Reuse C test and UVM sequence utilizing TLM2, register model and interrupt handler

Liu HongLiang
AMD Inc.
Andy.liu@amd.com

Gao Teng-Fei
AMD Inc.
Teng-Fei.gao@amd.com

ABSTRACT

This paper describes the challenges and solutions when reuse Universal Verification Methodology (UVM) sequence from IP block level to system-on-a-chip (SoC) level, where tests are written in C.

Register and interrupt verification need be done at both the IP block and SoC level. The IP block environment is usually implemented with UVM. The SoC level is more complicated, it may have C test or a C BFM model. This paper's approach is to directly reuse the IP UVM register sequence and interrupt handle sequence, and keep the SoC C test unchanged.

First, this paper describes how to replace the SoC C BFM model with the interface UVM component (iUVC), the iUVC will be reused from the IP block. A user created API collects the C data structure packet, and converts the C data structure packet to a TLM2 packet, including the generic payload and protocol specific extension. The TLM2 packet will be used by the iUVC . With this approach, the C test doesn't need to be modified. Furthermore, when the bus interface protocol changes, the SoC programmer just needs to update the extension part, keeping the TLM2 generic payload, which saves effort when doing horizontal project reuse especially when bus interface protocol is different in a new project.

Next, this paper describes the reuse register sequence with layer register model method, adding the IP register block to top register block, adding the IP register map as top register sub map. The IP register sequence can be directly used at the SoC level, the SoC user just needs to change the adapter and sequencer according to the new interface protocol. Moreover, the UVM register sequence and C test share the same interface UVC. When writing the regsiter sequence, it would be better to use find_by_name, which avoids the register model hierarchy problem, so when doing the vertically reuse, the register sequence is not restricted by register hierarchy.

Finally, this paper describes interrupt service sequence reuse. The interrupt is collected by the IP monitor, and the interrupt handle is stored in uvm_ resource_db. The interrupt service sequence retrieves the handler, gets the interrupt entry, and processes interrupt. When doing the vertically reuse to SoC, the IP monitor is replaced by the SoC monitor, but the interrupt service sequence remains same.

With above methods, the C test and UVM sequence work harmoniously at the SoC level, it solves the challenges that the IP environment is UVM , and the SoC level is C test, and both the UVM sequence and C test need to be reused.

Keywords: **C, interrupt service sequence, reuse, register sequence, UVM**.

I. INTRODUCTION

C test is used to generate write and read stimulus of register and memory at the system-on-a-chip (SoC) level. Universal Verification Methodology (UVM) 1.2 is on the road to an IEEE standard, and it is more and more popular in IP verification.

Some SoC environments will encounter a similar challenge, which is how to let C and UVM work together with less affect on legacy SoC C tests and new IP block UVM sequences.

This paper presents how to directly reuse UVM sequence from IP block to SoC level, and keeps SoC level C test unchanged within a three-layer SoC verification infrastructure. That means both SoC C test and IP UVM sequence can be reused directly.

C and UVM co-work infrastructure, register model sequence reuse, and interrupt sequence reuse are described in different chapters.

II. CHALLENGES TO REUSE SOC C TEST AND IP UVM SEQUENCE

The first challenge is how to keep the SoC C test unchanged when the SoC uses a new UVM bus agent.

SoC verification engineer may not be familiar with UVM; The SoC legacy test bench that we worked on was deeply developed in C; The UVM agent bus protocol may be changed in future projects. All above items require a loose coupling between the C test and the underlying bus agent. It is better to isolate them with a new layer.

The new layer should connect the C and UVM agent, and it needs to be carefully designed to be scalable to the UVM agent bus protocol. When the agent bus protocol changes, such as a change from PCIE to AXI, the effort to update the new layer needs to be as small as possible. This paper describes the detail and show the new layer in Fig 1.

The second challenge is directly reuse the IP UVM sequence to SoC.

