Using Simulation Acceleration to achieve 100X performance improvement with UVM based testbenches

Narla Venkateswara Rao , Avago Technologies, Bangalore, India, venkateswararao.narla@avagotech.com

K Ranjith Kumar, Avago Technologies, Bangalore, India, ranjith.k@avagotech.com

Vikas verma, Avago Technologies, Bangalore, India, vikas.verma@avagotech.com

Gautam Kumar, Avago Technologies, Bangalore, India, gautam.kumar@avagotech.com

Abstract

Time to market is one of the most important factors in determining the success of any product. This can only be possible if we have a bug-free product, on time. If a quality-based-product is released within a specified time frame, it is guaranteed to win the market. There is a huge amount of verification effort and design-cycle time that go into realizing these quality products of which the complexity and size are ever increasing. The verification effort needs to be augmented with the pre-silicon validation to iron-out long simulation time related issues and the same platform also acts as demonstrable platform to customers. Therefore realizing such a complex system requires modeling, verification, debug and analysis at various levels of abstraction with varying levels of precision. The advanced verification methodologies e.g. UVM and HVLs enable us to design testbenches which are automated and re-usable on various abstraction levels. With such higher complexity, the simulation throughput becomes a bottleneck. It is believed that significant portion of the development cycles are spent in functional verification. Improving simulation throughput with hardware-acceleration (synthesizable testbenches) or in-circuit emulation are two approaches, but they would result in losing the benefits of a coverage-driven constrained random verification environment, as well as discarding the earlier setup for the hardware software co-verification. Thus, an integrated solution that provides acceleration, configurability and reuse is required. This enables us to leverage the best of both worlds namely hardware providing the required speed and software providing the desired flexibility and ease-of-use of the advanced features of the HVL and the methodologies e.g. UVM.

In this paper, we describe how this can be achieved through the creation of a reusable transaction-level verification environment in HVL (System Verilog/System C/C++) for the system-level verification using hardware acceleration platform (Emulator). This environment is capable of working at different levels of abstraction. It can also be heuristically and optimally partitioned between the hardware and software such that the communication between the hardware and the software portions is in-terms of "***infrequent and information-rich***" data. In industry, we call this methodology as Transaction Based Acceleration (TBA), where one can accelerate UVM based test benches to run many times faster than on the simulation platform. This is achieved by partitioning the testbench in to software testbench (typically stimulus generation and checking part) and the hardware testbench (typically BFMs which interact with the DUT on signal base activity). Software testbench is simulated in software simulator on the conventional server whereas Hardware testbench along with DUT runs on the hardware box. The hardware testbench runs much faster than the software testbench. In order to achieve more speed, we need to make *"infrequent and information-rich"* data communication between the Software testbench and the Hardware testbench. Significant portion of our testbench should be on the hardware side, resulting in the software side as a lean and thin testbench. In certain applications, we can make both software and hardware portions run in parallel to further enhance the performance gain on the hardware acceleration platform (Emulator) over the software simulators. TBA enables us to leverage the best of both worlds namely hardware providing the required speed and software providing the desired flexibility and ease-of-use of the advanced features of the HVL and the methodologies e.g. UVM.

In this paper, we present our case study in which, we have taken UVM based testbench and partitioned the same into hardware and software testbench portions. We will explain about the mechanism of hardware–software interactions, how to access hardware memory from software, few things to keep in mind to make testbench reusable between simulation environment and TBA and we will share the performance improvements observed in testbench acceleration. We will also explain some of techniques employed, such as transaction bundling,

information rich and infrequent transactions, score boarding strategies and other useful debugging techniques which help in enhancing the overall performance gain, productivity and faster verification closure

I. INTRODUCTION

Nowadays, when the current designs complexities are increasing exponentially, we need a very strong verification environment which guarantees ensuring the full functional coverage and random scenarios. The advanced methodologies recommend the constraint driven random verification environment to achieve this. But the usage of high level of constraints causes higher simulation times for system level verification, and this has a big impact on overall design cycle time.

II. NEED FOR SIMULATION ACCELERATION

A. *Inadequate Simulation Throughput*

It is believed that almost 60%-70% of design-cycle-efforts are spent in functional verification and debugging in a project, hence it is very important to make the verification phase of the design-cycle more efficient and faster. To minimize the time spent on verification, we always look for a better verification environment, better tools and more advanced compute farm which altogether can help the company in enhancing the speed of simulation process. But no matter how fast our compute farms are, the software simulators suffer their native feature of being sequential. The software simulators fake concurrency and always run sequentially, forcing a lot of the synchronization required in the testbench-design on every tick to advance the simulation time. This very nature of simulators, being sequential, requires us to look/adopt some other/better methodology in order to overcome the simulation throughput bottlenecks in the functional verification phase.

B. *Reusable HVL Testbenches*

Some of the approaches to accelerate the simulation speed include:

- In-circuit emulation: It requires a target board on which the entire design is mapped. However, there are structural differences between the actual netlist and the mapped netlist, and this may lead to masking of some critical design issues. It may also not be available early enough in the design cycle; moreover the available solutions are very expensive in nature.
- Synthesizable Testbench (STB) Approach: The design of such a testbench can be time consuming and can be difficult to debug. In addition, these testbenches cannot take advantage of advanced testbench techniques like constraint random verification and functional coverage.

