Are you suffering to handle on-the-fly events in complex UVM scenarios?

Roman Wang - Verification@AMD
roman.wang@amd.com

DVClub Shanghai – 28th March 2014
AGENDA

The Story: The Event and Interrupt

Proposals: UVM on-the-fly events service at IP and SoC levels

Practical Example: Implementation in UVM Verification Environment
At the IP level, an interrupt/event may be a signal or upstream message.

At the SoC level, interrupts/events are turned into upstream “message” transfer or packed IRQ line.
- Different type of IRQ messages to system memory (Ex. A write to DDR or SRAM)
- Packed IRQ line to high level interrupt controller or CPU Core

Challenge:
How to handle concurrent different kinds of on-the-fly events and interrupts in complex UVM scenarios?

Complexity and handshake may be a big problem.
The Goal

Direction:
Create an event/interrupt service routine (ISR) infrastructure that could be re-usable from the block level to the full-chip level
ISR INFRASTRUCTURE

–Capturing the interrupt/event

–Identifying the interrupt/event

–Synchronization with UVM sequences via ISR_DB

1. Past generation: ovm_pool @OVM2.1.2
   use API get_global_pool() to get a singleton global pool

1. Current generation: uvm_resource_db @UVM1.1x

3. Next generation: uvm_resource_db+dap @ UVM1.2RCx
THE IP LEVEL

- From Picture – slide 6
- Interrupt /Event signals are captured by monitor of specific interface UVCs (Ex. Strap UVC or AMBA UVC) – Capture part
- Such iUVC monitor analysis_port is attached to the module predictor, functional coverage collector
- IP-level interrupt monitor is a part of module monitor – Identifying part
  - Class with several analysis_imps
  - Receives the iUVC monitor packet for the Interrupt/event line
  - When asserted, it sets a uvm_resource in the ISR_DB
  - This part could not be reused, because different IP has different identifying mechanism for events and interrupts
- IP Level ISR sequence – handshake part
  - Should be re-usable at the block and SoC levels
  - Instantiates resource handles and retrieves the resources: get_by_name
  - Then the ISR can wait on the resource: wait_modified
- IP level module UVC (mUVC) must be vertically reusable.
From Picture – slide 8

All interrupts and events must be supported
- Be careful with interrupt resource names so they do not conflict at the SoC level
- Recommend to add prefix. Ex. IP1_FUN1_

high_layer_iUVC monitor analysis_port is attached to an SoC module monitor
- Writes the different interrupts to the ISR_DB

ISR sequence
- Same as the IP level (vertical re-use)

Interrupt/event scoreboard in SoC mUVC
1. Connect each IP mUVC’s external interrupt/event analysis_port to interrupt/event Scoreboard
2. Connect high_layer_iUVC’s monitor analysis_port to interrupt/event Scoreboard
What’s the challenges in PMU UVM stand-alone if scenario need to cover most of them?

#1. There are many different type of interrupts

#2. There are many IP-related events

#3. There are many clients’ handshake events

#4. There are many power control events

.............

We expected to create complex power scenarios to handle on-the-fly interrupts and events in parallel and automatically on the background.
The Key

How to ensure the correct handshake?

What is the handshake mechanism?

Why it’s better to do that?
HOW THE HANDSHAKE WORKS IN PMU STAND-ALONE

- iUVC’s MTR captures the interrupt or event, then broadcasts to module MTR.
- Module MTR will identify exact source and write 1 to resource DB with specific index.
- The ISR sequence will be free-running on the background. All of subroutines will run concurrently and wait for 0 to 1 change in ISR_DB.
- After subroutine completed, write 0 to ISR_DB with the same index.
- The uvmreg main sequence programs the DUT’s registers, and then DUT will generate interrupt or event, this sequence should wait for 1 to 0 change in ISR_DB before move forward to next steps.
typedef uvm_resource #(int) ip_isr_db;

-In the module monitor

function void fch_acpi_module_monitor::write_IRQ_mtr(ref pkt_base pkt);
   strap_pkt#(`A_IN_WIDTH, `A_INTF_OUT_WIDTH) A_strap_pkt;
   $cast(A_strap_pkt, pkt);
   `uvm_info(get_type_name(), $psprintf("write IRQ mtr get one transaction :\n%s", A_strap_pkt.sprint()), UVM_HIGH);

   if(A_strap_pkt.dut_outs.value[2]) begin
      if(A_strap_pkt.dut_outs.value[32])
         trigger_isr_db(`IP_IRQ1); //A_strap_pkt.dut_outs.value[2]
      else
         trigger_isr_db(`IP_EVENT1); //A_strap_pkt.dut_outs.value[2]
   end

   Endfunction

   function void trigger_isr_db(string event_name);
      `uvm_info(get_type_name(), $psprintf("%s is writing 1 to ISR_DB.....", event_name), UVM_HIGH)
      if(!uvm_resource_db#(int)::write_by_name("uvm_test_top.ip_tb.IP_module_uvc.luvc.msr_agt.sqr", event_name, 1, this ))
         `uvm_error(get_type_name(), "Writing to ISR_DB failed!\n")
   endfunction : trigger_isr_db
In the user extended uvmreg_sequence
programming DUT registers to generate irq/event after completing specific staffs (Ex. DMA transfer)
is_isr_completed_new(`IRQ1); // it will stuck here until the ISR finished