The first step is choosing appropriate IP sequence to do reuse. SoC and IP are different verification levels. SoC verification usually cares about IPs connectivity, IPs register access in system view, system interrupt generate and processing. IP usually focus on functions verification. Thus, most IP functions sequence are not needed at SoC level. In this paper, we reuse IP UVM register sequence and IP interrupt sequence.

The second step is directly reuse IP UVM register sequence and IP interrupt sequence.
The best situation is that SoC programmer doesn't need learn and modify IP sequence, and just start IP sequence. This paper shows the directly reuse key points including layered UVM register address map, bus adapter, and UVM resource DB.

The third challenge is how to synchronize C and UVM in C-UVM mixed environment.

In system view, C and UVM are in separate processes. The function main() is the C test entry. When function main() finishes, the simulation finishes.

UVM has the phase mechanism to control the simulation progress, the global method run_test() is UVM's entry, and the end of simulation in UVM is controlled by managing phase objections raise and drop.

This paper adopts the simplest approach to synchronize the C and UVM: C is responsible for end simulation. First, As Fig 1 depicts, every C traffic is converted to UVM sequence item in UVM run phase, C test can finish the simulation once all C traffics finish; Second, when the reused IP UVM sequence run phase finish, it notices C test to finish simulation by trigger a DPI event.

III. LAYERED SOC C-UVM MIX ENVIRONMENT

Our test bench sets up a three-layer C environment prototype as shown in Fig.1. The top layer is application layer, the middle layer is transaction layer, and the bottom layer is bus layer. The C test resides in application layer. It issues write and read stimulus with standard PCIe® protocol data structure, and passes through the PCIe packet handle to the transaction layer.

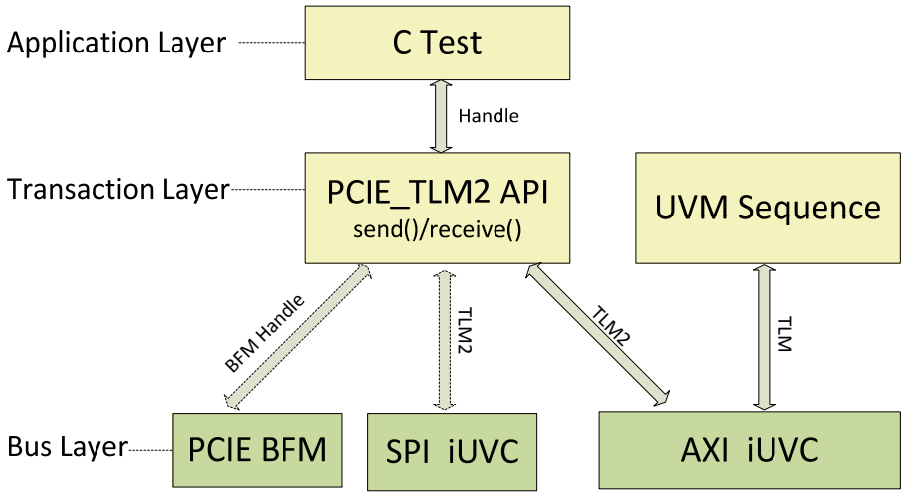

Fig. 1. Layered SoC C-UVM mix environment.

PCIE_TLM2 API sits in the transaction layer. One virtual function send(*pcie pkt) is implemented inside PCIE_TLM2 API for downstream traffic, it gets the application layer PCIe packet handle as input argument.

The send(*pcie pkt) invokes legacy PCIe C BFM model to drive the PCIe packet. We need to replace the C BFM with the interface UVM Component (iUVC) in the bus layer.

We import TLM2 to construct the send function, which is scalable to bus layer protocols. It doesn't directly convert the PCIe packet to bus layer protocol packet. Instead, it converts the PCIe packet to the TLM2 generic payload, and uses the TLM2 extensions to support different bus protocols.

So when the bus layer protocol changes, TLM2 generic payload conversion can remain the same, SoC programmers only need to update the TLM2 extensions.

