted and received ordered sets, DUT register updates, DUT state transition updates, global reset updates, etc. It receives all these updates through the following analysis ports and implementation ports:

- uvm_analysis_imp_dut_state_analysis#(device_state, ltssm_wb_comp) dut_state_analysis_imp
- uvm_analysis_imp_dut_reset_analysis#(reset_item, ltssm_wb_comp) dut_reset_analysis_imp
- uvm_analysis_port #(base_state) ltssm_state_analysis_port
- uvm_analysis_port #(shadow_update, ltssm_wb_comp) register_shadow_analysis_imp

Apart from this, the ltssm_wb_comp receives Tx/Rx ordered sets monitored by the PIPE Monitor. However, the LTSSM WB environment has PIPE monitor instances per lane. To handle this, ltssm_wb_comp uses 2 arrays (sized number of lanes) of os_subscriber class, one for Tx and the other for Rx. os_subscriber is extended from uvm_subscriber class and it implements a write method where it calls the txos() or rxos() method of a currently emulated state instance based on whether the OS is coming from the Tx subscriber or the Rx subscriber. This way, the monitored Ordered Sets can be pushed into the emulated state's queues so that it can look for the possible state transition conditions. Similar approach is used for sending monitored PIPE command and status interface transactions to the emulated state current_wb_state. This is shown graphically in the following diagram:

