UVM Tips and Tricks

by Sandeep Nasa and Shankar Arora, Logic Fruit Technologies Pvt. Ltd.

UVM is the most widely used Verification methodology for functional verification of digital hardware (described using Verilog, SystemVerilog or VHDL at appropriate abstraction level). It is based on OVM and is developed by Accellera. It consists of base libraries written in SystemVerilog which enables the end user to create testbench components faster using these base libraries. Due to its benefits such as reusability, efficiency and automation macros it is a widely accepted verification methodology.

UVM has a lot of features so it's difficult for a new user to use it efficiently. A better efficiency can be obtained by customizing the UVM base library and applying certain tips and tricks while building UVM testbenches, which is mainly the purpose of this article.

The Aim of this Article:

1. Focus on the common mistakes made by the novice engineers or experienced users while working on UVM Methodology.
2. Tricks to enhance the productivity using UVM Methodology.
3. Conventions for using UVM methodology features.
4. Common hierarchy giving well defined architecture which is easy to understand and manage.

Most of the engineers which are new to UVM or have RTL experience may not be able to create efficient and productive testbenches due to unfamiliarity with the OOPS concepts, UVM base class library and UVM verification environment architecture.

This article will furnish several examples to improve the performance of the UVM testbench by applying different optimizing techniques to random access generation, configuration database, objection mechanism, sequence generation, loop usage.

INTRODUCTION

The rise in level of complexity of the chips due to the addition of more features has direct impact on the level of abstraction at which the chips are designed and moreover on the verification of these chips which consumes almost 70 percent of the time to verify these chips.

Therefore, there is a requirement for a common verification platform which can provide standard structure, and standard base libraries with features such as reusability, simplicity and easy to understand structure. UVM methodology fulfills all these requirements and is universally an accepted common verification platform.

This is the reason why UVM is being supported by major vendors (Mentor®, Synopsys® and Cadence®) which is not the case with the other verification methodology developed thus far.

All aims mentioned above are explained in detail below.

1. COMMON UVM MISTAKES AND THEIR RESOLUTION WITH TIPS AND TRICKS:

1.1 Use of Macros to Overcome Errors Faced in Package:

In case of UVM project, source files are added in the packages by using `include directive. So, in bigger projects there might be a condition when two packages might include the same files and these packages may be included in the same scope. These may result in compile time errors. So, to overcome such a scenario, it is necessary to make proper use of compilation directives such as `ifndef, `ifdef and macros such as `define.

For example: consider a testbench that includes several files as shown below:

```
1. package      pkg1;
                    `include "adder_design.sv"
                    `include "tb.sv"
               endpackage
2. package pkg2;
                    `include "adder_design.sv"
                    `include "adder_tb.sv"
               endpackage
3.  module top()
                    import pkg1::*;
                    import pkg2::*;
               endmodule
```

So, in this case we observe that both the packages contain the same file adder_design.sv. Hence, the compilation of the top module may result in compilation error -> "multiple times declaration" since the adder_design file is included twice and code is duplicated.

Therefore, to prevent this error we need to write file adder_design.sv as shown below:

```
`ifndef ADDER_DESIGN
     `define ADDER_DESIGN
     ......... adder_design logic .........
`endif
```

So, in this case before including code we are specifically checking if ADDER_DESIGN is defined or not (by using `ifndef macro), if not defined only then define it by using (`define) macro and add the adder code. This overcomes the error encountered in Example 1 since at the time of compiling package pkg2, it will find that ADDER_DESIGN parameter was already defined. Therefore, it won't add the code present in this file again and overcome multiply defined error.

In addition it is recommended to place the most frequently used files inside a package and then import it inside other components. This is more efficient than using `include for file inclusion in components separately, because code inside the package will compile only once but code added using `include will compile as many times as it is present. Moreover, it is much easier to refer only to selected parameters, functions and tasks from packages as compared to using `include file directives, which cause all file contents to be added and compiled, even if we never use them.

1.2 Use of Fork-join_none Inside for Loop

Sometimes we are getting issues while using fork-join in for loop, the issue along with resolution is explained below with example.

