IEEE International Conference on Advances in Engineering & Technology Research (ICAETR - 2014),
August 01-02, 2014, Dr. Virendra Swarup Group of Institutions, Unnao, India

C-based Predictor for Scoreboard in Universal Verification Methodology

Srikanth Konale, N.Bheema Rao
Department of Electronics and Communication Engineering
National Institute of Technology Warangal
Warangal, India
konalesrikanth@gmail.com, nbr.rao@gmail.com

***Abstract*—Universal Verification Methodology (UVM) is a standardized hybrid methodology for verifying complex integrated circuit designs in the semiconductor industry. Predictor is a component in UVM based test bench that represents a golden model of the design under test (DUT), which generates an expected response against which the actual response of the DUT is compared in scoreboard. Predictors are mostly written in C or C++ for modelling the correct functionality of the DUT. It is provided in the form of compiled object code to the testbench and acts as a verification component. UVM uses SystemVerilog Direct Programming Interface (DPI) for communicating components written in C with other components of the test bench. This paper describes implementation of a UVM testbench consisting of a C based predictor, in the form of a complied object code for verification of a fused floating-point add-subtract design unit.**

Keywords- Predictor; UVM; C; System Verilog; Testbench; Functional Verification

I. INTRODUCTION

Universal Verification Methodology (UVM) represents the latest member of a family of methodologies and their associated base class libraries for using SystemVerilog for constrained random verification. UVM test benches are complete verification environments composed of reusable verification components. Scoreboard is a verification component in test bench that determines whether or not the design under test (DUT) is functioning correctly. The scoreboard predicts the expected response and then compares the expected response with the actual response form DUT. The prediction task is implemented by the predictor.

Predictors are written in C, C++ or SystemC which is a foreign language to the system Verilog based testbench. Using DPI the foreign language code functionality is included in the system Verilog. It is made available to the test bench in the form of a compiled object code .The created object code needs to be loaded to integrate the foreign language code into a SystemVerilog testbench. The rest of the paper describes the components of UVM based testbench and inclusion of foreign language code as a verification component.

II. DESIGN UNDER TEST

The design under test (DUT) which is to be verified is fused

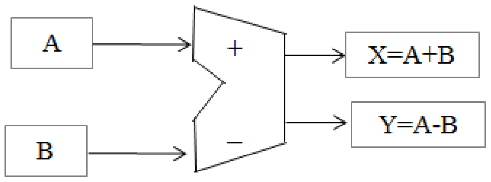


Figure 1. Fused Floating-Point Add-Subtract unit

floating-point add-subtract unit [7]. It produces the sum and difference of the two input floating point single precision numbers simultaneously represented in IEEE-754 format.

III. DUT-TESTBENCH COMMUNICATION

A UVM testbench has a top level SystemVerilog module which is a container for both the test bench and the DUT with its associated connection and support logic (such as clock generation). Virtual interfaces are used for communication because a class based test bench cannot reference Verilog/Vhdl ports and hence cannot directly connect to a Verilog or VHDL DUT. Instead a SystemVerilog interface instance is connected to the ports of the DUT and the test bench communicates with the DUT through the interface instance.

IV. UVM TESTBENCH ARCHITECTURE

A UVM test bench is composed of reusable verification environments called verification components. There are two main collective component types used to enable reuse- the agent and the environment. Fig. 3 shows a typical UVM testbench architecture consisting of test, environment, agent and its connection with the DUT.

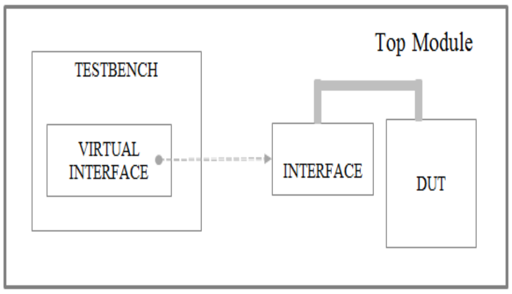


Figure 2. DUT- Testbench Communication

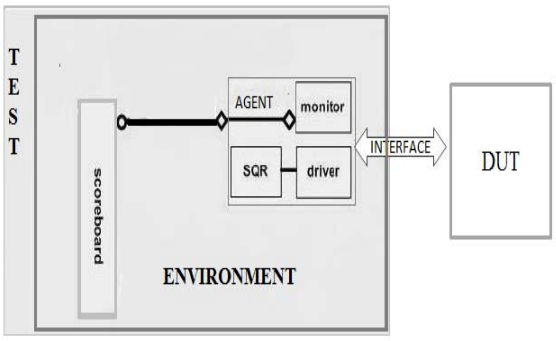


Figure 3. Testbench Architecture

A. Agent

The agent class is a top level container class for a driver, a sequencer and a monitor, and monitors. The components of an agent are described below.

