Wrapping Verilog Bus Functional Model (BFM) and RTL as Drivers in Customized UVM VIP Using Abstract Classes

Roman Wang
AMD Shanghai, China
Roman.Wang@amd.com

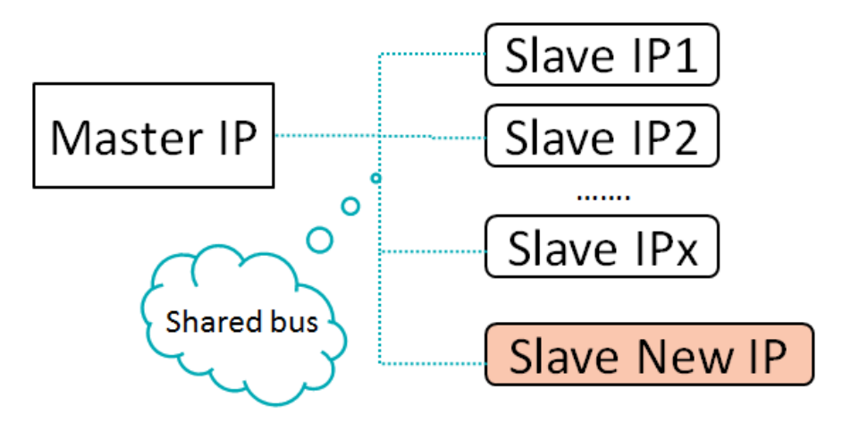
Thomas Bodmer
AMD Sunnyvale, USA
Thomas.Bodmer@amd.com

Abstract

With the increasing complexity of design and verification requirements, more and more verification teams adopted the UVM methodology to build their testbench. UVM could bring a highly reusable, scalable, extensible and configurable framework to engineers to shorten the verification schedule and ensure quality. UVM verification IP (VIP) usually stands for a specific standard protocol interface UVM verification component (iUVC), and could be easily reused to different level of UVM verification testbenches (IP or SoC).
In general, the verification team integrates in-house or 3rd party UVM VIP to verify the standard bus protocol. However, there may be no available UVM VIP for a particular bus protocol, and an engineer has to develop the customized UVM VIP as specific requirements.
In special cases, the master IP is connected to different slave IP through a shared bus, where timing is particular and a little bit complex. This master IP with bus interface had been proved in past projects at the SoC level. When the IP team wants to verify a new IP with this type of shared particular bus interface that will be connected to the same master IP at the SoC level, the customized UVM VIP is necessary to help verify the new IP at the UVM stand-alone level at an early stage. With such requirements, the master IP designer could decouple the RTL into a clean Verilog BFM model with two bus interfaces: one is to drive the new slave IP and easy, and another is simply to get drive from others. The efficient way to create a customized UVM VIP is to reuse and wrap the Verilog BFM model inside and use abstract classes to define functions that easily communicate BFM with the VIP driver. The SystemVerilog (SV) interfaces are bind between BFM and UVM VIP to make sure that BFM is invisible from the outside. The user could treat this UVM VIP as the general UVM VIP. In this way, when the master IP designer updates the bus timing, the UVM VIP could sync up the decoupled BFM without any change.
This paper will introduce a proposal to deploy this kind of customized UVM VIP, and discuss the reuse considerations. Users could extend this approach to implement other similar scenarios

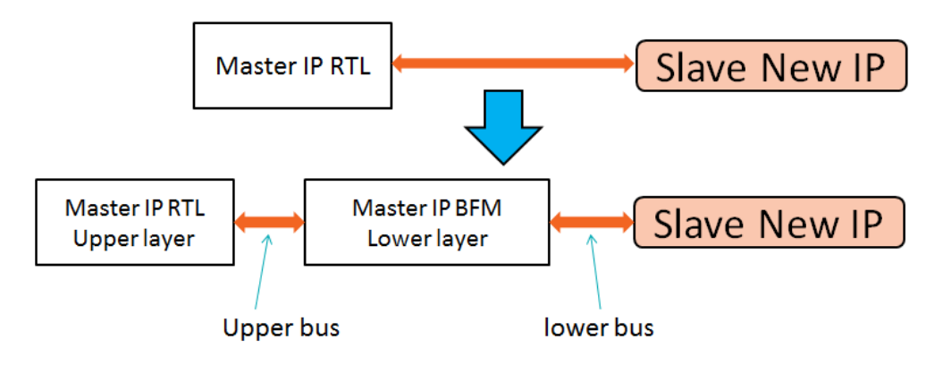
Verification Requirements and Challenges

When our group develops the new slave IP with the same particular bus interface which will be integrated in the SoC, we encounter the following challenges in UVM stand-alone verification.
•No available UVM VIP including both in-house and 3rd party.
•There is no Verilog BFM which has modular functions and tasks API.
•IP must be fully verified in UVM stand-alone, the SoC level verification is too late.
•If master IP changes the bus timing, the slave IP UVM verification environment must sync up without any changes.

