World Class Verilog, SystemVerilog & OVM/UVM Training

OVM/UVM Scoreboards - Fundamental Architectures

Clifford E. Cummings
Sunburst Design, Inc.
cliffc@sunburst-design.com
www.sunburst-design.com

ABSTRACT

One of the most complex components in an OVM/UVM testbench is the scoreboard. Simple tutorials on the theory behind and the creation of the scoreboard are scarce.

This paper will describe two fundamental OVM/UVM scoreboard architectures. The first architecture is a standalone scoreboard component with two UVM analysis implementation ports, which poses unique challenges when declaring and using UVM ports and methods. The second architecture is a highly reusable scoreboard with predictor class and comparator class and the fundamental theories that make this architecture relatively easy to use. The comparator also employs two uvm_tlm_analysis_fifos, to help simplify the implementation. This simple tutorial will assist engineers to become acquainted and proficient with scoreboard development.

The techniques described in this paper can be used with either OVM or UVM verification environments.

Table of Contents

Table of Figures

Table of Examples

1. Introduction

There are many examples and descriptions of OVM/UVM basic testbench components, but good tutorial materials related to the theories behind, and the development of OVM/UVM scoreboards are somewhat scarce.

Although there are multiple scoreboard architectures that can be employed, there are two fundamental architectures that, if well understood, can assist verification engineers to further develop and expand additional testbench scoreboard architectures.

This paper describes fundamental OVM/UVM testbench architectures. The author welcomes feedback and additional examples of high-level scoreboard architectures. I am sharing a couple of the best fundamental techniques known and others are encouraged to share with me the best techniques that they have discovered through their own experience.

2. Scoreboard Fundamental Purpose

The fundamental purpose of a scoreboard is to:

- Examine DUT inputs to predict the expected output.
- Compare the expected output to the actual output.
- Report the success or failure of those comparisons.

Outside of high tech environments, the term "scoreboard" conjures up the image of a sports scoreboard that keeps track of runs, points or goals, but that is an incomplete description of verification scoreboards. A verification scoreboard typically includes some way to predict expected outputs, compare the expected outputs to actual outputs, and to keep track of pass and failure rates identified in the comparison process.

3. New() Constructor -vs- Factory Creation

As a reminder, in OVM and UVM testbench environments, ports, and TLM fifos should be **`new()-`**constructed, because you should never substitute different functionality for port or TLM fifos, while all other UVM components and sequence_items/sequences should be created from the factory so that the top-level test can make test-specific substitutions without modifying multiple files (Cummings [1]).

4. Key Scoreboard Issues

There is certain basic functionality that should be included in every testbench scoreboard. Below is a simplified list of that activity.

In a simple transaction based verification environment, scoreboard development includes:
(1) take a copy of the input transaction.
(2) extract the sampled input signals.
(3) use the extracted input signals to predict the expected output.
(4) take a copy the sampled actual output transaction.
(5) compare the expected transaction outputs to the actual transaction outputs.
(6) track success and failure rates from the comparison.

(7) report comparison failures as they are detected.
(8) report the final success/failure results at the end of the simulation.

In an architected transaction based verification environment, scoreboard development includes:
Predictor functionality
(1) take a copy of the input transaction.
(2) extract the sampled input signals.
(3) use the extracted input signals to predict the expected output.
(4) convert the extracted output back into a transaction.
Comparator functionality
(5) copy the expected transaction to the comparator.
(6) copy the sampled actual output transaction to the comparator.
(7) compare the expected transaction outputs to the actual transaction outputs.
(8) track success and failure rates from the comparison.
(9) report comparison failures as they are detected.
(10) report the final success/failure results at the end of the simulation.

One of the key techniques to simplify required scoreboard activity is to make sure the base transaction was developed with functional, fully-coded, copy and compare methods. The copy method is used to collect the input transaction and sampled-output transaction in both scoreboard architectures, and is used to copy the expected output transaction from the predictor to the comparator in the architected scoreboard environment. The compare method is used to do the actual comparison between the expected and actual outputs.

The ability to include the comparison method in the transaction type definition greatly simplifies the comparison task over older Verilog and SystemVerilog directed testing techniques. Now the transaction data developer can specify what the important fields are that are required to be compared in a self-checking testbench.

5. Analysis port - export - implementation path

A quick review of the analysis-transaction path may be useful and is included in this section.

Analysis ports are basically broadcast ports that write a transaction to the analysis port for any component to take a copy. Unlike other Transaction Level Model (TLM) ports that require a one-to-one, port-to-export or port-to-implementation, connection, analysis ports can broadcast to any number of receivers including no receiver at all. The analysis port does not wait for any confirmation of receipt of the broadcast transaction so each connected export or implementation must provide a void write function that takes the transaction copy in zero time.

Engineers with Verilog testbench experience are accustomed to passing data from module output ports through wire types to module input ports, making a direct connection between the modules. SystemVerilog classes do not have traditional module input and output ports and are not connected by wire data types, so classes have to pass copies of the transaction from one class to another by way of TLM connections, which is similar to copying data from one module to another without any module ports but instead by using hierarchical references.

Analysis TLM communication follows a path of the form: analysis-port to analysis port (if needed) to one or more analysis exports, each connected to another analysis export (if needed) and finally connected to an analysis implementation. The analysis implementation(s) is required to provide a write void function(s) that takes a local copy of the broadcast transaction without consuming any simulation time.

Examples of analysis paths, from simple to more complex) include:
`analysis_port -> analysis_implementation` (simplest)
`analysis_port -> analysis_export ->analysis_implementation` (common)
`analysis_port` (monitor) **`-> analysis_port`** (agent) **`-> analysis_export`** (scoreboard) **`-> analysis_implementation`** (scoreboard predictor)

6. Transaction Class Definition

The transaction class includes definitions of all of the input and output data and control signals. Input signals are typically declared to be **`rand`** variables (variables that can be randomized with specified constraints), while outputs are typically not randomizable, since there is no need to randomize data outputs that will never be driven to the DUT.

Three other important features of the transaction class include the definition of **`copy()`**, **`compare()`** and **`convert2string()`** methods. The **`copy()`** and **`compare()`** methods can either be defined by the user by doing a manual override of the **`do_copy()`** and **`do_compare()`** methods or can be auto-generated by using UVM Field Macros.

```
class trans1 extends uvm_sequence_item;
       logic [15:0] dout;
  rand bit   [15:0] din;
  rand bit          ld, inc, rst_n;

  `uvm_object_utils_begin(trans1)
    `uvm_field_int(dout,  UVM_ALL_ON)
    `uvm_field_int(din,   UVM_ALL_ON | UVM_NOCOMPARE)
    `uvm_field_int(ld,    UVM_ALL_ON | UVM_NOCOMPARE)
    `uvm_field_int(inc,   UVM_ALL_ON | UVM_NOCOMPARE)
    `uvm_field_int(rst_n, UVM_ALL_ON | UVM_NOCOMPARE)
  `uvm_object_utils_end

  typedef enum {reset, load, incr, any} cmd_e;
  rand cmd_e cmd_type;

  constraint c1 {(cmd_type==reset) -> ( rst_n        =='0);}
  constraint c2 {(cmd_type==load ) -> ({rst_n,ld}    =='1);}
  constraint c3 {(cmd_type==incr ) -> ({rst_n,ld,inc}==3'b101);}

  function new (string name="trans1");
    super.new(name);
  endfunction

  function string convert2string();
    string s;
    s = $sformatf("dout=%4h  din=%4h  ld=%b  inc=%b  rst_n=%b",
                  dout, din, ld, inc, rst_n);
```

```
      return s;
    endfunction

    function string output2string();
      string s;
      s = $sformatf("dout=%4h", dout);
      return s;
    endfunction
  endclass
```

Example 1 - Transaction class defined using UVM filed macros

The **`copy()`** method should copy all of the signals, inputs and outputs, while the **`compare()`** method should typically only include the outputs; the signals that will be compared in the scoreboard.