The transaction layer data flow is shown in Fig. 2.

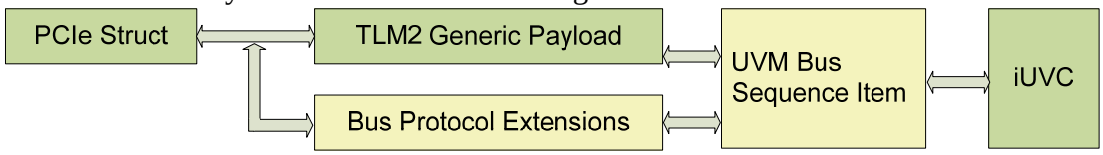

Fig. 2. Transaction layer data flow.

The following code segment shows the TLM2 generic payload and AXI protocol extensions.

```
tlm::tlm_generic_payload *ptr_tlm_trans = new tlm::tlm_generic_payload;
gp_extensions_axi * ptr_tlm_extensions  = new gp_extensions_axi;
ptr_tlm_trans->set_write(pkt.is_write());
ptr_tlm_trans->set_address(pkt.addr);
ptr_tlm_trans->set_data_length(pkt.len);
ptr_tlm_trans->set_data_ptr(pkt.data);
ptr_tlm_trans->set_byte_enable_ptr(pkt.be);
ptr_tlm_trans->set_response_status();

ptr_tlm_extensions->set_id();
ptr_tlm_extensions->set_burst();
ptr_tlm_extensions->set_cache();
ptr_tlm_extensions->set_prot();
ptr_tlm_extensions->set_user();
```

Another receive(tlm::tlm_generic_payload &pkt) function is implemented for upstream traffic. It handles the response from the UVM agent layer via TLM2.

The bus layer is the bus agent layer. It is iUVC if the agent implemented in UVM . The UVM sequence and C transaction layer share this layer. Bus layer is usually reused from the bus IP UVM environment. A typical iUVC encapsulates the sequencer, driver, port monitor, agent configuration and a virtual interface, an iUVC is shown in Fig. 3.

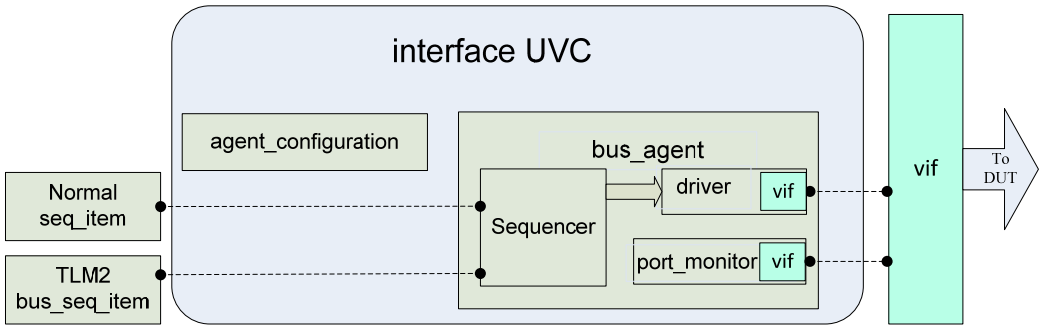

Fig. 3. Interface UVC block diagram.

To save the reuse effort, this paper recommends using module bind and virtual interface to connect iUVC to the IP design, as shown in Fig. 4. The bind connection can be directly reused from the IP block to the SoC level, together with the IP design module reused to the SoC.

