2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

A 360 Degree View of UVM Events (A Case Study)

Deepak Kumar E V, Sathish Dadi, Vikas Billa

elitePLUS Semiconductor Technologies Pvt Ltd

Bangalore, Karnataka, India.

elitePLUS SEMICONDUCTOR

2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Agenda

- Introduction
- System Verilog event, uvm_event and uvm_event_pool
- Case Study
  - Reset Aware Testbench
  - RAL
  - Interrupt Mechanism
  - Callbacks
- Conclusion
- References

2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Introduction

- In a verification environment, the test-bench components often communicate in a synchronized manner, to effectively implement the time accurate checks.
- System Verilog events are dedicated and are widely used data types to achieve the desired synchronized communication between the components.
- UVM library has a built-in dedicated class around System Verilog events which has broaden the application and usage of event based communication.
- This paper is a collective case study of projects, highlighting the usage and benefits of uvm_event.

System Verilog & UVM events

A System Verilog event is a data type which has no storage. It can be triggered using the "**->**" operator, and an event triggering occurrence can be captured by using "@" operator or inbuilt .triggered method.

Declaration:
event sv_event;

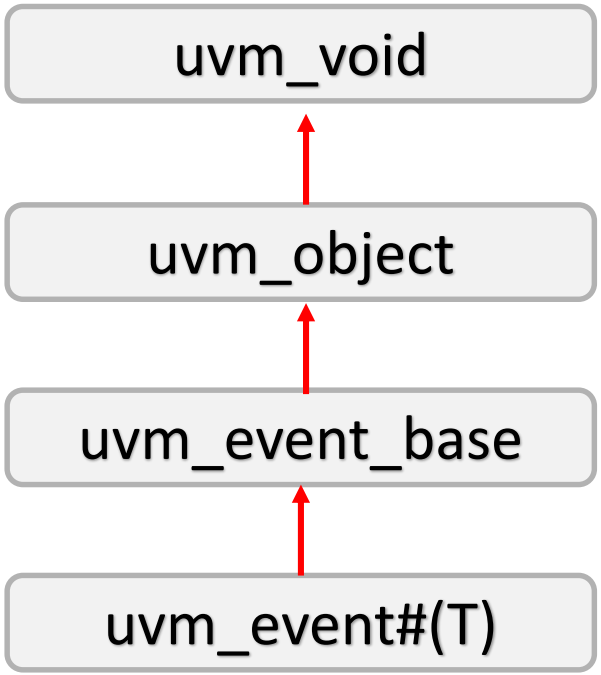
Triggering an event
->sv_event

Waiting for an event:
@sv_event
prone to race conditions

Persistent Trigger:
wait(sv_event.triggered)
eliminate common race conditions.

uvm_event

- Hierarchy



- Signature

  class ***uvm_event*** #(type T=uvm_object) extends ***uvm_object***;