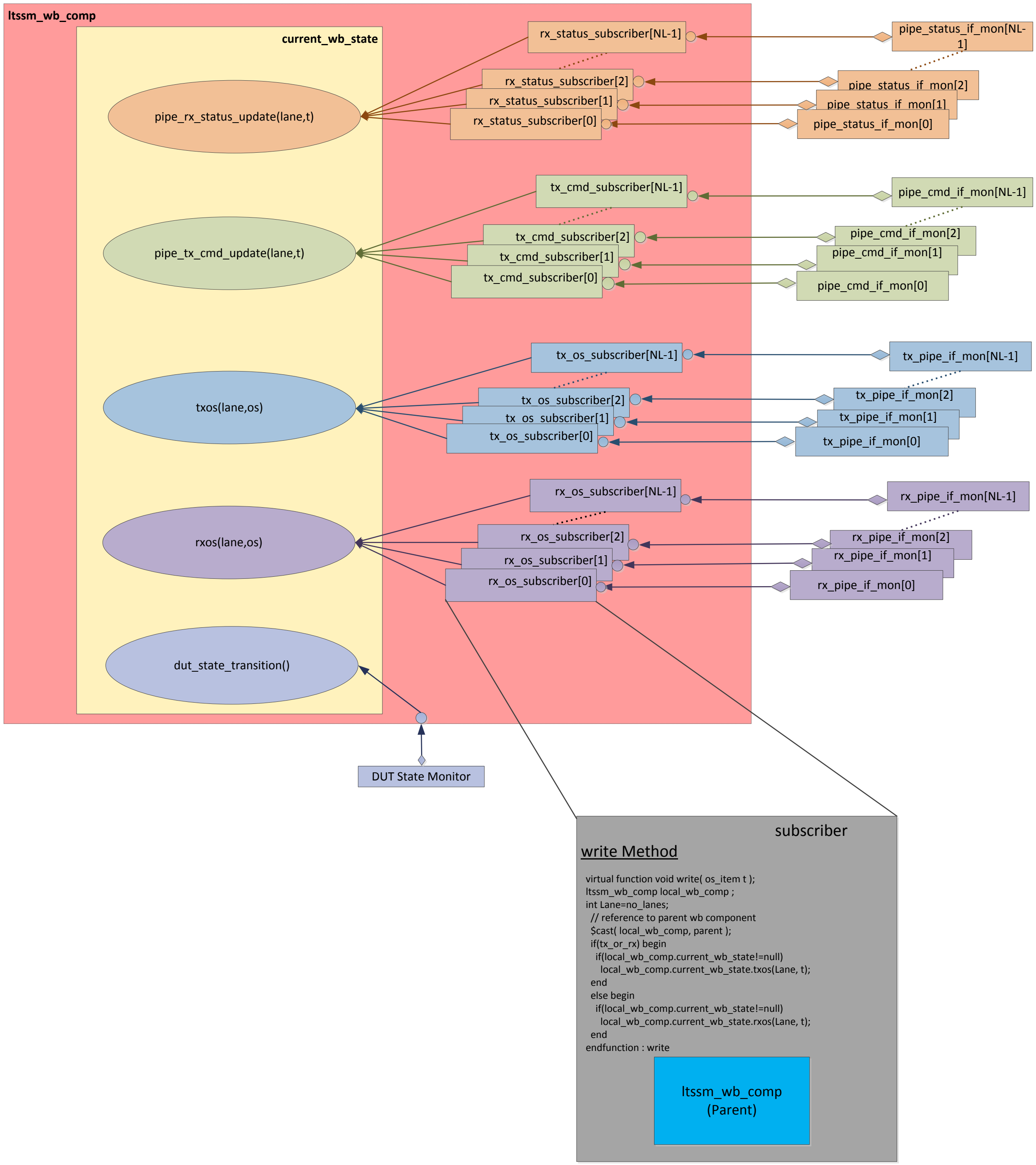


Figure 4 PIPE Monitor to LTSSM WB Component Connection

4. Rx PIPE Agent & Stimulus Generation

Normally, the link training process is self-initialized by the two devices connected over the serial PCIe link. During the normal "link-up" operation, the LTSSM would only go through a very few state transitions that are required for graceful linkup with link width, link speed and bit-lock symbol-lock/block-alignment. To fully test the LTSSM, we need to generate stimulus that stimulates the DUT with different random corner case scenarios, including negative stimulus. It is very difficult to control the stimulus from the remote device in such a way that we can hit all the scenarios. In the past, VIPs have been used to provide this stimulus. But, this has proved time consuming and often creates dependencies that tie us to another project.

To generate the stimulus of our interest, we need a means to control or manipulate the ordered sets that a device is receiving. To avoid hitting the "normal working conditions" of any LTSSM sub-state, we need to corrupt the otherwise good PIPE interface traffic before it is received by the device. The pipe_rx_agent is a component that provides errors injection on the Rx PIPE interface. The 3 steps for corrupting the stimulus that is being received over the PIPE interface:

1. Monitor "un-touched" ordered sets on the PIPE interface of remote link partner
2. Inject required errors in the monitored ordered sets
3. Drive the corrupted ordered set (negative stimulus) on the DUT PIPE interface

The pipe_rx_agent is situated between the PHY and the MAC and uses the phy_mac_if virtual interface. The callback approach is used for injecting errors on the Rx interface traffic along with some API functions. The following block diagram explains this:

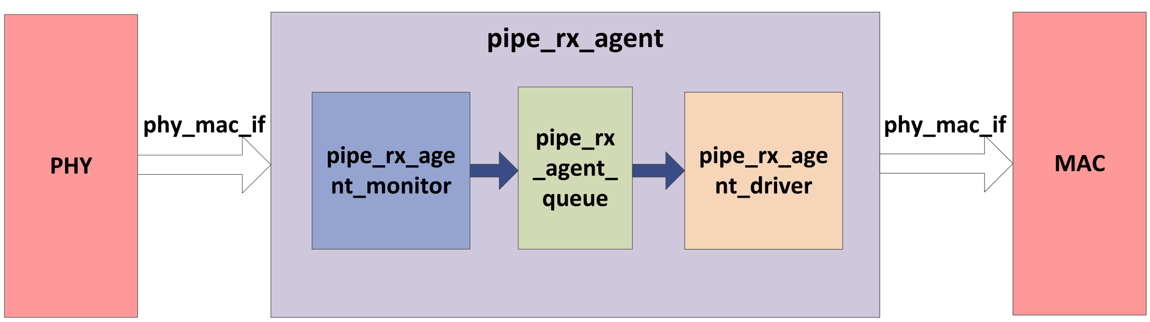


Figure 5 PIPE Rx Agent Block Diagram

The PIPE Rx Agent consists of a monitor, storage queue and driver. During the normal mode of operation without any error injection, the pipe_rx_agent_monitor monitors the traffic on the phy_mac_if and creates OS packets of type os_item, which are then pushed into the fifo and driven back to the phy_mac_if connected to the DUT. Whenever the callback is implemented and added along with supporting APIs, the pipe_rx_agent injects errors in the ordered sets taken out from the FIFO before driving them back on the phy_mac_if.

