IJCTA, 9(10), 2016, pp. 4377-4392
© International Science Press

APB Based AHB Interconnect Testbench Architecture Using uvm_config_db

***Nitiyanka Dohare **Sonali Agrawal**

Abstract **:** This paper presents object-oriented testbench architecture for APB based AHB interconnect which is based on universal verification methodology (UVM) to build an efficient and structured verification environment. UVM-SV based AHB System that follows AHB Protocol, consists three AHB master, four AHB slave, an AHB interconnect (Design Under test) and one APB configure model which communicate with each other on the AHB bus and APB configure model decodes slave address range and generates signals for slave selection reduces interface complexity. System verilog interfaces in the testbench and DUT and virtual interfaces in the class based test environment cannot make use of type parameterize, results in very cumbersome code. To overcome this problem, this paper is using approach of pushing the virtual interface into the configuration database from top-level using uvm_config_db to reuse verification test environment and APB based AHB interconnect functional model is implemented which is configured as UVM component from testbech using uvm_config_db. A uvm_config_db method allows reuse of UVM components easily and configures uniformly. This paper is showing how APB based AHB Interconnect testbench is build using a uvm_config_db. The simulations are done using Mentor Graphics advanced Questasim 10.0b simulator with UVM base class library version 1.1d.

Keywords : Verification Intellectual Property (VIP),Universal Verification Methodology (UVM), AHB(Advanced High-Performance Bus) Interconnect, APB(Advanced Peripheral Bus), UVM verification components (UVC), Agent, Monitor, Driver, Sequencer, Sequences.

1. INTRODUCTION

Universal verification methodology (UVM) is a systematic way to build test environment, with rich set of standard rules and guidelines. UVM contains set of base class library that helps to build robust reliable and complex verification environment and reduces time to build testbench. On Feb 21, 2011 Accellera approved the 1.0 version of UVM with aim of make system verilog as an universal language [1,2,3]. Universal verification methodology supports Transaction Based Verification, coverage Driven Verification, Constrained Random Testing, and Assertion Based Verification that allow to keep enough small gap between verification capabilities and verification needs to get users the confidence level high sufficient to actually tape out their designs[4]. UVM based verification is used in many applications such as Smart-Sensor Systems[5], SPI[6], First Input-First Output (FIFO) buffer module and an I2C EEPROM module[7] , Bluetooth Low Energy Controller[8], and AHB-Lite Protocol[9].

In UVM based verification of above applications, parameterized interfaces are used to connect static modules: SV testbench and DUT, and dynamic class which is verification environment, consists of dynamic objects. Interfaces are initialized in the testbench to access signals between DUT and test environment [10,11,12,13]. The drawback of parameterized interfaces is that, virtual interfaces which are handles on the SV interfaces must be same type specialization as SV interfaces. Using BFM model this problem can be solved. But for complex BFM, it cannot be

* Dept of Electronics and Cmmunication Engineering Amrita School of Engineering, Bengaluru Amrita Vishwa Vidyapeetham Amrita University *India* Email:nitiyanka@gmail.com,

** Dept of Electronics and Cmmunication Engineering Amrita School of Engineering, Bengaluru Amrita Vishwa Vidyapeetham Amrita University India Email: a_sonali@blr.amrita.edu

reused and configured from the test verification environment, results in very cumbersome code [14]. All these problems can be solved by using UVM technology. The configuration database in the UVM helps in passing of objects and data to other components in the testbench to create reusable test environment [15,16]. This paper presents object-oriented UVM verification environment for APB based AHB interconnect to verify a communication between three AHB bus master and four AHB slave through an APB configuration model. and complex APB based AHB interconnect functional model is configured from testbech using uvm_config_db. Block diagram of UVM Testbench for APB based AHB Interconnect is shown in fig.1. This paper is using approach of pushing the virtual interface into the configuration database from top-level using uvm_config_db to reuse UVM component Agents from verification test environment. AHB and APB system verification interface are instantiated at top level module and AHB and APB virtual interface are handles on AHB and APB system verification interface respectively, added into database, thus it can be accessed to APB based AHB interconnect module and other component of UVM. AHB and APB interfaces stored into database are type parameterize.

2. UVM BASED TESTBENCH FOR APB BASED AHB INTERCONNECT

Basic building block of UVM based Testbench are components and transactions. A transaction is data item which is eventually or directly processed by DUT. UVM components are the static classes that make testbench architecture. These classes present live throughout the simulation. UVM component contain different features like Hierarchy Searching, Phasing, Configuring, Reporting, Factory and Transaction Recording. UVM Components are Agent, Driver, Monitor, Scoreboard and Sequencer which inbuilt in UVM libraries uvm_agent, uvm_driver, uvm_monitor, uvm_scoreboard and uvm_sequencer respectively. All these component are extended from uvm_component. UVM Transactions are sequence_item, sequence, virtual sequence. UVM Transactions are extended from uvm_object. UVM based Testbench for APB based AHB Interconnect is shown in Fig.1.

