Verification without DUT

Thinh Ngo

Broadcom Corporation
Austin, Texas, USA

ABSTRACT

The more testbench development completed before a DUT is ready, the less left to be done afterwards. However, pre-DUT testbench development is limited by the inability to simulate and debug.

To boost pre-DUT testbench development, a virtual DUT can be used to enable testbench simulation and debugging. The virtual DUT provides randomized legal responses to testbench (i.e. stimulus generators, responders, coverage, assertions). With interface protocol-compliant responses, transactions can flow between the testbench and the virtual DUT.

The virtual DUT has agents for DUT stimulus, response and state interfaces. Each agent consists of a driver and a sequencer. Interfaces can be shared between the testbench and the virtual DUT. Similarly, drivers can mirror the testbench drivers. Responses are randomized, independent of stimulus contents, focusing on completeness. Testbench stimulus generators, monitors, checkers, responders, assertions, and coverage can be independently simulated, tested and debugged. Failures will be bypassed. Post-DUT testbench adjustments are expected.

Table of Contents

Table of Figures

1. Introduction

Testbench development can be divided into two stages: before and after the DUT is available (e.g. pre-DUT and post-DUT). Post-DUT stage can be further sub-divided into three stages namely the bring-up, the stability and the fine-tune stages. In the pre-DUT stage, testbench coding is performed without testing/debugging as simulation cannot run without a DUT. In the bring-up stage, the DUT is incrementally released - allowing testing/debugging of the testbench and ramping up verification. In the stability stage, the testbench is mostly mature and the verification is at full speed to verify the completed DUT. Finally, in the fine-tune stage, the testbench is targeting at corner cases and at achieving required coverage. Figure 1 below shows a typical testbench development cycle demonstrating coverage vs. time relationship of these stages.