The **`convert2string()`** method is user-defined and should also be defined by the creator of the transaction class. The **`convert2string()`** method is a courtesy provided by the transaction class creator to allow all users of the transaction class to call this method to print out the current contents of a transaction class. The **`convert2string()`** method is a "show my contents" method.

In addition to a **`convert2string()`** method, it is also useful to provide an **`output2string()`** method that can be called by the comparison code whenever the predicted output does not match the actual output. The predicted output should be reported using the **`convert2string()`** method as it will show what the inputs were that were used to calculate the predicted output, while the actual output transaction could be shown using the **`output2string()`** method. This way, the error message can report: transaction inputs, predicted output and actual output as part of the miscompare message. Having all of this information can be useful to help debug the problem.

One frequently asked question is, should the transaction also include the built-in error message to be used by the comparator in the case of a mismatch between predicted input and actual outputs. The answer is no. Exactly how the error is formatted and reported is the job of the engineer who is coding the comparator. Having access to the transaction-convenience methods **`convert2string()`** and **`output2string()`** is useful but imposing a requirement upon the transaction coder to determine the output format of miscompare methods exceeds reasonable expectations.

7. Simple scoreboard architecture

The simple scoreboard architecture uses a class extended from the **`uvm_scoreboard`** base class and includes two **`uvm_analysis_imp`** (implementation) ports. A block diagram of the first scoreboard architecture is shown in Figure 1.

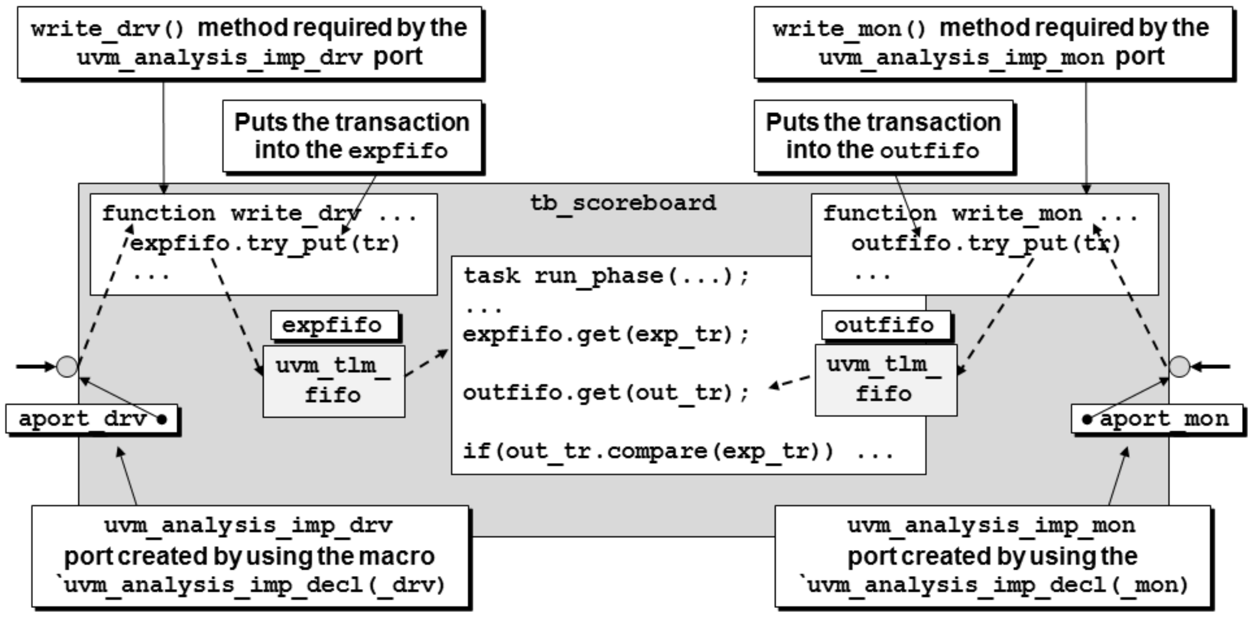

write_drv() method required by the uvm_analysis_imp_drv port
Puts the transaction into the expfifo
Puts the transaction into the outfifo
write_mon() method required by the uvm_analysis_imp_mon port
tb_scoreboard
function write_drv ...
expfifo.try_put(tr)
...
task run_phase(...);
...
expfifo.get(exp_tr);
outfifo.get(out_tr);
if(out_tr.compare(exp_tr)) ...
function write_mon ...
outfifo.try_put(tr)
...
expfifo
uvm_tlm_ fifo
outfifo
uvm_tlm_ fifo
aport_drv
aport_mon
uvm_analysis_imp_drv port created by using the macro `uvm_analysis_imp_decl(_drv)
uvm_analysis_imp_mon port created by using the `uvm_analysis_imp_decl(_mon)

Figure 1 - First scoreboard architecture block diagram using uvm_tlm_fifos

Implementation ports require the implementation of a method called **`write()`**, which must be a void function (it must execute in 0-time and does not return a value). Since the simple scoreboard architecture uses two implementation ports, in theory each implementation port must have its own **`write()`** method but since there are two implementation ports, the simple scoreboard would be required to have two methods both named write, which is illegal. To address this problem, UVM provides a macro called **`` `uvm_analysis_imp_decl(``** *_suffix* **`)`** that is used to declare unique **`uvm_analysis_imp`** ports with unique names that include the *_suffix* argument included in the macro call.

The macro suffix names are required in two places and typically used in other places. The required places are:
(1) as part of the **`uvm_analysis_imp_suffix`** port declarations.
(2) as part of the **`write_suffix`** function name.

```
`uvm_analysis_imp_decl( _drv )
`uvm_analysis_imp_decl( _mon )

class tb_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(tb_scoreboard)

  uvm_analysis_imp_drv #(...) ...;
  uvm_analysis_imp_mon #(...) ...;
```

```
    ...

    function void write_drv(...);
      ...
    endfunction

    function void write_mon(...);
      ...
    endfunction
    ...
  endclass
```

Example 2 - `uvm_analysis_imp_decl(_suffix) usage in port-type and write-method name

The suffix names are typically also included in:
(1) the port name of the port declaration
(2) the port string name when the declared ports are constructed in the **`build_phase()`** method.

```
  `uvm_analysis_imp_decl( _drv )
  `uvm_analysis_imp_decl( _mon )

  class tb_scoreboard extends uvm_scoreboard;
    `uvm_component_utils(tb_scoreboard)

    uvm_analysis_imp_drv #(trans_item, tb_scoreboard) aport_drv;
    uvm_analysis_imp_mon #(trans_item, tb_scoreboard) aport_mon;
    ...

    function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      aport_drv = new("aport_drv", this);
      aport_mon = new("aport_mon", this);
    endfunction
    ...

    function void write_drv(trans_item tr);
      ...
    endfunction

    function void write_mon(trans_item tr);
      ...
    endfunction
    ...
  endclass
```

