Appendix B

Using UVM for Functional Verification Closure of NON-UVM Testbenches

Earlier chapters discussed various aspects of UVM with complete environments. When creating environments from scratch or developing new functionality, verifying with UVM is a good approach. In the real world, however, many non-UVM testbenches for silicon-proven RTL are prevalent. It is usually not justified to rewrite a UVM testbench for such designs unless there are significant bugs can be found by this effort. The question, therefore is whether would be possible to have UVM co-exist with the existing environment and ***use UVM for only the portions that need some additional verification without extensive changes to the current environment.*** If this were the case, all other tests that test the functionality available from the non-UVM approaches may be leveraged. This chapter attempts to illustrate a method to perform an assessment of the existing non-UVM environment and add missing coverage measurements using a UVM environment. For the discussion below, the non-UVM environment is referred to as a ***legacy*** environment.

This chapter covers the following areas:

- Illustrate one of many ways to integrate UVM into a non-UVM environment
- Explore functional coverage with UVM.
- Identify the phasing synchronization points between the legacy and UVM environment.
- Identify goals for the UVM environment.

Why would an UVM testbench even be considered? The existing environment would have defintely yielded a few bugs, and this environment could be easily extended with additional tests or covergroups without migrating to UVM. The ***key is the amount of effort involved.*** The tools and other related verification IP's enable quick development in UVM, which supports randomization, coverage, and other features in addition to the reasons described in Chapter 1.

With very little effort, one may be able to leverage many features in verification IP's and use some of the most advanced debug features available in modern tools that support UVM. Leveraging modern tools and techniques could save verification engineers a great deal of time. This common debug problem is faced worldwide, and this makes the discussion in this chapter worthwhile. You will find that reusing a legacy testbench with UVM is not difficult once you read this chapter. You can save a significant amount of effort as functional areas *that are not addressed by the existing tests can be identified,* and the verification activity can focus on areas where more testing is needed.

The description below is a recipe to interface legacy testbenches to UVM. The principles of this method can be tweaked based on the particular implementation under assessment and based on the specifics of the testbench.

Note that this chapter does not cover interoperability between UVM and other methodologies like OVM/VMM/eRM because each of those methodologies have their own messaging and phasing rules that would need more extensive treatment than is provided in this book. The approach presented is ***only one of many possible approaches to accomplish this goal.***

B.1 Legacy Environment Description

The Wishbone DMA module is designed to provide DMA transfers between two different wishbone interfaces. The DMA/Bridge core has two master and slave wishbone interfaces. It contains a 31 channel DMA engine that can be programmed with a priority arbiter to transfer data between the interfaces. It is capable of performing DMA using a linked list of descriptors. There are some registers in the core that provide both control/status and configuration. An interrupt mechanism is present in the core, and allows an interface with the CPU in the system if desired. See the provided documentation in the download for a detailed description of the core in the ”docs” directory.

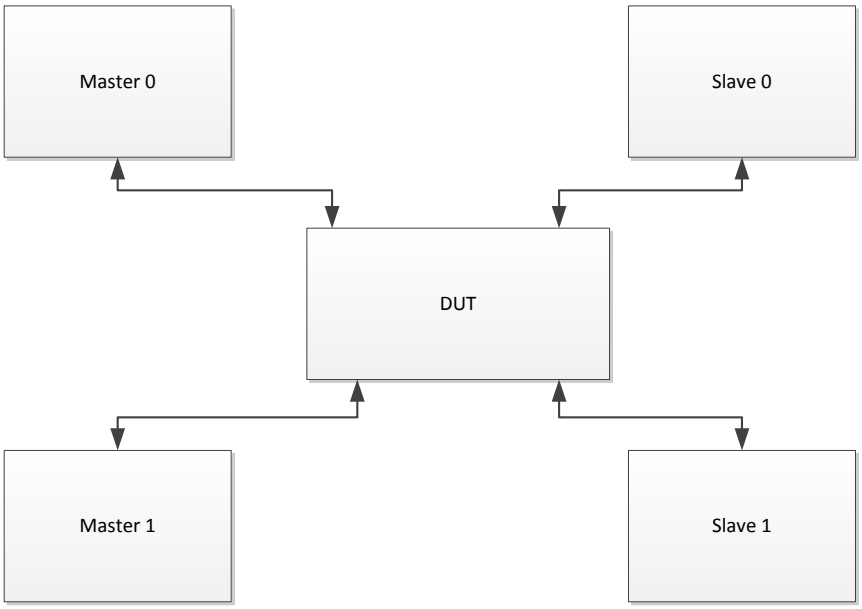

Fig. B.1: Wb_dma Legacy Testbench Block Diagram

A simplified block diagram of the testbench is shown in Figure B.1. The WB_DMA RTL is wrapped in a simple Verilog testbench. This testbench instantiates a couple of master BFM modules written in Verilog. These modules have interface pins. These modules supply the pin level wiggles and provide a few tasks to perform simple wishbone operations. The slave modules contain a memory and perform simple BFM functions.

Typical tests in this testbench are tasks. One of the tasks is displayed in Listing B.1. These tasks typically program the DUT to perform a particular operation after setting up the corresponding source and destination information for the test. Listing B.1 lists one of the tests supplied with the DMA core from its legacy environment as an example of a legacy test. ***We have NOT provided a detailed explanation for this listing. You must study the core to understand this specific listing. The study of the core is not required for you to understand the rest of the content however as we are only discussing concepts which are not specific to this core.***

Listing B.1: Example Task test for WB_DMA

```
task pt10_wr;
// Misc Variables
reg [31:0]  d0,d1,d2,d3;
integer   d,n;
begin
$display("\n");
$display("*****************************************************");
$display("*** Running Path Through 1->0 Write Test ....     ***");
$display("*****************************************************\n");

s0.fill_mem(1);
s1.fill_mem(1);
d=1;
n=0;
for(d=0;d<16;d=d+1)
 begin
   $display("INFO: PT10 WR4, delay %0d",d);
   for(n=0;n<512;n=n+4)
     begin
  d0 = $random;
  d1 = $random;
  d2 = $random;
  d3 = $random;
  m0.wb_wr4(n<<2,4'hf,d,d0,d1,d2,d3);
  @(posedge clk);
  if( (s1.mem[n+0] !== d0) | (s1.mem[n+1] !== d1) |
    (s1.mem[n+2] !== d2) | (s1.mem[n+3] !== d3) )
     begin
    $display("ERROR: Memory Write Data (%0d) Mismatch: (%0t)",n,$time);
    $display("D0: Expected: %x, Got %x", s1.mem[n+0], d0);
    $display("D1: Expected: %x, Got %x", s1.mem[n+1], d1);
    $display("D2: Expected: %x, Got %x", s1.mem[n+2], d2);
    $display("D3: Expected: %x, Got %x", s1.mem[n+3], d3);
    error_cnt = error_cnt + 1;
     end
      end
 end
$display("\nINFO: PT10 WR1");
   for(n=0;n<512;n=n+1)
     begin
  d0 = $random;
  m0.wb_wr1(n<<2,4'hf,d0);
  @(posedge clk);

  if( s1.mem[n+0] !== d0 )
     begin
    $display("ERROR: Memory Write Data (%0d) Mismatch: (%0t)",n,$time);
    $display("D0: Expected: %x, Got %x", s1.mem[n+0], d0);
    error_cnt = error_cnt + 1;
     end
      end
show_errors;
```



```
53 $display("*************************************************");
54 $display("*** Test DONE ...                              ***");
55 $display("*************************************************\n");
56 end
57 endtask
```

