A Unified Testbench Architecture Solution for Verifying Variants of the PLL IP

Deepa Ananthanarayanan, Malathi Chikkanna
AMD India Pvt Ltd,
Bangalore, India

***Abstract*- Universal verification methodology (UVM) has managed to standardize testbench development in the industry. UVM offers a coverage driven verification (CDV) environment that is reusable, scalable, and configurable. For verifying multiple variants of an IP, the typical approach would be to create identical verification environments with many of the common components replicated in each testbench. This may seem like a viable solution, but it comes with the overhead of creating and maintaining separate testbenches.**

The authors of this paper present a case where UVM was adopted to create a single plug-and-play verification platform that can be configured and re-used for different design variants of the phase locked loop (PLL). Most of the verification components are retained while scaling the verification suite for any additional variant of the PLL, leading to a drastic reduction in verification bring-up time for the new PLL.

***Keywords* - Coverage Driven Verification (CDV), Constrained Random Verification (CRV), Intellectual Property (IP), Phase Locked Loop (PLL) , Universal Verification Methodology (UVM), IP, Phase Locked Loop (PLL).**

I. INTRODUCTION

Thorough verification is the key to the success of any design. Developing effective stimulus for sweeping the gamut of functionality and automating response checking are the crux of verification. A robust testbench architecture built within the framework of a well-structured methodology, such as universal verification methodology (UVM), helps achieve verification effectiveness.

One of the main challenges of verification activities is the time spent on debugging testbench inadequacies when scaling or reusing testbench components. We seek to make improvements to this challenge by proposing a generic solution for building a highly scalable and reusable testbench architecture with just a one-time effort spent on constructing the base template, and minimal time spent on extending, customizing, or reusing the components for various use cases of the testbench.

II. THE DESIGN UNDER TEST

Phase locked loops (PLLs) are the heart of any SoC design, with applications ranging from frequency synthesis to clock skew cancellation and clock data recovery. The PLL in the design is based on a digital architecture, which offers area and power saving for fine line width, low voltage technology nodes. The core of the design is the feedback loop consisting of the time to digital converter (TDC), digital loop filter (DLF), digitally controlled oscillator (DCO), and the feedback divider as seen in Fig 1. Such an architecture is typically used for the frequency synthesis application, such as high-speed clock generation.

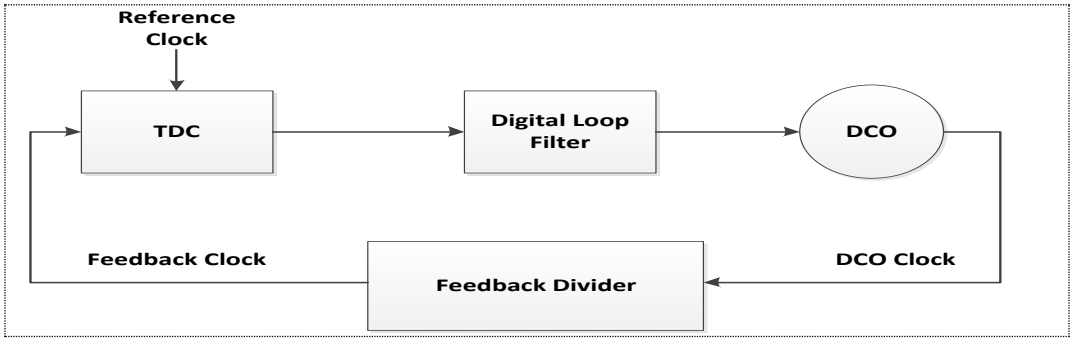

Figure 1. Digital PLL Architecture.