There is a need to accelerate HVL simulation testbenches as they are robust, proven and based on advanced methodologies e.g. UVM/OVM. Challenge involved here is to ensure maximum reuse between the simulation and acceleration platforms in verification phase.

III. DESIGN AND SIMULATION TESTBENCH OVERVIEW

Our design is a DMA engine which has 5 shell master ports, 1 AXI master ports, 1 AXI slave port. In the design, the initial programming is done using AXI slave port. Then the DUT reads through one of its master ports and writes through another master port. The data transfer commands have a format called as command descriptor buffer (CDB) which can be chained together to form a chain. Each of the commands can be of simple elements or scatter gather elements. Data sizes can vary from few bytes to bug chunks. Once a CDB transfer is complete, a completion interrupt is triggered by the DUT.

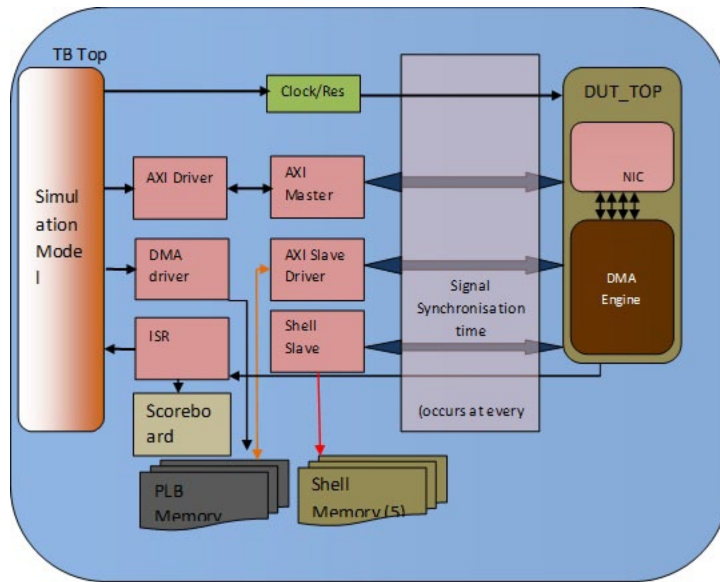

Figure 1: Simulation Acceleration Timing Profile

IV. Setting up the Transaction-based Acceleration Framework

A. Planning Simulation Acceleration approach

Transaction-based acceleration (TBA) offers the required boost in the simulation throughput without compromising on any of the feature of advanced verification methodology. It allows the reuse of almost all the components of the simulation platform testbench with no or minimal modification. With a careful and methodological planning of verification environment, TBA approach can provide a significant level of simulation throughput improvement. Once the methodology is established, the initial testbench can be easily and quickly brought up.

In order to use TBA effectively, these two basic rules must be followed.

- Maximize overall performance by optimally partitioning the testbench between SW and HW
- Maximize simulation testbench reuse to save development time and effort

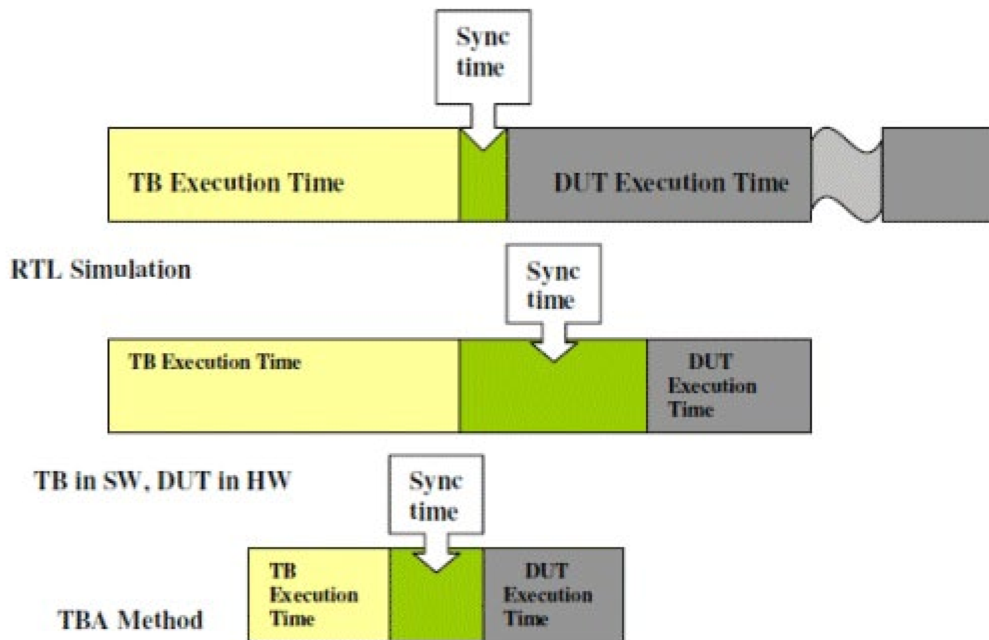

Figure 2: Simulation Acceleration Timing Profile

B. Two Top Modules (sw_top and hw_top) Architecture

As the conventional single top testbench architecture is not suited for TBA, the first step is to rearrange and create dual HVL (SW) and HDL (HW) top level module hierarchies. The HDL side must be synthesizable and should contain essentially all clock synchronous code, namely the RTL DUT, clock and reset generators and the BFM code for driving and sampling DUT interface signals. The HVL side should contain all other (untimed) testbench code including the various transaction-level testbench generation and analysis components and proxies for the HDL transactors.

