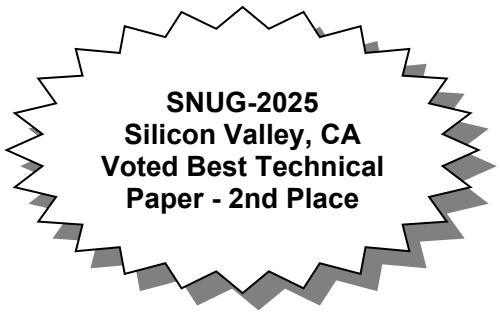
uvmtb_template Files – An Efficient & Rapid Way To Create UVM Testbenches



Clifford E. Cummings

Paradigm Works, Inc.
Provo, Utah, USA

www.sunburst-design.com
cliff.cummings@paradigm-works.com

ABSTRACT

There is a common misconception that due to the complexity and required structure of UVM testbenches that UVM is too complex to be used for block level and simple design testing. People believe that Verilog and SystemVerilog testbenches are better suited for simple design testing, but this is not true.

There is also a widely held belief that commercial or in-house UVM testbench generation tools are the best way to build simple UVM testbenches, and that without those tools, it is too difficult to quickly assemble a UVM testbench.

This paper describes and shares a set of open and simple testbench files called the uvmtb_template directory, which enables an engineer to quickly create simple UVM testbenches by modifying 9 of the 23 template files. The full set of uvmtb_template files, and a description of the required modifications are included in this paper.

Table of Contents

Table of Figures

Table of Examples

1. Introduction

There are three commonly repeated myths surrounding UVM testbench development:

(1) It is easier to create a Verilog/SystemVerilog self-checking testbench than it is to create a UVM self-checking testbench.
(2) UVM testbenches are hard to create.
(3) I should create simple testbenches using Verilog/SystemVerilog and only use UVM to create more complex testbenches.

All three of these statements are false. If you believe that self-checking UVM testbenches are harder to create than Verilog/SystemVerilog testbenches, you have improperly learned UVM or you have taken the wrong UVM training.

This paper shows how to implement simple UVM testbench development techniques using simple template files that you can control.

2. Why is UVM Hard to Learn?

Many engineers believe they can learn UVM by picking up and reading some UVM book, along with the UVM User Guide. They quickly discover this is exceptionally difficult to do. Why is it so hard to learn UVM from existing materials?

Through years of experience, Sunburst Design and Paradigm Works have identified the following reasons why engineers struggle with existing UVM tutorial materials:

(1) The UVM User Guide was written by Cadence and teaches Cadence recommended methods, which includes the use of a large number of UVM macros.
(2) The UVM tutorials on VerificationAcademy.org are shown using Siemens / Mentor recommended methods, which includes the use of fewer UVM macros and more UVM method calls.
(3) The OVM Cookbook (predecessor to UVM) was written by Mentor employees and is based on an earlier version of OVM (the latest techniques are not shown in the book).
(4) The User Guide, tutorials and Cookbook do not acknowledge or explain that alternate methods exist, so users are left to draw erroneous conclusions that some of the examples and explanations shown in these materials are flawed, which is not true. Learners need to be taught the pros and cons of alternate methods so that they understand why there are differences in the various methods presented.
(5) All the people who have written UVM materials are really, really smart software engineers who assume that engineers already understand SystemVerilog syntax and semantics, object-oriented programming and polymorphism semantics, and they do not know how to teach these concepts to beginners.
(6) Many of those who have written UVM materials are software engineers who do not have a strong grasp of good hardware design practices, and it shows in many of the examples.
(7) The UVM User Guide (chapter 2) and the OVM Cookbook (chapter 3) introduce Transaction Level Modeling (TLM) concepts, including put, get and transport communication, but do a poor job of tying the concepts into the rest of the OVM/UVM materials. Engineers often wonder why TLM was introduced in these texts.
(8) Most UVM materials show the driver on the right and the monitor on the left (right to left data-flow inside of the agent). This contradicts known good hardware block diagramming methods (data should flow from left to right in block diagrams) and adds an unnecessary level of confusion to the learning process for those who are familiar with good block diagramming techniques.
(9) There is a huge shortage of complete simple examples. Most of the publicly available example code is in abbreviated code-snippet form, leaving the new user to guess what is missing. Finding full online examples is rare. One notable example shows OVM used on a large VHDL design, which introduces yet another unknown to the learning process.

(10) Of course, you must understand classes, class-extension, virtual classes, virtual methods, dynamic casting, polymorphism, randomization, constraints, covergroups, coverpoints, interfaces and virtual interfaces before you can learn UVM. Too many engineers try to learn UVM without a full understanding of these SystemVerilog fundamentals (this is not the fault of UVM authors).
(11) Classes are applied as stimulus and sampled for verification. Existing materials do not explain why classes are used instead of structs?
(12) Interfaces, virtual interfaces and their recommended usage-models are somewhat buried in the materials and are poorly explained (most authors assume you understand these concepts without much explanation - they are wrong).
(13) There are a substantial number of typos and mistakes sprinkled throughout the materials and examples. The mistakes leave the learner to try to figure out which coding styles are correct, and which have typos.

Proper UVM training should address each of these issues.

3. Why Template Files? Why Not Gen-Tools?

There are many available UVM testbench generation tools, so why create and use UVM testbench template files?

First note that if you use and like a specific UVM testbench generation tool, then keep using it. The tool is probably giving you testbench development value.

If you are considering using third-party UVM testbench generation tools, you might discover that the generation tools all have their own unique required input syntax and a small learning curve. Many of the generation tools also create good albeit somewhat complex set of files based on a testbench patterned after the style of the tool developers. To the developer, the constructed files make sense but to the end-user some of those files might be unnecessary, overly complex, and confusing. I have seen engineers use the UVM testbench generation tools but not understand much of the generated testbench code.

What do you do if the generation tool creates a UVM testbench style that is not desired by the end user? What if the user would like the tool to add some functionality that is non-standard for that tool? The user might try to contact the tool developer and ask for additional options or features and the user is then at the mercy of the tool developer to provide the requested functionality.

What if the user wants the tool to generate less code or a less complicated version of a UVM testbench? Often the user is stuck with the set of files generated by the tool and the user has no way to coerce the tool to generate simpler code. Exactly how the tool generates the UVM files is often a mystery to the end user.

What if you had a simple way to control the testbench structure and files created for your UVM testbenches? That is the topic detailed in this paper.

4. Focused Experience versus Splitting Time

In the introduction of this paper, I described three common myths surrounding testbench development and UVM.

Many engineers have told me that "It is easier to create a Verilog or SystemVerilog self-checking testbench than it is to create a UVM self-checking testbench." This is not true.

Those same engineers claim, "UVM testbenches are hard to create." That might be true if you are creating UVM testbenches from scratch, but using a set of template files makes UVM testbenches easier to create than Verilog or SystemVerilog testbenches.

Many of those same engineers have arrived at the conclusion that they should, "create simple testbenches using Verilog/SystemVerilog and only use UVM to create more complex testbenches." This is a bad idea.

4.1 Simple testbenches using SystemVerilog

If you test simpler blocks using SystemVerilog, you will use your own, nonstandard methodology, and you will be reinventing the wheel. Many of these techniques do not translate to UVM methodologies, so you are largely reinforcing bad habits.

If you subsequently do larger blocks and multiblock testing using UVM, you will be using a different methodology and you will need to change your mindset to use the different methodology.

If you reserve UVM for only the most complex testing situations, you will be less proficient with two different methodologies and not as effective at developing rapid UVM testbenches.

4.2 UVM is SystemVerilog Verification

Very smart developers have taken the SystemVerilog verification enhancements and turned them into a powerful and extensible verification methodology called UVM. With the release of UVM back in 2011, all three major EDA vendors agreed that this was the methodology that verification engineers should use. Whenever any company askes me about SystemVerilog Verification training, I tell them, "UVM is SystemVerilog Verification!"

The more practice you get developing UVM testbenches, the quicker you will put together both simple and more complex UVM testbenches. You should focus on developing your UVM skills as opposed to splitting time between two methodologies.

With some simple template files as a starting point, you can quickly assemble simple block-level UVM testbenches. You can also restructure the simple template files to meet your needs and coding style.

Learning SystemVerilog Verification decoupled from UVM is largely a waste of time. You are not just reinventing the wheel; you are basically trying to reinvent the Tesla by reinventing the wheel.

5. What Makes UVM Easy to Use?

Once you understand a few basics, you quickly discover that very smart people implemented very clever and useful techniques to create powerful self-checking testbenches.

A few UVM features that make testbench development relatively easy to do are detailed below.

5.1 0-Timed Testbench

One reason the UVM testbench lends itself to template implementation is because the entire UVM testbench is almost entirely 0-timed with two handshaking interfaces: (1) sequencer-driver and (2) monitor analysis port, which broadcasts a sampled transaction to all subscribers (all listeners).

