SYNOPSYS®

CUSTOMER EDUCATION SERVICES

SystemVerilog Verification UVM 1.1 Workshop

Lab Guide

40-I-054-SLG-004 2012.09-SP1

Synopsys Customer Education Services
700 East Middlefield Road
Mountain View, California 94043

Workshop Registration: http://training.synopsys.com

www.synopsys.com

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AEON, AMPS, ARC, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, CoMET, Confirma, CODE V, Design Compiler, DesignSphere, DesignWare, Eclypse, Formality, Galaxy Custom Designer, Global Synthesis, HAPS, HapsTrak, HDL Analyst, HSIM, HSPICE, Identify, Leda, LightTools, MAST, MaVeric, METeor, ModelTools, , NanoSim, NOVeA, OpenVera, ORA, PathMill, Physical Compiler, PrimeTime, SCOPE, SiVL, SNUG, SolvNet, Sonic Focus, STAR Memory System, SVP Café , Syndicated, Synplicity, logo Synplify, Synplify Pro, Synthesis ConstraintsOptimization Environment, TetraMAX, UMRBus, VCS, Vera, YieldExplorer .

Trademarks (™)

AFGen, Apollo, ASAP, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Encore, EPIC, Galaxy,HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIMplus, i-Virtual Stepper, IC Compiler, IICE, in-Sync, iN-Tandem, Intelli, Jupiter, BEST, Columbia, Columbia-CE, Cosmos, CosmosLE, CosmosScope, CRITIC, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Macro-PLUS, Magellan, MAP-in SMMars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, ORAengineering, Physical Analyst, Planet, Planet-PL, Platform Architect, Polaris, Power Compiler, Processor Designer, CustomExplorer, CustomSim, CustomWaveView, DC Expert, DC Professional, DC Ultra, Design Analyzer, Raphael, RippledMixer, Saturn, Scirocco, Scirocco-i, SiWare, SPW, Star-RCXT, Star-SimXT, StarRC, Synphony ModelSystem Compiler, System Designer, System Studio, TAP-in SMTaurus, TotalRecall, TSUPREM-4, VCSi, VHDL Compiler, Virtualizer, VMC, Worksheet Buffer.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license. ARM and AMBA are registered trademarks of ARM Limited. Saber is a registered trademark of SabreMark Limited Partnership and is used under license. All other product or company names may be trademarks of their respective owners.

Document Order Number: 40-I-054-SLG-004
SystemVerilog Verification UVM 1.1 Lab Guide

1 UVM Environment

Learning Objectives

After completing this lab, you should be able to:

- Create a simple UVM test environment
- Embed report messages
- Compile the test environment
- Run simulation and observe results
- Add data, sequencer and driver classes to the environment
- Compile and simulate the environment to observe behavior

Lab Duration: 45 minutes

Getting Started

UVM consists of a set of coding guidelines with a set of base classes and macros. The set of base classes and macros assist you in developing testbenches that are consistent in look and feel. The set of coding guidelines enable you to develop testbench components which are robust and highly re-usable. As a result, you will spend less time modifying, maintaining the verification infrastructure and more time verifying your designs.

In this first lab, you will start the process of build a UVM verification environment using the UVM base classes and macros following the UVM coding guidelines:

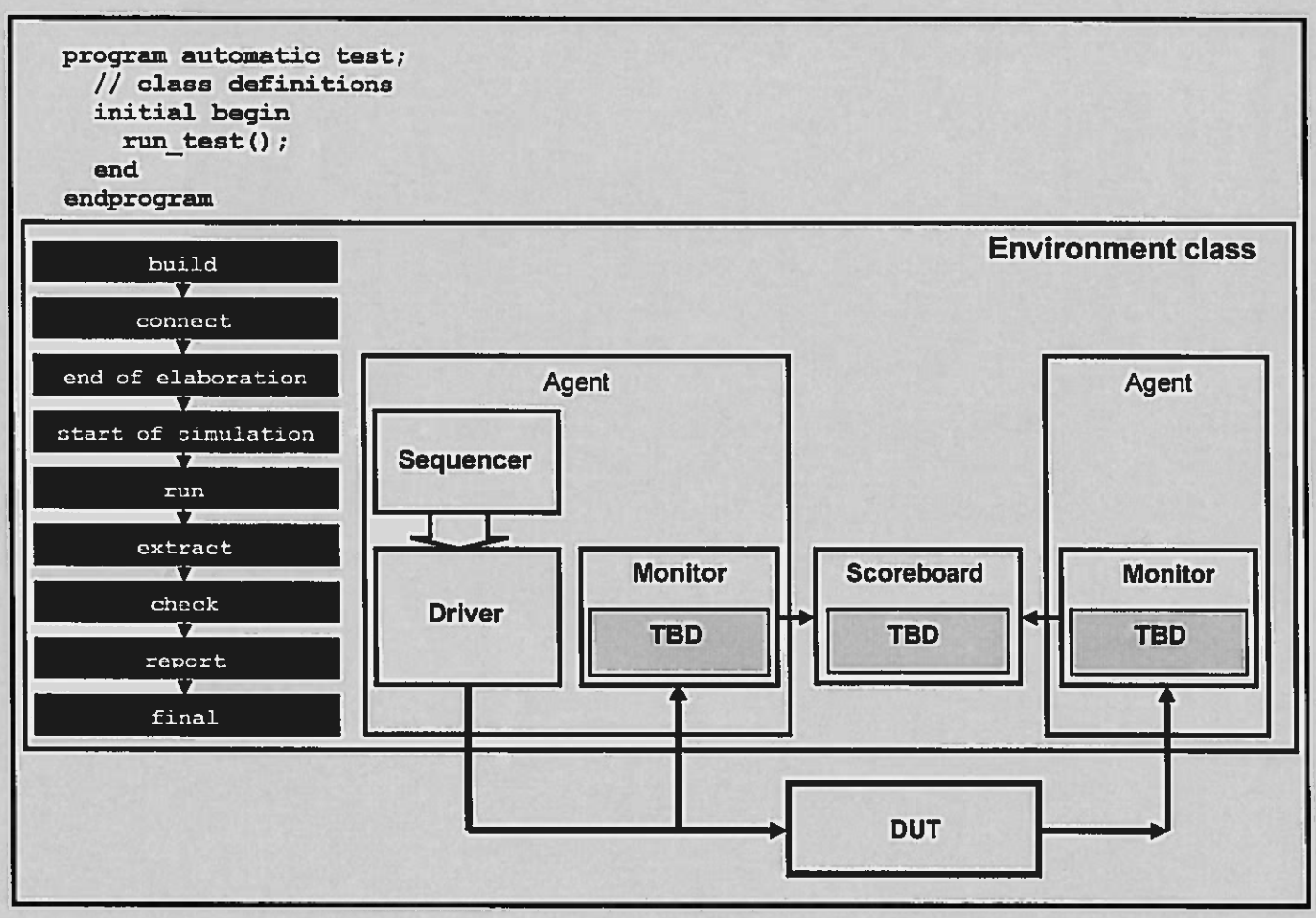


Figure 1. Lab 1 Testbench Architecture

Lab Overview

After you log in, you will see three directories: **`labs`**, **`solutions`** and **`rtl`**.