B.2 Adding UVM Environment to Legacy Testbench

Upon completion of legacy code review, you may come to the conclusion randomization and coverage approaches may provide more extensive coverage than the results obtained through legacy approaches. You must identify the additional tests that must be developed by an assessment. This assessment may use or a verification plan leveraged from earlier efforts. A new one may also be created and populated after an assessment of the legacy tests. The assessment itself can be performed either using:

- Automated assessments
- Manual assessments

Manual assessments are usually time-consuming and are not the focus of this chapter. To perform automated assessments, the UVM testbench is added in parallel to the existing testbench. It provides monitoring of the activity in the testbench in this configuration. This monitoring is used to collect information about the tests, so that one can then identify additional tests that must be written. A UVM testbench, with automatically generated covergroups, can quickly extract information from the test environment. You may choose to add specific covergroups, in addition, to the automatically generated ones to observe specific conditions. To connect a UVM testbench running in parallel with the existing testbench, you must accomplish the following steps in UVM:

- The UVM testbench is a monitor in the initial stages. The testbench may be reconfigured later to become an active agent with sequencer, driver, and monitor.
- The phasing between the UVM testbench and the legacy testbench must be identical.
- The UVM testbench must finish the test reporting and other necessary activity before the tests terminate.

Often in legacy testbenches, the tests typically are comprised of tasks that are called in a sequence. These tasks comprise the bulk of the logic used to create the test. Sometimes, at completion of the test, the testbench may call other tasks/tests or terminate the testbench using a $finish call. A simple handshake between the legacy testbench and the UVM testbench is required to inform the UVM testbench that the Verilog tests have completed their activity before terminating the simulation.

B.2.1 Changes to the Top Level Testbench

The UVM testbench uses SystemVerilog interfaces to allow the UVM class hierarchy to interface with the DUT. The legacy environment has a collection of wires that is wired between the modules. To connect this to the UVM environment, we need to connect the I/O from the core into a SystemVerilog interface, so that the core can interface with the UVM agents used to perform the assessment. Two master interfaces and two slave interfaces are instantiated into the top-level module for this purpose. These interfaces are wired to the existing ports in the master and slave interfaces of the DUT. The signals of the legacy master BFM are connected to the interfaces of the master interfaces, and the corresponding wiring occurs on the slave interfaces as well. An additional bit (used as a semaphore) is added to help with the phasing and is described later on in this chapter. The UVM package is imported into the testbench.

The block diagram of the interconnections in the testbench along with the UVM components is shown in Figure B.2. The UVM components are shown to be connected in parallel to the existing master and slave bus functional models that already exist in the legacy testbench.

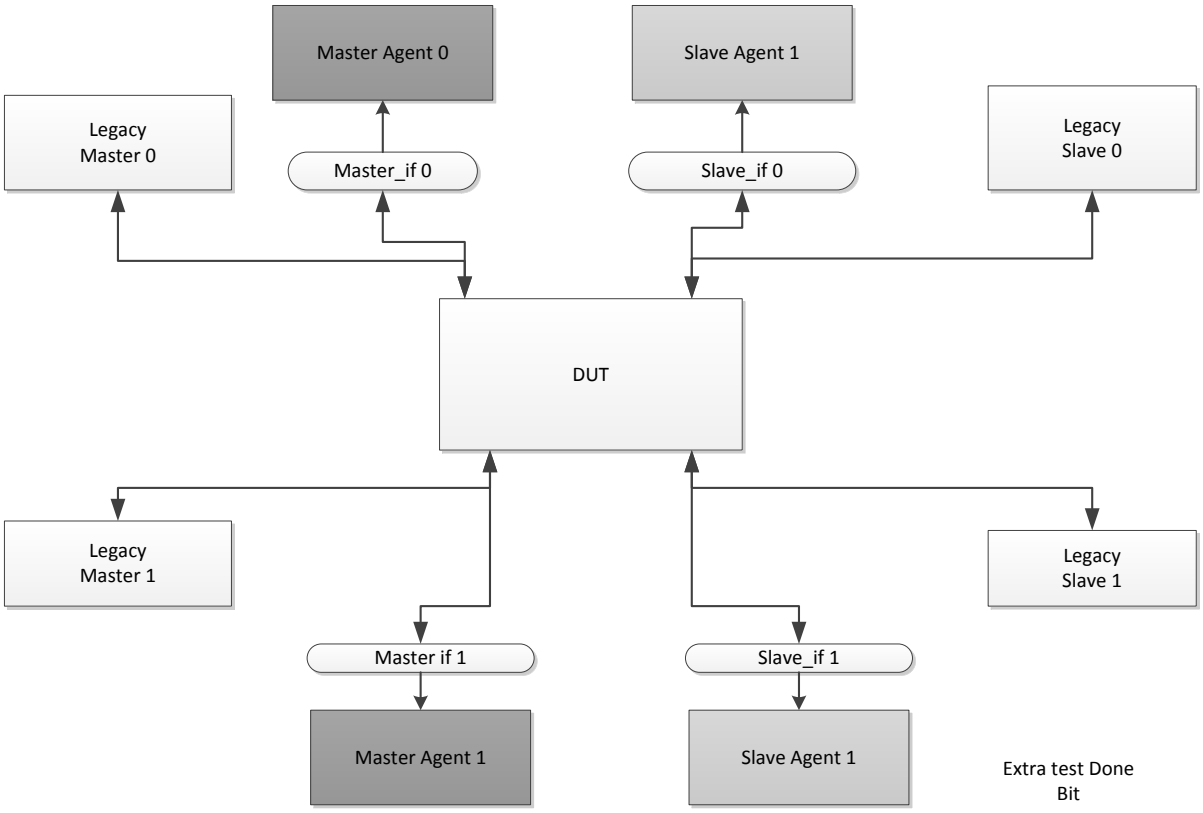

Fig. B.2: Block Diagram Of The Wb_dma With The UVM And Legacy Testbenches Included

These steps integrating the code are shown in the listing below.