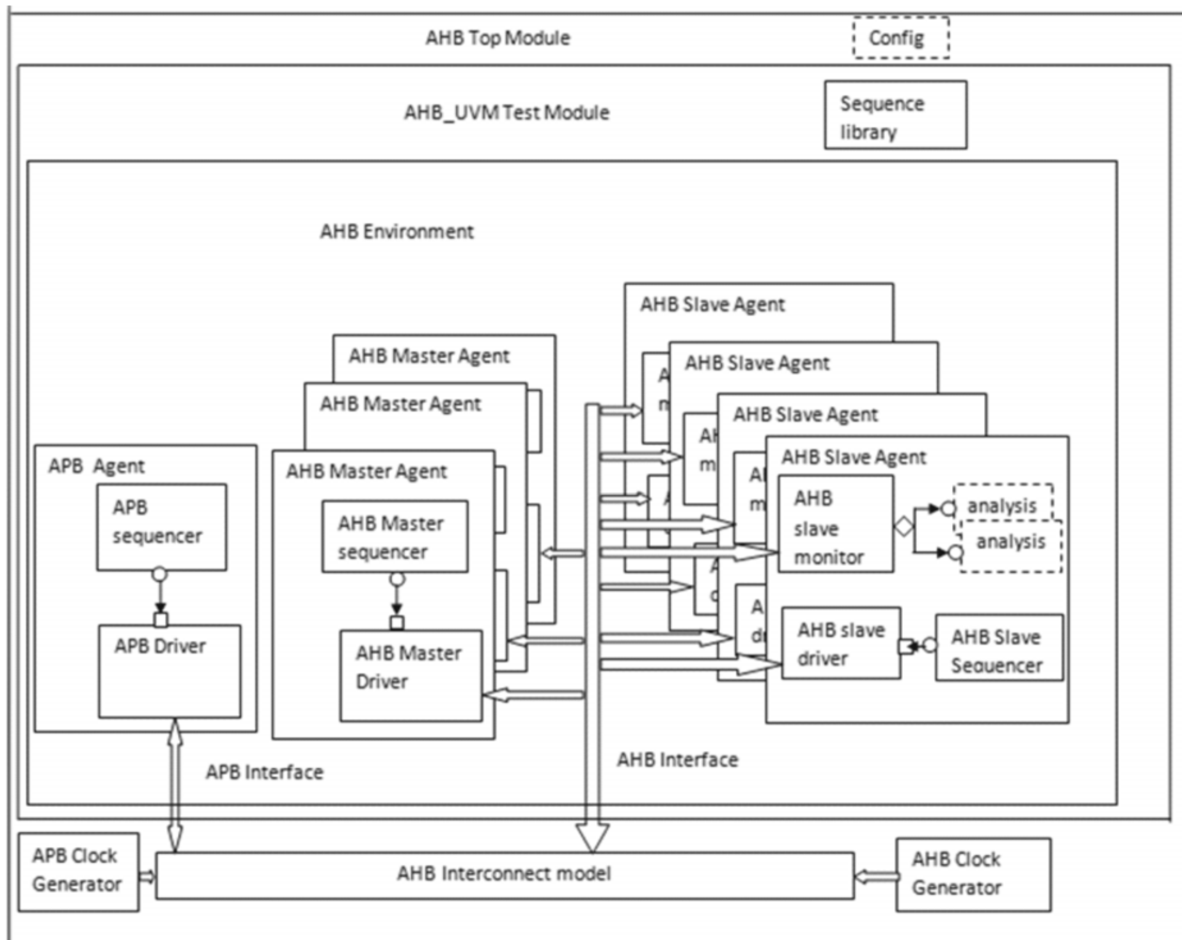

Fig. 1. UVM based Testbench Architecture for APB based AHB Interconnect.

A. AHB UVM Top Level Module

```
module top;
   `include "test_lib.sv"
   reg clk, rst, pclk, prst;
   ahb_intf mvif0(clk, rst);
   ahb_intf mvif1(clk, rst);
   ahb_intf mvif2(clk, rst);
   ahb_intf svif0(clk, rst);
   ahb_intf svif1(clk, rst);
   ahb_intf svif2(clk, rst);
   ahb_intf svif3(clk, rst);
   apb_intf cvif(pclk, prst);
   initial begin    clk = 0;
            forever #5 clk = ~clk;
.................................
 begin      uvm_config_db#(virtual ahb_intf)::set(uvm_root::get(), "*", "mvif0", mvif0);
            uvm_config_db#(virtual ahb_intf)::set(uvm_root::get(), "*", "mvif1", mvif1);
            uvm_config_db#(virtual ahb_intf)::set(uvm_root::get(), "*", "mvif2", mvif2);
            uvm_config_db#(virtual ahb_intf)::set(uvm_root::get(), "*", "svif0", svif0);
            uvm_config_db#(virtual ahb_intf)::set(uvm_root::get(), "*", "svif1", svif1);
            uvm_config_db#(virtual ahb_intf)::set(uvm_root::get(), "*", "svif2", svif2);
            uvm_config_db#(virtual ahb_intf)::set(uvm_root::get(), "*", "svif3", svif3);
            uvm_config_db#(virtual apb_intf)::set(uvm_root::get(), "*", "cvif", cvif);
              end
   initial begin
            run_test();
   end
endmodule
```

Fig. 2. AHB UVM Top Level Module

To allow the virtual interfaces to be widely accessible from anywhere within the testbench, instantiate interfaces in the top level and then need to be added virtual interfaces to the configuration database using the set() function of uvm_config_db as shown in fig2. To start search at the top of hierarchy, the first argument, uvm root::get() is used. Second argument of set() is wildcard, "*" which is used when the interface is available to all components. The third argument is the field name which is a label mvif0, mvif1, mvif2, svif0, svif1, svif2, svif3, & cvif, used for lookup. Finally, the value argument is the actual instance of the interface, mvif0, mvif1, mvif2, svif0, svif1, svif2, svif3, and cvif because configuration database entry is type parameterized to the SV interface. Clock(clk, pclk) and reset(rst, prst) signals for AHB system bus and APB peripheral bus are generated and run test UVM task is called here for running the test.