Example 3 - `uvm_analysis_imp_decl(_suffix) usage in port-name and port construction

Using the **`` `uvm_analysis_imp_decl() ``** macro allows the construction of two analysis implementation ports with corresponding, uniquely named, write methods. The write methods are called automatically whenever a source-component issues an analysis-port write command, and the write methods have the responsibility to copy all of the required transaction fields that will be used by the appropriate functionality. The copy operation is typically accomplished either by direct assignment from the transaction or by calling the transactions own **`copy()`** method.

```
`uvm_analysis_imp_decl( _drv )
`uvm_analysis_imp_decl( _mon )
class tb_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(tb_scoreboard)

  uvm_analysis_imp_drv #(trans1, tb_scoreboard) aport_drv;
  uvm_analysis_imp_mon #(trans1, tb_scoreboard) aport_mon;

  uvm_tlm_fifo #(trans1) expfifo;
  uvm_tlm_fifo #(trans1) outfifo;

  function new (string name, uvm_component parent); ...

  ...

  function void write_drv(trans1 tr);
    ...
  endfunction

  function void write_mon(trans1 tr);
    ...
  endfunction

  task run_phase(uvm_phase phase);
    trans1 exp_tr, out_tr;
    forever begin
      `uvm_info("scoreboard run task", "WAITING for expected output",
                UVM_DEBUG)
      expfifo.get(exp_tr);
      `uvm_info("scoreboard run task", "WAITING for actual output",
                UVM_DEBUG)
      outfifo.get(out_tr);
      if (out_tr.compare(exp_tr)) begin
        PASS();
        `uvm_info ("PASS ", $sformatf("Actual=%s   Expected=%s \n",
              out_tr.output2string(), exp_tr.convert2string()), UVM_HIGH)
      end
      else begin
        ERROR();
        `uvm_error("ERROR", $sformatf("Actual=%s   Expected=%s \n",
                       out_tr.output2string(), exp_tr.convert2string()))
      end
    end
  endtask

  int VECT_CNT, PASS_CNT, ERROR_CNT;

  function void report_phase(uvm_phase phase);
    super.report_phase(phase);
    if (VECT_CNT && !ERROR_CNT)
      `uvm_info("PASSED",
$sformatf("\n\n\n*** TEST PASSED - %0d vectors ran, %0d vectors passed ***\n",
                 VECT_CNT, PASS_CNT), UVM_LOW)
    else
      `uvm_error("FAILED",
$sformatf("\n\n\n*** TEST FAILED - %0d vectors ran, %0d vectors passed, %0d vectors failed ***\n",
                 VECT_CNT, PASS_CNT, ERROR_CNT))
  endfunction

  function void PASS();
    VECT_CNT++;
    PASS_CNT++;
  endfunction

  function void ERROR();
    VECT_CNT++;
    ERROR_CNT++;
  endfunction
endclass
```

Example 4 - Simple scoreboard example code

This simple scoreboard also employs two **uvm_tlm_fifo** components named **expfifo** (expected data transaction fifo) and **outfifo** (actual output transaction fifo). These TLM fifos store the calculated and actual transaction data until both an expected output and actual output have been stored.

The **write_drv()** method can either take a copy of the broadcast input transaction or it can directly calculate the expected output in zero time, which is how the **write_drv()** method of Example 5 is implemented. If a state value must be maintained from one clock cycle to the next, then state variables can be declared as static (the **next_dout** value in this example must be kept between cycles) to store the value between calls to the **write_drv()** method. After calculating the expected output value, it is placed back into the transaction that was passed to this method, overwriting the existing output values, and then the transaction is put into a TLM fifo using the **expfifo.try_put(tr)** fifo-method call.

```
  function void write_drv(trans1 tr);
    static logic [15:0] next_dout;
           logic [15:0] dout;
    //----------------------------------------------------
    `uvm_info("write_drv STIM", tr.convert2string(), UVM_HIGH)
    dout = next_dout;
    if      (!tr.rst_n) {next_dout,dout} = '0;
    else if ( tr.ld)     next_dout       = tr.din;
    else if ( tr.inc)    next_dout++;
    tr.dout = dout;
    void'(expfifo.try_put(tr));
  endfunction
```

Example 5 - write_drv() method to store the expected output transaction

The job of the **write_drv()** method was to collect the input transaction, calculate an expected output, copy the expected value into back into the transaction, then put the expected output transaction into the expfifo TLM fifo.

The **write_mon()** method simply collects the broadcast output transaction and puts into a TLM fifo using the **outfifo.try_put(tr)** fifo-method call as shown in Example 6.

```
function void write_mon(trans1 tr);
  `uvm_info("write_mon OUT ", tr.convert2string(), UVM_HIGH)
  void'(outfifo.try_put(tr));
endfunction
```

Example 6 - write_mon() method to store the actual output transaction

The job of the **`write_mon()`** method was to collect the output transaction, then put the actual output transaction into the **`outfifo`** TLM fifo.

The **`run_phase()`** has a forever loop the gets the expected and actual-output transactions and compares them with the used-defined compare method that should have been added to the transaction code. If the transaction class was properly coded, the compare method only compares the outputs between transactions, which makes it easy to do the comparison in the scoreboard.

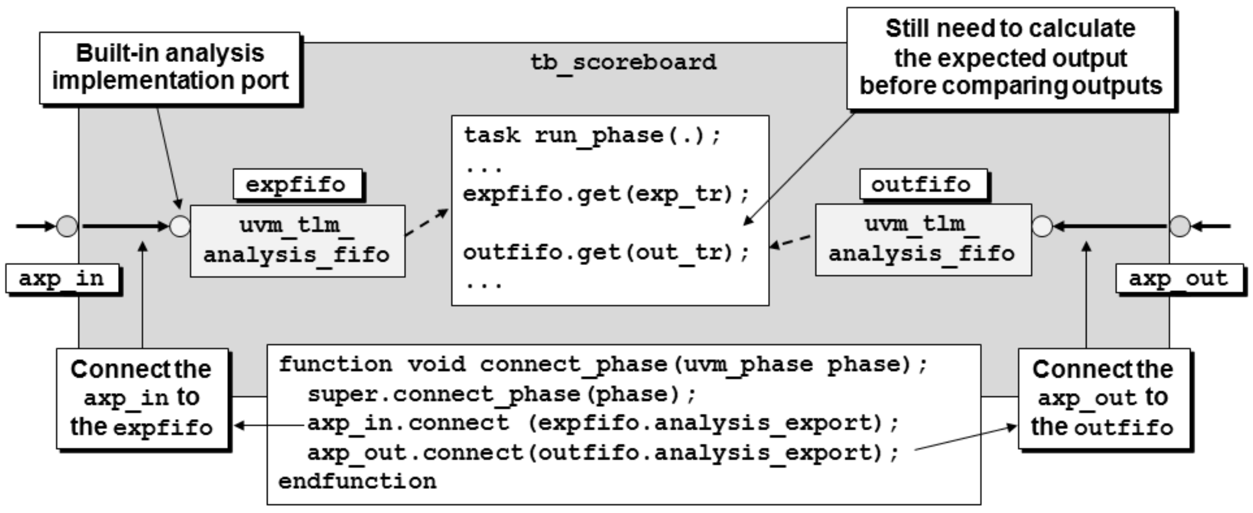
The **`run_phase()`** code also includes calls to **`PASS()`** and **`ERROR()`** methods which increment the vector count (**`VECT_CNT`**), passing vectors count (**`PASS_CNT`**) and failing vectors count (**`ERROR_CNT`**) int-variables respectively.