Problem: Whenever a fork-join_none block is used inside a for loop for executing several threads in parallel and takes loop parameter as an input, the generated result holds only the last value of the variable. For example if the code along with its simulated output is as written below:

```
module top;
    initial begin
        for (int i = 0; i < 4; i++)begin
            fork
              display(i);
            join_none
        end
        task display(int i);
         $display("i = %d", i);
        endtask
endmodule
        Output:   i = 4
                  i = 4
                  i = 4
                  i = 4
```

The code above gets successfully compiled but does not produce expected results since it will only print the last value of "i" i.e. 4 for each iteration of the loop. The reason for this problem is given in the SystemVerilog LRM Section 9.6 which states that *"The parent process continues to execute concurrently with all the processes spawned by the fork. The spawned processes do not start executing until the parent thread executes a blocking statement."*

Solution: The solution to the given problem is also suggested in the LRM which states that *"Automatic variables declared in the scope of the fork…join block shall be initialized to the initialization value whenever execution enters their scope, and before any processes are spawned. These variables are useful in processes spawned by looping constructs to store unique, per-iteration data."*

Therefore, in this case the variable l is declared as automatic and for every loop iteration, a new value is allocated to variable l and passed to the respective task.

The modified code along with output is as shown below:

```
module top;
    initial begin
        for (int i = 0; i < 4; i++)
            fork
              automatic int l = i;
              display(l);
            join_none
        end
        task display(int i);
            $display("i = %d", i);
        endtask
endmodule
        Output:   i = 0
                  i = 1
                  i = 2
                  i = 3
```

1.3 Improving the Randomization Techniques

Various scenarios are explained below for improving randomization.

a. SystemVerilog UVM sequence generates interesting scenarios by randomizing and constraining the data items of the transaction class.

Generally, the constraints are specified in the transaction item class. SystemVerilog also allows you to add in-line constraints in the sequence body, by using the `randomize() with `construct.

These in-line constraints will override the transaction class constraints and provide one more control from top level. The following example shows a common mistake and its resolution, as shown on the following page:

```
// Transaction class
class transaction extends uvm_sequence_item;
    rand [31:0] addr;
    rand [31:0] data;
endclass
// Sequence class
class seq extends uvm_sequence#(seq_item);
   bit [31:0] addr;
   task body();
       transaction trans;
       bit [31:0] addr = 32'h11001100;
       assert(trans.randomize() with {  trans.addr == addr; });
   endtask
endclass
```

Here the "with" feature is used to override the address but instead the code generates a transaction with addr as hbfdf5196. So why is the intended value of trans.addr not applied with the inline trans.addr == addr constraint.

The problem arises when you try to make the transaction item address equal to the address in the calling sequence class using the above in-line constraint. The result is undesirable since the constraint will actually cause the seq_item address (trans.addr) to be equal to itself. This gotcha in SystemVerilog arises because we have addr as a variable defined in both the transaction class as well as the sequence class. SystemVerilog scoping rules pick the variable which is part of the object being randomized.

The SystemVerilog P1800-2012 LRM states that:

"Unqualified names in an unrestricted in-lined constraint block are then resolved by searching first in the scope of the randomize() with object class followed by a search of the scope containing the method call –the local scope."

In order to overcome the above problem we can prefix `local::` before the address of sequence class seq. Thus, we could modify the code as:

```
// Transaction class
class transaction extends uvm_sequence_item;
        rand [31:0] addr;
        rand [31:0] data;
endclass
// Sequence class
class seq extends uvm_sequence#(seq_item);
        bit [31:0] addr;
     task body();
         transaction trans;
         bit [31:0] addr = 32'h11001100;
         assert(trans.randomize()
              with { trans.addr == local::addr; });
      endtask
endclass
```

Now with these modifications above code generates the following address:

```
# Name    Type          Size    Value
# trans   transaction   -       @636
# addr    integral      32      'h11001100
```

This use of "local::" makes sure that the constraint solver looks for the address following the local:: only in the local scope (i.e. the address in the sequence class seq). So, now the constraint will be the desired one which states that while randomizing the address of the transaction class, the constraint solver should make sure that the address of the transaction should be equal to the address in the sequence seq.

```
class Ethernet;
   rand bit [3:0]payload[];
   task display();
      $display("Randomize Values");
      $display("-------------------------------");
      $display("payload : %p",payload);
      $display("-------------------------------");
   endtask
```

```
endclass
module test();
   Ethernet Eth;
      initial begin
         repeat(2)
            begin
              Eth=new();
              assert(Eth.randomize());
              Eth.display();
              end
        end
 endmodule
```

b. Dynamic arrays declared as rand can be a source of error for some new users. It won't randomize the size of dynamic array when we try to randomize it based on how the code is written. Consider the example of an Ethernet packet.

```
class Ethernet;
    rand bit[3:0]payload[];
    constraint c{ payload.size() ==4;}
     task display();
        $display("Randomize Values");
        $display("-------------------------------");
        $display("payload : %p",payload);
        $display("-------------------------------");
      endtask
endclass
module test();
    Ethernet Eth;
       initial begin
         repeat(2)
            begin
              Eth=new();
              assert(Eth.randomize());
              Eth.display();
            end
         end
endmodule
```