The design under test (DUT) is a digital phase locked loop (PLL) IP for frequency synthesis, with frequency application specific variants. The digital PLL IP is a highly configurable design with programmable options for the PLL bandwidth, feedback loop stability, and divider settings (i.e., reference clock, feedback path and post divider). Application specific PLL based clock generation IPs may implement digital blocks, in addition to the core design that typically contains:

1. Protocol specific interfaces to consuming blocks
2. Register space for configuring PLL settings
3. Design for test (DFT), scan implementation
4. Level shifters and isolation cells
5. Dividers on the reference clock or the DCO clock for more frequency options, etc.

III. PRIOR WORK – PLL VERIFICATION

In prior programs, the PLL verification was directed in nature. Each PLL variant had a unique testbench and a designated person to develop, maintain, and verify the PLLs. Fig 2 shows a standalone verilog based verification infrastructure which was developed for each of the PLL IPs. This consists of stimulus generator, checker and testcases. Digital stimulus generator drives both random and directed stimulus to the DUT. Digital Checker block implements always on checks and task based checks. Verilog test cases were developed for specific verification scenarios with tasks and functions for driving digital stimulus at the DUT (pins/interface).

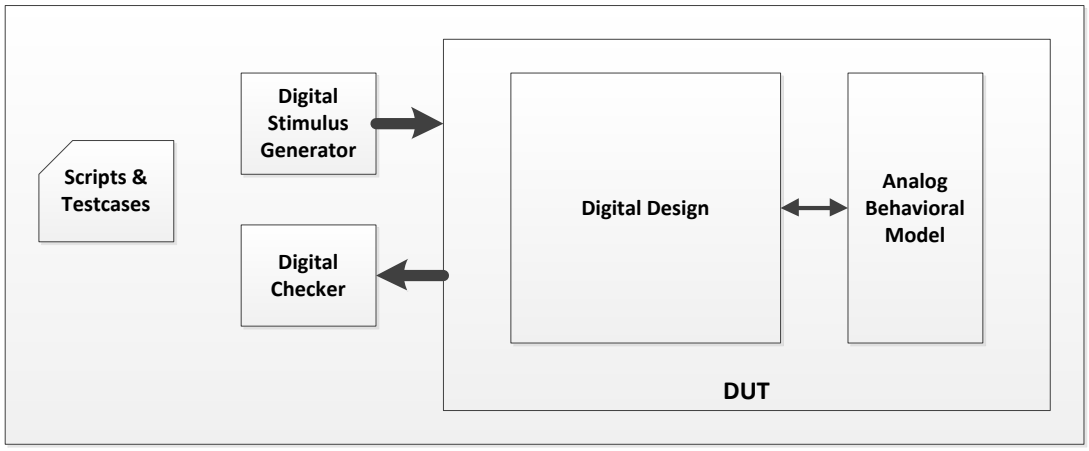

Figure 2. Verilog based testbench.

IV. VERIFICATION CHALLENGES IN PRIOR WORK

A. Testbench Reusability

As we know, in an SoC, there are multiple variants of a PLL and each one requires standalone verification setup for thorough testing. This poses a huge verification challenge in terms of building and maintaining multiple testbench setups. With the verilog-based verification methodology, building a reusable testbench becomes a huge challenge.

Attempts were made towards creating common verilog tasks and functions for implementing stimulus drivers and automated checkers. These tasks/functions were then reused across the individual PLL testbenches. However any addition or change to the input protocol or output check meant editing the original verilog task/function and the associated calls to them.

B. Lack of Constrained Random Verification

In addition to the reusability challenge, verilog based testing falls short of the need for constrained random verification (CRV) of a PLL design with a lot of programmable use cases. Frequency testing, a subset of the complete PLL functional verification alone involves checking for the PLL output frequency for a large combination of:

1. Reference clock frequency and associated divider settings
2. Feedback divider settings: integer or fractional
3. Post divider settings to further divide down the PLL output clock

4. Bandwidth and feedback loop settings

Overall functional verification completeness of the PLL IP requires not only signing off on all supported frequencies, but on all functional features such as:

1. Low power mode support
2. Interface specific protocol compliance when programming the PLL
3. Power Aware Verification: Level Shifter and Firewalling checks
4. Calibration algorithm

To address constrained random verification, the previous work involved developing a SystemVerilog wrapper around the verilog testbench for implementing transaction level randomization and for defining cover points to capture functional coverage of the PLL DUT.