As shown in Figure 1, the two components that have timing are the `tb_driver`, which splits the transaction item signals to send as stimulus to the DUT inputs, and the `tb_monitor`, which samples the DUT inputs at the same time that the DUT samples the inputs, and samples the outputs at the end of the cycle after all DUT signals have settled for that cycle.

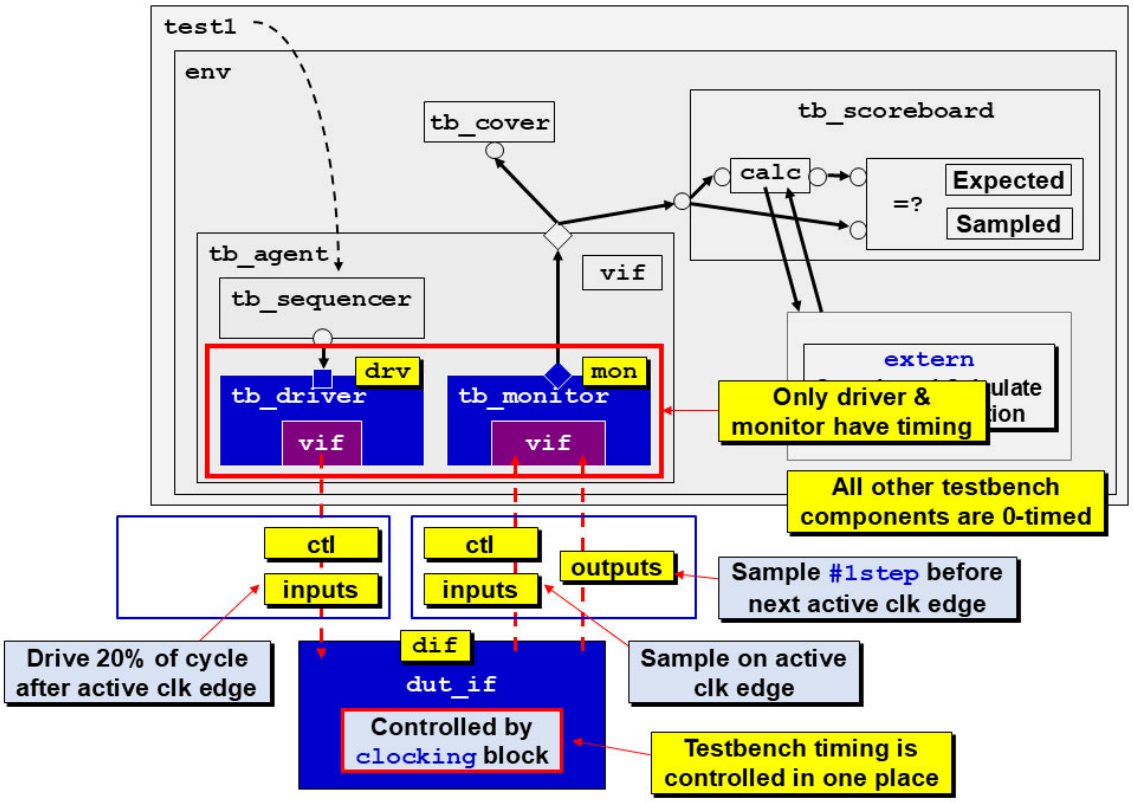


Figure 1 - Component timing in a UVM Testbench

The `tb_sequencer` is always ready to send the next randomized transaction item, but the `tb_driver` uses handshaking to grab the next item (transaction), then the `tb_sequencer` must wait until the `tb_driver` is done with the current item before the next transaction is retrieved from the `tb_sequencer`. The `tb_sequencer` waits for `item_done`.

The `tb_monitor` reassembles the sampled inputs and outputs into a common transaction item and then broadcasts the transaction (whether there are 2 or 1,000 signals) to all other subscribers for analysis or processing. The `tb_monitor` does not wait for any subscriber to acknowledge receipt of the broadcast transaction. All subscribers either need to examine the broadcast transaction in 0-time or they need to take a copy of the broadcast transaction to examine later.

All analysis subscribers wait for a broadcast transaction before processing the transaction, and the broadcast transactions should NEVER be modified [2].

The other handshaking interface occurs in the `sb_comparator`. The 0-timed `sb_comparator` uses blocking `get()` methods to wait until there are expected and actual sampled outputs available from `sb_comparator's` TLM FIFOs before waking up and comparing output fields. The comparator will then increment `PASS` and `ERROR` status counters during the UVM `run_phase()` and ultimately report the final results in the post-run `report_phase()`.

5.2 Common Transaction Item

When building a UVM testbench, the data that is passed around the testbench has all the necessary signals (inputs, outputs control signals), functions, and randomization constraints enclosed within a common transaction unit.

Traditional Verilog testbenches partition the input data, control signals, and output data into separate signals. The UVM-way places all the signals and functions into a common transaction definition and those signals and functions are either used … or they are not, depending on which part of the UVM testbench is using the common transaction.

5.2.1 2 or 1,000 Signals

Most of the UVM testbench structure does not care if the transaction contains 2 signals, or 1,000s of signals, the majority of the UVM testbench connections are identical. This is one reason it is easy to put large portions of the UVM testbench into standard template files.

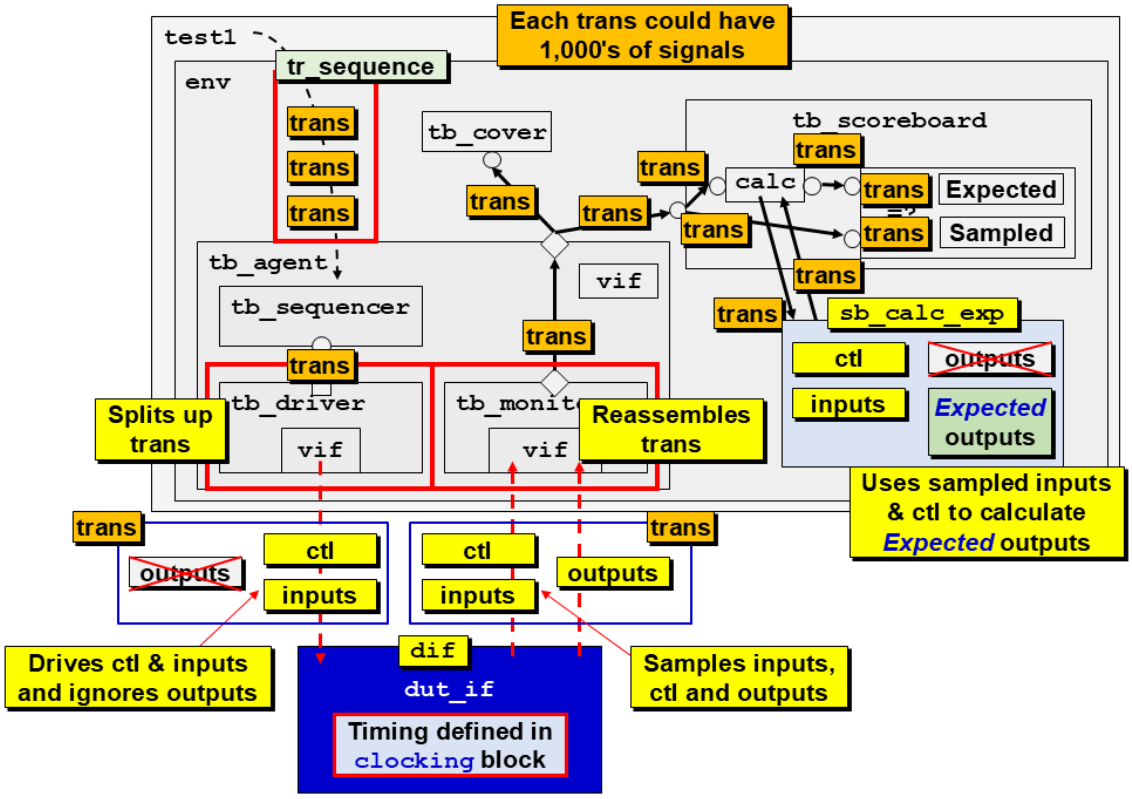


Figure 2 - Common Transaction used in a UVM Testbench

As shown in Figure 2, most of the components are passing around the common transaction unit and the only components or methods that need to split-out the signals are the `tb_driver` component, the `tb_monitor` component and the `sb_calc_exp()` function.

The `tb_driver` grabs an incoming transaction from the `tb_sequencer` (whether there are 2 or 1,000 signals) and splits up the signals and sends the inputs and control signals with the correct timing to the DUT interface (`dut_if`). The driver completely ignores the dedicated output signals.

The `tb_monitor` samples the DUT inputs and control signals on the active clock edge (because that is when the DUT samples those signals) and the `tb_monitor` samples the outputs at the end of the cycle after all the outputs have settled. The `tb_monitor` takes the sample signals and then

reassembles them back into a common transaction (whether there are 2 or 1,000 signals) to be broadcast to the rest of the UVM testbench.