There are many possible error injection scenarios that we would be interested in while verifying the LTSSM sub-state functionalities. Some of the examples are,

- Corrupting link and lane numbers of the TS Oss being received in the Configuration sub-states

- Corrupting data rate identifier of the TS ordered sets
- Corrupting training control bits like Hot Reset, Disabled, Loopback, etc.
- “Missing COM” error injection
- Changing OS type from TS1 to TS2 and vice versa
- Corrupting sync header at Gen3 rate, etc.

Whenever we intend to verify any sub-state by injecting errors in stimulus, we need to implement the callback for pipe_rx_agent. A test will register the implemented callback into the pipe_rx_agent and these callbacks will be used to modify the OS data, sync header, PIPE command and status data, etc. Also, we would implement the APIs that are used to inject errors in pipe_rx_valid, pipe_rx_phy_status, etc.

The pipe_rx_agent callback error injection is random and is controlled through randomization. For this, we use a class pipe_rx_agent_control extended from uvm_object. This class consists of random members that control the following:

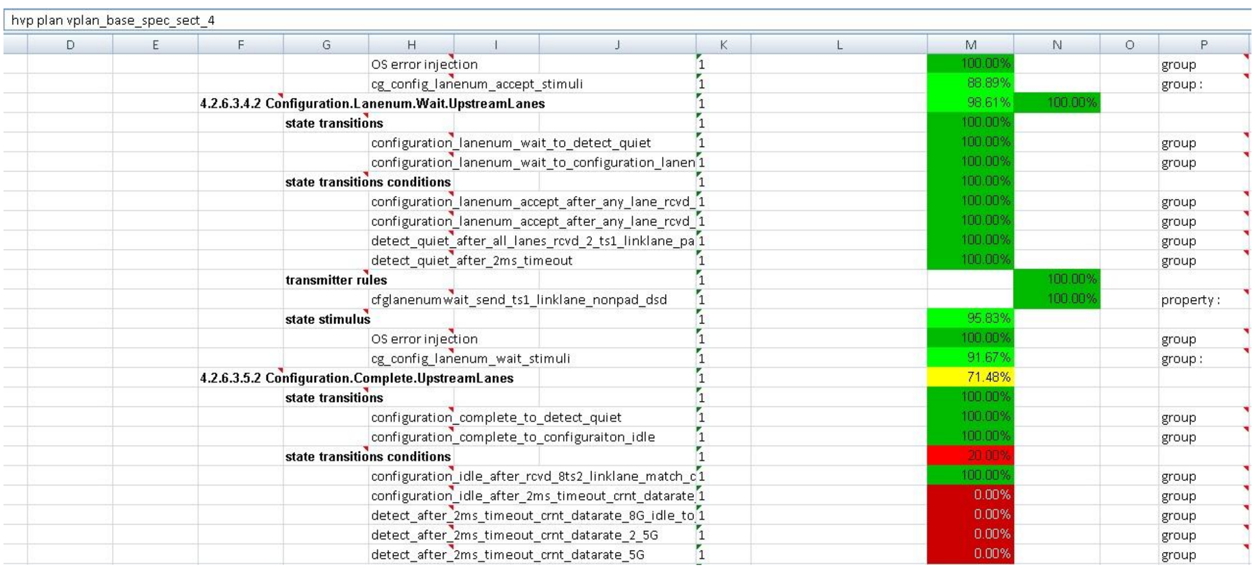
- Distribution of erroneous and good Ordered Sets
- Types of errors to be injected
- Control of whether to inject errors on ‘all lanes’ or ‘any lane’
- Error Persistence for injecting errors continuously

