2018 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Challenges and Mitigations of Porting a UVM Testbench from Simulation to Transaction-Based Acceleration (Co-Emulation)

Vikas Billa, Microsemi India Pvt. Ltd., Hyderabad, India. vikas.billa@microsemi.com

Sundar Haran, Microsemi India Pvt. Ltd, Hyderabad, India. Sundararajan.Haran@microsemi.com

***Abstract*— Growing SoC designs are pushing the limits of massive system-level scenarios in simulation platforms. Its need of the hour, that verification experts port their complex testbench and DUT to Emulation platforms. This could be the way going forward in augmenting the overall performance, faster verification closure and exercising system level scenarios that might require longer simulation runtime and huge disk consumption at run time.**

Today's traditional verification flow involves verification at multiple abstraction levels; accordingly, the testbench needs to be adjusted/modified from RTL simulation to hardware acceleration/emulation. Simulation offers a great springiness in debugging and the emulation offers mammoth performance gain, an ideal solution is to make use of these offerings to develop a single, unified testbench that can be used for both simulation and emulation platforms, which helps in enhancing the overall performance, productivity and faster verification closure. In this paper, we will discuss a case study based on one of our native UVM testbench, we partitioned the testbench into two top architecture that can be used not only for software simulation, but also for hardware acceleration/emulation without compromising on simulation competences such as coverage-driven, constraint-random and assertion-based verification techniques.

The key for the future projects is to plan the emulation portable as soon as we start the testbench development to avoid creating extra work. Creating emulation-ready testbench needs careful architectural consideration, but the performance benefits can be substantial. We will also touch upon on the performance improvements, coding guidelines in developing the accelerated testbench and the efforts required for porting it from the native simulation UVM testbench.

Keywords—Emulation, Simulation, Two Top Arhcitecture, HDL, HVL.

I. INTRODUCTION

In this paper, the authors have used Mentor's Veloce TBX solution to develop the emulation ready testbench. To create a unified testbench for both simulation and emulation we need to adhere to the below steps.

1. *Employing two separate domains (two top architecture [2]): an untimed hardware verification language (HVL-TOP) domain and a synthesizable hardware description language (HDL-TOP) domain.*

2. *Modeling all the timed testbench code for emulator synthesis in the HDL domain (BFM), leaving the HVL domain untimed (proxy).*

3. *A transaction-level, probably an interface task/function or a pipe based approach (which approach to be used depends on the testbench implementation i.e. streaming or reactive based on the protocol) to be used as a communication API between the HVL and HDL domains, Only packed data type is allowed to be passed via communication API's.*

a. Two-Top Architecture:

The primary requirement for the two top architecture is to have the HVL and HDL top-level module hierarchies as shown in Figure 1. The HDL domain must be synthesizable i.e. logic present in Driver BFM, Monitor BFM, DUT, clock and reset generators. The HVL domain contains non-synthesizable code such as logic present in transactions, scoreboards, coverage collectors etc.

The communication from either way i.e. from HDL to HVL or HVL to HDL establishes through a communication API such as an interface task/function or a pipe based approach. The classes on the HVL side, which acts as a proxy to the BFM interfaces, will call appropriate tasks and functions declared inside the BFMs via virtual interface handles to drive and sample DUT signals.

The key challenge in porting to an emulation ready testbench needs a careful architectural consideration and also needs proper communication API channel from HDL to HVL or HVL to HDL.