The only other file that cares about the individual signals is the scoreboard-calculate-expected (`sb_calc_exp()`) function, which must examine the sampled inputs, the sampled control signals, and the saved reference-model state information, to calculate what the expected outputs should be. The `sb_calc_exp()` function takes a common transaction (with 2-1,000+ signals) as an input and returns an expected transaction (again with 2-1,000+) signals, which is why the predictor, which calls the `sb_calc_exp()` function, does not require modification whether there are 2 or 1,000 signals.

As seen in Figure 2, most of the UVM testbench just passes around a common transaction of any size, which is not dependent on the number of signals defined in the transaction.

5.2.2 Enables Common Copy, Printing and Other Common Methods

Because all the transaction fields are co-located in a common transaction definition, a `copy()` operation can copy all the fields using a single command as opposed to calling one function to copy inputs and another function to copy outputs.

The same transaction can also print all the transaction fields or a portion of the signals if desired.

That same transaction should have been coded with a common `compare()` method, defined to compare the outputs of one transaction to another. The inclusion of a properly coded `compare()` method will also greatly simplify the creation of self-checking testbenches.

5.2.3 Use the transaction signals … or don't!

Just because all the input, control and output signals are defined in a single transaction does not mean it is required to use all the signals every time a reference is made to the transaction item.

In the stimulus driving half of the UVM testbench as shown in Figure 3, the `tb_driver` gets a transaction from the `tb_sequencer`, which includes input-signals, control-signals and output signals, but the driver does nothing with the output signals. The outputs are included in the common transaction definition, but they are not required when generating stimulus.

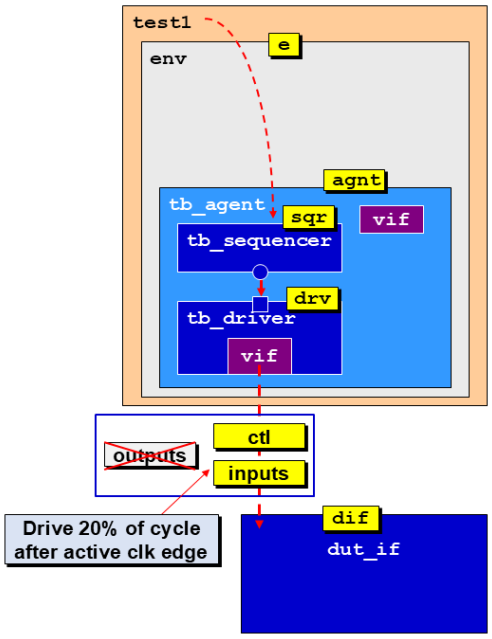


Figure 3 - Stimulus driving half of UVM testbench

In the output sampling half of the UVM testbench as shown in Figure 4, the **tb_monitor** will sample all the inputs and control signals on the active clock edge and sample the outputs at the end of the cycle, so the **tb_monitor** will use all the signals defined in the transaction.

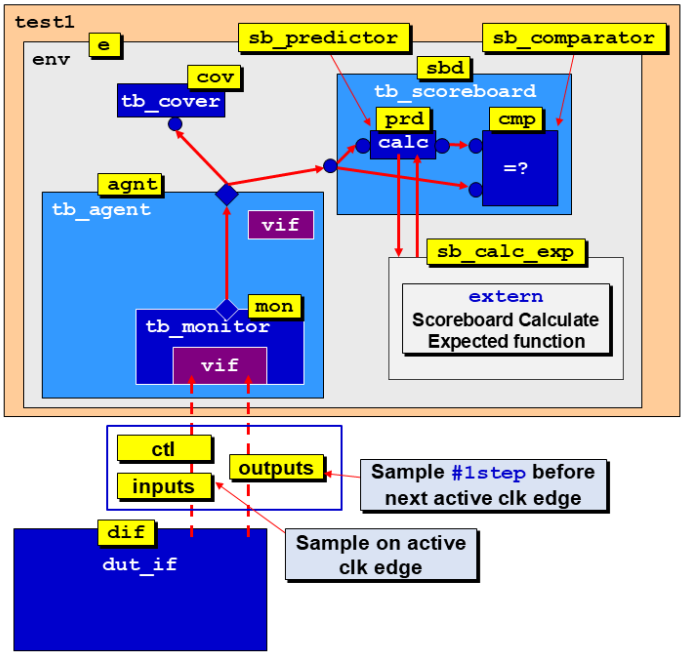


Figure 4 - Output sampling half of UVM testbench

The **sb_predictor** (scoreboard predictor) will pass the full transaction to the **sb_calc_exp()** function. As shown in Figure 5, the **sb_calc_exp()** function ignores the sampled output signals. The **sb_calc_exp()** function will take a copy of the broadcast transaction and use the sampled inputs and control signals to help generate the expected outputs. The expected transaction will then be broadcast over to the **sb_comparator** (scoreboard comparator).

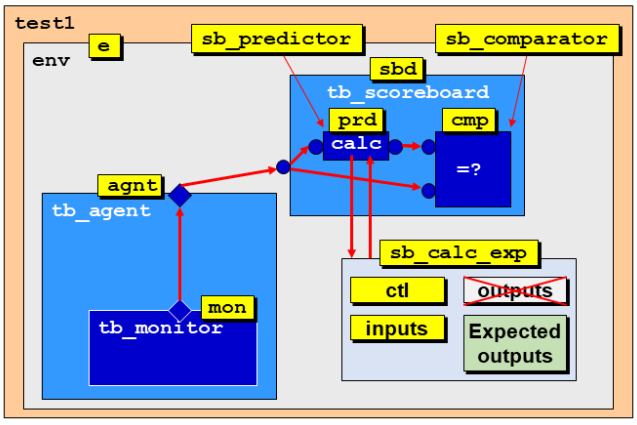


Figure 5 - Prediction logic of UVM testbench

As shown in Figure 6, the **sb_comparator** will get the sampled output transaction from the **tb_monitor** and will get the generated expected transaction from the **sb_predictor**, and then call the common **compare()** method, which was included in the transaction definition, to compare the outputs from the two transactions. The comparator will ignore the input and control signals present in

both the sampled and expected transactions.

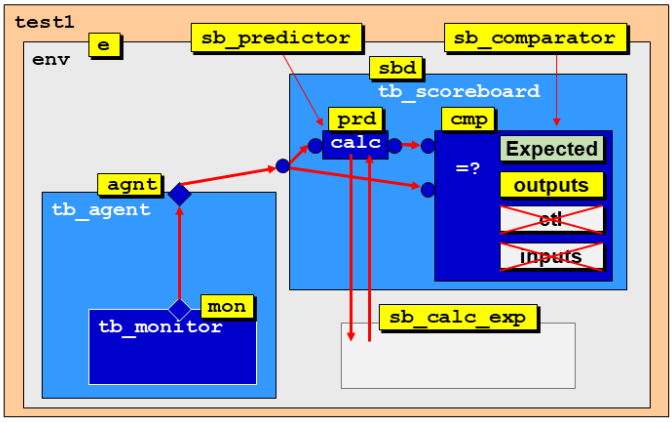


Figure 6 - Output comparison logic of UVM testbench

It can be seen that a common transaction definition is used or partially used through the entire UVM testbench.

5.3 Correct Stimulus & Verification Timing

The description of the 0-Timed Testbench section shows that there are three important testbench times that must be considered in any standard testbench [1].

(1) When should the stimulus be driven (from the stimulus driving half of the agent).
(2) When should the monitor sample the DUT inputs (part of the output sampling half of the agent).
(3) When should the monitor sample the DUT outputs (the other part of the output sampling half of the agent).

5.3.1 Clocking Block Time Budgeting

The clocking block can control the stimulus driving time (#1 from above) and the DUT output sampling time (#3 from above). The DUT input sampling time (#2 from above) should happen simply on the active clock edge.

The testbench timing is controlled from a `clocking` block that is placed in the DUT interface. It is important to recognize that the input/output descriptions in the clocking block are with respect to the testbench. Testbench outputs are the stimulus to the inputs to the design, while testbench inputs are the sampled signals from the design outputs. Effectively, testbench input and output directions are the opposite of design input and output directions, and the clocking block uses the testbench signal directions to specify timing.

Testbench sourced stimulus should never be driven on the active clock edge and there are many engineers that continue to make this mistake. This would be the same as changing the design inputs on the active clock edge in a real design, which would violate setup and/or hold times and probably cause metastability issues.

A properly timed UVM testbench would drive the stimulus after holding the last vector for 10-20% of the clock period. This would allow the required time to meet hold times in a gate level simulation (GLS) with delays and timing checks, plus allow 80-90% of the clock period to traverse any combinational inputs to the design to meet setup times in a GLS.

In the `uvmtb_template` files, the 10%-hold / 90%-setup or 20%-hold / 80% setup timing is controlled by a single number, as shown on line 10 of Example 2 (the example is shown on page 13 of this paper). The setup/hold time descriptions are included after the example.