5. Coverage & Debug Features

For each of the LTSSM sub-states, we create a verification plan, which consists of the following coverage items:

- All possible state transitions for the sub-state
- All possible state transition conditions/reasons for the sub-state
- Transmit rules for the sub-state
- Stimulus coverage for the sub-state

The coverage plan for state transitions and state transition conditions is extracted using a script from the emulated WB state. This scripts reads the queues valid_next_states[$] and valid_reasons[$] mentioned within the emulated WB state and auto-generates the coverage plan . However, the corresponding coverage bins are hand-written. The following is a screenshot of final annotated coverage report that includes all the above mentioned items for the states:



hvp plan vplan_base_spec_sect_4

D	E	F	G	H	I	J	K	L	M	N	O	P
				OS error injection			1		100.00%			group
				cg_config_lanenum_accept_stimuli			1		88.89%			group :
		4.2.6.3.4.2 Configuration.Lanenum.Wait.UpstreamLanes					1		98.61%	100.00%		
			state transitions				1		100.00%			
				configuration_lanenum_wait_to_detect_quiet			1		100.00%			group
				configuration_lanenum_wait_to_configuration_lanen			1		100.00%			group
			state transitions conditions				1		100.00%			
				configuration_lanenum_accept_after_any_lane_rcvd_			1		100.00%			group
				configuration_lanenum_accept_after_any_lane_rcvd_			1		100.00%			group
				detect_quiet_after_all_lanes_rcvd_2_ts1_linklane_pa			1		100.00%			group
				detect_quiet_after_2ms_timeout			1		100.00%			group
			transmitter rules				1			100.00%		
				cfglanenumwait_send_ts1_linklane_nonpad_dsd			1			100.00%		property :
			state stimulus				1		95.83%			
				OS error injection			1		100.00%			group
				cg_config_lanenum_wait_stimuli			1		91.67%			group :
		4.2.6.3.5.2 Configuration.Complete.UpstreamLanes					1		71.48%			
			state transitions				1		100.00%			
				configuration_complete_to_detect_quiet			1		100.00%			group
				configuration_complete_to_configuraiton_idle			1		100.00%			group
			state transitions conditions				1		20.00%			
				configuration_idle_after_rcvd_8ts2_linklane_match_c			1		100.00%			group
				configuration_idle_after_2ms_timeout_crnt_datarate			1		0.00%			group
				detect_after_2ms_timeout_crnt_datarate_8G_idle_to			1		0.00%			group
				detect_after_2ms_timeout_crnt_datarate_2_5G			1		0.00%			group
				detect_after_2ms_timeout_crnt_datarate_5G			1		0.00%			group

Figure 6 Annotated XVP Report Example

As Synopsys treats each configuration of the IP separately, we require that the Verification plan must be configurable as the RTL itself. Such a configurable verification plan can exclude those cover properties which are applicable only if some feature is supported in the DUT. While generating the final annotated coverage report, it simply ignores the coverage bins/properties that are not applicable for the given configuration of the IP. For example,

We cover all state transitions by using transition coverage bins. As all the sub-states are modelled as objects of type device_state, the sampling is done whenever the emulated WB state makes a state transition. For this, we use "with function sample" construct to write the cover bins of our interest. Apart from state transitions, we also cover different link speeds, supported link widths, different Ordered Sets, etc. where we use custom function along with “with” construct. This is a custom VCS feature that is supported under LCA. Following are few of the examples:

1. State Transitions:

```
covergroup cg_ltssm_state_transitions (string grp_name, string comment, ref
device_params params) with function sample( device_state tr  );
  cp_ltssm_state_transitions : coverpoint  tr.current_ltssm_state {
    bins detect_quiet_to_detect_active = (S_DETECT_QUIET => S_DETECT_ACT);
    bins detect_active_to_polling_active = (S_DETECT_ACT => S_POLL_ACTIVE);
    bins polling_active_to_polling_compliance = (S_POLL_ACTIVE => S_POLL_COMPLIANCE);
    bins polling_active_to_polling_configuration = (S_POLL_ACTIVE => S_POLL_CONFIG);
    …
    …
    …
    bins loopback_exit_to_detect_quiet = (S_LPBK_EXIT => S_DETECT_QUIET);
    bins hot_reset_to_detect_quiet = (S_HOT_RESET => S_DETECT_QUIET);
}
endgroup : cg_ltssm_state_transitions
```