B. AHB Interconnect Module

AHB interconnect in this paper have AHB Masters, AHB Slaves and an APB Configure Model. An APB Configure Model decodes slave address range and generates one peripheral select signal to select corresponding slave. In APB Configure Model, APB access is program controlled which can be change later and single clock edge operation. APB Configure Model reduces interface complexity. Fig3 shows implementation of AHB interconnect functional model which can reuse from test verification environment. An ahb_ic

```
class ahb_ic extends uvm_component;
        int num_masters;
        int num_slaves;
        virtual ahb_intf mvif[];
        virtual ahb_intf svif[];
        virtual apb_intf cvif;
        bit [3:0] grant_master;
        bit [3:0] master_active;
        bit [3:0] slave_active;
        .......
        ........
        `uvm_component_utils_begin(ahb_ic)
                `uvm_field_int(num_masters, UVM_ALL_ON)
                `uvm_field_int(num_slaves, UVM_ALL_ON)
        `uvm_component_utils_end
        `NEW
        function void build_phase(uvm_phase phase);
                super.build_phase(phase);
        endfunction
        function void connect_phase(uvm_phase phase);
                super.connect_phase(phase);
                $display("num_masters=%d", num_masters);
                $display("num_slaves=%d", num_slaves);
                mvif=new[num_masters];
                svif=new[num_slaves];
                uvm_config_db#(virtual apb_intf)::get(this, "", "cvif", cvif);
                for (int i = 0; i < num_masters; i++) begin
                                $sformat(inst_name, "mvif%0d", i);
        uvm_config_db#(virtual ahb_intf)::get(this, "", inst_name, mvif[i]);
                 end
                for (int i = 0; i < num_slaves; i++) begin
                        $sformat(inst_name, "svif%0d", i);
        uvm_config_db#(virtual ahb_intf)::get(this, "", inst_name, svif[i]);
                 end
        endfunction
        task run_phase(uvm_phase phase);
        ..........
       ...........
        endtask
endclass
```

Fig. 3. AHB Interconnect Module.

class extends from predefine uvm_component base class then registers fields of the AHB interconnect with UVM factory using micro uvm_component_utils then call 'NEW' function set parent variable then super.build phase(phase) must be called in the build phase() function to automatically update the values of parameters of ahb_ic class. To retrieve the virtual interfaces, say in ahb_ic from top level module, a get() on the configuration database is used. The first argument of get() "this" is used as the context if a generic call to get() function is within a class. third argument is field name which are labeled as cvif, inst name and inst name for APB, AHB master and

slave interface respectively and the value stored in this entry would be assigned to the cvif, mvif[i] and svif[i] property respectively. Finally task is run in run phase by calling the predefined UVM task run_phase (uvm_phase phase). Functionality of APB based AHB interconnect is written in task run_phase(uvm_phase phase).

C. AHB UVM Test Module

AHB_base_test is derived from the uvm_test class which instantiates environment. AHB_base_test creates directed test scenarios and test sequences for their design. Write followed by read test scenarios and test sequences for AHB system can be created as shown in fig4. APB Sequencer entering the configuration phase, will create

```
class ahb_base_test extends uvm_test;
        `uvm_component_utils(ahb_base_test)
        ahb_env env;
        uvm_table_printer printer;
        `NEW
        function void build_phase(uvm_phase phase);
                super.build_phase(phase);
                phase.phase_done.set_drain_time(this, 500);
                env = ahb_env::type_id::create("env", this);
                printer = new();
                printer.knobs.depth = 3;
        endfunction
        function void end_of_elaboration();
                `uvm_info(get_type_name(), $psprintf("Printing the test topology :\n%s",
this.sprint(printer)), UVM_LOW)
        endfunction
        task run_phase(uvm_phase phase);
                super.run_phase(phase);
        endtask
endclass

class ahb_wr_rd_test extends ahb_base_test;
        `uvm_component_utils(ahb_wr_rd_test)
        function new(string name, uvm_component parent);
                super.new(name, parent);
        endfunction
        function void build_phase(uvm_phase phase);
                uvm_config_db#(int)::set(this, "env", "agent_f", 1);
                uvm_config_db#(uvm_object_wrapper)::set(this,
"env.pagent.sqr.configure_phase", "default_sequence", apb_config_seq::get_type());
                uvm_config_db#(uvm_object_wrapper)::set(this, "env.magent[0].sqr.main_phase",
"default_sequence", ahb_wr_rd_seq::get_type());
                uvm_config_db#(uvm_object_wrapper)::set(this, "env.magent[1].sqr.main_phase",
"default_sequence", ahb_wr_rd_seq::get_type());
                uvm_config_db#(uvm_object_wrapper)::set(this, "env.magent[3].sqr.main_phase",
"default_sequence", ahb_wr_rd_seq::get_type());
                uvm_config_db#(int)::set(this, "*", "num_masters", 3);
                uvm_config_db#(int)::set(this, "*", "num_slaves", 4);
                super.build_phase(phase);
        endfunction
```