5.3.2 The Program Block Mistake & Avoidance

The SystemVerilog **program** keyword “enhancement” was a mistake. The only advantage the **program** had over a **module** is that it allowed stimulus to be applied to design inputs on the active clock edge without simulation race conditions … WHICH YOU SHOULD NEVER DO! The justification for this statement is described in the previous section.

If you are currently using **program** blocks and applying stimulus on the active clock edge, REPENT! And use the timing recommendations shown in the previous section. Quit using the **program** statement in all testbenches (not just UVM testbenches) [1].

6. uvmtb_template Code

There are eight of the **uvmtb_template** files that require modification: **top.sv**, **dut_if.sv**, **trans1.svh**, **tb_driver.svh**, **tb_monitor.svh**, **sb_calc_exp.svh**, **tb_cover.svh**, and **tr_sequence.svh**. Those modifications are described in the sections below. As summarized in Section 6.2 large portions of each of these files are already coded. The ninth file, the **sb_comparator.svh**, only needs modification if the design's reset is synchronous or if the design has pipelined logic that must be flushed before the outputs become valid.

The file-extension naming conventions used in the templates are, files that are directly compiled use a "**.sv**" file extension, while files that use **`include** to compile them into package files use a "**.svh**" file extension. This is a common file naming convention on large verification projects but technically is not required.

6.1 Eight Template Files to be Modified

The eight (or nine) files that require modification, along with the modifications that are required are shown in this section. Note: even in the files that require modification, large portions of the files are pre-coded in the **uvmtb_template** files.

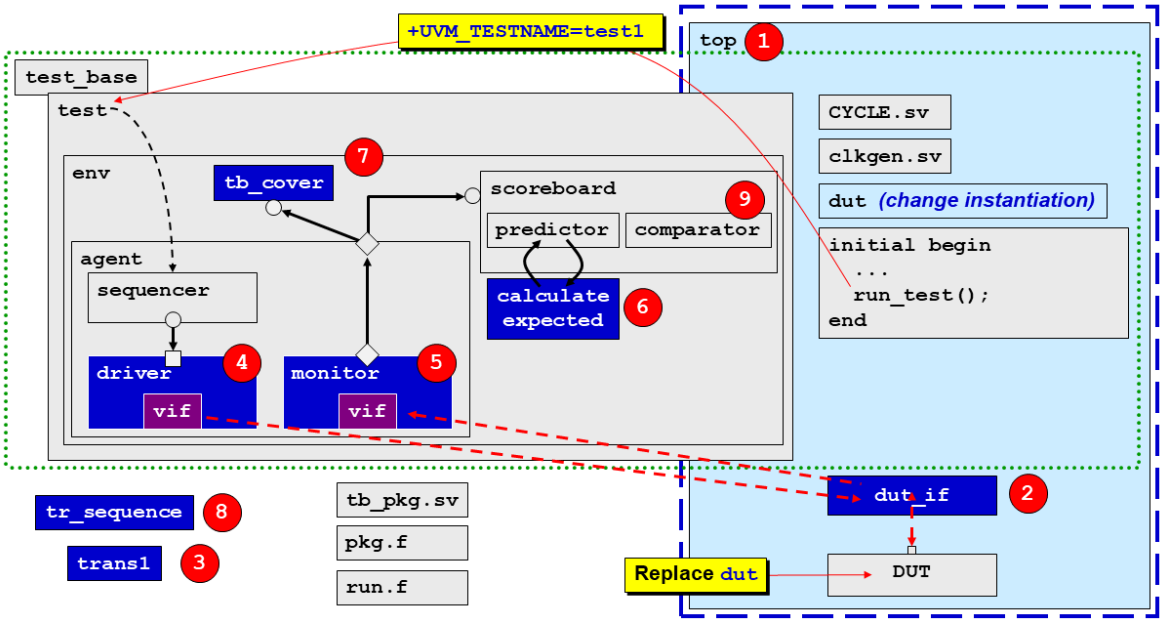


Figure 7 - uvmtb_template files that require modification

6.1.1 (1) top.sv

Replace the "dut" instantiation (lines 10-11) with the actual design instantiation (line 12). The template files include a dummy register-dut instantiated in the `top` module to make the template files compile and initially run. This dut-file should be deleted to prevent accidental usage.

```
 1 `include "CYCLE.sv"
 2 `include "uvm_macros.svh"
 3 module top;
 4   import uvm_pkg::*; // import uvm base  classes
 5   import  tb_pkg::*; // import testbench classes
 6   logic       clk;
 7
 8   clkgen         ck  (clk);
 9
10   dut            i1  (.dout(dif.dout), .din  (dif.din),
11                       .clk (clk),      .rst_n(dif.rst_n));
12   // instantiate actual DUT here
13
14   dut_if         dif (clk);
15
16   initial begin
17     uvm_resource_db#(virtual dut_if)::set("*", "vif", dif);
18     run_test();
19   end
20 endmodule
```

Example 1 - uvmtb_template – (1) top.sv file

Note that 17 of the 20 lines of code shown in this file typically do not require modification.

6.1.2 (2) dut_if.sv

Modify the interface signals (lines 5-7) to reflect the actual signals, sizes and data types to be connected to the dut and similarly modify the signals in the clocking block (lines 11-13) with all testbench driven signals (dut inputs) listed as outputs, and all testbench sampled signals (dut outputs) listed as inputs. The clocking block descriptions do not require size or data type information.

```
 1 `include "CYCLE.sv"
 2 `define Tdrive #(0.2*`CYCLE)
 3
 4 interface dut_if (input clk);
 5   logic [15:0] dout;
 6   logic [15:0] din;
 7   logic        rst_n;
 8
 9   clocking cb1 @(posedge clk);
10     default input #1step output `Tdrive;
11     input  dout;
12     output din;
13     output rst_n;
14   endclocking
15 endinterface
```

Example 2 - uvmtb_template – (2) dut_if.sv file

Line 2 defines the stimulus drive time to be 20% of the defined `` `CYCLE `` time (`#(0.2*`CYCLE)`).

If the stimulus drive time is required to be shortened to 10% of the `CYCLE delay, then an engineer simply changes 0.2 to 0.1, and all stimulus drive times will automatically scale to the 10% number.

Line 10 defines all clocking block-controlled testbench inputs (dut outputs) should be sampled at the last possible moment (#1step) at the end of the cycle, after all dut outputs have settled. Line 10 also specifies that the testbench stimulus drive times (dut inputs) should occur using the drive time definition (`Tdrive) shown on line 2. Any modifications to the drive time on line 2 will automatically update the timings in this clocking block.

As described at the beginning of Section 5.3 , (1) the testbench stimulus (dut inputs) is driven using the clocking block output `Tdrive specification, (2) the testbench inputs (inputs sampled by the dut on the active clock edge) are sampled on the active clock edge, *NOT* using the clocking block, and (3) the rest of the testbench inputs (dut outputs) are sampled using the clocking block input #1step specification.

Note that 9 of the 15 lines of code shown in this file typically do not require modification.

6.1.3 (3) trans1.svh

Make the following modifications to the trans1 transaction class definition:

Modify or add all the transaction signals (lines 4-6). Dedicated inputs and control signals are generally declared to be randomizable (rand or randc) signals.

Modify or add the signals (lines 16-18) referenced by the do_copy() method.

Modify or add the output signals to be compared (lines 25-26) used by the do_compare() method.

Modify or add the input and control signals to be printed (lines 36-37) with proper formatting by the input2string() method.

Modify or add the output signals to be printed (line 41) with proper formatting by the output2string() method.

The convert2sring() method (lines 44-46) simply returns the signals that were listed and formatted by the input2string() method concatenated to the signals listed and formatted by the output2string() method. The convert2string() method typically does not require any user-modification.

```
 1 class trans1 extends uvm_sequence_item;
 2   `uvm_object_utils(trans1)
 3
 4       logic [15:0] dout;    // outputs not randomized
 5   rand bit   [15:0] din;
 6   rand bit          rst_n;
 7
 8   function new (string name="trans1");
 9     super.new(name);
10   endfunction
11
12   function void do_copy(uvm_object rhs);
13     trans1 tr;
14     if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_copy() cast")
15     super.do_copy(rhs);
16     dout  = tr.dout;
17     din   = tr.din;
18     rst_n = tr.rst_n;
19   endfunction
20
21   function bit do_compare(uvm_object rhs, uvm_comparer comparer);
22     trans1 tr;
23     bit    eq;
24     if(!$cast(tr, rhs)) `uvm_fatal("trans1", "ILLEGAL do_compare() cast")
25     eq  = super.do_compare(rhs, comparer);
26     eq &= (dout === tr.dout);
27     return(eq);
28   endfunction
29
30   function void do_print(uvm_printer printer);
31     $display("\n\n\t\t*** print() and sprint() are not implemented ",
32              "for this transaction type ***\n\n");
33   endfunction
34
35   virtual function string input2string();
36     return($sformatf("din=%4h  rst_n=%b",
37                      din,     rst_n));
38   endfunction
39
40   virtual function string output2string();
41     return($sformatf("dout=%4h", dout));
42   endfunction
43
44   virtual function string convert2string();
45     return({input2string(), "  ", output2string()});
46   endfunction
47 endclass
```

Example 3 - uvmtb_template – (3) trans1.svh file

On Lines 4-6, designs with dedicated outputs are not randomized, while design inputs and control signals are typically specified to be randomizable (**rand** or **randc**) variables.