Per the expectation, the call to randomize() must generate random values for the payload array. But unfortunately, this doesn't happen. Instead, the randomize call will exit with no error, warning. The payload array has no value.

Results:

```
Randomized Values
-------------------------------------------------------------
payload : '{}
-------------------------------------------------------------
Randomized Values
-------------------------------------------------------------
payload : '{}
-------------------------------------------------------------
```

The solution to this issue is that the size of the dynamic array or queue declared as rand should also be constrained.

With the addition of constraint the result of the compilation is as below:

```
Randomized Values
-------------------------------------------------------------
payload : '{'he, 'h4, 'h4, 'h8}
-------------------------------------------------------------
Randomized Values
-------------------------------------------------------------
payload : '{'h6, 'he, 'h5, 'h3}
-------------------------------------------------------------
```

c. It's very important to check the return value of the randomize() function while applying this function on the object of the transaction item type. The randomize() function returns 1 if all the rand variables attain a valid value otherwise it returns zero. It is very important to check whether the randomization is successful or failed. In case randomization is not successful (due to invalid constraints or any other reason) then its rand variables will hold the previous values. But it's always recommended that we should check the randomization using assert statement instead of using if statement because the use of assert statement makes sure that the simulation gets terminated when randomization fails.

For example:

```
class Ethernet;
          rand bit [47:0] src_addr = 4'h5;
          rand bit [47:0] dest_addr;
          constraint c{src_addr > 48'h4;}
          constraint c1{src_addr == 48'h4;}
          task display();
            $display("Randomize Values");
            $display("-----------------------------");
            $display("src_addr : %p",src_addr);
            $display("-----------------------------");
          endtask
        endclass
module test();
     Ethernet Eth;
        initial begin
          repeat(2)
            begin
              Eth=new();
              Eth.randomize();
              Eth.display();
            end
        end
 endmodule
```

In the code above we are not checking whether the return value of the randomize() function (for the Ethernet packet) is correct or not. Here is the result of randomization:

```
Randomized Values
------------------------------------------------------------
src_addr :                  5
------------------------------------------------------------
```

In this case, we can see that the source address holds the value of 5 and does not follow the constraint. Therefore, it is necessary to detect whether the randomized value matches the constraint or not but here no message is printed. So it's important to check whether the randomized Ethernet packet satisfies the source address constraint for the Ethernet packet or not. This can be done either by using an if statement or by using an assertion.

The solution with if statement is as below:

```
class Ethernet;
   rand bit [47:0] src_addr = 4'h5;
   rand bit [47:0] dest_addr;
   constraint c{src_addr > 48'h4;}
   constraint c1{src_addr == 48'h4;}
   task display();
       $display("Randomize Values");
       $display("-----------------------------");
       $display("src_addr : %p",src_addr);
       $display("-----------------------------");
   endtask

endclass

module test();
   Ethernet Eth;
     initial begin
       repeat(2)
         begin
           Eth=new();
           If (Eth.randomize())
                $error("Randomization failed");
           Eth.display();
          end
        end
 endmodule
```

The result of this code is as below:

```
Error: "testbench.sv", 23: test: at time 0 ns
Radomization failed
Randomized Values
------------------------------------------------------------
src_addr :                  5
------------------------------------------------------------
```

The solution with using assert for checking the successful randomization is as shown here:

```
class Ethernet;
   rand bit [47:0] src_addr = 4'h5;
   rand bit [47:0] dest_addr;
   constraint c{src_addr > 48'h4;}
   constraint c1{src_addr == 48'h4;}
    task display();
       $display("Randomize Values");
       $display("-----------------------------");
```

```
        $display("src_addr : %p",src_addr);
        $display("-----------------------------");
     endtask
 endclass

 module test();
    Ethernet Eth;
      initial begin
        repeat(2)
          begin
            Eth=new();
            assert(Eth.randomize());
            Eth.display();
       end
     end
 endmodule
```