The simple scoreboard also includes a **`report_phase()`** function to report pass-fail messages at the end of the simulation. The **`report_phase()`** function does a simple determination of pass-fail based on the vector counts generated in the **`run_phase().`**

The full **`tb_scoreboard`** code can be found in the appendix.

7.1. Using uvm_tlm_analysis_fifos

The simple scoreboard architecture can also be implemented using **`uvm_tlm_analysis_fifos`** instead of **`uvm_tlm_fifos`**, as shown in Figure 2.


Built-in analysis implementation port
tb_scoreboard
Still need to calculate the expected output before comparing outputs
expfifo
uvm_tlm_analysis_fifo
outfifo
uvm_tlm_analysis_fifo
axp_in
axp_out
Connect the axp_in to the expfifo
Connect the axp_out to the outfifo

```
task run_phase(.);
...
expfifo.get(exp_tr);

outfifo.get(out_tr);
...
```

```
function void connect_phase(uvm_phase phase);
  super.connect_phase(phase);
  axp_in.connect (expfifo.analysis_export);
  axp_out.connect(outfifo.analysis_export);
endfunction
```

Figure 2 - First scoreboard architecture block diagram using uvm_tlm_analysis_fifos

The **uvm_tlm_analysis_fifo** component has a built-in analysis implementation port, so the **tb_scoreboard** simply declares analysis exports instead of analysis implementation ports and connects the analysis exports to the **uvm_tlm_analysis_fifo** components. When an analysis port broadcasts a transaction to the **tb_scoreboard**, that transaction is passed through the analysis exports directly to the connected analysis implementation port of the **uvm_tlm_analysis_fifo**. The **uvm_tlm_analysis_fifo** has implemented the required **write()** method, which stores the transaction into the fifo storage. The **tb_scoreboard** then extracts the required transaction when it is needed for comparison.

Using the **uvm_tlm_analysis_fifo** in the **tb_scoreboard** eliminates the need to call the **`uvm_analysis_imp_decl()** macros and corresponding port-restrictions and multiple-**write_suffix()** methods.

These **uvm_tlm_analysis_fifo** components will be used in the second scoreboard architecture.

7.2. Second scoreboard architecture

A second approach to scoreboard development is to use the simple scoreboard architecture shown in Figure 3.

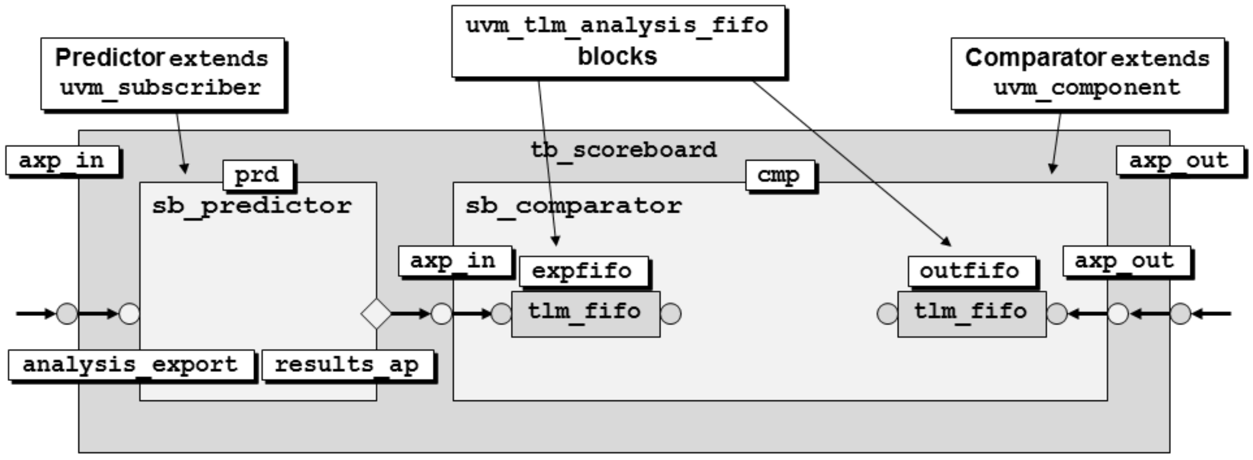


Figure 3 - Second scoreboard architecture block diagram

The second testbench architecture uses a scoreboard that declares two **uvm_analysis_exports**, which do not require write functions. The **uvm_analysis_export** is a pass-through port that passes the transaction handles through to **uvm_analysis_imp** ports implemented in the **uvm_tlm_analysis_fifos**, which provide the required write methods. By deferring the implementation to the fifo **uvm_analysis_imp** ports, each **uvm_tlm_analysis_fifo** is a separate object of this class and each only has one implementation port, therefore no multi-**write()** method problem exists. Plus the **uvm_tlm_analysis_fifo** is pre-coded with the necessary analysis implementation and corresponding **write()** method and includes an unbounded, parameterized (to the transaction type) SystemVerilog **mailbox** that acts as a blocking FIFO to be used by the comparator. A blocking FIFO, in mailbox-form, provides a **`get()`** method that can be used by the comparator to "get" the predicted output (wait until a predicted output is available before continuing execution of the comparator code; hence, the comparator operation is blocked until the **`get()`** method succeeds), then the comparator can do a "get" on the actual output **`uvm_tlm_analysis_fifo`** and wait (block) until an actual output transaction has been sampled and stored into the actual-output **`uvm_tlm_analysis_fifo`**.

The biggest advantage of the second scoreboard architecture is that most of the code can be used as-is and only requires the user to properly setup common transaction methods and to code an external **`sb_calc_exp()`** method. Details are in the following sections.

The second scoreboard architecture consists of four files, three of which are fully coded:

`tb_scoreboard.sv`	- No code modification required.
`sb_predictor.sv`	- No code modification required.
`sb_calc_exp.sv`	- External function called by the **`sb_predictor`** - user implementation required.
`sb_comparator.sv`	- No code modification required.

The second scoreboard architecture also requires that the transaction class include **`copy()`** and **`compare()`** methods either user-defined or auto-generated using UVM field macros.

7.3. tb_scoreboard

The tb_scoreboard code is mostly just a wrapper that includes the necessary export-ports to capture transactions from both stimulus monitor and sampling monitor, plus the predictor and comparator. If the user's base transaction class is named **`trans1`** or a factory substituted derivative of **`trans1`**, this code can be used without modification. If the user would prefer a different default name for the transaction class, simply do a global replacement of **`trans1`** with the user-named transaction class.