Note that 36 of the 47 lines of code shown in this file typically do not require modification.

6.1.4 (4) tb_driver.svh

Modify the stimulus driving methods **initialize()** (lines 24-25) and **drive_item()** (lines 31-32) with correct stimulus signals.

```
 1 class tb_driver extends uvm_driver #(trans1);
 2   `uvm_component_utils(tb_driver)
 3
 4   virtual dut_if vif;
 5
 6   function new (string name, uvm_component parent);
 7     super.new(name, parent);
 8   endfunction
 9
10   task run_phase(uvm_phase phase);
11     trans1 tr;
12     initialize();
13     forever begin
14       `uvm_info("DEBUG: Driver Run",  "... getting next item ...", UVM_FULL)
15       seq_item_port.get_next_item(tr);
16       drive_item(tr);
17       seq_item_port.item_done();
18       `uvm_info("DEBUG: Driver Run",     "... next item done ...", UVM_FULL)
19     end
20   endtask
21
22   virtual task initialize (); // @0 - Does not use clocking block
23     `uvm_info("INIT", "Initialize (time @0)", UVM_HIGH)
24     vif.rst_n <= '0;
25     vif.din   <= '1;  // Initialize din with 1's to verify that reset works
26     @vif.cb1;         // @(posedge vif.clk);
27   endtask
28
29   virtual task drive_item (trans1 tr);
30     `uvm_info("drive_item", tr.input2string(), UVM_FULL)
31     vif.cb1.rst_n <= tr.rst_n;
32     vif.cb1.din   <= tr.din;
33     @vif.cb1;         // @(posedge vif.clk);
34   endtask
35 endclass
```

Example 4 - uvmtb_template – (4) tb_driver.svh file

Note that 31 of the 35 lines of code shown in this file typically do not require modification.

Also note that on lines 14 and 18, the template file includes useful debug messages to show the **tb_driver** has retrieved and processed a transaction from the **tb_sequencer** if verbosity is set to **UVM_FULL** or higher.

Also note that on line 30, the template file includes a useful debug message to show the transaction being driven by the **tb_driver** if verbosity is set to **UVM_FULL** or higher. The transaction **input2string()** method is used to display the stimulus inputs and control signals, but does not show the dedicated outputs, which are typically uninitialized. Displaying uninitialized outputs can add confusion to the debug process.

6.1.5 (5) tb_monitor.svh

Modify the **sample_dut()** sampling method (lines 29-30 and 32-33) with correct sampled input and output signals.

The **sample_dut()** method that starts on line 26 is assumed to be already synchronized to the active clock edge (typically **posedge clk**) so inputs and control signals sampled on lines 29-30 are sampled directly from the virtual interface (**vif**) on the active clock edge. Then on line 31 of **sample_dut()**, the method resynchronizes to the next active clock edge using the **@vif.cb1** syntax. If the design has an *asynchronous* reset, then the test and assignment shown on line 32 is required. With asynchronous reset, the presence of an active reset must be tested both at the beginning and end of the cycle. Line 32 shows that the output(s) are sampled using **clocking** block timing at the end of the cycle.

```
 1 class tb_monitor extends uvm_monitor;
 2   `uvm_component_utils(tb_monitor)
 3
 4   virtual dut_if vif;
 5
 6   uvm_analysis_port #(trans1) ap;
 7
 8   function new (string name, uvm_component parent);
 9     super.new(name, parent);
10   endfunction
11
12   function void build_phase(uvm_phase phase);
13     super.build_phase(phase);
14     ap = new("ap", this);
15   endfunction
16
17   task run_phase(uvm_phase phase);
18     trans1 tr;
19     //-------------------------------------
20     forever begin
21       sample_dut(tr);
22       ap.write(tr);
23     end
24   endtask
25
26   task sample_dut (output trans1 tr);
27     trans1 t;
28     t = trans1::type_id::create("t");
29     t.din   = vif.din;
30     t.rst_n = vif.rst_n;
31     @vif.cb1;                // @(posedge vif.clk);
32     if (!vif.rst_n) t.rst_n = '0;
33     t.dout   = vif.cb1.dout;
34     tr      = t;
35     `uvm_info("sample_dut", tr.convert2string(), UVM_FULL)
36   endtask
37 endclass
```

Example 5 - uvmtb_template – (5) tb_monitor.svh file

Note that 33 of the 37 lines of code shown in this file typically do not require modification.

Also note that on line 35, the template file includes a useful debug message to show the sampled transaction that is being broadcast by the **tb_monitor** if verbosity is set to **UVM_FULL** or higher.

6.1.6 (6) sb_calc_exp.svh

Modify the calculate-expected **extern** function to predict the correct outputs based on sampled inputs (lines 10-13) and any saved state information that is stored as **static** variables (line 3) in the function. The **static** variables allow the **sb_calc_exp()** function to save any required reference-model state information that might be required the next time the **sb_calc_exp()** function is called.

By declaring the **sb_calc_exp()** function to be an **extern** function, it can be kept in a separate file. This means that the **sb_predictor** component file does not require any modification.

```
 1 function trans1 sb_predictor::sb_calc_exp(trans1 t);
 2
 3   static logic [15:0] ex_dout;
 4
 5   trans1 extr = trans1::type_id::create("extr");
 6   //---------------------------
 7   `uvm_info("CALC #1", t.convert2string(), UVM_FULL)
 8   extr.copy(t);
 9
10   if      (!extr.rst_n) ex_dout = '0;
11   else                  ex_dout = extr.din;
12
13   extr.dout = ex_dout;
14
15   `uvm_info("CALC #2", extr.convert2string(), UVM_FULL)
16   return(extr);
17 endfunction
```

Example 6 - uvmtb_template – (6) sb_calc_exp.svh file

Note that 12 of the 17 lines of code shown in this file typically do not require modification. This fille will typically require significant additional coding effort to mirror the values expected to come from the dut. This basically becomes the reference model for this block-level test.

Also note that line 7 shows the incoming transaction, and line 15 shows the calculated expected transaction generated by this function. These can be especially useful for debugging the **sb_calc_exp()** function if verbosity is set to **UVM_FULL** or higher.

6.1.7 (7) tb_cover.svh

Modify the **covergroup** as desired (lines 7-13) to perform functional coverage of sample signals as specified by the block-level test plan.

```
 1 class tb_cover extends uvm_subscriber #(trans1);
 2   `uvm_component_utils(tb_cover)
 3
 4   trans1 tr;
 5
 6   covergroup cg;
 7     option.per_instance = 1;
 8     option.at_least     = 10;
 9     dout      : coverpoint tr.dout  {bins dout[8]  = {[0:$]};}
10     din       : coverpoint tr.din   {bins  din[8]  = {[0:$]};}
11     rst       : coverpoint tr.rst_n {bins dorst    = {'0};}
12     norst     : coverpoint tr.rst_n {bins norst    = {'1};}
13     doutXnorst: cross dout, norst;
14   endgroup
15
16   function new (string name, uvm_component parent);
17     super.new(name, parent);
18     cg = new();
19   endfunction
20
21   function void write (trans1 t);
22     tr = t;
23     `uvm_info("tb_cover", "Taking covergroup sample ...", UVM_FULL)
24     cg.sample();
25   endfunction
26 endclass
```

Example 7 - uvmtb_template – (7) tb_cover.svh file

Note that 19 of the 26 lines of code shown in this file typically do not require modification. This fille will typically require significant additional code to capture the desired functional coverage information for this UVM testbench.

Also note that on line 23, the template file includes a useful debug message to show the sampled transaction that that was broadcast by the **tb_monitor** to this coverage collector if verbosity is set to **UVM_FULL** or higher.

6.1.8 (8) tr_sequence.svh

Modify the `tr_sequence` (lines 11 and 17) to perform some level of proper testing.

```
 1 class tr_sequence extends uvm_sequence #(trans1);
 2   `uvm_object_utils(tr_sequence)
 3
 4   function new (string name = "tr_sequence");
 5     super.new(name);
 6   endfunction
 7
 8   task body;
 9     trans1 tr = trans1::type_id::create("tr");
10     //---------------------------------
11     repeat(100) do_item(tr);
12   endtask
13
14   task do_item (trans1 tr);
15     `uvm_info("do_item", "executing", UVM_FULL)
16     start_item(tr);
17     if (!(tr.randomize() with {tr.rst_n=='1;}))
18         `uvm_fatal("TR_S", "tr_sequence randomization failed")
19     `uvm_info("do_item", tr.input2string(), UVM_FULL)
20     finish_item(tr);
21   endtask
22 endclass
```

Example 8 - uvmtb_template – (8) tr_sequence.svh file

Note that 20 of the 22 lines of code shown in this file typically do not require modification. Initial block-level testing can frequently use this `tr_sequence` without any modification, but later block-level testing will likely enhance this `tr_sequence` to perform additional, interesting test activity.