So, in this case we get an error message that randomization failed and simulation stopped.

```
//Results generated by the previous code
Error-[CNST-CIF] Constraints inconsistency
failure testbench.sv, 22
    Constraints are inconsistent and
    cannot be solved. Please check
    the inconsistent constraints being
    printed above and rewrite them.

    "testbench.sv", 22: test.unnamedSS_3.unnamedSS_2
            started at 0ns failed at 0ns
            Offending 'Eth.randomize()'
     Randomize Values
.....................................................................
      src_addr:          5
.....................................................................
      Time: 0ns
```

d. Some common issues related to random variables are with variables defined as randc, but the generated random results are not perfectly cyclic because of the constraint applied on them.

For example: Consider the following code:

```
class Ethernet;
   randc bit [1:0] a1;
   randc bit [6:0] b1;
   constraint c{(a1 != 2'b01) -> (b1<7'h10);
                (a1 == 2'b01) -> (b1>=7'h10);}
         task display();
            $display("Randomize Values");
            $display("-----------------------------");
            $display("a1 : %p",a1);
            $display("b1 : %p",b1);
            $display("-----------------------------");
          endtask
endclass
      module test();
          Ethernet Eth;
               initial begin
                 repeat(2)
                   begin
                     Eth=new();
                    assert(Eth.randomize());
                          Eth.display();
                  End
end
endmodule
```

The result of compilation of this code will give an error as mentioned below:

a1 = 0, b1 = 0

In this particular scenario, the problem occurs due to the conflict between constraint solving and cyclic randomization. This is also tool dependent. So, if the tool wants to solve one of the variables first, it has to compromise with the cyclic nature of the randc type of the variable.

So, in this case there are two options:

- Either to compromise with the intended cyclic behavior (as previous results)
- The solution to the above problem is to make sure that there is no conflict between the generated randomized values (which can be attained by removing constraint)

The code with removed constraint is as below:

```
class Ethernet;
   randc bit [1:0] a1;
   randc bit [6:0] b1;
         task display();
            $display("Randomize Values");
            $display("------------------------------");
            $display("a1 : %p",a1);
            $display("b1 : %p",b1);
            $display("------------------------------");
         endtask
endclass
```

In this code we have removed the constraint which won't conflict with the cyclic behavior of randc variables. This is the only solution to overcome this issue.

2. TRICKS TO ENHANCE THE PRODUCTIVITY USING UVM METHODOLOGY

2.1 Avoid using uvm_config_db for Replicating the Changes between Components

The "uvm_config_db" should not be used to communicate between different components of the testbench when the number of variables becomes too much. Instead it's better to have a common object. By doing this we can avoid the calling of get and set functions and improve efficiency. For example, setting a new variable value inside one component and getting it inside a in another.

For example: A less efficient way is shown below:

Producer component

```
int id_value = 0;
   forever begin
    `uvm_config_db#(int)::set(null,"*", "id_value", id_value);
      id_value++;
end
```

Consumer component

```
int id_value = 0;
   forever begin
`uvm_config_db#(int)::wait_modified(this,"*","id_value");
if(!uvm_config_db#(int)::get(this,"","id_value", id_value) begin
     `uvm_error(.......)
     end
end
```

The more efficient way is as mentioned below:

Creation of config object

```
  //config object containing id_value field
packet_info_cfg pkt_cfg = packet_info_cfg::type_id::create
                                   ("pkt_info");
//This created in the producer component and the consumer
component has a handle to the object.
```

Producer component

```
//In the producer component
    forever begin
       pkt_info.id_value = id_value;
       id_value++;
    end
```

Consumer component

```
//In the consumer component
     forever begin
        @(pkt_info.id_value);
          //Code to containing the logic using new id_value --------
     end
```

The above code results in higher performance due to absence of the get() and set() calls used in the uvm_config_db along with the use of the expensive wait_modified() method. In this case, both the consumer and producer share the handle to the same object, therefore any change made to the id_value field in the producer becomes visible to the consumer component via handle.