2. Link Width:

```
covergroup cg_ltssm (string grp_name, string comment, ref device_params params)
with function sample(device_state tr  );
  type_option.comment = "Link state/mode coverage";
  option.name = grp_name;
  option.comment =  comment;

  cp_link_width: coverpoint tr.current_link_width {
    bins cb_sup_widths[] = cp_link_width with (
      is_width_supported(e_link_width'(item), params  ));
  }
endgroup : cg_ltssm

function bit is_width_supported(e_link_width width, device_params params);
  is_width_supported=0;
  if(width==LINK_X1)
    is_width_supported=1;
  else begin
    case (width)
      LINK_X2 : begin
        if( params.num_lanes inside { [LINK_WIDTH_2:LINK_WIDTH_32]} )
            is_width_supported=1;
  end
       LINK_X4 : begin
         if( params.num_lanes inside { [LINK_WIDTH_4:LINK_WIDTH_32]} )
           is_width_supported=1;
       end
       LINK_X8 : begin
         if( params.num_lanes inside { [LINK_WIDTH_8:LINK_WIDTH_32]} )
           is_width_supported=1;
       end
       LINK_X16 : begin
         if( params.num_lanes inside { [LINK_WIDTH_16:LINK_WIDTH_32]} )
           is_width_supported=1;
       end
       LINK_X32 : begin
         if( params.num_lanes inside { LINK_WIDTH_32} )
           is_width_supported=1;
       end
    endcase
  end
endfunction : is_width_supported
```

Transaction Logging:

Usually, any testbench has many debug displays that would help debug failures and narrow down the issues to either the testbench issue or the RTL bug. However, for the complex testbench like the one for LTSSM, it may be difficult to debug using displays as there are many ordered sets being sent and received over the PIPE interface. Transaction logging is a means to create a separate transaction log which would just log the required transaction and displays within. The most important part is that this log can be viewed within the waves along with the design signals. This helps debugging as we can see the design and model behaviour side by side. The following screenshots shows and example of LTSSM transaction logger seen in the waves:

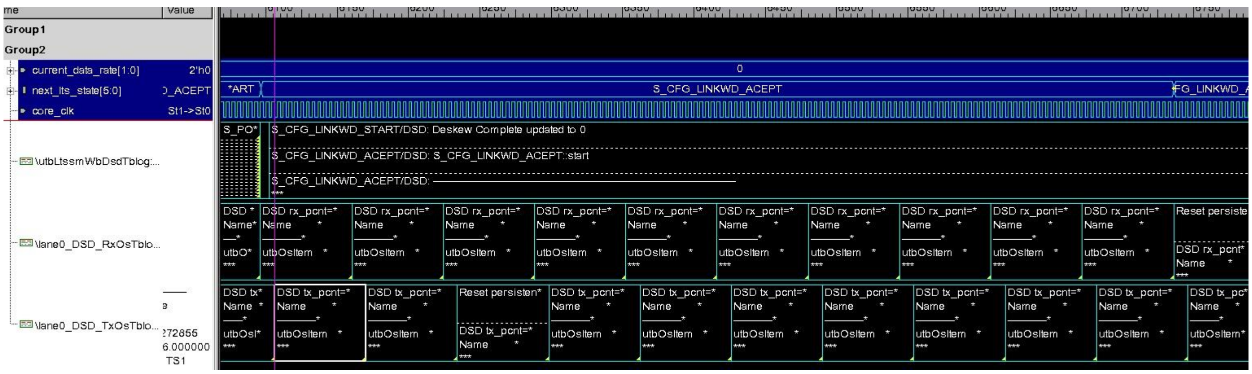


Figure 7 LTSSM Transaction Logger Example In Waves

6. Results

The following are the examples of some of the corner case RTL bugs that were discovered using this testbench:

(1) Link number F7 (non-PAD) when mixed with F7 (PAD) link number, the DUT was not resetting its counter for hitting required transition condition.
State transition condition:
The next state is Configuration.Idle immediately after all Lanes that are transmitting TS2 Ordered Sets receive eight consecutive TS2 Ordered Sets with matching Lane and Link numbers (non-PAD) and identical data rate identifiers (including identical Link Upconfigure Capability (Symbol 4 bit 6)), and 16 TS2 Ordered Sets are sent after receiving one TS2 Ordered Set.
Description of bug:
The value F7 can be a valid non-PAD link number when its corresponding K-symbol signal is 0. However, when K=1, the same F7 value is treated as PAD. We discovered a case in Configuration.Complete LTSSM sub-state where there were 6 TS2 Ordered Sets received with F7 non-PAD link number. This was followed by 2 additional TS2 Ordered Sets with link number PAD (F7 with K-symbol=1). This meant the condition of 8 consecutive TS2s with non-PAD link number was not satisfied. However, the DUT's checking was improper, which was causing state transition. The WB model was robust enough to catch this RTL bug.

(2) The LTSSM was not detecting EQTS2s as valid Ordered Sets in Configuration sub-states.
State transition condition:
The next state is Configuration.Lanenum.Accept if any Lane receives two consecutive TS2 Ordered Sets.
Description of bug:
The Equalization TS Ordered Sets are used in Recovery Equalization states. However, they are considered as TS2 ordered sets anyway. Now, there is a state transition in Configuration.Lanenum.Wait sub-state, which required receiving 2 TS2 Ordered Sets. Due to error injection, we were converting the standard TS2 OSs to EQTS2 OSs, which would still satisfy the state transition condition. Hence the WB model was making a proper state transition, whereas the DUT timed out in the sub-state as it did not treat EQTS2s as valid TS2 Ordered Sets inside Configuration sub-states.

This UVM based LTSSM WB model and the Rx PIPE Agent have proven very effective in verifying the tricky LTSSM conditions. As compared to a traditional approach, this testbench is able to hit many corner scenarios very quickly and hence finding the RTL bugs. Also, we have created around 25 directed random and around 20 constrained random tests for verifying all Configuration and Recovery states. In a fully directed environment, verifying the same functionality would have required around 500 directed testcases. Using this approach, many RTL bugs have been discovered that were never discovered using the existing testbench using directed testing approach. Also, the pace at which the bugs were discovered was very fast as compared to legacy testbench. We have discovered more than 60 RTL bugs within a verified IP using this testbench. While the directed tests have their share of verification coverage and stimulus, most of these bugs were discovered through the constrained random tests.

7. Summary

Initial debug efforts are required to make the LTSSM WB reference model stable. However, once the WB model is robust, the generated random stimulus uncovers a lot of mismatches between WB and DUT behaviour, which in turn helps discover many LTSSM RTL issues. While this whole approach has proved very effective and fast in uncovering RTL bugs of complex LTSSM, it is not just limited to it. We can use this approach to verify any other complex state machines as well.

Lessons Learned:
As mentioned earlier, the initial approach of using VIP for injecting stimulus errors had a dependency on another project. The VIP's known issues and limitation had an impact on the project that slowed the progress of LTSSM verification initially. This inspired the Rx Pipe Agent concept which can be used irrespective of whether the remote link partner is the VIP or the DUT.

Next Steps/Ideas for Improvement:

1. Currently, whenever the WB state transition condition gets hit first, the model always waits till the DUT also makes a state transition. This can be enhanced so that the model shouts error after number of Rx/Tx OSs greater than the threshold are seen after WB state transition condition is hit.
2. Another limitation of the model is that we simply match between the next state of WB and the DUT, but we do not check whether the state transition reasons/conditions are the same. This is a little tricky thing to do, but would surely make the model even more robust.

8. References

[1] PCI Express ® Base Specification Revision 3.0 Version 1.0 November 10, 2010

[2] Universal Verification Methodology (UVM) 1.1 Class Reference