2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

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

Kaustubh Kumar, Verification Architect, SiliConch Systems, Bengaluru, India
(*k.kaustubh@siliconch.com*)

Munnangi Sirisha, Verification Architect, SiliConch Systems, Bengaluru, India
(*sirisha.munnangi@siliconch.com*)

Lokesh Kumar, Verification Lead, SiliConch Systems, Bengaluru, India
(*lokesh@siliconch.com*)

Abstract— **Reset Modeling in verification IPs is a crucial part of functional verification and its complexity increases with the architecture complexity. Reset testing might vary from resetting the complete environment (global reset) to resetting only a set of components (domain-specific reset) in the environment. With the growing complexity of protocols and design features, multiple variations of resets and other power-related features need to be tested and modeled in verification IPs as well. A standardized, scalable and protocol-independent approach is needed to handle all the possible complex scenarios to achieve a well synchronized reset across the testbench. This paper will present our technique of standardizing the reset management by deploying a centralized UVM-based Reset Handler which handles and manages all the possible reset conditions at both levels, global as well domain specific. This paper will also present the extensibility of the proposed technique to handling different kinds of reset using the proposed Reset Handler.**

Keywords—reset management; domain-specific resets; reset variations; reset master-slave relationship; asynchronous reset handling

I. INTRODUCTION

A major requirement in Verification IPs (VIPs) is to handle dynamic resets, targeted to a certain set of VIP components or to the entire VIP. Modeling resets in the verification IPs is a crucial part of functional verification and its complexity increases with the architecture complexity. Reset testing might vary from resetting the complete environment (global reset) to resetting only a set of components (domain-specific reset) in the environment. With the growing complexity of protocols and design features, multiple variations of resets and other power-related features need to be tested and modeled in verification IPs as well.

This paper presents a scalable, protocol independent and a centralized way to handle reset assertion and execution throughout the VIP. This standardization is achieved by deploying a centralized UVM-based `reset_handler` component which handles the reset assertion conditions and invokes the concerned VIP entities to execute the reset operation. Some of the reset-based scenarios common in VIPs, that are expected to be handled by `reset_handler`, are mentioned below:

1) **VIP Initial Reset/Global VIP Reset:** Initial reset of VIP components is required to put the logical threads/variables/interfaces in their reset state. This initialization step may happen at simulation time 0 or any non-zero time instant as well to introduce reset delay randomization. Again, based on protocol/scenario requirements, entire VIP environment might be required to undergo reset operation dynamically/asynchronously.

2) **Domain-specific Reset:** Based on the protocol under verification, only a subset of all the VIP components might be required to undergo reset operation at any time instant. These subsets of VIP components constrained to undergo reset together, form a reset-domain.

3) **Modes/Variants of Reset:** VIP might be required to undergo different varieties of reset operations, for example, “COLD” and “WARM” resets. The VIP components undergoing the reset operation in this case may expect an input about the mode of reset in order to execute the operation differently, as per protocol requirements.

Apart from “reset” functionality alone, `reset_handler` functionality can be extended to execute operations such as disable, clock gating, clock un-gating, etc.

The proposed technique declares `reset_handler` as a centralized *singleton* class, which works on a principle of Master-Slave relationship. In this relationship, a VIP component responsible for asserting reset(s) to other component(s) is defined as the *Reset Master* (abbreviated as RM). The VIP component(s) which are invoked because of reset assertion by an RM is/are defined as *Reset Slave*(s) (abbreviated as RS). Based on the VIP architecture, some of the components can serve as both master and slave (RMS). Hence, the various groups of RMs and RSes/RMSes together form the multiple reset-domains. A reset operation execution in a reset-domain can be confined to that domain alone, without impacting other reset-domains and without changing the phase of any of the UVM components.

Existing technique of UVM phase jumping [3] to `reset_phase()` is suitable for resetting all the components in the environment but cannot be used where only a set of components in a verification environment need to be reset. It is also difficult to handle different types of resets that can applied to the components using UVM phase jumping approach as it requires proper coordination between the components. Our approach solves these problems by creating multiple reset-domains and handling different modes of reset.