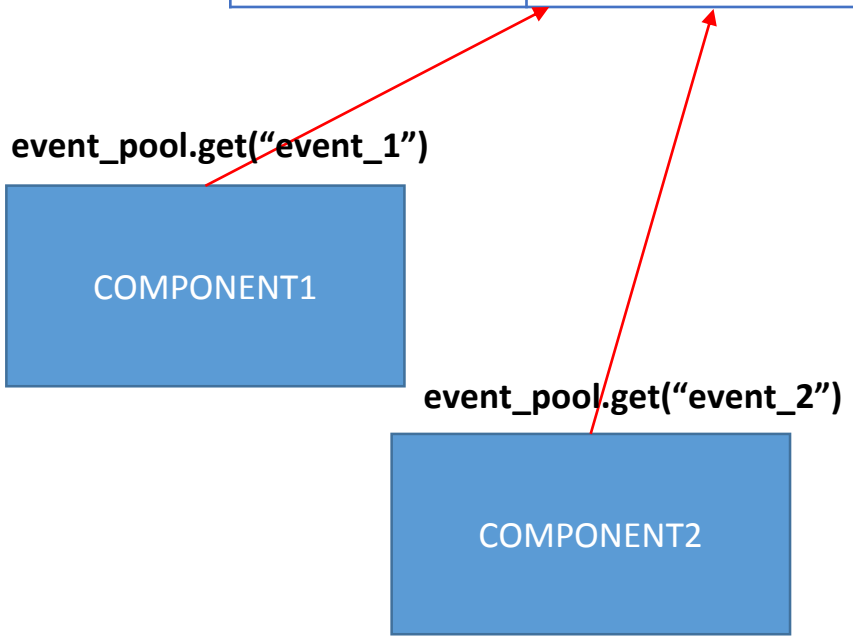
uvm_event

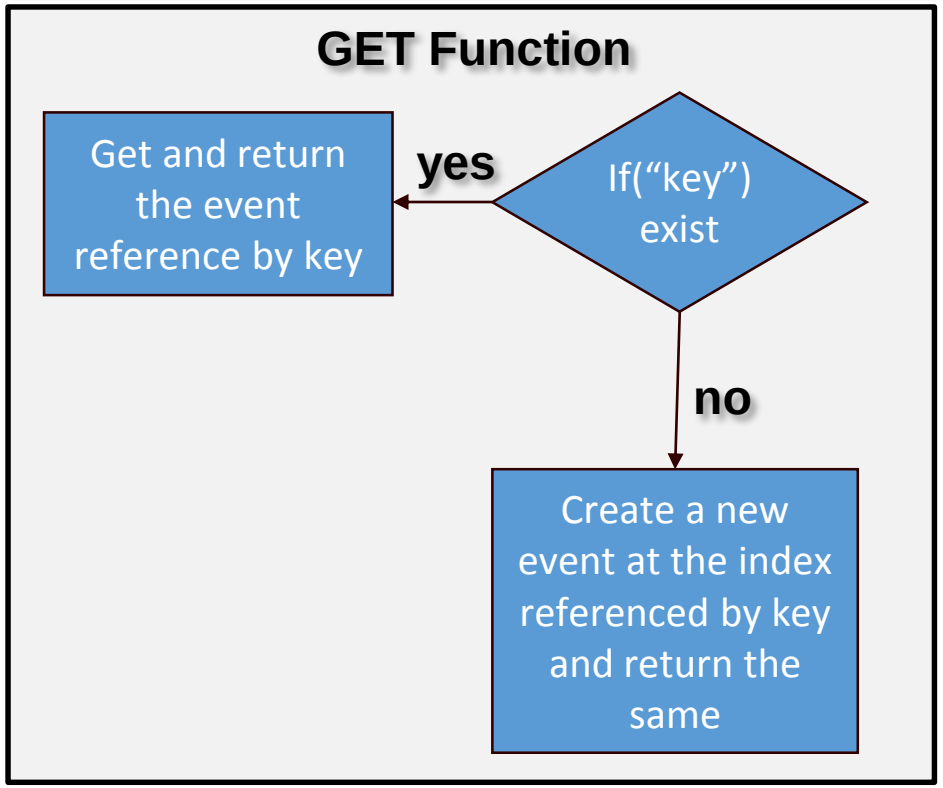
Frequently used Methods	Description
trigger(T data = null)	Trigger an event.
wait_trigger()	Waiting for an event. A system Verilog equivalent of @ operator.
wait_ptrigger()	Waiting for a persistent trigger. A system Verilog equivalent of .triggered. Avoids any race condition.
get_trigger_data()	Gets data if any provided by the last call to trigger.
wait_trigger_data(output T data)	Call and returns get_trigger_data method value followed by wait_trigger.
wait_trigger_pdata(output T data)	Call and returns get_trigger_data method value followed by wait_ptrigger.

uvm_event_pool

String Key	uvm_event
“event_1”	event_1_h
“event_2”	event_2_h
“event_3”	event_3_h
“event_4”	event_4_h