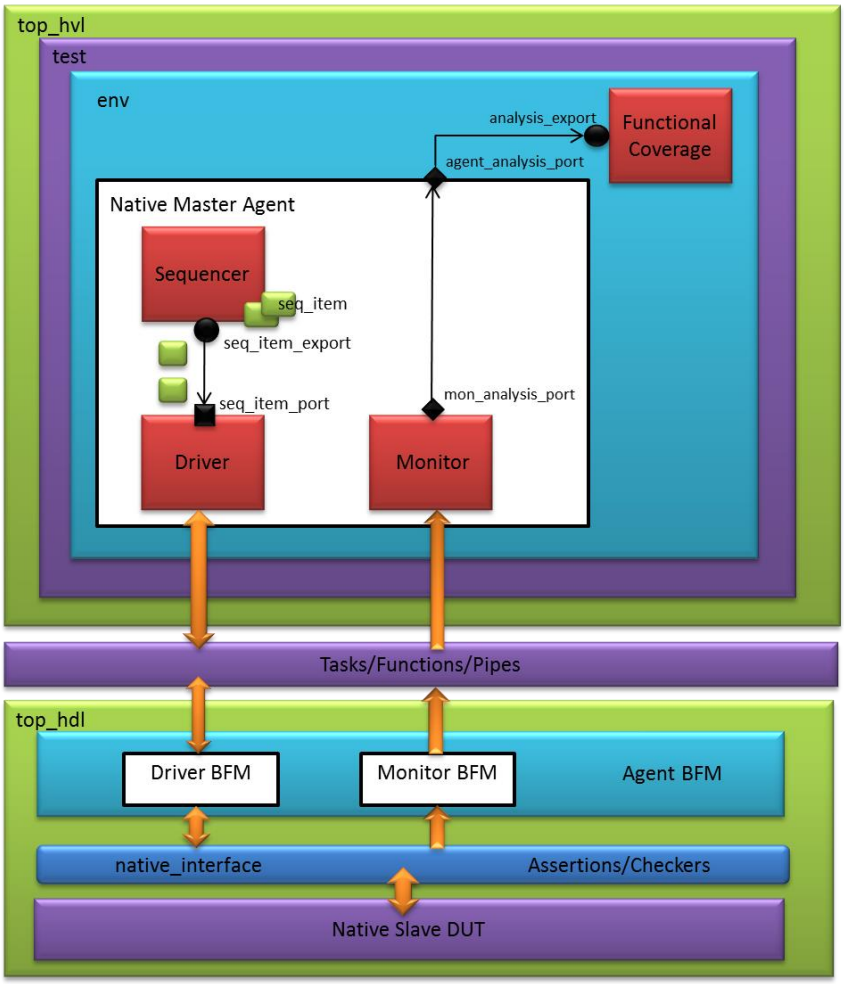

Figure 1: Two- Top Architecture

II. A CASE STUDY

This paper is a collective case study of PolarFire project executed by the authors in the organization. PolarFire is an FPGA with an ARM Cortex M3 Processor and programmable analog, offering full customization, IP protection and ease-of-use.

The following architecture details the functional partition of the PolarFire, This is a logical diagram and doesn't reflect actual physical floorplan (neither positioning nor sizing of blocks), as the diagram is mainly intended to explain the main architectural features.

The PolarFire Architecture is mainly categorized into 3 blocks namely Fabric, contoller and analog. The FPGA fabric is a non-volatile block, which facilitates the implementation of programmable user logic. The subsystem as an ARM cortex M3 processor which is used for initialization and programing besides other functionalities such as power management and security etc.

