Getting started on Co-Emulation: Why and How to Transition your Design and UVM Testbench to an Emulator

Jigar Savla

December 31, 2018

Getting started on Co-Emulation: Why and How to Transition your Design and UVM Testbench to an Emulator

Jigar Savla

IEEE MTV18, Austin

Juniper Networks

Sunnyvale, CA, USA

jigar.savla@gatech.edu

ABSTRACT

As we move to more complex and intricate designs, time spent in testing is ever crucial. With several avenues to test our design, we have to pick and choose best ways to optimize the overall time spent on testing.

In our endeavor to move some test benches and designs to Emulation, we learnt several things that could be optimized from TestBench (TB) setup, Design changes to even SVA changes to achieve better simulation performance.

We also identified the kinds of tests and the nature of test benches to run on Emulation that would give the most ROI.

In this paper, we start with an overview and then boil down to some code samples. Then we'll dig into things to be mindful of, in making effective use of the emulator platform.

Table of Contents

Table of Figures

*Getting started on Co-Emulation: Primer on Why and How to Transition
your Design and UVM Testbench to an Emulator- IEEE MTV18*

Table of Tables

Getting started on Co-Emulation: Primer on Why and How to Transition your Design and UVM Testbench to an Emulator- IEEE MTV18

1. Introduction

As we move to more complex and intricate SoC designs, test time is ever crucial. With several avenues to test our design, we have to pick and choose the best ways to optimize the testing and time spent. Emulation stands out as a way to improve your testing time, though the porting process is not seamless.

1.1 Short walkthrough of the paper

In this paper, we start with an overview, the reasons why emulation is awesome. Understand the architecture. Understand the industry standards in this frontier.

Then boil down to some code samples, then things to look out for and finally how to plan effectively.

We'll also cover guidelines to help make your process smooth and rewarding.

We'll come up with an improved Virtual Interface (Vif++) to help us in our porting process. As we know, a virtual interface(Vif) is a shared piece of code between components in an interface.

For purposes of this document, the term Emulator is used interchangeably with any simulator capable of executing RTL or gate-level models, including software HDL simulators. Highest speedup will be for hardware based ones and that'll be our focus for this document.

2. Why Emulation?

To verify complex designs we now have several tools at our disposal such as Simulators, Emulators, SVAs, functional coverage and Formal verification. All of them target different aspects to bring us closer to the ultimate goal of getting bug-free design as fast as possible.

With proper planning and effort, Emulators can help speed up runtimes in the order of 5x-1000x or more compared to CPU based RTL simulation (Level 1 performance in Fig1) .

2.1 Goal: TB Speedup
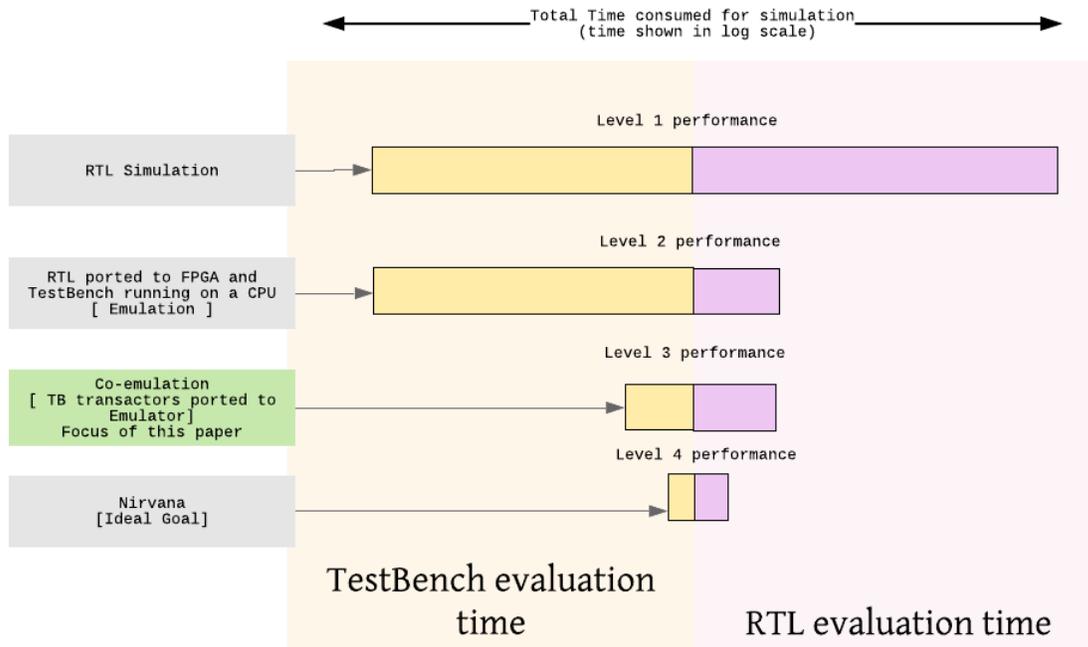
Figure 1: Reaching the TB speedup promise land

In Figure 1, we show the time consumed by TB and RTL at various performance levels and how it compares to the ideal goal the industry is aiming for. This paper discusses optimizations to get closer to the ideal goal.