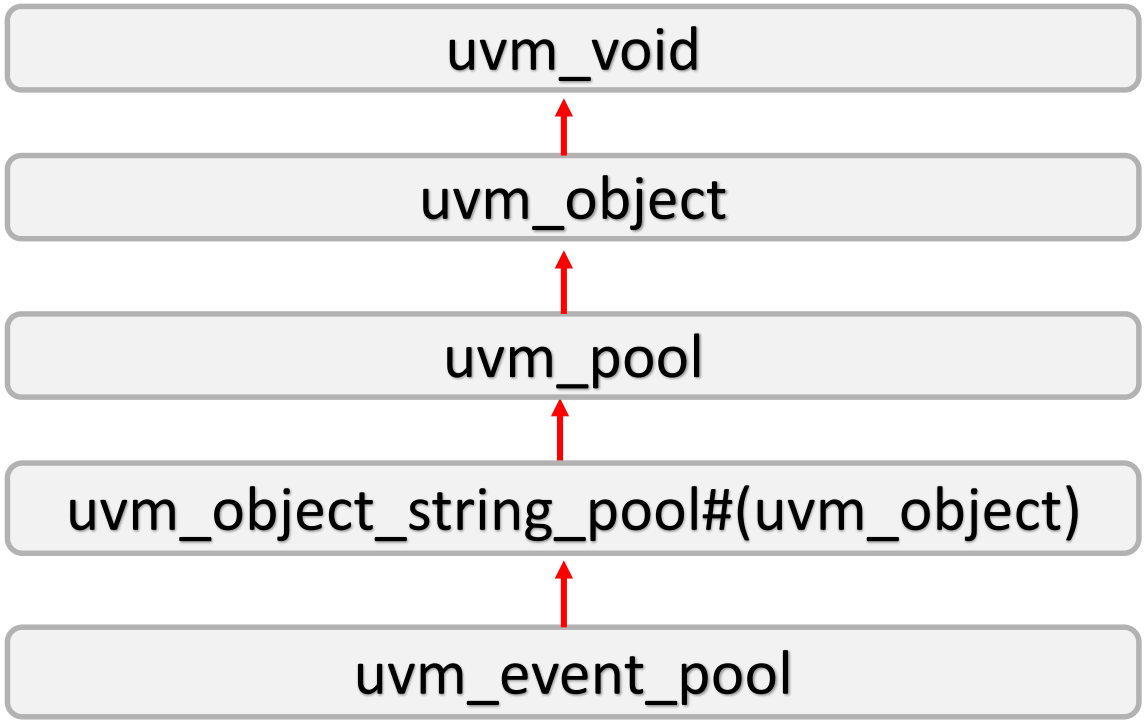
To get the handle to event_pool
uvm_event_pool event_pool;
event_pool = uvm_event_pool::get_global_pool();





uvm_event_pool

- uvm_event_pool is a singleton class built around an associative array of uvm_events indexed by string.
- Hierarchy



uvm_event_pool

- uvm_pool class is a wrapper class built around an associative array(pool)
- The pool associative array can be accessed through the key, by default key is of type int

```
class uvm_pool #(type KEY=int, T=uvm_void) extends uvm_object;
  const static string type_name = "uvm_pool";
  typedef uvm_pool #(KEY,T) this_type;
  static protected this_type m_global_pool;
  protected T pool[KEY];
```

- uvm_object_string_pool extends from the uvm_pool with fixed string type key

```
class uvm_object_string_pool #(type T=uvm_object) extends
uvm_pool #(string,T);
```

- uvm_event_pool extends from uvm_object string with uvm_event as array elements;

```
typedef uvm_object_string_pool #(uvm_event) uvm_event_pool;
```

