On the Fly Reset

by Mark Peryer, Verification Methodologist, DVT, Mentor Graphics

INTRODUCTION

A common verification requirement is to reset a design part of the way through a simulation to check that it will come out of reset correctly and that any non-volatile settings survive the process. Almost all testbenches are designed to go through some form of reset and initialization process at their beginning, but applying reset at a mid-point in the simulation can be problematic. The Accellera UVM phasing sub-committee has been trying to resolve how to handle resets for a long time and has yet to reach a conclusion.

In a UVM testbench, stimulus is generated by sequences which create and shape sequence items which are sent to a driver for conversion into pin level activity compliant with a specific protocol. Sequences pass sequence items to drivers via a hand-shake and arbitration mechanism implemented in the sequencer. This mechanism is based on the assumption that each protocol transaction will complete. If a reset occurs, then a sequence will need to be terminated early. Killing a sequence abruptly potentially results in the driver trying to return a handshake or a response to a sequence that has been deleted from the sequencer, causing a fatal error in the simulation. One way to avoid this problem, is to implement the driver so that when it detects a reset it terminates the in-progress sequence items with a reset status so that the stimulus control process can handle the reset appropriately.

HOW TO HANDLE A RESET

A reset can be considered as a highly disruptive event that occurs unexpectedly. Typically, a hardware reset will cause a DUT to immediately stop what it is doing and re-apply default values to its register contents. On exit from reset the DUT will most likely need to be re-initialized or re-configured before it can start normal operations. There are two ways in which a reset can be generated in a UVM testbench; the first is when the UVM stimulus thread is in control of the reset and the second is when the static side of the testbench generates a reset asynchronously to the flow of the UVM stimulus process.

Generally speaking, if the reset is generated under the control of the UVM testbench, the stimulus control process can make sure that all of the stimulus generation machinery is in a quiescent state before asserting the reset signal. The control process would de-assert reset and then go through the required initialization process before resuming the rest of the test. In many circumstances this may be good enough, but there may be situations where an external asynchronous reset is necessary to check DUT behavior, or resets may be generated by the UVM testbench at random intervals.

An asynchronous reset places two requirements on the UVM testbench code – it must first detect that a reset has occurred and then it must react to the reset. In other words, a testbench needs to be coded so that components such as drivers, monitors and scoreboards are reset aware and the stimulus generation process needs to be able to react to the occurrence of a reset.

The stimulus generation process in a UVM testbench may well use a hierarchy of sequences and there could be multiple sequences sending sequence items to a driver at any point in time. At the top level of the sequence hierarchy there will be a master execution thread that is coordinating the execution of all the different sequence threads, usually in the test or in a virtual sequence. A hardware reset will be detected by the driver since it is driving and monitoring the pin level interface. At the start of a reset, the driver needs to halt its operations and then signal back to all the sequences that are sending sequence items that a reset is in progress. In turn the sequences need to terminate, returning a reset status. If the sequences are hierarchical, then they need to terminate with a reset status recursively until the main control loop is encountered. The control loop then needs to change the stimulus generation flow so that a new re-initialization process is started before re-applying traffic stimulus.

