Designing Portable UVM Test Benches for Reusable IPs

Xiaoning Zhang
Baosheng Wang
Terry Li
Advanced Micro Devices, Inc.
1 AMD Place
Sunnyvale, CA 94088-3453

***Abstract*- System on Chip (SoC) methodology through reusable Intellectual Properties (IPs) has been widely deployed at both Advanced Micro Devices, Inc. (AMD) and the general semiconductor industry. Due to large varieties of SoCs, designing portable universal verification methodology (UVM) based test benches to validate the IP and integrate this IP into different SoC verification environments is challenging. In this paper, we present a generic method to design and implement a portable UVM test-bench from scratch for highly reusable IPs. Such a UVM-based verification environment can be re-used in various SoC applications for IP-level sign off and SoC-level integration. The contributions of this paper are mainly two-fold. On one hand, our UVM test bench development process is very generic. We start from DUT analysis in terms of interface grouping, functionality classification based on all possible IP usage models and directory structures of UVM components. In other words, our UVM test bench development methodology is very suitable for verification engineers who even have no previous experiences of UVM test bench bring up. Moreover, our test bench framework is also very generic. We deploy the generic UVM environment so that both the IP-specific components and the rest of the UVM components can be modulated. On the other hand, at an early design stage, we architect our test bench by fully considering the future integration requirements. Those considerations mainly include 1) using SystemVerilog bind method to attach our test bench into the IP top-level module; 2) always coding UVM components at both IP mode and IP usage mode. As a result, our test bench is transparent to higher-level verification environments, no matter whether they are UVM-based or C++ based. We demonstrate the effectiveness of our methodology using one of AMD power management IPs in this paper.**

I. Introduction and DUT Analysis

UVM (Universal Verification Methodology) is currently the prevalent digital design verification methodology in semiconductor industry [1]. It defines a framework that has been proven very effective for block level verification since prototype of a UVM test bench and its corresponding components can be very quick. The well-defined architecture allows users to easily design their key components in the framework so that they can focus on project specific functions to achieve data-path sanity verification in a shorter period. Such architecture also allows multiple components to be developed at the same time by different verification engineers. During the test plan execution, expanding existing functions or adding a new function is very straightforward. Moreover, this architecture and TLM connections allow components to be reused as much as possible. With this methodology, verification owners can use flexible constraints to generate abundant stimulus. UVM supports functional coverage collection and register handling. A layered and modular structure is ideal for integration or reuse at upper block level, especially for an SOC or CPU core which may contain thousands of IPs.

At AMD, our team verifies a set of power management IPs which are widely used in the SOCs and CPU cores. Those IPs are attached to functional blocks in the system and manage the voltage, frequency and temperature so that the chip can consume power very efficiently. They are mostly digital and/or mixed blocks. In this case, an ideal test bench for those reusable IPs is the one that can thoroughly verify all IP-level functions while enables seamless integration at the upper level verification environment. In other words, the prospective test bench should have the following features:

1) The test-bench should be generic enough for all the standalone IPs in the set to ensure methodology consistency and reduce maintenance burden;

2) The monitor and scoreboard must be reusable in higher level after configuring the test-bench and verification components into passive mode.

Luckily, UVM methodology provides such a powerful capability to meet our requirements.

To illustrate our design and implementation of such a test-bench and UVM Components (UVC), the example of an IP known as digital voltage regulator (DVR) shown in Figure 1will be used to present our methodology [2]. The

DVR is designed to regulate the voltage to the block it controls. It can stabilize the block voltage to a reference value, and can also smooth the curve when the block voltage jumps between power status of the CPU. Clearly, this can be a highly reusable IP since it provides wide range of voltage inputs while still allowing users to configure it for different usages.

