UVM architecture for performance: Go hierarchical!

Paul Lungu, Zygmunt Pasturczyk, James Strober,
Ciena, Ottawa, Canada

Christopher Taylor,
Coveloz, Ottawa, Canada

Chris Thompson,
Synopsys, Ottawa, Canada

www.ciena.com
www.coveloz.com
www.synopsys.com

ABSTRACT

Ever growing complexity in the ASIC world always requires verification techniques to be adapted in order to keep up with changing requirements and more aggressive deadlines. This paper presents a flexible UVM based architecture used to tapeout complex optical transport ASICs with an aggressive schedule. The architecture focuses mostly on creating every component of the UVM environment in a hierarchical fashion mimicking the design hierarchy. Hierarchical interfaces and configuration classes passed along the hierarchy according to UVM rules are also part of the discussion. The paper will show how virtual sequences and virtual sequencers are hierarchically built starting from the leaf level and following the design hierarchy. At the same time this dual toplevel architecture leaves the door open for acceleration using emulation technology just in case the project requires it. A set of base classes and other useful UVM verification techniques used as building stone of the environments will be presented. "Hierarchical everything" architectural concept started years ago with VMM environments and it has been adapted to UVM recently. It helped the verification team to quickly and efficiently adapt to a constant increase in complexity and ensured maximum productivity as the team size increased towards the end of the project. The paper will address the standardization concept at different levels of environment and explain how this methodology reduced verification time overall as well as the learning curve for different contributing teams, maximizing efficiency.

Table of Contents

Table of Figures

1. Introduction

The exponential increase in complexity of the ASICs in the last several years has driven verification engineers to search for solutions to adapt to the new challenges. Since the adoption of UVM as a standard vendor agnostic verification methodology many papers detailing the use of the methodology have been written. Presenters and authors made efforts to teach the UVM community how to use the new methodology under different circumstances. UVM was initially challenging to the VMM users especially in the first years of adoption.

This paper presents a proposal for an UVM based architecture for large ASICs starting from the assumption that a significant group of engineers will work independently in different parts of the verification environment and at the end everything has to smoothly converge in a toplevel integration.

Section 2 will explain why the hierarchical architecture was the best choice and the team managed to deliver the verification in record time due to this simple yet powerful concept.

The paper will introduce the base class concept in Section 3 as a foundation for reusable code and present the necessary examples of how to build the architecture in the subsequent Section. Section 4 will give the user all the necessary tools and knowledge to quickly build the main components of the verification architecture in a hierarchical fashion starting from the design hierarchy. This will allow the verification team to work independently in every corner of the device with little or no interaction.

The device under test (DUT) at Ciena was a complex DSP ASIC with hundreds of millions of gates. A simplified block level diagram of the ASIC is shown in Figure 1. As seen below, the chip is made of five subsystems and each subsystem is made of multiple functional blocks.

Each block has its own interfaces to allow inter-block or inter-subsystem communication. There is also a processor interface to allow software to configure and service the ASIC. The diagram in Figure 1 shows the actual logical partitioning of the design.

Block level verification requires an agent for traffic generation/monitoring and a it also requires a scoreboard. The block is called in this paper leaf level. However, as the paper will describe in the subsequent sections, all UVM components for block, subsystem or chip level are the same the only difference consists in the position in the hierarchy. For example: block level (leaf level) components will instantiate the agent components while a subsystem wills instantiate all the block components. Sub-system and chip level verification will reuse the block level verification in terms of traffic generation and/or software configuration. Chip level components will have only sub-system components as members and not the block level ones. Hence a very strict hierarchical subordination: chip -> sub-system -> block -> agent.

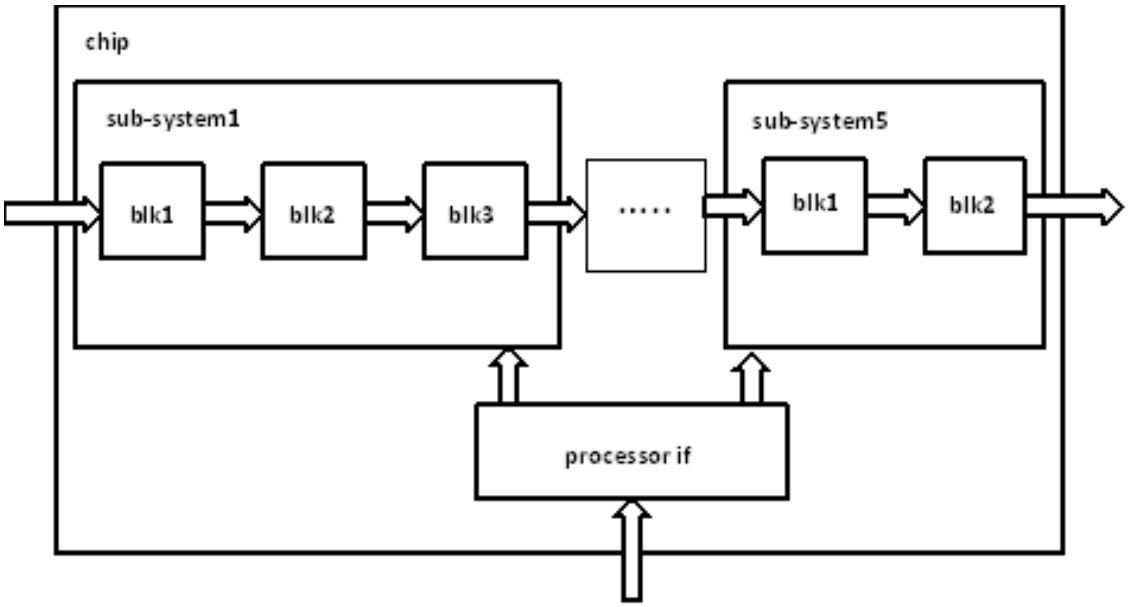

Figure 1. Chip level diagram

Chiplevel verification started way before block level verification at different levels was completed. Due to the simplicity and hierarchical reuse of the components toplevel verification was possible in early stages of the verification and it ran in parallel with the block or subsystem verification.

The VCS-MX release used at the end of the project was 2016.06.

2. Why hierarchical architecture?

One of the main requirements of productivity in verification is to allow as many verification engineers as possible to work independently at the same time from different places and with little interaction. This is possible only when the DUTs can be separated and the verification code is not common. In order to achieve this we had to mimic the design hierarchy as much as possible in verification by creating components which require little interaction once the verification progresses towards the chip level. At the same time, the architecture had to be flexible enough to allow starting the verification at higher levels when only the basic verification components have been developed for the lower level. This architecture speeds up the progress on verification and allows much faster progress at higher levels of the hierarchy while only basic verification has completed at leaf level. The main advantage UVM offers is the factory override concept that can be used at different levels of the hierarchy without changing the fundamental structure of the class if functional changes are needed. All the tests written for this project used factory override to create their own configuration classes or sequences while preserving the foundation and base

functionality of the existing classes. This concept along with the ability to pass the hierarchical interface encapsulated inside the configuration class via the configuration database is at the base of the proposed architecture.

2.1 What UVM components must be hierarchical?

Any UVM architecture uses several UVM standard components: env, sequencer, sequence, config class, agents, base UVM test and interfaces to connect with the DUT. Based on the diagram shown in Figure 1 was determined that for this specific project the components have to be hierarchical and follow the design hierarchy. The architectural view of this hierarchical approach is shown in Figure 2. The only non-hierarchical component is the test but in this case was determined that a base class is still required to avoid code duplication from one block to another. However, at different levels of the hierarchy the user will need to implement another base test. Also, at the same hierarchy level, in case HW acceleration is needed, a separate base test for acceleration has to be developed. The only non-UVM component which has to be hierarchical is the interface. The higher level components will have the lower level ones as members for both UVM and non-UVM components. Section 4 will give details about each component at lower and higher levels.

