Scalable Multi-Domain, Multi-Variant Reset Management in Verification IPs

Kaustubh Kumar, SiliConch Systems
Munnangi Sirisha, SiliConch Systems
Lokesh Kumar, SiliConch Systems
Why Reset Handling in VIPs?

• Coordination of reset in various VIP components:
  – Termination and reboot of all processes/task executions in the concerned Drivers and Monitors
  – Re-initializing all concerned flags, variables and data elements in the BFM s, such as counters, status flags, etc.
  – Clearing all sequence item/transaction queues/FIFOs/TLM ports to/from any VIP component involved
  – Flushing off all pending FIFO elements in the scoreboards and checkers
When to Handle Reset in VIPs?

• Initial DUT and VIP reset
  – At simulation time 0 OR
  – After a random initial delay

• Multiple asynchronous resets during simulation
What Resets to Handle in VIPs?

• Global vs Domain-specific resets
  – Performing reset to a subset of VIP components
  – Reset Porting during porting of VIP to higher level environment

• Variants of resets
  – Protocol based “COLD”/ “WARM” resets
  – Clock gating and un-gating
  – Differentiation of all reset variants and corresponding actions
Identification of Reset Domains

- Identify different domains for Reset operations.
- Categorize components as:
  - Reset Master (RM)
  - Reset Master and Slave (RMS)
  - Reset Slave (RS)

Reset relationship of the components (Master-Slave) **may NOT** be the same as **UVM hierarchy**
Database of Reset Domains

- Reset relationships can be visualized as a *database* of multiple reset domains.
- Reset Domain := Queue {Reset Master, Associated Reset Slaves}
- Index to the *database* is the *Domain ID* for that reset domain.

<table>
<thead>
<tr>
<th>DOMAIN ID</th>
<th>RESET MASTER</th>
<th>RESET SLAVES</th>
</tr>
</thead>
<tbody>
<tr>
<td>“DID_0”</td>
<td>C1</td>
<td>C2 C5</td>
</tr>
<tr>
<td>“DID_1”</td>
<td>C2</td>
<td>C6</td>
</tr>
<tr>
<td>“DID_2”</td>
<td>C3</td>
<td>C4</td>
</tr>
</tbody>
</table>
Reset Interface

- Declare SV interface class

```verbatim
interface class reset_interface;
    pure virtual task do_reset(string reset_type= "");
endclass : reset_interface
```

- Any component qualified as RM/RMS/RS shall implement the interface class and override `do_reset()` task to define the custom implementation.
Reset Handler Class

- **Singleton** `reset_handler` class
- Centralized handler; no factory registration

```plaintext
class reset_handler extends uvm_component;

protected static reset_handler _m_reset_handler;

local function new (string name, uvm_component parent);
    super.new(name, parent);
endfunction : new

static function reset_handler get(uvm_component parent);
    if(_m_reset_handler == null) begin
        _m_reset_handler = new("reset_handler", parent);
    end
    return _m_reset_handler;
endfunction : get
endclass : reset_handler
```
Reset Handler Class

• Database of reset domains represented as *associative array*:
  – *Domain ID (string type)* is the index of the array
  – Queue of `reset_interface` handles represent each element of the array. Each handle points to the Reset Master/Slave component.

```plaintext
class reset_handler extends uvm_component;

protected static reset_handler _m_reset_handler;
typedef reset_interface t_reset_if_q[$];

// Database declarations
protected t_reset_if_q m_reset_db [string] ;
protected bit m_master_sts_db [string] ;
...
...
endclass : reset_handler
```

Database of reset domains
Reset Handler Operations (R-A-N-C)

- **Registration**: Registration of Reset Master and Slave components to Reset Handler.
- **Asserton**: Servicing Assertion of reset request by a registered Reset Master.
- **Notification**: Notification of asserted reset request to the registered Reset Master/Slaves or Slaves.
- **Completion**: Monitoring for the Completion of the notified reset requests.
Operation 1 – REGISTRATION
Operation 1 – Registration

• Reset Master/Slave registration can be done using `reset_handler.register()` function in the UVM `build_phase()` of the respective components.

```verbatim
class C1 extends uvm_component implements reset_interface;
    `uvm_component_utils(C1)
    reset_handler m_reset_handler;

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        m_reset_handler = reset_handler::get(this);
        m_reset_handler.register( .is_master ( 1'b1 ),
                                .h_component ( this ),
                                .domain_id ( this.get_name()) );
    endfunction : build_phase
endclass : C1
```
Operation 1 – Registration

- Example of Reset Slave registration:

```verilog
class C2 extends uvm_component implements reset_interface;
  `uvm_component_utils(C2)
  reset_handler m_reset_handler;

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    m_reset_handler = reset_handler::get(this);
    m_reset_handler.register(.is_master(1'b0),
                            .h_component(this),
                            .domain_id("C1"));
  endfunction : build_phase

  virtual task do_reset(string reset_type = "");
    //implementation details
  endtask : do_reset
endclass : C2
```

Registering C2 slave with Domain ID = “C1”

Overriding do_reset as C2 implements reset_interface
Operation 2 – ASSERTION
Operation 2 – Assertion

- Reset Master can assert reset operation using `reset_handler.assert_reset()` function whenever needed.

```verilog
class C1 extends uvm_component implements reset_interface;
    `uvm_component_utils(C1)
    reset_handler m_reset_handler ;

    virtual task run_phase(uvm_phase phase);
        ...
        m_reset_handler.assert_reset( .domain_id (this.get_name()),
                                    .h_component (this ),
                                    .slaves_only (1 )
        );
        ...
    endtask : run_phase
endclass : C1
```

Asserting reset for “C1”
Domain ID only for slaves
Operation 3 – NOTIFICATION

REGISTRATION → ASSERTION → NOTIFICATION → COMPLETION
Operation 3 – Notification

• When an `assert_reset()` call is done from a Reset Master, the `reset_handler` validates the Reset Master handle and locally updates a list of all active Reset Domains that have been requested reset.