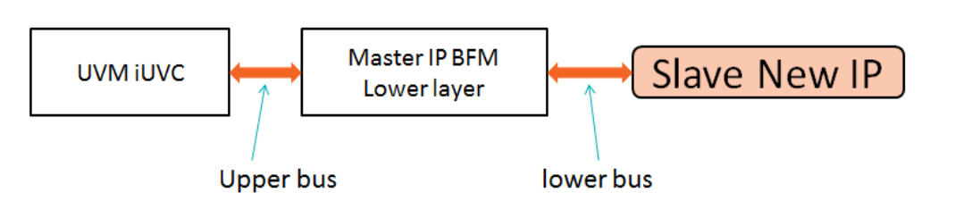


Decouple the Master IP RTL into Verilog BFMs

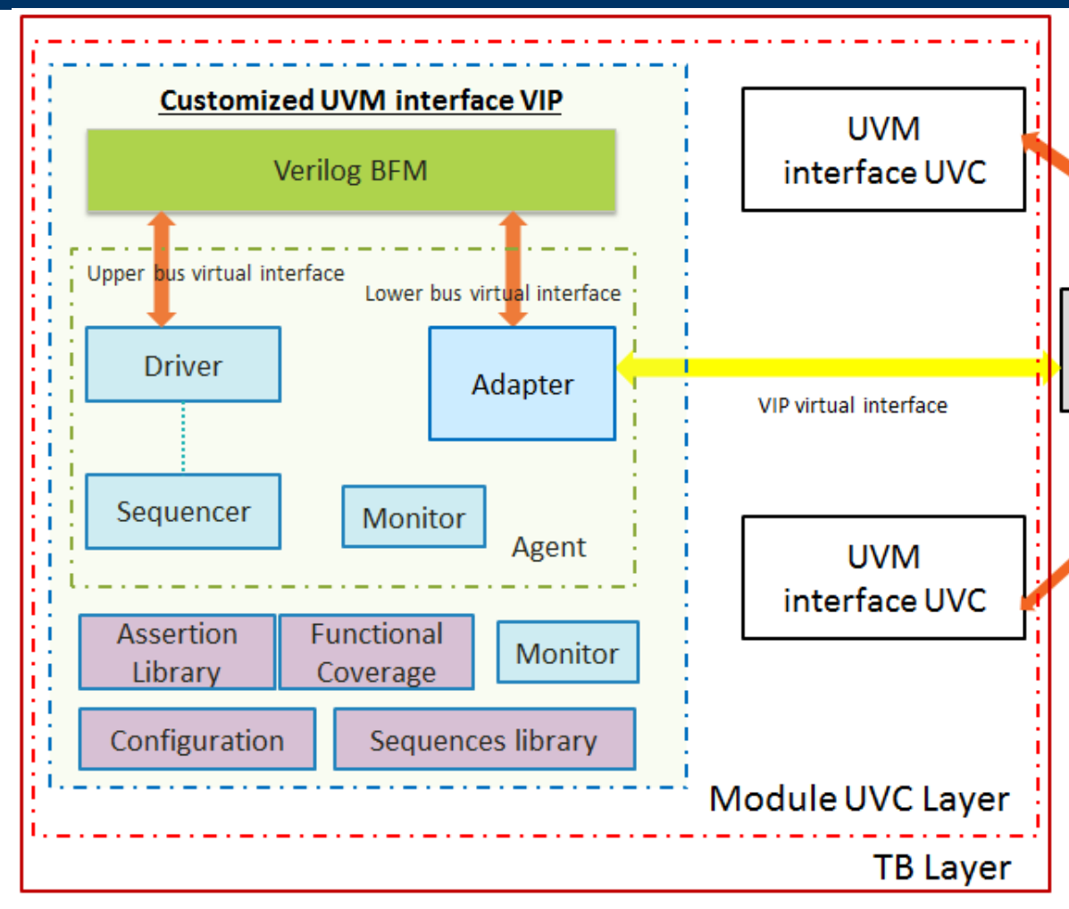
the master IP designer could decouple the RTL into a clean Verilog BFM model with two bus interfaces: the one called the **lower bus** is there to communicate with the slave IP and the other called the **upper bus** is to get drive from high layers. The designer also makes the upper bus timing simple and easy to drive. Master IP has slave VIP to verify itself in its standalone.