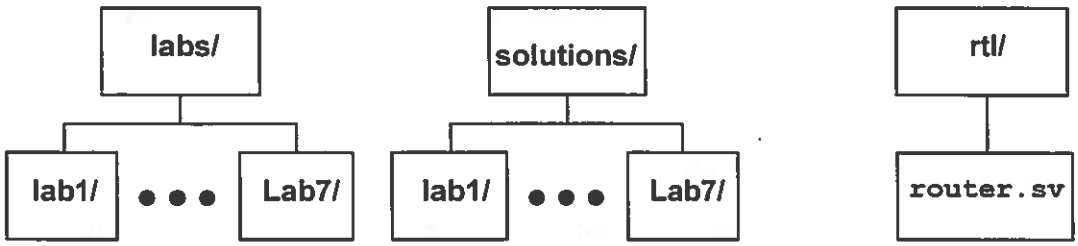


Figure 2. Lab Directory Structure

For each individual lab, you will work in the specified lab directory. Should you have a question during the lab and want to know what the potential solution is, you can reference the sample solution provided in the **`solutions`** directory.

The work flow for this lab is illustrated as follows:

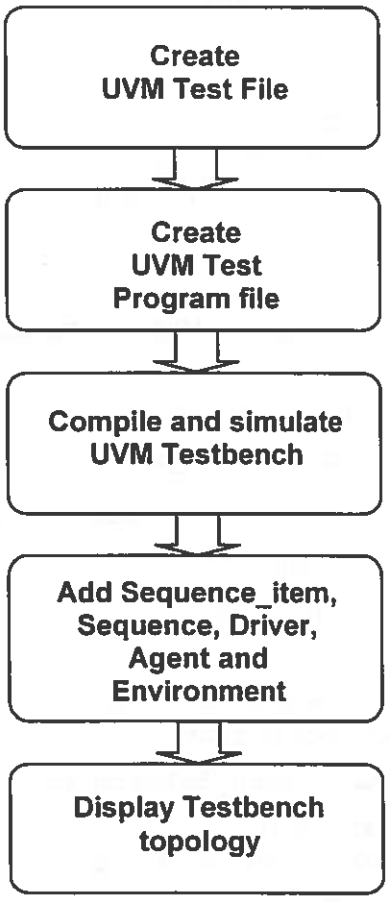


Figure 3. Lab 1 Flow Diagram

Building a UVM Testbench

Task 1. Create a Simple Test

For this first task, you will create a simple test and a program block to execute this simple test. Use the lecture material as your reference.

1. Go into **lab1** directory:

```
> cd labs/lab1
```

2. Open the existing **test_collection.sv** file with an editor, and look for the comment that start with: "**// Lab 1: task 1, step 2**".
   - Enter the class declaration as described in the comment
3. Look for the "**// Lab 1: task 1. Step 3**" comment.
   - Register the class with the factory
4. Look for "**// Lab 1: task 1. Step 4**" comment.
   - Enter the constructor code as described in the comments

   Note: You will be asked to enter the following statement at the beginning of each method. This is to help you during debugging. With this statement embedded in every method, you can easily see how things are being executed sequentially by setting the report verbosity to UVM_HIGH. Within the statement, **%m** is a format specifier that prints the current hierarchical path.

```
`uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
```

5. Near the bottom of the file, look for "**// Lab 1: task 1, step 5**" comment in **start_of_simulation_phase()** method.
   - Enter the helpful debugging code as described in the comments
6. Save and close the file.
7. Open a new file, call it **test.sv**. Enter the following test code:

```
program automatic test;
import uvm_pkg::*;

`include "test_collection.sv"

initial begin
  $timeformat(-9, 1, "ns", 10);
  run_test();
end

endprogram
```

8. Save and close the file.

9. Compile and simulate this simple UVM testbench:

```
> vcs -sverilog -ntb_opts uvm-1.1 test.sv
> simv +UVM_TESTNAME=test_base
```

At the end of simulation, you should see something like the following:

```
UVM_INFO @ 0: reporter [RNTST] Running test test_base...
#### Factory Configuration (*)
  No instance or type overrides are registered with this factory

All types registered with the factory: 37 total
(types without type names will not be printed)
  Type Name
  ---------
  test_base
```

Test as specified by +UVM_TESTNAME

You will be modifying the test via factory configuration with overrides in future labs

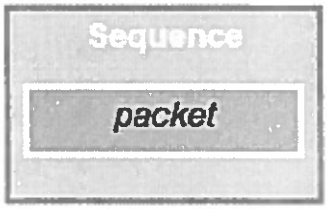
`factory.print()` displays all class types registered in the factory via the `` `uvm_component_utils `` macro

Note: The compilation switch to enable UVM is `-ntb_opts`. There are several flavors of UVM switches: `uvm`, `uvm-1.0`, `uvm-1.1` (in the future there will be `uvm-1.2`, `uvm-2.0` etc.). Be careful and make sure that you are using the proper version for your project.

Task 2. Create a Simple Environment

In this task, you will build a simple environment that includes sequence item (`packet`), sequence (`packet_sequence`), sequencer (`sequencer`), driver (`driver`) agent (`input_agent`) and environment (`router_env`). With very little effort, you can start to generate stimulus and see interactions between the sequencer and the driver.

1. Open the `packet.sv` file with an editor, and look for the comments that start with: "`// Lab 1`" and enter the following: (If necessary, look in the slides for the exact syntax)



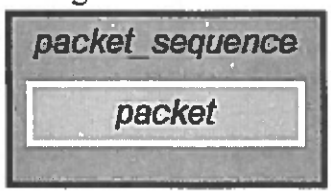
   - The **class declaration**
   - The `sa`, `da`, and `payload` properties
   - And, the **constructor**

Note: There is a major difference between data class constructor and component class constructor. In the data class constructor, there is no parent component handle in the argument.