Also note that on line 19, the template file includes a useful debug message to show the randomized transaction that will be processed by the `tr_sequence` if verbosity is set to `UVM_FULL` or higher.

6.1.9 (9) sb_comparator.svh

The **sb_comparator** is fully coded and typically requires no modification.

If the design has *synchronous* reset signals or if the design has pipelined logic that must be flushed on startup, modify lines 8-11 in the **run_phase()** by uncommenting the **repeat**-loop and modifying the **repeat**-count value to express how many samples should be sampled but ignored.

When uncommented, lines 9 and 10 get the next expected (**exp_tr**) and sampled output (**out_tr**) transactions but do nothing with them; hence, they are effectively discarded.

```
 1 class sb_comparator extends uvm_component;
 2   `uvm_component_utils(sb_comparator)
 3   trans1 exp_tr, out_tr;
 4   static int VECT_CNT, PASS_CNT, ERROR_CNT;
 5
 6   ...
 7   task run_phase(uvm_phase phase);
 8     // repeat (1) begin
 9     //   expfifo.get(exp_tr); // Throw away expected sample
10     //   outfifo.get(out_tr); // Throw away output   sample
11     // end
12     forever begin
13       `uvm_info("SB_CMP", "WAITING for expected output", UVM_FULL)
14       expfifo.get(exp_tr);
15       `uvm_info("SB_CMP", "WAITING for actual output", UVM_FULL)
16       outfifo.get(out_tr);
17       if (out_tr.compare(exp_tr)) PASS();
18       else                        ERROR();
19     end
20   endtask
21   ...
22
23 endclass
```

Example 9 - uvmtb_template – (9) sb_comparator.svh file

6.2 Summarizing template-file code

Even though the user is required to modify eight files to complete a UVM block-level testbench, in these subsections of Section 6. we have seen that for each of these block-level testbenches:

(1) **top.sv**: 17 of the 20 lines of code are already included in the template file.
(2) **dut_if.sv**: 9 of the 15 lines of code are already included in the template file.
(3) **trans1.svh**: 36 of the 47 lines of code are already included in the template file.
(4) **tb_driver.svh**: 31 of the 35 lines of code are already included in the template file.
(5) **tb_monitor.svh**: 33 of the 37 lines of code are already included in the template file.
(6) **sb_calc_exp.svh**: 12 of the 17 lines of code are already included in the template file.
(7) **tb_cover.svh**: 19 of the 26 lines of code are already included in the template file.
(8) **tr_sequence.svh**: 20 of the 22 lines of code are already included in the template file.

For all eight files, 177 of the 219 lines of code, or 81% of the initial code, are already included in the template files. This is one reason the template files greatly simplify UVM testbench development.

6.3 Fourteen More Fully-Coded & Usable Template Files

The `uvmtb_template` directory also includes 14 more files that can typically be used without modification to create a UVM testbench for block-level testing.

(1)	`clkgen.sv`	Clock generator with 50% duty cycle
(2)	`CYCLE.sv`	Defines the clock cycle to be 10ns. This can be user-modified
(3)	`dut.sv`	~~Simple register DUT.~~ *This file should typically be deleted*
(4)	`env.svh`	Typical environment - fully coded
(5)	`pkg.f`	Package file to compile testbench classes
(6)	`run.f`	Command file that references all required simulation files
(7)	`sb_comparator.svh`	Scoreboard style #1 comparator file - fully coded
(8)	`sb_predictor.svh`	Scoreboard style #1 predictor file - fully coded
(9)	`tb_agent.svh`	Typical agent - fully coded
(10)	`tb_pkg.sv`	Typical package - more tests and sequences can be added
(11)	`tb_scoreboard.svh`	Scoreboard style #1 wrapper
(12)	`tb_sequencer.svh`	Typical sequencer - fully coded
(13)	`test_base.svh`	Useful test base class - fully coded
(14)	`test1.svh`	Typical test that starts the `tr_sequence`

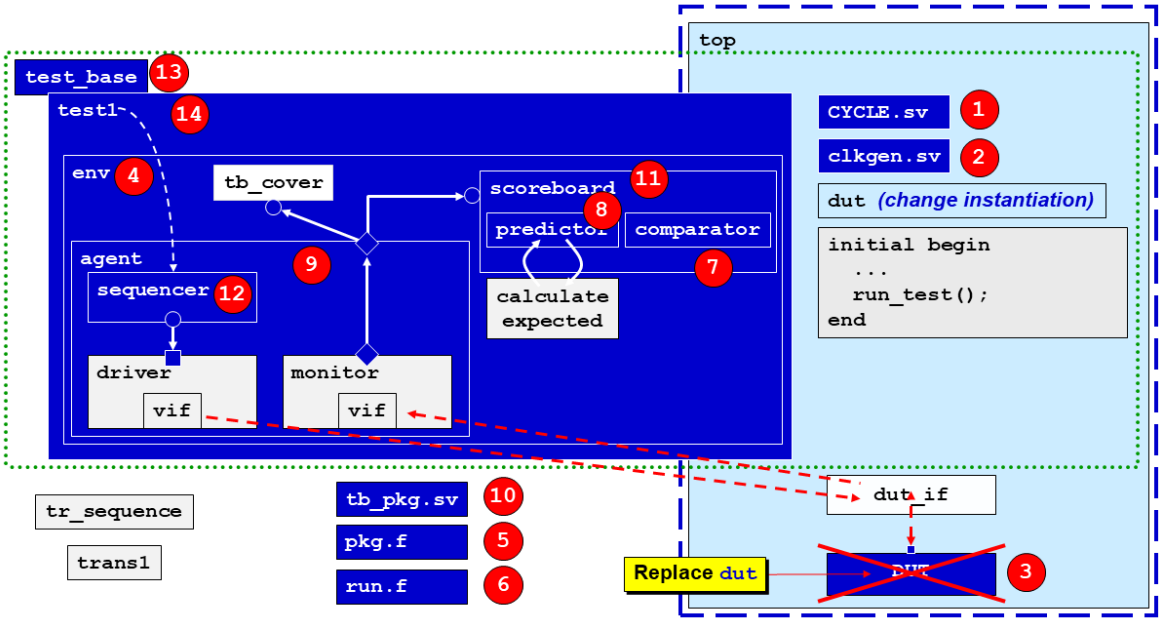


Figure 8 - uvmtb_template files that do not require modification

7. UVM Testbench Structure Easier than SV

If an entire UVM testbench had to be coded from scratch for each block-level testbench, one could make the reasonable argument that UVM testbenches are harder to code than Verilog testbenches. As shown in the previous sections, with a simple and powerful set of 23 template files, design and verification engineers can quickly and easily assemble a block-level UVM testbench with much less effort than creating a Verilog or SystemVerilog testbench from scratch.

Based off my personal experience of using and teaching UVM testbench development for more than a decade, I have found that a proper set of user-defined template files makes it easier to assemble and deploy a UVM testbench than it is to assemble and use Verilog or SystemVerilog for simple

testbench development.

Another nice feature of the UVM testbench template files is that you and your company can modify this starting set of template files with features that are important to your design and verification teams. *You* get to control the content of the reusable template files. This is not controlled by a third-party UVM testbench generation tool.

7.1 UVM Scoreboards Are Easier than SV

My experience has also shown that creating UVM scoreboards using the **`uvmtb_template`** files is also much easier than creating Verilog or SystemVerilog scoreboards.

Transaction definitions should already include a properly coded **`compare()`** method. The engineer who codes the transaction item should have identified the fields that should be compared when running tests.

For designs that include dedicated inputs and dedicated outputs, there typically is no reason to compare the inputs. The outputs and possibly some input-control signals should be compared.

Note that comparing certain output fields might be conditional. Consider the case of synchronous FIFO testing. When the FIFO empty flag is false, the dataout fields *should be compared*. When the empty flag is true, the dataout fields are invalid and *should not be compared*.

When defining the transaction using UVM field macros, this is not easy to specify. When defining the transaction using the **`do_compare()`** method, it is easy to unconditionally compare most of the outputs while conditionally comparing the dataout field only when the FIFO is not empty.

Using the **`do_compare()`** method, also makes it easy to add conditional printed messages for specifically detected commands or conditions, which might help verification engineers understand and debug any actions that routinely occur during normal design operation.

For packet-based designs, the **`do_compare()`** method can be coded to examine all the packet bits that are input to, and output from the design.

In a Verilog testbench, verification engineers typically add the comparison functionality while coding the scoreboard. There is typically no built-in **`compare()`** method that can be called.

