Improving simulation performance at SoC/Subsystem level using LITE environment approach

Avni Patel, Data Center Group, Intel, Bangalore, India (avni.n.patel@intel.com)

Heena Mankad, Data Center Group, Intel, Bangalore, India (heena.mankad@intel.com)

***Abstract*—Increasing hardware design complexity has resulted in significant challenges for hardware design verification. Verification Challenges increases exponentially as the effort moves from IP to SubSystem to SoC level. Subsystem/SoC verification involves validating of end to end data paths, inter IP communications, performance analysis, quality of service and other system level scenarios. Multiple IP level verification environments are integrated to build Subsystem/SoC verification environment. Time taken to simulate system level scenarios in such complex environment degrades the simulation performance and hence impacts overall verification convergence. In this paper, we propose a smart approach called "LITE ENV" that helps to reduce simulation time taken at SoC/Subsystem level enabling verification team to focus more on debugging, bug hunting and fixing complex SoC bugs rather than waiting for hour's long simulation to complete.**

***Keywords*— SubSystem, SoC, LITE ENV, Simulation**

I. INTRODUCTION

It is a well known fact that Verification takes the largest amount of time in chip designing by taking around 70% of total time. Verification is done at various levels i.e. IP level for individual design blocks, SubSystem level for major chip clusters and SoC level for full chip. As the design become more complex and integrated at higher level, the verification becomes challenging. At Subsystem/Soc level engineers have to learn details of different IPs, plan system level scenarios, integrate IP level environments and create complex test bench/test cases. These tests runs very long as they configure and cover various IPs inside. The simulation performance degradation and increased memory footprint due to complex Subsystem/SoC VE further increases the test run time. To address this issue, there is need of a solution which can help to improve simulation performance in existing verification environment without spending much effort.

We aim to propose a Verification framework to solve the above described problem. Our approach is based on dynamically restructuring the Verification environment to its lighter version. We propose two such LITE ENV approaches which can be used individually or combined. This LITE version significantly helps to reduce the Subsystem/SoC test run time. This aids verification team to run more and heavy traffic scenarios in lesser time and accelerates the verification closure process.

In the next sections of the paper, we will gradually walk through traditional SubSystem/SoC level verification environment, challenges faced and our solutions for this. We will describe both solutions in detail along with real results we saw with them.

II. SUBSYSTEM VERIFICATION ENVIRONMENT

Similar to the way small IP designs integrate to form cluster designs and they further integrate to form the overall SoC design; IP level verification environments are integrated to build SubSystem/SoC level verification environment. Subsystem/SoC level verification environment integrate multiple IP level verification environments. In a typical networking design, there are many such IPs that form the overall system. Each IP have multiple interfaces for data, credits, sideband, reset, timer, interrupts, etc. The verification environment for these IPs will have BFM/VIP sitting at these interfaces. When all these IP verification environments integrate, all these active VIPs become part of the overall SubSystem/SoC verification environment; hence making is too bulky.