In the acceleration testbench architecture, the sw_top runs in the simulator while the hw_top runs in the hardware box at actual speed. These two tops synchronize by way of exchanging the messages. Therefore, messages between the sw_top and the hw_top should be at the highest applicable level of abstraction. Maximizing the performance of the testbench by minimizing the time spent in the testbench would also offer the best results.

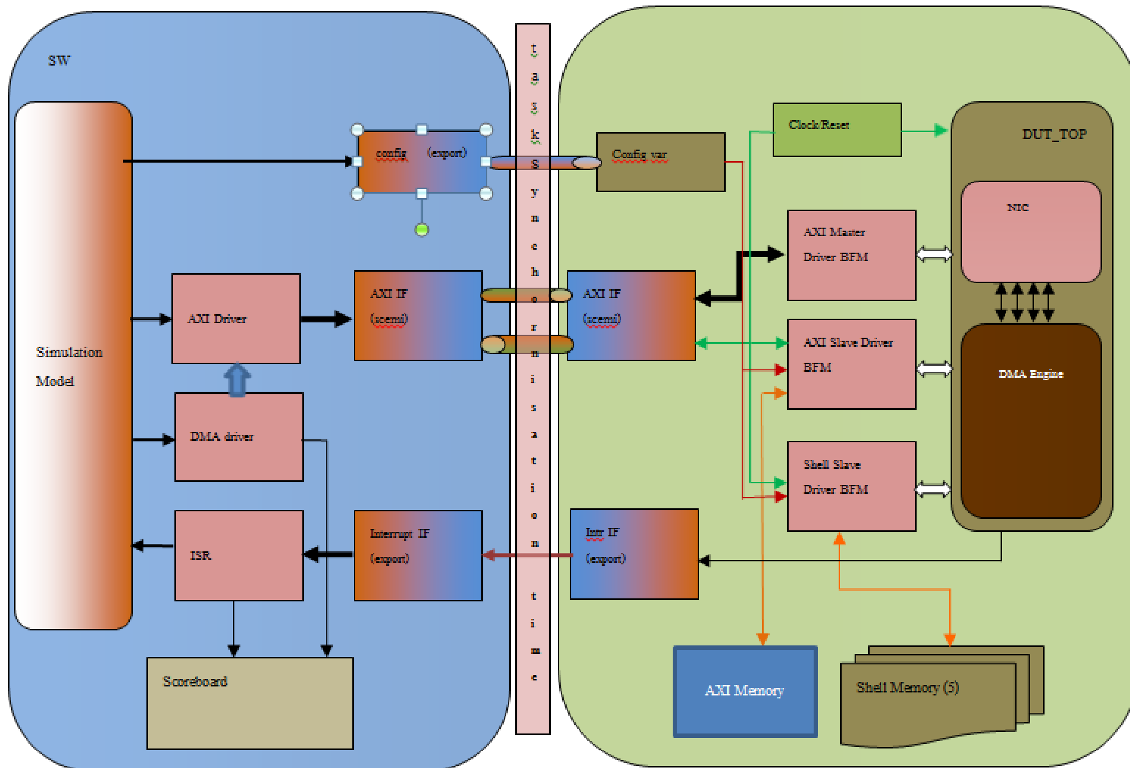

Figure 3: Acceleration friendly testbench

The architecture should incorporate the following principles:

- The most active part of the testbench (BFM/monitors) should run in the hardware at actual speed.
- The testbench that resides on the SW side should be abstracted to higher-level data items or user transaction-level API to make it run significantly faster
- Testbench profiling should be done to improve the simulation performance.
- The BFMs/monitors are the only testbench components requiring clocks. When running on the accelerator, all clocks can be generated inside the HW side partition, avoiding synchronization with the SW side on every clock edge.
- BFMs/monitors can provide or gather “transaction data” over multiple clock cycles. During these periods the HW side can run without interruption.
- Transactions are stored in a buffer on the HW side and transactions are fetched only when the buffer level falls below a threshold to decouple SW and HW interactions on cycle by cycle basis.

C. Partitioning testbench between timed and untimed

UVM based Simulation testbenches have transactors like drivers, monitors which are coded in classes to get advantages of OOPs but classes are not synthesizable types hence we need to partition the testbench in such a way that transactions are generated from software world and then it is given to hardware transactor which actually has the timing information with which the transaction should be applied to the bus and once transaction is complete then information comes back to software world.

One thing we need to make sure is that no simulation time is consumed in HVL/software side.There are 2 approaches by which we can partition the testbench in software and hardware.

