Advanced Testbench Design using Reusable Verification Component and OVM

Viney Malik
M.E Student (ECE Deptt.)
NITTTR, Sec - 26, Chandigarh
India

Rajesh Mehra
Assoc. Professor (ECE Deptt.)
NITTTR, Sec - 26, Chandigarh
India

Surender Ahlawat
Senior Manager
Mentor Graphics
India

ABSTRACT
The paper describes the additional proven techniques for creating highly effective testbenches. This paper presents topics that are likely to be used by most test-benches. Samples of the techniques, as well as the underlying concepts, are presented. The paper shows several ways to use VIP with OVM technology and provides the knowledge to customize, modify, and extend the techniques to suit the needs of SoC designers. The basic steps to create a first constrained random testbench with VIP and OVM is also presented. It can be a template to develop more complex and powerful test-benches using other computing methods and features of OVM and VIP.

Keywords
Open Verification Methodology, Verification Intellectual Property, System on Chip, Design Under Test, Transaction Level Modeling.

1. INTRODUCTION
Open Verification Methodology (OVM) [1]–[2] is a complete verification methodology that codifies the best practices for development of verification environments targeted at verifying large gate-count, IP-based SoCs. Verification productivity stems from the ability to quickly develop individual verification components, encapsulate them into larger reusable verification components, and reuse them in different configurations and at different levels of abstraction. OVM supports "bottom-up" reuse by allowing block-level components and environments to be encapsulated and reused as blocks that can be composed into a system. "Top-down" reuse allows transaction-level verification environments to be assembled with system-level models of the design, and then reused as the design is refined down to RTL. The remainder of this paper is organized as follows. Section 2 presents features of OVM for modular communication between components. Section 3 explains about reusable verification components. In Section 4, Transaction Level Modeling with Multiple Abstraction Levels is provided in VIPs by providing complete stimulus control to user at any abstraction level. Section 5, explains the steps for building a test environment using Verification IP (VIP) and OVM. Section 6 draws final conclusions.

2. FEATURES OF OVM
The Open Verification Methodology (OVM) is an open-source System Verilog class library and advanced methodology that defines a framework for reusable verification IP (VIP) and tests. It is based on the IEEE 1800 System Verilog standard and provides building blocks (objects) and a common set of verification-related utilities.

The features of OVM are as under:

i. Data Design - Infrastructure for class property abstracting and simplifying the user code for setting, getting, and printing property variables.

ii. Stimulus Generation - Classes and infrastructure to enable fine-grain control of sequential data streams for module- and system-level stimulus generation. Users can randomize data based on the current state of the environment, including the Design Under Test (DUT) state, interface, or previously-generated data. Users are provided out-of-the-box stimulus generation, which can be customized to include user defined hierarchical transactions and transaction streams.

iii. Building and Running the Verification Environment - Creating a complete verification environment for a SoC containing different protocols, interfaces and processors is becoming more and more difficult. Base classes are provided for each functional aspect of a verification environment in the System Verilog OVM Class Library [3]. The library provides facilities for streamlining the integration of user-defined types into the verification environment. A topology build infrastructure and methodology provide users flexibility in defining required testbench structures. A common configuration interface enables the user to query and set fields in order to customize run-time behavior and topology.

iv. Coverage Model Design - Best-known practices for incorporating coverage into a reusable verification component.

v. Built-in Checking Support - Best-known practices for incorporating physical- and functional layer checks into a reusable verification component.

OVM [4] has features that greatly help with reuse such as the configuration mechanism, class factories, TLMs and sequences.

3. VERIFICATION & MODELING
The reusable components, called intellectual property (IP) blocks or cores, are typically synthesizable register-transfer level (RTL) designs (often called soft cores) or layout level designs (often called hard cores). The concept of reuse can be carried out at the block, platform, or chip levels, and involves making the IP sufficiently general, configurable, or programmable, for use in a wide range of applications [5]. Reusable verification components are system which works at multiple abstraction levels and one only need to replace Transaction Level Modeling (TLM) level master/slave with RTL master/slave once cores are ready. These are system verilog based OVM compliant verification component. [6].