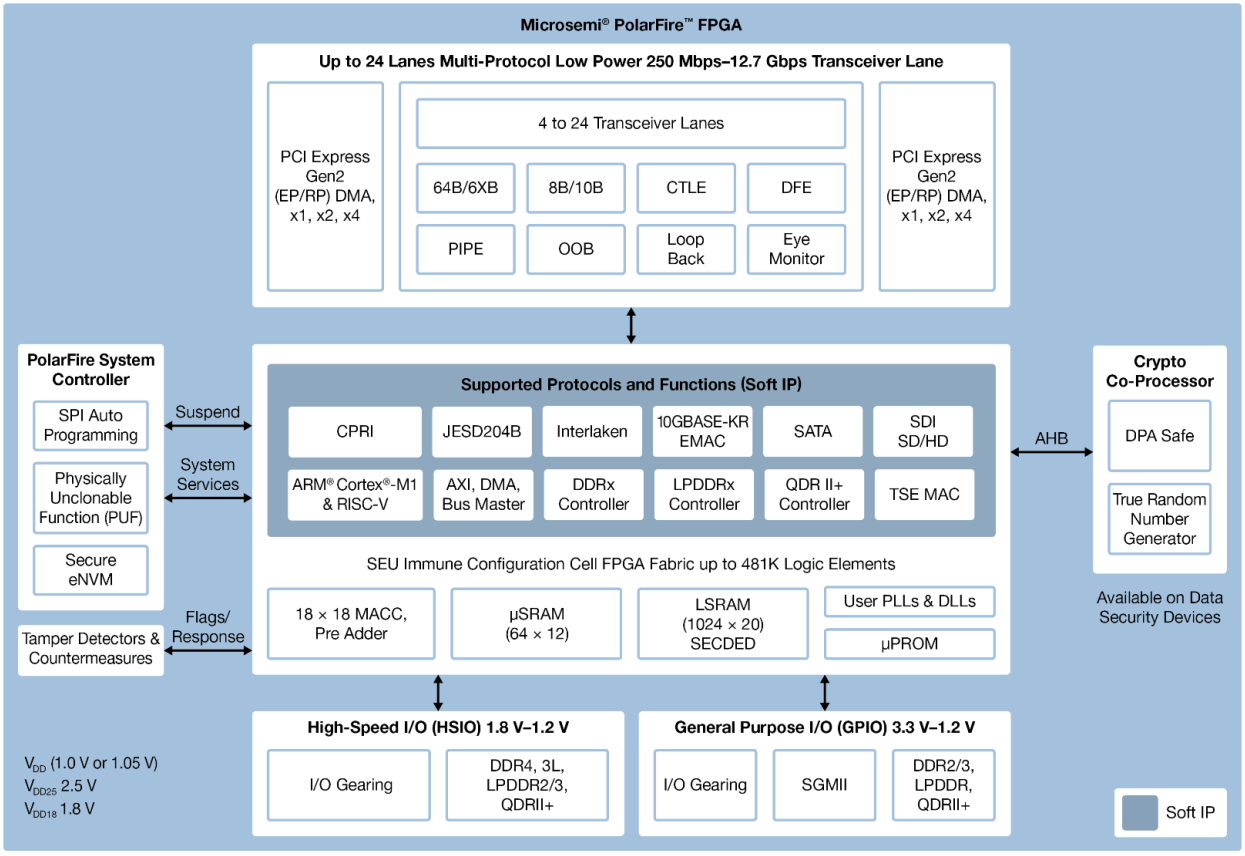

Figure 2: PolarFire Architecture

Reference: https://www.microsemi.com/products/fpga-soc/fpga/polarfire-fpga

The verification of PolarFire involves thorough testing of all the functional data paths in addition to basic connectivity checks at the top level to ensure that the design works correctly. The authors have used the best of UVM to implement the reusable verification environment achieving the overall productivity and faster verification closure.

a. Simulation Challenges:

Adopting the UVM does not address the other verification needs such as the ability to run, debug, and collect metrics for a large number of tests in a short amount of time. The speed of the simulation is the primary bottleneck as modern SOC/ASICs are getting bigger and bigger in terms of gate counts and supported features. Limiting the number of simulation tests to meet requirements of tight schedules is alarming and raises doubt about the completeness of verification. The remedy for ever increasing simulation times is using emulation techniques.

b. Adopting Emulation:

As discussed in the above section to overcome the simulation limits, the authors have decided to have a emulation ready verification environment for their PolarFire derivate projects, they have analyzed their current verification architecture and decided to restructure it to the Two-Top Architecture as discussed in figure 1.

c. Behavioral Models:

In our verification environment we used behavioral models for analog programming and for fabric blocks; we have ported 78 models to emulation platform which was a huge effort i.e. it took us around 3 months to completely port to synthesizable models.

Few changes in the HDL models are mentioned below which came across while converting it to synthesizable models.