1) Partitioning testbench between timed and untimed

In this approach we will implement methods in interface and those methods are exported /being made visible to software world, Software transactor calls methods of hardware transactor, in this way, all the time consuming methods are inside hardware

```
interface axi_if (input clk,input reset);
  logic [31:0] axi_addr;
  logic [31:0] axi_data_out;
  ….
always @(posedge clk or posedge reset)
  begin
  …
  end
  end

// pragmas to register the below task with sw

task wait_for_reset();
if(reset)
 @(negedge reset);
endtask

  task drive_data(inout axi_transfer_pkd_t
transfer_pkd_data);
    reset_seq();
   @ (posedge clk);
   ….
    …
   endtask
endinterface
```

```
class axi_master_driver extends uvm_driver
#(axi_transfer);
`uvm_component_utils(axi_master_driver)
virtual interface axi_if axi_vif;
// run phase
virtual task run_phase(uvm_phase phase);
   super.run_phase(phase);
   fork
      get_and_drive_vif();
   join
endtask : run_phase
//*****************************************
*******************************
// This task will collect the item and drive it to
interface
//*****************************************
*******************************
axi_transfer_pkd_t put_data;
virtual task get_and_drive_vif();

      axi_vif.wait_for_reset();
   forever
   begin
       //get next item
       seq_item_port.get_next_item(req);
       if(req.direction == RD ) put_data.direction
=0;
       if(req.direction == WR ) put_data.direction
=1;
       put_data.addr = req.addr;
       put_data.data  = req.data;
       axi_vif.drive_data(put_data);
```

In this mechanism transaction is send from software to hardware using SCEMI pipes and once transaction is received in the hardware side then transaction is applied to the bus, Hardware portion of transaction is made using interface or modules so that it can be synthesized. SCEMI pipes also have software and hardware portions

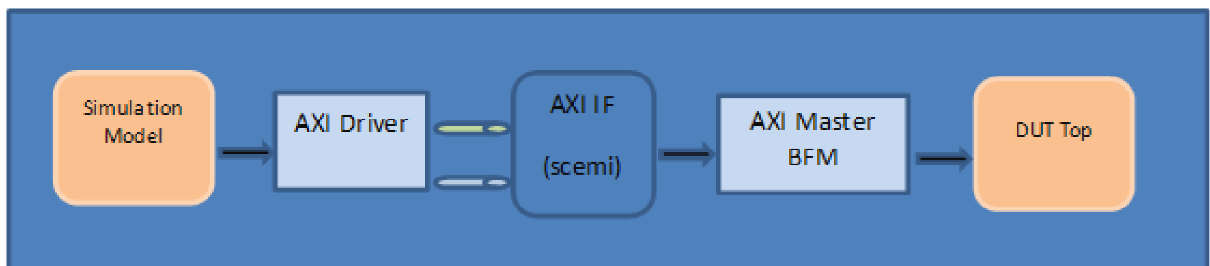

Figure 4: Scemi interface

```
class axi_master_driver extends uvm_driver
#(axi_transfer);
uvm_accel_input_pipe_proxy #(axi_transfer)
axi_ip; // output to hw bfm
 uvm_accel_output_pipe_proxy #(axi_transfer)
axi_op; // input to hw bfm
virtual function void build_phase(uvm_phase
phase);
        super.build_phase(phase);
        hw_hdl_path =
"tb_top.hw_top.u_axi_master_if";
    uvm_config_db#(string)::set(this,"axi_ip",
"hdl_path", {hw_hdl_path,".inbox_driver"});
..
endfunction
virtual task get_item_n_drive();
   pack_data_t pack_data;
   forever
   begin
       //get next item
       seq_item_port.get_next_item(item);
       //pack and put it in scemin interface
       axi_ip.put(item);
       if(item.Dir == RD)
       begin
          //wait for read data
            rcv_read_data();
            seq_item_port.item_done(item);
       end
       else
          seq_item_port.item_done();
   end
endtask : get_item_n_drive
```

```
interface axi_master_interface( input          clk,
output  [4:0]    AxiCmdType
                              …
                             );

….
scemi_input_pipe  #(4, 1,50000) inbox_driver ();
scemi_output_pipe #(4, 1,50000) outbox_driver
();
always @(posedge clk or negedge reset)
begin
    if(~reset)
    begin
      ….
    end
    else
    begin

inbox_driver.receive(1,msg_no,i_pipe_data,eom)
;
AxiCmdType      <= i_pipe_data[4:0];
endrtask
endinterface
```

D. Data exchange between hw and sw

1) Data exchange using Virtual interface

This approach is easy to implement and needs minimal changes to testbench, but in this mechanism data transfer size is fixed all the times which makes it inappropriate for the communication where transfer size varies as in that case to be able to use virtual interface data transfer size has to be maximum in all the transfers, because of which bandwidth is spent even when it is not needed

In this mechanism we have to follow below steps while sharing data between hardware and software