As shown in Figure 3, the **`tb_scoreboard`** in Example 7 includes declarations for two **`uvm_analysis_export`** ports called **`axp_in`** (input analysis export) and **`axp_out`** (output analysis export). The same example declares handles for the **`sb_predictor`** and **`sb_comparator`** components, called **`prd`** and **`cmp`** respectively.

The **`tb_scoreboard`** includes the standard **`new()`** constructor that is included with almost all UVM components.

In the **`build()`** method, the analysis exports are **`new()-`**constructed while the **`sb_predictor`** and **`sb_comparator`** are factory-created.

As can be seen from Figure 3, three connections are required: (1) connect the **`axp_in`** analysis export to the **`sb_predictor`**, (2) connect the **`sb_predictor`** to the **`sb_comparator`**, and (3) connect the **`sb_comparator`** to the **`axp_out`** analysis export. This is all done in the **`connect()`** method.

```
class tb_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(tb_scoreboard)

  uvm_analysis_export #(trans1) axp_in;
  uvm_analysis_export #(trans1) axp_out;
  sb_predictor                  prd;
  sb_comparator                 cmp;

  function new(string name, uvm_component parent);  ...

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    axp_in  = new("axp_in",  this);
    axp_out = new("axp_out", this);
    prd     =  sb_predictor::type_id::create("prd", this);
    cmp     = sb_comparator::type_id::create("cmp", this);
  endfunction

  function void connect_phase( uvm_phase phase );
    axp_in.connect        (prd.analysis_export);
    axp_out.connect       (cmp.axp_out);
    prd.results_ap.connect(cmp.axp_in);
  endfunction
endclass
```

Example 7 - sb_scoreboard.sv - Scoreboard code with all required scoreboard components properly included

The **`sb_scoreboard`** code shown in Example 7 is relatively simple and requires no user modification.

7.4. sb_predictor

The **`sb_predictor`** in this architecture is extended from the **`uvm_subscriber`** class, which includes a built-in **`uvm_analysis_imp`** port. The built-in analysis implementation port named **`analysis_export`** that is connected to the **`tb_scoreboard`** **`axp_in`** analysis export and is used to capture the transaction that is passed to the **`tb_scoreboard`** from the analysis port on the stimulus monitor.

The **`sb_predictor`** code is mostly a wrapper that includes a built-in **`uvm_analysis_imp`** port that the necessary export-ports use to capture transactions from both stimulus monitor and sampling monitor, plus the predictor and comparator. If the user's base transaction class is **`trans1`** or a factory substituted derivative of **`trans1`**, this code can be used without modification.

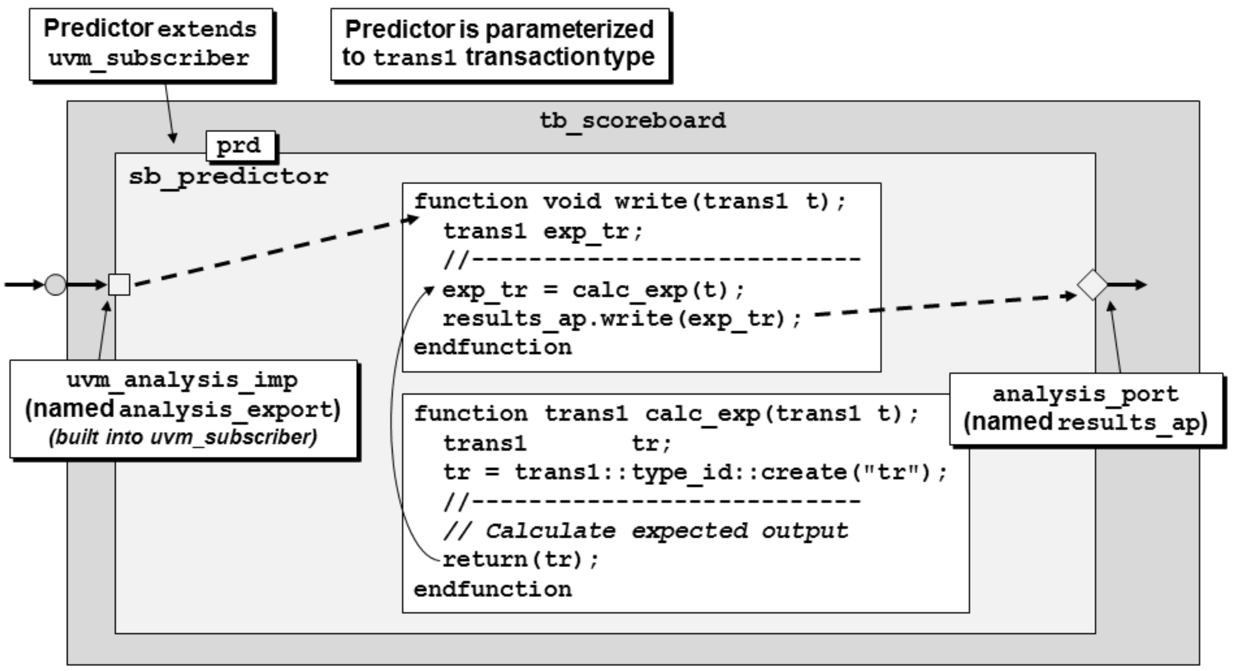

Predictor extends uvm_subscriber
Predictor is parameterized to trans1 transaction type
tb_scoreboard
prd
sb_predictor
function void write(trans1 t);
trans1 exp_tr;
//--------------------------
exp_tr = calc_exp(t);
results_ap.write(exp_tr);
endfunction
function trans1 calc_exp(trans1 t);
trans1 tr;
tr = trans1::type_id::create("tr");
//--------------------------
// Calculate expected output
return(tr);
endfunction
uvm_analysis_imp (named analysis_export) (built into uvm_subscriber)
analysis_port (named results_ap)

Figure 4 - Scoreboard predictor block diagram

```
class sb_predictor extends uvm_subscriber #(trans1);
  `uvm_component_utils(sb_predictor)

  uvm_analysis_port #(trans1) results_ap;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    results_ap = new("results_ap", this);
  endfunction

  function void write(trans1 t);
    trans1 exp_tr;
    //--------------------------
    exp_tr = sb_calc_exp(t);
    results_ap.write(exp_tr);
  endfunction

  extern function trans1 sb_calc_exp(trans1 t);
endclass
```

Example 8 - sb_predictor.sv - Scoreboard predictor code with extern function sb_calc_exp() reference

The **`sb_predictor`** has implemented a **`write()`** method (called when an analysis **`port.write()`** method is externally executed), to copy the broadcast transaction and pass it to an **`sb_calc_exp()`** method. Declaring the **`sb_calc_exp()`** method to be an extern method means that this predictor can be used without modification (if **trans1** is the transaction class name), and the **sb_calc_exp()** method is the only scoreboard file that needs to be user modified.

7.5. sb_calc_exp

The **sb_calc_exp()** external method has been placed in a separate file and is the only file that needs to be completed by the user for the second scoreboard architecture.