2.2 Goal: Software / Driver Bringup

An essential part of improving overall product development time is an improved bringup environment for Software prior to actual chip delivery in the lab. Software bringup is anything which needs as close to silicon functionality for developing or testing driver, software and systems.

With Software bringup / driver development, the options available for target design:

OPTIONS	PROS	CONS
CYCLE ACCURATE SIMULATOR	No need to maintain a special code base	Level 1 performance, relatively slow. Overkill sometimes.
SIMPLIFIED / ABSTRACT MODEL	Very fast and can be customized to consumer's (SW / Drv) tastes	Need to maintain a separate a code base
EMULATOR	Fast, no separate code base	Some maintenance needed, but lower than the other options.

Table 1 : Software / Driver Silicon Bringup options

Getting started on Co-Emulation: Primer on Why and How to Transition your Design and UVM Testbench to an Emulator- IEEE MTV18

2.3 Tests which are great candidates to be run on an Emulator

We found that performance, long running tests which have already undergone significant cleanup (high level of testing and running clean for some time) in simulation are great candidates.

- Performance tests: Tests that may need to be run for long periods of time to get an accurate measure of performance.
- Long running tests: Tests which do resource or memory exhaustion, aging (any element in the design which has a timeliness property to it), system reconfiguration tests, hot banking, recycling used objects.
- Reproduced '*wedges*': 'Wedges' are tests where several things have to happen together and can take some time to reach a triggering state. These kind of scenarios are typically hard to hit or find at level 1 performance (ref Figure 1: Reaching the TB speedup promise land).
- Host interface connectivity checked very quickly by being able to quickly run a test that touches every register in the design.
- Infrastructure check test (with IOs) – Tests which check for connectivity within blocks or SoCs, ring test, etc. You can get any custom IOs integrated. Encrypted Verilog is not easily handled, so you might have to create fake behavioral model to replace it.

Remember that in emulators, you generally have support for only two state values. Tests which depend on 'X' and 'Z' values, are better checked using X-prop and Formal solutions.

3. Architecture

To ensure consistency across generations and vendors it's best to follow industry standards. For Design and Verification language, we have SystemVerilog[2]. For DV libraries we have UVM. Fortunately, in Emulation too there's an interface working group: Accellera's SCE-MI2 [1]. Leading Emulation vendors support the SCE-MI2 transactor approach.

Before we dig into the SCE-MI2 standard, lets understand the architecture.

3.1 Architecture: Timed and Untimed sections and DPI-C

We call the untimed sections as *HVL (Hardware Verification Language)* domain and the timed sections as *HDL (Hardware Description Language)* domain. In this section 'time' implies, simulation time: which advances RTL simulation. Most UVM Testbench code sits in the untimed domain. We can think of Testbenches as executing in zero simulation time. Drivers and monitors are the primary exception as they drive data with respect to clock edges.

Designs are obviously timed or clock edge aware. There is even untimed code which sits in modern designs: Concurrent SVAs[8]a and some Functional Coverage.

DPI: As defined in the manual [2]: *"DPI* (Direct Programming Interface) *is an interface between SystemVerilog and a foreign programming language. It consists of two separate layers: the SystemVerilog layer and a foreign language layer. Both sides of DPI are fully isolated. Which programming language is actually used as the foreign language is transparent and irrelevant for the SystemVerilog side of this interface."*

DPI communication is one of the central ideas used to separately execute, yet synchronize the timed (HDL) and the untimed (HVL) domain.

Page 6

Getting started on Co-Emulation: Primer on Why and How to Transition your Design and UVM Testbench to an Emulator- IEEE MTV18

3.2 SCE-MI2 modes of communication

There are three primary modes of SCE-MI2 communication, for more details refer [1] –

1. Function based – Functions (leveraging DPI – C) providing mid level abstraction.
2. Macro based – Message passing interface intended to be used in several use cases and by different groups of users.
3. Pipe based - A transaction pipe is a construct that is accessed via function calls that provides a means for streaming transactions to and from the HDL side

