OVM & UVM Techniques for On-the-fly Reset

Muralidhara Ramalingaiah

Cypress Semiconductor Technology India Pvt. Ltd.
65/2 Bagmane Tech Park,
C.V. Raman Nagar, Bangalore, INDIA.
(91)-80-6707-3638
mura@cypress.com

Boobalan Anantharaman

Cypress Semiconductor Technology India Pvt. Ltd.
65/2 Bagmane Tech Park,
C.V. Raman Nagar, Bangalore, INDIA.
(91)-80-6707-3102
boh@cypress.com

ABSTRACT

When the Design Under Test (DUT) is reset during normal operation, the testbench must act accordingly and must not behave abnormally or give ambiguous results. The Open Verification Methodology (OVM) and the new Universal Verification Methodology (UVM) both have a number of ways to implement on-the-fly reset in various testbench components like the monitor, driver and scoreboard.

In the Open Verification Methodology (OVM) there is no reset phase. So it will be difficult to stop a testbench component (like driver) during “on-the-fly reset" while the component is processing a transaction. The reset information must reach the scoreboard too, to halt the checks and come to the initial state. The test case should have control to apply “on-the-fly reset” to the testbench components and the scoreboard should be well able to recognize an “on-the-fly reset”.

In the Open Verification Methodology (OVM) the driver/monitor/scoreboard code should be able to react to a reset, but the real potential problem occurs whenever reset happens during normal operations.

This paper is going to explain the techniques through which the driver can be stopped immediately whenever there is a reset and start driving the reset values on the bus and send the same items to the analysis port, so even the scoreboard and coverage model work well whenever there is an on-the-fly reset.

OVM Verification Components (OVCs) or UVM Verification Components (UVCs) needs to support on-the-fly reset, instead of the user controlling it from the test case. This paper will show effective techniques to implement on-the-fly reset for components like the driver, monitor, and scoreboard in OVM and UVM. This paper will also present, how one can easily implement on-the-fly reset logic in the Universal Verification methodology (UVM) with minimal changes using phasing for the same code implemented in OVM.

Keywords:

DUT, OVM, UVM, OVC, UVC, driver, monitor, virtual sequencer, scoreboard, coverage model.

1. INTRODUCTION

On-the-fly reset happens during DUT operation in the form of hard reset or soft reset. The reset that happens through an external reset (XRES) input is a hard reset. The reset that happens by writing into a register (e.g. REG1) through firmware is a soft reset. Figure 1 shows the sources for on-the-fly reset.

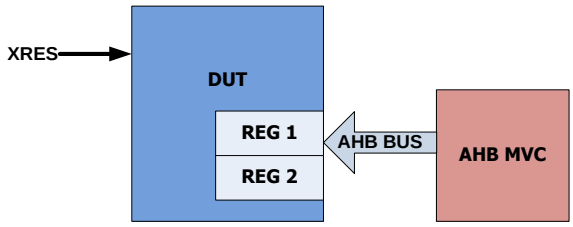

Figure 1 On-the-fly Reset Sources

During on-the-fly reset the testbench must not behave abnormally or give ambiguous results.

On-the-fly reset logic can be implemented in various testbench components like monitors, drivers and scoreboards. During on-the-fly reset the following things should happen in the testbench,

- Scoreboard variables should be reset
- The monitor should recognize on-the-fly resets and qualify the transactions
- Coverage update logic should not trigger any transition coverage except reset coverage
- The driver should recognize resets and stop transactions at the right time
- Data Checkers should delete the transactions which are collected in FIFOs or QUEUEs
- Protocol Checkers should recognize on-the-fly reset

2. OVM RESET TECHNIQUES

The Generic OVM reset technique for on-the-fly reset implementation is based on global reset events that are generated by monitoring the on-the-fly reset sources in the DUT through the reset interface monitor.

The global events will be used by the scoreboard to reset the scoreboard variables, to reset registers in the register model, and to qualify the data obtained through the analysis ports of interface OVCs/monitors.

The same on-the-fly reset global events will be used to exclude the functional coverage of all DUT functionalities other than reset functionality.

Figure 2 shows a sample OVM testbench environment with on-the-fly reset logic implementation.