Inside of the external **sb_calc_exp()** function, the scoreboard creator must examine the transaction inputs that were sampled on the **posedge clk** by the monitor and predict what the outputs should be for those inputs. The transaction outputs that are passed to this function are ignored - the outputs are being predicted in this function.

```
function trans1 sb_predictor::sb_calc_exp (trans1 t);
  static logic [15:0] next_dout;
         logic [15:0] dout;
  trans1 tr = trans1::type_id::create("tr");
  //---------------------------
  `uvm_info(get_type_name(), t.convert2string(), UVM_HIGH)
  // async reset: reset the next_dout AND current dout values -OR-
  // non-reset  : assign dout values & calculate the next_dout values
  dout = next_dout;
  if      (!t.rst_n) {next_dout,dout} = '0;
  else if ( t.ld)     next_dout       = t.din;
  else if ( t.inc)    next_dout++;
  // copy all sampled inputs & outputs
  tr.copy(t);
  // overwrite the dout values with the calculated values.
  // dout values were either calculated in the previous cycle
  //      or asynchronously reset in this cycle
  tr.dout = dout;
  return(tr);
endfunction
```

Example 9 - sb_calc_exp.sv - Example scoreboard external calc_exp() function definition

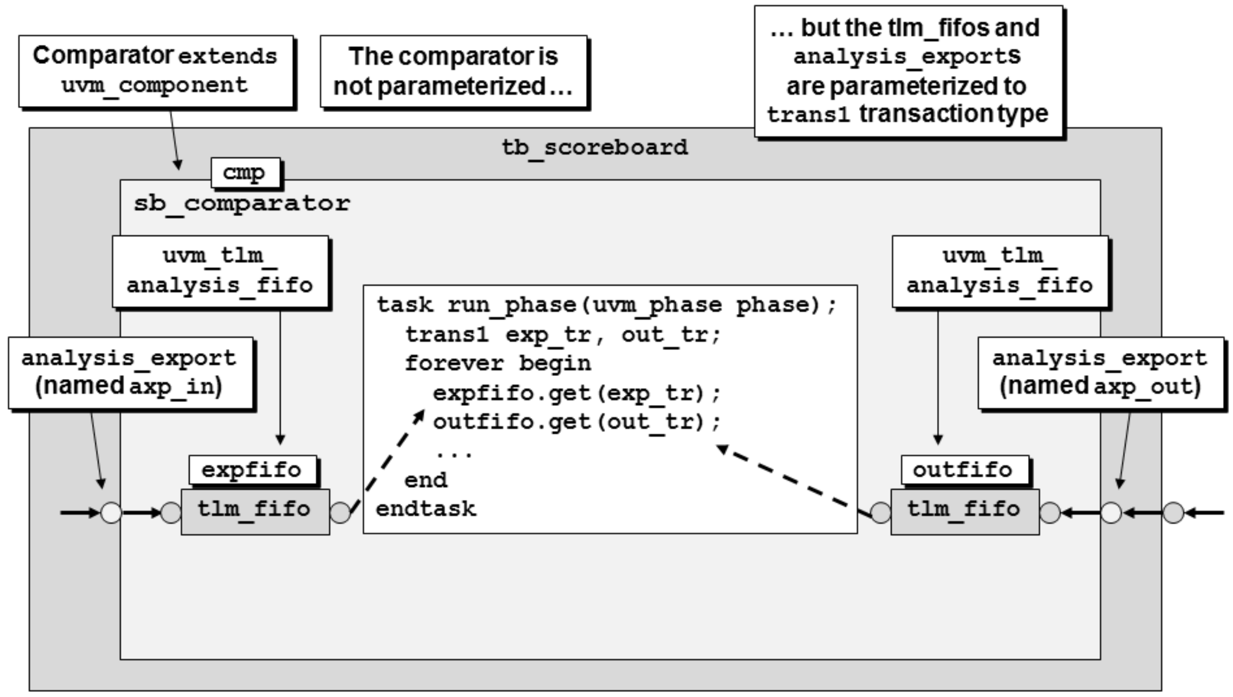
Using the **copy()** method provided by the transaction coder, all of the transaction inputs and outputs are copied into the **tr** transaction object, then the predicted outputs are used to overwrite the copied outputs in this transaction object. This transaction object now includes the inputs that were captured on the **posedge clk** of the DUT and the corresponding expected output value(s).

7.6. sb_comparator

The **sb_comparator** in this architecture is extended from the **uvm_component** class. Two analysis exports are declared and constructed in the **sb_comparator** and connected to two declared and constructed **uvm_tlm_analysis_fifo** components. Each **uvm_tlm_analysis_fifo** component includes a built-in **uvm_analysis_imp** port, and since the tlm_fifos are connected directly to the **sb_comparator analysis_exports**, any transaction broadcast to these external exports will be automatically put onto the respective tlm_fifo storage arrays (SystemVerilog mailboxes).

The **`sb_comparator run_phase()`** has a **`forever`** loop that gets an expected transaction when one is available (controlled by a blocking **`expfifo.get()`** call), then gets an actual transaction with sampled outputs when one is available (controlled by a blocking **`outfifo.get()`** call), and uses the user-defined **`compare()`** method from the transaction (if properly coded) to determine if the actual output matches the expected output.

The **`sb_comparator`** code is mostly a wrapper that includes a built-in **`uvm_analysis_imp`** port that the necessary export-ports use to capture transactions from both stimulus monitor and sampling monitor, plus the predictor and comparator. If the user's base transaction class is **`trans1`** or a factory substituted derivative of **`trans1`**, this code can be used without modification.


Comparator extends uvm_component
The comparator is not parameterized...
... but the tlm_fifos and analysis_exports are parameterized to trans1 transaction type
tb_scoreboard
cmp
sb_comparator
uvm_tlm_ analysis_fifo
uvm_tlm_ analysis_fifo
analysis_export (named axp_in)
analysis_export (named axp_out)
expfifo
tlm_fifo
outfifo
tlm_fifo

```
task run_phase(uvm_phase phase);
  trans1 exp_tr, out_tr;
  forever begin
    expfifo.get(exp_tr);
    outfifo.get(out_tr);
    ...
  end
endtask
```