Transaction Level Modeling with multiple abstraction levels is provided in VIPs by providing complete stimulus control to user at any abstraction level. For e.g. in OCP VIP there are 2 abstraction levels - Phase level (with Request, Data-handshake and Response phases) and Transfer level (Simple transfer and Burst transfer). It makes much easy for user to move from one abstraction level to another. This provides user a flexibility to provide different pattern of delays between phases by running stimulus at Phase level, this helps in running different scenarios of in order as well as out of order stimulus. It also provides full freedom to control

pipelining in transactions. [7]. As mentioned above, VIP provides flexibility of running stimulus at mixed abstraction levels. Further, Simulator transaction viewing capability feature provides view of all the VIP transactions in its waveform display. The transactions hierarchically from top to bottom abstraction can be viewed in this display. Using this unique transaction viewing feature along with VIP extends level of debugging. VIP provides full logging capability for all the transactions. Logging can be turned on or off according to user requirement. [8].

Figure 1 exhibits a snapshot of VIP transactions viewing with simulator waveform transaction viewing. Both Phase and Transfer level transactions can be easily seen and parameters of the respective transaction can also be seen.

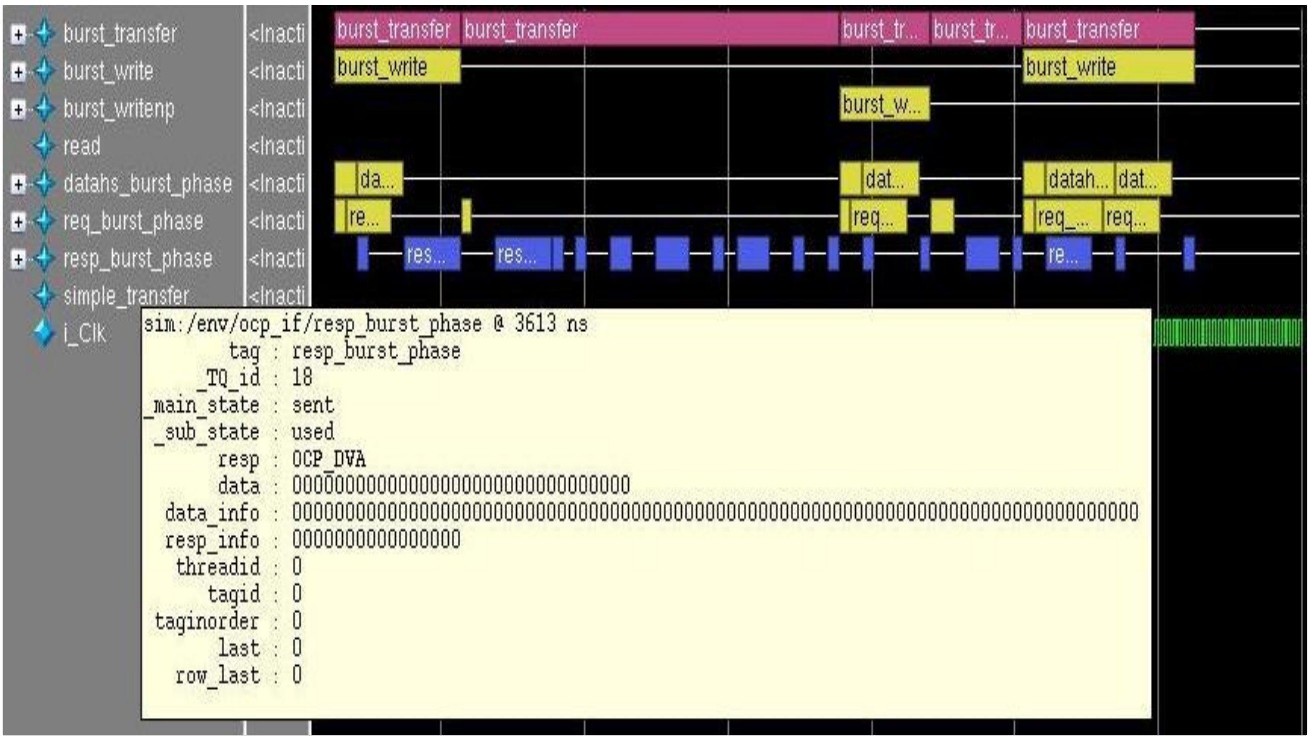

Figure 1: Exhibits a snapshot of VIP transactions viewing with simulator waveform