Another technique [4] involves plugging a virtual interface to monitor reset assertion and using a `uvm_thread` instance as *reset_handler* and calling *reset_handler*.`notify(ACTIVATE)` to assert reset. The disadvantage of this technique is the inability to handle domains-specific reset using a single instance of `uvm_thread`. Also, this technique is not scalable to different modes of reset.

II. Identification of Reset Domains

The proposed `reset_handler` can manage Global VIP Reset or Domain-specific Reset. This is achieved by using Reset Master and Slaves relationship with optional Reset Master/Slaves in between (RM-*RMS*-RS). Creating multiple-reset domains in a VIP environment employs following initial steps:

1) Identifying the RM components as VIP components which can assert reset based on any *simulation event*, such as a unique sequence item, trigger of a UVM event/callback, etc. At this step, it is possible that a Reset Master may qualify for reset assertion based on multiple simulation events as well as standalone. In Figure 1 below, components `C1` and `C3` are identified as Reset Masters.

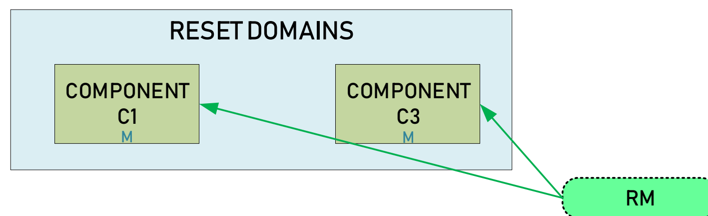


Figure 1. Identifying Reset Master (RM) components

2) Identifying the VIP components which depend on another *reset event* triggered by a Reset Master (RM) to assert reset to a set of VIP components and labeling them as Reset Master/Slaves (RMS). RMS components may undergo reset operation apart from issuing resets to another set of components. In Figure 2 below, component `C2` is identified as an RMS component.

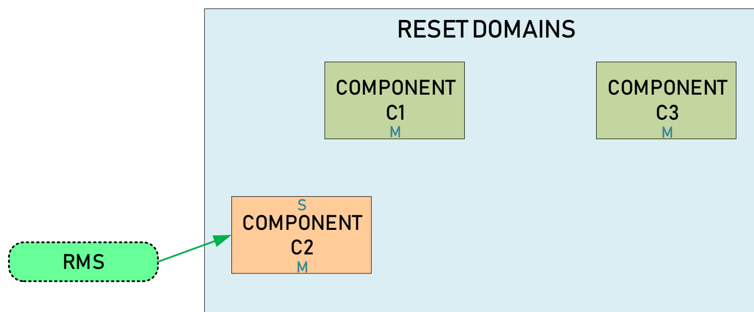


Figure 2. Identifying Reset Master/Slave (RMS) components

3) Identifying the VIP components which undergo reset operation based on *reset event* triggered by an RM/RMS and labeling them as Reset Slaves (RS). These components are the endpoints of a reset assertion hierarchy and issue no further resets to any other component. In Figure 3 below, components `C4`, `C5` and `C6` are identified as Reset Slaves.