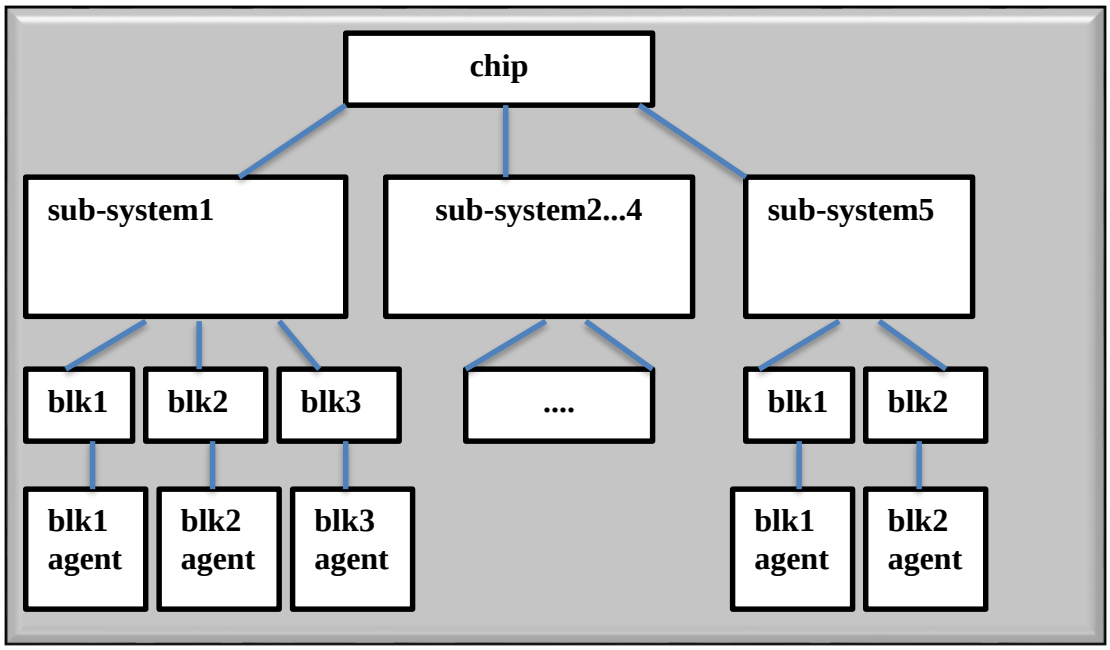

Figure 2. Chip level architectural view

Figure 2 describes the architectural diagram from leaf level to chip level. The leaf level is the level containing the physical interface associated with an agent in the verification environment. It is the lowest architectural level described in this paper. The hierarchical architecture assumes that the agents are encapsulated inside an UVM environment defined in the next sections as blk_agent_env. All UVM components will follow the hierarchy described in Figure 2.

In the subsequent sections chip level won't be treated as a standalone entity simply because it behaves like any other block or sub-system and all the descriptions will stop at block or sub-system level. The only difference between a chip level and block or sub-system would be that the chip level has no further hierarchy above in a simple single chip simulation. In the case of multi-chip simulation the chip level would be part of another hierarchy and the top level in this case will oversee two or more chip level components.

2.2 Accommodate emulation

Architecting with emulation in mind is another requirement the verification engineers have to be aware of. In order to allow HW emulation in the future, the verification (non-synthesizable) constructs have to be part of a HVL toplevel while the synthesizable constructs have to be part of a HDL toplevel. This rule has to be applied at any level of the hierarchy since it is hard to predict from the beginning which block or subsystem will need to be accelerated. Adding hooks in place for emulation at the start of the project is always a good idea but changing the code later on to accommodate the emulation can also be done [1].

3. Reusable set of parameterized base classes

The strategy to implement and support a set of base classes for fundamental verification component building blocks provides significant flexibility benefits for building and managing a library of reusable and integrable verification environments. When we require verification components to inherit from these base classes, it provides a hook we can use to introduce generic behaviour at any point in time. Furthermore, one can introduce functionality that will have broad value across all existing and future verification environments without disturbing the code base of individual existing environments.

Focusing key functionality and infrastructure to a common locality for all verification environments provides universal access for debug instrumentation and valuable common convenience methods and utilities.

One can extend this idea to implement common or generic elements by leveraging System Verilog's capacity to support parameterizable classes. Specifically, one can achieve a generic architectural framework by providing a set of parameterizable base classes with key architectural classes as the parameters.

In the absence of the ability to extend our parameter set for these classes after they are adopted, it becomes imperative to carefully consider which set of class parameters to support for each base class. However, once this is done, the arduous and often error-prone tasks of implementing the instantiation and construction of all elements within a desired UVM framework is contained in the implementation of the parameterized base classes.

3.1 Configuration base classes

3.1.1 common_config base class

The common_config base class shown in Figure 3 encapsulates two more classes, those being a DUT configuration class and an environment configuration class as well as a virtual interface. All of these entities are necessarily defined as parameters of the common_config base class

The example below shows that both the DUT configuration class and the environment configuration class should be extensions of two more base classes described in 3.1.2 and 3.1.3 below.

The virtual interface parameter should refer to a defined interface that hierarchically encapsulates all required virtual interfaces required to stimulate and monitor the DUT. Depending on the hierarhical structure, DUT represents either a block or subsystem level or even the chip level.

```
class common_config #(type ENV_CONFIG_T = uvm_object,
                      type DUT_CONFIG_T = uvm_object,
                      type VIF_T = virtual dummy_if
                     ) extends uvm_object;

  rand ENV_CONFIG_T m_env_config;
  rand DUT_CONFIG_T m_dut_config;
  VIF_T itf;
...

endclass: common_config
```

Figure 3. common_config base class

3.1.2 common_dut_config base class

The dut_config base class should be extended to include configuration attributes that are tightly coupled to the initial target DUT for the verification environment being developped. The nature of this configuration maybe tightly coupled to the DUT software interface if appropriate.

The motivation behind isolating DUT specific configuration items within class is that we may at some future point re-use most elements of a verification environment to verify a similar but new DUT with a presumably different software interface. If that's the case, then this is the only one class that must be re-defined.

```
class common_dut_config extends uvm_object;
...

endclass: common_dut_config
```

Figure 4. common_dut_config base class

3.1.3 common_env_config base class

The common_env_config base class should be extended to encapsulate configuration attributes that are specific to the environment and independent of the DUT. In this manner, we provide a mechanism to decouple configuration of the environment and configuration of the DUT.

For any given DUT, there is a necessary requirement to align both DUT and environment configuration so that they are consistent. This bridging function should be implemented outside of the common_env_config and common_dut_config extensions for a given verification environment.

This approach adds a marginal architectural complexity and overhead in order to facilitate substituting the current DUT with a different but functionally similar DUT at some future time. If discipline is maintained in encapsulating DUT and environment confiuration attributes into their approriate class, the effort of substituring with a new DUT is reduced to re-defining a new extension of the common_dut_config base class and re-implementing the method that aligns DUT and environment configuration.