4. PROPOSED TESTBENCH DESIGN

VIP and OVM provide the option to rapidly build a well-architected, advanced verification infrastructure. In this section, a few basic OVM concepts and techniques have been applied to quickly achieve a basic constrained random testbench. The major steps to create a test environment using VIP and OVM are:

i. Creation of test environment
ii. VIP Configuration
iii. Constrained random sequence generation
iv. Control the test (Start and Stop)

i. Creation of Test Environment

The testbench is instantiated in a top-level module to create a class-based simulation environment. The top module contains the typical HDL constructs and a configuration class which contains a System Verilog interface. This interface is used to connect the class-based testbench to the DUT. The environment inside the testbench uses a virtual interface variable to refer to the System Verilog interface (Abstraction Adaptor / VIP BFM). Protocol checking assertion monitor can also be instantiated and connected to the DUT in this top level module. [9] - [12]. The basic architecture of a VIP is as shown below:

The main feature of this new architecture of test bench is that it requires very few connections of ports and exports of the TLM components to get the test up and running. In some cases it might even not require any connections to be done and just requires the instantiation of the TLM components. VIP includes VIP_base library which takes care of the connections of ports and exports.

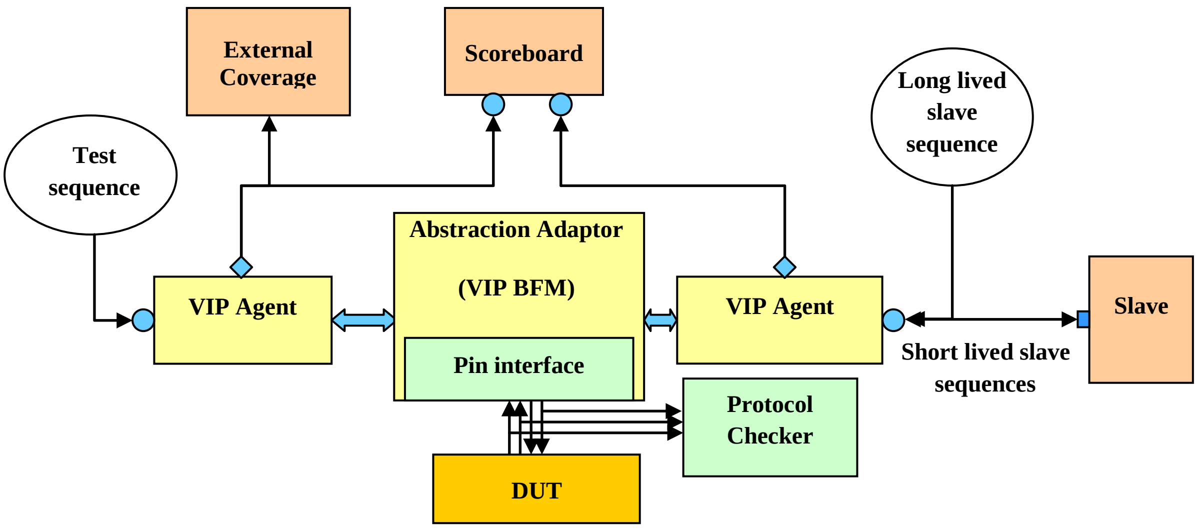

Figure 2: Basic Architecture of VIP

The agent at either end of the bus plays an active part in the protocol. A master agent might send requests to and get responses from the DUT, while a slave agent will get requests and send responses. Both master and slave agents will also have passive functionality. A monitor inside the agents observes interesting transactions on the bus and publishes those transactions for use inside or outside the agent.

```
// Top level testbench module
module env ();
  top_level_config top_cfg;      // MVC top level configuration object
  usb           usb_if(1'bz);  // MVC BFM interface
  mvc_env         env;
// Setting environment configuration
  initial
  begin
    top_cfg = new (usb_if);
    set_config_object("*", s_top_level_config_id , top_cfg , 0);
     mvc_env_config::configure_interfaces();
    env = new("env");
    mvc_barrier_pool::run_test();
  end
endmodule
```