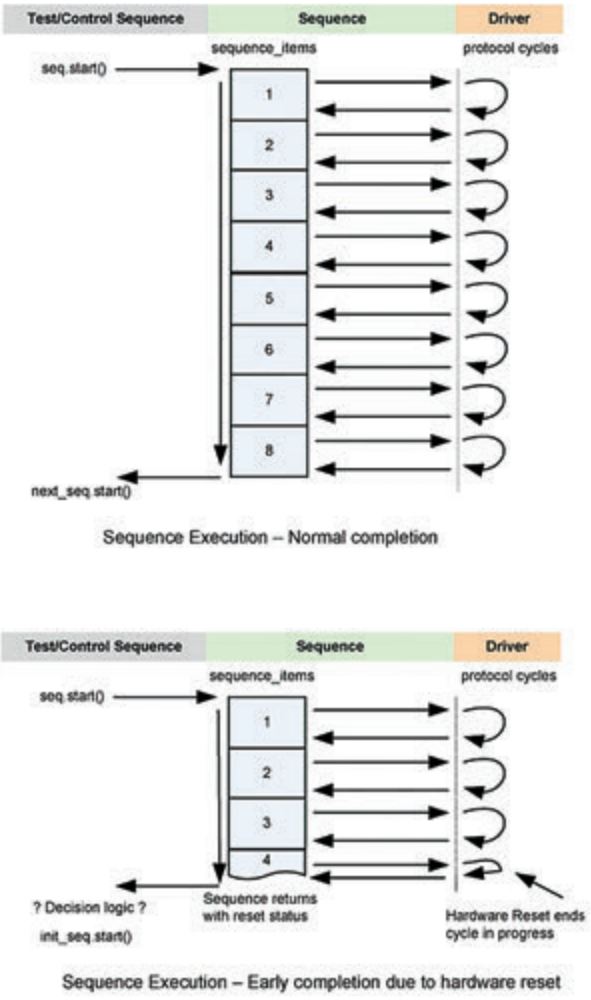


Figure 1 - Comparison of sequence execution flow for normal completion and a reset

If a driver is not active, it will detect a reset but it will not be able to return the reset status. If all the drivers, sequences and sequence items in the testbench are reset aware, then it is very likely that at least one of them will return a reset status that will allow the stimulus process to go into a reset handling thread. However, if this is not the case and the reset status is important, then the main stimulus generation loop should monitor the state of the reset signal in parallel with its normal activities and initiate a change in stimulus if a reset occurs.

IMPLEMENTING RESET HANDLING

The key to being able to handle an asynchronous reset in UVM testbenches is to be able to flag detection of a reset condition back to the process that controls the test and the stimulus generation. The most convenient way to do this is to add a status field to sequence items and sequences to communicate that a reset has been detected. This status field can then be tested as part of the stimulus generation process to determine what to do. The UVM base classes do not contain a suitable field, so this is something that the user has to add. The recommended approach is to add a variable called **status** of type **uvm_tlm_ response_ status_e** to sequence items and sequences. When a sequence item or a sequence completes successfully, the status field would be set to UVM_TLM_OK_RESPONSE. When a reset occurs then status value of UVM_TLM_ INCOMPLETE_RESPONSE should be set.

```
class adpcm_seq_item extends uvm_sequence_item;

rand logic[31:0] data;
rand int delay;

// Response - used to signal whether
transfer was aborted due to reset
uvm_tlm_response_status_e status;

`uvm_object_utils(adpcm_seq_item)

// etc

endclass: adpcm_seq_item
```

Figure 2 - Sequence Item with status field

In the body method of a reset aware sequence, the status field of each sequence item should be checked after driver has completed its execution – usually on return from the finish_item() or the get_response() call. If the status returned is normal, then the sequence continues as usual. If the status field indicates that a reset has occurred then the sequence should stop what it is doing and return, having first set its status field to indicate that a reset has been encountered. The next level in the stimulus generation hierarchy must also test the status of the returning sequence and determine what to do with a reset status. For instance, instead of starting the next planned traffic sequence, it may start an initialization sequence which will start as soon as the hardware reset sequence has completed.

```
class adpcm_tx_seq extends uvm_sequence
#(adpcm_seq_item);

`uvm_object_utils(adpcm_tx_seq)

// ADPCM sequence_item
adpcm_seq_item req;

// Status – used to signal reset status
back to originating thread:
uvm_tlm_response_status_e status;

function new(string name = “adpcm_tx_seq”);
  super.new(name);
endfunction

task body;
  req = adpcm_seq_item::type_id::create(“req”);

  for(int i = 0; i < 10; i++) begin
    start_item(req);
    if(!req.randomize()) begin
      `uvm_error(“body”, “req randomization failure”)
    end
    finish_item(req);
    status = req.status;
    if(status == UVM_TLM_INCOMPLETE_RESPONSE)
    begin
      `uvm_warning(“body”, “Interface reset occurred”);
      return;
    end
  end
endtask: body

endclass: adpcm_tx_seq
```