Function based is the easiest to implement.
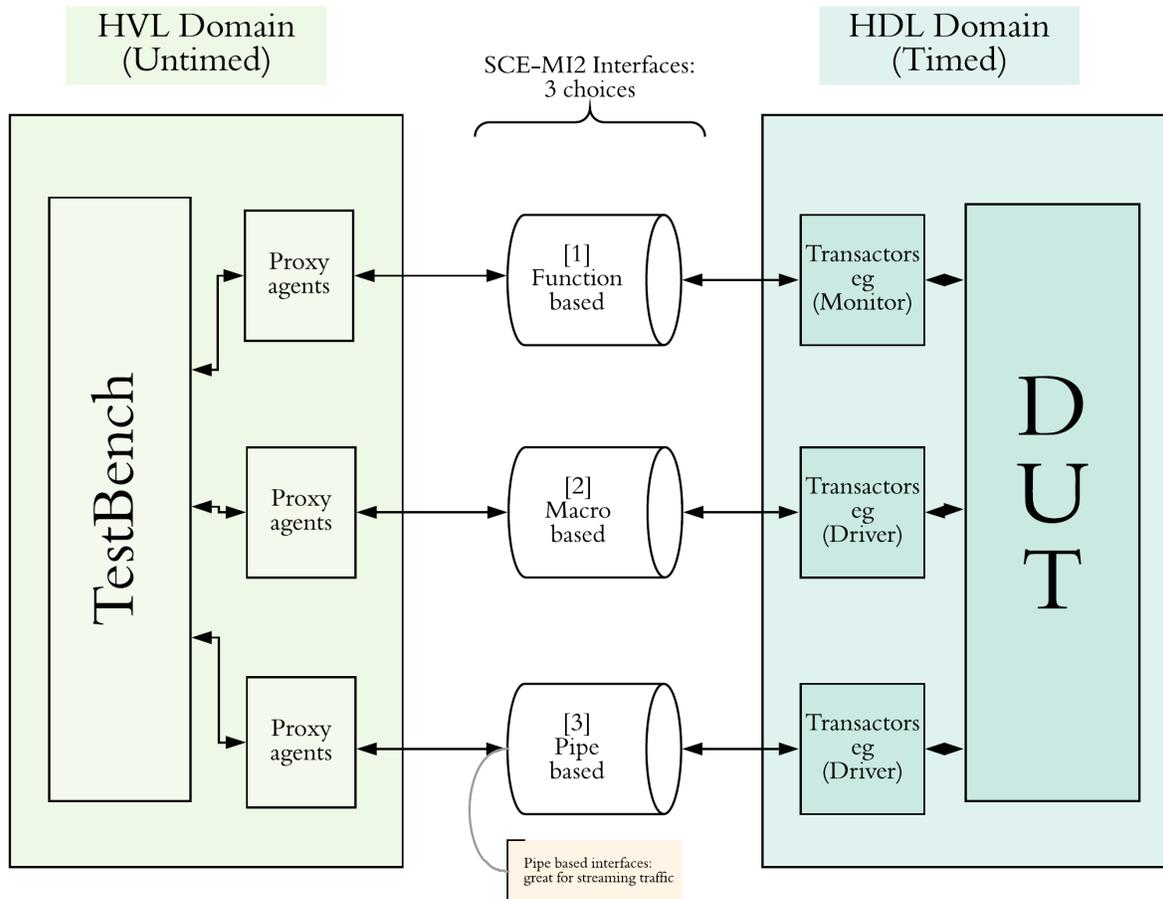
Figure 2: SCE-MI2 Modes of communication

One Source of slowdown may be the communication between timed (HDL) domain and the untimed (HVL) domain.

4. Porting and Optimizations

As long as we ensure the HVL and HDL side transactors are SCE-MI2 compatible, we have won most of the battle. Of course, that's easier said than done, so let's see what it takes.

Emulation gets it's major speedup from the fact that it's running on FPGAs, and other such accelerators mapped to hardware. Therefore, we need to have synthesizable code.

Constructs like : `class`, `constraint`, `intersect`, `throughout`, `solve`, `super`, `within`, `tagged unions`, `rand`, `randc`, `extern` are not synthesizable. This list is non-exhaustive and is only used as a representation.

4.1 Design Porting and optimizations

Design is built for synthesis, making it easier to port. But when you start looking at the overall picture, there might be changes needed or you might have to cut down on parts of the chip.

Eg: Multiple clock domains. Analog logic such as Serdes or PLL's must be replaced with simple digital models.

Since the advent of SVAs and Functional coverage (FCov), we've had code added to Design which is typically not synthesizable. This kind of code would generally be guarded by Synopsys pragmas to avoid compilation errors during synthesis as it's non-synthesizable. There exist syntheziable variants, but that adds mental overhead when writing these SVAs.

SVA and FCov: Emulation vendors have started offering specialized hardware to take care of a sub-set of SVA and FCov constructs. Refer to their reference material to see if all of the constructs used in your design are supported. We can then port nearly the whole RTL code unmodified.

You can always choose to not let this code run on your emulator but then you'll lose out on the test and coverage checking which these functions give.

Eg:

```
module test_design ( input clock,
    ...
    );
always_ff @ (posedge clk) begin : test_block
    // some RTL code
end

// synopsys translate_off
// assert start

property test_property (clk);
...
endproperty: test_property

test_assert : assert property
(test_property(clk)) else
$display("test_assert failed");

// assert end

// fcov start
    // A Functional coverage instantiation here
// fcov end

// synopsys translate_on

endmodule
```

Figure 3: Example code for synopsys translate on off

Guideline: If possible, write SVAs with Synthesis in mind. Emulation vendors also respect the synopsys pragmas, so you'll have to find creative ways to include SVAs and FCovs in Emulation compiles but not in design synthesis.