ii. VIP Configuration

The configuration class contains a virtual System Verilog interface which is used to connect the testbench to DUT. Apart from this, it is used to configure the VIP. This includes passing the information how the VIP is going to be used and protocol related parameter values. This may also be used to instantiate internal passive components (scoreboard and coverage).

```
// MVC top level Configuration class
 class top_level_config extends mvc_env_config;
  typedef top_level_config this_t;
  typedef axi_vip_config   config_t;  // MVC BFM configuration class
  typedef virtual axi      axi_if_t;
  typedef axi_master_rw_nolock_transaction axi_rw_nolock_trans_t;
  typedef axi_master_rw_transaction axi_rw_trans_t;
  config_t m_master_config , m_slave_config;
  extern function new( axi_if_t axi_if );
  extern task do_configure_interfaces();
  extern local function void do_master_config( axi_if_t
  axi_if);
  extern local function void do_slave_config( axi_if_t axi_if );
  extern local function void do_common_config( config_t t ,
axi_if_t axi_if );
endclass
function top_level_config::new( axi_if_t axi_if );
  m_master_config = new();
  m_slave_config = new();
  do_master_config( axi_if );  // Master agent configuration
  do_slave_config( axi_if );   // Slave agent configuration
// Creation of master and slave agents and assigning config to agent
  m_configs["master"] = m_master_config;
  m_configs["slave"]  = m_slave_config;
endfunction
function void top_level_config::do_master_config( axi_if_t axi_if );
  do_common_config( m_master_config , axi_if );
  // Adding sequence item to library
```

```
  m_master_config.add_item_to_library(
axi_rw_nolock_trans_t::get_type() );

  // Setting master default sequence item

  m_master_config.set_default_sequence(
axi_random_sequence::get_type() , 10 );

  // Adding listener item which can be used by the passive
components

  m_master_config.set_analysis_component( "trans_ap" ,
"listener" ,

 mvc_item_listener #( axi_rw_trans_t )::get_type() );

endfunction

function void top_level_config::do_slave_config( axi_if_t
axi_if );

  do_common_config( m_slave_config , axi_if );

  //Setting slave default sequence item

  m_slave_config.set_default_sequence( axi_slave_sequence ::
get_type() , 1 );

endfunction

function void top_level_config::do_common_config( config_t
t , axi_if_t axi_if );

  // Master + Slave are TLM

  t.m_master_map     = TLM; // This can be set to RTL when
Master is DUT

  t.m_slave_map      = TLM; // This can be set to RTL when
Slave id DUT

  // VIP generates clock and reset

  t.m_clock_source   = TLM;

  t.m_reset_source   = TLM;

  t.m_write_data_before_addr = 0;

  t.m_write_addr_to_data_mintime = 0;

  t.m_write_data_to_addr_mintime = 0;

  t.per_instance = 0;

  t.coverage_name ="coverage";

  t.m_bfm = axi_if;

endfunction

task top_level_config::do_configure_interfaces();

  m_master_config.configure_interface();

endtask

endclass
```

iii. Constrained Random Sequence Generation

Sequence Writing

A bfm supports a set of sequence items. These items are different to normal sequence items in that they have methods which know how to apply the item to the bfm (in VIP_driver) and receive it from the bfm (in VIP_monitor). The VIP_driver and VIP_monitor are part of the VIP_base library which is provided with the VIP.

Using VIP sequence items is no different to using any other kind of item. The start and finish a VIP_sequence_item would be created like any other sequence item. The only restriction is that it has been done so from inside an VIP_sequence provided in the VIP_base library. While creating the sequence item, the sequence item properties can be randomized with constraints as shown in the below code snippet.

```
class apb3_test_sequence #( int SLAVE_COUNT   = 1 ,

                            int ADDRESS_WIDTH = 32,

                            int WDATA_WIDTH   = 32,

                            int RDATA_WIDTH   = 32 ) extends
mvc_sequence;

  typedef apb3_host_read
#(SLAVE_COUNT,ADDRESS_WIDTH,

                            WDATA_WIDTH, RDATA_WIDTH )
write_read_t;

  …

  task body();

    write_item_t write_item =
write_item_t::type_id::create("write_seq");

    read_item_t  read_item =
read_item_t::type_id::create("read_seq");

    …

    forever

    begin

      assert(write_item.randomize() with {write_item.addr
inside {[m_slave_start_address : m_slave_end_address]};});

      read_item.addr = write_item.addr;

      read_item.slave_id = write_item.slave_id;

       start_item( write_item );

      finish_item( write_item );

      start_item( read_item );

      finish_item( read_item );

    end

  endtask

  …

  endclass
```