Listing B.2: Unified Testbench

```
489    wb_master_if wb_mstr_if0(clk,rst);
490    wb_master_if wb_mstr_if1(clk,rst);
491    wb_slave_if wb_slv_if0(clk,rst);
492    wb_slave_if wb_slv_if1(clk,rst);
493    //assignments to wires in the interface
494
495    assign wb_mstr_if1.DAT_I =  wb1m_data_o;
496    assign wb_mstr_if1.DAT_O =  wb1m_data_i;
497    assign wb_mstr_if1.TGD_I =  15'b0;
498    assign wb_mstr_if1.TGD_O =  15'b0;
499    assign wb_mstr_if1.ACK_I =  wb1_ack_o;
500    assign wb_mstr_if1. ADR_O =  wb1_addr_i;
501    assign wb_mstr_if1.CYC_O =  wb1_cyc_i;
502    assign wb_mstr_if1.ERR_I =  wb1_err_o;
503    assign wb_mstr_if1.LOCK_O =  1'b0;
504    assign wb_mstr_if1.RTY_I  =  wb1_sel_o;
```



```
505    assign wb_mstr_if1.SEL_O =  wb1_sel_o;
506    assign wb_mstr_if1.STB_O =  wb1_stb_i;
507    assign wb_mstr_if1.TGA_O =  16'b0;
508    assign wb_mstr_if1.WE_O =  wb1_we_i;
509
510    assign wb_mstr_if0.DAT_I =  wb0m_data_o;
511    assign wb_mstr_if0.DAT_O =  wb0m_data_i;
512    assign wb_mstr_if0.TGD_I =  05'b0;
513    assign wb_mstr_if0.TGD_O =  05'b0;
514    assign wb_mstr_if0.ACK_I =  wb0_ack_o;
515    assign wb_mstr_if0. ADR_O =  wb0_addr_i;
516    assign wb_mstr_if0.CYC_O =  wb0_cyc_i;
517    assign wb_mstr_if0.ERR_I =  wb0_err_o;
518    assign wb_mstr_if0.LOCK_O =  0'b0;
519    assign wb_mstr_if0.RTY_I  =  wb0_sel_o;
520    assign wb_mstr_if0.SEL_O =  wb0_sel_o;
```

It must be noted that none of the original testbench connections was modified to remove the existing bus functional models at this stage. The connections are implemented using the code shown in Listing B.2. You could, in theory, contain all this code inside an include file to help you manage the changes to the testbench.

In Listing B.2:

- Lines 489 – 490 instantiate the master interfaces.
- Lines 491 – 492 instantiate the slave interfaces.
- Lines 495 – 557 (only up to Line 520 shown) connect the master and slave interfaces to the respective wires in the existing testbench. Note that the connections are made in a way to reflect the sampling of the signals in monitor mode only. For reasons of brevity, only some of the connections are shown here. See the source code for the rest of the listing.

B.2.2 Connecting UVM Testbench to the Top Level

The UVM testbench is connected to the interfaces instantiated in the top-level testbench by setting the handles to the interfaces in the config_db. The test is started from the top-level run_test() task. Listing B.3, shows how to accomplish this easily.

Listing B.3: Top Level Test Bench

```
556    initial begin
557       uvm_active_passive_enum is_active = UVM_PASSIVE;
558       uvm_config_db #(v_if2)::set(null,"uvm_test_top.env.slave_agent0","
             slv_if",test.wb_slv_if0);
559       uvm_config_db #(v_if2)::set(null,"uvm_test_top.env.slave_agent1","
             slv_if",test.wb_slv_if1);
560       uvm_config_db #(v_if1)::set(null,"uvm_test_top.env.master_agent0","
             mst_if",test.wb_mstr_if0);
561       uvm_config_db #(v_if1)::set(null,"uvm_test_top.env.master_agent1","
             mst_if",test.wb_mstr_if1);
562       uvm_reg::include_coverage("*",UVM_CVR_ALL) ;
563       run_test();
564    end
```

- Lines 556 – 563 is a **new** initial block added to the testbench. This block runs in parallel with the other initial blocks in the system.

- Lines 557 – 560 set the config_db settings for the master and slave interfaces into the respective agents.
- Line 562 turns on the register coverage. This is explained later in Section B.3.1
- Line 563 kicks off the UVM portion of the testbench in parallel with the rest of the environment.

Note that other than the parallel connections in the environment, the call of the run_test() task, and config_db settings, very few changes are required. The uvm_pkg from the Accellera distribution is imported into the top-level testbench.

B.3 Coverage Model

The coverage model for the UVM portion of the testbench is comprised of automated and manually generated covergroups. As in many RTL designs, the modes of the design are programmable using registers in this WB_DMA design. Hence, the coverage for various registers will give us a good insight into the testing completed on the various DUT modes. The discussion can, therefore, be broken up into the following main categories:

- An automatically generated register model from RALGEN
- A hand-written coverage model.

B.3.1 Register Layer Based Coverage Model

The use of a register abstraction layer allows us to generate some covergroups can be easily generated by simply turning on the required coverage switches in the register model generator. In this case, we chose the RALGEN© tool from Synopsys. This tool supports the creation of automated covergroups from the register models, and this can save you some time. Alternatively, you can supplement the generated register models and add the required covergroups of interest. Choose an approach to have the tools generate the models to simplify maintenance.

Depending on a specific scenario, the coverage models could be generated differently. Consult the simulator vendors documentation to ensure that the models are generated correctly.

In this example, the register model for the WB_DMA block is available in abstract form. For purposes of this discussion, only the CHN_CSR register is shown as an illustration for the various covergroups that are automatically generated using RALGEN©from Synopsys. Listing B.4 shows these groups. Consult the source code for all the covergroups used in the simulation.

Listing B.4: RAL Autogenerated model Using RALGEN with Autogenerated Covergroups

```
5
6 class ral_reg_CSR extends uvm_reg;
7    rand uvm_reg_field PAUSE;
8    local uvm_reg_data_t m_data;
9    local uvm_reg_data_t m_be;
10   local bit            m_is_read;
11
12   covergroup cg_bits ();
13      option.per_instance = 1;
14      option.name = get_name();
15      PAUSE: coverpoint {m_data[0:0], m_is_read} iff(m_be) {
16   wildcard bins bit_0_wr_as_0 = {2'b00};
17   wildcard bins bit_0_wr_as_1 = {2'b10};
18   wildcard bins bit_0_rd_as_0 = {2'b01};
19   wildcard bins bit_0_rd_as_1 = {2'b11};
20   option.weight = 4;
21      }
22   endgroup
23   function new(string name = "CSR");
24      super.new(name, 32,build_coverage(UVM_CVR_REG_BITS));
25      if (has_coverage(UVM_CVR_REG_BITS))
26  cg_bits = new();
27   endfunction: new
28   virtual    function void build();
29      this.PAUSE = uvm_reg_field::type_id::create("PAUSE",,get_full_name()
           );
30      this.PAUSE.configure(this, 1, 0, "RW", 0, 1'h0, 1, 0, 1);
31   endfunction: build
32
33   `uvm_object_utils(ral_reg_CSR)
34
35
36   virtual    function void sample(uvm_reg_data_t data,
37               uvm_reg_data_t byte_en,
38               bit            is_read,
39               uvm_reg_map    map);
40      if (get_coverage(UVM_CVR_REG_BITS)) begin
41   m_data    = data;
42   m_be      = byte_en;
43   m_is_read = is_read;
44   cg_bits.sample();
45      end
46   endfunction
47 endclass : ral_reg_CSR
```

The entire register code is present in the provided source listing. As this discussion focusses on coverage, Only a few lines of autogenerated code are relevant to the discussion.