SystemVerilog helped address CRV; however scalability of the testbench remained a challenge.

V. PROPOSED SOLUTION

The universal verification methodology using SystemVerilog constructs sets the stage for building reusable and scalable testbench architecture with coverage driven verification (CDV) for the design under test (DUT). CDV combines automatic test generation, self-checking testbenches, and coverage metrics to significantly reduce the time we spent while verifying the multiple variants of PLL.

Fig 3 depicts a typical UVM based testbench architecture comprising universal verification components (UVC or Agent) that includes: Sequencer, Driver, Monitor, Scoreboard, Coverage collector etc. With four variants of the PLL to verify, it was ideal to create a UVM testbench template as shown in Fig 3 and replicate it with customization for each of the PLL variants.

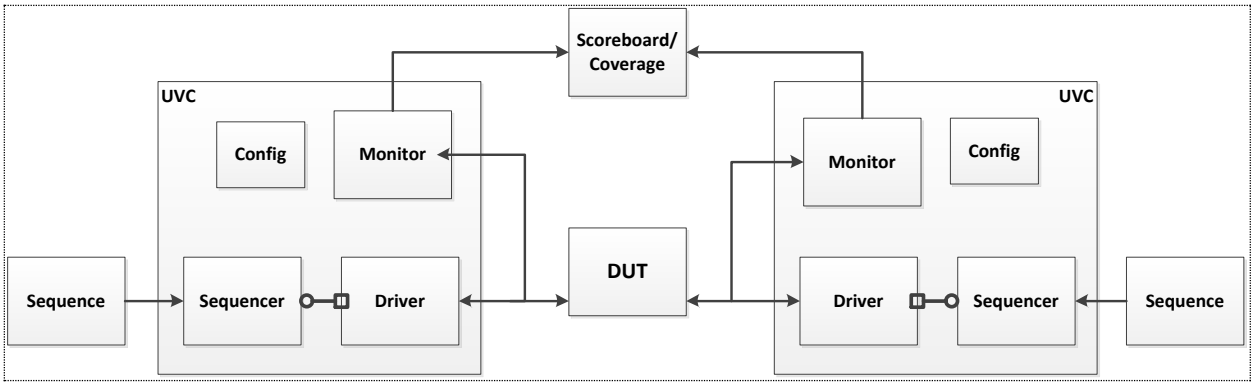

Figure 3. Typical UVM based verification environment.

In our approach, the need for duplicating and maintaining multiple testbenches was eliminated by developing a configurable UVM testbench as shown in Fig. 4. A single comprehensive UVM-based verification environment was developed with the flexibility to append unique verification methods on a case-by-case basis. The agent consists of the PLL base class driver (pll_base_driver) , pll sequencer ,monitor (pll_base_monitor) and configuration object . For a given PLL variant, which is determined by the configuration parameter, the base class components are overridden by the corresponding derived components (pll_type2_driver, pll_type2_monitor, etc) and the remaining components are re-used. This resulted in considerable reduction in the verification bring up time.

We could thoroughly verify our design by changing testbench parameters in the configuration object or changing the randomization seed by adopting constrained-random testing. This enabled us to devote effort into writing time-consuming, directed tests for scenarios that were difficult to reach randomly. Coverage monitors were added to the environment to measure the progress and identify non-exercised functionality. A common sequence library, with a comprehensive list of sequences, was used across the variants of the PLL IP. SystemVerilog assertion methodology was used for developing extensive temporal checks of the PLLs.