- Create a packed structure which has all relevant fields required between hardware and software
- Convert transaction object to packed structure in software BFM

```
typedef struct packed {
                        bit [31:0] axi_addr;
                        bit [31:0] axi_data;
                        bit        direction; //
                      } axi_transfer_pkd_t;



class axi_master_driver extends uvm_driver
#(axi_transfer);
…..
// This task will collect the item and drive it to
interface
axi_transfer_pkd_t put_data;
virtual task get_and_drive_vif();

      axi_vif.wait_for_reset();
   forever
   begin
       //get next item
       seq_item_port.get_next_item(req);
       if(req.direction == RD ) put_data.direction
=0;
       if(req.direction == WR )
put_data.direction =1;
       put_data.axi_addr = req.axi_addr;
       put_data.axi_data = req.axi_data;
       axi_vif.drive_data(put_data);
```

```
interface axi_if (input clk,input reset);
  logic [31:0] axi_addr;
  logic [31:0] axi_data_out;
  ….
always @(posedge clk or posedge reset)
  begin
  …
  end
  end

task wait_for_reset();
if(reset)
 @(negedge reset);
endtask

  task drive_data(inout axi_transfer_pkd_t
transfer_pkd_data);
    reset_seq();
   @ (posedge clk);
   ….
    …
   endtask
endinterface
```

2) Data exchange using SCEMI Pipes

SCEMI pipes are like fifos which has ends both in software and hardware works, SCEMI is Acellera standard and vendors have created wrapper around SCEMI standard to make it more user friendly

In SCEMI variable length of data can be send per transaction that makes it useful for the protocols where data sizes can vary. In using SCEMI data is serialized and de-serialized using UVM pack and unpack functions.

- hardware side need to extract the serialized data and drive the interface ,data was serialized by UVM packer
- when serial data is received then UVM unpacker is used to create transaction

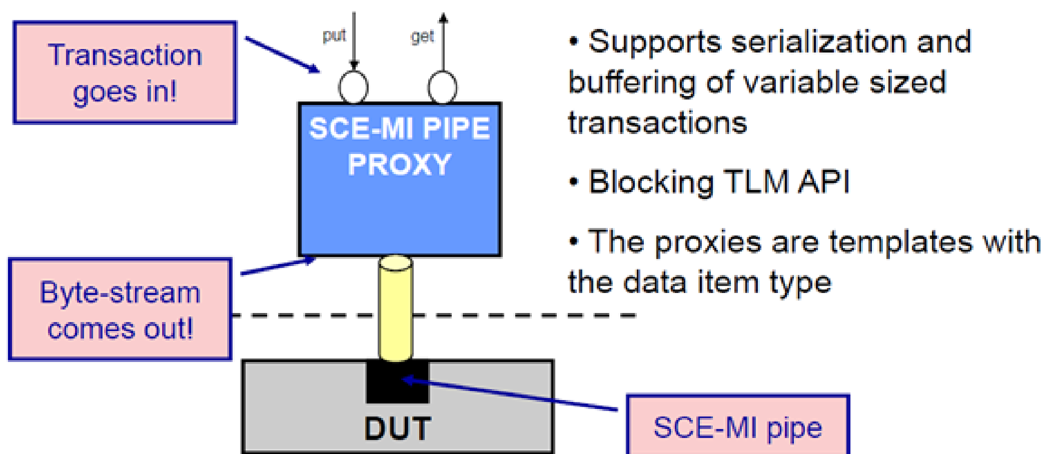

Figure5: Acceleration friendly testbench

```
class ShellSeqDataPkt extends uvm_sequence_item;
   rand  ShellCommand_e                          CmdType;      // 2 bit
   rand  bit  [(SHELL_CMD_LENGTH - 1) : 0]      CmdLength;      //16
   ….
   rand  bit  [(SHELL_DATA_WIDTH -1) : 0]      ShellOutBoundData [];
   rand  bit  [(SHELL_DATA_WIDTH -1) : 0]      ShellInBoundData  [];
   rand  bit  [(SHELL_DATA_WIDTH -1) : 0]      ShellInBoundExpectedData  [];
   `uvm_object_utils_begin(ShellSeqDataPkt)
    ….
    `uvm_field_array_int(ShellOutBoundData,UVM_ALL_ON + UVM_NOPACK )
      `uvm_field_array_int(ShellInBoundExpectedData,UVM_ALL_ON + UVM_NOPACK )
  `uvm_object_utils_end
  function void do_pack (uvm_packer packer);
     foreach(ShellOutBoundData[i])
      begin
       packer.pack_field(ShellOutBoundData[i],128);
      end
     foreach(ShellInBoundData[i])
      begin
       packer.pack_field(ShellInBoundData[i],128);
      end
   endfunction
   function void do_unpack (uvm_packer packer);
     bit [15:0] no_of_xfer;
     no_of_xfer = ((CmdLength + CmdLocalAddr[3:0] + 15)/16);
     if (CmdType==POSTED_WRITE || CmdType==NON_POSTED_WRITE) ShellOutBoundData = new
[no_of_xfer]; //size was automatically unpacked
     else ShellOutBoundData = new [0];
     if (CmdType==READ) ShellInBoundData = new [no_of_xfer]; //size was automatically unpacked
     else ShellInBoundData = new [0];
       ShellInBoundData[i] = packer.unpack_field(128);
      …..
   endfunction
```

V. Memory access between HW and SW

It's very common to access memory which is in on hardware box, few example of the same are

- Poking command structure in Slave HW BFM memory
- Peeking data in slave memory for scoreboarding
- Poking data in slave memory for expected read data movement

Care must be taken while accessing DUT memory as this involves communication between hardware and software world. For performance reasons, we should always try to combine access together. Sometimes we have to combine multiple packets together to achieve the same, but this yields better performance throughput.