Figure 5 - Scoreboard comparator block diagram

The comparator code is surprisingly simple in concept. The comparator will include two **`uvm_analysis_export`**(s) and two **`uvm_tlm_analysis_fifo`**(s) (blocking FIFOs), one that is used to store the predicted/expected transactions (**`exp_fifo`**), one that stores the actual output transactions (**`outfifo`**) and a forever-running **`run_phase()`** task that blocks until both an expected transaction and actual output transaction can be retrieved and compared. If the transaction coder properly developed the transaction class such that only the output signals are tested in a **`compare()`** method, the comparator code simply calls
`if (out_tr.compare(exp_tr))` ... and the comparison is automatically executed. Doing comparisons in Verilog directed tests was never this easy! The Verilog testbench coder had to manually compare the predicted output signals to the actual output signals to determine if the transaction had been successful. Verilog testbenches required real work!

```
class sb_comparator extends uvm_component;
  `uvm_component_utils(sb_comparator)

  uvm_analysis_export   #(trans1) axp_in;
  uvm_analysis_export   #(trans1) axp_out;
  uvm_tlm_analysis_fifo #(trans1) expfifo;
  uvm_tlm_analysis_fifo #(trans1) outfifo;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    axp_in  = new("axp_in",  this);
    axp_out = new("axp_out", this);
    expfifo = new("expfifo", this);
    outfifo = new("outfifo", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    axp_in.connect (expfifo.analysis_export);
    axp_out.connect(outfifo.analysis_export);
  endfunction

  task run_phase(uvm_phase phase);
    trans1 exp_tr, out_tr;
    forever begin
      `uvm_info("sb_comparator run task",
                "WAITING for expected output", UVM_DEBUG)
      expfifo.get(exp_tr);
      `uvm_info("sb_comparator run task",
                "WAITING for actual output", UVM_DEBUG)
      outfifo.get(out_tr);
      if (out_tr.compare(exp_tr)) begin
        PASS();
        `uvm_info ("PASS ", $sformatf("Actual=%s   Expected=%s \n",
                   out_tr.output2string(),
                   exp_tr.convert2string()), UVM_HIGH)
      end
      else begin
        ERROR();
        `uvm_error("ERROR", $sformatf("Actual=%s   Expected=%s \n",
                   out_tr.output2string(),
                   exp_tr.convert2string()))
      end
    end
  endtask

  int VECT_CNT, PASS_CNT, ERROR_CNT;

  function void report_phase(uvm_phase phase);
    super.report_phase(phase);
    if (VECT_CNT && !ERROR_CNT)
      `uvm_info(get_type_name(),
      $sformatf(
"\n\n\n*** TEST PASSED - %0d vectors ran, %0d vectors passed ***\n",
                 VECT_CNT, PASS_CNT), UVM_LOW)
    else
      `uvm_info(get_type_name(),
      $sformatf(
"\n\n\n*** TEST FAILED - %0d vectors ran, %0d vectors passed, %0d vectors failed ***\n",
                 VECT_CNT, PASS_CNT, ERROR_CNT), UVM_LOW)
  endfunction

  function void PASS();
    VECT_CNT++;
    PASS_CNT++;
  endfunction

  function void ERROR();
    VECT_CNT++;
    ERROR_CNT++;
  endfunction
endclass
```

Example 10 - sb_comparator.sv - Scoreboard comparator code - no modification required

It should be noted that as long as four requirements have been met, the **`sb_comparator`** code can be used as is without any modification. The four requirements are:

(1) The base class transaction class was called **`trans1`** (if a different transaction class name was used, simply replace the **`trans1`** references with the new transaction class name).
(2) The transaction class has properly implemented the **`compare()`** method to only compare the required output signals.
(3) The transaction class has properly implemented the **`convert2string()`** method to display all transaction signals.
(4) The transaction class has properly implemented the **`output2string()`** method to only display the compared transaction output signals.

8. Conclusions

The first scoreboard architecture shows how to handle a scoreboard that requires two analysis implementation ports. The clear explanation of how this is accomplished will help any user who wants to develop a scoreboard with two analysis implementation ports or any other testbench component that might also require two analysis implementation ports. Good UVM verification engineers should understand this architecture and the requirements that it imposes.

The second scoreboard architecture includes some very attractive features, which currently makes it my own preferred architecture for developing a testbench scoreboard.

In the second scoreboard architecture, all of the scoreboard blocks are pre-coded and can be used as-is, with the exception of the external **`sb_calc_exp()`** method. What is often perceived to be a daunting task to create a **`tb_scoreboard`** has now been reduced to coding the correct prediction logic in the **`sb_calc_exp()`** function of a single file. All other files can be copied and used as is.

9. Acknowledgements

I am grateful to my colleague and friend Al Czamara for his review and suggested improvements to this paper.

10. References

[1] Clifford E. Cummings, " The OVM/UVM Factory & Factory Overrides - How They Works - Why They Are Important," SNUG (Synopsys Users Group) 2012 (Santa Clara, CA), March 2012. Also available at www.sunburst-design.com/papers

[2] Universal Verification Methodology (UVM) 1.1 Class Reference, May 2011, Accellera, Napa, CA. www.accellera.org/home

[3] UVM source code (it is sometimes easier to grep the UVM source files than to search the UVM Reference Guide)

11. AUTHOR & CONTACT INFORMATION

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 32 years of ASIC, FPGA and system design experience and 23 years of SystemVerilog, synthesis and methodology training experience.

Mr Cummings has presented more than 100 SystemVerilog seminars and training classes in the past nine years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Mr Cummings has participated on every IEEE & Accellera SystemVerilog, SystemVerilog Synthesis, SystemVerilog committee, and has presented more than 40 papers on SystemVerilog & SystemVerilog related design, synthesis and verification techniques.

Mr Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

Sunburst Design, Inc. offers World Class Verilog & SystemVerilog training courses. For more information, visit the www.sunburst-design.com web site.
Email address: cliffc@sunburst-design.com

Last Updated: October, 2014

Appendix

This appendix contains fully coded scoreboard examples that correspond to the abbreviated examples shown throughout this paper.

```
class trans1 extends uvm_sequence_item;
       logic [15:0] dout;
  rand bit   [15:0] din;
  rand bit          ld, inc, rst_n;

  `uvm_object_utils_begin(trans1)
    `uvm_field_int(dout,  UVM_ALL_ON)
    `uvm_field_int(din,   UVM_ALL_ON | UVM_NOCOMPARE)
    `uvm_field_int(ld,    UVM_ALL_ON | UVM_NOCOMPARE)
    `uvm_field_int(inc,   UVM_ALL_ON | UVM_NOCOMPARE)
    `uvm_field_int(rst_n, UVM_ALL_ON | UVM_NOCOMPARE)
  `uvm_object_utils_end

  typedef enum {reset, load, incr, any} cmd_e;
  rand cmd_e cmd_type;

  constraint c1 {(cmd_type==reset) -> ( rst_n        =='0);}
  constraint c2 {(cmd_type==load ) -> ({rst_n,ld}    =='1);}
  constraint c3 {(cmd_type==incr ) -> ({rst_n,ld,inc}==3'b101);}

  function new (string name="trans1");
    super.new(name);
  endfunction

  function string convert2string();
    return($sformatf("dout=%4h  din=%4h  ld=%b  inc=%b  rst_n=%b",
                     dout, din, ld, inc, rst_n));
  endfunction

  function string output2string();
    return($sformatf("dout=%4h", dout));
  endfunction
endclass
```