2.2 Minimize Factory Overrides for Stimulus Objects

Using UVM factory provides an override feature where an object of one type can be substituted with an object of derived type without changing the structure of the testbench. This feature could be applied to change the behavior of the generated transaction without modifying the testbench code. This override results in extended lookup in the factory each time the object gets created.

```
// Low performance code
class genrate_seq extends uvm_sequence#(seq_item);

  task body;
    seq_item item;
      repeat(200) begin
           item = seq_item::type_id::create("item");
           start_item(item);
           assert(item.randomize());
           finish_item(item);
   endtask

endclass
```

Therefore, to minimize the costly impact of this factory overriding, first create an object and then clone it each time it is used to avoid the use of factory.

```
//High performance code
class generate_seq extends uvm_sequence#(seq_item);
   task body;
seq_item orig_item =seq_item::type_id::create("item");
       seq_item item;

       repeat(200) begin
            $cast(item, orig_item.clone());
                start_item(item);
                assert(item.randomize());
                finish_item(item);
     endtask
```

2.3 Avoid the Use of uvm_printer Class

Initially, the uvm_printer class was designed to be used with uvm_field_macro in order to print the component hierarchy or transaction fields in several formats. This class comes with performance overhead.

```
//Low performance code
seq_item req = seq_item::type_id::create("req");

repeat(20) begin
     start_item(req);
     assert(req.randomize());
     finish_item(req);
     req.print();
end
```

This performance overhead can be avoided by using convert2string() method for objects. The method returns a string that can be displayed or printed using the UVM messaging macros.

```
//High Performance Code
seq_item req = seq_item::type_id::create("req");

repeat(20) begin
start_item(req);
assert(req.randomize());
finish_item(req);
 `uvm_info("BUS_SEQ", req.convert2string(), UVM_DEBUG)
end
```

2.4 Minimize the Use of get_register() or get_fields() in UVM Register Code

The call to get_register() and get_fields() methods returns queues of object handles where queue is an unsized array. When these methods are called, they result in these queues getting populated which can be an overhead if the register model is of reasonable size. It is not worthy to repeatedly call these methods. So they should be called once or twice within a scope.

```
//Low performance code
    uvm_reg reg_i[$];
    randc i;
    int regs_no;

repeat(200) begin
      reg_i = decode.get_registers();
      regs_no = regs.size();
      repeat(regs_no) begin
            assert(this.randomize());
            assert(reg_i.randomize());
            reg_i[i].update();
      end
end
```

In the code above get_registers is called inside the loop which is less efficient.

```
//High Performance Code
   uvm_reg reg_i[$];
   randc i;
   int regs_no;

   reg_i = decode.get_registers();


 repeat(200) begin
      reg_i.shuffle();
      foreach(reg_i[i]) begin
            assert(reg_i[i].randomize());
            reg_i[i].update();
      end
    end
```

In efficient code, call to get_registers is kept outside the repeat loop, so that only one call is made to get_registers() and avoids the overhead associated with the repeated call.

2.5 Use of UVM Objections

UVM provides an objection mechanism to allow synchronization communication among different components which helps in deciding when to close the test. UVM has built-in objection for each phase, which provides a way for the components and objects to synchronize their activity.

Objections should only be used by the controlling threads, and it is also very necessary to place the objections in the run-time method of the top level test class, or in the body method of a virtual sequence. Using them in any other place is likely to be unnecessary and also cause a degradation in performance.

```
//Low Performance code
class sequence extends uvm_sequence#(seq_item);
  //..................
   task body;
     uvm_objection objection = new("objection");
 seq_item item = seq_item::type_id::create("item");
     repeat(5) begin
          start_item(item);
          assert(item.randomize());
          objection.raise_objection(this);
          finish_item(item);
          objection.drop_objection(this);
     end

sequencer seqr;

   task body;
sequence seq = sequence::type_id::create("seq");
     seq.start(seqr);
   endtask
```

The code above is less efficient since the objection is raised per sequence_item.

The high performance code is given below.

```
//High Performance code
class sequence extends uvm_sequence#(seq_item);

task body;
seq_item item = seq_item::type_id::create("item");
repeat(5) begin
   start_item(item);
   assert(item.randomize());
   finish_item(item);
 end

 sequencer seqr;

task body;
 `uvm_objection objection = new("objection");
  sequence seq=   sequence::type_id::create("seq");
  objection.raise_objection(seqr);
  seq.start(seqr);
  objection.drop_objection(seqr);
endtask
```

In this code, the objection is raised at the start of the sequence and dropped at the end, therefore enclosing all the seq_items sent to the driver.