There are multiple steps involved while transferring memory contents from hardware to software. Below are the steps involved for transferring data from DUT memory to software memory.

- Create an XlSvMemoryTransactor object pointing to the memory instantiation in the hardware side.
- Pass handles to the components that require the back door memory access.
- Implement methods in the memory module to poke/peek the data into memory
- Write to DUT memory using load method at a given address.
- Read DUT memory buffer at the given address using unload method

```
 class dma_env extends uvm_env;
   `uvm_component_utils(dma_env)
   ...
   XlSvMemoryTransactor XlSvMemoryTransactor_obj[*];
   dma_scoreboard       scoreboard;
   ...
   function void build_phase(uvm_phase phase);
     super.build_phase(phase);
     ...
     XlSvMemoryTransactor_obj[0] =
new("hw_top.u_axi4_slave.memoryTransactor");//AXI_RGN
     XlSvMemoryTransactor_obj[1] =
new("hw_top.u_shell_slave_driver_bfm_0");   //PCE0_RGN
     scoreboard                =
dma_scoreboard::type_id::create("scoreboard",this);
     ...
   endfunction : build_phase

   virtual function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);
      scoreboard.XlSvMemoryTransactor_obj[0] =
XlSvMemoryTransactor_obj[0];
      scoreboard.XlSvMemoryTransactor_obj[1] =
XlSvMemoryTransactor_obj[1];
      ...
   endfunction : connect_phase
   ...
endclass

class dma_scoreboard extends uvm_component ;
     `uvm_component_utils(dma_scoreboard)
   XlSvMemoryTransactor XlSvMemoryTransactor_obj[*];
   ...
   task GetData(int mem_handle, int addr, int len,ref int dout[]);
     int unsigned   data_to_unload[];

     dout = new[len];
     XlSvMemoryTransactor_obj[mem_handle].unload(addr, len,
dout);
   endtask
   ...
endclass
```

```
  module shell_slave_bfm (
 Clock,
 Reset,
…
export "DPI-C" function
XlMemoryTransactorLoad;
  function void XlMemoryTransactorLoad(
          input bit [ADDR_WIDTH-1:0] address,
          input bit
[BACK_DOOR_PAYLOAD_WIDTH-1:0]
payload,
          int unsigned numWords );
      for( int i=0; i<numWords; i++ )
          begin
          mem_hw[address+i]
             = payload[ ((i+1)*DATA_WIDTH)-1 -:
DATA_WIDTH ];
          i = i+1;
          if(i != numWords)
             mem_hw[addressReg+i]
             = payload[ ((i+1)*DATA_WIDTH)-1 -:
DATA_WIDTH ];
          end
  endfunction

export "DPI-C" function
XlMemoryTransactorBulkRead;
  function void XlMemoryTransactorBulkRead(
             input bit [ADDR_WIDTH-1:0] address,
          input bit [ADDR_WIDTH:0] numWords
);
       reg [BACK_DOOR_PAYLOAD_WIDTH-
1:0] payload;
      for( int i=0; i<numWords; i++) begin
          payload[
(((i%PAYLOAD_WORDS)+1)*DATA_WIDTH)
-1 -: DATA_WIDTH ]
             = mem_hw[address+i];

XlMemoryTransactorReturnReadData(payload );
        end
  endfunction
…
endmofule
```

VI. SCOREBOARDING STRATEGIES

In simulation environments, scoreboarding would be done as soon as a packet has been processed by DUT. Doing so in TBA environment causes lot of context switching between HW and SW and lowers simulation throughput. Choosing the scoreboarding strategy greatly decides the simulation throughput. We have described few score boarding strategies implemented in our TBA environments.

A. HW scoreboarding

In this approach, we do scoreboarding on HW side (in Emulator) itself. As soon as a packet is processed by DUT (typically DUT returns completion response), scoreboarding logic in BFM (in Emulator) can start comparing the actual data with expected data. In this approach, there is no data transfer from HW side to SW side. Since the scoreboarding logic runs on hardware, it will consume some clock cycles. For ex: if we have a loop of 100 iterations, we might need 1 cycle to do the scoreboarding for each iteration. While scoreboarding is happening for previously completed packet, DUT can process next packet. Hence, emulator would NOT pause until scoreboarding finishes, which is the case in SW scoreboarding. Hence, this approach gives best throughput.

B. SW scoreboarding

In this approach, as soon as DUT completes processing a packet, response would be sent to SW side. To avoid frequent context switching, we can choose to buffer the responses on HW side and push them to SW in bunches. Note that buffering them on HW side has a penalty of occupying hardware resources. Hence, a reasonable trade off should be made between throughput vs hardware resources.