To create the supporting methods (print, copy, compare, etc.) for properties in a UVM class, you should use `` `uvm_field_* `` macros embedded within the `` `uvm_object_utils_begin/end `` macros (to be covered in the next unit). These are already done in the file for you, along with a constraint.

2. Save and close the file

3. Open the **packet_sequence.sv** file with an editor, and look for the comments that start with: "**// Lab 1**". Enter the following:



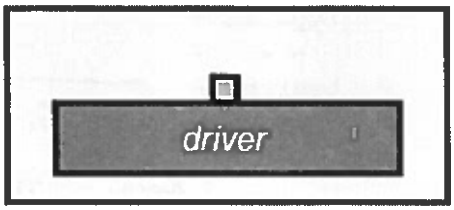
   - The **class declaration**
   - And the **body()** method (see comments)

4. Save and close the file.

With the **packet** and the **packet_sequence** classes, you have created the base mechanism for generating stimulus in UVM.

The next step is to create the component that will be processing the stimulus.

5. Open the **driver.sv** file with an editor, and look for the comments that start with: "**// Lab 1**". Enter the following:



   - The **class declaration**
   - Print the request (**req**) sequence item in the **run_phase()** method

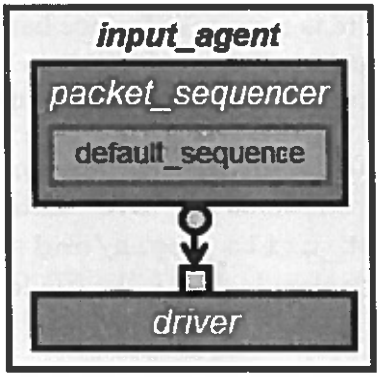
For now, the **driver** just gets a **packet** (**req**) from the sequencer via the built-in **seq_item_port** and displays the content on the console. In future labs, you will be calling the device driver to drive the contents of the request object through the DUT.

6. Save and close the file.

Once you have the stimulus generating mechanism and the component to process the stimulus, you are ready to create the container that will house them. This container is called an agent.

7. Open the **input_agent.sv** file with an editor, and look for the comments that start with: "**// Lab 1**". Enter the following:

   - Use typedef to create a **packet_sequencer** class for packet
   - Declare the **input_agent** class
   - Create an instance of **packet_sequencer** and **driver**
   - Create these in the build phase
   - Connect the driver's and sequencer's TLM ports in the connect phase

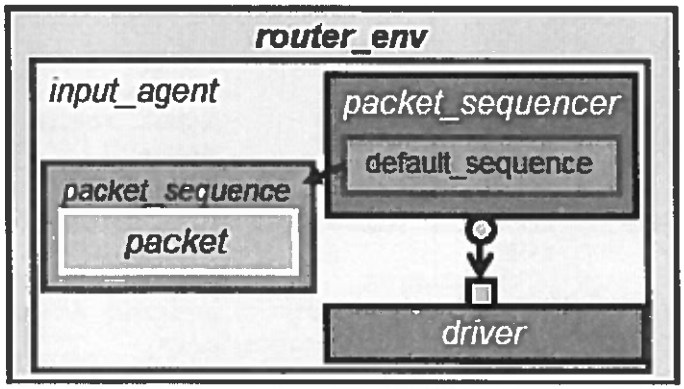


8. Save and close the file.

You are now ready to build an environment class for your DUT.

9. Open the **`router_env.sv`** file with an editor, and look for the comments that start with: "**`// Lab 1`**". Enter the following:

   - Declare a **`router_env`** class
   - Create an instance of **`input_agent`**
   - Construct it in the build phase
   - Then, set the agent's sequencer to execute **`packet_sequence`** as the default sequence in the main phase



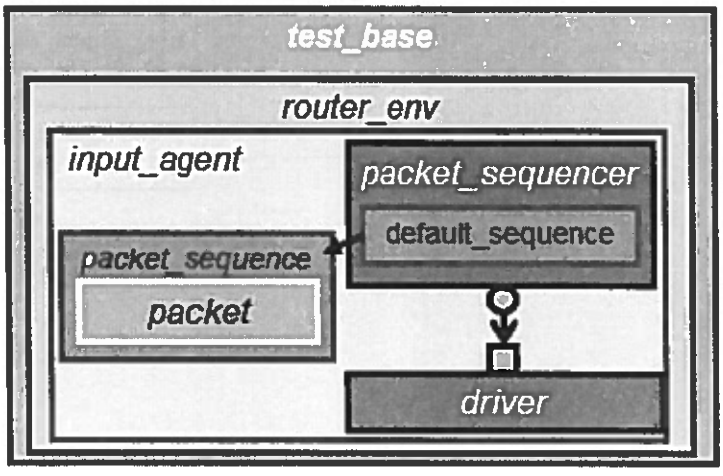
10. Save and close the file.

An environment consists of a set of components that are configured to process some default stimulus through the DUT. Where these configurations and stimulus will be managed is at the test level.

11. Open the **`test_collection.sv`** file with an editor and look for the comments that start with with: "**`// Lab 1, task 2, step 11`**". Enter the following:

   - Include the environment file
   - Create an instance of the environment
   - Construct the environment object
   - Print the test topology
   -

12. Save and close the file.



Task 3. Run Test

1. Use **make** to compile and run simulation.

   ```
   > make
   ```

   You should see that ten packets were printed.

2. Take a look at the simulation result:

   ```
   > less simv.log
   ```

   You should see something like the following:

```
UVM_INFO @ 0: reporter [RNTST] Running test test_base...
UVM_INFO @ 0: reporter [UVMTOP] UVM testbench topology:
----------------------------------------------------------------
Name                        Type                        Size  Value
----------------------------------------------------------------
uvm_test_top                test_base                   -     @460
  env                       router_env                  -     @469
    i_agent                 input_agent                 -     @477
      drv                   driver                      -     @599
        rsp_port            uvm_analysis_port           -     @614
        sqr_pull_port       uvm_seq_item_pull_port      -     @606
      seqr                  uvm_sequencer               -     @490
        rsp_export          uvm_analysis_export         -     @497
        seq_item_export     uvm_seq_item_pull_imp       -     @591
        arbitration_queue   array                       0     -
        lock_queue          array                       0     -
        num_last_reqs       integral                    32    'd1
        num_last_rsps       integral                    32    'd1
----------------------------------------------------------------

#### Factory Configuration (*)

  No instance or type overrides are registered with this factory

All types registered with the factory: 42 total
(types without type names will not be printed)

  Type Name
  ---------
  driver
  input_agent
  packet
  packet_sequence
  router_env
  test_base
```

Test topology printed in **table** format.
You can display the content in tree format with:
uvm_top.print_topology(uvm_default_tree_printer);

factory.print() result.
All class type registered in the factory via the **`uvm_component_utils()** and the **`uvm_object_utils()** macro are printed.