Eg:

Getting started on Co-Emulation: Primer on Why and How to Transition your Design and UVM Testbench to an Emulator- IEEE MTV18

1. If writing an else statement, can't use $plusargs but have to tie it to a pin which can toggle.
2. Avoid concurrent statements in SVAs[8]a or guard them with another pragma.
3. Instead of doing inline or end of file additions, create a separate file with your SVAs and FCovs, So that you can create file-lists during compilation.
4. Analyze your SVAs and FCovs for vacuous passes. In our experience, we've seen significant slowdown happen due to very loose trigger mechanisms and antecedent code (like triggering at every posedge clk, even though valid condition is seen rarely).
5. To reduce state space issues while dealing with SVAs and FCov, code up some small combo logic (RTL) to set the trigger and counting bits, instead of having it all sit in the property itself. This has helped speed up our runtimes.

4.2 TestBench Porting and Optimizations

Clearly this is the section where most time is going to be spent during the porting process.

Key things to remember:

1. Keep the HVL and HDL sections separate. *Ref: 3.1 3.2 Figure 2: SCE-MI2 Modes of communication*
2. Keep the HVL sections untimed. Most UVM code is naturally untimed. Use events wherever time is needed. We do this to avoid needing clocks in HVL untimed domain. (eg: Incorrect Driver code: Will have ##s instead of @vi.driver_cb)
3. HDL sections to not have any non-synthesizable code.
4. Communications to happen through the standard interface protocols of SCE-MI2, *Ref: Figure 2: SCE-MI2 Modes of communication, 4.2.3*

To keep, HVL and HDL sections separate there are several approaches. Quite a few of them, would involve breaking the Monitor or Driver down into two separate components which can communicate with each other.

4.2.1 Key principle

We take the approach of making small changes in the monitor and driver to make them feel nearly the same. Through this, we'll have less code to maintain. As we know, a virtual interface(Vif) is a shared piece of code between components in an interface. Naturally, we can have the HDL aware code sit in a Vif. We'll have the monitor and driver access functions and tasks as defined in the Vif to toggle or read the actual DUT pins.

You can think of these calls to code within the Vif as DPI-C calls: Define a call in one language, call it from the other [1]. It's also a System-verilog standard [2]

4.2.2 How Vif++ uses SCE-MI2 modes of communication underneath

Lets look at some code samples for the 'function' approach described in Section 3 of this paper. You can consider the function / task *call* as the '*transaction*' and the function / task *itself* as the *transactor*. [1]

We'll start with the interface improvements and then how the modified driver and monitor would look:

4.2.3 Virtual Interface++

As we see in Figure 3, we've added several tasks which are clock aware as well as deal with the actual toggling of the RTL signals.

```
// Interface my_interface: simple ack valid and data        Standard I/f code
interface my_interface (input clk, input reset);
    logic             ack           ;
    logic             vld           ;
    logic[7:0]        data          ; //Single cycle
    parameter INPUT_SKEW=1;
    parameter OUTPUT_SKEW=1;
  clocking drive_from_sender_cb @(posedge clk);
      default input #INPUT_SKEW output #OUTPUT_SKEW;
       input  reset;
       input  ack;
      output  vld;
      output  data;
  endclocking//drive_from_sender_cb
  modport drive_from_sender_port(clocking drive_from_sender_cb);
  clocking drive_to_sender_cb @(posedge clk);
     ...
  endclocking//drive_to_sender_cb
  modport drive_to_sender_port(clocking drive_to_sender_cb);
  clocking monitor_cb @(posedge clk);
     ...
  endclocking//monitor_cb
  modport monitor_port(clocking monitor_cb);

    // interface specific SVAs
    check_sop_eop : assert property (...) else
    $logE("...");
  //---------------------------------------------------------------
  // Additional tasks for driver/monitor
  // -- For emulation purpose.          Emulation support code
  //---------------------------------------------------------------
  task send_trans_to_dut ( input logic vld_i, input logic[7:0] data_i);
    @(drive_from_sender_cb);
    vld     <= vld_i;
    data    <= data_i;
  endtask : send_trans_to_dut

  // generic clock to be used anywhere in driver, monitor
  task wait_one_clk();
    begin
      @(drive_from_sender_cb);
    end
  endtask
  task wait_one_drv_credit_clk();
    begin
      @(drive_from_sender_cb);
    end
  endtask

  task wait_one_seq_clk();
    begin
      @(posedge clk);
    end
  endtask

  task get_monitor_trans(output logic vld_m, output logic [7:0] data_m, output logic ack_m);
    @(monitor_cb);
    if (!reset) begin
      vld_m     = vld;
      data_m   = data;
      ack_m    = ack;
    end

  endtask : get_monitor_trans

endinterface :my_interface
```

