Building And Modelling Reset Aware Testbench For IP Functional Verification

Naishal Shah, Intel Technology India Pvt. Ltd., Bangalore, India (naishalsl@gmail.com)

***Abstract*— Now a days one of the major initiatives taken in the automobile and industrial application world is to develop Functional Safety feature compliant IPs and integrate them to the SoCs. Supporting parity generation and detection features in the IP is one of the important aspects of the Functional Safety. Checking correctness of parity features requires functional verification environment to be entirely reset aware. Parity error handling and verification at IP level proves a significant challenge for verification engineers in terms of time and efforts for recovery from various error scenarios. Most of the errors put an IP into the unrecoverable state and applying reset to the IP is one of solutions offered for recovery from such error scenarios. Modelling reset in all the verification components is an enhancement in the usual testbench development process followed. Subsystems and SoCs can leverage this methodology to validate Functional Safety features like parity at the cluster level also. Different best known methods described in this paper will be useful in reduction of time and efforts for functional verification engineers across different projects.**

Keywords*— *Functional Safety feature validation, parity , stop and scream behavior, stop_sequences(), Quiscence state

I. INTRODUCTION

Reset is a state that exists for almost every single IP, and is usually controlled by at least one, in some cases more, input signal from reset controlling unit. Typically, a design under test (DUT) boots up in the reset state and when the reset signal is de-asserted, the IP comes out of the reset state and walks through its functional states. The reset state may also be the state which the IP starts in when powered up after being power gated. In addition, the IP might need to be reset during run-time for various reasons, either functional reasons such as power gating/power up or recovering from incorrect behavior such as parity error or any functional hazard happened to the IP and where IP exhibits stop and scream behavior . An IP can be reset during operation for verification purposes as well, to check that the IP will behave gracefully if reset is asserted at any time while the IP is in it functional states. All of above makes the reset state and the signals affecting it of great concern to the verification of the IP functionality. The verification environment designed to verify the IP and its Functional Safety related features is required not only to be aware of the reset state and signals causing any state transition to and out of this state, but also be able to verify the IP behaved as expected when going into and coming out of the reset state.

This paper describes how to develop your testbench to make it reset-aware, starting by discussing how this affects the testbench architecture, and then going into more details about each verification component and what changes are required for it to take into consideration the reset conditions.

II. TESTBENCH ARCHITECTURE AND COMPONENTS

Initially, as part of defining the testbench architecture, some questions need to be answered that will define how reset-aware testbench needs to be. The first obvious question is does the IP have a reset state and if it does, what are the conditions and inputs causing the DUT to go into these states are. Several questions follow, when can the DUT enter the reset state? Is it only during power up sequence? And can the DUT go into this state any time the reset signal is asserted, and can the reset be asserted any time during run time? Does the DUT have to be in a certain state before going to reset, idle for example? Can the reset be asynchronous? More interestingly, does the DUT have one reset or does it have a reset signal for each interface in addition to a functional reset?

Answers to the above questions need to be determined from the design specification and the design architects while deciding the testbench architecture. This will allow the testbench architects to account for reset functionality and design the various verification components that comprise the verification environment to be able to handle the different reset conditions and verify that the DUT behaved correctly. Moreover, the answers will define verification requirements that will be translated into test scenarios to be generated and covered as well as checkers to be developed to verify the correctness of the reset state.

A typical OVM verification environment is built using one or more OVM agents representing the different device models in the system, some system-level analysis components like scoreboard and coverage collectors, as well as a virtual sequencer which directs sequences to the appropriate agents to run on their sequencer. In the coming sections, we will discuss in details how each verification component can be designed to account for reset.

A. Development of Testbench Components

a. Driver: The driver is a component of any active agent that takes in transactions from the sequencer and translates them to pin level activity and wiggles the interface pins. It may optionally wait for the response or read data to appear on the interface and send it back to the sequencer. All this activity takes place in the run phase of the driver.

A driver that doesn't handle resets will look like as shown below:

```
task run();
  forever
  begin
    seq_item_port.get_next_item(req);
    // Wiggle Pins and drive bus…
    seq_item_port.item_done();
  end
endtask : run
```

The above code shows a driver that continuously gets a sequence item from the sequencer and drives the bus, regardless which state the bus is in.

The second case is the driver needs to keep monitoring the reset signal throughout the simulation time, and once reset is asserted, the driver needs to make its first decision based on what has been described in the testbench architecture. The driver can either abort the ongoing transaction if one is being driven on the bus, then when reset is de-asserted it can ask the sequencer for a new transaction by calling get_next_item again, or it can stall while saving the current transaction waiting for reset to be de-asserted, this is when it can re-start driving this transaction on the bus from the beginning.