As seen in Figure 5 below, the supported parameters include information to dimension generally needed elements such as number of clocks, number of resets and number of host interfaces. Some variables are needed to control all environments: enable_in/enable_out and is_toplevel. Enable_in/out switches are critical in controlling the flow by enabling drivers/monitors at the leaf level from higher levels of the hierarchy. is_toplevel variable decides when that specific block is run in a simulation as a snadalone block or in a hierarchical structure. All constraints at

this level are either soft or default. Default constraints are chosen when hard constraints are present at the highe rlevel. Hierarchical default contraints do not work, they just silently fail.

```
class common_env_config #(int CLK_NUM_ENB=0,
                          int RST_NUM_ENB=0,
                          int EBUS_NUM_ENB=0,
                          int GMS_NUM_ENB=0) extends uvm_object;

  int  m_clk_num_enb=CLK_NUM_ENB;  //number of clocks enabled
  int  m_clk_periods[CLK_NUM_ENB];
  ...

  rand bit m_enable_in;
  rand bit m_enable_out;

  rand bit m_is_toplevel;
  rand clk_pkg::clk_config                        m_clk_config[];

  constraint enable_in_c  { soft m_enable_in  == 1; }
  constraint enable_out_c { soft m_enable_out == 1; }

  // by default every block is a toplevel
  // toplevel env config will have to overrite this to 0 for each sub env
  default constraint is_toplevel_c { m_is_toplevel == 1; }

  // factory registration macro
`uvm_object_param_utils_begin(common_env_config#(CLK_NUM_ENB,RST_NUM_ENB,E
BUS_NUM_ENB,GMS_NUM_ENB))
     `uvm_field_int(m_clk_num_enb, UVM_DEFAULT)
     ...
     `uvm_field_int(m_enable_in, UVM_DEFAULT)
     `uvm_field_int(m_enable_out, UVM_DEFAULT)
   `uvm_object_utils_end
endclass: common_env_config
```

Figure 5 common_env_config base class

3.2 Sequence base class

The code snippet describing the base class for sequences is shown in Figure 6 and shows the base class that needs to be extended for all phase launched sequences. The type based parameters supplied in order to create a specialization of this base class describe the specific verification environment component types that a typical sequence would require.

The base class implementation (not shown) accesses the p_sequencer handle in order to populate handles for all the relevant verification components.

As a convenience, the code to raise and drop objections is implemented (not shown) in the base class.

```
class common_base_phase_seq #(type ENV_T = uvm_env,
                               type CONFIG_T = uvm_object,
                               type ENV_CONFIG_T = uvm_object,
                               type DUT_CONFIG_T = uvm_object,
                               type API_T = uvm_object,
                               type BASE_SEQR_T = uvm_sequencer,
                               type RAL_T = uvm_reg_block,
                               type VIF_T = virtual dummy_if,
                               int  RST_NUM_ENB = 0) extends uvm_sequence;


`uvm_object_param_utils(common_base_phase_seq#(ENV_T,CONFIG_T,ENV_CONFIG_T,
DUT_CONFIG_T,API_T,BASE_SEQR_T,RAL_T,VIF_T,RST_NUM_ENB))

   ENV_T         m_env;
   CONFIG_T      m_config;
   DUT_CONFIG_T  m_dut_config;
   ENV_CONFIG_T  m_env_config;
   RAL_T         RAL;
   VIF_T         itf;

   uvm_status_e         status;

   // Example of a functional Sequence Library
   rst_seq_lib              rst_lib[RST_NUM_ENB];
...
  `uvm_declare_p_sequencer(BASE_SEQR_T)
...
endclass : common_base_phase_seq
```

Figure 6 common_base_phase_seq base class

3.3 Sequencer base class

It can be seen from Figure 7 that the sequencer base class encapsulates sequencers expected to be common to all verification environment sequencers, namely reset and processor interface type sequencers.

The only other type-based parameter is to specify a common_config class extension. All other verification component handles are accessible through an environment handle which is accessible through a get_parent() call. Using this verification environment architecture all necessary handles can be accessed and populated in the common base class layer.

```
class common_base_sequencer #(type CONFIG_T = uvm_object ,
                              type RAL_T = uvm_reg_block,
                              int  RST_NUM_ENB  = 0,
                              int  EBUS_NUM_ENB = 0,
                              int  GMS_NUM_ENB  = 0
                             ) extends uvm_sequencer;


`uvm_component_param_utils(common_base_sequencer#(CONFIG_T,RAL_T,RST_NUM_ENB
,EBUS_NUM_ENB,GMS_NUM_ENB))

  //---------------------------------------
  // Create handle for all sequencers
  //---------------------------------------

  rst_pkg::rst_sequencer                    m_rst_seqr[RST_NUM_ENB];
  ebus_pkg_verif::ebus_sequencer            m_ebus_seqr[EBUS_NUM_ENB];
  gms_master_pkg::gms_master_sequencer      m_gms_seqr[GMS_NUM_ENB];

  //---------------------------------------
  // Create handle for cfgs + RAL
  //---------------------------------------

  CONFIG_T             m_config;
  RAL_T                RAL;

 ...
endclass : common_base_sequencer
```

Figure 7 common_base_sequencer base class

3.4 Environment base class

The code example in Figure 8 shows the implementation of the common_env base class. This class encapsulates agent environments for resets, clocks and processor interfaces (there is an assumption that all blocks have the same type of processor interface access). This common infrastructure is defined and implemented only once, thus shortening typical and necessary integration and debugging effort.

With the supplied parameters, the common class can successfully perform all steps at appropriate phases to realize a verification environment complete with all the necessary common infrastructure.

Note that string-based parameters are supplied when specializing and extending this base class for a specific verification environment targeted for verifying a specific DUT. These are necessary for operations of fetching object handles from the configuration database that are instantiated and registered in the base test, as is a standard approach for UVM environments.

```
class common_env  #(type CONFIG_T = uvm_object,
                    type BASE_SEQR_T = uvm_sequencer,
                    type RAL_T = uvm_reg_block,
                    string CFG_STR = "",
                    string RAL_STR = "",
                    string BASE_SEQR_STR = ""
                   ) extends uvm_env;


`uvm_component_param_utils(common_env#(CONFIG_T,BASE_SEQR_T,RAL_T,CFG_STR,RA
L_STR,BASE_SEQR_STR))

  rst_env        m_rst;
  clk_env        m_clk;
  ebus_env       m_ebus;
  gms_master_env m_gms;

  CONFIG_T m_config;
  RAL_T RAL;

  BASE_SEQR_T m_base_seqr;
 ...
endclass: common_env
```

Figure 8 common_env base class

3.5 UVM test base class

Based on a survey of generally accepted techniques for re-using verification environment across different hierarchies, we came across no examples for test cases. This is attributable to the fact that different hierarchies require different testcases.

However, we recognize that this does not preclude the implementation of a common approach to creating a base test implementation. Typically the base test case for a verification environment is tasked with instantiation and registration of the key verification components namely: the environment, the configuration objects, and the register models.

Also, our approach is to take advantage of default sequences. Therefore we need to identify the phase sequences through suitable parameters so that we may register them as default sequences which will kick-off automatically when we enter the corresponding phases.

Because of the significantly large customizable content, the base test base class necessarily incorporates a large number of parameters. This key part of the verification infrastructure de-risks the schedule in terms of integration and debugging at the slight expense of a cumbersome specialization definition.