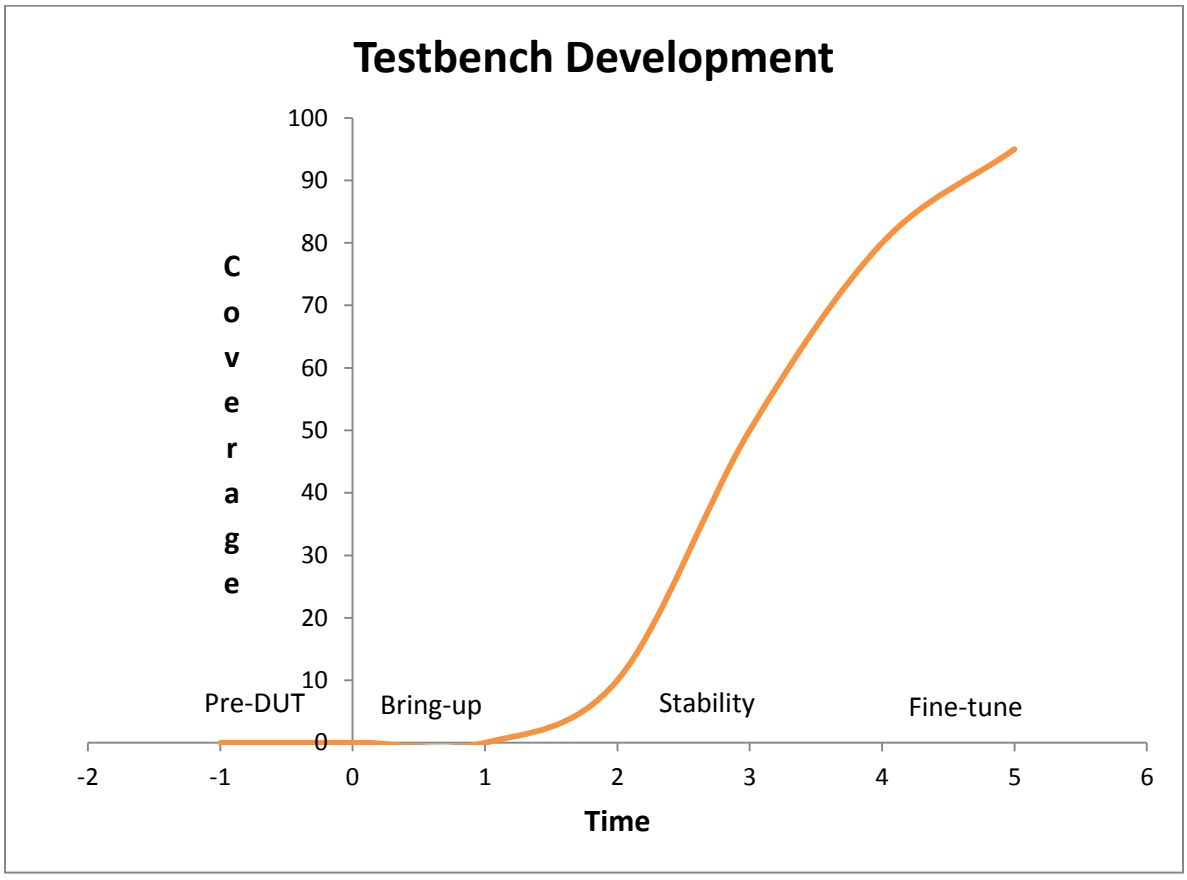


Figure 1 Testbench Development Cycle

In this paper, we present a methodology to boost pre-DUT testbench development productivity which will shorten all the three post-DUT stages.

2. Two opportunities in post-DUT stage

There are two opportunities to reduce the post-DUT testbench development time. With these techniques, verification closure time will be reduced.

In the post-DUT stage, a significant amount of time is spent testing and debugging testbench work done in the pre-DUT stage. The gap exists from the time the DUT is dropped (e.g. DUT readiness) and the time the testbench is ready to verify it (e.g. testbench readiness). This gap occurs every time there is an incremental DUT block. This DUT-to-testbench readiness gap can be a few days to several weeks depending on how much pre-DUT work has done and how extensive the DUT drop is. The first opportunity is to reduce this testbench readiness gap in the post-DUT stage.

In the stability stage, stimulus generation and checkers are the focus. In the fine-tune stage, coverage (e.g. cover-groups, cover assertions) is developed and its data are used to update stimulus generation and checkers. If the coverage can be made available sooner, less time is needed to develop complete stimulus generation and checkers. The gap exists from the time the DUT is ready to the time coverage data is ready. Typically, this coverage readiness gap unnecessarily extends beyond the testbench readiness. The second opportunity is to reduce coverage readiness in the post-DUT stage. Figure 2 shows a typical RTL, testbench and coverage development timelines.