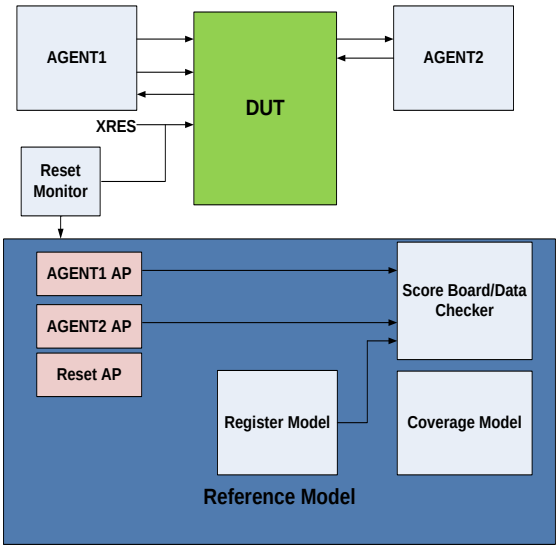

Figure 2 Sample testbench environment

The Reset monitor shown in Figure 2 monitors the external reset (XRES) pin. Whenever the reset occurs through XRES, the monitor will create/trigger a global reset event. The Reset monitor code is shown below,

RESET Monitor

```
function void build(); //reset event creation
    oep = oep.get_global_pool();
    is_reset = oep.get("RESET_EVENT");
endfunction

task run() begin
    forever begin
        @(negedge rstagent_if_monitor.reset)
        is_reset.trigger();
    end
 endtask
```

The scoreboard shown in Figure 2 looks for the global reset event from the reset monitor and resets the registers in the Register model. The scoreboard code is shown below,

Scoreboard

```
task run() begin
    ovm_event_pool oep;
    ovm_event is_reset;
    oep = oep.get_global_pool();
    is_reset = oep.get("RESET_EVENT");
    forever
    begin //{
        is_reset.wait_trigger();
        register_map.reset();
    end //}
endtask
```

The Interface Agent1/Agent2 driver and monitor will have their own reset logic and work independent of the global reset monitor event.

2.1 Issues with OVM reset technique

The following are issues with the OVM reset technique explained above,

- Drivers don't recognize the on-the-fly reset between transactions
- Irrespective of the on-the-fly reset, the monitor will send data to scoreboard/coverage components
- During on-the-fly reset, it is difficult to control the sequencers of the sub-sequences from the virtual_sequence

2.1.1 Driver issue

In the above OVM reset technique, drivers will be driving the bus without considering that on-the-fly reset occurred during execution of some transactions. Figure 3 shows a driver that has two driver tasks that are running independently. The RESET DRIVER task gets a reset in between and drives the interface with default values. But the DRIVER task that is running in parallel to the RESET DRIVER task will be driving the interface still without considering the reset. Due to this, some unwanted transactions will be driven to the interface during reset.

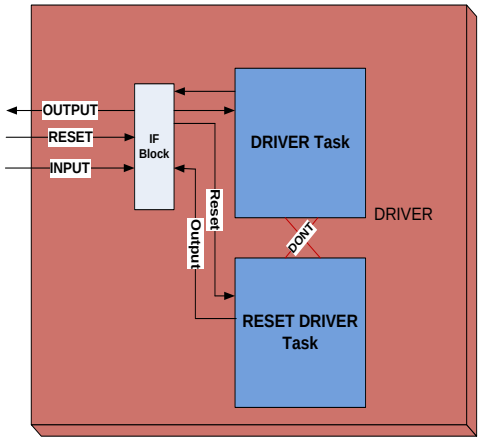

Figure 3 Driver Issue

The code below shows the top level driver that has the driver task get_and_drive() and the reset task reset_signals(), which are running in parallel.

Driver

```
 task run();
    fork
       get_and_drive();
       reset_signals();
    join_none
endtask

task get_and_drive();
   @(wait for reset to finish);
    forever begin
       seq_item_port.get_next_item(req);  or try_next_item
       $cast(rsp, req.clone());
       rsp.set_id_info(req);
       drive_transfer(rsp);
       @(posedge intf.cb);
       send_idle(rsp);
       seq_item_port.item_done(rsp);
   end
endtask

 task reset_signals();
    forever begin
       @(wait for reset);
       //Reset value of DAT_out
       intf.TB.cb.DAT_out  <= 'b1;
       intf.clk_en   = 'b0;
    end
endtask
```

2.1.2 Monitor issue

In the above OVM reset technique, Irrespective of the on-the-fly reset, the monitor will send the transactions to scoreboard/coverage components. These transactions are invalid because the monitor should send only the default values during reset. Figure 4 shows two monitors that are sending data to a scoreboard.