Figure 3 - Reset aware sequence

Both drivers and monitors need to be able to detect a reset condition and they need to be able to cope with the consequences. For a driver this means that a protocol pin level transfer has to be abandoned part way through, and that the status field of the current sequence item has to be updated to indicate that a reset has been encountered. The driver also needs to clear down any other sequence items that might be in the seq_item_port FIFO, marking them with a reset status. With more advanced protocols with out of order responses, there may be several outstanding

```
task run_phase(uvm_phase phase);
  adpcm_init_seq init_seq = adpcm_init_seq::type_
  id::create(“init_seq”);
  adpcm_cfg_seq config_seq = adpcm_cfg_seq::type_
  id::create(“config_seq”);
  adpcm_tx_seq traffic_seq = adpcm_tx_seq::type_
  id::create(“traffic_seq”);
  state_e state = INIT;
  uvm_tlm_response_status_e status = UVM_TLM_
  GENERIC_ERROR_RESPONSE;

  init_seq.status = UVM_TLM_GENERIC_ERROR_
  RESPONSE;
  config_seq.status = UVM_TLM_GENERIC_ERROR_
  RESPONSE;
  traffic_seq.status = UVM_TLM_GENERIC_ERROR_
  RESPONSE;

  phase.raise_objection(this, “starting adpcm test”);
  forever begin
    case(state)
      INIT: begin
              init_seq.start(m_sequencer);
              if(init_seq.status == UVM_TLM_OK_
              RESPONSE) state = CONFIG;
            end
      CONFIG: begin
                config_seq.start(m_sequencer);
                if(config_seq.status == UVM_TLM_OK_
                RESPONSE)   state = TRAFFIC;
                else state = INIT;
              end
      TRAFFIC: begin
                 traffic_seq.start(m_sequencer);
                 if(traffic_seq.status == UVM_TLM_OK_
                 RESPONSE) break;
                 else state = INIT;
               end
    endcase
  end
  phase.drop_objection(this, “finished test_seq”);
endtask: run_phase
```

Figure 4 – Test level control thread which restarts the sequence if it returns with a reset status

sequence items queued up within the driver, and all of these will have to be terminated with a reset status. These requirements usually mean that the main driver loop has to

```
class adpcm_driver extends uvm_driver #(adpcm_seq_item);

`uvm_component_utils(adpcm_driver)

virtual adpcm_if ADPCM;
typedef enum {IDLE, DELAY, FRAME} state_e;

function new(string name = “adpcm_driver”, uvm_component
parent = null);
 super.new(name, parent);
endfunction

task run_phase(uvm_phase phase);
 int idx;
 adpcm_seq_item req;
 state_e state;

 forever begin
  ADPCM.frame <= 0; // Default conditions:
  ADPCM.data <= 0;
  @(posedge ADPCM.resetn); // Wait for reset to end
  forever
   begin
    @(posedge ADPCM.clk or negedge ADPCM.resetn);
    if(ADPCM.resetn == 0) begin // Reset detected:
     ADPCM.frame <= 0;
     ADPCM.data <= 0;
     state = IDLE;
     if(req != null) begin
      req.status = UVM_TLM_INCOMPLETE_RESPONSE;
      seq_item_port.item_done();
     end
     while(seq_item_port.has_do_available()) begin
      seq_item_port.get_next_item(req);
      req.status = UVM_TLM_INCOMPLETE_RESPONSE;
      seq_item_port.item_done();
     end
     break;
    end
    else begin
     case(state)
      IDLE: begin // Waiting for something to do
            if(seq_item_port.has_do_available()) begin
             seq_item_port.get_next_item(req);
             req.delay--;
             state = DELAY;
             ADPCM.frame <= 0;
            end
           end
      DELAY: begin // Wait for the specified no of clocks
            if(req.delay > 0) begin
             req.delay--;
            end
            else begin
             state = FRAME;
             idx = 0;
            end
           end
      FRAME: begin // Send the ADPCM frame
            if(idx < 7) begin
             ADPCM.frame <= 1;
             ADPCM.data <= req.data[3:0]; // Send nibbles
             req.data = req.data >> 4;
             idx++;
            end
            else begin
             ADPCM.data <= req.data[3:0];
             req.status = UVM_TLM_OK_RESPONSE;
             seq_item_port.item_done();
             req = null;
             state = IDLE;
            end
           end
     endcase
    end
   end
endtask: run_phase
endclass: adpcm_driver
```

Figure 5 - Reset aware driver implemented using FSM

be implemented with a state machine so that it can treat the reset condition as a temporary state.