1) *Sequence item:* Data items also called as sequence item are the basic data objects which are passed to device under test (DUT). In a typical test, many data items are generated and sent to the DUT. By intelligently randomizing data item fields many large number of meaningful tests can be created. Sequence item (here simple_item) which is used for passing the input to DUT is extended from uvm_sequence item.

2) *Driver:* The driver is responsible for sending the data inside the sequence_items into pin level transactions to the DUT. The driver drives the inputs to the DUT through the virtual interface.

3) *Sequence:* Sequences are responsible for the stimulus generation flow and send sequence_items to a driver via a sequencer component. The sequencer is an intermediate component which implements communication channels to facilitate interactions between sequences and driver.

4) *Sequencer:* A sequencer is a component that runs sequences. The role of the sequencer is to route sequence_items from a sequence where they are generated to/from a driver.

5) *Monitor:* A monitor is a passive entity that samples DUT signals but does not drive them. It observes pin level activity and converts its observations into sequence_items which are sent to components such as scoreboards which use them to analyze what are happening in the test bench.

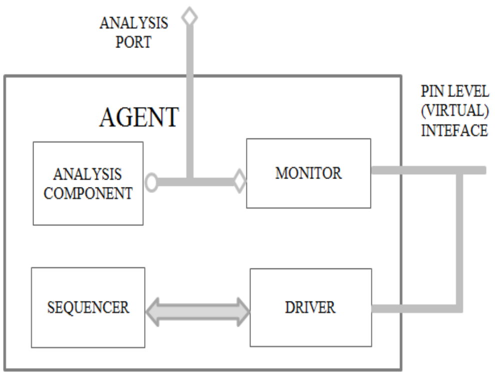


Figure 4. Agent Architecture

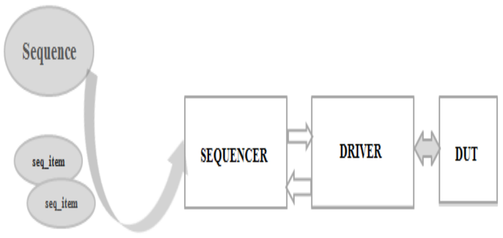


Figure 5. UVM Sequence Based Stimulus Generation Architecture

B. Environment

The environment (env) is the top-level component of the verification component. Environment can contain one or more agents. In a test bench consisting of multiple agents the environment (env) is used to collect together the agents needed to communicate with the DUT's interfaces together in one place. In addition to the agents, the environment will also contain components like scoreboard and monitors.

1) *Scoreboard:* The Scoreboard's job is to determine whether or not the DUT is functioning properly. The best scoreboard architecture is to separate the prediction task from the evaluation task (comparator). The prediction task is implemented by predictor whose response is compared with the actual response from DUT to verify the correct functionality of the design.

V. C-BASED PREDICTOR

A Predictor is a verification component that represents a golden model of the DUT. It takes the same input stimulus that is sent to the DUT and produces expected response data that is by definition correct. Predictors can be written in C, C++ or SystemC. The predictor model for verification of the fused floating point add-subtract unit design is written in C. SystemVerilog provides the Direct Programming Interface (DPI), an easier way to interface with C, C++, or any other foreign language. A pair of matching type definitions is required to pass a value through DPI, the SystemVerilog definition and the C definition. Table I shows the SystemVerilog types which are directly compatible with C types. First the C file implementing the float_add and float_sub functions is compiled using *gcc -c -fpic filename.c* command. It produces create a *filename.o* object file. The test bench is developed on opensuse OS having Questa simulator, which supports UVM from mentor graphics installed on it.