The definition of this test base class is shown in Figure 9.

```
class common_test_base #(type ENV_T = uvm_env,
                         type CONFIG_T = uvm_object,
                         type ENV_CONFIG_T = uvm_object,
                         type DUT_CONFIG_T = uvm_object,
                         type API_T = uvm_object,
                         type RESET_SEQ_T = uvm_sequence,
                         type CONFIG_SEQ_T = uvm_sequence,
                         type MAIN_SEQ_T = uvm_sequence,
                         type SHUTDOWN_SEQ_T = uvm_sequence,
                         type RAL_T = uvm_reg_block,
                         type VIF_T = virtual dummy_if,
                         string CFG_STR = "",
                         string RAL_STR = "",
                         string IF_STR = ""
                        ) extends uvm_test;


`uvm_component_param_utils(common_test_base#(ENV_T,CONFIG_T,ENV_CONFIG_T,DU
T_CONFIG_T,API_T,RESET_SEQ_T,CONFIG_SEQ_T,MAIN_SEQ_T,SHUTDOWN_SEQ_T,RAL_T,V
IF_T,CFG_STR,RAL_STR,IF_STR))

   uvm_watchdog m_test_watchdog;

   ENV_T         m_env;
   API_T         m_api;
   CONFIG_T      m_config;
   DUT_CONFIG_T  m_dut_config;
   ENV_CONFIG_T  m_env_config;
   MAIN_SEQ_T    m_seq;
   RAL_T         RAL;

   ...
endclass: common_test_base
```

Figure 9. common_test_base base class

4. Build the architecture

As discussed in Section 2 the architecture of the verification environment is hierarchical and most of the classes used in the development are extended from the based classes described in Section 3. The following sub-sections will describe in detail how subsystem and chip level verification components (including interfaces) need to be built based on base classes and following the hierarchical approach. An important part of the success of such an architecture is the ability to structure all the components hierarchically. These details will be described in the sub-sections 4.7, 4.10. Worth to mention that all the examples given in the subsequent sub-sections are based on a generic part of the environment and not necessarily a block, subsystem or top level. It is up to the verification prime or the verification engineer to assign the right hierarchy and approach based on the type of verification environment which needs to be built. Since the architecture assumes the same approach at any level of the hierarchy the naming convention (block, subsystem, chip etc.) will be used only as an example but any combinations of names is allowed.

4.1 Configuration classes

One of the most important aspects of a fully random verification strategy is the use of configuration classes. They are used to help the verification engineer to constrain the random stimulus and propagate it on a hierarchy if necessary. They also play the role of a container to carry other components across the hierarchy. In this project we used the config class to carry the hierarchical interface. This paper will describe the use of configuration classes from the chip level tests all the way down to the agent and how they stack up and help the verification engineers play with different knobs to reach interesting corner cases at the agent level or switch from one scenario to another while at higher levels of the hierarchy. For this particular project each hierarchical level (except the agents) used two types of configuration classes as described in 3.1: dut_config and env_config. While env_config class extends from a base class and it is fully reusable at higher levels, dut_config class is a standalone class not reusable when jumping from one level to another and it is DUT specific. Both of them need to be encapsulated in one config class along with the hierarchical interface for that specific block as specified in Section 3.1.

```
class blk_env_config extends common_env_config
                                          #(blk_params_pkg::NUM_CLK_IF,
                                            blk_params_pkg::NUM_RST_IF,
                                            blk_params_pkg::NUM_EBUS_IF,
                                            blk_params_pkg::NUM_GMS_IF);
  rand blk_agent_config  m_blk_agent_config ;
  rand bit               m_enable_sblk_in[blk_params_pkg::NUM_BLK_IF];
  rand bit               m_enable_sblk_out[blk_params_pkg::NUM_BLK_IF];
  constraint blk_clk_c {
    m_clk_config[BLK_CLK].m_period == SBLK_CLK1;
    m_clk_config[BLK_CLK].m_time_unit_spec == clk_pkg::clk_config::FS;.. }
   `uvm_object_utils_begin(blk_env_config)
     `uvm_field_object(m_sblk_agent_config, UVM_ALL_ON)
     `uvm_field_array_int(m_enable_blk1_in, UVM_ALL_ON)
...
  `uvm_object_utils_end

  function new(string name="blk_env_config");
    super.new(name);
  endfunction
endclass: blk_env_config
```

Figure 10. Example of a generic leaf level env_class

As detailed in the above sections the env_config class helps the user to constrain switches necessary in creating different scenarios at different levels of the hierarchy while dut_config is strictly used to constrain variables to configure the DUT. Figure 10 shows the env_config code for a generic leaf level of the hierarchy. The code example shows an agent configuration class being used in a generic env_class. This is needed only at lower levels of the hierarchy but not needed at higher levels. Only immediate block level environments will make use of the agents.

Now with the generic env_class defined the following example will show how to create the higher level of the config_class which is made of three important components for any level: env_class, dut_class and interface. On top of these the hierarchical config classes have to be added as appropriate based on the level of the hierarchy the component is defined at.

```
typedef common_config #(blk_env_config,blk_dut_config,virtual blk_if
                      ) blk_config_base_t;

class blk_config extends blk_config_base_t;
  rand blk1_config       m_blk1_config[];
  rand blk2_config       m_blk2_config[];

  `uvm_object_utils_begin(blk_config)
    `uvm_field_array_object(m_blk1_config, UVM_ALL_ON)
    `uvm_field_array_object(m_blk2_config, UVM_ALL_ON)
  `uvm_object_utils_end

  constraint is_top_level_blk_c   {
    foreach(m_blk1_config[i]) {
        m_blk1_config[i].m_env_config.m_is_toplevel==0;  }
    foreach(m_blk2_config[i]) {
        m_blk2_config[i].m_env_config.m_is_toplevel==0;  }
  }
  // propagate enables down the hierarchy
  constraint blk1_en_hier_c   {
    foreach(m_blk1_config[i]) {
      m_blk1_config[i].m_env_config.m_enable_in  ==
      m_env_config.m_enable_blk1_in[i];
      m_blk1_config[i].m_env_config.m_enable_out ==
      m_env_config.m_enable_blk1_out[i];   }
  }
  constraint blk2_en_hier_c   {
    foreach(m_blk2_config[i]) {
      m_blk2_config[i].m_env_config.m_enable_in  ==
      m_env_config.m_enable_blk2_in[i];
      m_blk2_config[i].m_env_config.m_enable_out ==
      m_env_config.m_enable_blk2_out[i];   }
  }
  function new(string name = "blk_config");
   string inst_name;  super.new(name);

   m_blk1_config   = new[NUM_SBLK1_IF];
   m_blk2_config   = new[NUM_SBLK2_IF];

   foreach(m_blk1_config[i]) begin $sformat(inst_name, "m_blk1_config[%0d]", i);
    m_blk1_config[i] = blk1_config::type_id::create({get_name(),".",inst_name});
   end
   foreach(m_blk2_config[i]) begin $sformat(inst_name, "m_blk2_config[%0d]", i);
    m_blk2_config[i] = blk2_config::type_id::create({get_name(),".",inst_name});
   end
  endfunction: new
```

Figure 11. Example of a higher level config class

Figure 11 shows how to build the configuration hierarchy at higher levels. The example assumes blk level of the hierarchy contains two sub-blocks sblk1 and sblk2. These sub-block levels can contain an agent which means they can be a leaf level. As described in the code the variable is_toplevel from the base class is constrained as needed by the higher level config class down to the hierarchy. This is the only way to tell the layers underneath who is the toplevel at any time. This variable has to be used everywhere when decisions regarding who is in charge have to be made. Also this code describes how to enable or disable any level of the hierarchy. In fact the code shows how to enable/disable the driver or monitor from being used at any levels. While the driver needs the ability to be enabled or disabled at different levels of the hierarchy based on the scenario set by that hierarchy level the monitor are usually on across the entire hierarchy, however the option to disable them has been given just in case one needs to use it.