One more point we need to consider is creating lighter completion packets before sending them to SW side. Typically a packet consists of lot of fields, and all of them may not be necessary for scoreboarding logic. So, in such case, we can create a lighter packet, which holds only valid fields required for scoreboard. This saves memory requirements on both HW and SW sides.

C. Illustration of above techniques with AXI4 example.

Assume, we are verifying memory controller and DDR memory (Usually in GBs). First, I will randomly pick up 2 regions of each 8MB size in entire DDR address space. I always issue all write transfers to write region and read transfers to read region.

Let’s look at read transaction scoreboarding in HW side. Before start of simulation, I will poke known data in to read region (8MB size). Instead of choosing random data for read region, we can use start_pattaren[7:0] and increment_pattern[7:0] to initialize. This allows us to compute expected data easily. Now, we can issue RD transfers with the address randomly chosen in this entire 8MB region. As soon as RD response beats available in HW side, we can compute expected data as follows and check against the received data.

$$expected\ 1st\ byte = (axi4pkt.addr\ -\ region_start_addr) * incr_pattern[7:0]\ +\ start_pattern[7:0]$$

$$expected\ 2nd\ byte = expected\ 1st\ byte + incr_pattern[7:0]$$

Let’s look at scoreboarding of WR transfers on SW side. Usually, axi4 packet contains data array whose size is equal to number of beats in that WR transfer. Hence, transferring data as it is from SW to HW requires more context switches and bigger buffer on HW side. So, instead of random data, we can have start_pattern[7:0] and increment_pattern[7:0] in axi4 WR packet. Now, this lighter packet can be sent to HW side. Once WR completion has been received in HW side, BFM in HW side can create lighter packet holding only ID, ADDR, start_pattern and increment_pattern fields. Now, this lighter packet can be pushed to SW side and pushed in to some FIFO. Instead of scoreboarding as soon as WR is finished, we can choose to wait until entire 8MB WR region is exhaustively written. Then all 8 MB data from DDR can be peeked and pushed to SW side. We can compute expected data for this 8MB of WR data by processing all packets in FIFO. This way we can minimize data transfer between SW and HW size and achieve better throughput.

D. One way optimization

While testbench is running on SW side (scoreboarding or stimulus generation), emulator would be paused, hence throughout would be low. We can make both testbench (in SW) and DUT (in HW) run simultaneously to achieve better throughput by streaming out completion packets from emulator. In this approach, emulator doesn't pause to send out completion packets. Software has to capture that streamed out data and do scoreboarding.

VII. DEBUG TECHNIQUES

A verification engineer spends more than 50% of time debugging the simulation failures. It's very important to adopt proper and efficient debugging techniques. A test can fail because of multiple reasons like a bug in design, bug in test bench, environment issues, and wrong configuration etc. log files and waveforms are the tools which help the verification engineer in root causing the failure.

Sometimes it is required to change the environment and rerun the test to arrive at the conclusion. Waveform dumping for big simulation costs lot of time. In the accelerated test bench where the part of DUT and test bench is in the box/hardware, the waveform dump has to be transferred from the hardware side (buffer in the box) to the software side. The buffer size is limited and has to be emptied to the software side (host machine) to accommodate next samples.

All the above facts slow down the debugging process in the accelerated test benches. Following are few techniques which improves the debugging efficiency.

- Recording SW and HW side activities and replaying from certain time instead of zero simulation time.
- Dumping selective signal in the waveform.
- Dumping the duration of interest only.

One can specify relevant duration (say t1 to t2) to save the HW states and capture the input vectors instead of capturing the input vectors at every time interval. This captured database can be replayed after enabling the monitors using set-register calls and can be used to capture the waveforms. While replaying it uses the captured stimulus.

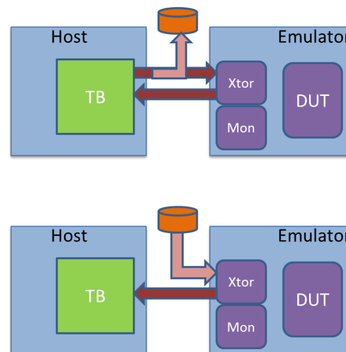

Figure 6: Testebnch backup-replay

In other mode, all transactions from the box/HW are saved to the host. It can be replayed without attaching the box/HW. And after a point of relevance the HW can be restored and test can be run as normal from that point onwards.

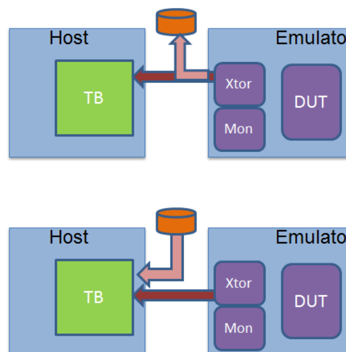

Figure 7: Hardware backup-replay

Example:

Consider a scenario where there is an issue in the test sequence after a long time of simulation (t2 in the figure below). Fixing the stimulus and rerunning takes a long time.