- Lines 12 – 22 describe the covergroup cg_bits. The PAUSE coverpoint makes a distinction between writing and reading values to the bits of the PAUSE field.
- Lines 23 – 27 are the most important and interesting lines. They form the constructor for this register, and pass the build_coverage parameter to the superclass. Note that Lines 25 – 26 check for the existence of the

UVM_CVR_REG_BITS options before building the covergroup. On account of this check, the values must be in resource_db before the construction of the environment. *Note that this is a common problem when using the register coverage.*

- Lines 36 – 46 show the automatically created sample() function called when the coverage options are turned on.

As you can see from the covergroups in the listing, if a WRITE or a READ happens to any bit in the fields of the register, the model captures that operation as a part of the coverage model. Since in many designs, the specific register can easily mean a mode; You can see that you have *achieved free mode coverage* from this autogenerated model without having to do much work at all.

B.3.2 The Functional Coverage Model

In addition to the generated register coverage model, you can add your own covergroups to ensure that additional coverage points are measured. In this example, we chose to simply cover the transactions on the master[0] interface. One could easily change this covergroup to cover more interesting scenarios, but this is sufficient to illustrate how coverage is collected in UVM.

Listing B.5: Functional Coverage Example

```
 9 class wb_dma_env_cov extends uvm_component;
10    event cov_event;
11    wb_transaction tr;
12    uvm_analysis_imp #(wb_transaction, wb_dma_env_cov) cov_export;
13    `uvm_component_utils(wb_dma_env_cov)
14
15    covergroup cg_trans;  ;
16       coverpoint tr.kind;
17       coverpoint tr.address {
18          bins lo =   {[0:100]};
19          bins med =   {[101:200]};
20          bins hi = {[300:$]};
21       }
22       tga: coverpoint tr.tga {
23          bins sane = {[0:255]};
24          ignore_bins hi = {[256:$]};
25       }
26       // tgc: coverpoint tr.tgc ;
27       // ToDo: Add required coverpoints, coverbins
28    endgroup: cg_trans
29
30
31      function new(string name, uvm_component parent);
32         super.new(name,parent);
33         cg_trans = new;
34         cov_export = new("Coverage Analysis",this);
35      endfunction: new
36
37    virtual function write(wb_transaction tr);
38       this.tr = tr;
39       cg_trans.sample();
40    endfunction: write
41
42 endclass: wb_dma_env_cov
```


In Listing B.5, component class wb_dma_env_cov has a covergroup inside it. It has a uvm_analysis_imp port in it. This analysis implementation provided a write() function that is called by the monitor's analysis port.

This component class contains the cg_trans covergroup that defines the coverage for the transaction kind, the transaction address, and the transaction TGA tags. The address has been divided into three bins: lo, medium, and high. The TGA pins are sampled in this covergroup. You would certainly want to take samples of other coverage points in the environment to suit the verification task at hand.

- Lines 15 – 28 define the covergroup. The covergroup consists of three main coverpoints and is triggered on the cov_event event.
- Line 16 defines the cover point for the transaction kind.
- Lines 17 – 20 define the cover point on the transaction address. This cover point is divided into three bins: lo, med,high based on the address.
- Lines 22 – 25 define the cover point on the transaction TGA tags.
- lines 31 – 35 define the constructor for the class. Note that the covergroup must be created before it is used, and hence the constructor for the covergroup has been placed inside the constructor for the class.
- Lines 37 – 40 describe the write implementation for the uvm_analysis_imp port. When an analysis port calls the write() function, the cov_event is triggered, causing the coverage to be sampled.

The covergroups are created inside a UVM component to sample the information at the appropriate times. The monitors convert the information from the pins that are being monitored and convert them into abstract transactions. These are either used to sample the manually written covergroups or fed to a register predictor to be converted to register operations that may be sampled. See Section B.5 to understand how the above component is connected to the monitor in the connect phase.

B.4 Taking Care of Phasing

Phasing is one of the most crucial aspects of this dual environment flow. Although it appears challenging, it is not the case. In most testbenches the tests form a standard pattern as shown in the figure:

Fig. B.3: Typical Verilog Testbench Phasing

The simulator builds the various modules and then begins to call all the initial blocks one by one before advancing time and applying reset to the design. The testbench then calls many tasks. An example of such a task is provided in the earlier Section B.1. This task is the most active part of the test. Examining the code, one can easily identify the various phases inside Listing B.1.

- Lines 6 – 28 in that listing set up the test before the master bus functional model performs a write.

- Lines 45 – 56 show the summarization of errors and other test cleanup.

Compared to a Verilog testbench, UVM has many well-defined phases we need to adhere to. These phases need to be executed at specific points in your simulation. This is illustrated in Figure B.4. We must somehow line up the two testbenches so that they work in lock step with one another.