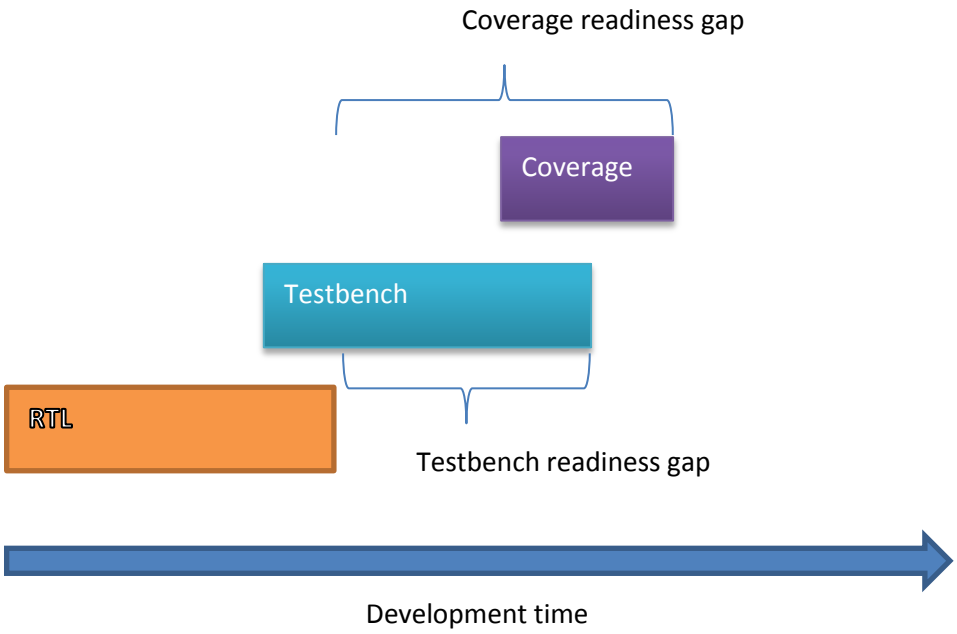


Figure 2 RTL, Testbench & Coverage Development Time

3. Pre-DUT testbench can be simulated and tested

The more work that is done in the pre-DUT stage the less is left to be done post-DUT. Additionally, time spent for pre-DUT work is not part of the verification closure time as it overlaps with that for design development (e.g. RTL coding). Therefore, it is preferable to have more work done in the pre-DUT stage. However, the pre-DUT work that could carry-over to the post-DUT is eventually limited by the inability to simulate and therefore to test and debug developing testbench code. Actually, simulation is possible without a DUT. If we shift the focus of interest from the DUT to the testbench, we can create another component in place of the DUT to allow simulation to take place and therefore to test the testbench. Hence, the testbench does the role of the DUT – device under test – and another component is needed to verify this new DUT. The new DUT consists of stimulus generators, monitors, checkers, responders, coverage and assertions.

For stimulus generators to continuously generate stimulus transactions, continuous provision of proper responses (i.e. acknowledges) is needed. Similarly, responders, DUT state coverage and white-box checkers require a continuous feed of desired transactions to operate and simulate. Stimulus and response monitors and coverage are exercised and tested via generated transactions on the DUT interface. Scoreboards, or black-box checkers, are exercised and tested from stimulus and response transactions fed from corresponding monitors. The component that is used to verify the testbench will generate these desired response transactions.

In order to provide proper responses, DUT interface protocols need to be defined. These include stimulus interfaces for testbench stimulus generators, response interfaces for testbench responders and state interfaces for functional coverage (e.g. covergroups and cover assertions), property assertions and checkers. Most likely these interfaces are sufficiently specified in order to develop the testbench in the first place. Moreover, these responses can be provided on one interface at a time in any preferred order.

4. A virtual DUT to test pre-DUT testbench

This paper presents the methodology of virtual verification which introduces the concept of a virtual DUT – one that responds to the stimulus and provides necessary data – to enable testbench simulation. Consequently, pre-DUT testbench can be developed, tested and debugged. A virtual DUT will help to realize the two opportunities to reduce the testbench development time; mentioned above, the testbench readiness gap reduction and the coverage readiness gap reduction. Figure 3 shows improved RTL, testbench and coverage development timelines.

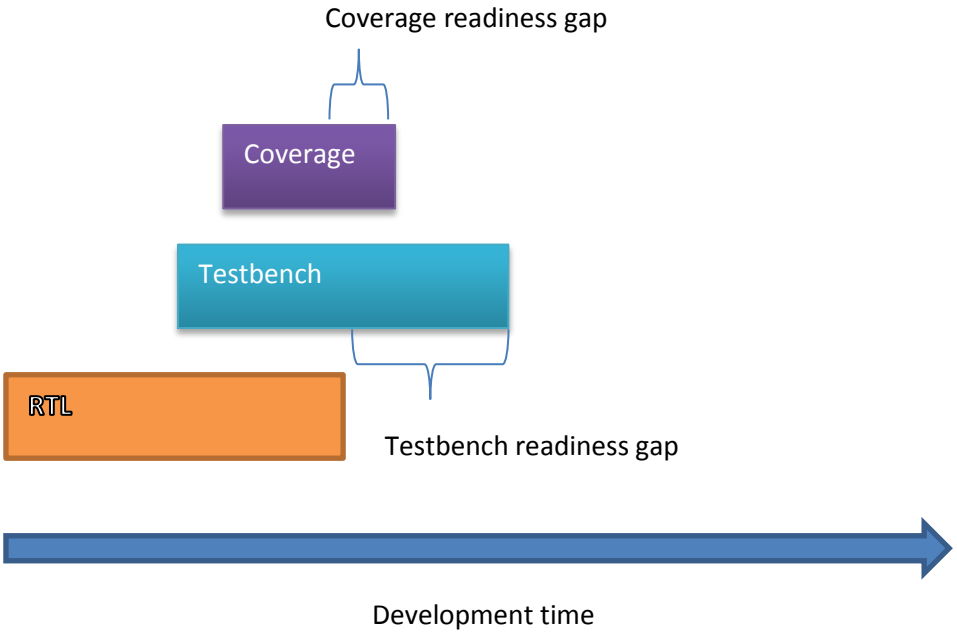


Figure 3 Improved RTL, Testbench & Coverage Development Time

Below are specific testbench items that benefit from this virtual verification methodology:

1. Stimulus scenarios will be generated, debugged and checked via stimulus coverage
2. Coverage for stimuli, states and checkers will be exercised and debugged
3. Input, output and state monitors will be exercised and debugged
4. Drivers will be exercised and debugged
5. Checkers (white-box and black-box) will be exercised, debugged and checked via checker coverage
6. Assertions (cover and property) will be exercised and debugged

Therefore, the more testbench items coded pre-DUT, the more will be tested/debugged and the more ready, and mature they will be post-DUT.

Figure 4 is an improved testbench development cycle with the addition of a blue curve representing the various DUT stages when using a virtual DUT to enable testbench simulation and debugging.

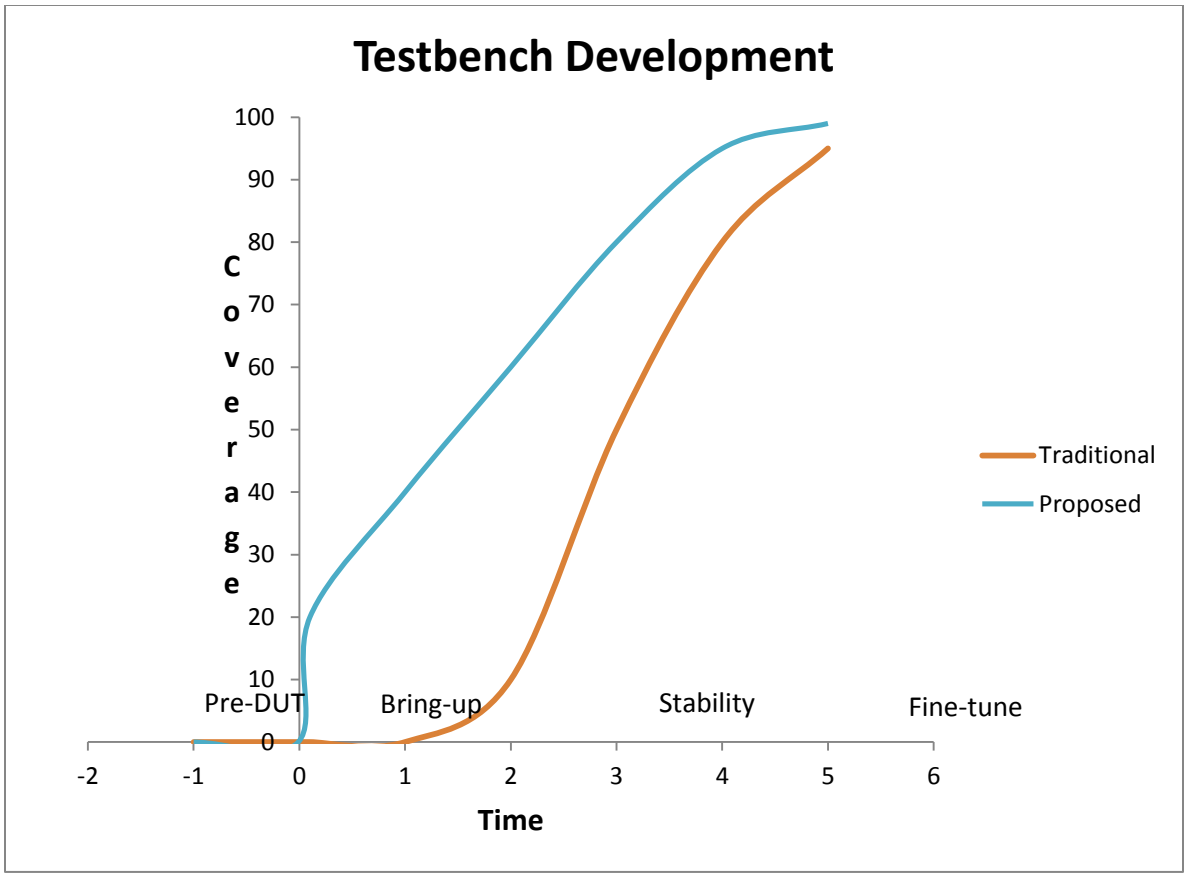
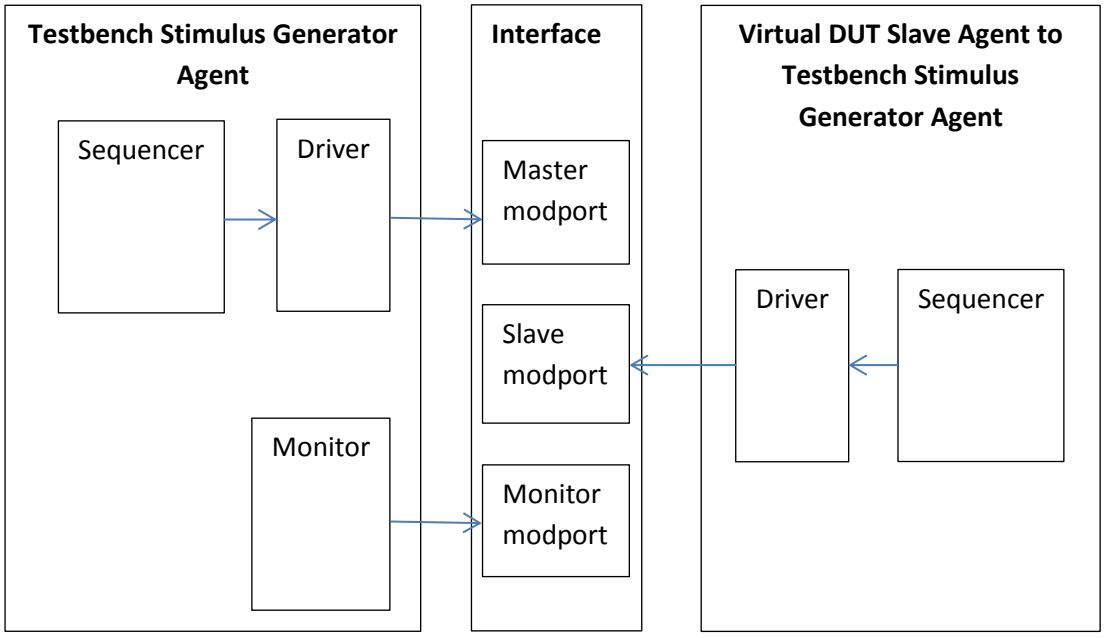