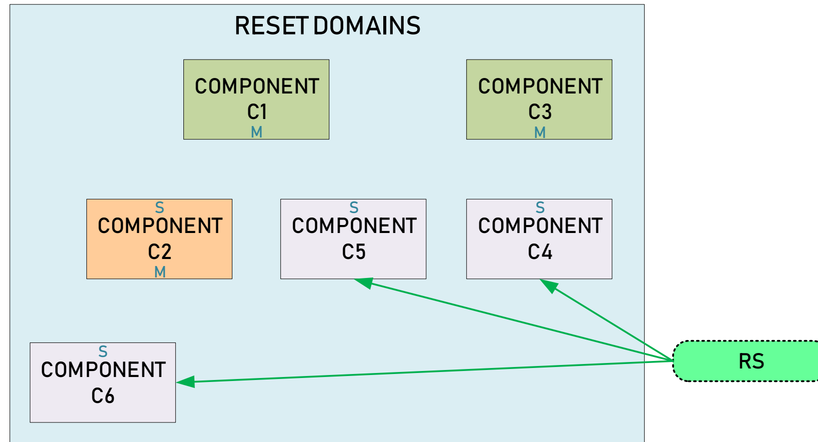
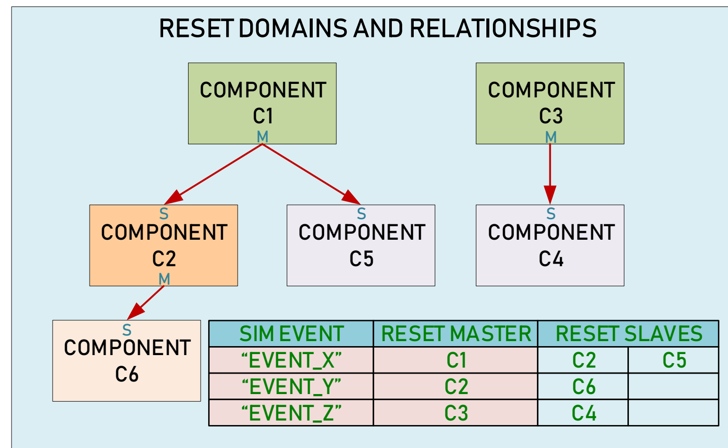


Figure 3. Identifying Reset Slave (RS) components

4) Link each of the RMs to the respective RMSes and RSes into separate reset-domains based on every *simulation event* which can cause the RM to assert reset operation. In case the RM asserts reset to the same set of RMSes/RSes for multiple *simulation events*, group all the *simulation events* into a single reset-domain. Figure 4 below shows the final reset hierarchy of the components in a sample VIP environment.



SIM EVENT	RESET MASTER	RESET SLAVES	
"EVENT_X"	C1	C2	C5
"EVENT_Y"	C2	C6	
"EVENT_Z"	C3	C4	

Figure 4. Linking Reset Master and Slaves based on simulation events

Hence, there can be multiple Reset Masters (RMs) in a VIP environment, each of them associated with one or many reset domains. This relationship can be visualized as a database of a Reset Master associated with its Reset Master/Slaves or Reset Slaves, the index to this database being the unique *simulation event* that causes the RM to assert reset.

III. RESET INTERFACE

A System Verilog `interface` class `reset_interface` is declared comprising of `pure virtual` System Verilog task `do_reset()`. Any VIP component qualified as an RM/RMS/RS shall **`implement`** the `reset_interface` and hence override the `do_reset()` task to specify the details of handling of the reset operation based on the protocol implementation.

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

```
endclass : reset_interface
```

Figure 5. Declaration of `reset_interface`

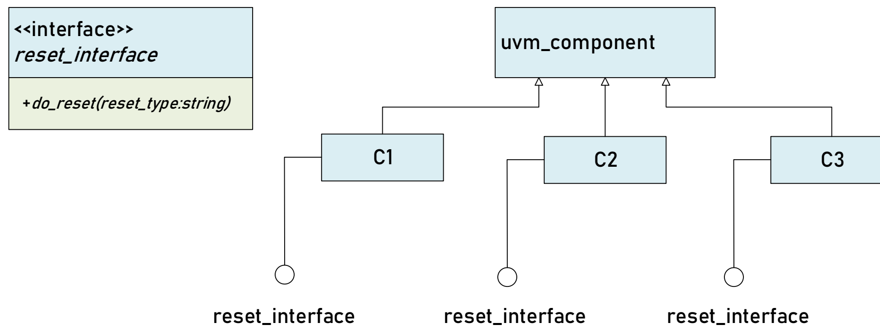


Figure 6. UVM hierarchy for RM/RMS/RS components

IV. RESET HANDLER OPERATION

Major operations handled by `reset_handler` is abbreviated as R-A-N-C (Registration-Assertion-Notification-Completion). Each of these operations are summarized in below points and described in detail in further sub-sections:

1) **R**egistration of Reset Master and Slave components to the `reset_handler` database
2) Servicing **A**ssertion of reset request by a registered Reset Master
3) **N**otification of asserted reset request to the registered Reset Master/Slaves or Slaves.
4) Monitoring for the **C**ompletion of the notified reset requests

Below code in Figure 7 shows the `reset_handler` class declaration. Note that since `reset_handler` is a singleton class, there is no factory registration done, as factory overrides are not required.

```
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
```

Singleton handle

Inaccessible by external classes

Accessible by external classes

Figure 7. Reset Handler singleton class definition

A. *Master and Slave Registration*

The database of RMs, RMSs, RSs indexed by *simulation event* can be easily represented by associative array. The `reset_handler` maintains the associative array, with following properties:

1) **Domain ID** (*`string`* type) as the index of the array; string representation of the *simulation event*.
2) Each element of the array is a System Verilog queue of type `reset_interface`. Each element of the queue is a `reset_interface` handle to the RM/RMS/RS associated with the Reset Domain. First element of the queue is the RM of the Reset Domain, followed by the RMSs and RSs associated with the domain.

Another database to maintain status of Reset Master registration is also maintained by `reset_handler`.

DOMAIN_ID	QUEUE OF RESET INTERFACE HANDLES
"DID_0"	{C1, C2, C5}
"DID_1"	{C2, C6}
"DID_2"	{C3, C4}

DOMAIN_ID	MASTER REGN STATUS
"DID_0"	1
"DID_1"	1
"DID_2"	1

Figure 8. Reset Handler local databases

```
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
```

Figure 9. Reset Handler singleton class definition and databases

Each of the VIP components associated with the Reset Domains registers itself with the `reset_handler`. This registration can be done in the UVM `build_phase()` of the VIP components, where the singleton handle of the `reset_handler` can be received and `reset_handler.register()` function called. There are two modes of `register()` function call, each for an RM and an RS. For RMS components, two `register()` function calls are required, one as an RM, another as an RS. Following are the inputs arguments for `register()` function:

1) `is_master (bit)`: Single-bit flag to specify whether the VIP component being registered is a Reset Master (`is_master = 1`) or a Reset Slave (`is_master = 0`).
2) `h_component (uvm_component)`: Handle of the VIP component being registered.
3) `domain_id (string)`: String representation of the Domain ID of the VIP component being registered. Based on implementation, string representation of master name with optional suffixes can also be used as Domain ID.

```
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);
        // Retrieve reset_handler singleton handle and register the component
        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

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

endclass : C1
```

Registering C1 master with Domain ID = "C1"

Overriding `do_reset` task as `C1` implements `reset_interface`

Figure 10. Registering a Reset Master component with Reset Handler

```
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);
        // Retrieve reset_handler singleton handle and register the component
        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"

Figure 11. Registering Reset Slave component with Reset Handler

The registration process is order independent, implying any RS can register itself before the RM registration. Order independence is mandatory as registering a component as RM/RS is done in UVM `build_phase()` and reset hierarchy can be different from the UVM hierarchy of the components. In other words, the order in which the VIP components are built in `build_phase()` may not be the same as an RM to RS relationship/hierarchy. This might cause a problem as there may be chances of a Reset Domain having no registration of a Reset Master, as there is no such compulsion in `build_phase()`.

To guarantee that all Reset Domains have a Reset Master registered, the `reset_handler` validates the associative array queues and master registration status database in the UVM `connect_phase()`. In the validation process, the `reset_handler` checks and flags errors, if found, in the following cases:

1) A Reset Slave does not exist for a Reset Master in a domain
2) A Reset Domain has no Reset Master registered

B. Reset Assertion

When a Reset Master needs to assert reset operation to its associated slaves, it uses the `reset_handler` handle to call `reset_handler.assert_reset()` function. `assert_reset()` is a System Verilog function and takes following input arguments:

1) `domain_id (string):` String name of the reset domain for which reset is being asserted
2) `h_component (uvm_component):` Handle of the Reset Master component from which reset is being asserted.
3) `reset_type (string):` String representation of the type/mode of the reset being asserted. This is an optional argument.
4) `slaves_only (bit):` Single bit to specify whether the reset operation must be performed on the Reset Slaves only (`slaves_only=1`) or on Reset Master as well as associated Reset Slaves (`slaves_only=0`). This is an optional argument, and the default value for `slaves_only` is 0.

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

    virtual task run_phase(uvm_phase phase);
        …
        …
        //Below code to assert reset to the associated Reset Slave(s)
        m_reset_handler.assert_reset( .domain_id    (this.get_name()),
                                      .h_component  (this          ),
                                      .slaves_only  (1             )
         );
```

```
        …
        …
    endtask : run_phase

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

endclass : C1
```