uvm_event_pool

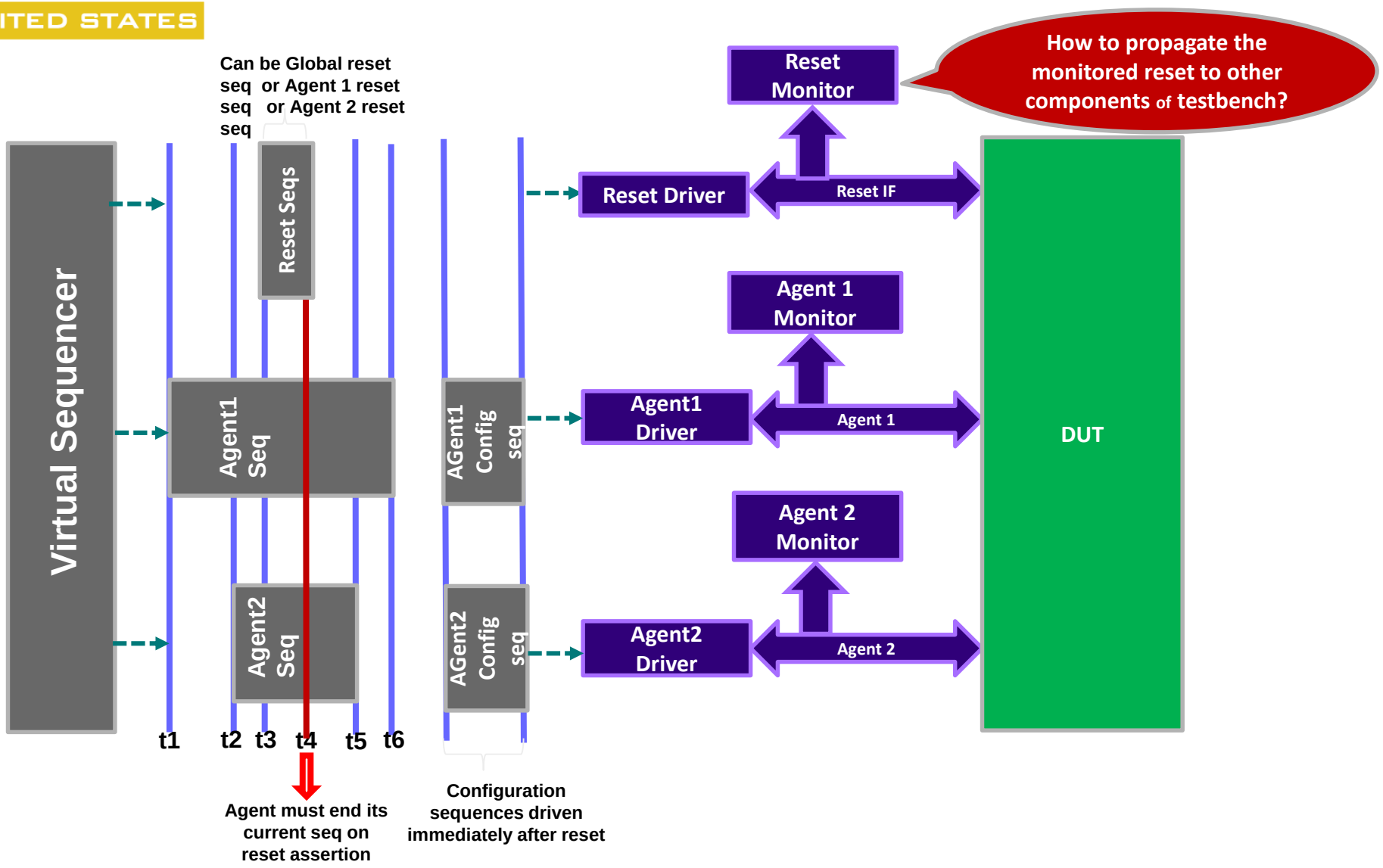
Frequently used Methods	Description
get_global_pool	Returns the singleton global pool for the item type, T.
get_global	Returns the specified item instance from the global item pool.
get	Returns the item with the given key.
add	Adds the given (key, item) pair to the pool. If an item already exists at the given key it is overwritten with the new item.

Reset Aware Testbench

- Handling on-the-fly reset is one of the challenges in testbench design. The UVM methodology does not define how an on-the-fly reset must be handled.

- The reset is a major disruptive event which can occur at any point of time, it’s very important to ensure that the chip exists out of reset and resumes normal operation without any issue.