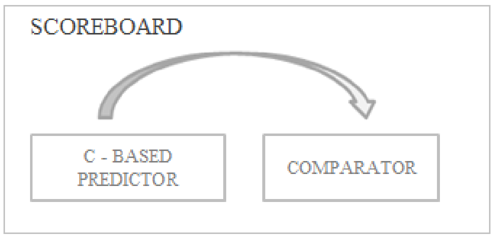


Figure 6. Scoreboard Architecture

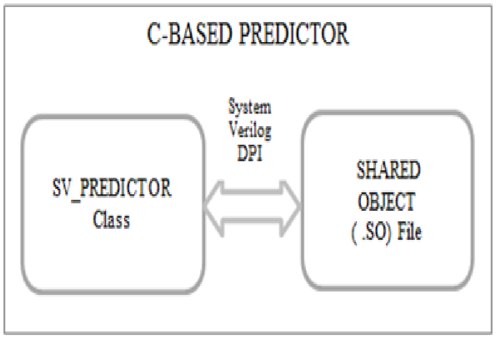


Figure 7. C-based Predictor Architecture

The object file is later converted to shared object file libc.so using *gcc -shared -o libc.so filename.o* command. This object file is included into a system Verilog testbench using –sv_lib switch. The command *–sv_lib libc* adds shared object file to the UVM library so it is accessible to testbench. The *sv_predictor* is *user defined* which is extended from uvm_subscriber class. It sends the two inputs A and B to the C code and receives the sum and difference using DPI. Sv_predictor class together with the shared object file is referred to as *C-based Predictor.*

Table I. System Verilog and C types

System Verilog Type	C Type
Byte	Char
Int	Int
Longint	Long long
Shortint	Short int
Real	Double
String	Const char*
Chandle	Void*

It is performing the prediction task of the scoreboard. C file must include the header file which defines all basic types, and all interface functions. The size of the Int data type is 32-bit in C so we are passing the two inputs a, b which is of 32 bits as Int. By applying masking on the 32bit data in C the sign, exponent and fraction bit is extracted and addition, subtraction are implemented and result are returned using return statement in C which are assigned to sum_result and diff_result respectively. These results are sent to the *comparator* using analysis ports, and are compared against the responses received from the DUT in the scoreboard.

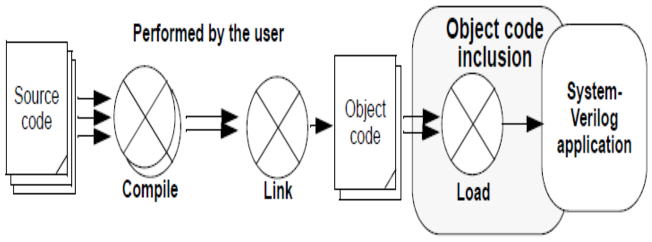


Figure 8. Inclusion of foreign language code

```
class my_env extends uvm_env;
`uvm_component_utils(my_env);
my_agent      m_agent;
sv_predictor  sv_predictor0;
scoreboard    scoreboard0;
function new (string name="my_env", uvm_component parent);
super.new(name,parent);
endfunction : new
function void build_phase(uvm_phase phase);
uvm_report_info(get_type_name(), $psprintf("Environment in BUILD_PHASE"));
$display ("Entering build in my_environment") ;
m_agent      = my_agent::type_id::create("m_agent",this);
sv_predictor0 = sv_predictor::type_id::create("sv_predictor0",this);
scoreboard0  = scoreboard::type_id::create("scoreboard0",this);
endfunction : build_phase
function void connect_phase(uvm_phase phase);
uvm_report_info(get_type_name(), $psprintf("Environment in Connect_PHASE"));
$display ("Entering connect in my_environment") ;
m_agent.command_ap.connect(sv_predictor0.analysis_export);
sv_predictor0.predictor_ap.connect(scoreboard0.expected_port);
m_agent.result_ap.connect(scoreboard0.actual_port);
endfunction : connect_phase
endclass : my_env
```

```
import "DPI-C" function int float_add (int a, int b);
import "DPI-C" function int float_sub (int a, int b);
class sv_predictor extends uvm_subscriber #(simple_item);
`uvm_component_utils(sv_predictor);
simple_item cmd_trn;
uvm_analysis_port #(simple_item) predictor_ap;
  function new (string name="sv_predictor", uvm_component parent);
     super.new(name, parent);
  endfunction : new
  function void build_phase(uvm_phase phase);
uvm_report_info(get_type_name(), $psprintf(" predictor in build_PHASE"));
$display ("Entering build in predictor") ;
   predictor_ap = new ("predcitor_ap", this);
   cmd_trn = new();
endfunction : build_phase
function void write(simple_item t);
$cast(cmd_trn,t.clone());
    t.sum_result = float_add(t.A,t.B);
    t.diff_result= float_sub(t.A,t.B);
 uvm_report_info(get_type_name(), $psprintf(" predictor calculating o/p"));
 $display ("Entering calculation o/p in predictor") ;
    `uvm_info("PREDICTOR",$sformatf("MONITOR: A: %2h  B: %2h  expected_sum:
%2h expected_diff: %2h",t.A, t.B, t.sum_result,t.diff_result), UVM_HIGH);
predictor_ap.write(t);
endfunction
```