Fig. 4. AHB UVM Test Module

an instance of the apb_config_seq sequence. AHB Master and AHB Slave Sequencer entering the main phase, will create an instance of the ahb_wr_rd_seq sequence. All sequence will randomize and start executing. uvm_config_db#(int)::set(this, "*", "num masters", 3) is used to get 3 AHB Master agent and uvm config db#(int)::set(this, "*", "num slaves", 4) is used to get 4 AHB slave agent. Questasim 10.0b simulator is printed test topology for write followed by read test case shown in fig5.

```
# UVM_INFO @ 0: reporter [RNTST] Running test ahb_wr_rd_test...
# UVM_INFO @ 0: uvm_test_top.env.pagent.drv [APB_DRV] DRIVER APB
# UVM_INFO @ 0: uvm_test_top.env.pagent.sqr [APB_SQR] Build_phase
# num_masters =         3
# num_slaves =         4
# UVM_INFO test_lib.sv(25) @ 0: uvm_test_top [ahb_wr_rd_test] Printing the test topology :
# ---------------------------------------------------
# Name            Type               Size  Value
# ---------------------------------------------------
# uvm_test_top      ahb_wr_rd_test        -    @463
#   env            ahb_env            -    @486
#     ic          ahb_ic            -    @571
#       num_masters  integral           32   'h3
#       num_slaves   integral           32   'h4
#     magent[0]     ahb_master_agent     -    @505
#       drv        ahb_master_driver   -    @583
#       sqr        ahb_master_squencer -    @609
#     magent[1]     ahb_master_agent     -    @516
#       drv        ahb_master_driver   -    @739
#       sqr        ahb_master_squencer -    @765
#     magent[2]     ahb_master_agent     -    @525
#       drv        ahb_master_driver   -    @895
#       sqr        ahb_master_squencer -    @921
#     pagent       apb_agent           -    @497
#       drv        apb_driver          -    @1051
#       sqr        apb_sequencer       -    @1077
#     sagent[0]     ahb_slave_agent      -    @534
#       drv        ahb_slave_driver    -    @1207
#       mon        ahb_slave_monitor   -    @1233
#     sagent[1]     ahb_slave_agent      -    @544
#       drv        ahb_slave_driver    -    @1246
#       mon        ahb_slave_monitor   -    @1272
#     sagent[2]     ahb_slave_agent      -    @553
#       drv        ahb_slave_driver    -    @1285
#       mon        ahb_slave_monitor   -    @1311
#     sagent[3]     ahb_slave_agent      -    @562
#       drv        ahb_slave_driver    -    @1324
#       mon        ahb_slave_monitor   -    @1350
#     agent_f       integral           1    'h1
#     num_masters   integral           32   'h3
#     num_slaves    integral           32   'h4
# ---------------------------------------------------
```

Fig. 5. Printed write followed by read test topology.

D. AHB UVM Test Environment

AHB UVM test environment as shown in fig 6 encapsulates APB Agent, AHB Master Agent and AHB Slave Agent. The ahb_env_class is drived from the uvm_env class, registered the field of uvm_env base class, then call uvm object NEW to set parent class uvm_env variables. The test environment merely has a handle on the Agent, as can be seen by the identical numbers in the Value column in fig5.

E. AHB UVM Agents

For APB based AHB Interconnect Testbench Architecture, this paper implements three AHB master agent , four AHB slave agent and one APB agent. The basic function of Agent is just an encapsulation of monitor, sequencer and driver as shown in fig.5. UVM Agent can be configured as 'ACTIVE' where all the components will run, or as 'PASSIVE' where only monitor component is running through configuration attribute 'is_active'. By default, UVM Agent is 'ACTIVE'.

```
class ahb_env extends uvm_env;
..............................................
..............................................
        apb_agent pagent;
        ahb_master_agent magent[];
        ahb_slave_agent sagent[];
        ahb_ic ic;
        `NEW
        function void build_phase(uvm_phase phase);
                super.build_phase(phase);
                pagent = apb_agent::type_id::create("pagent", this);
                magent = new[num_masters];
                for (int i = 0; i < num_masters; i++) begin
                        $sformat(inst_name, "magent[%0d]", i);
                        magent[i] = ahb_master_agent ::type_id::create(inst_name, this);
                        uvm_config_db#(int)::set(this, {inst_name,"*"}, "master_no", i);
                end
                sagent = new[num_slaves];
                for (int i = 0; i < num_slaves; i++) begin
                        $sformat(inst_name, "sagent[%0d]", i);
                        sagent[i] = ahb_slave_agent::type_id::create(inst_name, this);
                        uvm_config_db#(int)::set(this, {inst_name,"*"}, "slave_no", i);
                end
                ic = ahb_ic::type_id::create("ic", this);
        endfunction
endclass
```

Fig. 6. AHB UVM Test Environment.