Reset Aware Testbench



Reset Aware Testbench

In reset monitor:

```
global_reset_ev = cfg.event_pool.get("global_reset");
global_reset_ev.trigger();
```

global reset event

```
agent1_reset_ev = cfg.event_pool.get("agent1_reset");
agent1_reset_ev.trigger();
```

agent1 reset event

```
agent2_reset_ev = cfg.event_pool.get("agent2_reset");
agent2_reset_ev.trigger();
```

agent2 reset event

Reset Aware Testbench

In reset aware components

```
virtual function void build_phase(uvm_phase);
  global_reset_ev = cfg.event_pool.get("global_reset");
  agentX_reset_ev = cfg.event_pool.get("agentX_reset");
endfunction : build_phase

task run_phase() //task body(), in case of sequences
  ...
   forever begin
     fork
       agentX_reset_ev.wait_ptrigger;
       global_reset_ev.wait_ptrigger;
     join_any
     reset_procedure();
   end
  ...
endtask
```

RAL

- uvm_reg_cb provides some standard callback methods like pre_write, pre_read, post_write, post_read, post_predict.

- uvm_event can be used for communication from register callback class to other components like scoreboard, reference model, sequences etc.

- Adopting the uvm_event for RAL to TB communication requires minimal testbench code changes and also the data delivery through uvm_event helps a great deal especially in score-boarding and reference modeling.

RAL

```
class trans_capture_cb extends uvm_reg_cbs;
  uvm_event#(uvm_reg_item) wr_event;
  uvm_event#(uvm_reg_item) rd_event;
  //new constructor

  virtual function void post_write(uvm_reg_item rw);
    wr_event = cfg.event_pool.get("reg_write_event");
    wr_event.trigger(rw);
  endfunction

  virtual function void post_read(uvm_reg_item rw);
    rd_event = cfg.event_pool.get("reg_read_event");
    rd_event.trigger(rw);
  endfunction

endclass : trans_capture_cb
```

register callback class

triggering write event with packet as parameter

triggering read event with packet as parameter

RAL

capturing event from RAL

```
class ref_model extends uvm_component;
  uvm_event#(uvm_reg_item) wr_event;
  uvm_reg_item reg_wr;
  uvm_object reg_obj;
  //new constructor
  virtual function void build_phase(uvm_phase);
    wr_event = cfg.event_pool.get("reg_write_event");
  endfunction : build_phase
  task run_phase()
    forever begin
      wr_event.wait_ptrigger_data(reg_obj);
        if(!$cast(reg_wr, reg_obj)) begin `uvm_fatal(…) end
           process_wr_pkt(reg_wr);
        end
    end
  endtask

endclass
```

Adding callback to register

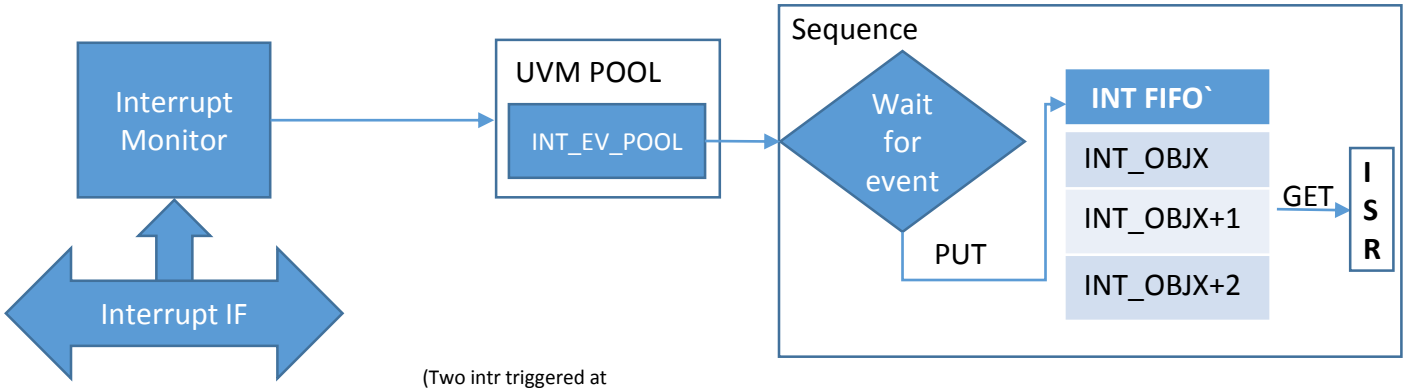
```
trans_capture_cb trans_cap_cb;
trans_cap_cb = new(…);
uvm_reg_cb::add(env.reg_model.reg*,
trans_cap_cb);
```