Example 11 - trans1 transaction class type w/ code to implement both copy() and output-compare() methods

```
`uvm_analysis_imp_decl( _drv )
`uvm_analysis_imp_decl( _mon )

class tb_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(tb_scoreboard)

  uvm_analysis_imp_drv #(trans1, tb_scoreboard) aport_drv;
  uvm_analysis_imp_mon #(trans1, tb_scoreboard) aport_mon;

  uvm_tlm_fifo #(trans1) expfifo;
  uvm_tlm_fifo #(trans1) outfifo;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    aport_drv = new("aport_drv", this);
    aport_mon = new("aport_mon", this);
    expfifo   = new("expfifo",   this);
    outfifo   = new("outfifo",   this);
  endfunction

  function void write_drv(trans1 tr);
    static logic [15:0] next_dout;
           logic [15:0] dout;
    //---------------------------------------------------
    `uvm_info("write_drv STIM", tr.convert2string(), UVM_HIGH)
    dout = next_dout;
    if      (!tr.rst_n) {next_dout,dout} = '0;
    else if ( tr.ld)     next_dout       = tr.din;
    else if ( tr.inc)    next_dout++;
    tr.dout = dout;
    void'(expfifo.try_put(tr));
  endfunction

  function void write_mon(trans1 tr);
    `uvm_info("write_mon OUT ", tr.convert2string(), UVM_HIGH)
    void'(outfifo.try_put(tr));
  endfunction

  task run_phase(uvm_phase phase);
    trans1 exp_tr, out_tr;
    forever begin
      `uvm_info("scoreboard run task",
                "WAITING for expected output", UVM_DEBUG)
      expfifo.get(exp_tr);
      `uvm_info("scoreboard run task",
                "WAITING for actual output", UVM_DEBUG)
      outfifo.get(out_tr);
      if (out_tr.compare(exp_tr)) begin
        PASS();
        `uvm_info ("PASS ", $sformatf("Actual=%s   Expected=%s \n",
                   out_tr.output2string(),
                   exp_tr.convert2string()), UVM_HIGH)
      end
```

```
        else begin
          ERROR();
          `uvm_error("ERROR", $sformatf("Actual=%s   Expected=%s \n"
                     out_tr.output2string(),
                     exp_tr.convert2string()))
        end
      end
    endtask

    int VECT_CNT, PASS_CNT, ERROR_CNT;

    function void report_phase(uvm_phase phase);
      super.report_phase(phase);
      if (VECT_CNT && !ERROR_CNT)
        `uvm_info("PASSED",
  $sformatf("\n\n\n*** TEST PASSED - %0d vectors ran, %0d vectors passed ***\n",
                   VECT_CNT, PASS_CNT), UVM_LOW)
      else
        `uvm_info("FAILED",
  $sformatf("\n\n\n*** TEST FAILED - %0d vectors ran, %0d vectors passed, %0d vectors failed ***\n",
                   VECT_CNT, PASS_CNT, ERROR_CNT), UVM_LOW)
    endfunction

    function void PASS();
      VECT_CNT++;
      PASS_CNT++;
    endfunction

    function void ERROR();
      VECT_CNT++;
      ERROR_CNT++;
    endfunction
  endclass
```

Example 12 - tb_scoreboard architecture implementation #1 - tb_scoreboard.sv

```
class tb_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(tb_scoreboard)

  uvm_analysis_export #(trans1) axp_in;
  uvm_analysis_export #(trans1) axp_out;
  sb_predictor                  prd;
  sb_comparator                 cmp;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    axp_in  = new("axp_in",  this);
    axp_out = new("axp_out", this);
    prd     =  sb_predictor::type_id::create("prd", this);
    cmp     = sb_comparator::type_id::create("cmp", this);
  endfunction

  function void connect_phase( uvm_phase phase );
    // Connect predictor & comparator to respective analysis exports
    axp_in.connect        (prd.analysis_export);
    axp_out.connect       (cmp.axp_out);
    // Connect predictor to comparator
    prd.results_ap.connect(cmp.axp_in);
  endfunction
endclass
```

Example 13 - tb_scoreboard architecture implementation #2 - tb_scoreboard.sv file

```
class sb_comparator extends uvm_component;
  `uvm_component_utils(sb_comparator)

  uvm_analysis_export   #(trans1) axp_in;
  uvm_analysis_export   #(trans1) axp_out;
  uvm_tlm_analysis_fifo #(trans1) expfifo;
  uvm_tlm_analysis_fifo #(trans1) outfifo;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    axp_in  = new("axp_in",  this);
    axp_out = new("axp_out", this);
    expfifo = new("expfifo", this);
    outfifo = new("outfifo", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    axp_in.connect (expfifo.analysis_export);
    axp_out.connect(outfifo.analysis_export);
  endfunction

  task run_phase(uvm_phase phase);
    trans1 exp_tr, out_tr;
    forever begin
      `uvm_info("sb_comparator run task",
                "WAITING for expected output", UVM_DEBUG)
      expfifo.get(exp_tr);
      `uvm_info("sb_comparator run task",
                "WAITING for actual output", UVM_DEBUG)
      outfifo.get(out_tr);
      if (out_tr.compare(exp_tr)) begin
        PASS();
        `uvm_info ("PASS ", $sformatf("Actual=%s   Expected=%s \n",
                   out_tr.output2string(),
                   exp_tr.convert2string()), UVM_HIGH)
      end
      else begin
        ERROR();
        `uvm_error("ERROR", $sformatf("Actual=%s   Expected=%s \n",
                   out_tr.output2string(),
                   exp_tr.convert2string()))
      end
    end
  endtask

  int VECT_CNT, PASS_CNT, ERROR_CNT;

  function void report_phase(uvm_phase phase);
    super.report_phase(phase);
    if (VECT_CNT && !ERROR_CNT)
      `uvm_info(get_type_name(),
$sformatf("\n\n\n*** TEST PASSED - %0d vectors ran, %0d vectors passed ***\n",
                  VECT_CNT, PASS_CNT), UVM_LOW)
    else
      `uvm_info(get_type_name(),
$sformatf("\n\n\n*** TEST FAILED - %0d vectors ran, %0d vectors passed, %0d vectors failed ***\n",
                  VECT_CNT, PASS_CNT, ERROR_CNT), UVM_LOW)
  endfunction

  function void PASS();
    VECT_CNT++;
    PASS_CNT++;
  endfunction

  function void ERROR();
    VECT_CNT++;
    ERROR_CNT++;
  endfunction
endclass
```

Example 14 - tb_scoreboard architecture implementation #2 - sb_comparator.sv file

```
class sb_predictor extends uvm_subscriber #(trans1);
  `uvm_component_utils(sb_predictor)

  uvm_analysis_port #(trans1) results_ap;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    results_ap = new("results_ap", this);
  endfunction

  function void write(trans1 t);
    trans1 exp_tr;
    //---------------------------
    exp_tr = sb_calc_exp(t);
    results_ap.write(exp_tr);
  endfunction

  extern function trans1 sb_calc_exp(trans1 t);
endclass
```

Example 15 - tb_scoreboard architecture implementation #2 - sb_predictor.sv file

```
function trans1 sb_predictor::sb_calc_exp(trans1 t);
  static logic [15:0] next_dout;
         logic [15:0] dout;

  trans1       tr;
  tr = trans1::type_id::create("tr");
  //---------------------------
  `uvm_info(get_type_name(), t.convert2string(), UVM_HIGH)

  // async reset: reset the next_dout AND current dout values -OR-
  // non-reset  : assign dout values & calculate the next_dout values
  dout = next_dout;
  if      (!t.rst_n) {next_dout,dout} = '0;
  else if ( t.ld)     next_dout       = t.din;
  else if ( t.inc)    next_dout       = ++next_dout;

  // copy all sampled inputs & outputs
  tr.copy(t);

  // overwrite the dout values with the calculated values
  // dout values were either calculated in the previous cycle
  //       or asynchronously reset in this cycle
  tr.dout = dout;
  return(tr);
endfunction
```

Example 16 - tb_scoreboard architecture implementation #2 - sb_calc_exp.sv file

This **`sb_calc_exp()`** method is used to test a simple program counter with asynchronous reset, and synchronous load and increment control signals.