4.2 Environment class

After creating the config class for a block level environment or a higher level of hierarchy the next step is to create the environment class. As detailed in Section 3.2 the environment class extends from a base class which is reused by all the hierarchical levels in the project including the chip level environment. For this project a reusable library for agents is used and all the agents from the library are architected to come as UVM environments. This is why in the example from Figure 12 the agent is treated similar to any other environment developed for the project. In the case of an environment class two separate examples are discussed: one for leaf level which contains the agent and the other for higher levels which doesn't contain agents. The last one contains only hierarchical components.

```
class sblk1_env extends common_env #(sblk1_config, sblk1_sequencer, dut_sys_ral,
                                      "sblk1_config","ral_sys_dut","sblk1_sequencer");
  `uvm_component_utils(sblk1_env)

  sblk1_agent_env m_agenv;  // agent subenv
  sblk1_scbd      m_scbd;   // scoreboard

  function new(string name = "sblk1_env", uvm_component parent = null);
    super.new(name, parent);
  endfunction: new

  // startup phases
  extern virtual function void build_phase(uvm_phase phase);
  extern virtual function void connect_phase(uvm_phase phase);
 endclass: sblk1_env
```

Figure 12. Example of a generic leaf level environment class

In this case the block generic agent is added and this is used to drive traffic into the DUT pins at block level or higher levels. Also the scoreboard is something specific to this particular block

level environment and it has to be added along with the other variables as a member of the class. The rest of the members passed thru the parameters are automatically set as members of the class due to the use of the base class.

The build_phase() and connect_phase() for this class in Figure 13 show how the agent and scoreboard are hooked up at the upper layer which is standard UVM procedure in this case. The user needs to take note of the super() call right at the beggining of the phase to ensure the content of the base classes is executed first.

```
function void blk1_env::build_phase(uvm_phase phase);
  super.build_phase(phase);

 foreach (m_config.m_env_config.m_blk1_agent_config[i]) begin
  uvm_config_db#(blk1_config)::set(this, "m_agenv",
  $sformatf("config[%0d]", i), m_config.m_env_config.m_blk1_agent_config[i]);
 end
 m_scbd = fec_enc_scbd::type_id::create("m_scbd", this);
 m_agenv= blk1_agent_env::type_id::create("m_agenv", this);
 ...
endfunction: build_phase

function void sblk1_env::connect_phase(uvm_phase phase);
  super.connect_phase(phase);

  m_base_seqr.m_drv_seqr = m_agenv.m_agents[0].m_drv_seqr;
  m_agenv.m_agents[0].ap_drv.connect(m_scbd.ap_drv);
  m_agenv.m_agents[0].ap_mon.connect(m_scbd.ap_mon);

endfunction: connect_phase
```

Figure 13. build_phase() and connect_phase() for a generic leaf level env class

The next example will show how to build a generic environment at higher levels of the hierarchy. All the environments containing blocks have to be built as described in Figure 14. This type of approach applies to sub-system or chip level environments basically everything which is higher than the leaf level environment. The user can use as many levels as necessary there is no restriction. Ideally the hierarchy at this level reflects the design hierarchy.

```
typedef common_env #(blk_config, blk_sequencer, dut_sys_ral,
                     "blk_config", "ral_sys_dut", "blk_sequencer")
blk_env_base_t;

class blk_env extends blk_env_base_t;
  `uvm_component_utils(onramp_env)

  blk1_env       m_blk1_env[];
  blk2_env       m_blk2_env[];
  blk_scbd       m_scbd; // optional

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

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


  // UVM domain phases
  extern virtual task reset_phase(uvm_phase phase);
  extern virtual task configure_phase(uvm_phase phase);
  extern virtual task main_phase(uvm_phase phase);
  extern virtual task shutdown_phase(uvm_phase phase);

  // final reporting phases
  extern virtual function void check_phase(uvm_phase phase);
  extern virtual function void report_phase(uvm_phase phase);

endclass: blk_env
```

Figure 14. Generic higher level environment class

In the next example, in

Figure 15, the build_phase() and connect_phase() of a generic higher level environment are discussed.

```
function void blk_env::build_phase(uvm_phase phase);
  string inst_name;
  super.build_phase(phase);

  m_blk1_env   = new[NUM_BLK1_IF];
  m_blk2_env   = new[NUM_BLK2_IF];

  `uvm_info(get_type_name(), "build blk1 sub_env ...", UVM_MEDIUM)
  for (int i=0; i<NUM_BLK1_IF;i++) begin
    $sformat(inst_name, "m_blk1_env[%0d]", i);

uvm_config_db#(blk1_config)::set(this,$psprintf("m_blk1_env[%0d]*",i),
"blk1_config", m_config.m_blk1_config[i]);
    m_blk1_env[i] = blk1_env::type_id::create(inst_name, this);
  end
  `uvm_info(get_type_name(), "build blk2 sub_env ...", UVM_MEDIUM)
  for (int i=0; i<NUM_BLK2_IF;i++) begin
    $sformat(inst_name, "m_blk2_env[%0d]", i);

uvm_config_db#(blk2_config)::set(this,$psprintf("m_blk2_env[%0d]*",i),
"blk2_config", m_config.m_blk2_config[i]);
    m_blk2_env[i] blk2_env::type_id::create(inst_name, this);
  end
uvm_config_db#(dut_sys_ral)::set(this, "*", "ral_sys_dut", RAL);
endfunction: build_phase

function void blk_env::connect_phase(uvm_phase phase);
  super.connect_phase(phase);

  // connect sequencers to subenvs
  for (int i=0; i<NUM_BLK1_IF;i++)
    m_base_seqr.m_blk1_seqr[i]   = m_blk1_env[i].m_base_seqr;
  for (int i=0; i<NUM_BLK2_IF;i++)
    m_base_seqr.m_blk2_seqr[i]   = m_blk2_env[i].m_base_seqr;
endfunction: connect_phase
```

Figure 15. build_phase() and connect_phase() of a generic higher level env class

Figure 15 shows for the first time how the hierarchy gets built in UVM at different levels. This project follows a directional architecture in the sense that components are passed thru the config_db using direction. This happens from higher levels to lower levels of the hierarchy. In the build_phase() of the higher level env class the config classes for the leaf levels are unrolled. They are received from the higher level of the hierarchy inside the m_config class via config_db

used inside the common_env class. The unrolling happens inside the build_phase() by setting them directionally using config_db. The set function of the config_db clearly specifies where every sub-block level config class goes. Important is the broadcasting of the RAL class which doesn't follow the directional rule of the config class. This is mostly because RAL class gets used in multiple places and there are a higher number of components benefiting from this broadcast. Also, in the connect_phase() function details of connecting the leaf level sequencer to the upper level ones basically creating the hierarchy of virtual sequencers.

4.3 Virtual sequencers

In this section two examples of a virtual sequencer will be given. The leaf level sequencer used in sub-block level or block level environments will contain the sequencer defined inside the agent environment. Similar with the hierarchical approach used while defining the env class the virtual sequencers follow the design hierarchy identical with other classes.