VI. THE STANDARD UVM PHASES

In order to have a consistent test bench execution flow, the UVM uses phases to order the major steps that take place during simulation. There are three groups of phases, which are executed in the following order:

a) Build phases - where the test bench is configured and constructed
b) Run-time phases - where time is consumed in running the test case on the test bench
c) Clean up phases – where results of the test case are collected and reported

VII. TESTBENCH BUILD

The UVM test bench is activated when the run_test() method is called in an initial block in the top level test module. This method is an UVM static method, and it takes a string argument that defines the test to be run and constructs it via the factory. Then the UVM starts the build phase by calling the test class build method. During the build_phase of test, the test bench component configuration objects are prepared and assignments to the test bench module interfaces are made to the virtual interface handles in the configuration objects.

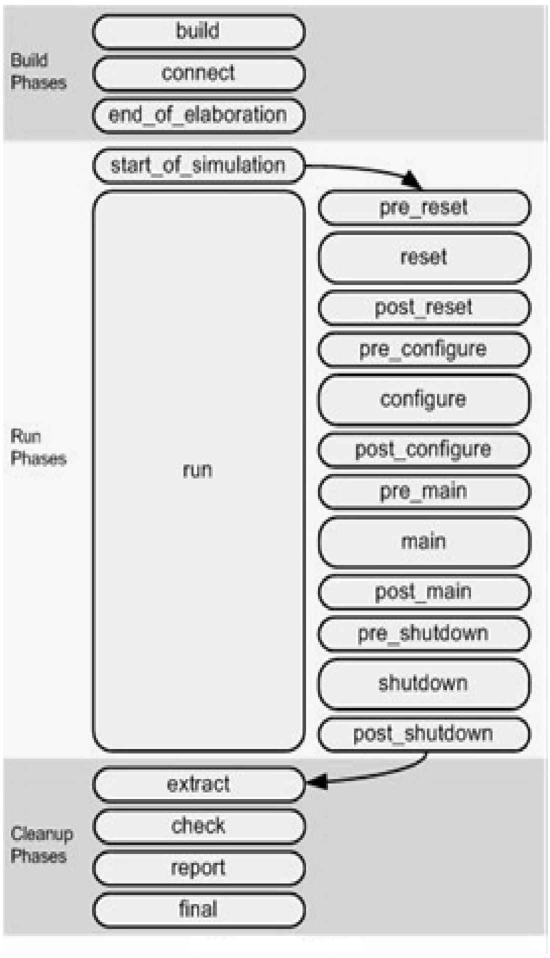


Figure 9. UVM Phases

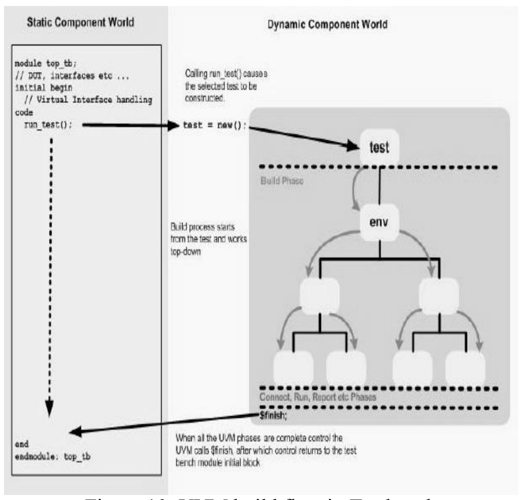
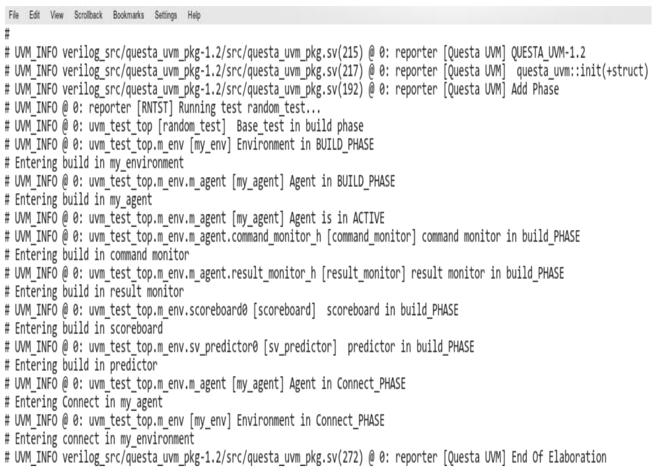

Static Component World
Dynamic Component World
module top_tb;
// DUT, interfaces etc ...
initial begin
// Virtual Interface handling code
run_test();
end
endmodule: top_tb