Figure 4: Virtual Interface++ code example

4.2.4 Typical Driver

We'll now call the driver related tasks as we defined in the Vif from the driver.

```
┌─────────────────────────┐                    ┌─────────────────────────┐
│      Pre - Emulation     │                    │   Emulation Environment │
│     Environment driver   │                    │     compatible driver   │
└─────────────────────────┘                    └─────────────────────────┘

//========================================================
// Pre - Emulation Environment driver
// Description: This driver will push transactions from a sequencer
// onto the assigned virtual interface.
//
//========================================================
class my_driver extends uvm_driver #(  my_packet );

   // ====================================================
   // Class Variables
   // ====================================================
   ...
   virtual my_interface vif;

   // ====================================================
   // Knob variables
   // ====================================================
   // ====================================================
   // Driver registration macro
   // ====================================================

   // ====================================================
   // Constructor
   // ====================================================

   // Port for incrementing credits from the montior
   // ====================================================
   // Build phase
   // ====================================================

   // ====================================================
   // Connect virtual interfaces
   // ====================================================

   // ====================================================
   // Run phase
   // ====================================================

   task run_phase ( uvm_phase phase );
      ...
      fork
        get_and_drive( );
      join_none

   endtask : run_phase
   task get_and_drive( );
      ...

      forever
      begin
         // Get packets from sequencer
         if (credits_available) seq_item_port.try_next_item( req );
            if ( req != null)
            begin
               // Drive the item
               send_to_dut( req );
               // Communicate item done to the sequencer
               seq_item_port.item_done( );

            end
            else begin
               assign_idle( );
            end
      end
   endtask : get_and_drive
```

┌──┐
│ // Transmit packet to model and DUT   │
│ task send_to_dut(my_packet pkt);   │
│   vif.drive_from_sender_cb.vld   <= 1'b1;   │
│   vif.drive_from_sender_cb.data   <= pkt.data;   │
│   // decrement credits   │
│   // DONT use any ##1 here !   │
│   @(vif.drive_from_sender_cb);   SEND_TO_DUT│
│ endtask : send_to_dut   │
├──┤
│ // Drive idles   │
│   task assign_idle(my_packet pkt = '0);// optional packet to be driven for│
│ idle case   │
│   vif.drive_from_sender_cb.vld   <= 1'b0;   │
│   vif.drive_from_sender_cb.data   <= (pkt != '0)? pkt.data : $urandom();│
│   // DONT use any ##1 here !   │
│   @(vif.drive_from_sender_cb);   │
│   ASSIGN_IDLE│
│ endtask : assign_idle   │
│   // UVM report_phase   │
│ endclass : my_driver   │
└──┘
```

Right side boxes:

```
// Transmit packet to model and
DUT
task send_to_dut( my_packet pkt
);

   // DONT use any ##1 here !
   vif.send_trans_to_dut(1'b1,
pkt.data);

endtask : send_to_dut
```

```
// Drive idles
task assign_idle(my_packet pkt =
'0 );