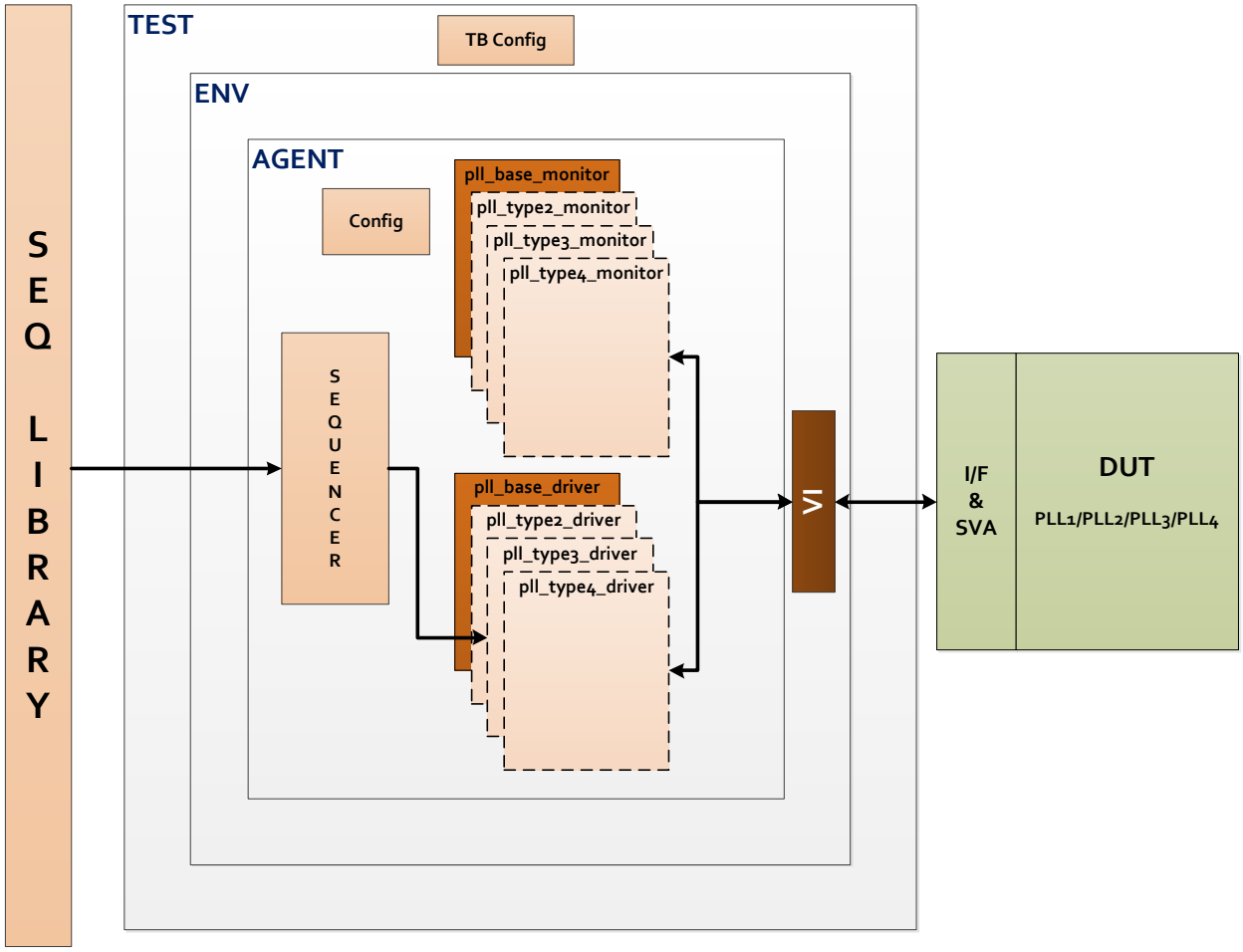

Figure 4. Common testbench architecture for PLLs.