1. #delays are not supported in Veloce; replaced with @ posedge clk or @ negedge clk, by using the internal clock generators.
2. Converted real datatype to integer datatype.
3. Removed tranif0 and tranif1 as is not supported in Veloce.
4. .vams files are recoded to .v files.

Fabric simulation model

```
module fabric_model();

//code not shown

 for (i=0; i<80; i++) begin
 tranif0 t0 (x_blnl[i],x_gbl[i], x_bln_gbl_sell_b);
 end

endmodule
```

Fabric emulation model

```
module fabric_model();

//code not shown

 for (i=0; i<80; i++) begin
 // tranif0 t0 (x_blnl[i],x_gbl[i], x_bln_gbl_sell_b);
assign x_blnl[i] = (x_bln_gbl_sell_b  === 1'b0) ?
x_gbl[i]  : 'bx;

 end

endmodule
```

As tranif0 and tranif1 is not supported in Veloce, the logic has been replicated using assign statement in the emulation model.

simulation model

```
module c_model();

//code not shown

always@(posedge clk)begin
 addr=$random();
end

endmodule
```

emulation model

```
module c_model();

//code not shown
int myseed = 10;

always@(posedge clk)begin
 //addr=$random();
  addr=$random(my_seed);
end
```

In Veloce we need to pass seed as $random (my_seed), seed value can be assigned in the module declarations or else can be passed through command line using $value$plusargs as shown below.
Usage: make all +RANDOM_SEED=200

```
initial begin
 if($value$plusargs ("RANDOM_SEED=%d ", my_seed)) begin
        my_seed=seed;
 end
```

d. Verification IP's:

The Microcontroller subsystem shown in figure 2 has 12 IP's out of which 10 are native protocol IP's and 2 are general protocol IP's.so correspondingly there are 10 native VIP's in the current verification environment which are needed to be ported to emulation by using Mentor Graphics TBX Flow.

For Generic Protocol VIP's Mentor Graphics has provided Veloce Transactor Library (VTL). The Veloce Transactor Library is an accelerated Verification IP which is easy to use and reduces the overall testbench development time.

The main challenge here is to port all the existing Simulation VIP's to emulation ready VIP's in a specified time. Before actual porting, the authors have followed the below steps.

1. Initially the authors went through the Veloce user guide, understood the emulation importance and Veloce architecture.
2. Ported a native protocol simulation VIP to emulation Platform, went through numerous phases in understanding the two-top architecture in practical. It took us several debug cycles to bring up the initial version of emulation ready VIP, all the best practices which we found during the porting are mentioned in coding guidelines section.
3. We have developed a two-top architecture UVM template generator script after our initial native VIP porting, which has generated all the files related to HDL and HVL domains, this made our work easier for other set of VIP porting.