The following code shows an example of a driver that will model a reset-aware bus driver that waits for reset de-assertion at the beginning of simulation. It then continuously drives the bus with traffic from incoming transactions while monitoring the reset condition. When the reset is asserted, it stops driving the bus, drops the current transaction, and drives the bus default values. Finally, it goes back waiting for reset de-assertion, to go through this process again.

```
task run();
  forever
  begin
    wait_for_reset_deassertion();
    fork
      drive_bus();
      monitor_reset_assertion();
    join_any
    disable fork;
    drive_default_values();
  end
endtask : run
```

It is very important to note that the driver needs to call item_done() when reset is asserted in the middle of simulation if a transaction was already being driven on the bus. This will prevent the driver from hanging producing the error that get_next_item() was called twice in a row without item_done() being called.

b. **Monitor:** The monitor is a passive component that resides in an agent, and monitors bus transactions on the pin interface to convert them to transaction level and send them out to the rest of the testbench for consumptions. That said, the monitor doesn't need to wait for reset de-assertion at the beginning of simulation since it is guaranteed there will be no transactions driven on the bus during reset assertion as that was modeled in the driver.

The monitor code changes are very similar to those made to the driver, yet somehow simpler. Below is an example for a monitor that will continuously monitor the bus until reset is asserted. Once reset is asserted, the monitor will drop the transaction it was handling and go back to monitoring a new transaction.

```
task run();
  forever
  begin
    fork
      monitor_bus();
      monitor_reset_assertion();
    join_any
    disable fork;
  end
endtask : run
```

It may be a testbench architecture requirement that the monitor observes and sends the incomplete transaction to its subscribers if needed when reset is asserted, but for the code shown above, the monitor drops the transaction. The code will have to change a little to handle this case. One solution is to make the object handle of the transaction the monitor_bus() method is building up visible outside the method. Then, before disabling the fork process when reset is asserted, clone this object and send it to the analysis port.

In addition to monitoring the reset assertion, the monitor_reset_assertion () task will also inform the agent's components of a reset using the OVM event pool. This event will be used later by other components of the agent to make decisions and take actions on the triggering of the reset event. Following is an example code for the monitor_reset_assertion () task.

```
task monitor_reset_assertion();
  ovm_event reset_event;

  @(negedge v_if.reset);
  reset_event = agent::event_pool.get("reset");
  reset_event.trigger();

endtask
```

c. **Sequencer:** The sequencer is the third component in an agent, and is responsible for managing and arbitrating between sequences running on it. The sequencer then forwards a single transaction to the driver when it asks. The sequencer has an internal queue that holds all pending transactions that are ready to go. In the simple case, the sequencer may need to do nothing when reset is asserted, since the driver will not require any transactions during the reset period. In this case, the sequencer will preserve its internal state during reset, and will continue from where it stopped when reset is de-asserted.

On the other hand, in the more complex cases, the sequencer may need to take the appropriate action when a reset is asserted. Several actions are reasonable depending on the testbench architecture and the answers given to the questions asked earlier in the process. All these actions are implemented in the run phase of the agent's sequencer, and will make use of the event that was declared in the agent's event pool and triggered by the monitor when reset was asserted.

Possible actions can be:

I. **Flush queues**: This step will help clear/empty the internal sequencer arbitration queue which holds the outstanding requests. The queue name is arb_sequence_q and it is a SV queue type, and hence emptying it is accomplished using the delete() method.

```
task run();
 ovm_event reset_event = agent::event_pool.get("reset");
 forever
 begin
   reset_event.wait_ptrigger();
   arb_sequence_q.delete();
  end
endtask : run
```

II. **Stop sequences**: Another possible action is to stop all the sequences running on the sequencer. This can be helpful if the testbench needs to go through some configuration steps that bring up the DUT after reset before it can run actual stimulus. This can be accomplished using the stop_sequences() method which is already implemented in the OVM sequencer base class.

```
task run();
 ovm_event reset_event = agent::event_pool.get("reset");
 forever
 begin
   reset_event.wait_ptrigger();
   stop_sequences();
  end
endtask : run
```

d. Sequences and Tests: Depending on how you architect your sequences and virtual sequences that are started from the test, you can decide what changes are needed for those to react to reset assertion. It is recommended that all the sequences that are run in a test are contained in one virtual sequence and start this virtual sequence on the environment's virtual sequencer in the run phase of the test. Depending on the DUT and what it needs to get out of reset, you may need to either restart the virtual sequence if the agents' sequencers stopped all the sequences on reset assertion. In this case, the virtual sequence is restarted in the test, and will go through the same steps that it went through the first time it ran, like configuring the DUT, setting up some registers…etc. However, if the agents' sequencers only emptied their queues, the virtual sequence and test don't need to take any further action, the sequences are expected to continue running when the DUT comes out of reset.