1. **AHB_Master_Agent :** AHB_Master_Agent as shown in fig 7 is UVM verification components (UVC) which is responsible for generating transactions. AHB_Master_Agent is extended from uvm_agent base class. Same Agent code is instantiated for each AHB master interface.

```
class ahb_master_agent extends uvm_agent;
	`uvm_component_utils(ahb_master_agent)
	ahb_master_driver drv;
	ahb_master_squencer sqr;
	function new(string name, uvm_component parent);
		super.new(name, parent);
	endfunction

	function void build_phase(uvm_phase phase);
		super.build_phase(phase);
		drv = ahb_master_driver::type_id::create("drv", this);
		sqr = ahb_master_squencer::type_id::create("sqr", this);
	endfunction

	function void connect_phase(uvm_phase phase);
		super.connect_phase(phase);
		drv.seq_item_port.connect(sqr.seq_item_export);
	endfunction
endclass
```

Fig. 7. AHB Master Agent

2. **AHB_Slave_Agent :** AHB_Slave_Agent as shown in fig 8 is UVM verification components (UVC) which is responsible for responding to master requests.

```
class ahb_slave_agent extends uvm_agent;
	`uvm_component_utils(ahb_slave_agent)
	ahb_slave_driver drv;
	ahb_slave_monitor mon;
	function new(string name, uvm_component parent);
		super.new(name, parent);
	endfunction
	function void build_phase(uvm_phase phase);
		super.build_phase(phase);
		drv = ahb_slave_driver::type_id::create("drv", this);
		mon = ahb_slave_monitor::type_id::create("mon", this);
	endfunction
endclass
```

Fig. 8. AHB Slave Agent

AHB_Slave_Agent is also extended from uvm agent base class. Same Agent code is instantiated for each AHB slave interface.

3. **APB_Agent :** APB_Agent as shown in fig 9 is UVM verification components (UVC) which is responsible for configuring slave address space range. APB Agent is also

```
class apb_agent extends uvm_agent;
        `uvm_component_utils(apb_agent)
        apb_driver drv;
        apb_sequencer sqr;
        function new(string name, uvm_component parent);
                super.new(name, parent);
        endfunction
        function void build_phase(uvm_phase phase);
                super.build_phase(phase);
                drv = apb_driver::type_id::create("drv", this);
                sqr= apb_sequencer::type_id::create("sqr", this);
        endfunction
        function void connect_phase(uvm_phase phase);
                super.connect_phase(phase);
                drv.seq_item_port.connect(sqr.seq_item_export);
        endfunction
endclass
```

Fig. 9. APB Agent.

extended from uvm_agent base class for APB interface.

F. AHB UVM Drivers

The uvm_driver has inbuilt sequence_item handle instance (req) and TLM ports to communicate with the sequencer. It drives sequences into the DUT through the interface after fetching it from the Sequencer through the TLM port.

1. **AHB_Master_Driver :** AHB master driver is extended from uvm_driver base class. Virtual interface on AHB interface ahb_intf in AHB_Master_Driver is retrieved by doing a get() on the configuration database as shown in fig 10. After getting virtual interface handle vif on AHB interface ahb_intf, AHB_Master_Driver classes connect UVM AHB Master Sequencer using TLM port to get sequence item of AHB Master and drive one complete transaction: Arbitration phase + Multiple address phase + Multiple data phases of AHB Master to DUT interface in run phase.

```
class ahb_master_driver extends uvm_driver#(ahb_tx);
.............................
.............................
        function void connect_phase(uvm_phase phase);
                super.connect_phase(phase);
                $sformat(name, "ahb_m%0dvif", master_no);
              uvm_config_db#(virtual ahb_intf)::get(this, "", inst_name, svif);
                if (!uvm_config_db#(virtual ahb_intf)::get(this, "", name, vif)) begin
                        `uvm_error("AHB_MASTER_DRV", "AHB IF not registered");
                end
        endfunction

        task run_phase(uvm_phase phase);
        while(1) begin
                seq_item_port.get_next_item(req);
                req.print();
                drive_transfer(req);
                seq_item_port.item_done();
        end
        endtask
        task drive_transfer(ahb_tx  tx);
        ...............
                        task arbitration_phase(ahb_tx tx);
                          ...............
                        endtask
                        task address_phase(ahb_tx tx );
                          ...............
                        task data_phase(ahb_tx tx);
                        endtask
                            .................
                        endtask
        endtask
endclass
```

Fig. 10. AHB Master driver

2. **AHB_Slave_Driver :** AHB_Slave_Driver is also extended from uvm_driver base class. Virtual interface on AHB interface ahb_intf in AHB slave driver is retrieved by doing a get() on the configuration database. After getting virtual interface handle vif on AHB interface ahb_intf, AHB_Slave_Driver classes drive response back to master requests to DUT interface in run phase. Fig11 shows connect phase of AHB_Slave_Driver in which virtual interfaces are retrieved.

```
class ahb_slave_driver extends uvm_driver#(ahb_tx);
...........................
...........................
        function void connect_phase(uvm_phase phase);
                super.connect_phase(phase);
                $sformat(inst_name, "svif%0d", slave_no);
        uvm_config_db#(virtual ahb_intf)::get(this, "", inst_name, svif);
                if (!uvm_config_db#(virtual ahb_intf)::get(this, "", name, vif)) begin
                        `uvm_error("AHB_DRV", "AHB IF not registered");
                end
        endfunction

        task run_phase(uvm_phase phase);
        ...............
        ..................
        endtask
endclass
```

Fig. 11. AHB Slave Driver.