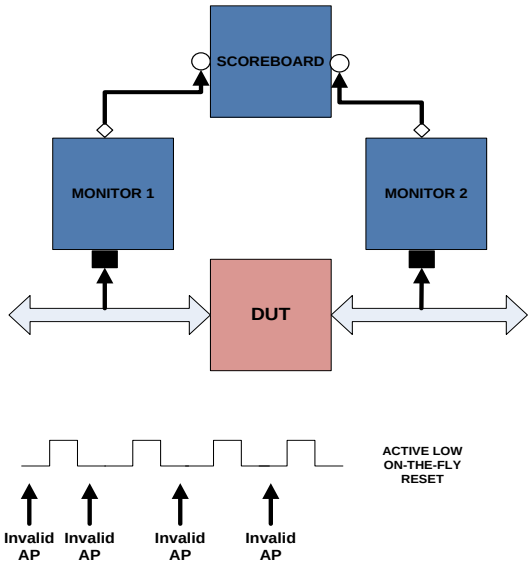

Figure 4 Monitor Issue

During the active low on-the-fly reset shown in Figure 4, the data sent by the monitor to the scoreboard analysis port is invalid.

The code below shows an agent monitor that has a monitor_transactions() task and a reset_transactions() task running in parallel. During the on-the-fly reset, the monitor_transactions() task will be sending data to the scoreboard without considering the reset.

Monitor

```
// Interface OVC/Agent MONITOR
task run();
    fork
       monitor_transactions();
       reset_transactions(); //Reset
    join_none
endtask
```

2.1.3 Sequence issue

In the above OVM technique, during on-the-fly reset the sub-sequencers/sub-sequences in a virtual_sequence cannot be controlled. Figure 5 shows a virtual sequence that has two threads running in parallel. One is the reset sequence and other is the normal layered sequence that has sub-sequences running in parallel. If the reset happens in between, the verification engineer cannot determine, when to stop the sub-sequencers/sub-sequences.

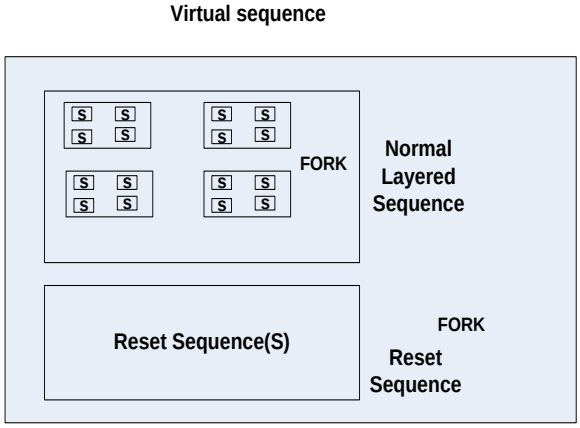

Figure 5 Sequence Issue

2.2 Solutions for OVM reset technique issues

The following are solutions for the OVM reset technique issues explained in section 2.1,

- Agent with qualified clock approach and qualified data from monitor to scoreboard/coverage

- Stopping the sequencer/sequence from the virtual sequence
- State Machine approach

2.2.1 Qualified clock and qualified data approach

This solution is for the driver issue specified in section 2.1.1 and the monitor issue specified in section 2.1.2. In an agent, the interface should have a clock that is qualified by reset. The code below shows the clock (clk) that is qualified by the reset (sig_reset).

Agent IF

```
assign qclk = (sig_reset==1) ? clk:0;
  // Actual Signals
  // USER: Add interface signals
 clocking cb @(posedge qclk);
```

In the driver, the qualified clock will be used to stop driving the data into the interface of DUT. The reset driving logic should not work based on the qualified clock. The code below shows the top level driver that has the driver task get_and_drive() and the reset task reset_signals() that are running in parallel.