Interrupt Mechanism

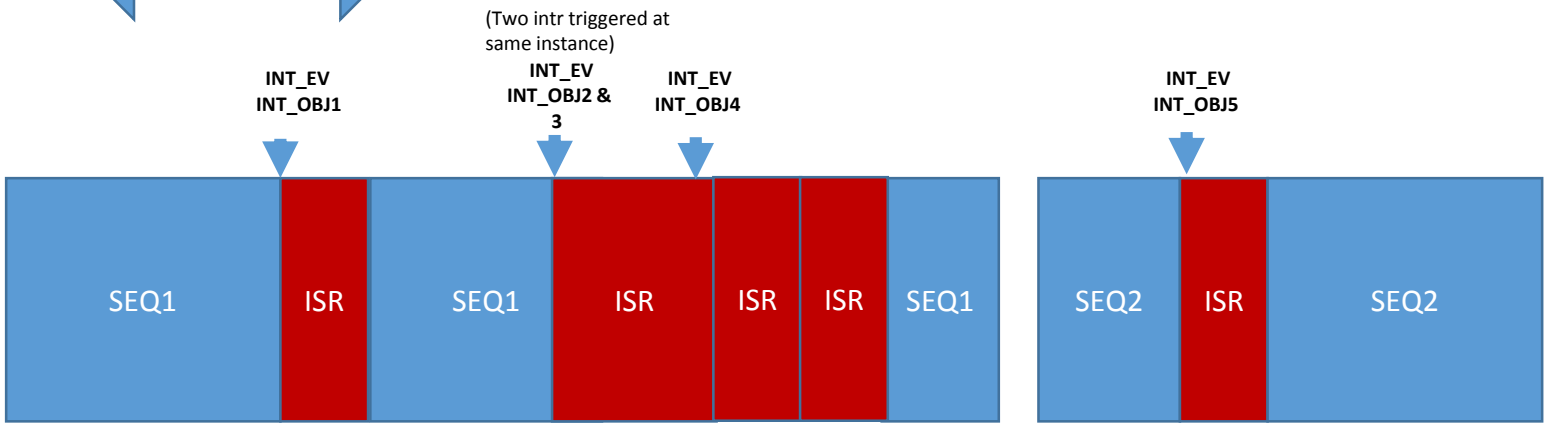
- Interrupt is an event that is triggered by a Design or an IP block once certain conditions are fulfilled; the CPU has to service these events.

- Actions that are to be taken care by CPU while servicing the interrupts are collectively called as -Interrupt Service Routines (ISR).

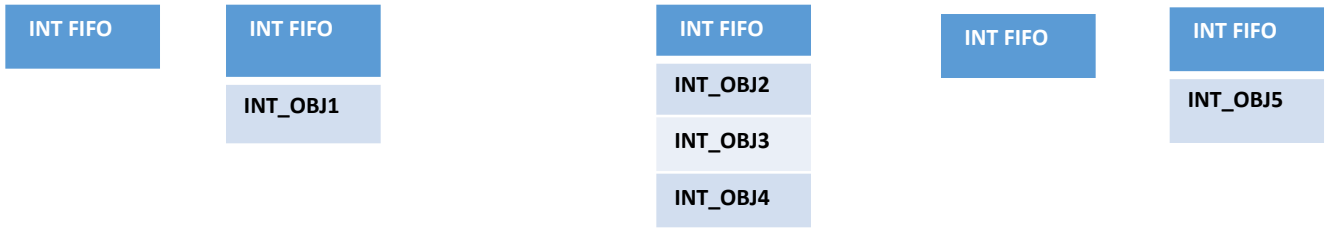
Interrupt Mechanism



TIME LINE:
Triggering of interrupts



FIFO Status



Interrupt Mechanism

```
class isr_monitor extends uvm_monitor;
 trans_c trans;
 uvm_event #(trans_c) isr_event;

 function buid_phase(uvm_phase phase);
   isr_event=cfg.event_pool.get("interrupt_event")
 endfunction


 task run_phase(uvm_phase phase
   forever begin
     //Capturing and triggering the event
     @(posedge isr_if.int_n);
     isr_event.trigger(trans);
   end
  endtask
endclass: isr_monitor
```