Figure 16 shows the virtual sequencer for a sub-block level used to drive data into the physical interface while Figure 17 shows how to build a virtual sequencer at higher levels of the hierarchy. The first example shows the driver sequencer being a member of this virtual sequencer extended from the base class defined in Section 3.3.

```
typedef common_base_sequencer #(blk1_config,
                                 dut_sys_ral,
                                 blk1_params_pkg::NUM_RST_IF,
                                 blk1_params_pkg::NUM_EBUS_IF,
                                 blk1_params_pkg::NUM_GMS_IF)
blk_base_sequencer_t;

class blk1_sequencer extends blk_base_sequencer_t;
  `uvm_component_utils(blk1_sequencer)

  blk1_agent_pkg::blk1_drv_sequencer  m_drv_seqr;

  function new (string name = "blk1_sequencer",
                uvm_component parent = null);
    super.new(name, parent);
  endfunction
endclass: blk1_sequencer
```

Figure 16. Example of a generic leaf level virtual sequencer

```
typedef common_base_sequencer #(blk1_config,
                                 dut_sys_ral,
                                 blk1_params_pkg::NUM_RST_IF,
                                 blk1_params_pkg::NUM_EBUS_IF,
                                 blk1_params_pkg::NUM_GMS_IF)
blk_base_sequencer_t;

class blk_sequencer extends blk_base_sequencer_t;
  `uvm_component_utils(blk_sequencer)

  blk1_env_pkg::sblk1_sequencer       m_blk1_seqr[];
  blk2_env_pkg::sblk2_sequencer       m_blk2_seqr[];

  function new (input string name="blk_sequencer", input uvm_component
parent=null);
    super.new(name, parent);

    m_blk1_seqr = new[blk_params_pkg::NUM_BLK1_IF];
    m_blk2_seqr = new[blk_params_pkg::NUM_BLK2_IF];

    foreach(m_blk1_seqr[i])
       m_blk1_seqr[i] =
blk1_sequencer::type_id::create($sformatf("m_blk1_seqr[%0d]",i), null);
    foreach(m_blk2_seqr[i])
       m_blk2_seqr[i] =
blk2_sequencer::type_id::create($sformatf("m_blk2_seqr[%0d]",i), null);

  endfunction : new
...
endclass : blk_sequencer
```

Figure 17. Example of higher level virtual sequencer class

4.4 Virtual sequences

One of the most important components of the entire environment is the virtual sequence at different levels of the hierarchy. As previously discussed in other sections the design hierarchy is ideally followed by different verification components. This applies to the virtual sequences too. Why it is important to get the sequences right? Because they can easily allow access to create stimulus or different scenarios from higher levels of the hierarchy just by playing with switches in configuration class rather than doing any extra work. This creates a huge potential for productivity gains as the verification process moves from leaf level to higher levels or chip

development. The virtual sequences for any level will extends from the base class discussed in Section 3.4. The idea is to ease the access from higher level into the lower levels just by playing with the switches rather than having to add significant amount of code which duplicates the existing one done at leaf level. The first example in Figure 18 shows how to build a virtual sequence at leaf level.

```
typedef common_base_sequence extends common_base_phase_seq #(
                               blk1_env,
                               blk1_config,
                               blk1_env_config,
                               blk1_dut_config,
                               ,
                               blk1_sequencer,
                               dut_sys_ral,
                               blk1_if,
                               blk1_params_pkg::RST_NUM_ENB = 0)
blk1_base_phase_seq_t;

class sblk1_main_seq extends blk1_base_phase_seq_t;
  `uvm_object_utils(blk1_main_seq)

  blk1_agent_din_seq  m_din_seq;  // input data sequence

  function new(string name = "blk1_main_seq");
    super.new(name);
  endfunction

  virtual task pre_body();
    super.pre_body();
    m_din_seq = blk1_din_seq::type_id::create("m_din_seq");
    m_din_seq.seq_sz = m_env_config.m_blk1_agent_config[0].din_num;
  endtask

  virtual task body();
    super.body();
    if (m_env_config.m_blk1_agent_config[0].m_enable_in == 0) begin
      `uvm_info(get_type_name(),"blk1_main_seq NOT started", UVM_NONE)
      return;
    end
    `uvm_info(get_type_name(),"starting blk1_main_seq...", UVM_NONE)
    m_din_seq.start(m_env.m_agenv.m_agents[0].m_drv_seqr);
    `uvm_info(get_type_name(),"sblk1_main_seq completed", UVM_NONE)
  endtask

endclass: blk1_main_seq
```

Figure 18. Example of a leaf level virtual sequence

This is a standard implementation of a driver sequence started in the main sequence of a leaf level environment. It can be seen from the example that the drive sequence extends the base sequence described in Section 3.4. It starts only when this particular block has enable_in set which means driver generation is enabled from either the test or higher level. This virtual sequence can now be started at higher levels depending on the switches controlled from these levels of the hierarchy via env_config class. For example: the chip level test can enable a leaf level driver by enabling the immediate sequence at the sub-system level. This level can also enable the sequence at block level and so on. The block level sequence will start the agent sequence which starts the driver. The following example in Figure 19 shows the user how to create a higher level virtual sequence which controls sequences at lower level.

```
class blk_main_seq extends blk_base_phase_seq_t;
  `uvm_object_utils(blk_main_seq)

  // add all block level sequences as members of this class
  blk1_main_seq       m_blk1_main_seq;
  blk2_main_seq       m_blk2_main_seq;

  function new(string name="blk_main_seq");
    super.new(name);
    m_blk1_main_seq = new;
    m_blk2_main_seq = new;
  endfunction : new

  task pre_body();
    super.pre_body();
    m_blk1_main_seq=blk1_main_seq::type_id::create("m_blk1_main_seq");
    m_blk2_main_seq=blk2_main_seq::type_id::create("m_blk2_main_seq");

  endtask

  task body();
     super.body();
    // start in paralell sub-block level sequences
    fork
    begin
      if(m_env_config.m_enable_blk1_in)
        m_blk1_main_seq.start(m_env.m_base_seqr.m_blk1_seqr);
    end
    begin
      if(m_env_config.m_enable_blk2_in)
        m_blk2_main_seq.start(m_env.m_base_seqr.m_blk2_seqr);
    end
    join_none
    wait fork;
  endtask
```

Figure 19. Example of a higher level virtual sequence

This example shows how to control a generic lower level subset of sequences from a generic higher level. Blk is the higher level and blk1, blk2 are the lower levels. Switches constrained at test level allow the choice of what sequence to be started at any level of the hierarchy. Eventually a driver sequence will be selected based on the leaf level switches. The higher level sequence is just a control sequence containing the lower level virtual sequences and so on. This is valid at any level of the hierarchy including the chip level.

4.5 UVM base test class

One of the most important components in the environment is the UVM base test class. This class is used in all tests for a particular level of the hierarchy: block, subsystem or chip level. Each level must have its own UVM base test class. On top of this in case of acceleration a separate UVM base test needs to be created for the subset of tests which one needs to accelerate. The common UVM class from which all these UVM base test classes extend has been described in detail in Section 3.5. The example in Figure 20 considers a block level which has two sub-blocks but it can be applied at any level of the hierarchy. This is a generic implementation of a block level UVM base test. As discussed in the previous sections the implementation is done in the common base class and it is inherited by every test at any level.

```
typedef  common_test_base#(blk_env,  blk_config, blk_env_config,
                           blk_dut_config, blk_env_api,  blk_reset_seq,
                           blk_config_seq, blk_main_seq, blk_shutdown_seq,
                           dut_sys_ral,  virtual blk_if,  "blk_config",
                           "ral_sys_dut",  "blk_itf")  blk_base_uvm_test_t;

class blk_base_uvm_test extends blk_base_uvm_test_t;
  `uvm_component_utils(blk_base_uvm_test)

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

  virtual function void test_config();
    // override any variable specific for this particulat level of hierarchy
    // this is a place to override constraints if needed
  endfunction

function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    m_config.m_blk1_config.itf = m_config.itf.sblk1_itf;
    m_config.m_blk2_config.itf = m_config.itf.sblk2_itf;
    m_config.itf.RAL = this.RAL;
endfunction : build_phase
endclass: blk_base_uvm_test
```

Figure 20. Generic UVM base test class for any level of the hierarchy

From the code example above, one can see the increased number of parameters to be used. This is needed to allow the user to have access to everything inside the tests and it is a good compromise as the base test is created only once. Also, the initialization the lower level interfaces has to be done inside build_phase() by passing the interfaces coming from the higher level. One important aspect of the UVM base test is the ability to override any constraint set by randomization of the config class inside the test_config() function.