   logic [7:0] data_to_be_sent =
(pkt != '0)? pkt.data :
$urandom();
   vif.send_trans_to_dut(1'b0,
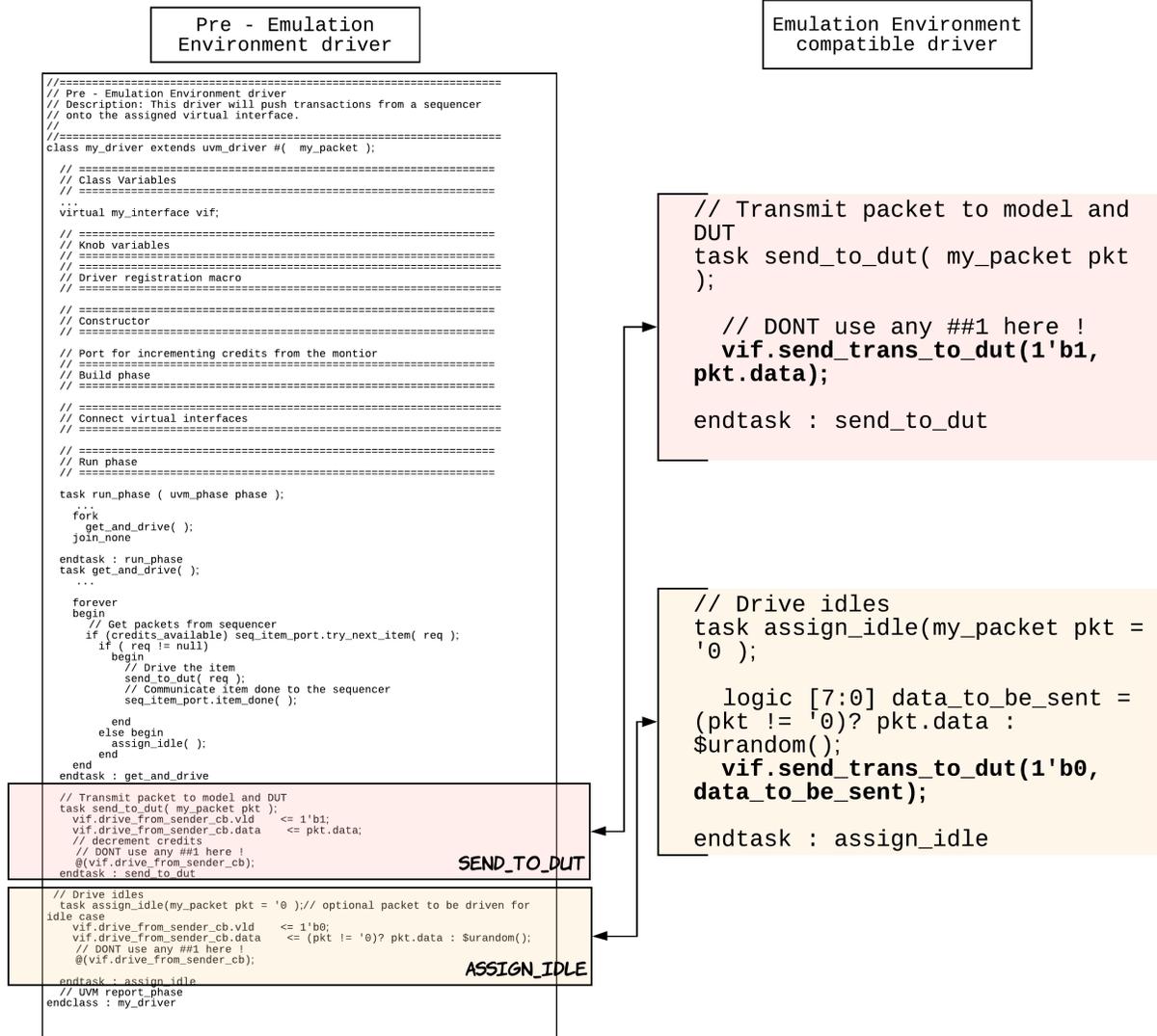data_to_be_sent);

endtask : assign_idle
```

Figure 5: Driver code example

4.2.5 Typical Monitor

We'll now call the monitor related tasks as we defined in the Vif from the monitor.

```
                                    ┌──────────────────┐        ┌──────────────────┐
                                    │   Pre-emulation  │        │     emulation    │
                                    │      monitor     │        │ compatible monitor│
                                    └──────────────────┘        └──────────────────┘
```

```
//====================================================
// Common base monitor
// Description: This monitor will capture transactions for
the scoreboard to parse.  Interface credit checks, and
performance monitor may also be done.
//
//====================================================
class my_monitor  extends uvm_monitor;
    virtual my_interface vif;
    // collected data
    // Count packets and responses collected
    //To set interface
    //new
    ////////////////////////////////////////
    //build
    ////////////////////////////////////////
    //connect
    ////////////////////////////////////////
    //run:
    extern virtual task collect_packet();
    extern virtual task collect_ack_response();
    virtual task run_phase(uvm_phase phase);
        ...
            fork
                collect_packet();
                collect_ack_response();
            join
            // coverage checking code goes here
            `uvm_info(get_name(), "end
run_phase()",UVM_LOW)
            super.run_phase(phase);
    endtask: run_phase
    // UVM report_phase() phase
endclass:my_monitor

// Collect Responses
task my_monitor::collect_ack_response();
    logic ack, vld;
    logic [7:0] data;
    forever begin
        @(vi.monitor_cb);
        if (vi.monitor_cb.bp_ack & ~vi.monitor_cb.reset)
begin
            num_ack_col++;
            bp_ack_port.write(num_ack_col);
        end
    end
    ...
endtask : collect_ack_response

// Collect packets
task my_monitor::collect_packet();
    `uvm_info(get_full_name(), "collect_packet called",
UVM_HIGH)
    packet_collected =
my_packet::type_id::create("packet_collected", this);
    forever begin
        @(vi.monitor_cb);
        if (vi.monitor_cb.bp_valid & ~vi.monitor_cb.reset)
begin
            packet_collected.data = vi.monitor_cb.data;
            sb_port.write(packet_collected);
        end
    end // forever begin
endtask : collect_packet
```

COLLECT_RESPONSES

COLLECT_PACKET

```
forever begin
  vif.get_monitor_trans(.vld_m(vld),
.ack_m(ack), .data_m(data));
  if (ack)
bp_ack_port.write(num_ack_col);
    ...
end
```