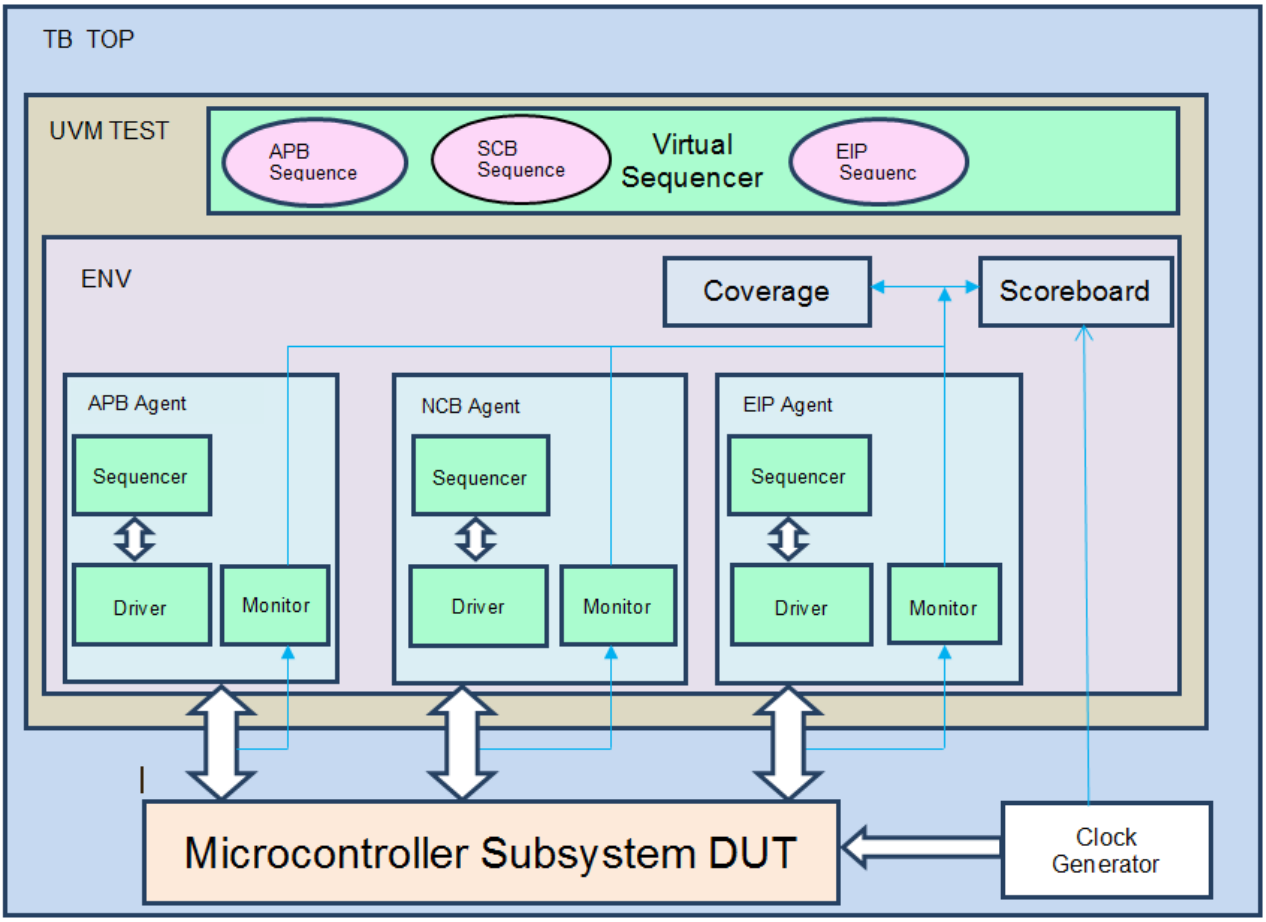

Figure 3: Microcontroller subsystem - Simulation Verification Architecture

As discussed the Microcontroller subsystem verification environment has 12 Verification IP's, for simplicity we have shown only 3 VIP's in the above figure i.e. Advance Peripheral Bus (APB) UVC, Native Configure Bus (NCB) UVC and External IP (EIP) UVC.

During the course of porting we have faced many challenges such as remodeling the exact behavior in HDL BFM's, using of correct pragma's, pragma is placed in HDL side to intimate the Veloce that the piece of code should be synthesized etc.

The below code snippets shows the major changes from simulation environment to the two top architecture environment.

HVL Driver

```
class apb4_master_driver extends uvm_driver
#(apb4_transaction_c);
   virtual apb4_master_driver_bfm BFM;
 // constructor
task run_phase(uvm_phase phase);
       pkt_t req,rsp;
       forever begin
          apb4_master_seq_item_s req_s, rsp_s;
          seq_item_port.get_next_item(req);

apb4_master_seq_item_converter::from_class(req,
req_s);
          BFM.drive_data(req_s, rsp_s);

apb4_master_seq_item_converter::to_class(rsp,
rsp_s);
           $cast(rsp, req.clone());
          rsp.set_id_info(req);
          seq_item_port.item_done(rsp);
       end// !forever begin
   endtask : get_and_drive
endclass : apb4_master_driver
```