Asserting reset for "C1" domain ID only for slaves

Figure 12. Reset assertion from Reset Master

C. Reset 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 following operations:

1) Waits for the list of active Reset Domains (Reset Domains that have been requested reset) to contain at least one element.
2) Gets the queue of `reset_interface` handles of all the RMs, RMSs and RSs in the reset domain requested
3) Parallelly, calls `do_reset()` task of all the RMs (if `slaves_only` = 0), RMSes and RSes. Since the number of parallel threads initiating `do_reset()` calls depends on the number of VIP components registered in the domain, the number of such threads is variable for each reset assertion. To handle triggering of variable number of threads each time reset is asserted, `fork…join_none` can be used inside a loop over all queue elements. Initiating parallel `do_reset()` calls is mandatory, as sequential calls may violate the purpose of achieving the simultaneous resets of all targeted VIP components.

Note that the `assert_reset()` function does not directly call `do_reset()` per VIP component, but instead just updates a list of active Reset Domains. This is necessary as `assert_reset()` must be non-time consuming so that the Reset Master can proceed ahead without consuming any simulation time or blocking other operations while waiting for reset completion.

```
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 ; // bit to notify reset assertion
    protected string active_domain_list[$]; //queue to store active reset domains

    function void assert_reset(string domain_id, …);
        reset_sts = 1 ;
        active_domain_list.push_back(domain_id);
        // Add code to record slaves_only option
    endfunction : assert_reset

    virtual task run_phase(uvm_phase phase);
        forever begin
            wait(reset_sts == 1);

            // Loop over all active reset domains
            while(active_domain_list.size > 0) begin
                automatic string curr_domain = active_domain_list.pop_front();
                if(m_reset_db.exists(curr_domain)) begin
                    // Loop over all the elements of the curr_domain
```

```
                foreach(m_reset_db[curr_domain][i]) begin
                    fork
                        automatic int comp_id = i ;
                        m_reset_db[curr_domain][comp_id].do_reset(…);
                    join_none
                    …
                end
            end
            else begin
                // Throw Error Message
            end
        end
        reset_sts = 0 ;
    end
endtask : run_phase

endclass : reset_handler
```

do_reset() call for the RM/RMS/RS components in the requested reset domain

Figure 13. Handling reset notification to RM/RMS/RS components in the requested Reset Domain

D. *Reset 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 and notify the completion. 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. Reset completion is notified when `await()` calls on all the process IDs are completed.

```
class reset_handler extends uvm_component;

    // Local Databases
    protected t_reset_if_q m_reset_db [string] ;
    …
    protected bit reset_done_sts [string]; // Stores reset completion status of
each reset domain

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

    // wait_reset_done task: to be used by master for waiting on reset completion
    task wait_reset_done(string domain_id);
        wait(reset_done_sts[domain_id] == 1);
    endtask : wait_reset_done

    virtual task run_phase(uvm_phase phase);
        forever begin
            …
            // Loop over all active reset domains
            while(active_domain_list.size > 0) begin
                automatic process pid_q [$];
                …
                if(m_reset_db.exists(curr_domain)) begin
                    // Loop over all the elements of the curr_domain
                    foreach(m_reset_db[curr_domain][i]) begin
                        fork
                            …
```