```
forever begin
  if (!reset) begin

  vif.get_monitor_trans(.vld_m(vld),
.ack_m(ack),
.data_m(packet_collected.data));
    if (vld) begin

  sb_port.write(packet_collected);
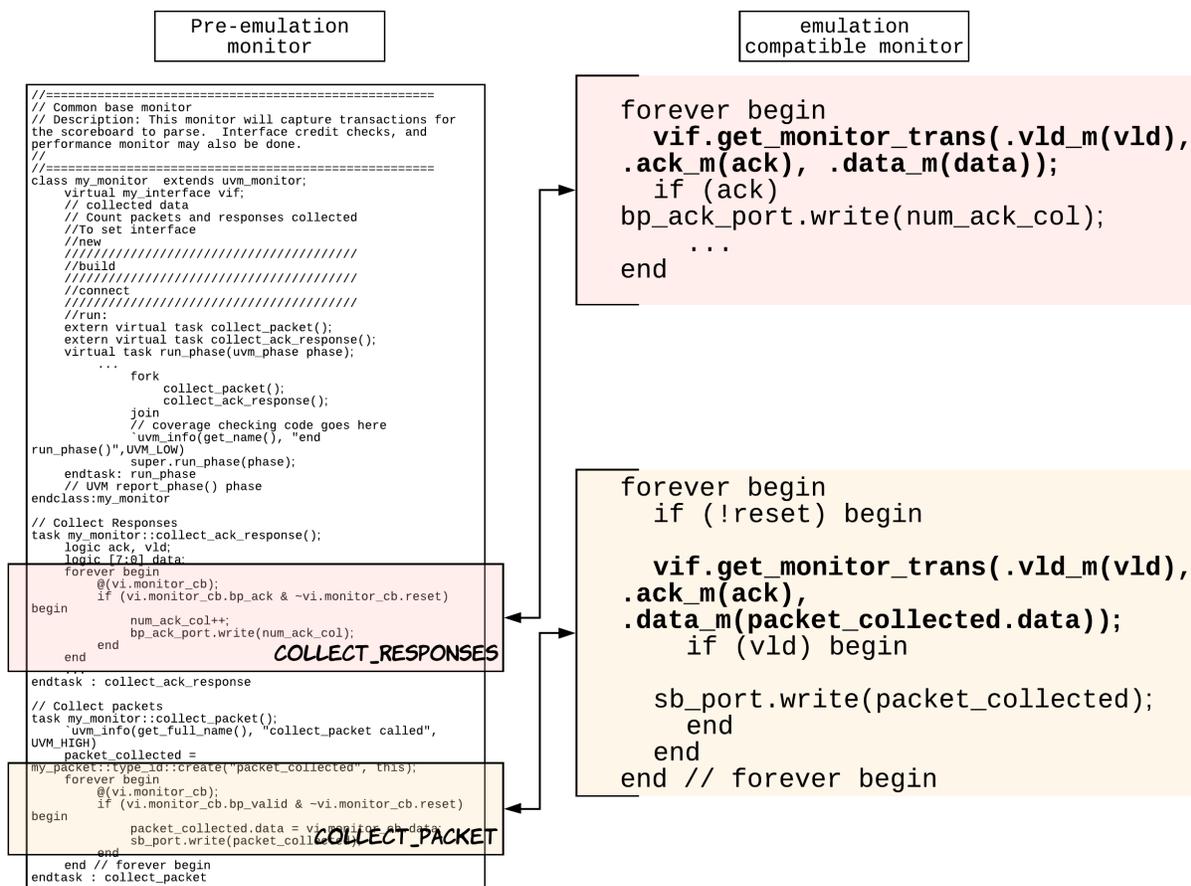    end
  end
end // forever begin
```

Figure 6: Monitor example

4.2.6 Typical Sequence

Generally, simple sequences are not clock aware. But for complex sequences, based on your coding style, there may be clock edges you need to wait for. For that, we'd recommend creating a Vif handle in your sequencer. A sequencer too, normally wouldn't need a Vif, but for the purposes of providing task handles to the sequence, we'll need to add a Vif here.

From the sequence then, it becomes a simple call to `p_sequencer.vif.wait_one_seq_clk()` whenever you need to consume time.

Additionally, as you progress in your emulation effort, you'll see that the interfaces from the HVL domain to HDL domain are not getting utilized as highly. You'd want to look into creating 'aggregate transactions', which are just several transactions bundled and sent over to the HDL domain. Then the HDL domain driver will handle the new bundled transaction.

5. Summary of Coding Guidelines

1. Write SVAs with synthesis in mind. If writing an else statement, can't use $plusargs but have to tie it to a pin which can toggle.

2. Avoid concurrent statements in SVAs[8]a or guard them with another pragma.

3. Instead of doing inline or end of file additions, create a separate file with your SVAs and FCovs, So that you can create file-lists during compilation.

4. Analyze your SVAs and FCovs for vacuous passes. In our experience, we've seen significant slowdown happen due to very loose trigger mechanisms and antecedent code (like triggering at every posedge clk, even though valid condition is seen rarely).

5. To reduce state space issues while dealing with SVAs and FCov, code up some small combo logic (RTL) to set the trigger and counting bits, instead of having it all sit in the property itself. This has helped speed up our runtimes.

6. Don't use ##1 anywhere. Clocking blocks exist for a reason. For the rest, use events.

7. Don't peek / poke into the RTL. Have the Design expose it as a register. Each time a value is sampled, it causes latency, slowing down the overall emulation. Adv: You'll be less sensitive to pipeline / hierarchy changes. Even Software / Driver folks can use it for debug just like DV folks during simulations.

8. In config_db, every time you have a set or a get with "*" as your context or Instance name, you are forcing a search through the whole object tree. With an SoC like environment, can be very expensive. Try to contstrain the search space.

Great thing about TB coding guidelines is that, these guidelines are also highly recommended to get most performance out of even regular simulators. As Amdahl's law states, a system's performance can't be improved beyond it's slowest link. In these co-emulation [1] environments, Testbench communication tends to be the slowest link.

You'll essentially get double the benefit by improving your coding style: Regular simulations would speed up as well as less work when porting to Emulation environments.

6. Miscellaneous things seen in TB / Design

- DPI is deprecated. Use DPI-C instead. There maybe code changes needed in your DPI call [2]

6.1 Mimicing atomic multiple register reads in passive mode:

Unlike simulation where we can do a backdoor read to gather the state of a register set, in an emulation environment multiple registers in a set cannot be read at the same time. We'll have to serialize the reads just like on the real hardware.

By serializing we'll be accessing individual registers within the set at different times. A single register read via the host interface based on a ring topology can consume several (order of 10^4) clocks. This is significant time lapse causing state in the design to change considerably, breaking the intent of time grouping.