HDL Driver

```
interface apb4_master_driver_bfm (apb4_interface
APB);
//pragma attribute apb4_master_driver_bfm
partition_interface_xif
   string tID;

   import
apb4_master_shared_pkg::apb4_master_seq_item_s;

task drive_data(apb4_master_seq_item_s req, output
apb4_master_seq_item_s rsp ); // pragma tbx xtf

   @(posedge APB.PCLK);
// code not shown here
endtask: drive_data

endinterface: apb4_master_driver_bfm
```

In the above code snippet, the apb4_master_driver has a virtual interface handle BFM from the corresponding apb4_master_driver_bfm BFM model which is synthesizable. The time consuming tasks are placed in the HDL driver and can be called by the HVL Driver as shown above.

HVL Monitor

```
class apb4_master_monitor extends uvm_monitor;

   virtual apb4_master_monitor_bfm BFM;
  // code not shown
    uvm_analysis_port #(item_t) sb_post;

    function void
end_of_elaboration_phase(uvm_phase phase);
       BFM.proxy = this;
   endfunction: end_of_elaboration_phase

   task run_phase(uvm_phase phase);
       forever begin
         BFM.collect_data();
       end
   endtask : run_phase

    function void write(apb4_master_seq_item_s
item_s);
       item_t item;
apb4_master_seq_item_converter::to_class(item,
item_s);
       this.item.copy(item);
       sb_post.write(this.item);
   endfunction: write
endclass : apb4_master_monitor
```

HDL Monitor

```
interface apb4_master_monitor_bfm
(apb4_interface APB);
// pragma attribute apb4_master_monitor_bfm
partition_interface_xif

   import
apb4_master_shared_pkg::apb4_master_seq_item_s;
   import apb4_agent_pkg::apb4_master_monitor;

   apb4_master_monitor proxy; // pragma tbx oneway
proxy.write

task collect_data(); // pragma tbx xtf

       apb4_master_seq_item_s item;

       @(posedge APB.PCLK);

       //code not shown here

      proxy.write(item);

endtask : collect_data

endinterface : apb4_master_monitor_bfm
```

For completeness of the transaction the response need to be sent back to HVL from HDL, this is happening through the proxy.write(item) method called in HDL and is implemented in the HVL as shown above.

hvl_top

```
module top_tb();

import uvm_pkg::*;
`include "uvm_macros.svh"

import apb4_agent_pkg::*;

`include "apb4_master_demo_tb.sv"
`include "apb4_master_test_lib.sv"

initial
begin
   $timeformat( -9, 3, " ns", 12);
   run_test();
end

endmodule : top_tb
```

hdl_top

```
module top_hdl();
`include "timescale.v"

logic PCLK;
logic PRESETn;

apb4_interface APB(PCLK, PRESETn);  // APB interface

// tbx vif_binding_block
initial begin
  import uvm_pkg::uvm_config_db;
  uvm_config_db #(virtual apb4_interface)::set(null,
"uvm_test_top", $psprintf("%m.APB") , APB);
end

apb4_master_monitor_bfm
APB_MONITOR(APB.apb4_mon_mp);
apb4_master_driver_bfm  APB_DRIVER (APB.apb4_mp);

// tbx vif_binding_block
initial begin
  import uvm_pkg::uvm_config_db;
  uvm_config_db#(virtual
apb4_master_driver_bfm)::set(null,
"uvm_test_top",$psprintf("%m.APB_DRIVER"),APB_DRIV
ER);
  uvm_config_db#(virtual
apb4_master_monitor_bfm)::set(null,"uvm_test_top",$pspri
ntf("%m.APB_MONITOR"),APB_MONITOR);
end

// DUT instance

// Clock and reset initial blocks
//tbx clkgen
Initial
forever #5 PCLK=~PCLK;

endmodule: top_hdl
```

In the hvl_top we have used only the run_test() and in the hdl_top the apb4_interface, apb4_master_monitor_bfm , apb4_master_driver_bfm are instantiated and are set using uvm_config_db. The hdl_top also has the dut instance, clock, and reset generators.