The development started with identifying and classifying common and unique features to these PLL variants. A SystemVerilog based UVM template was created with:

- A PLL agent with base components: driver, monitor and a common sequencer.
- An agent specific configuration object for customizing the verification components.
- Virtual interface for connecting the verification environment to the DUT interface.
- Top level module with instance of the DUT and the corresponding interface.
- Common sequence library.

Fig 5 shows the PLL agent composed of base components such as: driver (pll_base_driver), monitor (pll_base_monitor), configuration object (pll_cfg) and common sequencer. Depending on the configuration object parameter “is_active”, the agent is either ACTIVE (sequencer, driver, and monitor) or PASSIVE (monitor).

```
class pll_agent extends uvm_agent;

   `uvm_component_utils(pll_agent)

   // Driver, Monitor and Sequencer
   pll_sequencer mast_sqrh;
   pll_base_driver mast_drvh;
   pll_base_monitor mast_monh;

   // Configuration objects
   pll_cfg      m_pll_cfg;
   ...

endclass: pll_agent

   function void pll_agent:: build_phase(uvm_phase phase);
      super.build_phase(phase);
        mast_monh = pll_base_monitor::type_id::create("mast_monh", this);
      if (m_pll_cfg.is_active == UVM_ACTIVE) begin
         mast_sqrh = pll_sequencer::type_id::create("mast_sqrh", this);
         mast_drvh = pll_base_driver::type_id::create("mast_drvh", this);
      end
...
```

Figure 5. PLL agent.

Each agent has a configuration object (pll_cfg) for setting parameters specific to a PLL variant. Fig 6 below shows the configuration object for each of the PLL types. The code shows the divider settings for a particular application of the PLL that is constrained to a specific range of values.

```
typedef enum {PLL1, PLL2, PLL3, PLL4} pll_type_e;

class pll_cfg extends uvm_object;

   pll_type_e pll_type = PLL1;

   // Define test configuration parameters (e.g. how long to run)
    //--------------- Constraints for PLL frequency of operation ------------------
    rand int         feedback_divider;
    rand bit [5:0]   post_divider;
    rand bit [1:0]   ref_divider;

    constraint  ref_divider_c {
                ref_divider inside {0, 1, 2};
    }

    uvm_active_passive_enum is_active = UVM_ACTIVE;
   ...
```

Figure 6. PLL configuration settings.

Fig. 7 below shows the code for the base class driver (pll_base_driver).The pll_base_driver has the tasks for driving stimulus on the DUT interface based on the protocol. The virtual methods such as Coldbootpll, Freqchange, Reset_tog, etc. are the methods common to all the PLL variants. In the task tx_driver(), the transaction item (pll_seq_item) is fetched from the sequencer using the method get_next_item(). Depending upon the functionality defined by tr.kind, the appropriate methods (coldbootpll, frequency change, reset etc) are executed. These methods contain protocol specific procedures for driving the DUT signals.

```
class pll_base_driver extends uvm_driver #(pll_seq_item);
   // Phase methods
   // User defined methods
   extern protected virtual task cold_boot_task( pll_seq_item tr);
   extern protected virtual task tx_driver();
endclass: pll_base_driver

// Run phase
task pll_base_driver::run_phase(uvm_phase phase);
  tx_driver();
endtask: run_phase

// User defined methods
task pll_base_driver::tx_driver();
      pll_seq_item tr;
      seq_item_port.get_next_item(tr);
          case (tr.kind)
            pll_seq_item::COLDBOOTPLL: begin
              // Cold boot method
           end
            pll_seq_item::FREQCHANGE: begin
              // Frequency change method
           end
            pll_seq_item::RESET: begin
              // Reset method
           end
           ...

           endcase

          send(tr);
          seq_item_port.item_done();
          ...
```

Figure 7. PLL base class driver.