• In `run_phase()` task, `reset_handler` performs the following operations:
  – Waits for the list of *active* Reset Domains to contain at least one element.
  – Gets the queue of `reset_interface` handles of all the RM, RMS and RS components in the reset domain requested.
  – Parallelly, calls `do_reset()` task of all the RM (if `slaves_only = 0`), RMS and RS components.
class reset_handler extends uvm_component;

// Local Databases
protected t_reset_if_q m_reset_db [string] ;
protected bit m_master_sts_db [string] ;
...
protected bit reset_sts ;
protected string active_domain_list[$];

function void assert_reset(string domain_id, ...);
  reset_sts = 1 ;
  active_domain_list.push_back(domain_id);
endfunction : assert_reset
...

Bit to notify reset assertion
List of active reset domains
Update active domain list on reset assertion
virtual task run_phase(uvm_phase phase);
    forever begin
        wait(reset_sts == 1);
        while(active_domain_list.size > 0) begin
            automatic string curr_domain =
                active_domain_list.pop_front();
            if(m_reset_db.exists(curr_domain)) begin
                foreach(m_reset_db[curr_domain][i]) begin
                    fork begin
                        automatic int comp_id = i;
                        m_reset_db[curr_domain][comp_id].do_reset(...);
                    end
                end
                end join_none
            end
            reset_sts = 0;
        end
eendtask : run_phase
eendclass : reset_handler

Loop over all active reset domains
Loop over all elements of current domain
do_reset() call for the RM/RMS/RS in requested reset domain
Operation 4 – COMPLETION
Operation 4 – Completion

• Once all the `do_reset()` calls are parallelly triggered through multiple `fork...join_none` threads, it becomes mandatory for `reset_handler` to track all the launched threads and ensure all of them are completed.

• Reset Master may wait for reset completion using a separate task `reset_handler.wait_reset_done()` since `assert_reset()` call is non-time consuming.

• To track all the parallelly launched `do_reset()` threads, `process IDs` of all the threads are recorded at the launch time. After all the threads are launched, the recorded process IDs are used to wait for threads to get finished.
class reset_handler extends uvm_component;
  // Local Databases
  protected t_reset_if_q m_reset_db [string] ;
  ...
  protected bit reset_done_sts [string];

  function void assert_reset(string domain_id, ...);
    // Update initial status of requested domain
    reset_done_sts [domain_id] = 0;
  endfunction : assert_reset

  task wait_reset_done(string domain_id);
    wait(reset_done_sts[domain_id] == 1);
  endtask : wait_reset_done

Stores reset completion status of each reset domain

Used by master for waiting on reset completion on requested domain

© Accellera Systems Initiative
virtual task run_phase(uvm_phase phase);
  forever begin
    while (active_domain_list.size > 0) begin
      automatic process pid_q [$];
      if (m_reset_db.exists(curr_domain)) begin
        foreach (m_reset_db[curr_domain][i]) begin
          fork begin
            m_reset_db[curr_domain][comp_id].do_reset(...);
            pid_q.push_back(process::self());
          end join_none
        end
        fork
          foreach (pid_q[i]) begin
            wait (pid_q[i] != null);
            pid_q[i].await();
          end
          reset_done_sts[curr_domain] = 1;
        end
      end
    end
  endtask : run_phase

Reset Notification

Recording process IDs of all launched threads

Reset Completion

Waiting for all launched threads to complete using the recorded process IDs

Updating reset completion status
Beyond Plain Reset Handling

- Modeling clock gating and un-gating using `do_suspend()` and `do_resume()` in a similar way as `do_reset()`.
  - Reset Handler provides `assert_suspend()` and `assert_resume()` functions for the master to assert clock gating/un-gating.

- Multi-variant reset management based on protocol by using `reset_type string` in `assert_reset()`.
Multi-Variant Reset Assertion

- Protocol based distinguished reset can be asserted using `reset_type` string argument in `assert_reset()` function.

```verbatim
class C1 extends uvm_component implements reset_interface;
  `uvm_component_utils(C1)
  reset_handler m_reset_handler;
  virtual task run_phase(uvm_phase phase);
    m_reset_handler.assert_reset(.domain_id (this.get_name()),
      .h_component (this),
      .slaves_only (1),
      .reset_type ("COLD_RESET")
    );
  endtask : run_phase
endclass : C1
```

Asserting “COLD” reset from master C1
Multi-Variant Reset Handling

- Since, the Reset Slaves are protocol dependent, the `do_reset()` task in the Reset Slave shall be implemented to handle all the reset types according to the protocol.

```verilog
class C2 extends uvm_component implements reset_interface;
  `uvm_component_utils(C2)
  reset_handler m_reset_handler;

  virtual task do_reset(string reset_type = "");
    case(reset_type)
      "COLD_RESET" : begin
        // Handling of "COLD RESET" feature
        end
    endcase
  endtask : do_reset
endclass : C2
```
Conclusion and Summary

- A major requirement in Verification IPs (VIPs) is to handle different types of resets which may vary from protocol to protocol.

- This paper presents a scalable, protocol independent and a centralized way to handle reset assertion and execution throughout the VIP.

- Reset handler works on the principle of Master and Slave Relationship.

- Reset Handler also provides simple and standard interface for handling other operations such as clock gating and un-gating.
Questions?