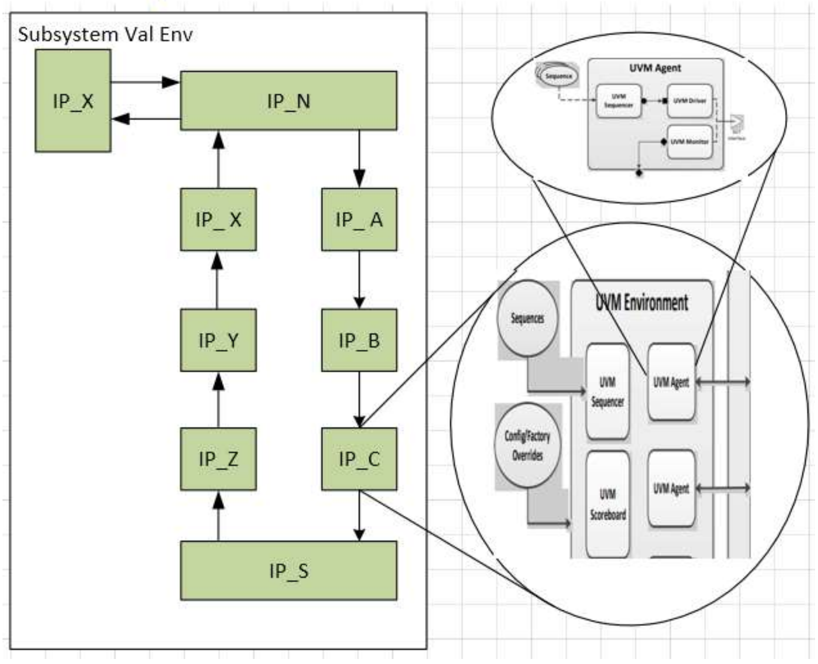


Figure 1. Networking Design SubSystem VE

Figure 1 shows a typical Networking Design SubSystem Verification Environment with an IP environment zoomed in. As seen, each is a UVM based environment with various BFMs, monitors, scoreboards, reference models, etc connected to the next IP. Figure 1. Shows a typical Networking Design Verification Environment.

We have implemented the LITE ENV approach at SubSystem level. Hence, will continue using this term going forward in the paper.

III. LITE ENV APPROACHES

In Networking Designs, the Subsystem verification scenarios are targeted to validate end to end data paths. These scenarios spawns from sanity tests for receive and transmit paths to Quality of Service, Performance and Stress scenarios. There are scenarios where IPs of a particular datapath are exercised and other IPs are silent. There are also some sideband IPs that get exercised only in few targeted scenarios. The sanity testing initially eliminates the basic bugs and the later performance and QoS tests targets validating architectural pipelines. Before the later phase is reached, all basic bugs are already dealt with. The LITE ENV approaches described in later sections leverages on above Subsystem scenario targets.

A. LITE ENV APPROACH # I : Build What You Need

The scenarios targeted for receive path do not exercise transmit path and vice versa. Also, data might not pass from some sideband Offload IPs. Hence, the verification environment of these IPs are not needed to be created. For example, as shown in Figure 2 while running Tx traffic, Rx IP environments (shown in Red color) are not required. So, we can remove them while running Tx scenario and reduce simulation time and memory consumed by these environment components.

Further improvement in simulation performance can be achieved by turning off the clocks to the RTL design for the blocks that are not exercised. Once reset de-asserts, you should wait for few clock cycles to ensure reset propagates into the design and then turn off the clocks.

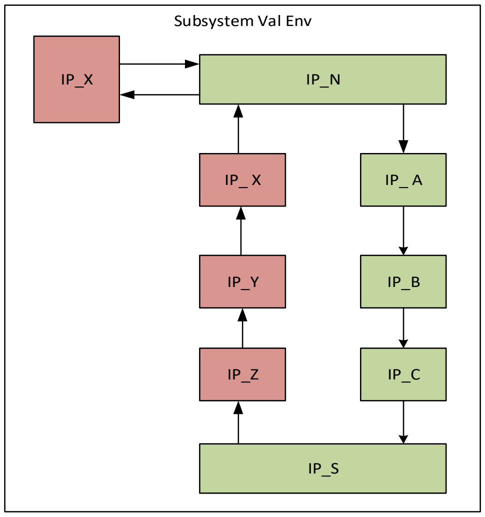


Figure 2. Build What You Need

B. LITE ENV APPROACH # II : Turn On What You Need

Subsystem verification environment contains multiple IP environments and each of them contains multiple monitors/scoreboards/checkers and reference models. These all adds to simulation time require to run any scenario. Subsystem verification environment have end to end scoreboard to validate end to end data integrity which reduces dependency on individual IP level scoreboards.