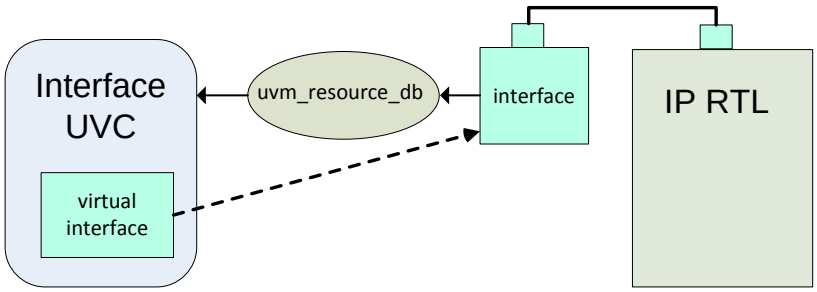

Fig. 4. Module bind and virtual interface.

```
bind  amba_rtl  axi_slave_if  amba_axi_slave_if (
      .araddr(araddr),
      .rdata(rdata),
      ...
);
```

uvm_resource_db is used to pass the real interface handle to interface UVC as following:

```
module   register_amba_rtl_axi_slave_if;
initial   begin
   uvmpkg::uvm_resource_db#(virtual axi_slave_if)::set("interface_register ", intf_ins_path,
   amba_rtl.amba_axi_slve_if);
end
endmodule
```

IV. REUSE IP REGISTER MODEL SEQUENCE

Register access and memory map IO access are common scenarios in both IP and SoC verification. The Accellera UVM user guide describes register model constructing, randomization, and front-door and back-door access mechanism. So this paper starts from the register model access sequence, which is called the register abstraction layer (RAL) sequence in UVM.

It is hard to reuse common UVM sequences, when the IP port bus protocol is different from the SoC port bus protocol. That is because the common UVM sequence is usually implemented with `uvm_do() or seq.start(), the sequence generates a protocol-specific sequence_item and run on a protocol-specific sequencer. Thus, once the bus protocol is changed, the common UVM sequence can't work; this means the common UVM sequence can't be reused for this case.

However, the RAL sequence works at a high abstract layer other than the physical bus protocol layer, as shown in Fig. 5, the RAL abstract layer generates generic bus transaction - uvm_reg_bus_op, it doesn't depend on any specify bus protocol.

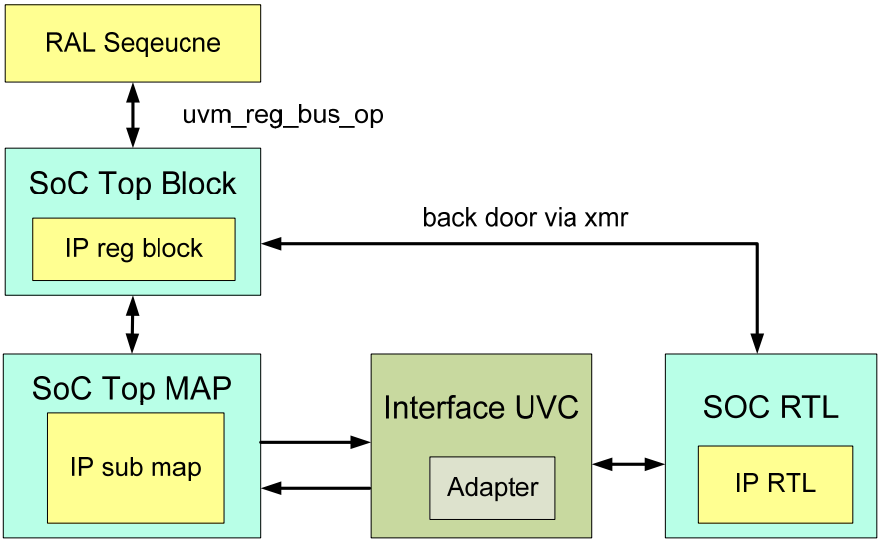

Fig. 5. RAL sequence structure.

The adapter and address map and are key points to reuse RAL sequence from IP to SoC.

The SoC top block instantiates IP register block. It creates SoC top map according to the SoC bus interface, IP address map is added as a sub map of SoC top map.

The adapter converts uvm_reg_bus_op to a bus protocol-specific sequence item, and an interface UVC receives the bus protocol-specific sequence item and drives the interface.