task to wait for reset completion of the requested reset domain

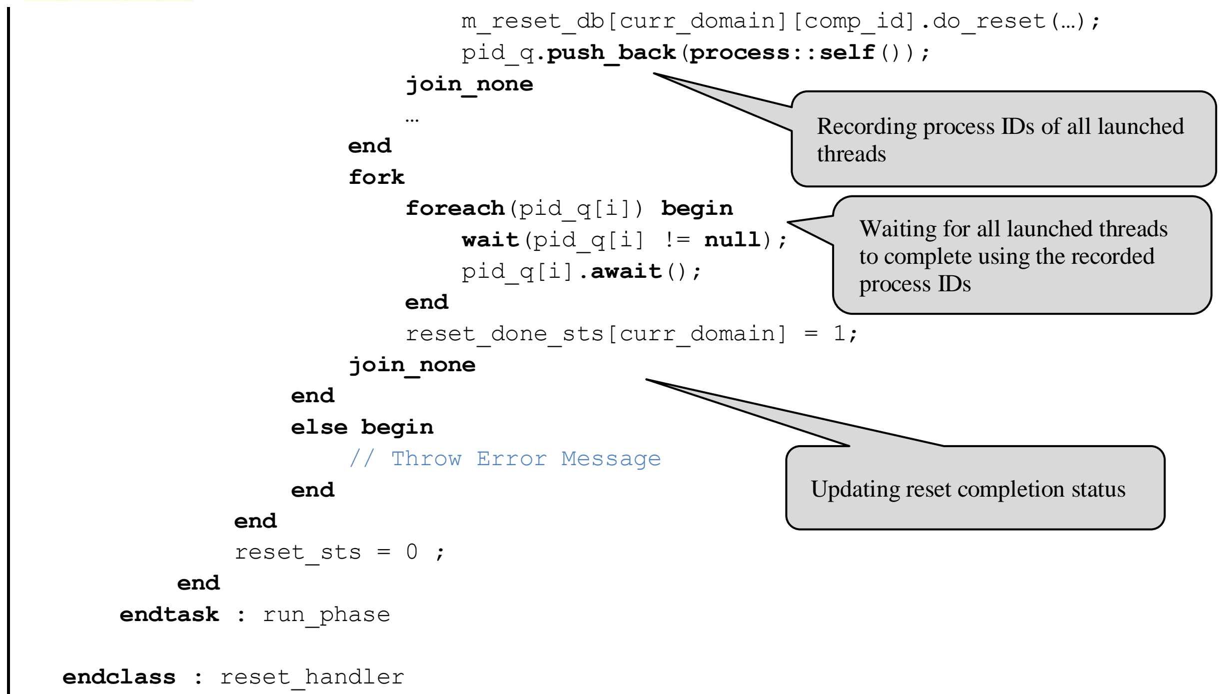


Figure 14. Monitoring for Reset Completion of Active Reset Domains

V. Beyond Plain Reset Handling

A. *Clock Gating and Un-Gating*

When verifying a design with clock gating features, the VIP environment might also be required to model clock-gating of a set of VIP components. Like reset management, `reset_handler` can be used to manage clock-gate assertions and notifying respective VIP components. Note that a VIP component must employ implementation dependent mechanism to clock-gate its own processes when notified to do so by `reset_handler`. One of the ways that VIP component can use to clock-gate its own processes is to use `suspend()` calls on all its process IDs.

To achieve clock-gating management, two tasks are added to `reset_interface`: `do_suspend()` and `do_resume()`. These tasks shall be overridden in the VIP component which implements the `reset_interface`. When a Reset Master asserts clock-gating in a Reset Domain using `reset_handler.assert_suspend()` function, `reset_handler` calls `do_suspend()` on all the RMs, RMSes and RSes registered in the requested Reset Domain. Similarly, a Reset Master uses `reset_handler.assert_resume()` function call to issue `do_resume()` for the requested Reset Domain and mimics clock un-gating feature.