The task get_and_drive() has two threads running in parallel. One thread drives the transactions and the other thread looks for the RESET_EN event from the reset_signals() task. When reset occurs in between, the drive_transfer task will not drive any transactions and the thread that looks for the RESET_EN event will end the transaction execution and then suspend the drive_transfer task.

Note: Don't use the kill() function in this case, because it may hang the simulation. Use the suspend() function instead of that.

DRIVER

```
task run();
   fork
      reset_t = RESET_DE;//Initialize reset_t variable
      get_and_drive();
      reset_signals();
   join_none
endtask

 task get_and_drive();
    //wait for the initial Power On Reset
    @(reset)
    reset_t = RESET_DE; //disable event or variable
    fork
       forever begin
          seq_item_port.get_next_item(req);
          ovm_report_info(get_type_name(), "sequencer got
                                  next item");
          drive_transfer(rsp);//Works on qualified clock
         @(posedge intf.clk);//normal clock
          seq_item_port.item_done(rsp);
       end

       begin
       //Block to suspend the driver and say end of
       //transfer(solution)
          forever begin
             if(reset_t==RESET_EN)begin
                ovm_report_info(get_type_name(), " ON-THE-
                                       FLY  RESET");
                this.end_tr(rsp);
                reset_t = RESET_DE;
                //drive_transfer.kill();//Never use this will make
                                        //to hang
                 drive_transfer.suspend();//Use this but still it will
                                              //send data after reset
             end
             @(posedge intf.clk);
          end
       end
    join_none
endtask
```

Figure 6 shows two monitors that are collecting and sending data to a scoreboard. During the active low on-the-fly reset shown in Figure 6, the monitors will not send data to the scoreboard.

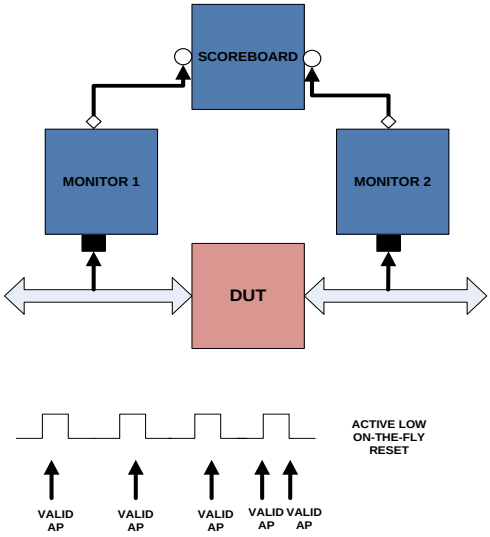

Figure 6 Qualified Data from Monitor

The code below shows a monitor that has two tasks monitor_transactions() and reset_transactions() running in parallel. The monitor_transactions() task has a collect_transfer() task that collects the data only when there is a qualified clock. This data is the qualified data for the scoreboard.

Monitor

```
task run();
   fork
      monitor_transactions();
      reset_transactions();//To reset all monitor variables
    join_none
endtask
```

```
task monitor_transactions();
begin
   @(Reset event);//From negative to positive
    forever begin
       trans_collected = new();
       @(posedge intf.cb);
       //don't collect data when there is no clock
       collect_transfer();
       data_trans();
    end
    if (checks_enable)  // Check transaction
       perform_transfer_checks();
    if (coverage_enable)   // Update coverage
       perform_transfer_coverage();
    // Publish to subscribers
     item_collected_port.write(trans_collected);
 end
endtask

task collect_transfer();
   void'(this.begin_tr(trans_collected));
   trans_collected.trans_kind = WRITE;
  @(posedge intf.cb);//qualified clock
  data.push_back(intf.TB.cb.DAT_out);
  forever begin
     @(posedge intf.cb);//qualified clock
  end
endtask
```

Limitation: During reset, the driver will not drive anything. But if the driver has some wait logic like @posedge of some signals, then the driver will run the logic under that wait logic before suspending the driver task that leads to malfunction of the driver/monitor and the analysis port.

2.2.2 Stopping the sequencer from virtual sequence

This solution is for the sequence issue specified in section 2.1.3. During on-the-fly reset, the sequence/sequencer should be stopped from the virtual sequence or test case.

The code below waits for the "on-the-fly reset" event and stops the sequencer when there is a on-the-fly reset.