The derived components house exclusive methods for verifying a unique PLL variant. Fig. 8 below shows the derived component for one variant of PLL (PLL2). In the code, pll_type2_driver is extended from the base class driver pll_base_driver and the methods are modified, reused or overridden, based on the PLL2 feature.

```
class pll_type2_driver extends pll_base_driver;

   extern protected virtual task cold_boot_task( pll_seq_item tr);

endclass

task pll_type2_driver::cold_boot_task( pll_seq_item tr);
  // Modify original task
```

Figure 8. Driver extended for a different variant of the PLL.

Fig 9 shows the top testbench configuration object pll_tb_cfg that has the parameters defined for different pll types. The parameters such as pll_type_e (an enumeration that defines supported PLL types) and number_of_plls (number of PLL instances to be created) are defined along with other parameters that configure the testbench.

```
class pll_tb_cfg extends uvm_object;
   enum {PLL1, PLL2, PLL3, PLL4} pll_type_e;

   // Can be set to PLL1, PLL2, PLL3, PLL4
   pll_type_e pll_type = PLL1;

   //Array of configuration objects
   pll_cfg pll_cfgh[*];

   //Parameters
   rand bit [1:0] no_of_pll_inst;
   constraint no_of_pll_inst_c {
     soft no_of_pll_inst == 1;
   }
```

Figure 9. Top level TB configuration.

In Fig. 10, pll_env is an encapsulated, ready-to-use, configurable verification environment. Using factory overrides (set_type_override_by_type) the TB components (driver, monitor and sequencer) for a specific PLL variant is created. In the test, set the appropriate parameter (i,e pll_type) in the top configuration object (pll_tb_cfg). When the parameter pll_type = PLL1, all the base class components are instantiated. When the parameter pll_type = PLL2, the base class components are overridden by derived components (pll_type2_driver, pll_type2_monitor) using the factory override as shown below.

```
class pll_env extends uvm_env;
   pll_sb sbh;
   mon_2cov covh;
   pll_tb_cfg tb_cfg;
   pll_agent pll_agnth[*];


   extern virtual function void build_phase(uvm_phase phase);
   ...

endclass: pll_env

function void pll_env::build_phase(uvm_phase phase);
   // Get the tb config object from the database
   if(!uvm_config_db #(pll_tb_cfg)::get(this, "", "pll_tb_cfg", tb_cfg))
     `uvm_fatal(get_type_name(), "Failed to get pll_tb_cfg from the database")
   super.build_phase(phase);
   if (tb_cfg.pll_type == PLL1) begin
       for (i=0; i<tb_cfg.no_of_pll1_inst; i++)
       begin
         uvm_config_db #(pll_cfg)::set(this, $sformatf("pll_agnth[%0d].*",i), "pll_cfg", tb_cfg.pll_cfgh[i]);
         pll_agnth[i] = pll_agent::type_id::create($psprintf("pll_agnth[%0d]",i),this);
       end
   end

   else if (tb_cfg.pll_type == PLL2) begin
     set_type_override_by_type(pll_base_driver::get_type(), pll_type2_driver::get_type());
     set_type_override_by_type(pll_base_monitor::get_type(), pll_type2_monitor::get_type());
        for (j=0; j<tb_cfg.no_of_pll2_inst; j++)
        begin
         uvm_config_db #(pll_cfg)::set(this, $sformatf("pll_agnth[%0d].*",j), "pll_cfg", tb_cfg.pll_cfgh[j]);
         pll_agnth[j] = pll_agent::type_id::create($psprintf("pll_agnth[%0d]",j),this);
        end
```

Figure 10. Top level PLL environment.

Fig 11 shows multiple user defined tests which are selected for execution from the command line.
The base_pll_test instantiates the top level environment (pll_env) and user defined tests (pll1_test, pll2_test etc) are added based on the PLL variant.

```
class base_pll_test extends uvm_test;
  `uvm_component_utils(base_pll_test)

  pll_type_e pll_type;
  int no_of_pll_inst;
  pll_tb_cfg tb_cfg;
  slave_cfg slv_cfgh;
  pll_env envh;

  ...

class pll1_test extends base_pll_test;

  virtual function void build_phase(uvm_phase phase);
    pll_type = PLL1;
    no_of_pll_inst = 1;

    ...
class pll2_test extends base_pll_test;

  virtual function void build_phase(uvm_phase phase);
    pll_type = PLL2;
    no_of_pll_inst = 2;

    ...
```

Figure 9. PLL base test and derived test.