When the RAL sequence is reused from IP block to SoC level, if the SoC bus protocol is same as IP bus protocol, the IP adapter and interface UVC can be reused to SoC directly; if the SoC bus protocol is different from IP bus protocol, the IP adapter and interface UVC need be replaced with SoC adapter and interface UVC.

As shown in the code segment 1, the IP register sequence uses the IP register model as the top layer to issue write and read transactions; however, at the SoC level, the SoC register model is the top layer. Thus, register model hierarchy is different in IP block and SoC level.

code segment 1:

```
ip_reg_model.reg0.write(status, value, .parent(this));
```

SoC programmer usually needs to manually change the register model hierarchy to do RAL sequence reuse, as demonstrated in the code segment 2.

code segment 2

```
soc_reg_model.ip_reg_model.reg0.write(status, value, .parent(this));
```

This paper proposals a register sequence style as following:

```
ip_reg = reg_model.get_reg_by_name(reg0);
ip_reg.write(status, value, .parent(this));
```

Function get_reg_by_name() searches the register model topology tree to find the correct register. Make sure the register name is unique in the environment. At IP block, the IP verification environment passes through the IP register model handler to reg_model in the IP register sequence, so the search start point is the IP register model.

At SoC level, the SoC verification environment passes through the SoC top register model hander to reg_model in the reused IP register sequence, so the search start point is the SoC register model.

With above register sequence style, the register sequence can be directly reused from the IP block to SoC level.

V. REUSE IP INTERRUPT SERVICE SEQUENCE

In general, the system interrupt can be a sideband port signal or system message like PCIE. In this paper, the interrupt verification target in the UVM environment is as follows: when one interrupt happens, it should be monitored, and the related interrupt status register should get the correct value; when interrupt processing is done, the related interrupt status register should be cleared.

During one simulation, there may be several interrupts, so each interrupt should have a unique identity. The interrupt service sequence needs to identify the correct interrupt entry, find out the related status register, and process the interrupt task.

This paper proposes an interrupt handler infrastructure, which includes a UVM monitor, a central UVM resource DB, and a reusable interrupt service sequence, IP interrupt handler infrastructure is as described in Fig. 6.

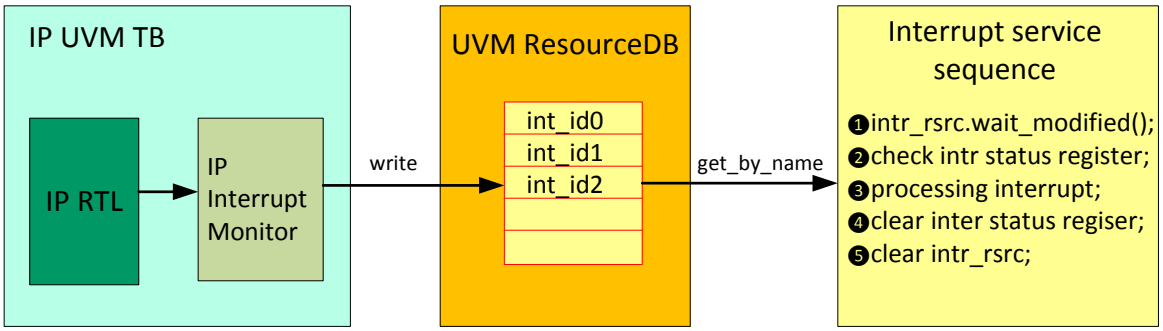

Fig. 6. IP interrupt handler infrastructure.

The UVM resource DB is static and global, the interrupt entry and value are available in both monitor and interrupt service sequence; moreover, uvm_resource provides interrupt service sequence request trigger function wait_modified().

When interrupt monitor catches an interrupt, the write function will be called, the interrupt entry and interrupt value are written into one uvm_resource.