2.6 Tip: Loop Performance Optimization

a) The performance of a loop depends on the work that is done inside the loop

b) The checks in the conditional portion of the loop to determine whether it should continue or not

Therefore, it's recommended to keep the work within the loop to a minimum, and the checks that are made on the loop bound should have a minimum overhead.

For example: consider the dynamic array

```
//Less efficient code
        int arr[];
        int total = 0;

        for(int i = 0;i< arr.size();i++) begin
        total += arr[i];
        end
```

This code above is not very efficient since the size of the array is calculated during each iteration of the loop.

The efficiency of the code can be improved by calculating the size of the array outside the loop and assigning it to a variable which is then checked in the conditional portion of the loop.

```
//High Performance Code
        int arr[];
        int arr_size;
        int tot = 0;
        arr_size = arr.size();
        for(int i = 0; i < arr_size; i++) begin
            tot += arr[i];
        end
```

So, in this case the size of the array is not calculated during every iteration of the loop. Instead it's calculated before the starting of the loop.

2.7 In uvm_config_db set() or get() Method Calls, Use Specific Strings

The regular expression algorithm used for search, attempts to find the closest match based on the UVM component's position in the testbench hierarchy and the value of the key string. The use of wildcards in either the set() or get() function call results in unnecessary overhead. When the context string is set to "*", this means that the entire component hierarchy will be searched for uvm_config_db settings before returning the result:

```
//Low Performance Code
sb_cfg = sb_config::type_id::create("sb_cfg");
uvm_config_db#(sb_config)::set(this, "*", "*_config", sb_cfg);

//In the env.sb component
sb_config cfg;
if(!uvm_config_db#(sb_config)::get(this," ", "_config", cfg)) begin
   `uvm_error(....)
end
```

More efficient code:

```
sb_cfg = sb_config::type_id::create("sb_cfg");

uvm_config_db#(sb_config)::set(this, "env.sb", "sb_config",
sb_cfg);

In the env.sb component
sb_config cfg;
if(!uvm_config_db#(sb_config)::get(this, " ", "sb_config", cfg))
begin
   `uvm_error(..........)
end
```

Here in efficient code, we used particular hierarchy compared to "*" in less efficient code for scope of set parameter/object.

2.8 Use the Testbench Package to Pass Virtual Interface Handles

Reduce the number of virtual interface handles passed via uvm_config_db from the TB module to the UVM environment. Generally uvm_config_db can be used to pass virtual interfaces into the testbench. But it is recommended to minimize the number of uvm_config_db entries.

```
module top;
import uvm_pkg::*;
import test_pkg::*;
ahb_if AHB();
apb_if APB();
initial begin
  `uvm_config_db#(virtual ahb_if)::set("uvm_test_top", " ", "AHB",
AHB);
  `uvm_config_db#(virtual apb_if)::set("uvm_test_top", " ", "APB",
APB);
   run_test();
end
class test extends uvm_component;

  ahb_agent_config ahb_cfg;
  apb_agent_config apb_cfg;

function void build_phase(uvm_phase phase);
     ahb_cfg = ahb_agent_config::type_id::create("ahb_cfg");
     if(!uvm_config_db#(virtual ahb_if)::get(this, " ", "AHB", ahb_
cfg.AHB)) begin
`uvm_error("build_phase", "AHB virtual interface not found in
uvm_config_db")
end
endfunction
endclass
```

Efficient code below:

```
package tb_if_pkg;

  virtual ahb_if AHB;
  virtual apb_if APB;

endpackage

class test extends uvm_component;

   ahb_agent_config ahb_cfg;
   apb_agent_config apb_cfg;

function void build_phase(uvm_phase phase);
    ahb_cfg = ahb_agent_config::type_id::create("ahb_cfg");
    ahb_cfg.AHB = tb_if_pkg::AHB;
    apb_cfg = apb_agent_config::type_id::create("apb_cfg");
    apb_cfg.APB = tb_if_pkg::APB;
 endfunction
endclass
```

The second example shows how a shared package passes the virtual interface handles from the top level testbench module to the UVM test class. The uvm_config_db::set() and get() calls get eliminated and also the entry from uvm_config_db for each virtual interface handle got eliminated. When the virtual interface handles are used more, a significant improvement in the performance is observed.

3. CONVENTIONS FOR USING UVM METHODOLOGY FEATURES

a. It is a good practice to set the variables used in different files to be declared in a single file by using `define macro so that they can be referred by that name and moreover any update in the value will be changed only in that file in which the variable is defined and the change will be reflected in all the files.