e. **Coverage:** As far as coverage is concerned, it is recommended to cover that the "reset in the middle of simulation" scenario occurred at least once. This way, you can ensure that the DUT is guaranteed to behave as expected if this scenario happened. This can be achieved either through the covergroups or cover directive.

I. **Covergroups**: In the coverage collector class, declare a variable of type bit with the default value of 0, and in a task set it to 1 on the triggering of the reset event. A coverpoint is then created for this variable for which 100% coverage means that the reset event was triggered. It would probably be more appropriate if a bin is created for this coverpoint to cover the value of 1 only, this way you either get a 0% or 100% coverage for this coverpoint. The code below shows an example for using covergroups to cover the scenario of reset during simulation.

```
class cvg extends ovm_component;

 bit rst_in_sim;
 covergroup mycvg;
  rst_in_sim: coverpoint rst_in_sim {bin asserted = {1};}
 endgroup

 task run;
  fork
   cover_reset_in_simulation();
  join_none
  //.....
 endtask

 task cover_reset_in_simulation();
   ovm_event reset_event = agent::event_pool.get("reset");
   forever
   begin
     reset_event.wait_ptrigger();
     rst_in_sim = 1'b1;
     mycvg.sample();
     rst_in_sim = 1'b0;
   end
 endtask
endclass
```

II. **Cover directive**: In this case, instead of using the coverage collector class, a SVA property will be declared that describes the scenario that reset started as asserted, then it got de-asserted sometime later, and at any time before the end of simulation it got asserted and de-asserted again. Then the cover directive is applied to this property to ensure it was covered. An example property code is shown below. This code is usually implemented in the interface, or any other file where other SVA properties are declared and bound (using the SV bind construct) to the interface.

```
property reset_in_sim;
 @(posedge clk) (!reset [*1:$] ##1 reset )[->1];
endproperty

cover property (reset_in_sim);
```

On the other hand, the normal transaction coverage collection is halted by default during reset since the monitor is no longer broadcasting transactions on its analysis port during reset assertion.

f. **Scoreboard and reference model:** The scoreboard usually has several internal analysis fifos that store transactions collected from the corresponding agents and consume those transactions as needed when the scoreboard needs to compare actual results with expected results. Figure 1 shows a simple block diagram of a general scoreboard architecture. In fact, when reset is asserted, the scoreboard doesn't need to do anything with its internal fifos, since their contents are transactions that already took place and hence are complete. The incomplete transactions are being taken care of by the monitors and the monitors either will not send those incomplete transactions to the scoreboard, or send a transaction indicating it is incomplete and the scoreboard should be able to handle this case.

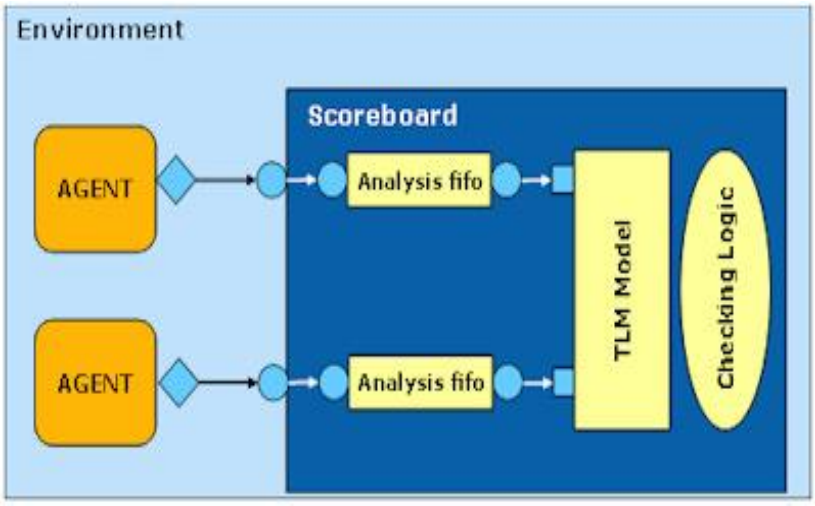

Figure 1. Scoreboard General Architecture

However, the scoreboard usually has the TLM or behavioral reference model representing the DUT that computes the expected behavior of the DUT. Since this model resembles the DUT, it is very important that it monitors for reset assertion and takes necessary action too. First, the model can use the reset event triggered by the agent's monitor to know when reset is asserted; the same way the coverage collector used it. Once reset is asserted, the model needs to take the same actions the DUT takes in reset state. These actions may include but are not limited to resetting registers and internal storage or memory, resetting a finite state machine to the default state, empty its queues or stop expecting more messages of a certain type.