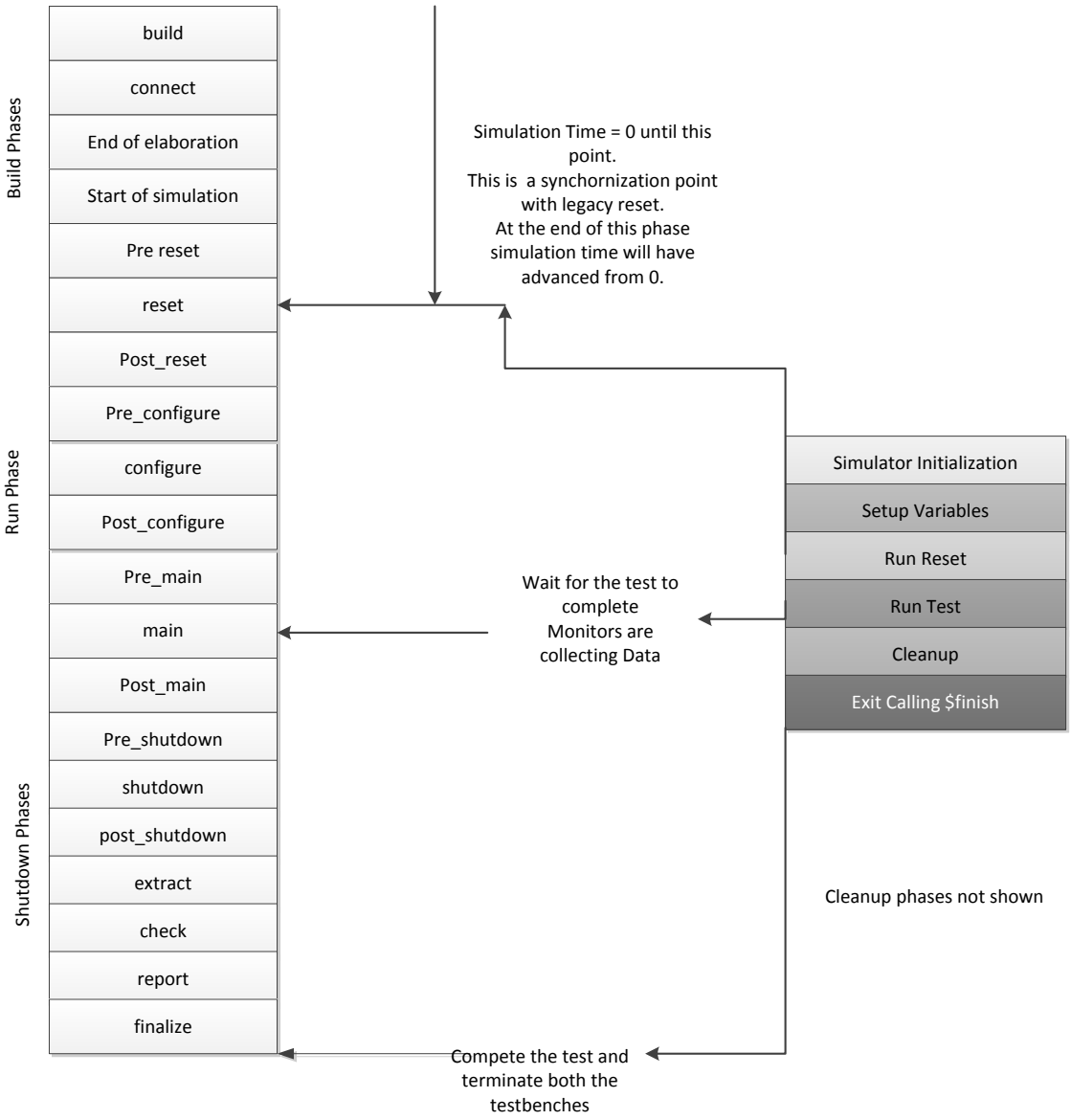

Fig. B.4: Phasing Of Both The Testbenches

The first main task in the UVM hierarchy where the time starts advancing is the reset_phase(). In this phase, the DUT is reset. In a legacy environment, the simulator builds all the components at the beginning of the simulation and then starts the tasks in the initial blocks. However, in the case of UVM, a complete class-based environment has to be built BEFORE the reset task is called. This environment is built using the phasing constructs in the uvm_component classes. Since it is

essential that the two parts (legacy and UVM) of the testbench remain in lock step, the UVM portion of the testbench must be built before the reset phase continues.

In this case, this implies that the simulation must complete the common domain tasks in the UVM phase schedule before the reset portion of the test proceeds. To enable this synchronization, the monitors in the agents must be sensitive to the reset signal and provide objections to the phase from completing until they are detected.

B.4.1 The Reset Agent

This feature of waiting for the reset to complete has been incorporated into the wb_master_mon classes in the UVM testbench. Note that in the case of a commercial protocol, many bus based VIP agents could provide this functionality if configured to do so.

Listing B.6: Synchronizing Reset

```
1   task wb_master_mon::reset_phase(uvm_phase phase);
2      super.reset_phase(phase);
3      @(negedge this.mon_if.rst);
4   endtask: reset_phase
```

Listing B.6 shows the monitor is set up to wait for the negative edge (de-assertion) of reset. As the testbench is simple, the reset is pulsed only once. When the reset goes through *multiple* assertions and de-assertions in the testbench, then care must be taken to ensure that the proper phasing is achieved and you exit this stage correctly. At the end of this reset phase, the reset is de-asserted, and the monitor drops the objection allowing the rest of the test to proceed.

B.4.2 Synchronizing the Phases in Legacy and UVM Testbenches

The testbench has been synchronized to the reset phases, and testbench is in the main phase. The testbench needs to have all the components actively monitoring the signals when the main test is active. In this configuration, the Verilog testbench drives the signals using the existing Verilog bus functional models. The UVM agents capture these transactions using the monitors in the master and slave agents. The agents ,configured as monitors, do not know when to stop monitoring because there is no sequence completion to terminate the UVM portion of the testbench. Once the test is completed from the legacy side, the UVM testbench needs to be told to move past the run phase. To do so, a handshake is required between the legacy testbench and the Verilog testbench. This is accomplished by adding a ‘test_done‘ bit to the top-level testbench that is initialized to 0.

Listing B.7: Top Level Test Bench Synchronization

```
296  begin
297     #100;
298     if($test$plusargs("pt10_rd"))
299       pt10_rd;
300     if($test$plusargs("pt10_wr") )
301       pt10_wr;
302
303     if($test$plusargs("pt01_rd") )
304       pt01_rd;
305     if($test$plusargs("pt10_rd") )
306       pt01_wr;
307
308     if($test$plusargs("sw_dma13") )
309       sw_dma1(3);
310     if($test$plusargs("hw_dma13") )
311       hw_dma1(1);
312
313       repeat(1000) @(posedge clk);
```

```
314       test_done = 1;
315         repeat(1500) @(posedge clk);
316         $finish;
317   end
```

Upon completion of the test tasks, the ‘test_done‘ bit is set to 1. The test in the main phase raises an objection to the UVM test termination at the start of the run phase. It then waits for the test_done handshake to complete on the legacy side and then drops the objection. In this manner, the two remain synchronized. This ‘test_done‘ bit can be added and set to 1 at a point in the test right after the test task if the test task completes [1].

Listing B.8 shows the signal from the top-level test being used to synchronize the two testbenches. You could have used the UVM phase wait_for_state() method as an alternative. It is not shown here for reasons of simplicity.

Listing B.8: Synchronizing the Main Phases

```
53    virtual task main_phase(uvm_phase phase) ;
54       if (phase != null)
55         phase.raise_objection( this, get_type_name() );
56       wait (test.test_done==1);
57       phase.drop_objection( this, get_type_name() );
58
59    endtask
```

- Line 54 – 55 raise an objection in the run_phase() of the test class. This prevents the test from completing.
- Line 56 makes the UVM testbench wait for the legacy testbench to complete driving and analyzing stimulus. During this process, the monitor threads in all the agents are active and collecting transactions. These are being sent to the coverage collection mechanisms.
- Line 59 upon completion of the ‘test_done‘ from the legacy side drop the objection that prevents the run_phase from ending.
- The rest of the reporting and cleanup phases complete in zero time and then coverage collection/statistics are reported by the testbench and the UVM testbench terminates the test.