3. **APB_Driver :** APB_Driver class as shown in fig 12 is also extended from uvm_driver base class. Virtual interface on APB interface apb_intf in APB Driver class is retrieved by doing a get() on the configuration database. After getting virtual interface handle vif on APB interface apb_intf, APB_Driver class connects UVM APB Sequencer using TLM port to get APB sequence items and drives it to DUT interface in run phase.

```
class apb_driver extends uvm_driver#(apb_tx);
...........................
...........................
        function void connect_phase(uvm_phase phase);
                super.connect_phase(phase);
                uvm_report_info("APB_DRV", "DRIVER APB", UVM_NONE);
        uvm_config_db#(virtual apb_intf)::get(this, "", "cvif", vif);
                if (!uvm_config_db#(virtual apb_intf)::get(this, "", name, vif)) begin
                        `uvm_error("APB_DRV", "APB IF not registered");
                end
        endfunction

        task run_phase(uvm_phase phase);
        ...............
        ..................
        endtask
endclass
```

Fig. 12. APB Driver

G. AHB_ Slave_Monitor

AHB_Slave_Monitor as shown in fig 13 is extended from uvm_monitor base class. AHB_Slave_Monitor is a passive component, it does not drive any signal into the DUT. Virtual interface on AHB interface ahb_intf in AHB slave monitor class is retrieved by doing a get() on the configuration database. If flag master slave f is "1",virtual interface on AHB interface ahb_intf retrieves for AHB master otherwise for AHB slave. After getting virtual interface handle vif on AHB interface ahb_intf, AHB slave monitor builds observed transaction and pass them onto scoreboard via an analysis port.

```
class ahb_slave_monitor extends uvm_monitor;
        int master_no;
        int slave_no;
        int master_slave_f;
        string inst_name;
        `uvm_component_utils(ahb_slave_monitor)
        virtual ahb_intf vif;
        uvm_analysis_port#(ahb_tx)ap;
        `NEW
        `BUILD_PHASE

        function void connect_phase(uvm_phase phase);
                super.connect_phase(phase);
                if(master_slave_f==1) begin
                        $sformat(inst_name, "mvif%0d", master_no);
                end
                if(master_slave_f==0) begin
                        $sformat(inst_name, "svif%0d", slave_no);          end
                uvm_config_db#(virtual ahb_intf)::get(this, "", inst_name, vif);
                if(!uvm_config_db#(virtual ahb_intf)::get(this, "", inst_name, vif)) begin
                        `uvm_error("AHB_MON", "AHB IF not registered");
                end
        endfunction

        task run_phase(uvm_phase phase);
        ................
        ....................
        endtask
endclass
```

Fig. 13. AHB Slave Monitor

H. AHB Master UVM Sequencer, AHB Slave UVM Sequencer, APB UVM Sequencer

When multiple sequences are running in parallel, Sequencer is responsible for arbitrating between parallel sequences. When requests for new transaction are initiated by driver then upon such requests, a sequence from a list of available sequences is selected by the sequencer. After that sequencer produces and delivers the next transaction item to execute. The uvm_sequencer has inbuilt TLM interface 'sequence_item_export' and 'rst_export' to communicate with uvm_driver.

```
class ahb_master_squencer extends uvm_sequencer#(ahb_tx);
        `uvm_component_utils(ahb_master_squencer)
        function new(string name, uvm_component parent);
                super.new(name, parent);
        endfunction
        function void build_phase(uvm_phase phase);
                super.build_phase(phase);
        endfunction
endclass

class apb_sequencer extends uvm_sequencer#(apb_tx);
        `uvm_component_utils(apb_sequencer)
        function new(string name, uvm_component parent);
                super.new(name, parent);
        endfunction
        function void build_phase(uvm_phase phase);
                super.build_phase(phase);
                uvm_report_info("APB_SQR", "Build_phase", UVM_NONE);
        endfunction
endclass
```

Fig. 14. AHB Master UVM Sequencer and APB UVM Sequencer.

I. AHB UVM Sequence

Stimulus transactions of AHB for ahb_wr_rd seq and of APB Configuration model for apb_config_seq are created inside a UVM sequence extended from uvm_sequence base class.

J. AHB UVM Sequence item

AHB and APB sequence item class are driven from uvm_sequence_item to add AHB and APB field and constrains respectively. In APB sequence item class, constrain for ad- dress space range of slave is added. In AHB sequence item class, constrains are added for address space at different length of transfer, for transaction size at different burst and for order of solving length before burst.

K. AHB UVM Sequence Library

A Sequence Library class is a repository of sequences of AHB Master, AHB Slave and APB configure data type. To enhance reusability, Sequence Library class encapsulates in a package and base sequence, AHB sequences and APB sequences are added to package .

2. RESULTS

The simulation has done using Mentor Graphics advanced Questasim 10.0b simulator with UVM base class library version 1.1d. Questasim 10.0b simulator tool is developed by Mentor Graphics for advanced functional verification like Transaction Based Verification, coverage Driven Verification, Constrained Random Testing, and Assertion Based Verification.