4.6 Scoreboards

The scoreboards are used to compare data received from DUT with the expected data. In case of the block level environments, the expected data comes from C models. The C model gets the same input as DUT. The output from C is expected to be the same as the output from RTL. Figure 21 shows the standard scoreboard approach for a generic leaf level block used in the current project.

```
`uvm_analysis_imp_decl(_our_drv)
`uvm_analysis_imp_decl(_our_mon)

class blk_scbd extends uvm_scoreboard;
  uvm_analysis_imp_our_drv #(my_trans, our_scbd) ap_drv;
  uvm_analysis_imp_our_mon #(my_trans, our_scbd) ap_mon;

  extern function void write_our_drv(blk_trans tr);
  extern function void write_our_mon(blk_trans tr);
endclass: blk_scbd

class dsp_driver extends uvm_driver #(blk_trans);

  uvm_analysis_port #(blk_trans)   ap;
  virtual interface blk_agent_if  vif;

  task run_phase(uvm_phase phase);
    blk_trans tr;
    forever @vif.data_ready begin
      seq_item_port.get_next_item(tr);
      ap.write(tr);  // pass sent data to scoreboard
      seq_item_port.item_done();
    end
  endtask: run_phase

endclass: blk_driver

class blk_monitor extends uvm_monitor;

  uvm_analysis_port #(blk_trans)   ap;
  virtual interface blk_agent_if  vif;

  task run_phase(uvm_phase phase);
    blk_trans tr;
    forever @vif.data_valid begin
      tr.m_data = vif.data_out;
      ap.write(tr);  // pass received data to scoreboard
    end
  endtask: run_phase

endclass: blk_monitor
```

Figure 21. DSP scoreboard interactions with the driver and the monitor

Since there is no concept of time on the C side, C and RTL synchronization is needed. The synchronization is achieved by using RTL as a master and C as a slave. The RTL drives the C model

by calling C functions at some key points in RTL operation. The C functions are imported to SV side by DPI methods.

The scoreboard is instantiated in the blk_env class extended from common_env class. The same class also contains an instance of the agent which typically contains a driver and a monitor. Both, the driver as well as the monitor are connected to the scoreboard through a dedicated uvm_analysis_port. The code in Figure 22 shows how a typical scoreboard could be implemented and how it could be integrated with the environment.

```
import "DPI-C" function void data_in(input int drv_data [123]);

import "DPI-C" function void data_out(input  int channel,
                                      output int word [32]);

class blk_trans extends uvm_sequence_item;
  bit [255:0]    data_in [8];  // input per channel
  logic [509:0]  data_out;     // RTL output from monitor
endclass: blk_trans

class blk_env extends common_env;

  blk_agent_env  m_agnt;  // agent env
  blk_scbd       m_scbd;  // scoreboard

  function void build_phase(uvm_phase phase);
    m_scbd =  our_scbd::type_id::create("m_scbd", this);
    m_agnt = our_agent::type_id::create("m_agnt", this);
  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    m_agnt.m_drv.ap.connect(m_scbd.ap_drv);
    m_agnt.m_mon.ap.connect(m_scbd.ap_mon);
  endfunction: connect_phase

endclass: blk_env
```

Figure 22. Scoreboard implementation and integration with the environment

The synchronization works as follows: when a driver drives data onto RTL input, the scoreboard calls an appropriate C function to pass the same data to the C model. Similarly, when a monitor receives data from RTL output, the scoreboard calls another C function to get the output data from C model. At this time the scoreboard calls the checker which compares C data with RTL data. Any mismatch triggers an error.

4.7 Hierarchical interfaces

Similar with the rest of the environment components, interfaces used at every level must be hierarchical and follow the design hierarchy. This will simplify passing of the interfaces from the chip level down to the UVM base test by using only one interface rather than sending several of them via the uvm_config_db. Figure 23 shows an example of such an hierarchical interface used for a block or subsystem level. Chip level interface will contain all subsystem interfaces plus its own clock and reset interfaces. Based on the current approach the clock and reset interfaces present at every layer of the hierarchy are not reused at higher level since every level in the design has its own resets and clocks which may or may not include the ones used at lower levels.

```
interface blk_if();

  import blk_params_pkg::*;

  ...

  clk_if        clk_itf[NUM_CLK_IF]();
  rst_if        rst_itf[NUM_RST_IF](1'b0);
  sblk1_if      sblk1_itf[NUM_SBLK1_IF]();
  sblk2         sblk2_itf[NUM_SBLK2_IF]();
  gms_if        gms_itf[NUM_GMS_IF]();
  blk_misc_if   misc_itf(.clk(clk_itf[BLK_CLK].clk));
  ...
endinterface
```

Figure 23. Generic hierarchical interface for block/subsystem/chip level

As indicated in Figure 23 all these hierarchical interfaces will contain reset, clock and a processor interface (ie. gms_if) for block level processor access. Only the chip level interface is an exception. At chip level, the processor interface is different than the block/subsystem level. misc_if is also present at each level and this helps with HW acceleration in the future. As described in Section 4.8 the hierarchical interface is set into the toplevel HVL via the uvm_config_db and it is designed to be retrieved only from the UVM base test. This one as described in Section 3.5 we will embed it inside the config class and unrolling it to make it available to the lower level components using directional setting into uvm_config_db.

4.8 Keep it open for emulation

Since this UVM verification environment is targeted to a very complex and big ASIC it is expected that simulation at toplevel will be a challenge in terms of simulation speed. One of the solutions to solve this problem is to have an open mind about future emulation but the architecture of the verification environment has to be capable to accommodate the basic requirements of the emulation: time consuming and non-time consuming constructs have to be in separate top levels. To achieve this two top levels (HVL and HDL top levels) have been considered at the beginning and steps to move all the time consuming constructs into the synthesizable toplevel have been taken. For example test owners have the ability to use a task to introduce delays instead of using the standard repeat(n) @ (posedge clk) or #delay inside their sequences or time consuming constructs. The tasks considered are wait_clks and wait_ns. These should be part of the hierarchical interface structure for every level of the hierarchy. A special misc_if has been created to accommodate these requirements and other non-emulation requirements down the road.

Figure 24 shows how to build these tasks inside an interface.

```
interface onramp_misc_if(input clk);

  ...

  task wait_clks(int n_clk);
    repeat(n_clk) @ (posedge clk);
  endtask

  task wait_ns(int t_ns);
    #(t_ns*1ns);
  endtask

endinterface
```

Figure 24. misc_if created to leave door open to emulation

At any point in a sequence the test can use itf.misc_itf.wait_clks(10) or itf.misc_itf.wait_ns(1000) to introduce the necessary delay rather than use the standard delay procedures which are non-synthesizable and incompatible with a hardware acceleration box.

Creating two top levels is explained in the next two examples. Figure 25 shows the verification toplevel which contains all non-synthesizable constructs and code which doesn't move the time.

```
module blk_hvl_top;

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

  ...
  initial begin
...
    // setup common interface
    uvm_config_db#(virtual blk_if)::set(null, "uvm_test_top", "blk_itf",
blk_hdl_top.itf);
   ...
    `uvm_info(modName, "run_test() ---- start ----", UVM_NONE)
    run_test();
    `uvm_info(modName, "run_test() ---- done ----\n", UVM_NONE)
  end
endmodule: blk_hvl_top
```

Figure 25. Generic HVL toplevel

As seen in Figure 25 all non-synthesizable constructs non-time consuming are present in this toplevel. Uvm_config_db interface setup must be added in this toplevel since it's an UVM construct. However, since it deals with the hierarchical interface which is defined in the HDL toplevel it uses the hierarchical approach to access it.

```
module blk_hdl_top;
...
  blk_if     itf();
  blk_model  model( itf );
  blk_dut    dut
  (
    .i_clk    (itf.clk_itf[blk_params_pkg::BLK_CLK].clk),
    .i_rst_n  (itf.rst_itf[blk_params_pkg::BLK_RST].async_n_rst),
...
  );
...
endmodule: blk_hdl_top
```

Figure 26. Generic HDL toplevel

Figure 26 shows a generic example of a synthesizable toplevel which instantiates the DUT of the block/sub-system or chip level along with the model and the hierarchical interface. All these three entities use synthesizable constructs. It is important to keep verification constructs separated from the synthesizable ones at all times. Not only it helps with a future acceleration but it is a cleaner approach to begin with.