In some cases, the test tasks call $finish in some environments. In such cases, the UVM testbench will not complete collecting statistics. Unless it is a significant amount of trouble, one can make a copy of the tests, and simply replace the $finish with the assignment of ‘test_done‘ bit to 1. IE:
replace $finish; with
test_done =1 ;

You can replace the $finish call with a function that does this more elegantly. There are multiple ways to approach this handshake problem. What is shown here is only one of several ways. Instead of the test_done bit, you could have also used a config_db variable, used an event, or used something else. Use whatever is expedient to your situation. Once the assessment is complete, the bigger question of whether to supplement the core’s existing tests or write a new testbench would have been answered. You could choose to preserve this modified testbench or tests, or do something else.

[1] you may need to devise an intercept point if the test calls a $finish

B.5 UVM Environment

The UVM environment testbench has master and slave agents corresponding to each of the DUT interfaces. For our example, a number of concepts from chapter 21 are employed here. The agents developed in Section 8 will be reused in this section. The master and slave agents are configured using the configuration objects described in Section 21.3. The configuration settings for the min_address and max_address are chosen in order to allow the entire range of the WB_DMA to be accessible to the master and slave agents.

The environment class contains the following components:

- The master and slave agents.
- The register model.
- An explict predictor.
- A coverage collection subscriber.
- An register adapter.
- Connections between the various components.

These will be described in the following paragraphs.

B.5.1 Environment Block Diagram

The top-level UVM environment is a container class that contains two master agents, two slave agents, the register model and the coverage components. An explicit predictor is wired on to the master agent monitors, and the coverage collectors are used to collect coverage.

Figure B.5 shows two master agents and two slave agents as a part of the environment class. These two master agents connect to the two master interfaces of the DMA module. The two slave agents are instantiated in this environment and connect to the two slave interfaces. A register model that describes the register functionality is developed using the DMA specification. The register model is connected to an external predictor and then used to sample coverage.

B.5.2 The Register Model

The register model for the WB_DMA module is created using a register model generator. In this case, I have used the RALGEN tool from Synopsys to create the register model based on the specification of the DMA module provided with the core. Refer to the documentation for the registers provided for the RTL core to get more information about the registers. For now, it is sufficient to assume that the register model is present. Details of how to create a register model are present in Section 12 and other chapters.

B.5.3 The External Predictor Model

Section 20.1.3.3 provides a detailed discussion on the various kinds of register prediction mechanisms. The legacy environment drives transactions straight into the DUT, and the master agents are initially configured to act as monitors. A simple external predictor model is built into the environment to capture all transactions from the legacy environment.

B.5.4 The Coverage Models

Section B.3 describes the model in some detail. These models are built in the build phase.

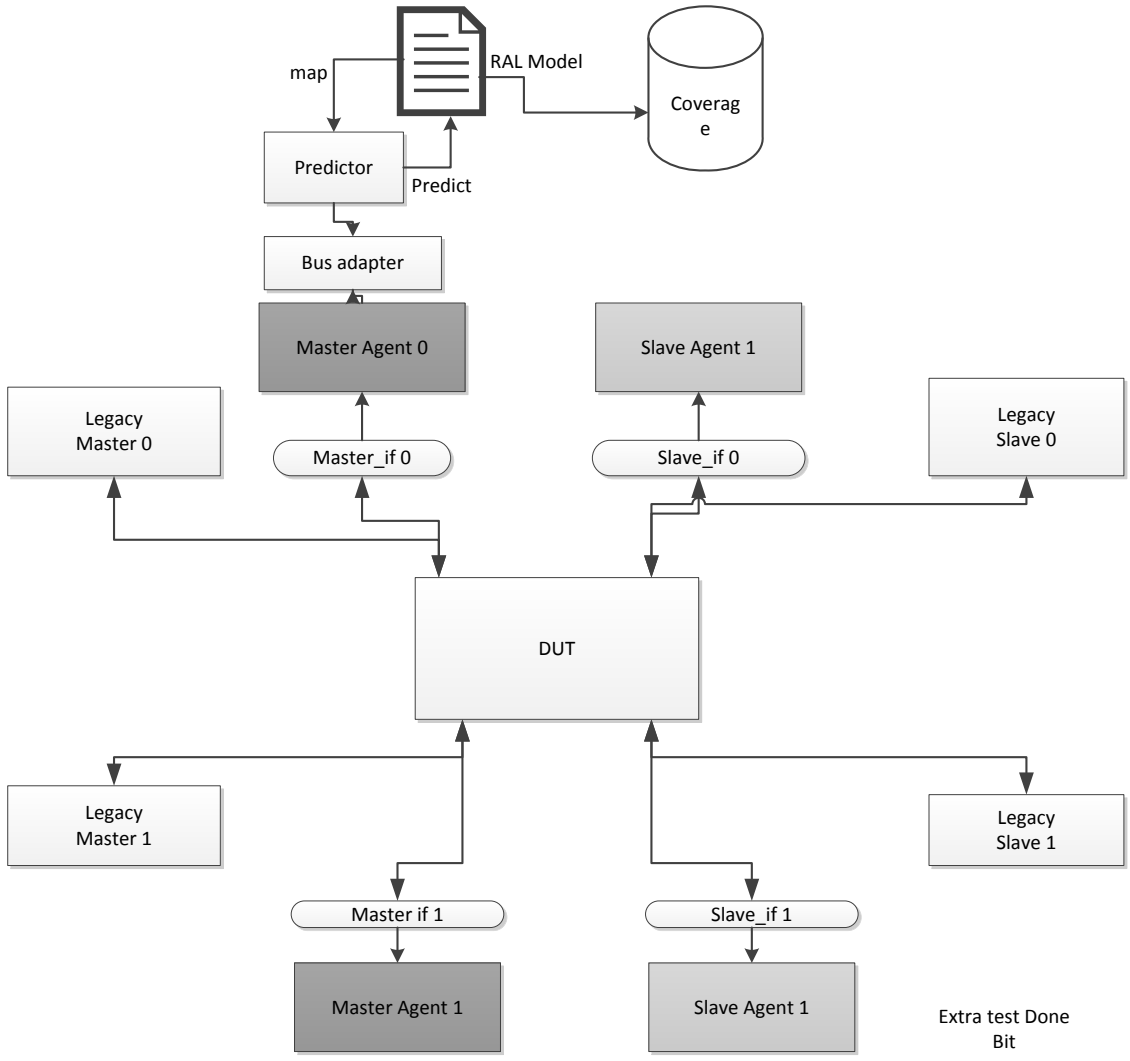

Fig. B.5: Block Diagram Of The Wb Dma Environment

B.5.5 The Build Phase

The build phase creates all the components in the UVM testbench which is shown below.