The write code segment in interrupt monitor:

```
virtual function void write(T intr_mtr_value);
    uvm_resource #(int) intr_rsrc;
    uvm_resource_db#(int)::set("", "int_id0", int_mtr_value, this);
    intr_rsrc=uvm_resource_db#(int)::get_by_name("", "int_id0");
    intr_rsrc.write(int_mtr_value, this);
endfunction
```

uvm_resource_db#(int)::set() creates a new uvm_resource.

The interrupt service code segment is as following. Once an interrupt is written into uvm_resource in monitor, wait_modified() will be triggered and interrupt service will be executed.

```
    uvm_resource #(int) intr_rsrc;
    intr_rsrc = uvm_resource_db#(int)::get_by_name("", " int_id0");
    intr_rsrc.wait_modified();
```

When the IP interrupt service sequence is reused to the SoC level, the IP interrupt monitor needs to be replaced with the SoC interrupt monitor, but the write function and interrupt service sequence can remain the same, SoC interrupt handler infrastructure is as shown in Fig. 7.

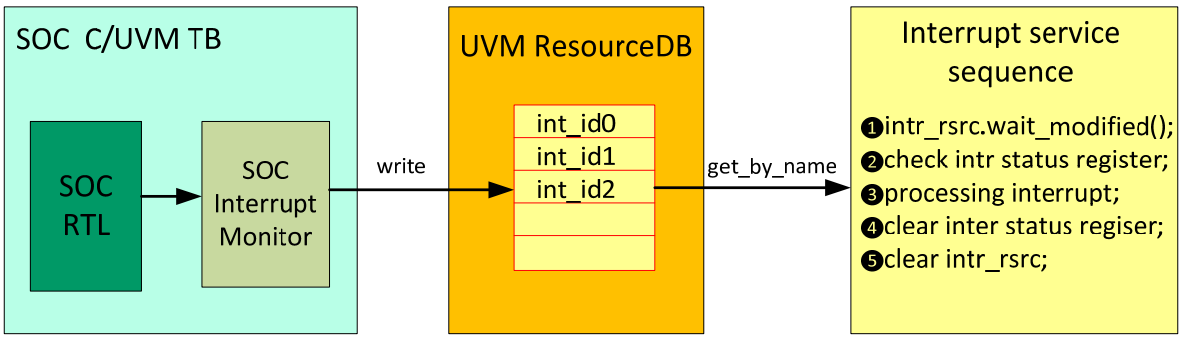

Fig. 7. SoC interrupt handler infrastructure.

VI. SUMMARY

This paper describes the challenges to reuse SOC C test and IP UVM sequence in SoC C-UVM mixed environment. A three-layer SoC verification environment infrastructure is introduced. The

C test and UVM sequence work together within this structure, sharing the bus layer agent; TLM2 generic payload and bus protocol specific extensions provide flexibility to the bus layer agent implementation, and the bus layer agent can be directly reused from IP UVM environment.

In the three-layer infrastructure, UVM sequence can be created by SOC programmer or be reused from IP block; This paper proposes to directly reuse IP register model sequence and interrupt service sequence, for register and interrupt verification usually needs to be covered in both IP block and SOC level. UVM sequence reuse challenges are elaborated and the related solutions are described. Core source code snippets are also presented for both SoC and IP programmer.

VII. REFERENCES

[1] OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL
[2] "IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2012, 2012. http://standards.ieee.org/getieee/1800/download/1800-2012.pdf.
[3] UVM 1.2, http://www.accellera.org/downloads/standards/uvm1.2
[4] Rich Edelman, Raghu Ardeishar. “UVM SchmooVM – I Want My C Tests”, Proc. of DVCon 2014, San Jose.
[5] Liu HongLiang, Whiting Karl. “Highly Configurable UVM Environment for Parameterized IP Verification”, Proc. of DVCon 2015, San Jose.
[6] Bryan Sniderman, Vitaly Yankelevich. “Multi-Language Verification:Solutions for Real World Problems”, Proc. of DVCon 2014, San Jose.