In the uvmreg_base_seq
task is_isr_completed_new(string schedule_item);
  ip_isr_db isr_db;
  `uvm_info("is_isr_completed",$psprintf("Waiting for %s ISR finished!\n",schedule_item), UVM_HIGH);
  isr_db = uvm_resource_db#(int)::get_by_name(m_sequencer.get_full_name(), schedule_item);
  do begin
    isr_db.wait_modified(); // It’s blocking!!
  end
  while(isr_db.read()); // move on only when detect 1->0 change
  `uvm_info("is_isr_completed",$psprintf(" %s ISR finished, the main sequence gets moving forward!\n",schedule_item), UVM_HIGH);
endtask

EXAMPLE CODE COUNT...
In the routine uvmreg_sequence

class fch_ip_routine_seq extends fch_ip_base_sequence; // unique sequence and including all irq/event handling staffs
string isr_id = "roman";
  virtual task body();
  m_sequencer.grab(this); // Grab => Immediate exclusive access to sequencer
  super.body();
  `uvm_info(get_type_name(),$psprintf("%s is starting...\n",get_sequence_path()), UVM_HIGH)
  `uvm_info(get_type_name(),$psprintf("Current isr_id is %s \n",isr_id ), UVM_HIGH)

  // IRQ1 --------------------------------------------------------------------
  if(isr_id == `IRQ1) begin
  // checking the RO status , W1C and really clear, etc.
  reg_data = IP.ABC_REG__block.IRQ_Reg_Status.Status.value; // UVM register model
  `uvm_info(get_type_name(),$psprintf("\n %s Happened :: reg_data = %h \n",isr_id, reg_data),UVM_HIGH)
  IP.ABC_REG__block.IRQ_Reg_Status.Status(status, UVM_CHECK, .parent(this));
  ......
EXAMPLE CODE COUNT...

--- In the ISR sequence

```
task is_event_happen(string schedule_item);
    ip_isr_db isr_db;
    uvm_resource_db#(int)::set(m_sequencer.get_full_name(), schedule_item, 0, this);  // Initial
    isr_db = uvm_resource_db#(int)::get_by_name(m_sequencer.get_full_name(), schedule_item);
    do begin
        isr_db.wait_modified();
    end
    while(!isr_db.read()); // wait for 0->1 change
    `uvm_info(get_type_name(), $psprintf("%s Event happened just now!\n", schedule_item), UVM_HIGH);
endtask

```

```task clear_isr_db(string schedule_item);
    ip_isr_db isr_db;
    isr_db = uvm_resource_db#(int)::get_by_name(m_sequencer.get_full_name(), schedule_item);
    isr_db.write(0);
    `uvm_info(get_type_name(), $psprintf("%s Event ISR completed and back to register sequence right now!\n", schedule_item), UVM_HIGH);
endtask
```
In the ISR sequence

class ip_isr_handler extends base_sequence;
   // Declare the ISR sequence here
   ip_routine_seq irq1_isr;
   ip_routine_seq event1_isr;
   .......

virtual task body();
   `uvm_info(get_type_name(), "Kickoff the System ISR Handler...", UVM_LOW)
   // Set up sequencer to use priority based on FIFO order
   m_sequencer.set_arbitration(SEQ_ARB_STRICT_FIFO);
   nop; // wait for few ticks
   fork
      forever begin
         is_event_happen('IRQ1);
         `uvm_info(get_type_name(), "irq1ISR Routine...", UVM_LOW)
         irq1_isr = fch_acpi_routine_seq::type_id::create("irq1_isr");
         irq1_isr.isr_id = 'IRQ1;
         irq1_isr.start(m_sequencer,this,HIGH);
         irq1_isr = null;
         clear_isr_db('IRQ1);
      end
      .......
   join
   ......
EXAMPLE CODE COUNT...

- In the ISR sequence

  forever begin
    is_event_happen(`EVENT);
    ........
    clear_isr_db(`EVENT);
  end

- In the virtual sequence_base

  // The ISR sequence is free-running background.
  virtual task body();
  ........
  ip_isr = ip_isr_handler::type_id::create("ip_isr");
  fork
    ip_isr.start(p_sequencer.a_msr);
  join_none
  ....
  Endtask
WHAT THE MODULE PREDICTOR SHOULD DO?

Update the RO status field value in the UVM register model according to the specific IRQ/Event

1. Update the UVM register model for sequence

2. Update the module UVM register model (self-maintained) for functional coverage collection, scoreboard and prediction.
DO NOT ABUSE THE USAGE OF UVMRESOURCE_DB!

We must limit this to only a few very important events - can't use this methodology too widely otherwise we might have performance

UVM Coding Guideline:

Do NOT use `uvm_config(resource)_db to do frequent communication between components/sequences

Avoid this usage otherwise it will reduce the simulation performance. Using the resource database for frequent communication between components is an expensive way to communicate. The component that is supposed to receive new configuration information would have to poll the configuration space which would waste time. Instead, standard TLM communication should be used to communicate frequent information changes between components. TLM communication does not consume any simulation time other than when a transaction is being sent. Infrequent communication such as providing a handle to a register model is perfectly acceptable.
WE WELCOME YOUR IDEAS

QUESTIONS?

Please send questions to me via roman.wang@amd.com

Thanks for contribution of Karl Whiting @AMD.
DISCLAIMER & ATTRIBUTION

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREBIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION

© 2014 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Other names are for informational purposes only and may be trademarks of their respective owners.