The purpose of a monitor is to track pin level activity against the protocol and to generate transactions which record completed transfers. One important monitor design decision is whether to send out a transaction when a reset is detected, even though a protocol transfer has not completed. Arguably, if the monitor has to handle a reset, then reset is part of the protocol and it should generate and broadcast a transaction with reset status at the end of the reset. Whether the reset transaction should contain information on the transaction that was in progress when the reset occurred is determined by the protocol and how much information the monitor had extracted from the pin level activity before the reset. Like the driver, the monitor

```
class adpcm_monitor extends uvm_component;

`uvm_component_utils(adpcm_monitor)

typedef enum {WAIT_FOR_FRAME, IN_FRAME}
monitor_state_e;

uvm_analysis_port #(adpcm_seq_item) ap;

virtual adpcm_if ADPCM;

function new(string name = “adpcm_monitor”, uvm_
component parent = null);
 super.new(name, parent);
endfunction

function void build_phase(uvm_phase phase);
 ap = new(“ap”, this);
endfunction: build_phase

task run_phase(uvm_phase phase);
 adpcm_seq_item txn;
 adpcm_seq_item analysis_txn;
 int nibble_count;
 monitor_state_e state;

 forever begin
  @(posedge ADPCM.resetn); // Wait for reset to end
  if(txn != null) begin
   txn.status = UVM_TLM_INCOMPLETE_RESPONSE;
   ap.write(txn);
  end
  txn = adpcm_seq_item::type_id::create(“txn”);
  forever begin
   @(posedge ADPCM.clk or negedge ADPCM.resetn);
   if(ADPCM.resetn == 0) begin // Reset detected
    state = WAIT_FOR_FRAME;
    break;
   end
   else begin
    case(state)
     WAIT_FOR_FRAME: begin
                      if(ADPCM.frame) begin
                       txn.data = 0; // Initialise the data
                       txn.data[3:0] = ADPCM.data;
                       state = IN_FRAME;
                       nibble_count = 1;
                      end
                     end
     IN_FRAME: begin
               if(ADPCM.frame) begin
                if(nibble_count > 7) begin
                 `uvm_error(“run_phase”, “Too many
nibbles in ADPCM frame”)
                end
                case(nibble_count)
                 1: txn.data[7:4] = ADPCM.data;
                 2: txn.data[11:8] = ADPCM.data;
                 3: txn.data[15:12] = ADPCM.data;
                 4: txn.data[19:16] = ADPCM.data;
                 5: txn.data[23:20] = ADPCM.data;
                 6: txn.data[27:24] = ADPCM.data;
                 7: txn.data[31:28] = ADPCM.data;
                endcase
                nibble_count++;
               end
               else begin
                state = WAIT_FOR_FRAME;
                txn.status = UVM_TLM_OK_RESPONSE;
                assert($cast(analysis_txn, txn.clone()));
                ap.write(analysis_txn);
                txn.data = 0;
               end
              end
    endcase
   end
  end
 end

endtask: run_phase

endclass: adpcm_monitor
```

Figure 6 - Reset aware monitor

will most likely have to be implemented using a state machine in order to handle resets.

Analysis components such as scoreboards and functional coverage monitors also need to be designed to handle resets. What a scoreboard should do in the case of a reset is determined by the behavior of the DUT. For instance, if the DUT flushes all of its data buffers on a reset, then the scoreboard will need to do the same, or it may mark outstanding transactions as flushed to check that the DUT has flushed its data. If recovery from an on the fly reset is important for certain conditions, then functional coverage monitors can be made reset aware by checking the state of the design when it entered reset.

Event	Initiated by	Recommended Approach
Interrupt	DUT	Implement a context switch in the stimulus generation, no disruption to protocol transfers. See: http://verificationacademy.com/uvm-ovm/Stimulus/Interrupts
Error	DUT	Should be signalled as an error status, stimulus should have error handling embedded
Power Down	Testbench	Originated by the stimulus process, an orderly transition with protocol level transactions completing normally before entering power-down state.
Power Loss	DUT (Hardware)	Sudden loss of power is like an interrupt and should cause a context switch to enter a power-down state.

HANDLING OTHER HARDWARE EVENTS

An asynchronous reset is probably the most disruptive type of hardware event that can happen to a DUT. Other types of hardware events can be handled in other ways or by adapting the techniques outlined, the exact requirements are dependent on device behavior and the nature of the event. The most common forms of disruptive hardware events are summarized in the table above.