Using the **`uvmtb_template`** code there are four files that are included in the creation of a common UVM scoreboard. Three of the files are pre-coded and typically do not need to be modified. Those files are:

(1) **`tb_scoreboard.svh`** (testbench scoreboard), a wrapper class that includes the pre-coded **`sb_predictor`** and **`sb_comparator`**.
(2) **`sb_predictor.svh`** (scoreboard predictor), which loops taking the sampled transaction broadcast from the monitor, passes it to the **`sb_calc_exp()`** function and takes the returned expected transaction and broadcasts it to the **`sb_comparator`**.
(3) **`sb_comparator.svh`** (scoreboard comparator), which loops taking each sampled transaction that was broadcast from the monitor and calls the transaction **`compare()`** method to compare the appropriate fields to the corresponding fields in the expected transaction that was broadcast from the **`sb_predictor`**.

The fourth scoreboard file is the **`sb_calc_exp.svh`** (scoreboard calculate expected function) file, which is an **`extern`** function called by the **`sb_predictor`**. Making the **`sb_calc_exp()`** function an **`extern`** function in a separate file means that the **`sb_predictor`** file is fully coded and does not require modification. The **`sb_calc_exp.svh`** file is the only scoreboard file that typically needs to be modified for simple block-level testbenches.

The **`sb_calc_exp()`** function uses the inputs and control signals from the transaction sampled by

the **tb_monitor** to calculate the expected outputs, while the transaction outputs sampled by the **tb_monitor** are ignored. It is the job of the **sb_calc_exp()** function to calculate what the DUT output values should be based on (1) the sampled inputs, (2) the sampled control signals, and (3) any saved **static** state information saved by a previous call to the **sb_calc_exp()** function. Since a UVM testbench should never modify the signals of the transaction that was broadcast from the **tb_monitor**, the **sb_calc_exp()** function will declare and factory create its own local expected transaction (**extr**).

The **sb_calc_exp()** function is composed of the following elements and actions:

- Declare any necessary state variables as **static**. Any function field that relies on a previous state value must be declared **static** so that it can be used in subsequent calls to the **sb_calc_exp()** function.
- Create a local transaction called **extr** (expected-transaction).
- Copy the incoming transaction to the **extr**. The incoming transaction was broadcast from the **tb_monitor**.
- Test the **extr** inputs and control signals to calculate and assign the expected outputs to the expected transaction (**extr**). The **sb_calc_exp()** function ignores any sampled outputs.
- Return the **extr** transaction to the **sb_predictor**, which will broadcast the **extr** to the **sb_comparator**.

8. uvmtb_template MIT Licensed

The **uvmtb_template** files are copyrighted under the MIT License instead of the GPL License.

The MIT License is a much more user-friendly license since it allows the user to freely use, modify and share the **uvmtb_template** files without an obligation to send the modifications back to the originator of the files.

Quoting from the Endor Labs Blog website [4]:

"One of the key differences between the MIT and GPL licenses is how they handle derivative works. With the MIT license, derivative works can be licensed under any terms, while the GPL license requires that derivative works must be released under the same license. This means that when using the MIT license, a developer can use the code in a commercial product and keep the source code closed, while the GPL license forces to release the code and any modifications made to it under the same license."

A long-time complaint I have had about GPL licenses in source code is the verbose comment block at the beginning of each GPL-licensed file.

Colleagues from Emerson (see names in Acknowledgements - Section 12. pointed me to The Linux Foundation Projects website, which describes a concise way to add the MIT license information to the source files using SPDX License IDs. Each **uvmtb_template** files includes the concise ID string:

```
// SPDX-License-Identifier: MIT
```

For more information about SPDX License IDs and their usage, readers are encouraged to visit the appropriate "The Linux Foundation Projects" webpage shown in reference [5].

The **uvmtb_template** source files are freely available on Github [6] at https://github.com/paradigm-works/uvmtb_template

9. Teaching UVM testbenches

If you want users to quickly learn UVM testbench techniques, they need practice building multiple

simple UVM full, self-checking testbenches.

Teaching UVM is not like teaching most programming languages, including Verilog and SystemVerilog. Typical programming languages can be taught from basic-to-advanced, from start-to-end. Teaching UVM does not lend itself to this teaching approach. To most users, every UVM concept, structure and command is new, and it is difficult to see how the pieces fit together.

The strategy I recommend for teaching UVM is patterned after the teaching style for one of the best classes I took at BYU during my undergraduate studies. That class was Differential Equations, or 4th semester calculus. What made the class so good was the teaching style of the Professor. The professor would go through each chapter three times. He would go through the chapter quickly the first time and assign the easy homework problems, He would then go quickly through the chapter a second time and assign the hard homework problems. Then he would go through special topics from the chapter a third time and assign more of the hard homework problems. By the time we had been through the chapter three times, we had a good understanding of the material.

I have taken the same approach to teaching UVM. In my training class, we take our first full path through UVM in section 2 of the training class and engineers are assigned their first full self-checking UVM testbench on day-1. Engineers are told that they will not fully understand all the UVM concepts during this first pass but not worry, because they will go through all the concepts two, three or four more times before the conclusion of the class. The objective of this training style is to get engineers through as many full self-checking UVM testbenches as they can in a 3-day training class. The goal is that engineers on the last day of class can say, “Cliff, this UVM-stuff is so boring, because it is so easy!” The only way to achieve this goal is to go through the concepts multiple times so that students see how the parts fit together and relate to each other, and to give engineers lots of practice coding and running full self-checking UVM testbenches.

10. Summary of uvmtb_template Files That Require Modification

This section includes a summary list of **`uvmtb_template`** files that require modification. I keep this list and the diagram shown in Figure 7 taped to the wall in my office. I also have a Linux alias called "**`viuvm`**" that opens in Vim the first eight of these files in the order shown. This allows me to go down the list and quickly create simple UVM testbenches.

These are typical **`uvmtb_template`** modifications.

(1) **`top.sv`**: Replace "**`dut`**" instantiation with actual design instantiation.

(2) **`dut_if.sv`**: Modify signals in the interface and in the **`clocking`** block with the correct signals.

(3) **`trans1.svh`**: Modify signals and all standard methods.

(4) **`tb_driver.svh`**: Modify driving methods **`initialize()`** and **`drive_item()`** with correct stimulus signals.

(5) **`tb_monitor.svh`**: Modify **`sample_dut()`** sampling method with correct sampled input and output signals.

(6) **`sb_calc_expect.svh`**: Modify the calculate-expected extern function to predict correct output based on sampled inputs.

(7) **`tb_cover.svh`**: Modify **`covergroup`** to sample signals as desired.

(8) **`tr_sequence.svh`**: Modify the sequence(s) to do proper testing.

(9) **`sb_comparator.svh`**: **IF** the design has *synchronous* reset or piped logic to be flushed - Modify the **`run_phase()`** - uncomment pre-**`forever repeat`**-loop & modify count value.

11. Conclusions

The more I use UVM, the more I recognize that very smart verification engineers developed a methodology that incorporated very smart and powerful verification capabilities.

Engineers who split time between simple Verilog or SystemVerilog testbenches, and simple UVM testbenches, are generally less efficient at both testbench coding styles, and are not developing important UVM testbench skills as quickly as they could if they focused on using the `uvmtb_template` template files described in this paper.

The more I use UVM, the more I find that, with a small set of template files, I can assemble simple, yet powerful UVM testbenches easier than I could ever assemble Verilog or SystemVerilog self-checking testbenches.

Using third party UVM testbench generation tools works, but I always seem to find them lacking features I want to commonly add to a UVM testbench, plus the generated code is often needlessly complex for simple block-level testbench generation.

The `uvmtb_template` files can be freely downloaded from the website: https://github.com/paradigm-works/uvmtb_template

Use the `uvmtb_template` files to simplify your UVM testbench development efforts and enhance your UVM skills!

12. Acknowledgements

I am grateful to my long-time friend and colleague, Ronald Goodstein for his very thorough review of the draft version of this paper. Ron caught many typographical errors and a duplicate/missing figure. This paper is a more polished work due to Ron's review.

I am also grateful to Olivia Poon for her comments that improved the content of the paper.

I am also grateful to my colleagues from Emerson, Wade Fife, Andrew Moch, and Kevin Cuzner, for their recommendations to put the `uvmtb_template` files under the MIT License and to identify the license by adding the concise "`//SPDX License-Identifier: MIT`" SPDX License ID to the top of each file.