Listing B.9: Top Level UVM Environmnent Build Phase

```
36 function void wb_dma_env::build_phase(uvm_phase phase);
37    super.build_phase(phase);
38    uvm_reg::include_coverage("*",UVM_CVR_ALL);
39
40    master_agent0 = wb_master_agent::type_id::create("master_agent0",this);
41    master_agent1 = wb_master_agent::type_id::create("master_agent1",this);
42    slave_agent0 = wb_slave_agent::type_id::create("slave_agent0",this);
43    slave_agent1 = wb_slave_agent::type_id::create("slave_agent1",this);
44
```



```
45    cov = wb_dma_env_cov::type_id::create("cov",this); //Instantiating the
          coverage class
46
47    mon2cov  = wb_master_mon_2cov_connect::type_id::create("mon2cov", this)
          ;
48    mon2cov.cov = cov;
49
50    this.regmodel = ral_block_wb_dma::type_id::create("regmodel",this);
51    void'(regmodel.set_coverage(UVM_CVR_ALL));
52    regmodel.build();
53    regmodel.default_map.set_auto_predict(0);
54
55    regmodel.lock_model();
56    ral_adapter = new("reg2host");
57    wb_reg_predictor = uvm_reg_predictor #(wb_transaction)::type_id::create
          ("Explicit Reg predictor",this);
58
59 endfunction: build_phase
```

- Line 38 sets the coverage option for the registers as the resource_db setting is made before the register model is built.
- Lines 40 – 43 create the various master and slave agents.
- Line 45 shows the coverage collector class for the functional coverage model being created.
- Line 50 shows the creation of the register model top-level class. This is only the top-level class, and the remaining register hierarchy is not yet built.
- Line 51 sets register coverage options.
- Line 52 builds the register model.
- Line 53 turns off the default predictor inside the register model and makes it use an external predictor.
- Line 55 locks the register model down and prevents any further changes to this model.
- line 56 creates the register adapter that is used to convert the type of transactions from wb_transaction to uvm_reg type for use by the model. See Section 20.1.2 for a description of the operation of the adapter.
- line 57 builds the external register predictor that is parameterized to take on an input of type wb_transaction.

At the end of this phase, the components are fully built, and can be connected.

B.5.6 The Connect Phase

During the connect phase, several interesting things happen. The coverage collector analysis port is connected to the master agent0 monitor. The same monitor is connected to the register predictor. In this manner, coverage for the transactions that pass on the master0 interface is covered. If one would like to cover the other master/slave interfaces, coverage needs to be added to those agents as well.

Listing B.10: Top Level UVM Environmnent Build Phase

```
61 function void wb_dma_env::connect_phase(uvm_phase phase);
62    super.connect_phase(phase);
63    master_agent0.mast_mon.mon_analysis_port.connect(cov.cov_export);
64
65 // Explicit Reg predictor connections
66    wb_reg_predictor.map = regmodel.default_map;
67    wb_reg_predictor.adapter = ral_adapter;
```

```
68     master_agent0.mast_mon.mon_analysis_port.connect(wb_reg_predictor.
           bus_in);
69
70 endfunction: connect_phase
```

In the Listing:

- Line 63 shows the master agent monitor‘s analysis port connected to the coverage collection port (See Section B.3.2 for background). The write() implementation is provided by the coverage collection agent.
- Lines 66 deal with the register model predictor. The map for the predictor is set to the default map for the register model. *Note that in this case, only one master is considered, and the register model is configured with only one map. The second master is not considered for in the example.*
- Line 67 sets the predictor‘s version of register model adapter to using the register adapter.
- Line 68 connects the master agent0 monitor‘s analysis port to the input bus port of the predictor.

Note that the master agent monitor puts out transactions of type ***wb_transaction.*** . The external register adapter converts these into uvm_reg transactions. The predictor then performs a lookup in the register map for the appropriate register and then updates the model. During the updating process, the register sample() is called on account of the coverage options being set in the model, resulting in coverage being collected.

B.6 Passive UVM Test Class

The test class in this UVM testbench is passive. It does not drive any stimulus into the DUT. On the other hand, it configures all the components so that they can monitor the legacy testbench and handles the phasing allowing the tests to terminate gracefully. The top-level test is therefore expected to do the following:

- Configure the master and slave agents with the required memory addresses.
- Set up the master and slave agents as passive agents.
- Provide the appropriate objections in the main_phase() to prevent the UVM test from completing until the other legacy side has completed.

An example top-level passive test is shown in Listing B.11.

Listing B.11: Top Level Passive Test

```
5 class wb_dma_env_test extends uvm_test;
6     wb_config master_configs[2];
7     wb_config slave_configs[2];
8
9     int slave_adr_max ;
10    int slave_adr_min;
11
12
13    `uvm_component_utils(wb_dma_env_test)
14
15    wb_dma_env env;
16    uvm_active_passive_enum is_active;
17
18    function new(string name, uvm_component parent);
19        super.new(name, parent);
20    endfunction
21
22    virtual function void build_phase(uvm_phase phase);
23        super.build_phase(phase);
```



```
24       slave_adr_max = 32'h0fffffe;
25       env = wb_dma_env::type_id::create("env", this);
26       is_active=UVM_PASSIVE;
27
28       for(int i = 0; i < 2; i++) begin
29          master_configs[i] = wb_config::type_id::create($psprintf("
               master_configuration[%02d]",i));
30          master_configs[i].randomize with {min_addr == 0; max_addr ==
               slave_adr_max; max_n_wss == 10; };
31          slave_configs[i] = wb_config::type_id::create($psprintf("
               slave_configuration[%02d]",i));
32          slave_configs[i].randomize with {min_addr == slave_adr_min;
               max_addr == slave_adr_max; max_n_wss == 10; };
33
34       end
35
36       uvm_config_db #(wb_config)::set(null,"uvm_test_top.env.master_agent0
            ","mstr_agent_cfg",master_configs[0] );
37       uvm_config_db #(wb_config)::set(null,"uvm_test_top.env.master_agent1
            ","mstr_agent_cfg",master_configs[1] );
38       uvm_config_db #(wb_config)::set(null,"uvm_test_top.env.slave_agent0
            ","slv_agent_cfg",slave_configs[0] );
39       uvm_config_db #(wb_config)::set(null,"uvm_test_top.env.slave_agent1
            ","slv_agent_cfg",slave_configs[1] );
40
41       uvm_config_db #(uvm_active_passive_enum)::set(null,"uvm_test_top.env
            .master_agent0",is_active,UVM_PASSIVE);
42       uvm_config_db #(uvm_active_passive_enum)::set(null,"uvm_test_top.env
            .master_agent1",is_active,UVM_PASSIVE);
43       uvm_config_db #(uvm_active_passive_enum)::set(null,"uvm_test_top.env
            .slave_agent0",is_active,UVM_PASSIVE);
44       uvm_config_db #(uvm_active_passive_enum)::set(null,"uvm_test_top.env
            .slave_agent1",is_active,UVM_PASSIVE);
45
46    endfunction
47
48    virtual task reset_phase(uvm_phase phase) ;
49       super.reset_phase(phase);
50    endtask
51
52    virtual task main_phase(uvm_phase phase) ;
53       if (phase != null)
54         phase.raise_objection( this, get_type_name() );
55       wait (test.test_done==1);
56       phase.drop_objection( this, get_type_name() );
57
58    endtask
```

In Listing B.11:

- Line 15 contains an instance of the testbench environment class for the test class. The test class configures the components in this environment class.
- Lines 18 – 20 provide the constructor for the class.