If you name your tests properly, you can easily identify all of your tests here.

Task 4. Run Testbench with Different Log Verbosity

In the previous task, you may have noticed that none of the embedded **`uvm_info`** messages were printed. This is because the default display verbosity of the UVM reporting mechanism filters out all levels below **`UVM_MEDIUM`**. In the **`router_env`**, each of the **`uvm_info`** message were set to the verbosity level **`UVM_HIGH`**. The verbosity being filtered out are: **`UVM_HIGH`**, **`UVM_FULL`** and **`UVM_DEBUG`**. The recommended way to treat these verbosity levels is to use:

- **`UVM_HIGH`** for tracing messages
- **`UVM_FULL`** for general debugging messages
- **`UVM_DEBUG`** for more detailed debugging messages

1. Turn on the trace messages:
   ```
   > make verbosity=UVM_HIGH
   ```
   You should now see the embedded tracing messages.

2. Turn on debugging messages at the most verbose level:
   ```
   > make verbosity=UVM_DEBUG
   ```

3. If you have free time, look through the source files in **`$VCS_HOME/etc/uvm`**. You will find that reading through each of the source code will give you a lot of insights into how each class and macro works.

You are done with Lab 1!

Answers / Solutions

test.sv Solution:

```
program automatic test;
import uvm_pkg::*;
`include "test_collection.sv"
initial begin
   $timeformat(-9, 1, "ns", 10);
   run_test();
end
endprogram
```

test_collection.sv Solution:

```
class test_base extends uvm_test;
  `uvm_component_utils(test_base)
  router_env env;

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction: new

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    env = router_env::type_id::create("env", this);
  endfunction: build_phase

  virtual function void start_of_simulation_phase(uvm_phase phase);
    super.start_of_simulation_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    uvm_top.print_topology();
    factory.print();
  endfunction: start_of_simulation_phase
endclass: test_base
```

driver.sv Solution:

```
class driver extends uvm_driver #(packet);
  `uvm_component_utils(driver)

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction: new

  virtual task run_phase(uvm_phase phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    forever begin
      seq_item_port.get_next_item(req);
      req.print();
      seq_item_port.item_done();
    end
  endtask: run_phase
endclass: driver
```

packet.sv Solution:

```
class packet extends uvm_sequence_item;
  rand bit [3:0] sa, da;
  rand bit [7:0] payload[$];

  `uvm_object_utils_begin(packet)
    `uvm_field_int(sa, UVM_ALL_ON | UVM_NOCOMPARE)
    `uvm_field_int(da, UVM_ALL_ON)
    `uvm_field_queue_int(payload, UVM_ALL_ON)
  `uvm_object_utils_end

  constraint valid {
    payload.size inside {[1:10]};
  }

  function new(string name = "packet");
    super.new(name);
   `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction: new
endclass: packet
```

input_agent.sv Solution:

```
  typedef uvm_sequencer #(packet) packet_sequencer;

  class input_agent extends uvm_agent;
    packet_sequencer seqr;
    driver drv;

    `uvm_component_utils(input_agent)

    function new(string name, uvm_component parent);
      super.new(name, parent);
      `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    endfunction: new

    virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
      seqr = packet_sequencer::type_id::create("seqr", this);
      drv  = driver::type_id::create("drv", this);
    endfunction: build_phase

    virtual function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);
      `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
      drv.seq_item_port.connect(seqr.seq_item_export);
    endfunction: connect_phase

  endclass: input_agent
```

router_env.sv Solution:

```
class router_env extends uvm_env;
  input_agent i_agent;
  `uvm_component_utils(router_env)

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction: new

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    i_agent = input_agent::type_id::create("i_agent", this);
    uvm_config_db #(uvm_object_wrapper)::set(this, "i_agent.seqr.main_phase",
"default_sequence", packet_sequence::get_type());
  endfunction: build_phase
endclass: router_env
```

packet_sequence.sv Solution:

```
class packet_sequence extends uvm_sequence #(packet);
  `uvm_object_utils(packet_sequence)

  function new(string name = "packet_sequence");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction: new

  task body();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if (starting_phase != null) begin
      starting_phase.raise_objection(this);
    end
    repeat(10) begin
      `uvm_do(req);
    end
    if (starting_phase != null) begin
      starting_phase.drop_objection(this);
    end
  endtask: body

endclass: packet_sequence
```

2 Modify Constraints in Test

Learning Objectives

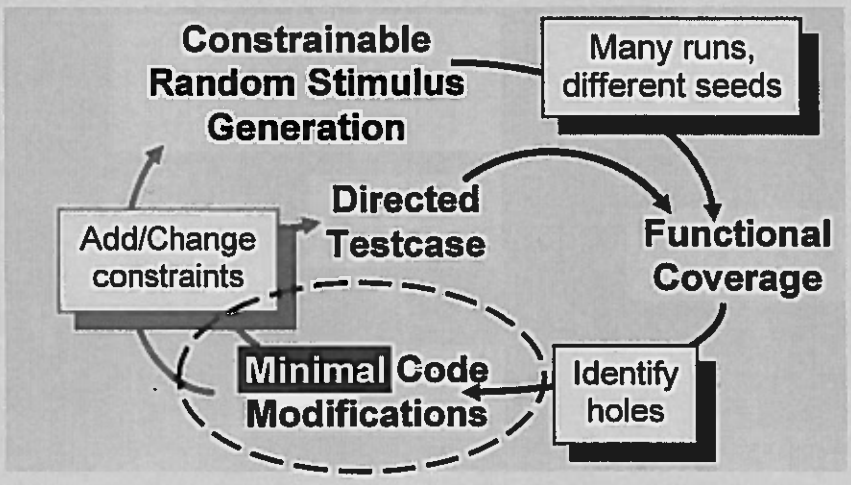
After completing this lab, you should be able to:

- Modify the existing packet constraint by creating an extension of the class
- Create a test to use this modified packet class
- Compile, simulate and check results

Lab Duration: 15 minutes

Getting Started

In Lab 1, you built a UVM testbench environment class with a packet class that fully randomized the source address and the destination address. In this lab, you will create a modified packet class to set specific set of address of interest and develop a test to execute it.



