Reboot your Reset Methodology: Resetting Anytime with the UVM Reset Package

Courtney Schmitt
Analog Devices, Inc.
Norwood, MA
courtney.schmitt@analog.com

Phu Huynh, Stephanie McInnis, Uwe Simm
Cadence Design Systems
San Jose, CA
phuynh@cadence.com, swaters@cadence.com,
uwes@cadence.com

***Abstract*— When verifying a device under test (DUT), a common requirement is to reset the design, restart the environment, and check to make sure that it comes out of reset without any problems. The primary purpose of a reset is to force the design into a known state for stable operations in a simulation or in the real DUT itself. It is necessary to verify that the design behaves as expected when going into and coming out of the reset state. The reset can be applied at the beginning of the verification process, which is quite common, but applying it anywhere else in the verification process can create problems if the verification environment was not designed to handle this scenario. In this paper we will describe a reset methodology that uses the UVM run_phase and works with the standard UVM library. This reset methodology consists of a reset package that provides a Reset Handler and two (2) additional run_phase APIs to allow the UVM components to gracefully shut the activities down when reset occurs, and to restart the components' normal activities after the reset goes away. The methodology and implementation also supports the situation where there are multiple reset regions. We implemented this package at ADI and found it easy to use with both new and existing testbench environments. The two additional classes were easily managed and we only had to make minimal modifications to our existing UVCs. The reset package and associated methodology documentation are available as an open source contribution on the Accellera UVM forums.**

Keywords— SystemVerilog; UVM; Reset Verification;

I. RESET VERIFICATION REQUIREMENTS

One of the important verification tasks for a design is to ensure that the DUT will come out of the reset correctly. It is also important to ensure that if a reset is active in the middle of a normal operation, the DUT will also shutdown gracefully and be able to restart when the reset goes away. To accomplish the reset verification task, additional requirements are imposed on the verification environment; specifically, the verification environment needs to do the following things when reset happens in the middle of the simulation:

- Activities and stimulus need to stop and possibly restart once the reset is de-asserted
- Assertions and checkers need to shut down gracefully
- Scoreboards and data structures need to be reset to proper initial values
- When reset goes away, the verification components need to restart from the initial state.

The current version of the UVM library (uvm-1.1d [1]) includes three (3) phases related to reset:

- pre_reset_phase()
- reset_phase()
- post_reset_phase()

However the current solution is still incomplete and has not yet been finalized. The reset phases are standardized but the activities to be performed by these phases have not been clearly defined yet. It is unclear how the above phases can be used to satisfy the reset verification requirements. Since the committee has still not finalized on all of the phasing components, we chose to use a valuable alternative.

II. CADENCE UVM_THREAD RESET METHODOLOGY

The reset methodology introduced in this paper uses the UVM run_phase(). It is designed to work with existing UVCs that have already been designed to handle reset at the start of the simulation. Our reset methodology relies on thread management instead of phase management, and it can be extended to handle multiple reset domains and different types of reset.

To use this reset methodology, you need the Cadence uvm_thread package which is available as an open source download from the Accellera UVM forums [2]. The reset package does not require any changes to the Accellera UVM base library, so it is compatible with any tools that support UVM. The uvm_thread package allows a UVM testbench to add the following components and capabilities:

- A **reset_handler**: this component is derived from a **uvm_thread** class (provided in the uvm_thread package); it keeps track of the UVM components that are resettable and manages the "run" threads of these components; specifically, when reset occurs, it will terminate the "run" threads of these components and call a **clean_up()** method to shut down the current operation gracefully and to re-initialize the internal data structure.
- A **reset_monitor**: this component is derived from an uvm_monitor class. This component monitors the RESET signal and has a handle to the **reset_handler**; when there is a change in the RESET signal, it will call the **notify()** method of the **reset_handler** with one of the following messages:
    - **TERMINATE**: when the RESET signal becomes active
    - **ACTIVATE**: when the RESET signal goes away
- Any UVM components that need to be resettable will need to be modified as follows:
    - Register themselves with the **reset_handler**
    - Implement two (2) new methods: **clean_up()** and **run_phase_new()**. The **clean_up()** method will be invoked by the **reset_handler** when reset occurs. The **run_phase_new()** will be invoked by the **reset_handler** when reset goes away. Any threads spawned by the **run_phase_new()**, including itself, will be terminated by the **reset_handler** when reset occurs.