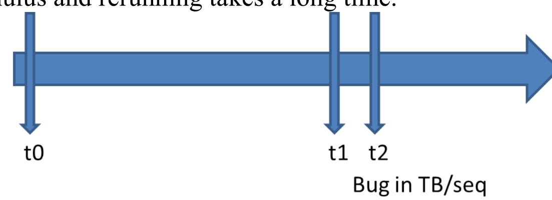

Figure 8: Simulation time without replay

So, in this case capture HW side activity in the host. Change the stimulus, compile the testbench/stimulus and rerun till the time where the change is going to take effect (little earlier t1). Till this point emulator is not needed. The time taken to run till time point will be very less as the there is no context switching between HW side and SW side.

Now attach the emulator and run father simulation to see the effect of the change.

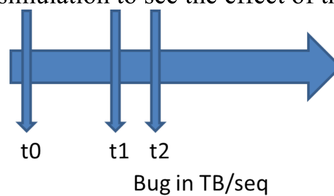

Figure 9: Simulation time with replay

VIII. HOW TO TUNE PERFORMANCE TO ACHIEVE MAXIMUM THROUGHPUT

As discussed earlier, Performance depends on several factors like

- Time spent in software versus time spent in hardware
- Memory access
- RTL being ported to box completely or there are some behavioral constructs

To find out all the factors affecting performance, there are a couple of profiling reports available. Below are the several steps to find out the root-cause of the performance throughput drop.

- Maximum compile time frequency versus actual frequency achieved

```
=========================================================================
Performance and Design Profile
-----------------------------------------------------------------------
--- Maximum HW operating speed (fclk freq): 1316.00 KHz
--- Clocking Mode: Default (2X).

--- HW executed 425111880 evals(or sim-steps) + (38135135 behav cycles) in 1785.08 sec (1781.73 CPU sec)
--- Acceleration speed achieved: 259.51 KHz (259510.76 evals/sec)
```

Above profile report clearly tells that though Box can run upto 1316KHz but it could only run at 259.51KHz. Since we are getting very less throughput, we need to find out where bigger portion of time is consumed

- Time spent in software world versus hardware world

```
--- Profile: (%)
73.47   SW-SIM (Elapsed: 1311.45 sec; CPU: 1308.99 sec)
19.74   HW-EMU (Elapsed: 352.31 sec; CPU: 351.65 sec)
 6.80   HW-MEM (Elapsed: 121.31 sec; CPU: 121.08 sec)

--- SW-SIM : TB, TBCalls, DPICalls, VPI/VHPI/PLI/E/SystemC.
--- HW-EMU : HW evaluations and Synchronization Latency.
--- HW-MEM : Memory Access (read/write) from/to HW.
```

The above profile suggests that the time spent on software is too high as compared to time spent on the hardware. This is something which needs to further analyzed since to get better throughput, time spent on hardware should be as much as possible. This profile tells that very less time spent on memory interactions.

- Synchronizations between hardware and software

```
--- Synchronizations due to TB/DPI/SVA/SysTask calls: 682.

--- Total number of TB/DPI/SVA/SysTask calls: 694.

---------------------------------------------------------------------
          TBCALL statistics: 694 tb-calls
---------------------------------------------------------------------
     587 tb_top.hw_top.u_dcr_if.drive_data    (Type = tb_export:pio)
          Profile: Total time =~ 0.37 sec (634.97 usec/call, input-args=0.31%, exec=99.48%,
output-args=0.20%)
          Loc: (/home/cheetah/team/sunkumar/cheetah/mss_mp/sim/bfm/hw_bfm/dcr_if/dcr_if.sv,
44)
…….
```

The above profile suggests that there are 587 times interaction between hardware and software. This needs to be looked at and reduced if possible, since every interaction consumes some amount of CPU time.

IX. Results and Conclusions

We developed the acceleration environment reusing a significant amount of components/functionality from the corresponding simulation platform. The results are very encouraging.

Below table shows the throughput numbers in the simulation and acceleration environment of the same design.

Data Transfer Size	Simulation wall clock time	Emulator wall clock time	Speed Up (X factor)
1KB- 64KB*100*8	25888sec	118 sec	**219**
64KB-256KB*100*8	46694sec	174 sec	**268**
512KB-1MB *100*8	43637sec	111sec	**393**

The testbench acceleration environment has boosted the simulation speed up-to 300 times faster against that of the core level pure software simulation. The huge throughput gain has enabled us to run longer simulation cycles which might not be possible on the pure simulation platform at all.

Our conclusion is that TBA methodology has great potential in enhancing the verification productivity and thereby quality. The TBA methodology should be deployed for the designs which need long cycle runs and it should complement the simulation platform for such long tests.

Our recommendation is to follow the above suggested guidelines while designing the simulation platform testbench to make it acceleration friendly. This would facilitate the same testbench being able to run on the simulation as well as the acceleration platform. The long tests, gate level simulations and the power-aware simulations are some of the obvious candidates to target on the acceleration which would yield the best results in throughput gain.

X. REFERENCES

[1] SCEMI (Standard co-Emulation Modeling Interface) Reference Manual
[2] Incisive Enterprise Palladium Series with Incisive XE Software datasheet
[3] TBX User guide from Mentor Graphics
[4] https://verificationacademy.com/
[5] http://www.mentor.com/products/fv/emulation-systems/veloce