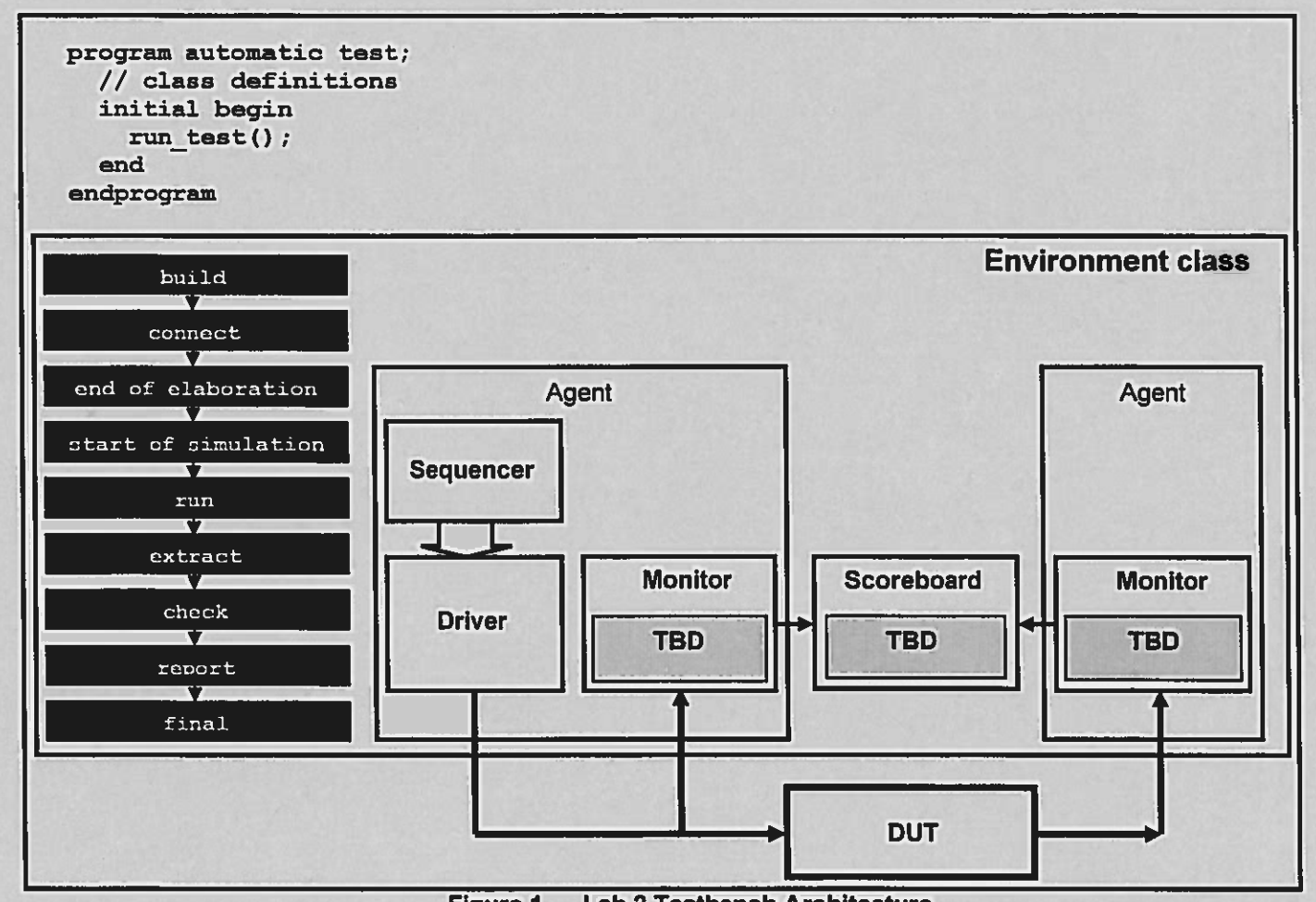


Figure 1. Lab 2 Testbench Architecture

Lab Overview

The work flow for this lab is illustrated as follows:

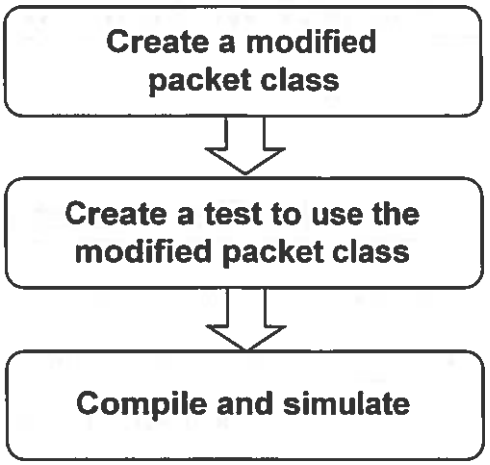


Figure 2. Lab 2 Flow Diagram

Implement Constrained Test

What does it mean to write a test? Often time, it just means to change constraints within stimulus classes. In this lab, you will create a directed test to exercise only destination address 3 of the DUT.

Task 1. Go into lab2 Working Directory

1. CD into the lab2 directory

```
> cd ../lab2
```

Task 2. Create a Modified Packet Class

In this task, create a simple contraint which constrains the destination address to 3.

1. Open the **packet_da_3.sv** file in an editor
2. Look for the ToDo comments and write a constraint to set destination address (**da**) to 3
3. Save and close the file

Task 3. Create a Test to Use the Modified Packet Class

1. Open **test_collection.sv** file in an editor
2. Bring in the new packet definition by including the **packet_da_3.sv** file
3. Inside the **test_da_3_inst** class, use instance override to configure the sequence to use **packet_da_3** instead of **packet** definition
4. Save and close the file

Task 4. Compile And Simulate with New Test

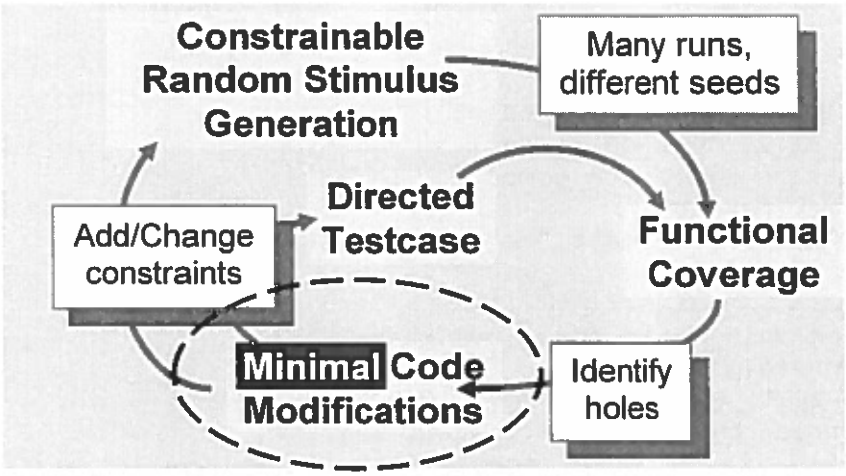
1. Compile and simulate the testbench

```
> make test=test_da_3_inst
```