Fig. 1 below illustrates how the uvm_thread package is used in a testbench and the interactions between the reset_monitor, the reset_handler, and the resettable components of a UVM testbench.

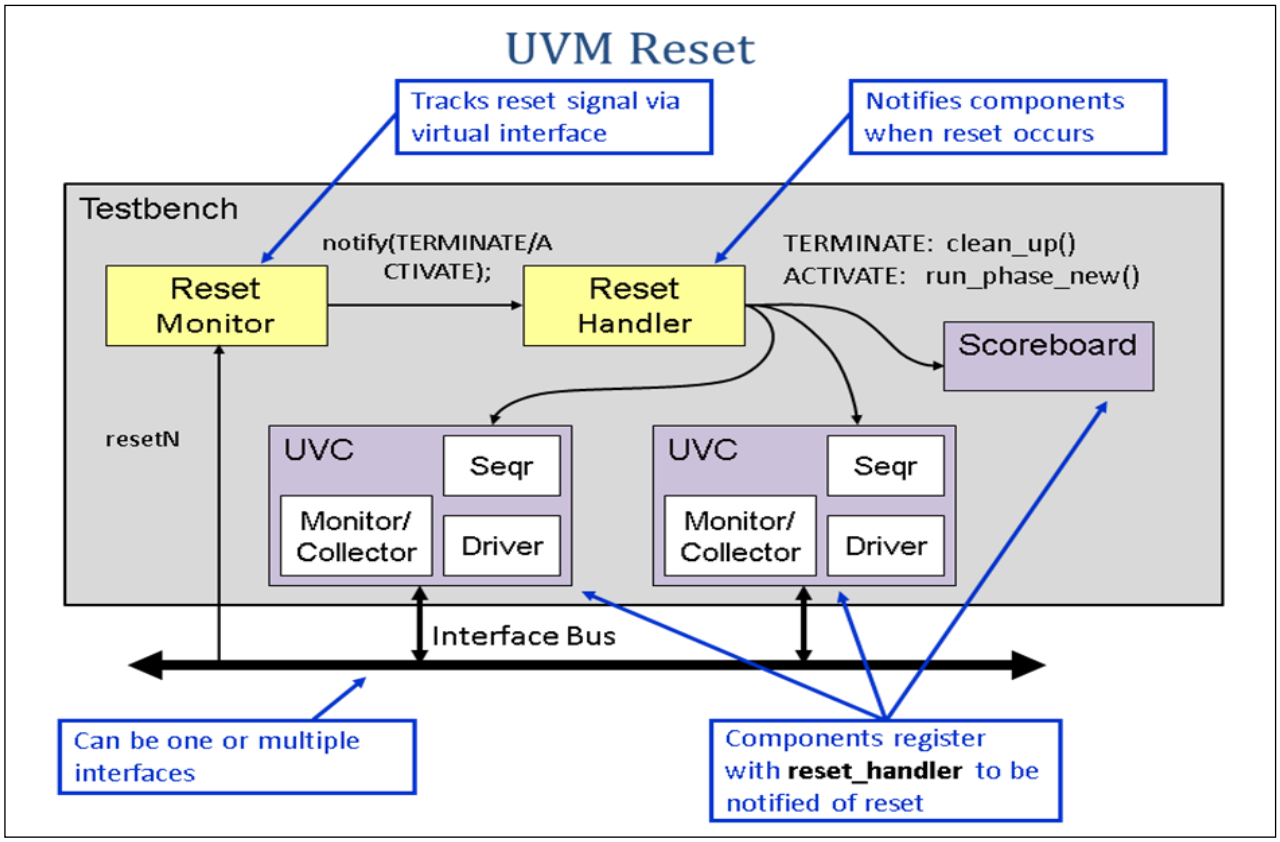

Fig. 1. Usage of the uvm_thread package

Fig. 2 below shows the relationship between reset (active low resetN signal), the UVM run_phase(), and the two (2) additional API methods: **clean_up()** and **run_phase_new()**.

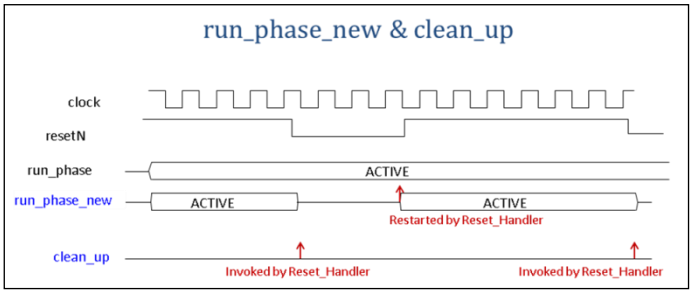

Fig. 2. run_phase_new and cleanup in relationship to reset and run_phase

III. Real-world Example – ADI Verification Environment

This section shows how the uvm_thread package was used to handle reset in an UVM environment at Analog Devices, Inc. (ADI).