III. CODING GUIDELINES

Few coding guidelines are mentioned below in-order to have a first cut emulation ready VIP.

a. Guideline 1: fork join

To achieve parallel process in SystemVerilog we use fork join construct, but this fork join is not synthesizable in HDL i.e. in driver bfm and monitor bfm files, in order to achieve such logic in HDL the fork join is moved to HVL domain, and the calle task is in HDL as shown below. The parallel process can also be achieved in HDL by using multiple procedural.

Emulation (HVL Part)

```
task run_phase(uvm_phase phase);
 box_transaction_c req;
box_transaction_c rsp;
BFM.idle_data();
forever begin
  box_master_seq_item_s req_s, rsp_s;
  seq_item_port.get_next_item(req);
  // to class
  if(req.fab_box_write == 1) begin
     BFM.write_data(req_s, rsp_s);
  end
  else if(req.fab_box_read == 1) begin
          fork
            BFM.read_data(req_s);
            BFM.read_capture_data(req_s, rsp_s);
          join
       end
   else begin
          `uvm_info(tID,$sformatf("Invalid
Combination"),UVM_MEDIUM)
            end
    // code not shown here
     end// !forever begin
  endtask : run_phase
```

Emulation (HDL Part)

```
interface box_master_driver_bfm (box_interface BOX);
  import
box_master_shared_pkg::box_master_seq_item_s;

task idle_data(); // pragma tbx xtf
 // code not shown
enstask: idle_data

task write_data(box_master_seq_item_s req, output
box_master_seq_item_s rsp); // pragma tbx xtf
      @(posedge BOX.fab_box_clk)
     // code not shown here
endtask: write_data

task read_capture_data(box_master_seq_item_s req,
output box_master_seq_item_s rsp);
   // code not shown
endtask: read_capture

task read_data(box_master_seq_item_s req); // pragma
tbx xtf
  // code not shown
endtask:: read_data
endinterface: box_master_driver_bfm
```

b. Guideline 2: configurations

As discussed the communication between HVL to HDL happens through packed structure or through pipe based approach. Care should be taken while we are configuring the agent as UVM_PASSIVE, as HDL driver is an interface it will be compiled and potential errors such as multiple drivers driving the net may occur in the top level environment. The key element is to enable the HDL drive_data task logic only if the is_active configuration is UVM_ACTIVE.

```
interface box_master_driver_bfm (box_interface BOX);
  import box_master_shared_pkg::box_master_seq_item_s;
  import box_master_shared_pkg::box_master_config_item_s;

 task  drive_data(box_master_seq_item_s req, box_master_config_item_s req_config,output
box_master_seq_item_s rsp); // pragma tbx xtf
      @(posedge BOX.fab_box_clk)
     if(req_config.is_active == 1'b1) begin // UVM_ACTIVE
      // code not shown here
    end
endtask: write_data

endinterface: box_master_driver_bfm
```

c. Guideline 3: $display control at runtime in HDL:

For runtime controllability, use test_plusargs/value_plusargs as shown below, VEL_INFO define is declared for $display usage, a set of 5 different verbosities such as HDL_UVM_NONE, HDL_UVM_LOW and others are defined as shown below.

For VEL_INFO define we are passing verbosity as 3rd argument and the global verbosity is passed through command line. If glbl_verbose is greater than the verbosity which passed through VEL_INFO define then the display will be written else it will be not written. Note that $display creates infrastructure i.e. in terms of area.

defines.sv

```
`define HDL_UVM_NONE       0
`define HDL_UVM_LOW        1
`define HDL_UVM_MEDIUM     2
`define HDL_UVM_HIGH       3
`define HDL_UVM_DEBUG      4
int glbl_verbose;
// Display Define
`define VEL_INFO(strID="", msg="",verbosity) \
if($test$plusargs("HDL_UVM_DEBUG")) \
   glbl_verbose = 4; \
if($test$plusargs("HDL_UVM_HIGH")) \
   glbl_verbose = 3; \
if($test$plusargs("HDL_UVM_MEDIUM")) \
   glbl_verbose = 2; \
if($test$plusargs("HDL_UVM_LOW")) \
   glbl_verbose = 1; \
if($test$plusargs("HDL_UVM_NONE")) \
   glbl_verbose = 0; \
if(glbl_verbose >= verbosity )  \
   $display("UVM_INFO @ %0t : %m[%s] %s", $time, strID, msg);
```