2. Check to see that the destination address for thes packets are all 3
3. Check to see that the **Factory Configuration** report displays the **Instance Overrides** correctly
4. Constrast it with running the fully randomized test:

```
> make
```

The goal of testbench methodology is to simplify development of individual tests. Can you see that writing new tests are easy to do once the environment is set up properly?



Task 5. Optional: Configure with Type Override

If you have time, try writing one more test to override all instances of **`packet`** with **`packet_da_3`**. Call this test **`test_da_3_type`**. Compile and simulation to make sure it works properly.

Congratuations, you have completed Lab 2!

Answers / Solutions

test_collection.sv Solution:

```
class test_base extends uvm_test;
...
`include "packet_da_3.sv"

class test_da_3_inst extends test_base;
  `uvm_component_utils(test_da_3_inst)
  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    set_inst_override_by_type("env.i_agent.seqr.*",
                              packet::get_type(), packet_da_3::get_type());
  endfunction
endclass

class test_da_3_type extends test_base;
  `uvm_component_utils(test_da_3_type)
  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    set_type_override_by_type(packet::get_type(), packet_da_3::get_type());
  endfunction
endclass
```

packet_da_3.sv Solution:

```
class packet_da_3 extends packet;
  `uvm_object_utils(packet_da_3)

  constraint da_3 {
    da == 3;
  }

  function new(string name = "packet_da_3");
    super.new(name);
   `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction
endclass
```

3 Configure Sequences

Learning Objectives

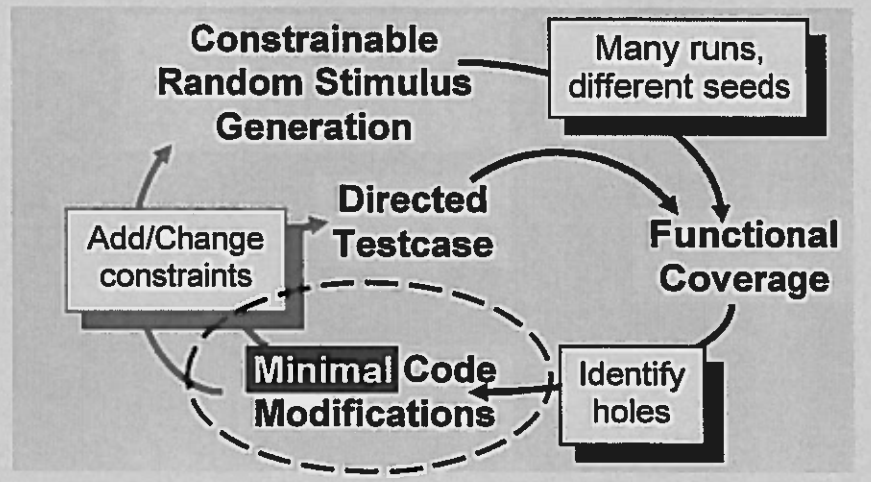
After completing this lab, you should be able to:

- Add configuration fields to packet_sequence
- Develop a test to set the packet_sequence's configuration fields.
- Create a reset sequence
- Develop a test to execute the reset sequence in the reset phase and packet_sequence in the main phase
- Compile and simulate

Lab Duration: 30 minutes

Getting Started

In Lab 2, through minimal code modification (OOP inheritance) you were able to develop a test to constrain the packet's destination address. In this lab, you will add configuration fields to packet_sequence to bring even more flexibility into the testbench.




```
program automatic test;
  // class definitions
  initial begin
    run_test();
  end
endprogram
```