For example: consider an Ethernet packet which has several fields of different size but the size of some of the fields are fixed (except payload field). Suppose initially we set the size of the data field to some fixed size:

```
Like: Preamble is of 8 bytes.
      Destination Address is of 6 bytes
      Source Address is of 6 bytes
      Type field is of 2 bytes
      Data field is of 100 bytes
      CRC is of 4 bytes
```

And when we go deep into the hierarchy of the Ethernet packet, we can explore further fields deep in the hierarchy of the Ethernet packet some of them with the same size.

So we can define the size of all the fields in one file. So, in the other files which needs to refer to the size of these fields we can just refer them by their names.

```
For example:  `define Pream_size        64
              `define Dest_addr_size    48
              `define Sour_addr_size    48
              `define type_field_size   16
              `define data_size         800
              `define crc_size          32
```

Now suppose we need to generate packets of data with size 200 bytes. So, instead of making changes in all the files referring to the size of the data, we can just change it in the file where we defined the data size.

```
`define data_size         1600
```

Now the Ethernet packets will have data of size 200 bytes (1600 bits) and all the files referring to the data size will be automatically updated with the data size 200 bytes.

b. All the enum fields should be placed in a separate file.

For example: consider the examples of an open source libtins. Since we can put different types of packets in the Ethernet packet, we can define them in a single file and then refer them as required in other files.

```
Tins::PDU* pdu_from_flag(PDU::PDUType type, const uint8_t*
buffer, uint32_t size) {
  switch(type) {
    case Tins::PDU::ETHERNET_II:
      return new Tins::EthernetII(buffer, size);
    case Tins::PDU::IP:
      return new Tins::IP(buffer, size);
    case Tins::PDU::IPv6:
      return new Tins::IPv6(buffer, size);
    case Tins::PDU::ARP:
      return new Tins::ARP(buffer, size);
    case Tins::PDU::IEEE802_3:
      return new Tins::IEEE802_3(buffer, size);
    case Tins::PDU::PPPOE:
```

```
        return new Tins::PPPOE(buffer, size);
          #ifdef TINS_HAVE_DOT11
        case Tins::PDU::RADIOTAP:
        return new Tins::RADIOTAP(buffer, size);
        case Tins::PDU::DOT11:
        case Tins::PDU::DOT11_ACK:
        case Tins::PDU::DOT11_ASSOC_REQ:
        case Tins::PDU::DOT11_ASSOC_RESP:
        case Tins::PDU::DOT11_AUTH:
        case Tins::PDU::DOT11_BEACON:
        case Tins::PDU::DOT11_BLOCK_ACK:
        case Tins::PDU::DOT11_BLOCK_ACK_REQ:
        case Tins::PDU::DOT11_CF_END:
        case Tins::PDU::DOT11_DATA:
        case Tins::PDU::DOT11_CONTROL:
        case Tins::PDU::DOT11_DEAUTH:
        case Tins::PDU::DOT11_DIASSOC:
        case Tins::PDU::DOT11_END_CF_ACK:
        case Tins::PDU::DOT11_MANAGEMENT:
        case Tins::PDU::DOT11_PROBE_REQ:
        case Tins::PDU::DOT11_PROBE_RESP:
        case Tins::PDU::DOT11_PS_POLL:
       case Tins::PDU::DOT11_REASSOC_REQ:
        case Tins::PDU::DOT11_REASSOC_RESP:
        case Tins::PDU::DOT11_RTS:
        case Tins::PDU::DOT11_QOS_DATA:
          return Tins::Dot11::from_bytes(buffer, size);
      #endif // TINS_HAVE_DOT11
      default:
        return 0;
  };
}
```