Virtual Sequence/Test Case

```
fork
   begin
      int_agent_seq.start(int_agent_sequencer);
   end
   begin
      wait(flyonreset)
       fork
          begin
             reset_seq.start(reset_agent_sequencer);
             int_agent_sequencer.stop_sequence();
          end
          begin
             //sub_sequence
          end
       join
   end
join
```

Limitation: The verification engineer should know when to stop the sequencer/sequence from the virtual_sequence or test case.

2.2.3 State machine approach

This solution is for the driver issue specified in section 2.1.1 and monitor issue specified in section 2.1.2. Inside an agent driver or monitor, a state machine approach could be used to look for the on-the-fly reset in every state. If on-the-fly reset happens in any of the states, the state machine will stop the transaction and go to the reset state.

Figure 7 shows the driver state machine. END is the Boolean value that signals the end of the sequence item. If END is false the state machine goes to the IRDA_DRIVE state from the IDLE state and the driver starts driving the transactions. If there is a on-the-fly reset, the sequence-item will be terminated and the state machine will go to the reset state.

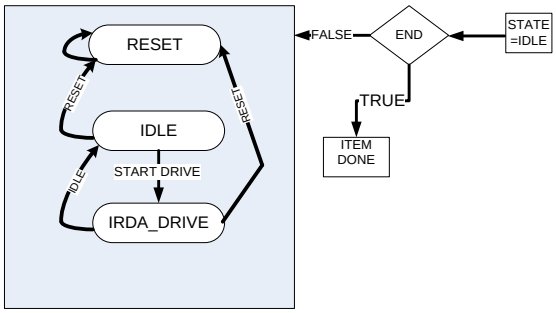

Figure 7 Driver State Machine

The code below shows the driver state machine implementation.

Driver

```
task run();
   fork
      get_and_drive();
    join_none
endtask

task get_and_drive();
   @(reset)
    forever begin
       seq_item_port.get_next_item(req);
       drive_transfer(rsp);//Works on qualified clock
      @(posedge intf.cb);//normal clock
       seq_item_port.item_done(rsp);
    end
endtask

task drive_transfer();
   STATE =IDLE;
   while(END==FALSE) begin
   case(STATE)   //STATE MACHINE
      RESET: begin
                  ovm_report_info(get_type_name(), "RESET
                                  STATE");
                  //STOPPING TEST CASE
```

```
                END = TRUE;
              end
      IDLE : begin //default values
                if(if.reset) STATE= RESET;
               else begin //put next logic
                  STATE = IRDA_DRIVE; end
             end
      IRDA_DRIVE : begin
                      for( number of data) begin
                        if(if.reset) STATE= RESET;
                       else begin //put next logic
                          STATE = IRDA_DRIVE;
                       end
                     end //End of test
                     END = TRUE;
                   end
   endcase
   @(posedge intf.cb);//normal clock
   end
endtask
```

Figure 8 shows the monitor state machine. After the initial reset, the monitor state machine will go to the IRDA_COLLECT state, collect the data and send transactions to the scoreboard. During on-the-fly reset, the monitor state machine will send the default values along with reset information to the scoreboard.

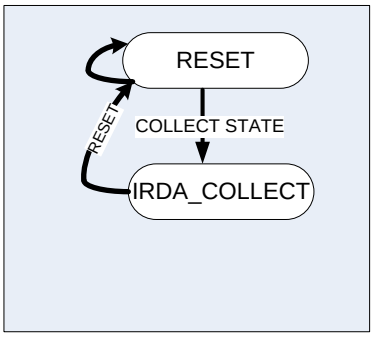

Figure 8 Monitor State Machine

The code below shows the monitor implementation.