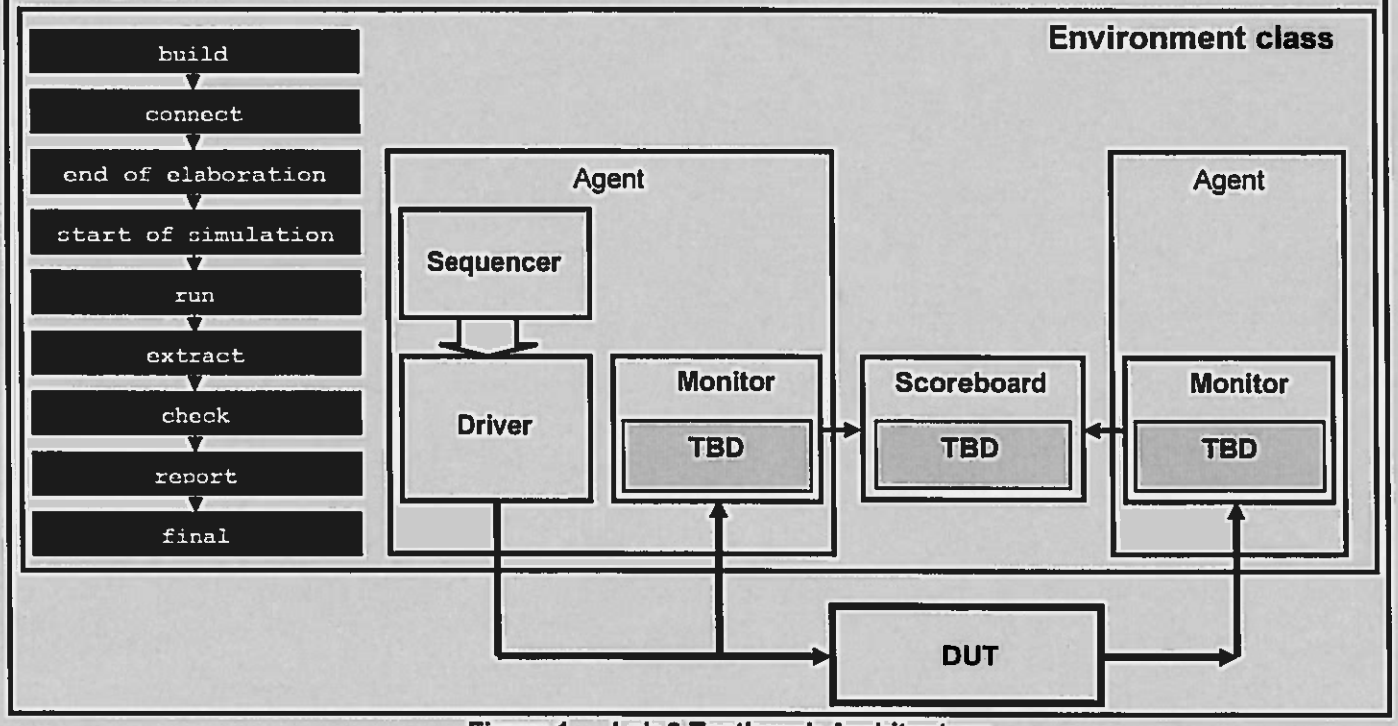


Figure 1. Lab 3 Testbench Architecture

Lab Overview

The work flow for this lab is illustrated as follows:

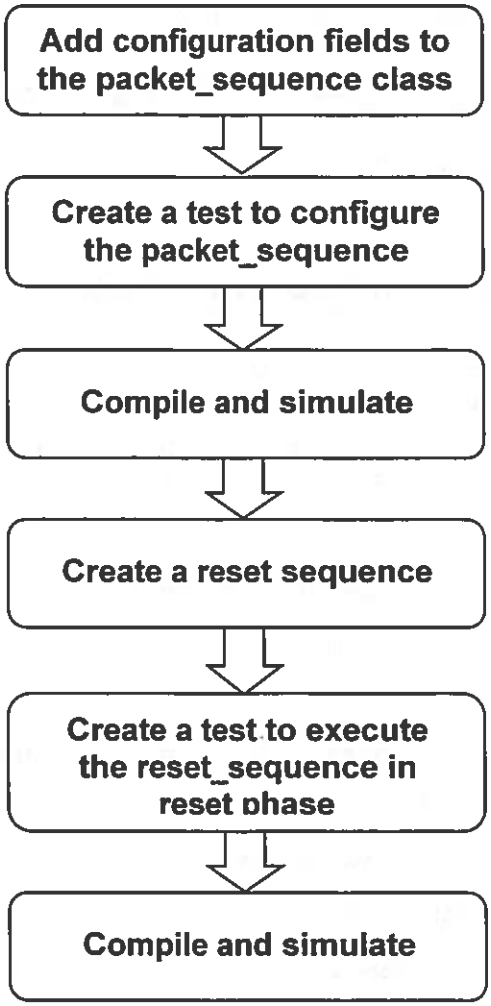


Figure 2. Lab 3 Flow Diagram

Implement Configurable Sequences

Modifying stimulus constraints through OOP inheritance is useful. But, by itself, can be limited in capability. In this lab, you will add configuration field to **packet_sequence** to bring more flexibility into the test development.

Task 1. Go into lab3 Working Directory

1. CD into the lab3 directory

```
> cd ../lab3
```

Task 2. Add Configuration Fields to Packet Sequence

Sequences are where stimulus are created. The ability to manage what happens within sequences is important in the development of individual tests. In this task, you will add configuration fields to constrain the source address and the destination address in each packet object. You will also add a field to control how many packets objects to generate per execution of the sequence's **body()** task.

1. Open the **packet_sequence.sv** file in an editor
2. Create the following fields:
   - `int item_count = 10;` **// number of items to create**
   - `int port_id = -1;` **// input port (source address)**
   - `bit[15:0] da_enable = '1;` **// destination address enable**
   - `int valid_da[$];` **// used to constrain da**
3. In **body()** task do the following:
   - Modify the **repeat()** statement to repeat for **item_count** number of times
   - Change **`uvm_do()** statement to constrain packet's **sa** and **da** (see comments in file for explanation)
4. Save and close the file

Task 3. Compile and Simulate

1. Compile and simulate the testbench to verify that simulation still runs the same with default configurations

```
> make
```

Task 4. Configure the Sequence in Test

In lab 2, you used OOP inheritance to set the packet's **da** to 3. With the updated packet_sequence, you can do the same thing by setting the sequence configuration.

1. Open **test_collection.sv** file in an editor
2. Create a test class extended from **test_base**, call it **test_da_3_seq**
3. Add the required **uvm_component_utils** macro and the **constructor** method
4. Create a **build_phase()** method to:
   - Set the packet_sequence's **da_enable** to **16'h0008**
   - Set the packet_sequence's **item_count** to **20**
5. Save and close the file

Task 5. Compile And Simulate with New Test

1. Compile and simulate the testbench

   ```
   > make test=test_da_3_seq
   ```

2. Check to see that 20 packets were generated and all destination address are 3's

   It may be difficult to count the number of packets processed by the driver. To make visual inspection of messages easier to interpret, change the way the transaction is being displayed in the driver.

3. Open the **driver.sv** file in an editor
4. Search for the **req.print()** statement and change it to:

   ```
   `uvm_info("DRV_RUN", {"\n", req.sprint()}, UVM_MEDIUM);
   ```

5. Save and close the file.
6. Compile and simulate the testbench again:

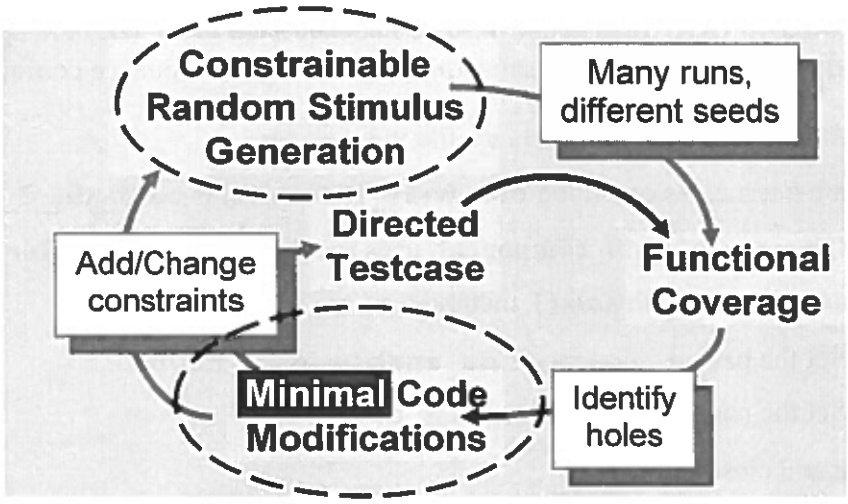
   ```
   > make test=test_da_3_seq
   ```

   At the end of simulation you should see:

   ```
   ** Report counts by id
   [DRV_RUN]     20
   ```

If you set the ID and Message fields of the UVM Report mechanism appropriately, you can make use of them during debugging to quickly isolate issues.