monitor

```
class env_cfg extends uvm_object;
  //event pool declaration
  uvm_event_pool event_pool;
endclass: env_cfg
```

event triggered

Interrupt Mechanism

```
class isr_seq extends uvm_sequence;
 trans_c trans;
 uvm_event #(trans_c) isr_event;
 task body();
   isr_event.wait_ptrigger_data(trans);
   // Store the event in the queue.
   m_sequencer.grab(this);
   // isr scenario
   m_sequencer.ungrab(this);
  endtask

endclass: isr_s
```

sequence

waiting for the event

grab the sequencer

Test

```
class env_base_test extends uvm_test;
  uvm_event_pool evt_pool;
  function buid_phase(uvm_phase phase);
    evt_pool=uvm_event_pool::get_global_pool();
    env_cfg.event_pool=evt_pool;
  endfunction
endclass: env_base_test
```

Callbacks

- The uvm_event_callback class is an abstract class that is used to create a callback object which is attached to uvm_event#(T) as shown in below code.
- The uvm_event_callback class has two empty virtual methods.
- The user has to create a callback class which extends from uvm_event_callback and has to override the virtual methods to implement the required functionality.
- The callbacks support is one of the main feature of any VIP.

```
virtual class uvm_event_callback#(type T=uvm_object) extends uvm_object;
```

Callbacks

driver class

```
class pkt_dvr extends uvm_driver;
   uvm_event #(trans) ev1;
   uvm_event #(trans) ev2;
   // new constructor
   virtual function void build_phase(uvm_phase phase);
      ev1 = uvm_event_pool::get_global("ev1");
      ev2 = uvm_event_pool::get_global("ev2");
endfunction
   task run_phase ( uvm_phase phase);
      seq_pkt pkt;
      start(pkt)
      ->ev1.trigger(pkt);
      process(pkt);
      ->ev2.trigger(pkt);
    endtask
endclass:pkt_dvr
```

ev1 triggered after the start method

Callbacks

user callback

```
class user_event_cb extends uvm_event_call_back #(trans);
 // new constructor

  virtual function bit pre_trigger(uvm_event ev, trans pkt);
    //change trans/data before triggering event
  endfunction

  virtual function bit post_trigger(uvm_event ev, trans pkt);
    //change data/trans after triggering event
  endfunction

endclass
```

pre_trigger

post_trigger

Callbacks

test

```
class my_test extends uvm_test;
 uvm_event ev1;
 uvm_event_pool evt_pool = uvm_event_pool ::
get_global_pool();
 user_event_callback cb_event;
 function new(string name = "my_env", uvm_component
parent=null);
  ev1 = evt_pool.get("ev1");
 endfunction
 virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  ev1.add_callback(cb_event);
  ……
  ev2.delete_callback(cb_event);
 endfunction
endclass
```

add callback

delete callback

2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Conclusion

- The uvm_event and event pool provides synchronization between multiple threads or concurrent processes in the verification environment.

- It is observed that by using uvm_event we can achieve better synchronization without making major changes to the exiting test-bench.

- The uvm_event wrapper class around the traditional system Verilog event with added methods makes uvm_event to be applicable to a broader and complex application than just synchronization.

2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

References

[1] Verilab - Advanced UVM Register Modeling – “There’s More Than One Way to Skin A Reg” – Mark Litterick and Marcus Harnisch.

[2] DVCON INDIA 2014 - Global Broadcast with UVM Custom Phasing Jeremy Ridgeway, Dolly Mehta - Avago Tech.

[3] Accellera, “UVM User Guide, v1.1d”, www.uvmworld.org

[4] Accellera, “UVM Reference Guide, v1.1d”, www.uvmworld.org

[5] Dialog Semiconductor – “The UVM Register Layer Introduction and Experiences” - Steve Holloway

Acknowledgement

We profoundly thank colleagues and management at **elitePLUS Semiconductor Technologies Pvt Ltd**, Bangalore for their valuable guidance and thought provoking discussions.

Thanks **DVCON-16** for providing this platform to share our views.