Calling run_test() causes the selected test to be constructed.
test = new();
test
Build Phase
env
Build process starts from the test and works top-down
Connect, Run, Report etc Phases
$finish;
When all the UVM phases are complete control the UVM calls $finish, after which control returns to the test bench module initial block

Figure 10. UVM build flow in Testbench

The build phase works *top-down* and so the process is repeated for each successive level of the test bench hierarchy until the bottom of the hierarchical tree is reached. The test class build method is the first to be called during the build phase. It is responsible for configuring the test bench, initiating the construction process by building the next level down in the hierarchy and by initiating the stimulus by starting the main sequence.

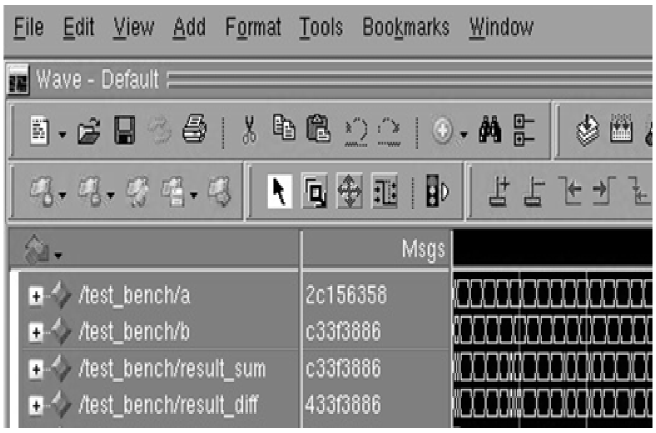
VIII. SIMULATION RESULTS

In the developed testbench, all testbench files are first compiled for any errors. Test which we want to run is specified with *+UVM_TESTNAME=NAMEOFTEST* in the command line. To run a test named random_test which is extended from uvm_test and which is registered with uvm_factory +UVM_TESTNAME= random_test is used. Once random_test starts running, first the build phase is started. It can be seen that the base_test which is extended from uvm_test is starting point for the test bench build followed by the environment, agent.