Simple Solution to create upper iUVC



Proposed Solution



- It reuses and wraps the Verilog BFM model which is invisible for users.
- We create one internal upper bus interface and one VIP interface (which is for the user to hookup to DUV). Actually, the internal lower interface is a copy of the VIP interface. The two internal interfaces are bind to Verilog BFM and invisible for users. These interfaces are parameterized and scalable allowing requirements to change.
- The abstract classes are created in SV interfaces. They assist to create the parameterized UVM component instances and provide APIs such as the tasks/functions/events defined in legacy BFM.
- The UVM driver is a UVM component and drives the sequence item to Verilog BFM through the upper bus interface.
- The adapter is a UVM component, and it acts like a bridge and passes down the bus data between the VIP interface from DUV and the lower bus interface to the Verilog BFM model at run_phase. The implementation can ensure no re-timing or pipelining. We also support few error injects in the adapter.
- The agent monitor spies the upper bus interface and sends back the bus data to sequence by uvm_event if necessary, e.g. sequence needs to get the return data to calculate the CRC to send the next sequence item.
- The iUVC's monitor spies the VIP bus interface (Lower bus) and broadcasts the transaction item to other subscribers, e.g. predictor or scoreboard.
- There is a bus protocol assertion library and written in assertion interface which will hookup to DUV by SV binding methodology and verify the timing of VIP interface.
- It has the built-in sequence library to verify the basic handshake between VIP and DUV.
- The functional coverage will cover the VIP's protocol and be vertical usable.
- Only the UVM driver and adapter are parameterized, the other UVM components are non-parameterized. That will be easy for VIP integration.
- To avoid Verilog BFM compile conflict with the same module name when reused to SoC level, we define the macro to exclude it from SoC compile.

Importing Photographs

we create the parameterized upper bus interface and the abstract class 'slave_vip_driver_api_base'. In the m_slave_upper_bus interface, we implement the pure virtual functions in the vip_driver_api class which extends the base abstract class

```
virtual class slave_vip_driver_api_base extends uvm_object;
  function new (string name = "salve_vip_drv_api_base");
     super.new(name);
  endfunction
  pure virtual function slave_vip_driver_base create_slave_driver (string m_name, uvm_component m_parent);
  .....
endclass
`define SLAVE_VIP_UPPER_IF_PARAM_DECL \
  LDATA1_WIDTH=`VIP_DEF_WIDTH1, \
  LDATA2_WIDTH=`VIP_DEF_WIDTH2, \
  ...
`define SLAVE_VIP_UPPER_IF_PARAM_MAP .LDATA1_WIDTH(LDATA1_WIDTH),.LDATA2_WIDTH(LDATA2_WIDTH),...
interface m_slave_upper_bus_if #(`SLAVE_VIP_UPPER_IF_PARAM_DECL) (
  input clk,
  input rst_n,
  inout [LDATA1_WIDTH-1:0] data_SLV2BFM;
  inout [LDATA2_WIDTH-1:0] data_BFM2SLV;
  .....
  );
  .....
  import uvm_pkg::*;
  `include "uvm_macros.svh"
  import slave_vip_uvc_pkg::*;
  class vip_driver_api extends slave_vip_driver_api_base;
    function new(string name = "vip_driver_api");
      super.name(name);
      uvm_config_db#(slave_vip_driver_api_base)::set(uvm_top, name, "vip_driver_api", this);
    endfunction
    function slave_vip_driver_base create_slave_driver (string m_name, uvm_component m_parent);
      slv_drv = vip_slave_driver#(`SLAVE_VIP_UPPER_IF_PARAM_MAP)::type_id::create(m_name,m_parent)
    endfunction
  ...
  endclass
  vip_driver_api driver_api = new($psprintf("%m"));
endinterface
```