A simplification view the DVR is shown with the following block diagram:

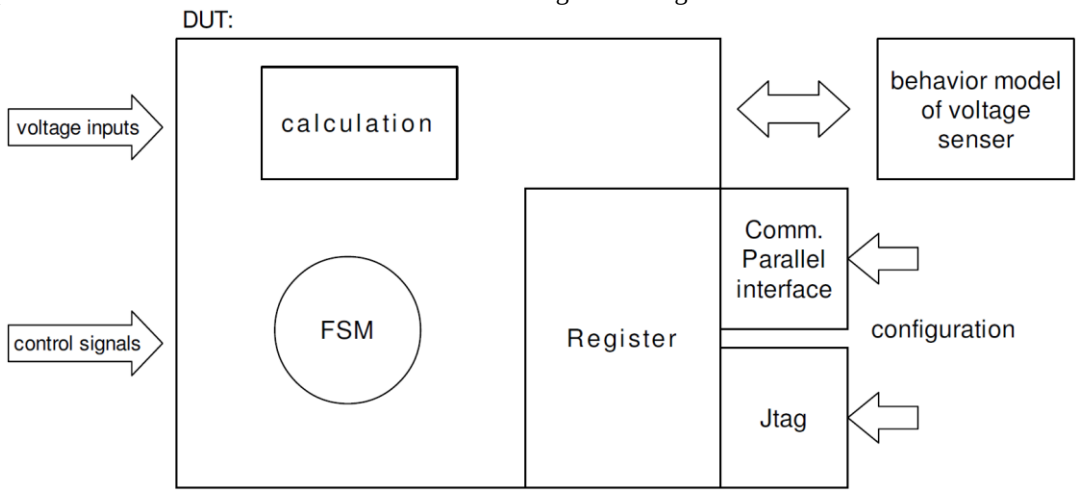

Figure 1. Simplified view of DVR datapath

Without paying too much attention to the protocols and functions of the block, let us review it from a verification perspective.

1) There are 2 interfaces to configure internal registers: the generic JTAG interface and an AMD-private parallel bus interface.

2) The DVR can accept an input voltage (reference voltage) and control the voltage of the block it manages in several states: regulator bypass state, regulation state and exit regulation state. The direct inputs to the DUT include the state control signals that go into the internal FSM and voltage input related signals that feeds the calculation logic. These control signals and voltage data signals can be grouped together into a direct interface.
3) There is a behavior model used in verification which models the analog voltage sensor, provided by the RTL designer.
4) There are usually 14 properly-named sub-directories to be initialized for containing all necessary UVM files, such as “interface”, “environment”, “driver”, “agent” and so on.

The rest of the paper is organized as follows. Section II describes how to design generic UVCs while Section III discusses the developments of the portable test bench and its corresponding connectivity. The integration-aware portion of the test bench development process is shared in Section IV. Section V summarizes.

II. Design UVC

We demonstrate our method through the UVC design of DVR DUT shown above.

Logically, because there are three interfaces, three agents will be implemented to communicate with these interfaces. The three agents are named as JTAG agent, parallel agent and direct agent in this paper. Figure 2 shows the layout of the UVC. JTAG and parallel agents are register access agents. Their read and write functions are implemented based on serial and parallel protocols respectively.

Each agent contains a sequencer, a driver and a monitor. The drivers and monitors get handles of the virtual interfaces through UVM configuration database (configDb).

Sequences start on the sequencer of the agent in test cases (test cases usually hold the handle of the sequencer in a virtual sequencer instance). The sequence creates data and control information that are packed as transactions. Transactions are sent to drivers through TLM connections. Drivers create the timing patterns for these transactions and drive them onto interfaces.

On one hand, when read register sequence is implemented, “item_done (tx)” and “get_response (tx)” interaction mechanism of UVM is used in the driver and sequencer pair. In this mechanism, the sequence starts on a sequencer, from which, the transaction is passed to driver through TLM connection. The driver can access the ‘real’ wires of DUT inputs/outputs through virtual interface handle. It will translate values in transaction into pin wiggling on the wires. On the other hand, it collects output values from DUT and wraps them as transactions in the driver. The driver then takes advantage of “item_done (tx)” and “get_response (tx)” interaction to send the transaction back to sequence. In this way, the test can start the read sequence and receive the output of DUT in the same object. This provides a way to quickly self-check register access, without relying on the checking in scoreboard of the register model.