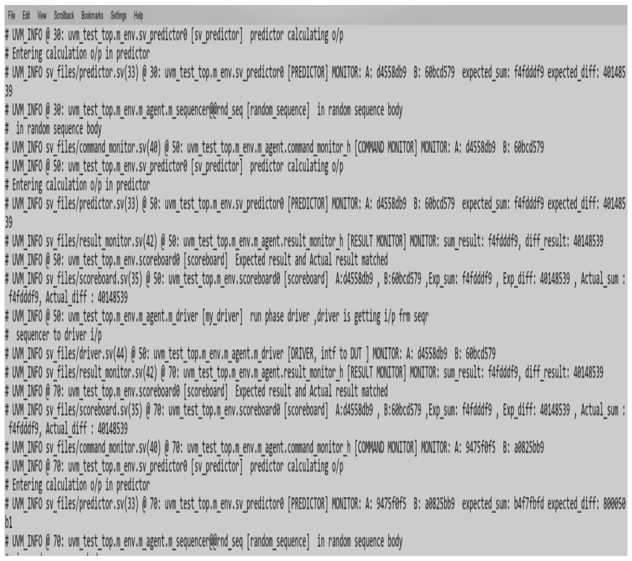
```
#
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(215) @ 0: reporter [Questa UVM] QUESTA_UVM-1.2
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(217) @ 0: reporter [Questa UVM]  questa_uvm::init(+struct)
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(192) @ 0: reporter [Questa UVM] Add Phase
# UVM_INFO @ 0: reporter [RNTST] Running test random_test...
# UVM_INFO @ 0: uvm_test_top [random_test]  Base_test in build phase
# UVM_INFO @ 0: uvm_test_top.m_env [my_env] Environment in BUILD_PHASE
# Entering build in my_environment
# UVM_INFO @ 0: uvm_test_top.m_env.m_agent [my_agent] Agent in BUILD_PHASE
# Entering build in my_agent
# UVM_INFO @ 0: uvm_test_top.m_env.m_agent [my_agent] Agent is in ACTIVE
# UVM_INFO @ 0: uvm_test_top.m_env.m_agent.command_monitor_h [command_monitor] command monitor in build_PHASE
# Entering build in command monitor
# UVM_INFO @ 0: uvm_test_top.m_env.m_agent.result_monitor_h [result_monitor] result monitor in build_PHASE
# Entering build in result monitor
# UVM_INFO @ 0: uvm_test_top.m_env.scoreboard0 [scoreboard]  scoreboard in build_PHASE
# Entering build in scoreboard
# UVM_INFO @ 0: uvm_test_top.m_env.sv_predictor0 [sv_predictor]  predictor in build_PHASE
# Entering build in predictor
# UVM_INFO @ 0: uvm_test_top.m_env.m_agent [my_agent] Agent in Connect_PHASE
# Entering Connect in my_agent
# UVM_INFO @ 0: uvm_test_top.m_env [my_env] Environment in Connect_PHASE
# Entering connect in my_environment
# UVM_INFO verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv(272) @ 0: reporter [Questa UVM] End Of Elaboration
```

Figure 11. Build and Connect Phases in Testbench



```
/test_bench/a           2c156358
/test_bench/b           c33f3886
/test_bench/result_sum  c33f3886
/test_bench/result_diff 433f3886
```