Monitor

```
task run();
   fork
      monitor_transactions();
      reset_transactions();//To reset all monitor variables
   join_none
endtask

task monitor_transactions();
   @(Reset event);//From negative to positive
    forever begin
       trans_collected = new();
      @(posedge intf.cb);
       collect_transfer();
       data_trans();
    end
    if (checks_enable)  // Check transaction
       perform_transfer_checks();
    if (coverage_enable)  // Update coverage
       perform_transfer_coverage();
    // Publish to subscribers
    item_collected_port.write(trans_collected);
endtask

task collect_transfer();
   void'(this.begin_tr(trans_collected));
   //Have the state machine here to collect data
      forever begin
       RESET: begin //default values
                 if(if.reset)begin
                    STATE= RESET;
                    trans_collected.state=Reset;
                    trans_collected.data=0;
                 end
                 else begin //put next logic
                    STATE = IRDA_COLLECT;
                     trans_collected.state=STATE;
                     trans_collected.data=0;
                 end
               end
      IRDA_COLLECT: begin
                       for( if.data.valid==1) begin
                         if(if.reset)  begin
                           STATE= RESET;
                           trans_collected.state=Reset
                           trans_collected.data=0;
                         end
                         else begin //put next logic
                           STATE = IRDA_COLLECT;
                           trans_collected.data=if.data;
                         end
                       end
         @(posedge intf.cb);//wait for one normal clk cycle
   end
   //During reset Transaction will have reset information
   // and default values
endtask
```

Benefits: This state machine approach makes life easier with additional logic (e.g. end transaction logic). If we follow this approach, the sequence issue specified in section 2.1.3 will not occur. This approach is very accurate compared to all of the previous methods and it is highly recommended to use this technique in OVM testbenches.

Limitation: Driver and monitor code is bound on reset at every clock cycle. The driver code will be complex.

3. UVM RESET TECHNIQUES

To understand the UVM reset techniques, the user should know about the UVM run-time phases.

3.1 UVM run-time phases

The run-time schedule is the pre-defined phase schedule that runs concurrently with the global run phase “uvm_run_phase.” The run-time phases are executed in a predefined order. The UVM run-time phases and the order in which they execute are shown in Figure 9 below.

Except for the UVM reset phases the other run-time phases are beyond the scope of this paper. We will examine the UVM reset phases below.

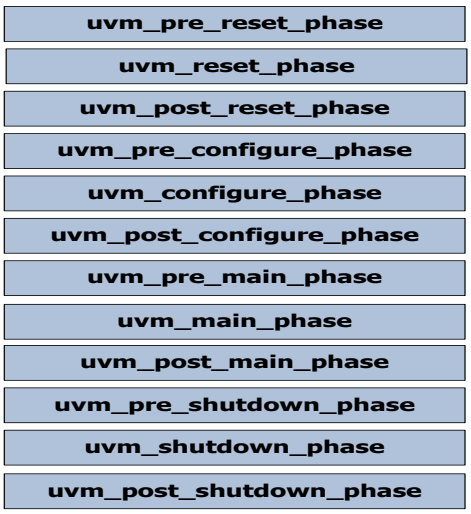

Figure 9 UVM run-time phases

3.1.1 Pre reset phase

The UVM pre reset phase is used to,

- Wait for power good
- Initialize the output of the components connected to virtual interfaces to X's or Z's
- Initialize the clock signals to a valid value
- Assign reset signals to X (power-on reset)
- Wait for reset signal to be asserted if not driven by the verification environment

3.1.2 Reset phase

The UVM reset phase is used to,

- Assert reset signals
- Drive the output of the components connected to virtual interfaces to their specified reset or idle value
- Initialize the components and environments to their initial states
- To start generating active edges from clock generators
- De-assert the reset signal(s) just before exit
- Wait for reset signal(s) to be de-asserted

3.1.3 Post reset phase

The UVM post reset phase is used to start the behavior appropriate for reset being inactive from the components. For example, components may start to transmit idle transactions.

Note: The UVM reset phase uses listed above in section 3.1.1, 3.1.2, and 3.1.3 are from UVM documentation.

3.2 UVM Reset Implementation using UVM Reset Phase

The on-the-fly reset implementation will be done in UVM using the UVM Reset phase.

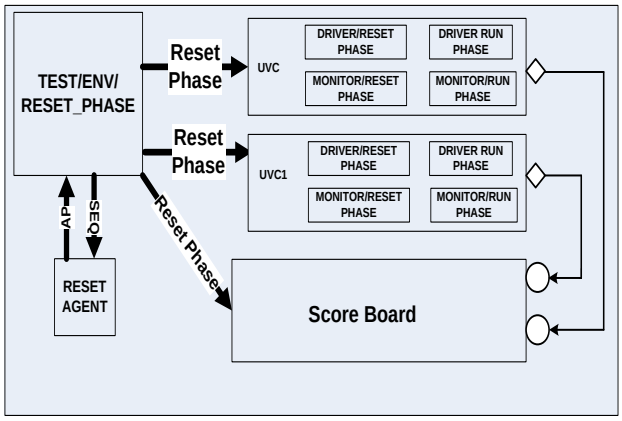

Figure 10 UVM Reset phase

Figure 10 shows how the reset_phase propagates from Test or ENV to all other components in the testbench.