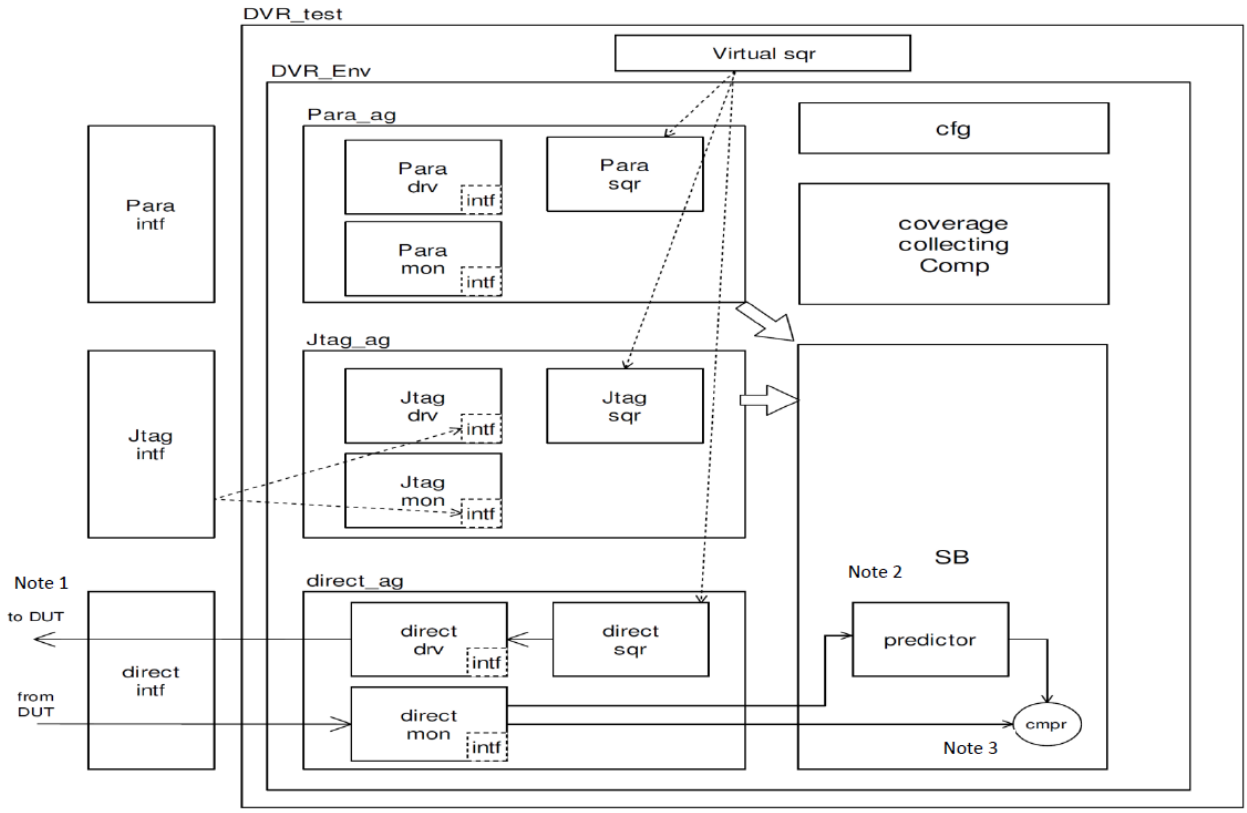

Figure 2. DVR UVM Environment.

There is a scoreboard instantiated in the UVM ENV. The scoreboard class contains predictors and comparators for verifying functional correctness of the DVR regulation states and voltage values: the UVM monitors keep monitoring signals output from DUT, bring back the state signals and voltage data output from DUT (refer to Note 1 in Figure 2). Parallel to DUT monitoring process, the monitor also has a process to monitor the stimulus input into DUT sent by drivers wiggling on interface wires. The monitored stimulus input is sent to predictor inside scoreboard via TLM analysis port connections (refer to Note 2 in Figure 2). The predicted transactions are compared with real DUT output transactions monitored by the same monitor. In fact, the monitored DUT output transactions are sent back to the scoreboard comparator via the other TLM analysis port connection (refer to Note 3 in Figure 2). As a result, there two analysis ports and two queues in the comparator to receive and hold the transactions coming from two TLM analysis port connections, i.e. from predictor and real RTL output. The comparator is a function running in the forever loop of the scoreboard run phase. It compares the transactions from these two queues as long as both the queues are not empty.