There are scenarios like Performance and Quality of Service where we need to run good amount of traffic and focus is not on verifying basic traffic flow. Bugs from normal traffic flow are already flushed off when we start performance or quality of service scenarios. Hence, the dependency of individual IP level monitors, checker and scoreboards reduces further. Hence, eliminating such active components from simulation gives comparative improvement in simulation performance. Results are achieved faster and hence debugged early. If needed, we can rerun with full environment. As shown in Figure 3 Monitors/Checker/Scoreboards for IPs in red color circle are disabled and the ones at the boundary IPs are only enabled as shown with green circles. Peripheral IP environments are needed to have active monitors to feed the Subsystem end to end scoreboard.

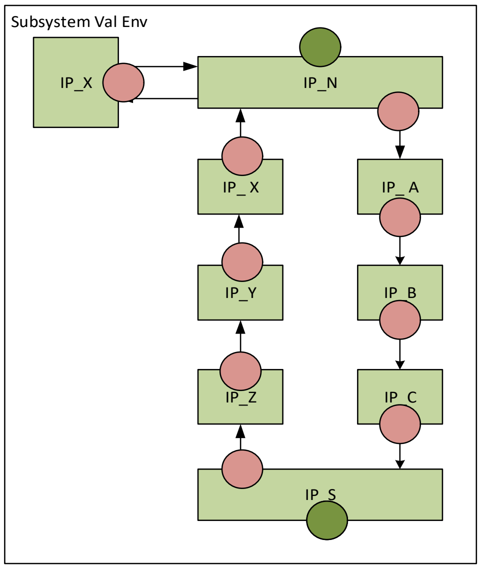


Figure 3. Turn On You Need

C. FLOWCHART : LITE ENV APPROACHES

The LITE ENV approaches are explained in figure 4 using a flow chart. Based on the type of traffic you need to exercise, appropriate approach is to be selected. The scenario might be unidirectional exercising only Tx path or Rx path or it might be bidirectional and exercise all the IPs. You can follow the flow chart and select appropriate approach.

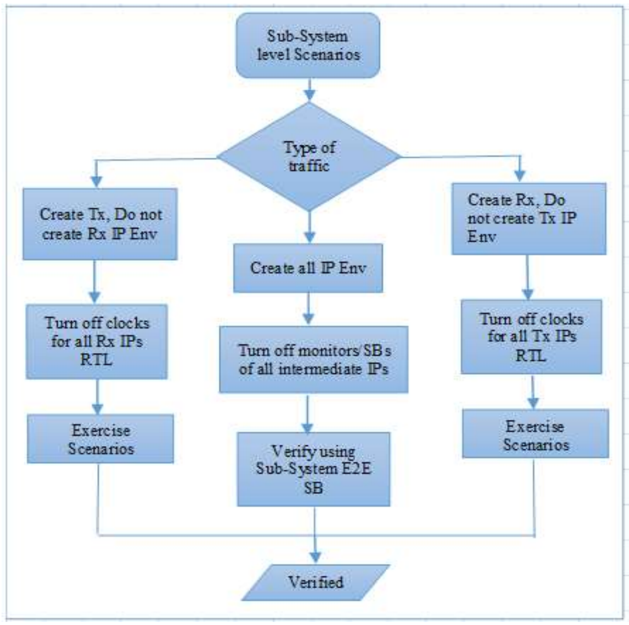

Sub-System level Scenarios
Type of traffic
Create Tx, Do not create Rx IP Env
Create all IP Env
Create Rx, Do not create Tx IP Env
Turn off clocks for all Rx IPs RTL
Turn off monitors/SBs of all intermediate IPs
Turn off clocks for all Tx IPs RTL
Exercise Scenarios
Verify using Sub-System E2E SB
Exercise Scenarios
Verified