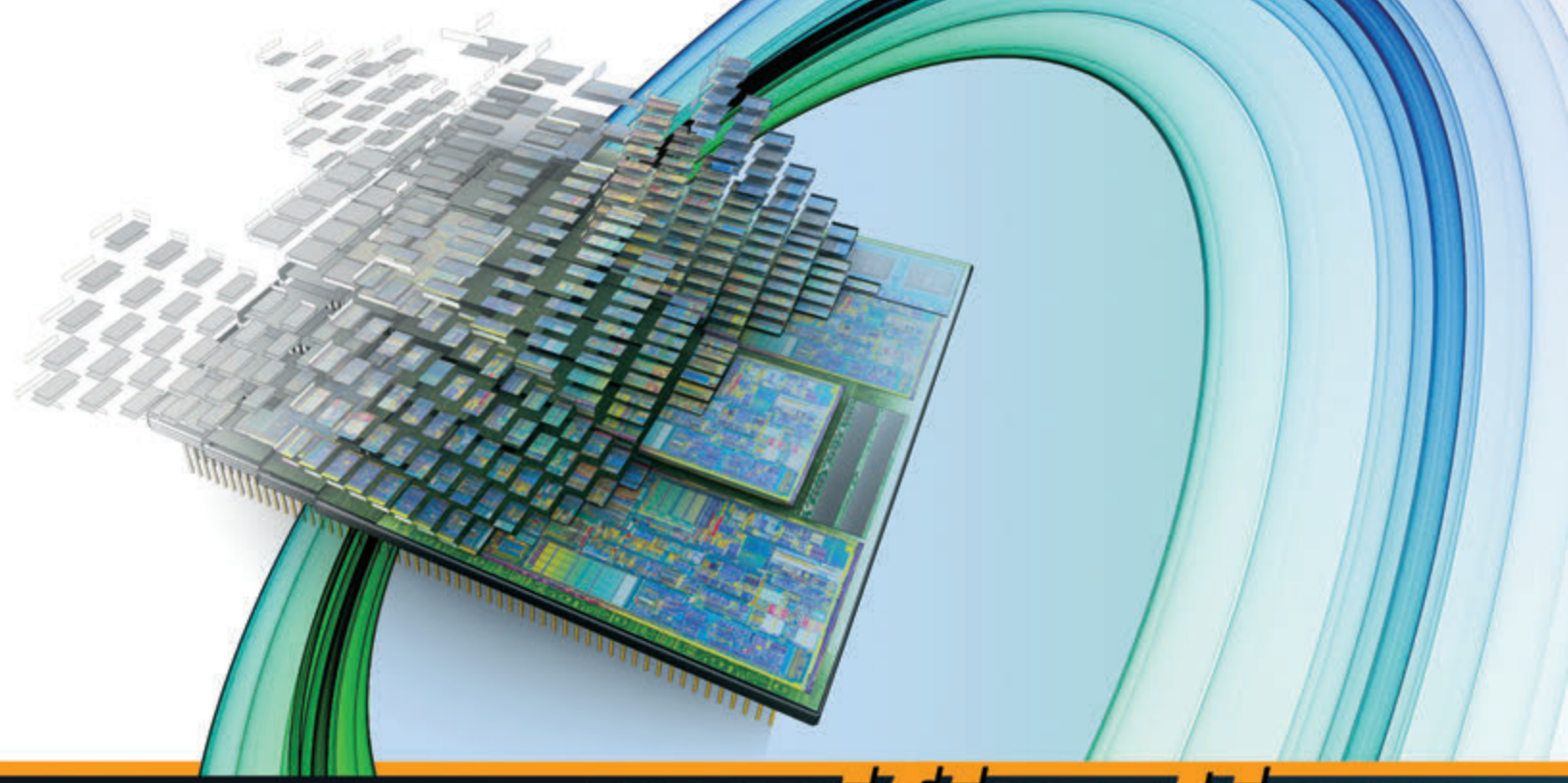
CONCLUSION

On the fly resets can be made manageable within a UVM testbench by building reset awareness into the agent, the sequence item and sequences. If the driver detects a reset condition, then it can bring the current transaction to an early close and return reset status through the sequence hierarchy to ensure a managed change in the stimulus generation path. This approach works well in the situation when all the testbench components have been implemented in this way, but if there are components that ignore reset, then they will need to be handled separately, allowing currently running sequences to come to halt of their own accord and dealing with the consequences.

As the reset aware technique does not rely on multiple run-time phases, it can also be used with OVM testbenches.

FURTHER EXAMPLES

The code examples shown in this article have been taken from a relatively simple unidirectional agent, other types of agent with bidirectional and pipelined protocols require a similar implementation approach. For more details and code examples please refer to the UVM cookbook available on the Verification Academy.



verification HORIZONS

Editor: Tom Fitzpatrick
Program Manager: Rebecca Granquist

Wilsonville Worldwide Headquarters
8005 SW Boeckman Rd.
Wilsonville, OR 97070-7777
Phone: 503-685-7000

To subscribe visit:
www.mentor.com/horizons

To view our blog visit:
VERIFICATIONHORIZONSBLOG.COM

Mentor Graphics®

www.mentor.com