Figure 4 Improved Testbench Development Cycle

Here is how to implement a virtual DUT:

Create a slave agent (e.g. responder) for each interface to respond to the stimulus including a driver, a sequencer, sequences, transactions. Figure 4 below shows al virtual DUT slave agent for a testbench's stimulus-generator agent. Much of the stimulus/master driver can be reused to create the slave/response driver and most of VIPs (i.e. AXI) come with both master and slave agents. Consequently, a virtual DUT enables and facilitates testbench simulation and testbench development well before the DUT is ready.

Here the provided responses are generated in terms of transactions with focus on completeness rather than intelligence (e.g. independent of the stimulus). Specifically, responders provide randomized transactions of all possible values and scenarios intended to test, debug and develop the testbench including stimulus, drivers, monitors, checkers, assertions as well as coverage. Checkers and property assertions will fail but failures will not be used to terminate the test but for negative testing and coverage of these checkers and assertions.



Once the DUT is ready, there are inevitably some adjustments needed to make the testbench works with the real DUT. These adjustments are for signal names (e.g. internal DUT state signals), interface protocols (e.g. temporal relationships), etc. The better the interfaces are specified, the fewer adjustments are needed. The time spent for these adjustments are expected to be much less than the time traditionally spent to debug the testbench.

5. Example

An UVM testbench to verify an 8-master, fixed priority arbiter is shown. A virtual DUT is coded and replaced the DUT to demonstrate the methodology. The testbench simulates including stimulus generation, monitors, checkers, cover and property assertions and coverage. There are additional classes created as shown below:

1. A slave agent, to respond to master requests from the testbench's stimulus generator, a slave transaction, a slave sequence and slave driver.
2. A rsp_slave_agent, to respond to testbench's response generator (e.g. responder), a rsp_slave_transaction, a rsp_slave_sequence and a rsp_slave_driver
3. Define the macro `VIRTUAL_DUT` to exclude the DUT instantiation
4. A property `has_virtual_DUT` to defer the virtual DUT instantiation to the testbench elaboration time and to bypass test failures
5. Exclusion of scoreboard and property assertions errors upon `has_virtual_DUT` set to allow simulation to continue

```
class arbiter_slave_transaction extends uvm_sequence_item;
```

```
   arbiter_package::master_t  m_master;   //for coverage
   bit                  m_done[`NUM_MASTER-1:0];
   rand bit [15:0]      m_rddata[`NUM_MASTER-1:0];
   bit                  m_sel[`NUM_MASTER-1:0];
   rand bit[3:0]        m_sel2done_cycles;
   rand bit[3:0]        m_done_cycles;
   bit                  m_abrupt_reset=0;

   constraint sel2done_cycles_cons
     {m_sel2done_cycles dist {0 := 1, [1:5] := 1, [6:15] := 1};}
   constraint done_cycles_cons
     {m_done_cycles dist {[1:5] := 1, [6:15] := 1};}

   function new(string name="arbiter_slave_transaction");
     super.new(name);
     for (int i=0; i<`NUM_MASTER; i++) begin
       m_done[i] = 1'b0;
       m_sel[i] = 1'b0;
     end
   endfunction : new
endclass: arbiter_slave_transaction

class arbiter_slave_sequence extends uvm_sequence
#(arbiter_slave_transaction);
  uvm_object_utils(arbiter_slave_sequence)
   …
  virtual task body();
  forever
    begin
      `uvm_do(req)
    end
  endtask: body
endclass: arbiter_slave_sequence

class arbiter_slave_driver extends uvm_driver
#(arbiter_slave_transaction);
  …
  task get_transaction(ref arbiter_slave_transaction cmd);
    seq_item_port.get_next_item(cmd);
    for(int i=0 ; i<`NUM_MASTER;i++)
      fork
        automatic int j = i;
        begin
          job[j] = process::self();
          wait (vif.slave_cb.req[j] |
               (vif.slave_cb.pre_req[j] & cmd.m_master == MDIO))
```

```
        cmd.print();
      end
    join_none
endtask: get_transaction

task assert_control_signals(arbiter_slave_transaction cmd);
  @(vif.slave_cb);
  for (int i=0; i< `NUM_MASTER; i++)
     vif.slave_cb.sel[i]<= cmd.m_sel[i];

  if (cmd.m_master == MDIO) wait(vif.slave_cb.req[0]);
  repeat(cmd.m_sel2done_cycles) @(vif.slave_cb);
  @(vif.slave_cb);

  for (int i=0; i< `NUM_MASTER; i++)
    vif.slave_cb.done[i]             <= cmd.m_done[i];
  for (int i=0; i< `NUM_MASTER; i++)
    vif.slave_cb.rddata[i]           <= cmd.m_rddata[i];

  repeat(cmd.m_done_cycles) @(vif.slave_cb);

  for (int i=0; i< `NUM_MASTER; i++)
    vif.slave_cb.done[i]             <= 1'b0;
  @(vif.slave_cb);
endtask

function void done_transaction(arbiter_slave_transaction cmd);
  seq_item_port.item_done();
  end_tr(cmd);
endfunction: done_transaction

task exec_transaction (ref arbiter_slave_transaction cmd);
  assert_control_signals(cmd);
  reset_signals();
  @(vif.slave_cb);
endtask : exec_transaction

task run_phase(uvm_phase phase);
  arbiter_slave_transaction cmd;

  reset_signals();
  monitor_clock_n_reset();

  forever begin
      get_transaction(cmd);
      fork
        exec_transaction(cmd);
```

```
          monitor_random_reset();
        join_any
        disable fork;
        done_transaction(cmd);
      end
  endtask: run_phase
endclass : arbiter_slave_driver
```

6. Conclusion

A virtual verification methodology is presented to boost pre-DUT testbench development via the usage of a virtual DUT which allows developing testbench to be simulated, tested and debugged. The more coded testbench pre-DUT, the more ready and tested the testbench is post-DUT. Testbench development time is reduced and consequently verification closure time is shortened.