The functional coverage component is also instantiated in the UVM ENV. It is a container component of all the covergroups and coverpoints defined to measure all kinds of stimulus from three interface agents. The handle of this coverage collecting component is passed down to the coverage collecting callback instances through the “new ()” method so that the callback instances can access any covergroup and coverpoint instances in the container through the pointer and hierarchical references. These callbacks are monitor callbacks. Their base classes are defined in the

monitors they would hookup to, and implementation of these callback classes can be written in the coverage collecting component. There are a set of UVM macros to register these callback classes to the corresponding monitor classes and add the callbacks into callback database in the UVM ENV. The coverage collecting functions defined in the callback classes will be called in the run phase of the UVM monitor, the position of the call is usually right after the transaction is created and sent to analysis port. Inside the coverage collecting function of the callback, values in the transactions are unpacked, corresponding covergroups are triggered to sample and hit on coverpoint bins.

During the simulation, functional coverage will be collected. The result is stored in database file. We have a flow including a series of scripts to process the database file into certain format and upload the result to a web-based coverage browser. Using this browser, the coverage information can be displayed in a customized way for coverage closure.

There is also a configuration object instantiated in the ENV. It carries information to configure all necessary fields and variables in the environment, such as the width of parallel bus, on/off of the JTAG agent and active/passive mode of the agents. Because of our test bench and UVC size, we only have this one layer of the configuration object. There is no embedded configuration object for a single agent. The configuration object is passed down to each component. Inside the component class, the "configure_phase" will look up the configuration object by string search to find out the value for specific field or variable.

One important usage case of configuration object in this paper is that an "ifdef" switch is set up to control the variables in configuration object, and these variables will be further passed into component class. Hence, UVC can be set up to certain status such as active mode in block level verification and passive mode in upper level integration depending on the "ifdef".

III. TEST BENCH LAYOUT AND CONNECTIVITY

UVC provides all stimulus related functionality and the test bench (TB) module takes care of the DUT and UVC connectivity with portable and flexible structures. In this paper, the test bench is defined as a module, as shown in Figure 3. There are UVC, interfaces, clock/reset generation logic and a System Verilog behavioral model instantiated in the TB module.