The Slave VIP adapter code example.

```
virtual class slave_vip_adapter_api_base extends uvm_object;
  function new (string name = "salve_vip_adptr_api_base");
     super.new(name);
  endfunction
  pure virtual function slave_vip_adapter_base create_slave_adaper (string m_name, uvm_component m_parent);
  pure virtual function bit [LDATA1_WIDTH-1:0] get_data1();
  pure virtual function void set_data1( bit [LDATA1_WIDTH-1:0] dat);
  pure virtual task toggle_data1();
  .....
endfunction

`define SLAVE_VIP_LOWER_IF_PARAM_DECL \
  LDATA1_WIDTH=`VIP_DEF_WIDTH1, \
  LDATA2_WIDTH=`VIP_DEF_WIDTH2, \
  ...
`define SLAVE_VIP_LOWER_IF_PARAM_MAP .LDATA1_WIDTH(LDATA1_WIDTH),.LDATA2_WIDTH(LDATA2_WIDTH),...
interface m_slave_adpter_bus_if #(`SLAVE_VIP_LOWER_IF_PARAM_DECL) (
  input clk,
  input rst_n,
  inout [LDATA1_WIDTH-1:0] data_SLV2ADPT;
  inout [LDATA2_WIDTH-1:0] data_ADPT2SLV;
  .....
  );
  .....
  import uvm_pkg::*;
  `include "uvm_macros.svh"
  import slave_vip_uvc_pkg::*;
  class vip_adpater_api extends slave_vip_adapter_api_base;
    function new(string name = "vip_adapter_api");
      super.name(name);
      uvm_config_db#(slave_vip_adapter_api_base)::set(uvm_top, name, "vip_adapter_api", this);
    endfunction
    function  slave_vip_adapter_base create_slave_adaper (string m_name, uvm_component m_parent);
      slv_adpter = vip_slave_adpater#(`SLAVE_VIP_LOWER_IF_PARAM_MAP)::type_id::create(m_name,m_parent)
    endfunction

    function bit [LDATA1_WIDTH-1:0] get_data1();
      return data_SLV2ADPT;
    endfunction
    function void set_data1( bit [LDATA1_WIDTH-1:0] dat);
      data_SLV2ADPT = dat;
    endfunction
    task toggle_data1();
     @(data_SLV2ADPT);
    endtask
    .....
  endclass
  vip_adapter_api adapter_api = new($psprintf("%m"));
endinterface
```

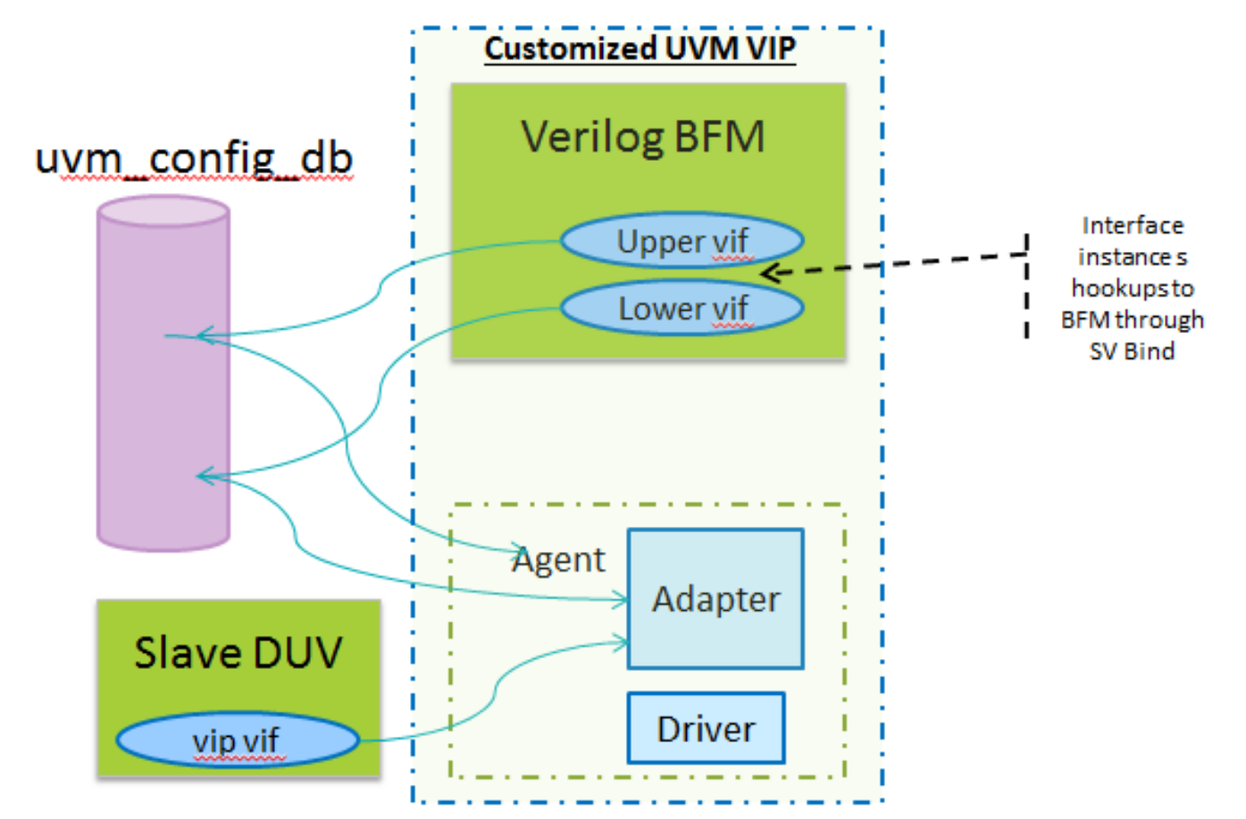
Interface binding and connection between the BFM/RTL and UVM component

```
class slave_vip_agt extends uvm_agent;
   vip_driver_api     driver_api;
   vip_adpater_api    adapter_api;

   slave_vip_adapter_api_base  adapter;
   slave_vip_driver_base       driver;

   function void build_phase(uvm_phase phase);
     ...
     if(!uvm_config_db#(slave_vip_adapter_api_base)::get(uvm_top, cfg.lower_vif_path,"vip_adapter_api",adapter_api))
       `uvm_fatal(get_type_name(),"Could not get adapter_api")
     if(!uvm_config_db#(slave_vip_driver_api_base)::get(uvm_top, cfg.upper_vif_path,"vip_driver_api",driver_api))
       `uvm_fatal(get_type_name(),"Could not get driver_api")
     adapter  = adapter_api.create_slave_adaper("adapter",this);
     driver   = driver_api.create_slave_driver("driver",this);
    ...
endclass
```