Our approach: Create a small synthesizable design to go along with your actual design. This new design module will be setup such that, only one write to a register within this new design causes the design itself to grab all relevant fields and store them in it's own memory.

The emulator can then read out all of this data at a later time, while still mainintaining relative atomicity.

Caveat emptor: This does not work if the test bench needs to react to a register value or needs to check register content on that cycle – so this may require a change in verification strategy.

6.2 Block or system configuration:

We use a similar approach of having a design do our work faster. In this case, you can use the spare registers in your design to store the configuration state and then a new logic can read and program all of them in the order you desire.

7. Planning

7.1 Timeline

Early engagement with any new platform or tool is always beneficial. Based on our experiences, and balancing time and effort with needs, we found that this worked for us.

Emulator land

Simulator land

Project Start

Initial
planning

Block level bring up
complete

Full chip / Sub system
integration starts

Emulation early work
begins

Block level
feature testing
complete

Sub-system testing starts

Full chip testing completes all
high priority items

Great
time
to
loop
in
SW/
Driver
folks

Tape-out

Time to prepare
scripts for Post –
Silicon bringup

Chip arrives in the lab –
Post Silicon bringup begins

Use
Emulator
to
reproduce
'wedges'[9]

Silicon Hand–off

Figure 7: Timeline

*Getting started on Co-Emulation: Primer on Why and How to Transition
your Design and UVM Testbench to an Emulator- IEEE MTV18*

8. Results

Our networking chips don't have widely used benchmarks like SPEC for CPUs, ResNet inference for ML, DL ASICs. What we do have our throughput benchmarks.

Across the board, we saw 2-3x improvement in run times. Our best improvement was on the order of 100x for a full fledged UVM TB. Clearly, we have more room to improve.

For a rather barebones UVM TB, we saw even higher improvement in run times.

Section 2.3 covers our current top tests we like to run on the emulator.

9. Conclusions

Good coding guidelines coupled with some planning right at the beginning can yield astounding results. Although, early functional or brinup tests have limited ROI, performance and long running tests have substantial savings on time.

The level and ability to peek into the RTL during simulations will vary with emulation systems, with some providing the same view, while others require planning for the parts of your design you wish to inspect. However, the raw throughput increase should more than compensate for the extra effort.

10. Acknowledgements

11. References

[1] Standard Co-Emulation Modeling Interface (SCE-MI)   Reference Manual, ver. 2.4, Accellera Interfaces Technical Committee, November, 2016.

[2] IEEE 1800 SystemVerilog: http://www.systemverilog.org/.

[3] Understanding the Accellera SCE-MI Transaction Pipes – Stickley et al - IEEE Design & Test 2012

[4] IEEE 802.3 Ethernet Standard: http://www.ieee802.org/3/.

[5] OSCI TLM-2.0 Standard: http://www.systemc.org/downloads/standards/tlm20.

[6] Yikes! Why is My SystemVerilog Testbench So Slooooow? - Kampf et al – DVCon 2012

[7] Are OVM & UVM Macros Evil? A Cost-Benefit Analysis - Erickson, Adam - DVCon 2011

[8] Concurrent SVA example:

      a.   Concur_sva_example : assert property (vld |=> ack);

[9] 'Wedges' are tests where several things have to happen together and can take some time to reach a triggering state. These kind of scenarios are typically hard to hit or find at level 1 performance (ref Figure 1: Reaching the TB speedup promise land).