In our experience, reset is usually an afterthought that occurs during the middle of the verification effort. At that stage it can be very difficult to modify existing code to handle the reset. We often end up having to compromise on the reset verification just to get something running. In order to get reset running in past projects we have turned off checks during reset, created a separate testbench just to handle reset, or created directed tests designed specifically to test reset. These modifications generally worked, but this custom reset code usually had problems in higher system-level testbench environments. Our past reset verification methodology was far from ideal and required a lot of extra work to get running. In an effort to improve our reset verification, we implemented the Cadence reset package on one of our existing block level testbenches.

We chose to use a System Oscillator testbench environment to evaluate the Cadence reset package. This environment provides a main system clock which is calculated using a crystal oscillator as the input. The design uses a watchdog timer to take advantage of the high-precision crystal oscillator to detect faults on the input clock. An active low, "rstb" disables the block (see Fig. 3 below).

We followed some stringent verification requirements for the System Oscillator:

- Check CLKO frequency based on the crystal frequency
- Activate various fault conditions and check that they are detected correctly. Some examples of that could be missing transitions or having a frequency above or below the spec.
- Check that the block shuts down gracefully when reset is asserted.
- Check that the block restarts correctly when reset is de-asserted.

The testbench architecture for the Watchdog Oscillator used UVM to communicate with the Oscillator and the Fault Checker (see Fig. 4 below).

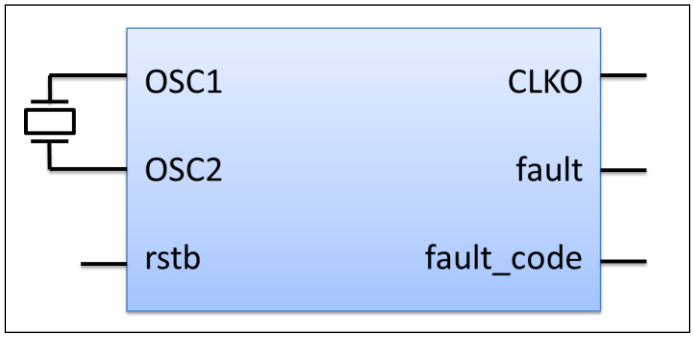

Fig. 3. System Oscillator with Watchdog

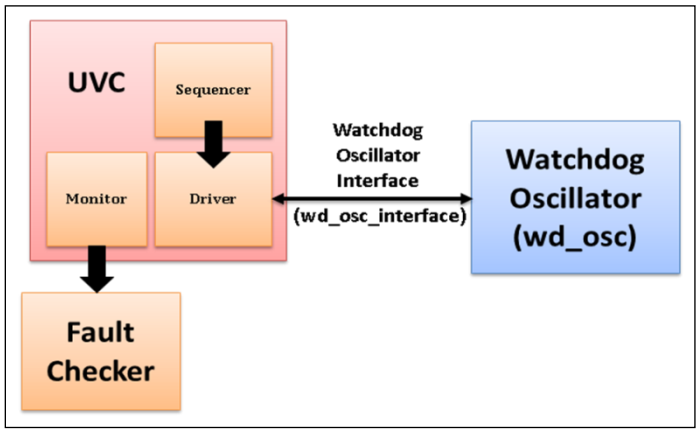

Fig. 4. Testbench Architecture for Watchdog Oscillator

In a UVM environment, an active agent would contain a sequencer, a driver, and a monitor. The driver takes transactions from the sequencer and translates them into a form that the DUT can understand (in this case, the Watchdog Oscillator). It then passes that transaction to the DUT via an interface (in this case, wd_osc_interface). Optionally it may wait for a response to appear on the interface and send that data back to the sequencer. The stimulus used to generate a transaction in a UVM environment is called a sequence. A Universal Verification Component (UVC) typically contains at least one, but in many cases, multiple agents, and the testbench contains multiple UVCs.

We were able to easily edit our existing testbench code to implement the Cadence reset package. Below we will show examples of the changes we made.