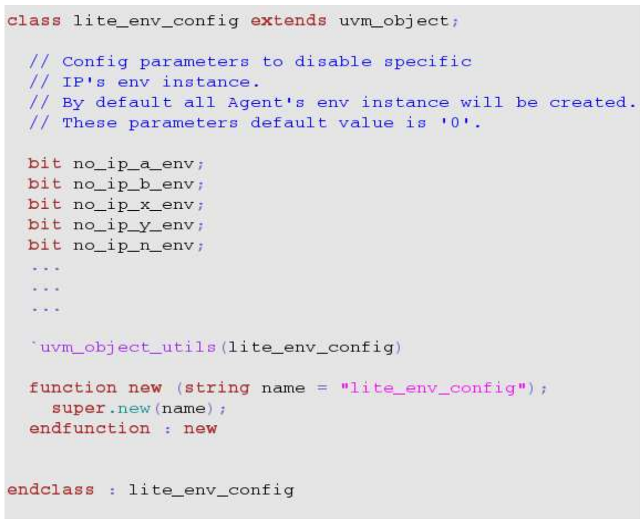
Figure 4. Flow chart of LITE ENV approaches

IV. METHODS TO IMPLEMENT LITE ENV

A. Approach #1 Implementation : Build What You Need

As per LITE ENV approach #1, we will remove instances of IPs environment which are not in use for specific scenarios. To implement the same:

1. Add lite_env_cfg class (Figure 4) and provide configuration parameters which can be set by user to remove different IP's environment instances.
2. Create object of this config class in base test (Figure 5) and pass its handle to subsystem environment (Figure 6). Provide an API in base test which user can use to override config parameter for each IP's environment depending on test requirement.



```
class lite_env_config extends uvm_object;

  // Config parameters to disable specific
  // IP's env instance.
  // By default all Agent's env instance will be created.
  // These parameters default value is '0'.

  bit no_ip_a_env;
  bit no_ip_b_env;
  bit no_ip_x_env;
  bit no_ip_y_env;
  bit no_ip_n_env;
  ...
  ...
  ...

  `uvm_object_utils(lite_env_config)

  function new (string name = "lite_env_config");
    super.new(name);
  endfunction : new


endclass : lite_env_config
```

Figure 4. LITE ENV Config Class Example

```
class temp_base_test extends uvm_test;

  subsystem_env env_ptr;
  lite_env_config lite_env_cfg_obj;

  `uvm_component_utils(temp_base_test)

   function new(string name = "", uvm_component parent=null);
       super.new(name, parent);
   endfunction : new

   function void build_phase();
     // Create Subsystem environment
     env_ptr = subsystem_env::type_id::create("subsystem_env",this);
     // Create lite environment config
     lite_env_cfg_obj = lite_env_config::type_id::create("lite_env_cfg_obj");

     update_lite_env_cfg();
     uvm_config_object::set(this,"*", "lite_env_config", lite_env_cfg_obj);

   endfunction : build_phase

   virtual function update_lite_env_cfg();
     // Update lite env parameters by overriding this function.
     // Example :
     // no_ip_a_env = 1'b1;
   endfunction : update_lite_env_cfg

endclass : temp_base_test
```