It is very important to note that the model should not stop computing the expected behavior of a transaction if reset is asserted. In other words, if the model happens to be calculating the output for a transaction at the time reset is asserted, it needs to continue and send that expected behavior to the scoreboard before it can take those reset actions.

B. *Implementation in UVM*

Most of what we described previously applies to both Open Verification Methodology and Universal Verification Methodology. However, UVM adds some more capabilities specifically to the testbench phases that allow the testbench developer have more fine grain control over the testbench behavior.

UVM introduces several new phases that didn't exist in the OVM. The one of interest to our discussion is the reset phase and its callbacks: pre_reset and post_reset. In those phases, the testbench is supposed to define the reset behavior and behave accordingly. For example, in the pre_reset phase, the driver should drive X's or Z's on the inputs of the DUT and wait for power good conditions before moving to the reset phase when the driver should drive default values of those signals.

The other thing UVM introduces is phase jumping. Jumping allows the testbench to move from one phase to another without restricting the order of the two phases, i.e. the testbench can jump back to a previous phase or jump forward to skip a phase. UVM provides two jump methods: one that jumps to a phase within the current phase schedule and another that makes all schedules jump to the specified phase.

The new UVM capabilities and phases allow us to change the implementation of a reset-aware testbench a little bit. First, the method drive_default_values () implemented in the driver is not needed anymore. This is the implementation of the reset_phase() of the driver. The code that resets the register model also is no longer in the main_phase () (note that it is no longer run, even though run_phase exists in UVM) of the agent, it is moved to the reset_phase(). The monitor will still need to monitor reset and trigger the reset event for the verification components to use.

Another major difference in the implementation is the agent behavior when the reset event is triggered. The agent needs to monitor the reset event and when triggered, it needs to jump and make all the components jump to the reset_phase() phase. It shall use the jump_all(reset_phase) so that all the sub-components (driver, monitor, sequencer…etc.) jumps to this phase and start from there again. This will ensure that the VC goes through the correct phases (reset, configure, main…etc.) and executes the corresponding code to bring the DUT out of reset and into functional mode

III. RESULTS

As of today we had fully validated parity checking feature (Functional Safety Feature Validation) which requires IP to implement the stop and scream behavior (indicates the hang on the interface / No traffic can be further progressed) using reset aware testbench. During execution of the feature validation we found many sequences not be able drop objections which are raised at the start of the simulations (hanging sequence) related problems because of void quiescence state (quiescence state is state where IP is processing traffic coming on interface and not able to achieve idle state). We have root caused, debugged this issues and to overcome this problems we have incorporated the methods mentioned (in body part) in IP verification collaterals. During design bring up we uncovered integrated sub-IP related bugs described as sub-IP was releasing credits in the parity error occurred window to the fabric , which allowed fabric to send further non-posted transactions unexpected to the IP.

IV. SUMMARY

Implementing good IP testbench is always must have foundation to enable a successful validation at the IP and Integration level. Reset is an important state of an IP and the testbench needs to be designed to accommodate and handle this state especially where IP implements Functional Safety features i.e (parity checking) and requires verification of such features. We have seen throughout the paper the necessary changes to the various testbench components in order to deal with the different types of reset conditions. We have also discussed how those changes differ if the testbench is UVM-based versus OVM-based. It is our goal to keep enhancing testbench structures by ensuring successful validation of any IP.

ACKNOWLEDGMENT

The author would like to thank bridge IP verification team for the support and help during execution of the work done to which I could present this article. Special thanks to Vikrant Pai for guiding and helping me to find solutions for the problems faced during verification of the feature. Special thanks to Vishal S. Namshiker for motivating me to write this paper.

REFERENCES

[1] “OVM User Guide, Version 2.1.2” , June 2011. [online] Available : http://www.specman-verification.com/source_bank/ovm-2.1.2/ovm-2.1.2/OVM_UserGuide.pdf

[2] "Intel System Fabric Specifications, Revision 1.2", May 8, 2015.

[3] "Internal IP High-Level Architectural Specification (HAS) Document" , October 2018