The simulation result for single burst Write followed by Read test case is shown in fig15. There are 8 interconnect registers. Whenever there is apb write transaction, the ahb interconnect model should be updated to program the register fields. In fig15 when apb signals pwrite and penable is high, First register is programmed with address 32’h100 and data 32’h10000000. Second register is programmed with address 32’h104 and data

32’h30000000. Third register is programmed with address 32’h108 and data 32’h40000000. Fourth register is programmed with address 32’h10c and data 32’h60000000. Fifth register is programmed with address 32’h110 and data 32’h70000000. Sixth register is programmed with address 32’h114 and data 32’h90000000. Seventh register is programmed with address 32’h118 and data 32’ha0000000. Last register is programmed with address 32’h11c and data 32’hc0000000. The starting address of slave s0 is represented by data value 32’h10000000 of register having address 32’h100. The ending address slave s0 is represented by data 32’h30000000 of register having address 32’h104. The starting address of slave s1 is represented by data value 32’h40000000 of register having address 32’h108. The ending address slave s1 is represented by data 32’h60000000 of register having address 32’h10c. The starting address of slave s2 is represented by data value 32’h70000000 of register having address 32’h110. The ending address slave s2 is represented by data 32’h90000000 of register having address 32’h114. The starting address of slave s3 is represented by data value 32’ha0000000 of register having address 32’h118. The ending address slave s3 is represented by data 32’hc0000000 of register having address 32’h11c. The minimum address space that is allocated to each slave is 1kB. At 85ns clock is changing low to high, AHB master mvif0 gets grant. At 95ns clock is changing low to high, AHB master mvif0 sends address control signal.

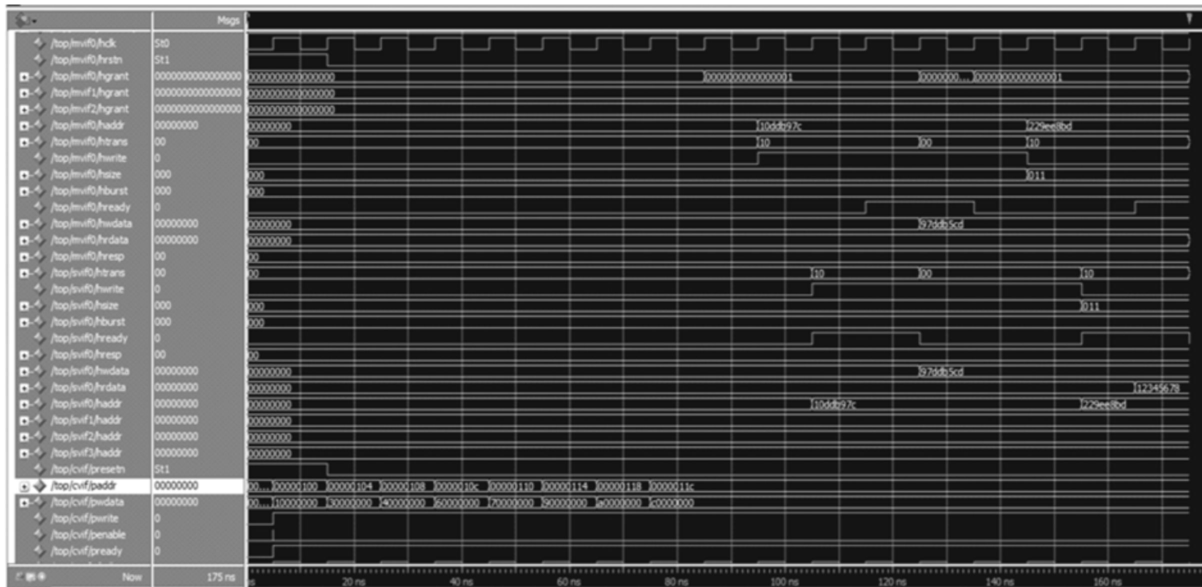

Fig. 15. Single Burst for Write followed by Read Test case

The value of address is 32’h10ddb97c. The value of control signal hburst = 000, indicates single burst transfer and hwrite = 1 indicates write transfer. Control signal htrans value is 10 for the transfer indicating non sequential transfer. Control signal Hsize = 000 indicating halfword transfer. At 105ns APB configuration module decodes slave address range based on AHB master mvif0 sending address 32’h10ddb97c and generates signal to select slave s0. The AHB Slave s0 receive address and control signal and send back hresp=00, indicating ok response and ready signal. At 115ns AHB master mvif0 send ready signal and at 125ns it send data 32’h97ddb5cd and AHB slave s0 receive that data. Now AHB master mvif0 wants to read the data from slave. Then again AHB master mvif0 which highest priority gets grant at 135ns. AHB master mvif0 sends address and control signal at 145ns. The value of address is 32’h229ee8bd. The value of control signal hburst = 000, hwrite = 0,Hsize = 011 and htrans = 00.Based on address signal of AHB master mvif0, slave s0 is selected at 155ns and it receives address and control signal of AHB master mvif0 and send back ready signal. At 165ns slave s0 sends data 32’h12345678. Fig2 is showing printed test topology for write followed by read test case. UVM report summary is shown in fig 4 which indicates no error in design, execution time is 175ns and number of iteration is 100. As compare to verilog based testbench (which has Execution time in milli seconds) and system verilog based testbench(which has Execution time in micro seconds), universal verification methodology-system verilog (UVM-SV) based testbench has less Execution time(in nano seconds) and number of iteration because of uvm config db method.