B. *Disable vs Reset*

Expanding the scope further, `reset_handler` can also be used to trigger disabling of a set of VIP components registered in a Reset Domain. A task `do_disable` can be added to the `reset_interface`. A VIP component implementing the `reset_interface` in this case, shall also override `do_disable` task to implement disabling of the processes in the component. The difference between `do_disable` and `do_reset` is that the component which is disabled does not start running the processes again automatically until a `do_reset` call is made to the component. On the other hand, `do_reset` call is intended to perform both, disabling of the processes in the component as well as re-starting and re-initialization of the processes.

C. *Multi-Variant Reset Management*

As mentioned earlier, `assert_reset()` call from a Reset Master has a string input argument `reset_type`. This argument can be used by a Reset Master to specify the mode/variant of reset to be issued to the Reset Slaves. For example, the argument `reset_type` can be set to `"COLD_RESET"` or `"WARM_RESET"`.

The reset_handler takes no action based on reset_type and directly forwards it to the Reset Slaves by passing the same argument to the do_reset() call for each Reset Slave. 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. Hence, it becomes simpler for a VIP component writer to handle different varieties of resets based on the string input argument of the do_reset() task.

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

    virtual task run_phase(uvm_phase phase);
        …
        …
        // Below code to assert "COLD" reset to the associated Reset Slave(s)
        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" for "C1" domain ID only for slaves

Figure 15. Asserting "COLD RESET" from Reset Master

```
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
             // Handle other variants of resets below
             …
             …
        endcase
    endtask : do_reset

endclass : C2
```

Figure 16. Handling "COLD RESET" in a Reset Slave

VI. Deployment in USB Power Delivery VIP

The proposed technique is deployed successfully in USB Power Delivery (PD) Verification IP environment and various possibilities like resetting sub-components with different types of resets have been extensively tested. The overall structure of environment for USB PD Verification IP and reset relationship is shown in Figure 17 below.

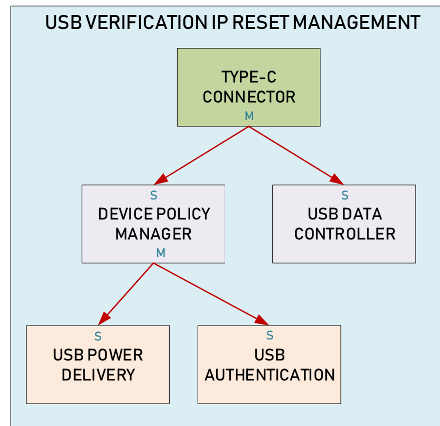


Figure 17. USB PD Verification IP Reset Hierarchy

This technique is also extended for disabling and suspending the sub-components apart from resetting. Also, it is used to handle SOFT_RESET and HARD_RESET variations in USB Power Delivery protocol. The failure debugs related to reset testing were faster and simpler using the proposed technique.

VII. Conclusion

This paper defines a standard flow which helps in performing dynamic reset operations and modeling reset management in a scalable, efficient and graceful way. Maintaining a central database eases the development of Reset Master/Slave components, which just need to define the handler tasks: `do_reset()`, `do_disable`, etc. Structuring the VIP with a centralized `reset_handler` also helps in faster debugs, modification of existing scenarios and adding new scenarios.

References

[1] Accellera, Universal Verification Methodology 1.2, June 2014.

[2] IEEE Computer Society, System Verilog, 2017.

[3] Brian Hunter, Ben Chen and Rebecca Lipon, "Reset Testing Made Simple with UVM Phases," in *Synopsys User Group*, 2013.

[4] Courtney Fricano, Stephanie McInnis, Uwe Simm and Phu Huynh, "Reboot your Reset Methodology: Resetting Anytime with the UVM Reset Package," in *Design and Verification Conference*, Europe, 2014.