Figure 5. Test Case Base Class Example

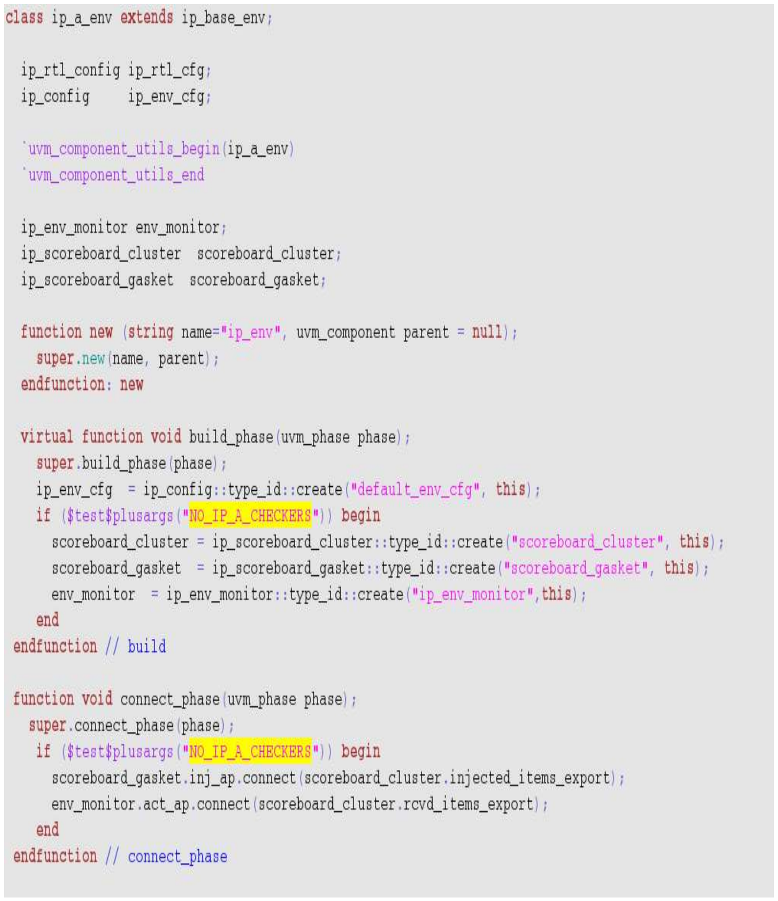
```
class subsystem_env extends uvm_env;

  `uvm_component_utils(subsystem_env);

  lite_env_config lite_env_cfg_obj;
  ip_a_env        ip_a_env_ptr;
  ip_b_env        ip_b_env_ptr;

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(uvm_config_object::get(this, "","lite_env_config",lite_env_cfg_obj)) begin
        assert($cast(lite_env_cfg,lite_env_cfg_obj));
    end
    else begin
      `uvm_info(get_type_name(), $psprintf("Taking default value for LITE ENV CFG"), UVM_LOW)
      lite_env_cfg_obj = lite_env_config::type_id::create("lite_env_cfg");
    end
    if(lite_env_cfg_obj.no_ip_a_env == 1'b1)
      ip_a_env_ptr = ip_a_env::type_id::create("ip_a_env_ptr");

    if(lite_env_cfg_obj.no_ip_b_env == 1'b1)
      ip_b_env_ptr = ip_b_env::type_id::create("ip_b_env_ptr");
    ...
    ...
    ...
  endfunction : build_phase

endclass : subsystem_env
```

Figure 6. SubSystem VE Example

B. *Approach #2 Implementation : Turn On What You Need*

To remove unused checkers of any intermediate IPs, $testplusargs can be provided by each IP environment. When this argument will be passed by sub system environment, IP should not create monitors, SB and GM instances inside their environment. For example, at Sub-System level, let's say in performance test where we are interested in performance and latency checks, we do not need checkers enabled from different intermediate IPs. Here, tester will be only interested in verifying total number of packets and performance monitor/checkers from IPs may be re-used.

In such cases, we can turn off checkers from agents those are not required by passing the $testplusargs at Sub-System level like "+NO_IP1_CHECKERS", "+NO_IP2_CHECKERS" etc. The difference in this approach and removal of complete agent's environment (approach-1) is that, here we can still reuse any other components, config class and sequences of that IP.



```
class ip_a_env extends ip_base_env;

  ip_rtl_config ip_rtl_cfg;
  ip_config     ip_env_cfg;

  `uvm_component_utils_begin(ip_a_env)
  `uvm_component_utils_end

  ip_env_monitor env_monitor;
  ip_scoreboard_cluster  scoreboard_cluster;
  ip_scoreboard_gasket  scoreboard_gasket;

  function new (string name="ip_env", uvm_component parent = null);
    super.new(name, parent);
  endfunction: new

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    ip_env_cfg  = ip_config::type_id::create("default_env_cfg", this);
    if ($test$plusargs("NO_IP_A_CHECKERS")) begin
      scoreboard_cluster = ip_scoreboard_cluster::type_id::create("scoreboard_cluster", this);
      scoreboard_gasket  = ip_scoreboard_gasket::type_id::create("scoreboard_gasket", this);
      env_monitor  = ip_env_monitor::type_id::create("ip_env_monitor",this);
    end
 endfunction // build

 function void connect_phase(uvm_phase phase);
   super.connect_phase(phase);
    if ($test$plusargs("NO_IP_A_CHECKERS")) begin
      scoreboard_gasket.inj_ap.connect(scoreboard_cluster.injected_items_export);
      env_monitor.act_ap.connect(scoreboard_cluster.rcvd_items_export);
    end
 endfunction // connect_phase
```