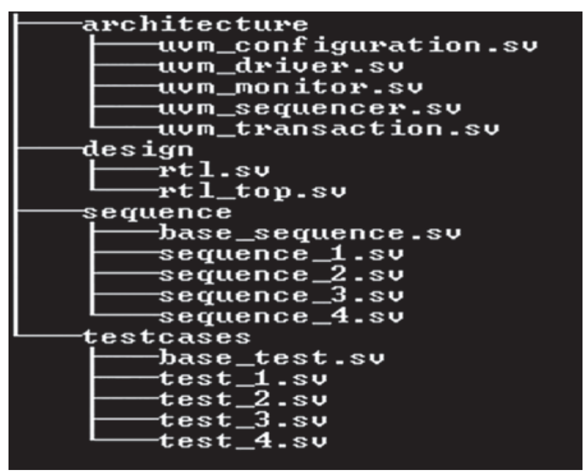
```
Constants::Ethernet::e pdu_flag_to_ether_type
                         (PDU::PDUType flag) {
  switch (flag) {
    case PDU::IP:
      return Constants::Ethernet::IP;
    case PDU::IPv6:
      return Constants::Ethernet::IPV6;
    case PDU::ARP:
      return Constants::Ethernet::ARP;
case PDU::DOT1Q:
      return Constants::Ethernet::VLAN;
    case PDU::PPPOE:
      return Constants::Ethernet::PPPOED;
    case PDU::MPLS:
      return Constants::Ethernet::MPLS;
    case PDU::RSNEAPOL:
    case PDU::RC4EAPOL:
      return Constants::Ethernet::EAPOL;
    default:
```

```
        if (Internals::pdu_type_registered<EthernetII>(flag)) {
          return static_cast<Constants::Ethernet::e>(
            Internals::pdu_type_to_id<EthernetII>(flag)
          );
        }
        return Constants::Ethernet::UNKNOWN;
    }
}
```

These are defined in the internal.cpp files in the libtins project and are referred by the files which have a header for different packets.

4. COMMON HIERARCHY GIVING WELL DEFINED ARCHITECTURE WHICH IS EASY TO UNDERSTAND AND MANAGE

It is better to create a proper project hierarchy to keep and manage and handle the project easily. For example the screenshot shows that how the different files are arranged in different project directories (test cases, sequences, architecture and design).



CONCLUSION

In summary, the article focuses on the common mistakes made by the novice in verification and provides the solution to these problems through various tips and programming examples. Moreover, the article also suggests various tricks which can be applied to enhance the performance of UVM Testbenches. It also covers various conventions to be followed for making the code simpler and how to maintain the project hierarchy.

REFERENCES

1. *"UVM and Emulation: How to Get Your Ultimate Testbench Acceleration Speed-up"* Hans van der Schoot & Ahmed Yehia, DVCon 2015.
2. *"Easier UVM for Functional Verification by Mainstream Users"*, John Aynsley, Duolos.
3. *"The Top Most Common SystemVerilog Constrained Random Gotchas"*, Ahmed Yehia, DVCon 2014.
4. *"Making the most of SystemVerilog and UVM: Hints and Tips for new users"*, Dr. David Long, Doulus.
5. https://verificationacademy.com/cookbook/uvm/performance_guidelines
6. www.libtins.github.io/download/

Mentor Graphics
www.mentor.com

Editor:
Tom Fitzpatrick

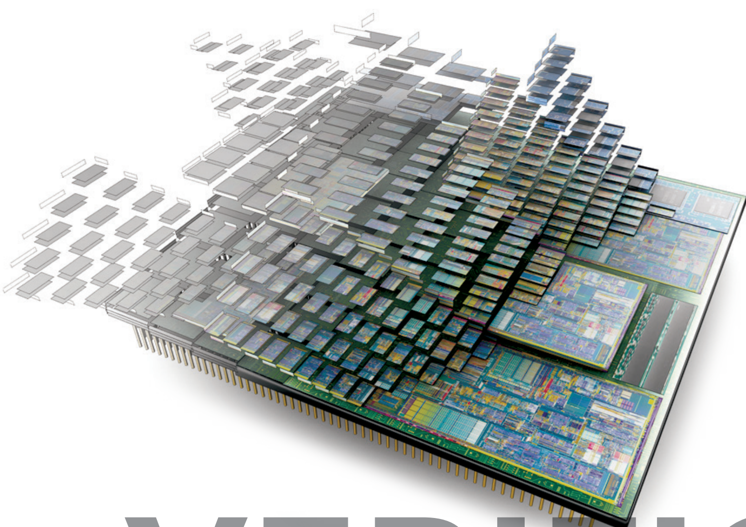
Program Manager:
Rebecca Granquist

Mentor Graphics Corporation
Worldwide Headquarters
8005 SW Boeckman Rd.
Wilsonville, OR 97070-7777

Phone: 503-685-7000

To subscribe visit:
www.mentor.com/horizons

To view our blog visit:
VERIFICATIONHORIZONSBLOG.COM



VERIFICATION HORIZONS

VH