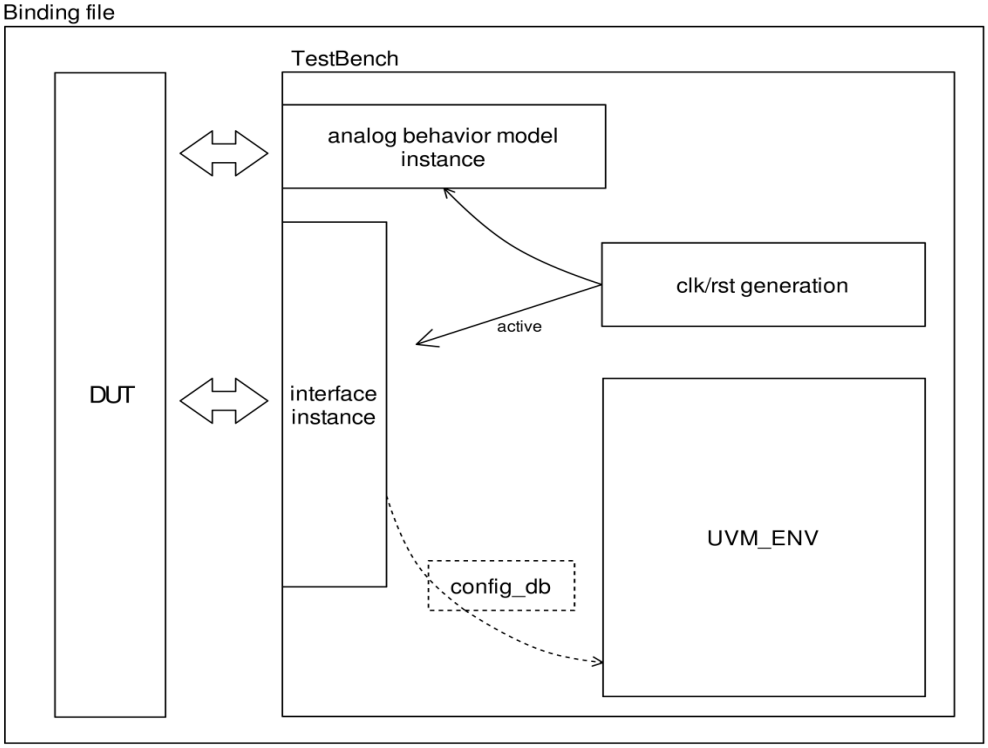

Figure 3. Testbench and binding.

All three interfaces (parallel interface, JTAG interface and direct interface) are instantiated as SystemVerilog interfaces in TB. The port signals of the TB module are defined as "inout" type, i.e., bidirectional ports. Signals in

interface are declared as “logic” type. TB module ports and interface signals are connected together through instance pin assignments, and thereby the connection is also bidirectional.

Inside interfaces, clocking blocks define the direction of interface signals in both active and passive modes. Because the interface pointers are passed to UVM drivers and monitors, according to the active/passive configuration in the configuration object, the same interface signal will be driven or monitored from different clocking blocks.

Clock/reset generation logic is instantiated in TB module. This clock/reset module drives the clock of both the DUT and the SystemVerilog behavior model, which is supplied by the RTL designers for verification purpose. The SystemVerilog model is a behavioral model to mimic analog sensor that can interact with the regulation logic of DVR. The behavioral model has 5 ports connected to DUT. The special voltage coding is implemented by this 5-port interface, passing voltage information back and forth between the model and DUT. The model provides reasonable random feedback to the DUT according to the voltage output from DVR. This random feedback is just a mimic of the real analog sensor in the circuit. There are “ifdef” switches that control the direction of the clock connection: when it is at block level, this local clock/reset generation is enable, otherwise, it will be turned off (e.g. when integrating to upper level, the clock is driven from that level. Clock becomes an input to this block level TB since the upper level has its own test bench and clock generation logic).

In the block level, the base test class declares a handle of the UVC, and the path of the UVC instance in TB module is passed to this handle, as shown in Figure 3. All block level test cases extend this base test, and thus can access the UVC hierarchy and start sequences on the sequencers in the UVC agents.

IV. INTEGRATION

When an IP is deployed in the upper level, the only thing which the integration folks need to worry is that the IP has to be used as intended to avoid false negatives. Since the IP-level stimulus can no longer be able to reuse, it is very important that the IP-level checkers can be reused to prevent illegal usages. There are generally two approaches to integrate IP-level checkers and monitors into upper level verification environments: one is to directly hook them up while the other is to link the whole TB in the upper level, as shown in Figure 4.