We first edited the testbench environment class (wd_osc_env) to add a pointer to the reset handler and the reset monitor. We also had to register the reset_monitor and the reset handler with the factory in the build phase. Finally, we stored the handle to the reset_handler in the config_db (see Fig. 5 below).

```
class wd_osc_env extends uvm_env;
  wd_osc_agent agent_wd_osc;
  wd_osc_scoreboard wd_osc_sb;
  virtual wd_osc_interface vif;

  reset_monitor reset_mon;                  // Pointer to reset_monitor
  uvm_thread reset_handler;                 // Pointer to reset_handler

function void build_phase(uvm_phase phase);
  …
// Register reset_mon and reset_handler with the factory
reset_mon = reset_monitor::type_id::create("reset_mon", this);
reset_handler = uvm_thread::type_id::create("reset_handler", this);

// Store the handle to reset_handler in the config_db
 uvm_config_db#(uvm_thread)::set(this, "*", "reset_handler",
reset_handler);

endfunction : build_phase
…
```

Fig. 5. Testbench Environment Class Modifications

Fig. 6 below shows the reset_monitor class. This class monitors the reset signal contained in the virtual interface, "reset_if" and notifies the reset handler when a reset occurs.

```
class reset_monitor extends uvm_monitor;
 virtual reset_if vif;
 uvm_thread reset_handler;

  `uvm_component_utils_begin(reset_monitor)
   `uvm_field_object(reset_handler, UVM_DEFAULT |
UVM_REFERENCE)
 `uvm_component_utils_end

function void connect_phase(uvm_phase phase);
  uvm_config_db#(virtual reset_if)::get(this, "", "vif", vif)
  uvm_config_db#(uvm_thread)::get(this, "", "reset_handler",
reset_handler)
 endfunction : connect_phase

virtual task run_phase(uvm_phase phase);
  @(vif.resetN);

  if (vif.resetN) begin
    reset_handler.notify(ACTIVATE);
    forever begin
      @(negedge vif.resetN)        reset_handler.notify(TERMINATE);
      @(posedge vif.resetN)        reset_handler.notify(ACTIVATE);
    end
  end else begin
    reset_handler.notify(TERMINATE);
    forever begin
      @(posedge vif.resetN)        reset_handler.notify(ACTIVATE);
      @(negedge vif.resetN)        reset_handler.notify(TERMINATE);
    end
  end
 endtask : run_phase

endclass : reset_monitor
```

Fig. 6. Reset Monitor Class

It was also necessary to make some modifications to the existing testbench UVC. We edited the sequencer, driver, monitor, and test classes to become reset-aware. These classes were all connected to the reset_handler so they could track the reset status. We also added the "clean_up" function as well as the "run_phase_new" task which are invoked by the reset_handler when reset is activated or terminated.

When reset is asserted, the sequencer stops sending transactions and the driver stops driving crystal oscillator input onto OSC1 and OSC2. In order to do this, the "clean_up" implementation is enforced. A call to "stop_sequences()" will stop all of the currently active sequences. If an object was raised in a sequence, it will use "do_kill()" to drop the objection.

When reset is de-asserted, the sequencer starts sending new transactions and the driver resumes driving OSC1 and OSC2. If it is a "warm reset", it will restart the default sequence.

Fig. 7 below shows the changes that were needed for the sequencer class and Fig. 8 shows the changes that were necessary for the driver class. Notice that the "clean_up" function is invoked when reset is terminated, and the "run_phase_new" task is invoked when reset goes away.

```
class wd_osc_sqr extends uvm_sequencer #(wd_osc_data);
  uvm_thread reset_handler;
  local uvm_thread_imp#(wd_osc_sqr) reset_export;
  local uvm_phase run_phase_handle;
  local bit first_run = 1;

  function new(string name, uvm_component parent);
    reset_export = new("reset_export", this);
  endfunction : new

  virtual function void connect_phase(uvm_phase phase);
    uvm_config_db#(uvm_thread)::get(this, "", "reset_handler",
reset_handler
    reset_handler.register(reset_export,
uvm_thread_pkg::default_map);
  endfunction : connect_phase
endclass

 virtual task run_phase(uvm_phase phase);
    run_phase_handle = phase;
    super.run_phase(phase);
  endtask : run_phase

  virtual task run_phase_new(uvm_phase phase);
    if (!first_run)    start_phase_sequence(run_phase_handle);
    else               first_run = 0;
  endtask : run_phase_new

  virtual function void clean_up();
     stop_sequences();
  endfunction : clean_up

endclass : wd_osc_sqr
```