The code below shows the on-the-fly reset implementation in the UVM Agent. In the reset_phase the reset_and_suspend() task will be called during the on-the-fly reset. The Agent reset_and_suspend() task will call the driver/monitor reset_and_suspend() tasks and then stop the sequences.

Agent

```
task reset_phase(uvm_phase phase);
    phase.raise_objection(this, "Resetting agent");
       reset_and_suspend();
    phase.drop_objection(this);
endtask

virtual task reset_and_suspend();
   fork
      drv.reset_and_suspend();
      tx_mon.reset_and_suspend();
      rx_mon.reset_and_suspend();
    join
   sqr.stop_sequences();//Stop sequences but can't
                        //stop driver immediately
endtask
```

The code below shows the on-the-fly reset implementation in the UVM driver. During the on-the-fly reset, the reset_phase will call the reset_and_suspend() task to drive default values to the interface.

Driver

```
task reset_phase(uvm_phase phase);
   phase.raise_objection(this, "Resetting driver");
      reset_and_suspend();
   phase.drop_objection(this);
endtask
```

```
 virtual task reset_and_suspend();
   //drive default values to all the interface
endtask

//This Task will be called from RUN_PHASE
task get_and_drive();
   //wait for the RESET event to complete to negedge of
   //reset
   //Wait for reset and suspend
   reset_and_suspend();
   forever begin
      seq_item_port.get_next_item(req);
      drive_transfer(rsp);//Works on qualified clock
      @(posedge intf.cb);//normal clock
      seq_item_port.item_done(rsp);
   end
endtask

virtual protected task run_phase(uvm_phase phase);
    forever begin
       //Call all methods to start
       get_and_drive();
   end
endtask
```

The code below shows the on-the-fly reset implementation in the UVM monitor. During the on-the-fly reset, the reset_phase will call the reset_and_suspend() task to clear all the local variables.

Monitor

```
task reset_phase(uvm_phase phase);
   phase.raise_objection(this, "Resetting driver");
      reset_and_suspend();
    phase.drop_objection(this);
endtask

 virtual task reset_and_suspend();
       //Clear all local variables
endtask

virtual protected task run_phase(uvm_phase phase);
 forever begin
    //Call all methods to collect data
end
```

Control the Reset phase from the test case, ENV or Reset monitor (as shown below),

```
task main_phase(uvm_phase phase);
   `uvm_info("TEST", "Jumping back to reset phase",
                UVM_NONE);
   phase.jump(uvm_reset_phase::get())
end task
```

Benefits in UVM: All components can be controlled from the Env or Test. The reset_phase of each component shall be in sync.

Limitation in UVM: The driver will still process the last received item. To overcome this limitation, the state machine approach explained in section 2.2.3 could be used.

4. OVM TO UVM MIGRATION - TIPS FOR RESET

The following are tips for migrating a testbench from OVM to UVM with respect to reset/on-the-fly reset,

- Ensure that all the reset logic implemented in OVM should be moved to the reset phase in UVM.
- Control the reset phases of each component such as Interface agents, monitors and drivers from the environment or test in UVM. Better option is to control from the environment.

5. SUMMARY

For the on-the-fly reset implementation, the state machine approach is the preferred method in OVM testbenches when compared to the other approaches explained in section 2.2.

To implement the on-the-fly reset, UVM has the reset_phase by which reset logic of all the components can be controlled from tests or verification environments. To avoid the driver limitation specified in section 3.2, the state machine approach is recommended in UVM testbenches too.

6. ACKNOWLEDGEMENT

The authors are grateful to the supportive family members and the Cypress Management.

7. REFERENCES

[1] SystemVerilog 3.1a Language Reference Manual
[2] OVM Class Reference Version 2.1.1
[3] Universal Verification Methodology (UVM) 1.1 User's Guide
[4] OVM golden reference guide from Doulos