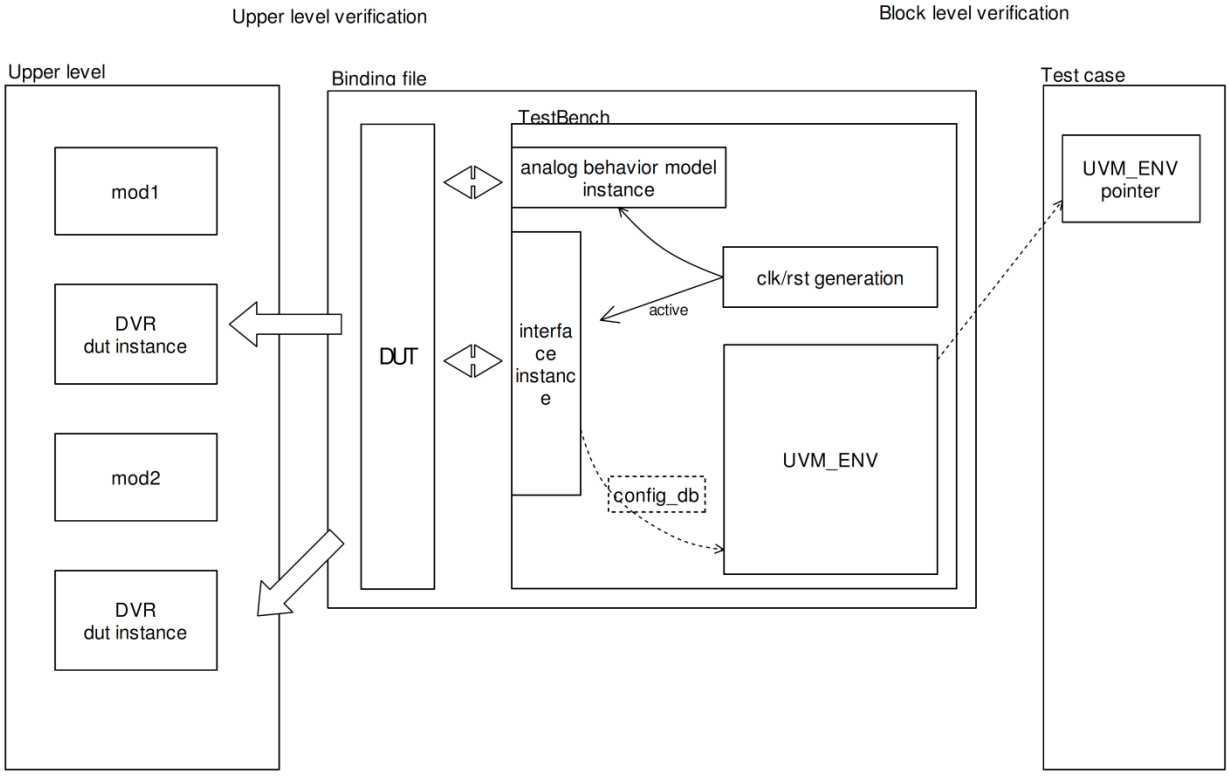

Figure 4. Integration to upper level.

To ensure that it is completely portable and can be seamlessly reused in upper level, in this paper, the block-level test bench module is attached to DVR DUT module through SystemVerilog bind method [3]. As a result, wherever the DUT is instantiated, TB module will also be linked under the DUT in the hierarchy. The upper level of DVR and

other IPs in the set can be CPU cores or a generic SOC. Usually, there will be multiple instances of the DVR IP, by binding it in this way, every DVR can automatically have a test bench and UVC attached to it. The upper level verification engineers do not need to touch anything in the block level bench. Such binding structure can be illustrated by the code snippet below for demonstrating the binding syntax with fake upper level RTL.

```
bind DVR_rtl DVR_tb_module#(
   .DVR_PARAM1   ( DVR_PARAM1 ),
   .DVR_PARAM2   ( DVR_PARAM2 ),
   .DVR_PARAM3   ( DVR_PARAM3 )
)
DVR_tb_inst(
//use .* if the tb port list is same as DUT
   .clk              ( clk           ),
   .rst              ( rst           ),
   .dvr_in_port1     ( dvr_in_port1  ),
   .dvr_in_port2     ( dvr_in_port2  ),
      ...
   .dvr_out_port1    ( dvr_out_port1 )
);
```

```
module core_level_rtl #(
   CORE_PARAM1 = 1,//or other values
   ...
)(
input core_clk,
input core_rst,
input core_in_port1,
...
output core_out_port1,
...
);

wire core_conn1;
...

upper_stream_module#(
        .PARAM (...
)
before_DVR(
        .clk        ( core_clk      ),
        .rst        ( core_rst      ),
        .in_port1   ( core_in_port1 ),
           ...
        .out_port1  ( core_conn1    ),
           ...
);

//anywhere the DUT DVR_rtl is instantiated,
//you can find the "...DVR_rtl_inst.DVR_tb_inst"
//under it through heirarchical call. This is
//because DVR_tb_module is bound to DVR_rtl module.
DVR_rtl DVR_rtl_inst#(
   .DVR_PARAM1   ( CORE_PARAM1 ),
   ... ...
)
)(
        .clk             ( core_clk      ),
        .rst             ( core_rst      ),
        .dvr_in_port1    ( core_conn1     ),
           ...
        .dvr_out_port1   ( core_out_port1 )
);

...

endmodule
```