13. References

[1] Clifford E. Cummings, "Applying Stimulus & Sampling Outputs - UVM Verification Testing Techniques," SNUG (Synopsys Users Group) 2016 (Austin, TX). Also available at: www.sunburst-design.com/papers/CummingsSNUG2016AUS_Verification TimingTesting.pdf

[2] Clifford E. Cummings, "UVM Analysis Port Functionality and Using Transaction Copy Commands," SNUG (Synopsys Users Group) 2018 (Austin, TX). Also available at: www.sunburst-design.com/papers/CummingsSNUG2018AUS_UVMAnalysisCopy.pdf

[3] Clifford E. Cummings, "UVM Message Display Commands – Capabilities, Proper Usage and Guidelines," SNUG (Synopsys Users Group) 2014 (Austin, TX). Also available at: www.sunburst-design.com/papers/CummingsSNUG2014AUS_UVM_Messages.pdf

[4] Ron Harnik, "Open Source Licensing Simplified: A Comparative Overview of Popular Licenses," January 24, 2023, Endor Labs Blog. https://www.endorlabs.com/learn/open-source-licensing-simplified-a-comparative-overview-of-popular-licenses

[5] "SPDX – Handling License Info," The Linux Foundation Projects, https://spdx.dev/learn/handling-license-info/

[6] The `uvmtb_template` source files are freely available for download at the GitHub website: https://github.com/paradigm-works/uvmtb_template

14. Author & Contact Information

Sunburst Design World Class Training

Sunburst Design merged with Paradigm Works in February of 2020 and still provides World Class SystemVerilog, Synthesis and UVM Verification training. For more information about SystemVerilog and UVM training, contact Cliff Cummings (cliffc@sunburst-design.com) or Michael Hoyt (michael.hoyt@paradigm-works.com)

Paradigm Works Expert Design Services, Verification Services & IP

Paradigm Works provides expert services in Semiconductor Architecture, Design, Synthesis, Functional Verification, and DFT. For more information about Paradigm Works services, contact Michael Hoyt at michael.hoyt@paradigm-works.com

Cliff Cummings is Vice President of Training at Paradigm Works and Founder of Sunburst Design. Paradigm Works and Sunburst Design merged in February of 2020. Cliff has more than 40 years of ASIC, FPGA and system design experience and more than 30 years of combined Verilog, SystemVerilog, UVM verification, synthesis, and methodology training experience.

Cliff has presented more than 100 SystemVerilog seminars and training classes in the past 20 years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Cliff participated on every IEEE & Accellera SystemVerilog, SystemVerilog Synthesis, SystemVerilog committee from 1994-2012, and has presented more than 50 papers on SystemVerilog & SystemVerilog related design, synthesis, and UVM verification techniques.

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

Email address: cliffc@sunburst-design.com

Last Updated: April 2025

15. Appendix I – Printing Testbench Structure & Factory Contents

The `test_base` template file includes a cool trick that I documented in a SNUG Austin paper back in 2014 [3]. Turning up verbosity to `UVM_HIGH` or higher will print out your UVM testbench structure and the classes that you have registered with the factory.

```
function void start_of_simulation_phase(uvm_phase phase);
  super.start_of_simulation_phase(phase);
  ...
  if (uvm_report_enabled(UVM_HIGH)) begin
    this.print;
    factory.print;
  end
endfunction
```

Example 10 - Conditional display of test configuration and factory configuration

All tests that extend the `test_base` class include this useful capability.

16. Appendix II - Running UVM Simulations

For those who might be new to running UVM simulations, especially with the **uvmtb_template** template files, follow the steps described below for your chosen simulator.

The **uvmtb_template** file includes two command files, **pkg.f** and **run.f**.

Using the **pkg.f** command file is optional and is intended to be used to compile all of the package files as a preliminary step to debug most of the files before running a simulation. Experience has shown that coding multiple files for a UVM testbench typically will introduce simple typos that are more easily debugged before trying to run the full simulation.

The **run.f** command file is used to run the simulation.

Running a UVM test requires the test name to added to the command line using the **+UVM_TESTNAME** command line switch to include the requested test as shown in the examples below.

16.1 VCS-Specific Command Line Switches

The VCS version used to test the command line switches was:

```
version T-2022.06-SP1-1_Full64
```

The following are VCS-specific command line switches used to run UVM simulations:

-full64 Runs 64-bit version of UVM. Many companies have already aliased this switch into their VCS command and the switch does not have to be included if that is true. If the **-full64** switch is missing, the following compilation error is reported:

```
g++ error: /home/vcs/linux/lib/ctype-stubs_32.a: No such file or directory
```

-sverilog Required to compile SystemVerilog code, including all the classes used by UVM.

-ntb_opt uvm Specifies *Native Testbench Options UVM*, to use the precompiled UVM testbench classes used by VCS.

-timescale=1ns/1ns In SystemVerilog, if any of the compiled files includes a **`timescale** directive, then SystemVerilog requires that the first compiled file must also include a **`timescale** directive. The UVM libraries do not include a **`timescale** and they are compiled first, so the **-timescale** command line switch sets a default **timescale** before compiling any files.

-R Runs the simulation immediately if the design compiled successfully.

16.1.1 VCS - 2-step simulation

2-step simulation allows the design with all tests to be compiled once and then each test can be run without recompilation.

To check for initial syntax errors, first compile with the **pkg.f** command file (this step is optional):

```
vcs -full64 -sverilog -f pkg.f -ntb_opts UVM -timescale=1ns/1ns
```

To compile the testbench for simulation, use the **run.f** command file:

```
vcs -full64 -sverilog -f run.f -ntb_opts UVM -timescale=1ns/1ns
```

The compilation step will create a **simv** executable.

To run the compiled simulation executable with **UVM_HIGH** verbosity, use the following command:

```
simv +UVM_TESTNAME=test1 +UVM_VERBOSITY=HIGH
```

To re-run the compiled simulation with the lower **UVM_MEDIUM** verbosity, use the following command:

```
simv +UVM_TESTNAME=test1 +UVM_VERBOSITY=MEDIUM
```

16.1.2 VCS - 1-step simulation

The 1-step simulation compiles the design and all tests and then immediately runs (**-R**) the simulation with **UVM_HIGH** verbosity after successful compilation. Running the 1-step command recompiles the UVM library each time the command is executed, so if none of the existing files have changed and if no new files are being compiled, you are paying a recompilation-time penalty by running the 1-step simulation.

```
vcs -full64 -sverilog -f run.f -R -ntb_opts UVM -timescale=1ns/1ns +UVM_TESTNAME=test1 +UVM_VERBOSITY=HIGH
```

16.2 QuestaSim-Specific Command Line Switches

The QuestaSim version used to test the command line switches was:

```
QuestaSim 2023.1
```

The following are QuestaSim-specific command line switches used to run UVM simulations:

-sv	Required to compile SystemVerilog code, including all the classes used by UVM.
-mfcu	If Verilog macros are defined separately from the file that uses the macros, QuestaSim requires a command line switch to specify that all files compiled must make up a single compilation unit. **-mfcu** stands for *Multi-File Compilation Unit*, which makes the macros from another file visible in a different file that references them.
-c	Run in command-line mode (batch mode) without bringing up the **vsim** GUI.
-do "run -all"	Do the **run -all** command to run the entire simulation in batch mode.
-R	What follows **-R** are the runtime command options.

16.2.1 QuestaSim - 3-step simulation

3-step simulation allows the design with all tests to be compiled once and then each test can be run without creating a **work** directory and without recompilation.

Create the **work** directory. This step is only required once per simulation directory.

```
vlib work
```

To check for initial syntax errors, first compile with the **pkg.f** command file (this step is optional):

```
vlog -sv -mfcu -f pkg.f
```

If there are no syntax errors, the compilation output will report:

```
Top level modules: --none--
```

To compile the testbench for simulation, use the **run.f** command file:

```
vlog -sv -mfcu -f run.f
```

If the design and UVM testbench successfully compile, the compilation output will report:

```
Top level modules: top (or whatever the top-module name is called)
```

The top-level module name is used in the following simulation command:

```
vsim -c -do "run -all" top _UVM_TESTNAME=test1 +UVM_VERBOSITY=HIGH
```

16.2.2 QuestaSim - 1-step simulation

The 1-step simulation compiles the design and all tests and then immediately runs the simulation with **`UVM_HIGH`** verbosity after successful compilation. Running the 1-step command recompiles the UVM library each time the command is executed, so if none of the existing files have changed and if no new files are being compiled, you are paying a recompilation-time penalty by running the 1-step simulation.

```
qverilog -sv -mfcu -f run.f -R +UVM_TESTNAME=test1 +UVM_VERBOSITY=HIGH
```

The runtime options **`+UVM_TESTNAME=test1 +UVM_VERBOSITY=HIGH`** must be placed on the command line after the **`-R`** (runtime) switch)

16.3 Xcelium-Specific Command Line Switches

The following are Cadence Xcelium-specific command line switches used to run UVM simulations:

`+sv`	Used to compile SystemVerilog code, including all the classes used by UVM.
`-uvm`	Used to use the precompiled UVM testbench classes used by Xcelium.

16.3.1 Xcelium - 1-step simulation

The 1-step simulation compiles the design and all tests and then immediately runs the simulation with **`UVM_HIGH`** verbosity after successful compilation.

```
xrun +sv -uvm -f run.f +UVM_TESTNAME=test1 +UVM_VERBOSITY=HIGH
```