iv. Control the test (Start/Stop)

Sequence Starting

Having written a sequence, it is required to start it on an agent. There are two ways to do this. One is to explicitly call start; the other is set the default sequence in the configuration.

Starting a sequence by calling start

```
class env …

  mvc_agent master_agent;

  …

  task run ();

    test_sequence_t test_seq;
```

```
   test_seq =
test_sequence_t::type_id::create("test_sequence");
   fork
     master_agent.mvc_export.start( test_seq );
     time_out();
   join_any
   global_stop_request();
  endtask
  task time_out();
   #10000;
   `mvc_report_info("sample_environment", "Test completed
due to timeout")
  endtask
  …
endclass
```

The run task above declares and creates a test sequence. It then calls the start method of the master agent's VIP_export. This export is connected to the sequencer, so the call to start is routed to the sequencer inside the agent.

In the code above, it is terminated either when a timeout or the sequence ends.

Starting a sequence by setting the default sequence

The sequence can be started by simply setting the default sequence in the configuration. This can be done as shown in the below code snippet:

```
 m_master_config.set_default_sequence(
axi_random_sequence::get_type() , 10 );
```

This means that axi_random_sequence will execute 10 random items on the master.

5. CONCLUSION

As evident from the result in the previous sections with the most latest technologies, VIP and OVM present new concepts and techniques. When applied effectively, the new practices will provide the maximum benefit from a constrained random verification methodology i.e. save verification time and effort, increase test effectiveness and coverage, increase reuse etc. The presented work provides an introduction to the use of VIP with OVM and opens avenues for researchers to work on VIP & OVM for new developments. It also, lays a solid foundation for creating advanced test-benches.

6. REFERENCES

[1] Mark Glasser"Open Verification Methodology Cookbook" 1st edition, Springer, 2009.

[2] "OVM Golden Reference Guide", version2.0, Doulos, 2008.

[3] Thomas L. Anderson, "Open Verification Methodology: Fulfilling the Promise of System Verilog" Information Quarterly (IQ) Volume 7, Number 1, pp. 52-54, 2008.

[4] Bryan Ramirez, Michael Horn "Parameters and OVM - Can't They Just Get Along?" Proceedings of Design and Verification Conference & Exhibition (DVCon '11), 2011.

[5] Stephen D'Onofrio, Ning Guo "Building reusable verification environments with OVM" proceedings of EDA Tech Design Forum - 08, pp. 1-9, 2008.

[6] A.Wakefield, B.J. Mohd, "Constructing Reusable Testbenches" Proceedings of the IEEE Conference, High-Level Design Validation and Test Workshop, pp. 151-155, 2002.

[7] Mikhail Chupilko, A. Kamkin, "A TLM-Based Approach to Functional Verification of Hardware Components at Different Abstraction Levels" Proceedings of the IEEE Conference, Test Workshop (LATW), pp. 1-6, 2011.

[8] L. Cai, D. Gajski, "Transaction Level Modeling: an Overview" First International IEEE Conference on Hardware/Software Co-design and System Synthesis (CODESS '03), pp. 19-24, 2003.

[9] Chris Spares "System Verilog for verification" 2nd edition Springer 2008.

[10] Rudra Mukherjee, Sachin Kakkar "Towards an Object-Oriented Design Methodology using SystemVerilog" Proceedings of Design and Verification Conference & Exhibition (DVCon '09), pp. 234-239, 2009

[11] "IEEE Standard for System Verilog—IEEE std.1800-2009"

[12] "System Verilog Testbench Constructs" www.testbench.in