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :    5
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [APB_DRV]     1
# [APB_SQR]     1
# [NO_DPI_TSTNAME]     1
# [RNTST]     1
# [ahb_wr_rd_test]     1
# ** Note: $finish    : C:/questasim_10.0b/verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 175 ns  Iteration: 100  Instance: /top
```

Fig. 16. UVM Report Summary for Write followed by Read.

4. CONCLUSION

uvm_config_db is used when the UVM component is avail- able for other component because it does not create new copies of that UVM component. Thus in this paper, uvm_config_db makes APB and AHB virtual interfaces available to multiple AHB master agents , AHB slave agents, an APB agent and AHB interconnect functional model by pushing the virtual interfaces into configuration database from top level. Pushing virtual interfaces into database and retrieving virtual interfaces from database is done by set() and get() function of uvm config db respectively. Reusing and configuring same AHB master and slave agent from Test Environment multiple time instead code multiple time reduces the time to build testbench of APB based AHB interconnect. Indirectly, time to market will also be reduced. In this paper, four AHB master, three AHB slave and one APB agent is used in test environment. Number of agent can be increase/decrease by changing last argument in set () and / or get() function of uvm_config_db as per requirement and uvm_config_db::set connect two interface (AHB and APB interface) to seven agent and AHB interconnect model with very less afford.

5. REFERENCES

1. Accellera, UVM 1.1 User Guide, 2011.
2. Y. N. Yun, J. B. Kim, N. D. Kim, B. Min, “Beyond UVM for Practical SOC Verification”, *IEEE International Conference on SoC Design (ISOCC)*, pp. 158-162, Nov. 2011.
3. Jonathan Bromley , “If SystemVerilog Is So Good, Why Do We Need the UVM? “, *IEEE Form on Specification & Design Languages (FDL)*, pp. 1-7, Sept. 2013.
4. R. Drechsler, C. Chevallaz, F. Fummi, A. J. Hu, R. Morad, F. Schirrmeis- ter,and A. Goryachev, “Future SoC Verification Methodology: UVM Evo- lution or Revolution?”, *IEEE Conference on Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1-5, March 2014.
5. F. Neumann,M. Sathyamurthy, L. Kotynia, E. Hennig, and R. Sommer, “UVM-based Verification of Smart-Sensor Systems”, *IEEE International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, pp. 21-24, Sept. 2012.
6. B. Ustaolu, Elektrik ve Elektron, “Creating test environment with UVM for SPI”, *IEEE 23nd Signal Processing and Communications Applications Conference (SIU)*, pp. 2373-2376, May 2015.
7. Juan Francesconi, J. Agustin Rodriguez, and Pedro M. Julian, “UVM Based Testbench Architecture for Unit Verification”, *IEEE Argentine Con- ference on Micro-Nanoelectronics, Technology and Applications (EAMTA)*, pp. 89-94, July 2014.

8. Maciej Moskala, Patryk Kloczko, Marek Cieplucha, and Witold Pleskacz, “UVM-based Verification of Bluetooth Low Energy Controller”, *IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, pp. 123-124, April 2015.
9. Ravi. Y and C Yadu Prasad, “System Verilog Methodology for Verification of AHB-LITE Protocol”, *International Journal For Technological Research In Engineering*, Vol.2, No.10, pp. 2216-2218, June-2015.
10. R. Madan, N. Kumar, and S. Deb, “ Pragmatic Approaches to Implement Self-Checking Mechanism in UVM Based TestBench”, *IEEE International Conference on Advances in Computer Engineering and Applications (ICACEA)*, pp. 632-636, March 2015.
11. Kab Joo Lee, Si Hyun Kim, and Hyo Seon Hwang, “SOC Bus Trans- action Verification Using AMBA Protocol Checker”, *Journal of Semi- conductor Technology and Science*, Vol.2,No.2, pp. 132-140, June 2002.
12. Akshay Mann,and Ashwani Kumar, “Assertion Based Verification of AMBA-AHB Using Synopsys VCS”, *International Journal of Scientific& Engineering Research*,Vol.4,No.11, pp.58-64,Nov 2013.
13. N.Karthik,M.Gurunadha Babu, and Muni Praveena Rela, “Assertion Based Verification of AMBA-AHB Using System Verilog”, *International Journal & Magazine of Engineering, Technology, Management and Research*, Vol.2,No.7, pp. 605-611, July 2015.
14. Gunther Clasen and Ensilica, “Flexible UVM Components: Configuring Bus Functional Models”, *Verification Horizon*, Vol. 9, No. 2, pp.58-63, June 2013.
15. Vanessa R. Cooper and Paul Marriott, “Demystifying the UVM Config- uration Database”, *configdb dvcon2014 poster*, pp.1-7, 2014.
16. Hannes Nurminen and Satya Durga Ravi, “Hierarchal Testbench Configuration Using uvm config db”, *White Paper on Synopsys Accelerating Innovative* , pp.1-12, June 2014.