Fig. 7. Testbench Environment Class Modifications : Sequencer

```
class wd_osc_drv extends uvm_driver #(wd_osc_data);
  uvm_thread reset_handler;
  local uvm_thread_imp#(wd_osc_drv) reset_export;

 function new(string name, uvm_component parent);
    reset_export = new("reset_export", this);
 endfunction : new virtual

 function void connect_phase(uvm_phase phase);
     uvm_config_db#(uvm_thread)::get(this, "", "reset_handler",
reset_handler)
     reset_handler.register(reset_export,
uvm_thread_pkg::default_map);
  endfunction

  virtual function void clean_up();
    `uvm_info(get_type_name(),"Starting clean_up...",UVM_NONE)
  endfunction : clean_up

  virtual task run_phase(uvm_phase phase);
  endtask : run_phase

  virtual task run_phase_new(uvm_phase phase);
    super.run_phase(phase);
    get_and_drive();
  endtask : run_phase_new

endclass : wd_osc_drv
```

Fig. 8. Testbench Environment Class Modifications : Driver

Fig. 9 below show the changes that were necessary for the monitor class and Fig. 10 shows the changes for the base_test class. Finally, Fig. 11 below shows the changes that were necessary for the basic_test class.

```
class wd_osc_mon extends uvm_monitor;
  virtual interface wd_osc_interface vif;
  uvm_thread reset_handler;
  local uvm_thread_imp#(wd_osc_mon) reset_export;

  function new(string name, uvm_component parent);
    super.new(name,parent);
    reset_export = new("reset_export", this);
  endfunction : new

 virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
     uvm_config_db#(uvm_thread)::get(this, "", "reset_handler",
reset_handler)
    reset_handler.register(reset_export,
uvm_thread_pkg::default_map);
  endfunction : connect_phase

  virtual function void clean_up();
    data_fifo.empty();
  endfunction : clean_up

endclass : wd_osc_mon
```

Fig. 9. Testbench Environment Class Modifications : Monitor

```
class base_test extends uvm_test;
  wd_osc_tb tb;
  uvm_thread reset_handler;

  virtual function void build_phase(uvm_phase phase);
    …
    reset_handler = uvm_thread::type_id::create("reset_handler",
this);
    uvm_config_db#(uvm_thread)::set(this, "*", "reset_handler",
reset_handler);
  endfunction : build_phase

endclass : base_test
```

Fig. 10. Testbench Environment Class Modifications : base_test

```
class basic_test extends base_test;
  local uvm_thread_imp#(basic_test) reset_export

  function new(string name, uvm_component parent);
     super.new(name,parent);
     reset_export = new("reset_export", this);
  endfunction : new

  virtual function void connect_phase(uvm_phase phase);
    uvm_config_db#(uvm_thread)::get(this, "", "reset_handler",
reset_handler))
    reset_handler.register(reset_export,
uvm_thread_pkg::default_map);
  endfunction : connect_phase

  task run_phase_new(uvm_phase phase);
    fork
      wd_osc_seq.start(tb.env.agent_wd_osc.sequencer);
    join
 endtask : run_phase_new

  function void clean_up();
    `uvm_info("COMPONENT","CLEANING",UVM_NONE)
  endfunction : clean_up

endclass : basic_test
```

Fig. 11. Testbench Environment Class Modifications : basic_test

The code examples in this section show that it was very easy to implement the reset package in our existing UVM environment. The waveform window in Fig. 12 below shows the simulation results for the watchdog oscillator with the integrated reset package. The waveforms show that when reset was asserted (the first set of red circles), the sequencer stopped sending transactions and the driver stopped driving the crystal oscillator inputs onto OSC1 and OSC2. Then when reset was de-asserted (the second set of circles), the sequencer started sending new transactions and the driver resumed driving OSC1 and OSC2. These results confirm that the reset package is behaving as expected and correctly controlling the design and testbench components.