Bridge between BFM/RTL and UVM Components



Extending to an SoC

At the early stage of SoC verification, We really want to verify the slave IP integration and basic function as early as possible. The master IP design could deliver the dummy stub and make the output signal as weak driving strength to avoid the multi-driven case. We reuse the IP level UVM sequences and customized UVM VIP on the SoC level, replace the master IP RTL with dummy stub in the Verilog file list, and execute the initial sequence on the VIP as active mode to the drive slave. When the master IP initial verification is available for basic integration and functions, it could do the handshake with slave IPs. Later, the passive mode existing customized UVM VIP could collect the functional coverage, checking the bus protocol timing with built-in supports.

Efforts Compare

Effort to create the proposed customized UVC VIP.
•Decouple the master RTL into upper/lower BFM. It depends on RTL designer.
• Upper layer UVC. One person in 100% effort within 1 week.
•Lower layer UVC adapter. One person in 100% effort within 2 days.
•Interface BFM tasks. One person in 100% effort within 2 days.
Effort to create the Master UVC VIP.
•It's totally un-predictable because it depends on the complexity of bus timing. It will introduce upgrade issue when master RTL timing changes.

Few Limitations

- Irritation/error injection on lower layer interface. Even we could do few error injections in the adapter, but adding delayed responses, back pressure on receiving or inserting errors at lower layer interface are nearly impossible. BFM model will always drive interface in the same way, based on its state machine and driver code.
- If the master RTL has minor changes, the possibility of bug missing should be low and the Verilog BFM deliver time should be short on the master IP verification point of view. However, if the master RTL has big changes, the customized UVC VIP mainly depends on verification quality and the time from master IP verification team.

To fully address the issues above, verification team should schedule the resource to create the proper lower layer UVC and build the protocol layering architecture with existing upper layer UVC.

Conclusion

With our verification requirements, we use the above approach to create a customized UVM VIP which reuses the decoupled Verilog BFM model, abstract classes and SV bind methodology. When the RTL designer decouples the RTL design into Verilog BFM, the verification engineer could build the VIP infrastructure and define the interface. The VIP can sync up with RTL design bus interface timing updates without any changes. By adopting the parameterized interface, the VIP can be more scalable. The base abstract class can be used by different concrete API classes in the customized VIP. Making use of the existing general iUVC template and reusing the Verilog BFM model, the creation effort can be significantly reduced from scratch.

The proposal depicted in this paper can be extended to the other prototype such as, a PCI, I2C like interface using bi-directional interface or an AMBA like interface with ORed structure and separated channels. The key is to decouple the RTL design into a clean BFM with a simple upper bus interface. The verification engineer and RTL design engineer should consider the reusable solution for both design and verification. In general, it only needs to decouple the specific module, but not the whole design. The less it impacts the RTL design the better.