A sequence library, as shown in Fig 12, houses various scenarios for testing PLL variants. In the planning phase, all scenarios for verifying the PLL variants are listed out and coded into sequence libraries. One such sequence library (pll_sequencer_sequence_library) extended from uvm_sequence_library is listed below. Sequences that are common to all PLL variants are registered with pll_sequencer_sequence_library. For example, cold_boot_pll_seq is a sequence that is common to all the PLL variants and is registered using the macro `uvm_add_to_seq_lib(). This sequence is then invoked by the testcase for executing the PLL cold boot scenario.

```
class pll_sequencer_sequence_library extends uvm_sequence_library # (pll_seq_item);


  function new(string name = "simple_seq_lib");
    super.new(name);
    init_sequence_library();
  endfunction

endclass

//Base sequence
class base_sequence extends uvm_sequence #(pll_seq_item);

...
endclass

class cold_boot_pll_seq extends base_sequence;
  `uvm_add_to_seq_lib(cold_boot_pll_seq,pll_sequencer_sequence_library)

   rand kind_e kind1;
   constraint kind1_c {kind1 == COLDBOOTPLL;}
   pll_cfg m_cfg;

   ...
```

Figure 10. Sequence library with sequences.

VI. CONCLUSION

A. *Summary with results*

Every time a new PLL IP has to be verified, only three steps (below) need to be followed, as the rest of the verification suite (sequences, sequencer, common methods in base components) will be reused as is.

1. Extend the PLL test with appropriate configuration object setting.
2. Extend the driver/monitor and add exclusive methods specific to the PLL variant.
3. Establish the DUT to TB connection through virtual interface.

Using the approach described above, a UVM based common verification environment was developed for four different variants of the PLL. Most of the verification environment is reusable and scalable with minimal effort. Our approach reduces:

1. Verification **effort**: verification bring up effort is minimized due to re-use of existing verification setup.
2. Testbench development and debug **time**: common features may be verified/debugged using the existing methods, thereby saving time for developing new scenarios.
3. Verification resources: with Verilog based testing, each PLL had a unique testbench requiring a dedicated resource. However, with aforementioned UVM based methodology, only one verification resource is required to manage multiple PLL variants.

   With this implementation in our current project, we see an almost 50% reduction in overall verification effort towards verifying the variants of the PLL.

B. Key Takeaways

- Single shared testbench architecture for all variants of PLL.
- One time creation of the overall testbench and reduced effort on extending the testbench for additional DUT variants.
- Our solution provides a unified verification environment template for the verification of IPs with similar design/functionality.

ACKNOWLEDGMENT

We would like to acknowledge our colleagues: G. Raju, M. Chiang, and D. Robinson.

REFERENCES

[1] Accellera Systems Initiative Universal Verification Methodology (UVM) 1.1 User's Guide, May, 2011.
[2] Accellera Systems Initiative Universal Verification Methodology (UVM) 1.1 Class Reference Manual, Jun, 2011.
[3] G. Eason, B. Noble, and I.N. Sneddon, "If SystemVerilog is so good, why do we need the UVM? Sharing responsibilities between libraries and the core language," Specification & Design Languages (FDL), 2013 Forum, Paris, France, Sep. 2013.
[4] IEEE, Standard 1800-2012 for SystemVerilog Hardware Design and Verification Language, New York, NJ: IEEE, 2012.
[5] Synopsys, Inc., Reference Verification Methodology, 2002.
[6] UVM World Website www.accellera.org

.