Counter.sv

```
module counterud (CLK, CLR, UP_DOWN, Q);
input CLK, CLR, UP_DOWN;
output [3:0] Q;
reg    [3:0] tmp = 0;

  always @(posedge CLK)
    begin
      if (CLR) begin
        tmp = 4'b0000;
        `VEL_INFO("counter", "Cleared", `HDL_UVM_MEDIUM);
      end
      else
          if (UP_DOWN) begin
            tmp = tmp + 1'b1;
            `VEL_INFO("counter", "Incrementing", `HDL_UVM_LOW);
          end
          else begin
               tmp = tmp - 1'b1;
               `VEL_INFO("counter", "Decrementing", `HDL_UVM_HIGH);
             end
         end
  assign Q = tmp;
endmodule
```

Usage: make all +HDL_UVM_MEDIUM

As per the usage mentioned above only the `VEL_INFO, which has verbosity less than or equal to the global verbosity i.e. HDL_UVM_MEDIUM, will be printed.

d. Guideline 4: $urandom_range equivalent code in HDL:

Systemverilog $urandom_range(MIN,MAX) construct is not synthesizable in HDL domain, we have replicated this behavior as shown below with $random(seed) as it is synthesizable.

```
module random();
logic CLK;
bit cnt=1;
int unsigned seed, my_seed;
bit [2:0] addr;
int unsigned MAX=6, MIN=2;

initial begin
 if($value$plusargs ("RANDOM_SEED=%d ", my_seed)) begin
        my_seed=seed;
  end
end
always@(posedge CLK) begin
   addr=$random(my_seed);
    addr= MIN+(addr %(MAX-MIN));
   $display("*****************************");
   $display( "Random Range addr=%0d", addr   );
   $display("*****************************");
end
// tbx clkgen
initial begin
  CLK = 0;
  forever begin
    #5ns CLK = ~CLK;
  end
end
endmodule
```

CONCLUSION

In this paper, we have discussed on how to port a simulation environment to a emulation ready UVM framework by using Mentor Veloce TBX Flow. To summarize we have shared the challenges faced and coding guidelines. The key highlights in porting are to understand the two-top architecture and the communication API between HDL and HVL and vice versa. The key guideline in porting is to know the usage of appropriate pragma's. It took us 30 man days to port the simulation environment to emulation environment. In the future work, we will share more details on the performance, as in the current environment we have black boxed few models for ease of use, we will revisit and share the details accordingly.

This ported emulation UVM testbench can also be used in simulation platform, thus achieving a unified testbench without conceding any of the UVM capabilities.

FUTURE WORK

The Behavioral models and VIP's are ported to emulation platform and are tested at block level, currently the authors are working on integrating these models and VIP's in to the top level verification environment, once this is done they are planning to do a performance analysis between pure simulation and emulation environments.

ACKNOWLEDGMENT

We profoundly thank colleagues and management team at Microsemi India Pvt. Ltd, Hyderabad and Rohit Malyan, Emulation Consultant, Mentor Graphics for their valuable guidance and thought provoking discussions.

REFERENCES

[1] UVM User Manual, uvmworld.org

[2] UVM Cookbook - Emulation, Mentor Verification Academy

[3] Standard Co-emulation Modeling Interface (SCEMI) Version 2.2, Accellera, January 2014

[4] Master.pdf – veloce user guide

[5] How to Boost Verification Productivity with SystemVerilog/UVM and Emulation, Hans van der Schoot, DVCon Europe 2015.

[6] STMicroelectronics: Simulation + Emulation = Verification Success.