Figure 12. Output Waveform



```
# UVM_INFO @ 30: uvm_test_top.m_env.sv_predictor0 [sv_predictor]  predictor calculating o/p
# Entering calculation o/p in predictor
# UVM_INFO sv_files/predictor.sv(33) @ 30: uvm_test_top.m_env.sv_predictor0 [PREDICTOR] MONITOR: A: d4558db9  B: 60bcd579  expected_sum: f4fdddf9 expected_diff: 401485
39
# UVM_INFO @ 30: uvm_test_top.m_env.m_agent.m_sequencer@@rnd_seq [random_sequence]  in random sequence body
#  in random sequence body
# UVM_INFO sv_files/command_monitor.sv(40) @ 50: uvm_test_top.m_env.m_agent.command_monitor_h [COMMAND MONITOR] MONITOR: A: d4558db9  B: 60bcd579
# UVM_INFO @ 50: uvm_test_top.m_env.sv_predictor0 [sv_predictor]  predictor calculating o/p
# Entering calculation o/p in predictor
# UVM_INFO sv_files/predictor.sv(33) @ 50: uvm_test_top.m_env.sv_predictor0 [PREDICTOR] MONITOR: A: d4558db9  B: 60bcd579  expected_sum: f4fdddf9 expected_diff: 401485
39
# UVM_INFO sv_files/result_monitor.sv(42) @ 50: uvm_test_top.m_env.m_agent.result_monitor_h [RESULT MONITOR] MONITOR: sum_result: f4fdddf9, diff_result: 40148539
# UVM_INFO @ 50: uvm_test_top.m_env.scoreboard0 [scoreboard]  Expected result and Actual result matched
# UVM_INFO sv_files/scoreboard.sv(35) @ 50: uvm_test_top.m_env.scoreboard0 [scoreboard]  A:d4558db9 , B:60bcd579 ,Exp_sum: f4fdddf9 , Exp_diff: 40148539 , Actual_sum :
 f4fdddf9, Actual_diff : 40148539
# UVM_INFO @ 50: uvm_test_top.m_env.m_agent.m_driver [my_driver]  run phase driver ,driver is getting i/p frm seqr
#  sequencer to driver i/p
# UVM_INFO sv_files/driver.sv(44) @ 50: uvm_test_top.m_env.m_agent.m_driver [DRIVER, intf to DUT ] MONITOR: A: d4558db9  B: 60bcd579
# UVM_INFO sv_files/result_monitor.sv(42) @ 70: uvm_test_top.m_env.m_agent.result_monitor_h [RESULT MONITOR] MONITOR: sum_result: f4fdddf9, diff_result: 40148539
# UVM_INFO @ 70: uvm_test_top.m_env.scoreboard0 [scoreboard]  Expected result and Actual result matched
# UVM_INFO sv_files/scoreboard.sv(35) @ 70: uvm_test_top.m_env.scoreboard0 [scoreboard]  A:d4558db9 , B:60bcd579 ,Exp_sum: f4fdddf9 , Exp_diff: 40148539 , Actual_sum :
 f4fdddf9, Actual_diff : 40148539
# UVM_INFO sv_files/command_monitor.sv(40) @ 70: uvm_test_top.m_env.m_agent.command_monitor_h [COMMAND MONITOR] MONITOR: A: 9475f0f5  B: a0825bb9
# UVM_INFO @ 70: uvm_test_top.m_env.sv_predictor0 [sv_predictor]  predictor calculating o/p
# Entering calculation o/p in predictor
# UVM_INFO sv_files/predictor.sv(33) @ 70: uvm_test_top.m_env.sv_predictor0 [PREDICTOR] MONITOR: A: 9475f0f5  B: a0825bb9  expected_sum: b4f7fbfd expected_diff: 800050
b1
# UVM_INFO @ 70: uvm_test_top.m_env.m_agent.m_sequencer@@rnd_seq [random_sequence]  in random sequence body
```

Once all the components of the test bench are built the build phase is completed. The next phase is the connect phase, which is bottom up. First, the child components starting from agent, predictor, scoreboard and environment are connected during the connect phase.

The screenshots were taken when random_test is run on the developed testbench in Questa Simulator from Mentor Graphics supporting UVM. The uvm_report_info and $display are used in the build, connect and run phase of the components for getting the information, as to know what is happening in the test bench. It is advisable to use uvm_debug in the command line option to get more detailed information which is stored in transcript file, generated by the simulator when the test is completed. A waveform file (.wlf) is generated automatically by the simulator once the test finishes. The wave file which consists of the signals from the DUT can be used for debugging the cause of the failure.

ACKNOWLEDGMENT

The proposed work was carried out in Stealth mode Company working on ASIC based designs. This work was greatly supported by National Institute of Technology, Warangal.

REFERENCES

[1] Sharon Rosenberg and Kathleen A Meade, “A Practical Guide to Adopting the Universal Verification Methodology(UVM)”, Cadence Design Systems, 2010

[2] IEEE Computer Society. IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language - IEEE 1800-2009.

[3] Young-Nam Yun, Jae-Beom Kim, Nam-Do Kim,Byeong Min ,” Beyond UVM for practical Soc Verification” , year 2011,pp 158-162.

[4] Accellera Organization, Inc. Universal Verification Methodology (UVM) May 2012

[5] J. Bergeron, “Writing Testbenches: Functional Verification of HDL models”, Kluwer Academic Publishers, 2003.

[6] www.uvmworld.org

[7] Jongwook Sohn and E. E. Swartzlander, Jr., "Improved Architectures for a Fused Floating-Point Add-Subtract Unit," IEEE Trans. on Circuits and Systems-I, vol 59, pp. 2285-2291, Oct. 2012