When integrating to upper level, there will be no test cases to start sequences on the sequencers and drivers path in block level UVC. Through ifdefs, all the agents will be turned into passive mode, the sequencers and drivers path will be disabled (as mentioned, this is done by using configuration object to set passive to the mode variables inside each agent). The signals in the interface which were driven by the UVM drivers in block level will now be driven by the upper-stream RTL in the upper level data-path. Because the ports in the test-bench module are defined as “inout” and the signals in the interfaces are “logic”, the connections are bidirectional. Therefore, the upper-stream RTL outputs can also drive into the UVM components through test-bench ports and interface signals. As these interface signals are included in both driver clocking block and monitor clocking block, UVM monitors can monitor them to wrap the upper-stream RTL stimulus into transactions to send to predictor inside the scoreboard. To further reduce the burden of the IP integration, we design the test bench and its UVCs passive mode at default. The following code snippets show the structure of test bench module and interface.

```
module DVR_tb_module #(
        ... ...
)(
inout wire      clk,
inout wire      rst,
inout wire      dvr_in_port1,
   ...
inout wire      dvr_out_port1
);

//import uvm pkg for using uvm configDB in the module
import uvm_pkg::*;


logic clk_tb;
logic rst_tb;

//clock generation logic
... ...


//instantiate interface
DVR_interface dvr_interface_inst(clk_tb,rst_tb);


`ifdef PASSIVE_MODE_FOR_BLOCK_LEVEL
   assign clk = clk_tb;
   assign rst = rst_tb;
   assign dvr_in_port1 =
dvr_interface_inst.drive_sig1;
      ...
   assign dvr_interface_inst.mon_sig1 = dvr_out_port1;
`else
   assign clk_tb = clk;
   assign rst_tb = rst;
   assign dvr_interface_inst.drive_sig1 =
dvr_in_port1;
      ...
   assign dvr_out_port1 = dvr_interface_inst.mon_sig1;
`endif

//behavioral model instantiation and connectivity


//instantiate the UVC
//and pass the interface handle into it
DVR_UVC dvr_uvc_inst;
string  m_name;

initial begin
   m_name = $psprintf("%m");
   dvr_uvc_inst = dldo_env::type_id::create({m_name,
".dvr_uvc_inst"}, uvm_top);

   dvr_uvc_inst.interface_inst = dvr_interface_inst;
end
endmodule
```

```
interface DVR_interface (input bit clk, input bit
rst);
   logic drive_sig1;
      ...
   logic mon_sig1;

   clocking driver_clocking_block @(posedge clk);
           output drive_sig1;
           ...
   endclocking : driver_clocking_block

   clocking mon_clocking_block @(posedge clk);
           input drive_sig1;
           ...
           input mon_sig1;
   endclocking : mon_clocking_block

endinterface : DVR_interface
```

At the upper level, there will usually be a dummy test to run through all the phases of UVM simulation. Since the upper level verification team typically wants to completely isolate anything belong to IP level (e.g. the DVR the UVC ENV) in their tests, we want to have UVC instantiated in the test bench module, and only pass the handle of UVC to the block level test cases, no matter whether the upper level TB is C++ or UVM based.

V. SUMMARY

In this paper, a method of building the entire UVM environment from scratch to verify our highly reusable IPs is reviewed. The DVR test-bench and its UVC are used as an example to illustrate our methodology from verification requirements analysis to test-bench architecture design and UVC implementation. With our methodology, IP verification team not only can robustly verify DUT at the block level but also could effectively integrate and reuse

the testability in upper level. With this style of test bench and its corresponding UVC design, both the vertical reuse (our verification environment is generic and suitable for a series of similar DUTs) and horizontal reuse (reuse our test-bench and UVC testability in upper level) can be achieved.

VI. REFERENCE

[1] Height, Hannibal. A practical guide to adopting the universal verification methodology (UVM). Lulu. com, 2010.

[2] Advanced Micro Devices, Inc. "AMD Digital Voltage Regulation Design Specification", *AMD Power Management Design Specification*, 2014, pp. 1-30

[3] Accellera Organization, Inc. "SystemVerilog 3.1a Language Reference Manual", 2004, Chapter 17.15, *http://www.eda.org/sv/SystemVerilog_3.1a.pdf*