4.9 Packages

There are two reasons for using packages as building blocks for our environment.

Encapsulation of commonly used data types. Our blocks often use similar or even identical names for variables, tasks or functions that refer to similar functionality. Without packages we would have to deal with name conflicts at top level where all blocks are integrated and compiled together.

The other reason for using packages is the ease of sharing parameters, data types and variables among multiple modules. This is most useful for global items that would be awkward to distribute through the hierarchy.

This example shows a package that holds all global parameters used by a given environment. We use one package of this type per block level environment.

```
package block_params_pkg;

  parameter NUM_CLK_IF  = 3;
  parameter type clk_t = enum { CLK_IN      // 0
                                CLK_OUT,    // 1
                                CLK_SWIF }; // 2
  parameter NUM_RST_IF  = 2;
  parameter type rst_t = enum { HARD_RST,   // 0
                                SWIF_RST }; // 1
  parameter CLK_IN   = 1234; // [ps]
  parameter CLK_OUT  = 2345; // [ps]
  parameter CLK_SWIF = 1000; // [ps]

endpackage: block_params_pkg
```

Figure 27. Example of parameter package

Another example shows a package that encapsulates specific component of the environment (i.e. sequence package). We use multiple packages of this type to build block level environment.

```
package block_seq_pkg;

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

  import sys_ral_pkg::*;
  import block_params_pkg::*;
  import block_config_pkg::*;
  import block_agent_pkg::*;
  import rst_pkg::*;

  `include "block_config_seq.sv"
  `include "block_main_seq.sv"
  `include "block_reset_seq.sv"
  `include "block_shutdown_seq.sv"

endpackage
```

Figure 28. Example of encapsulation package

4.10 Interface registration reuse

The interfaces defined at block level are reused at subchip level and of course at chip level. The reuse is achieved by putting interface registrations and HVL top module in separate files. The HVL top module is different at each level (block, sub-system, or chip). The one and only file containing interface registrations is reused at each level, by being included in the corresponding top module.

Let's look at the interface registration file. The first line defines an enum variable using parameter type "clk_t" from block_params_pkg package.

Note that the multiple interfaces (e.g. 3 clocks) are registered in uvm_config_db using generate block. While singular interfaces are registered within an initial block.

```
block_params_pkg::clk_t  block_clk_e;
   for (genvar j = 0; j < block_params_pkg::NUM_CLK_IF; j++) begin
     initial begin
       uvm_config_db#(virtual clk_if)::set(null,
          {"uvm_test_top.", `block_env_path, ".m_clk*"},
          $sformatf("clk_vif[%0d]", block_clk_e),
                           `block_itf_path.clk_itf[j]);
       `uvm_info(modName, $sformatf("vif set for clk[%0d]
                          %0s interface", j, block_clk_e.name))
       block_clk_e = block_clk_e.next;
     end
   end
   initial begin  //register agent interface
     uvm_config_db#(virtual block_agent_if)::set(null,
        {"uvm_test_top.", `block_env_path, ".m_agenv*"},
         "block_agent_vif[0]", `block_itf_path.block_agent_itf);
     `uvm_info(modName, "vif set for block_agent_if interface")
   end
```

Figure 29. Example of interface registration file

The key elements that support reuse of the above file are two defines: `block_env_path and `block_itf_path. Let's look at two top level files where the above file is included. At block level, the defines point to simple, one level hierarchy.

```
   module block_hvl_top;
...
     `define block_env_path "m_env"
     `define block_itf_path block_hdl_top.itf
     `include "block_itf_registration.sv"
...
   endmodule
```

Figure 30. Example of top module at block level

However, at top level, the same defines point to a different (multi-level) hierarchy.

```
   module top_hvl_top;
...

     `define block_itf_path top_hdl_top.itf.block_itf
     `define block_env_path "m_env.m_block_env"
     `include "block_itf_registration.sv"

     // this is followed by a long list of defines
     // and includes of interface registrations from
     // all the other blocks below the top level
...
   endmodule: top_hvl_top
```

Figure 31. Example of top module at chip level

Only one top level file is compiled as part of the test environment. Whatever it is, chip level or block level, the same interfaces will be driven or monitored.

5. Challenges

Obviously, creating any UVM architecture comes with challenges. The main one we faced was to explain why this architecture is better than others and why this choice. Well, this is one architectural choice among multiple other UVM choices suggested by vendors or others. We believe it better fits the type of devices built by Ciena. The architectural choice considerably reduced the verification time overall by allowing verification to start at virtually every level of the hierarchy with just basic components in place at the leaf level. Is this the best choice for everyone? It may or may not be but we came with a proposal which worked for Ciena and decided to share the experience.

While stitching together higher level of hierarchy we found that, sometimes, a significant number of files has to change to accommodate the stitching. This includes packages and manifest files needed to have the hierarchical infrastructure up to date. However, we found that most of the challenges came from trying to maintain the hierarchical approach in non-class based components of the environment (ie. modules, interfaces). This is an area we'll have to improve for the next generations.

6. Conclusions

The evolution from a VMM hierarchical architecture to an UVM based one was an important milestone at Ciena. We'd like to specify just a few of the UVM contributions which made the architecture perform ant and significantly increased the productivity overall: the generic uvm_env base class in UVM allows a virtually unlimited number of components on a hierarchy compared with VMM, the configuration database in UVM gives enormous flexibility in terms of decoupling the components and least but not last the factory allows an elegant approach to achieve reusability and flexibility at any level of the hierarchy in any component of the verification environment.

All above on top of using a hierarchical structure for all the UVM components plus the interfaces resulted in a significant reduction of the time to market by reducing the verification effort. The architectural choice demonstrated that by increasing the verification team's size the productivity continued to increase compared with other architectural solutions in the past which demonstrated some limits of productivity gains due to increased interaction between team members working in the same area at the same time.

Overall, this is an architectural approach which is recommended for large and complex ASICs involving a large verification team. The proposal works well for smaller projects (ie. FPGAs) especially if the base classes are carefully crafted to meet the needs of the project.

- **Future work**

After going from the theory to the implementation process and a quick and successful tapeout we realized that the architecture can be made even more generic to fit any device: FPGAs or ASICs. In this respect we decided to develop one more level of project specific classes to fit in-between the block/subsystem level and common base classes. These classes are named after the project and they will give verification primes even more flexibility to develop project specific reusable components which are not necessarily reused in other devices (ie. between FPGAs and ASICs). The common base classes will have to further be reduced to strictly contain common elements to a larger category or devices (ie. resets, clocks). The rest of the elements will be pushed in the project specific category of classes.

7. Acknowledgments

The authors would like to thank the entire Ciena ASIC verification team for the feedback given during the process of implementing this new methodology and architecture. We also greatly appreciate the management support to implement the architecture.

8. References

[1]. Architecting Your Way To Acceleration In UVM, Paul Lungu, Dean Justus, SNUG Canada 2015