- Lines 22 -46 contain the build phase.
- Lines 29 – 34 The build phase creates the configuration classes for the master and slave components, randomizes the configurations.
- Lines 36 – 39 show the configuration classes placed in the config_db.
- Lines 41 – 44 set the agents in the master and slave agents in the Env to being PASSIVE agents. In this mode, these agents do not drive the interface but just provide monitoring. The driver and sequencers in the agents are not built in this phase on account of the UVM_PASSIVE setting. To see the complete code for the agent, Look at Section 8.3.4 where the UVM_PASSIVE is described.
- Lines 52 – 58 describe the main_phase. This phase is waiting for the legacy testbench to end.

B.7 Observing Generated Coverage Results

It is time to observe the results available from the instrumented testbench. After running the simulations, the coverage is written from the simulation into a coverage database. You can run a report to get the report in many different formats. *(Note that the actual commands vary from one simulator to another.)* , The following results were obtained by running the VCS simulator and running the Unified Report Generator from Synopsys. The covergroups include both the register model coverage groups that were generated by the tool and the manually developed coverage groups.

Listing B.12: Top Level Coverage Results

```
Testbench Group List

Total Groups Coverage Summary
SCORE   INST SCORE WEIGHT
 35.32   26.83       1


Total groups in report: 16
```

SCORE	INSTANCES	WEIGHT	GOAL	AT LEAST	PER INSTANCE	AUTO BIN MAX	PRINT MISSING	COMMENT	NAME
24.40	25.00	1	100	1	1	64	64		test::ral_reg_CH0_CSR::cg_bits
24.40	25.00	1	100	1	1	64	64		test::ral_reg_CHN_CSR::cg_bits
25.00	25.00	1	100	1	1	64	64		test::ral_reg_CSR::cg_bits
25.00	25.00	1	100	1	1	64	64		test::ral_reg_INT_MASKA::cg_bits
25.00	25.00	1	100	1	1	64	64		test::ral_reg_INT_MASKB::cg_bits
25.00	25.00	1	100	1	1	64	64		test::ral_reg_INT_SRCA::cg_bits
25.00	25.00	1	100	1	1	64	64		test::ral_reg_INT_SRCB::cg_bits
25.00	25.00	1	100	1	1	64	64		test::ral_reg_CHN_SZ::cg_bits
25.00	25.00	1	100	1	1	64	64		test::ral_reg_CHN_A0::cg_bits
25.00	25.00	1	100	1	1	64	64		test::ral_reg_CHN_AM0::cg_bits
25.00	25.00	1	100	1	1	64	64		test::ral_reg_CHN_AM1::cg_bits
25.00	25.00	1	100	1	1	64	64		test::ral_reg_CHN_SWPTR::cg_bits
46.43	30.53	1	100	1	1	64	64		test::ral_reg_CHN_DESC::cg_bits
46.55	30.56	1	100	1	1	64	64		test::ral_reg_CHN_A1::cg_bits
73.33	—	1	100	1	0	64	64		test::wb_dma_env_cov::cg_trans
100.00	100.00	1	100	1	1	64	64		test::ral_block_wb_dma::cg_addr

Looking at the report, one sees that several covergroups are not adequately covered while others may have more coverage. Looking at the covergroups generated for the CH0_CSR register whose coverage was shown in Section B.3.1, one sees that not all the bits of this register are exercised in the simulations.

If you study the core and all related documentation, you may uncover functionality which is not exercised or tested completely. This is illustrated in the drill down report below.

Listing B.13: Drill Down of Coverage Results

```
Summary for Group Instance    CSR


CATEGORY   EXPECTED UNCOVERED COVERED PERCENT
Variables  4        3         1       25.00

Variables for Group Instance  CSR

VARIABLE EXPECTED UNCOVERED COVERED PERCENT GOAL WEIGHT AT LEAST AUTO BIN MAX COMMENT
PAUSE    4        3         1       25.00   100  4      1        0

------------------------------------------------------------------------------

Summary for Variable PAUSE

CATEGORY           EXPECTED UNCOVERED COVERED PERCENT
User Defined Bins  4        3         1       25.00

User Defined Bins for PAUSE

Uncovered bins

NAME           COUNT AT LEAST NUMBER
bit_0_rd_as_1  0     1        1
bit_0_wr_as_1  0     1        1
bit_0_wr_as_0  0     1        1

Covered bins

NAME           COUNT AT LEAST
bit_0_rd_as_0  34    1
```

I strongly suggest that the covergroups produced by the tools automatically be reviewed to ensure that the generated coverage points are necessary and sufficient for the verification task at hand and reduce “noise“ in the output reports.

Rather than modifying the generated covergroups, see if you can supplement them with additional coverage. To manage the problem of a noisy report, see if you can use the exclusion flows provided by your simulator vendor to fine tune the results.

B.8 Cover Additional Scenarios by Tweaking Additional Scenarios

You can run through the complete regression available of the legacy environment using the existing tests using this environment. The UVM testbench is set up as a monitor to get coverage on the DUT. To continue, you can simply comment out the existing legacy drivers from the testbench connecting to the DUT. The current passive test is converted into an active test by just making each of the agents active, as seen in Listing B.14.

Listing B.14: Top Level Active Test

```
42      uvm_config_db #(uvm_active_passive_enum)::set(null,"uvm_test_top.env.master_agent0",is_active,UVM_ACTIVE);
43      uvm_config_db #(uvm_active_passive_enum)::set(null,"uvm_test_top.env.master_agent1",is_active,UVM_ACTIVE);
44      uvm_config_db #(uvm_active_passive_enum)::set(null,"uvm_test_top.env.slave_agent0",is_active,UVM_ACTIVE);
45      uvm_config_db #(uvm_active_passive_enum)::set(null,"uvm_test_top.env.slave_agent1",is_active,UVM_ACTIVE);
```

All you now have to do is add sequences to the sequence library of the master agents and test features that need testing as revealed by the coverage reports and the test/verification plan.