Figure 7. IP VE supporting LITE ENV APP #2 Example

V. RESULTS

We have implemented above approaches in our Subsystem Verification Environment that has more than 12 IPs. We have separate tests for Tx and Rx. The improvement in Simulation Performance we get for a single directional traffic test using LITE ENV Approach#1 is shown in bar chart at figure(8). As seen the simulation time taken by the test reduced from 3331us to 2951us giving improvement of 11.5%.

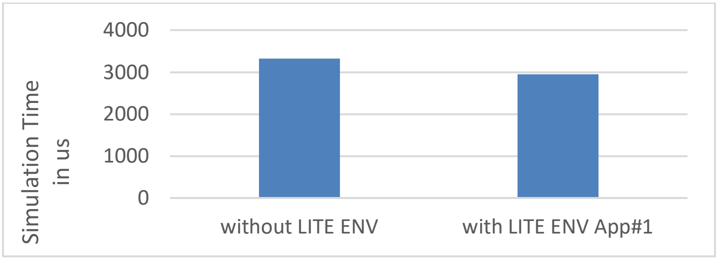


Figure 8. Simulation Performance with LITE ENV App#1

Bar chart at figure (9) shows the improvement in Simulation Performance we achieved for a test that has bidirectional traffic using LITE ENV Approach#2. We disabled the monitors/checkers/scoreboards at the intermediate IPs. As seen the simulation time taken by the test reduced from 3231us to 2074us giving improvement of 35%.

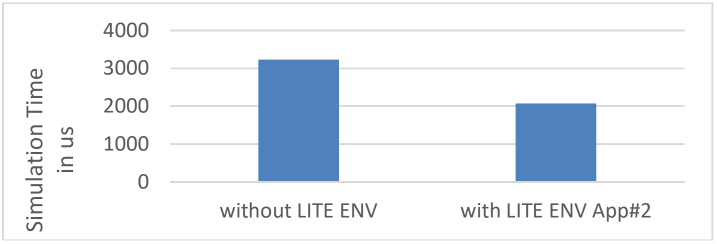


Figure 9. Simulation Performance with LITE ENV App#2

VI. CONCLUSION AND FUTURE WORK

Having worked on Subsystem level verification, we understood the challenges and hence explored ways to mitigate the problem with Simulation performance at this level. The LITE ENV approaches is easily implementable with collaborative work with the IP teams. This approach can be implemented for any designs that has various data paths to be exercised. Based on the test scenario, approach 1 or 2 can be applied individually or combined. With increase in simulation performance for heavy traffic scenarios, the debug cycle started early helping timely verification signoff.

We are exploring more approaches to improve the simulation performance further and its quick integration into the environment. We are also looking into implementing a standard way of integrating it into the verification methodology so future projects can leverage the benefits.

ACKNOWLEDGMENT

This work is done with valuable suggestions and feedbacks from the FrontEnd Verification team at Intel Corporation. We are thankful to the technical team of all IPs for supporting us implementing the approaches.

REFERENCES

[1] System Verilog 3.1a Language Reference Manual
[2] https://www.accellera.org/images//downloads/standards/uvm/uvm_users_guide_1.2.pdf
[3] ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies by Ashok B. M
[4] High-Level Verification: Methods and Tools for Verification of SystemLevel Designs by Sudipta Kundu, Sorin Lerner, Rajesh K. Gupta