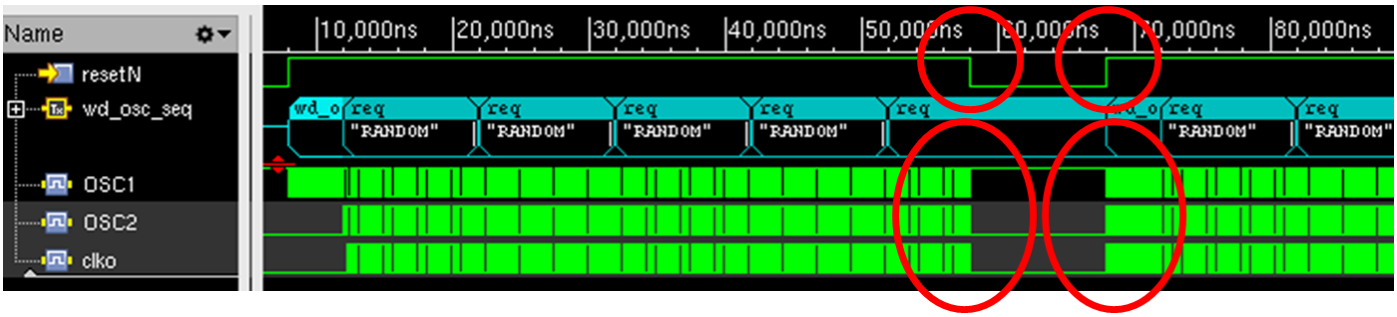

Fig. 12. Reset Operation Waveform

IV. EXTENDING TO AN SOC

Today's SoCs integrate many subsystems, including processor cores, memory, graphics, audio, networking, and various other I/O subsystems. Depending on the functional requirements, multiple resets may be needed to reset various subsystems of an SoC. We can easily extend the use of the reset methodology to handle multiple resets in an SoC.

At the system level, you would still have a reset monitor as we did at the block level. This monitor will contain a pointer to the reset_handler for each subsystem that wants to track their reset functionality. As illustrated in Fig. 13 and Fig. 14, the reset monitor has handles to multiple reset handlers (reset_handler_1, reset_handler_2, etc.). The Reset Monitor will call notify(ACTIVATE/TERMINATE) for the corresponding reset handler when its reset signal changes state. The notify call for each subsystem block will then inform that subsystem that a reset has been activated or terminated. If reset was not active, it would invoke run_phase_new() and if it was active, it would terminate all of the threads spawned by run_phase_new() and invoke the clean_up() method, the same way it behaved at the block level.

As you can see, the reset methodology is very flexible and can easily be extended to manage multiple resets at the block level or at the system level.

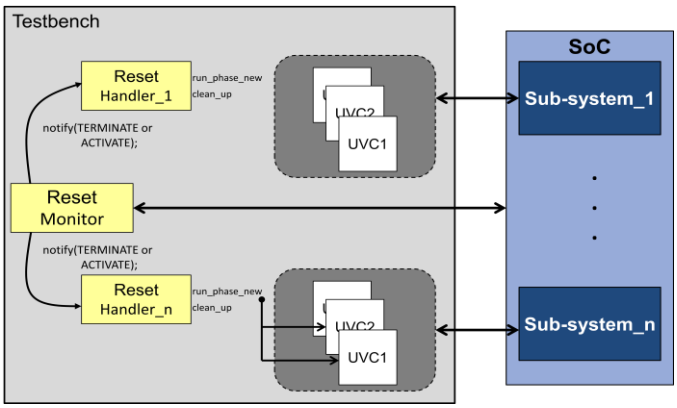

Fig. 13. Handling Multiple Reset Regions

```
class reset_monitor extends uvm_monitor;
  virtual reset_if vif;
  uvm_thread reset_handler_1,
             reset_handler_2;
  …
 virtual task run_phase(uvm_phase phase);
   fork
     begin
       // monitor vif.reset_1 and
       // notify(ACTIVATE |TERMINATE) reset_handler_1
     end
     begin
       // monitor vif.reset_2 and
       // notify(ACTIVATE |TERMINATE) reset_handler_2
     end
   join
 endtask
```

Fig. 14. Multiple Resets – Reset Monitor

V. CONCLUSIONS

The Cadence reset package has provided a valuable methodology for adding reset verification to a UVM environment. During the evaluation at ADI, we found that it is flexible, extendible, and works with the standard UVM library source code. The reset package described in this paper provides a standard reset methodology which can be used across teams, projects, and companies. It also allows for code reuse and is easy to implement in both new and existing UVM testbench environments with minimal modifications to existing verification components.

To find the Cadence reset methodology package as well as several example testbenches, please visit the Accellera community website at the following link. The relevant code is available under the user contributions area.

http://www.accellera.org/community/uvm/

REFERENCES

[1] Accellera UVM Class Library Code for Release 1.1d
[2] uvm_thread package - in UVM Contribution Area
[3] IEEE 1800-2012 SystemVerilog LRM