With sequence configuration, you now have the capability for both minimal code and constrainable stimulus generation!



Task 6. Examine the Reset Sequence

In typical testbenches, a reset sequence needs to happen before the functional verification sequence is executed. The best way to manage this is to make use of the scheduled phases. You will try this out with a simple reset agent and a simple reset sequence.

1. Examine the **`reset_sequence.sv`** file with an editor

   Notice that it only prints a message. Real reset operation will be implemented in the next lab.

2. Save and close the file

Task 7. Add Reset Agent to the Environment

In the interest of lab time, the reset agent has been done for you. You just need to embed it into the environment.

1. Open the **`router_env.sv`** file with an editor
2. Include the **`reset_agent.sv`** file
3. In the class, create an instance of **`reset_agent`** and call it **`r_agent`**
4. In the build phase, use proxy **`create()`** method to construct **`r_agent`**
5. And, configure **`r_agent`**'s sequencer (**`seqr`**) to execute **`reset_sequence`** at **`reset_phase`**.
6. Save and close the file

Task 8. Compile And Simulate

1. Compile and simulate the testbench

```
> make
```

   Do you see that the reset happens first?

   If you are not sure, change the report verbosity to UVM_FULL and look at the phase execution sequence to see when the reset is executed.

2. Simulate the testbench with UVM_FULL switch

```
> make verbosity=UVM_FULL
```

Task 9. Optional: Source Code Debugging in DVE

If you have time left in the lab, you may be interested in continuing with a DVE source code debugging session starting in the next page. Otherwise, you are done.

Congratulations, you have completed the regular portion of Lab 3!

Task 10. Optional: Interactive Source Code Debugging

There are times when you need to trace the problem in simulation by conducting an interactive source code debugging session. The following steps will take you through a simple example.

1. Take a look at the run-time command:

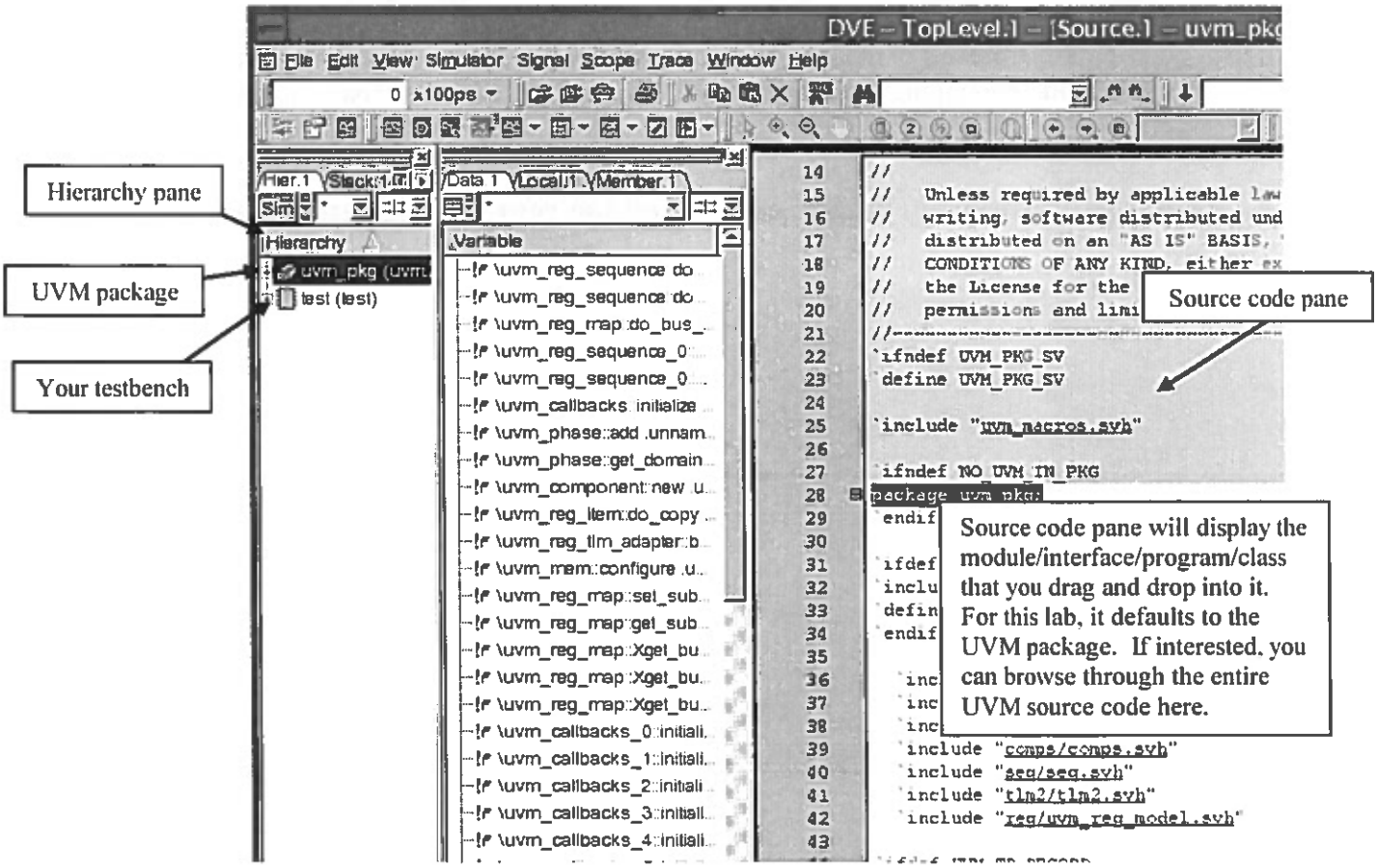
```
> make -n dve_i
```

You will see that run-time switch used for the interactive session is `-gui`

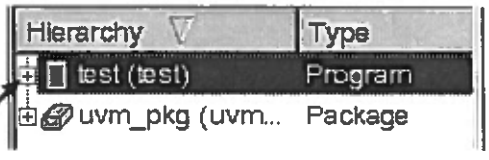
2. Start DVE in interactive mode

```
> make dve_i
```

You should see something like the following DVE display:

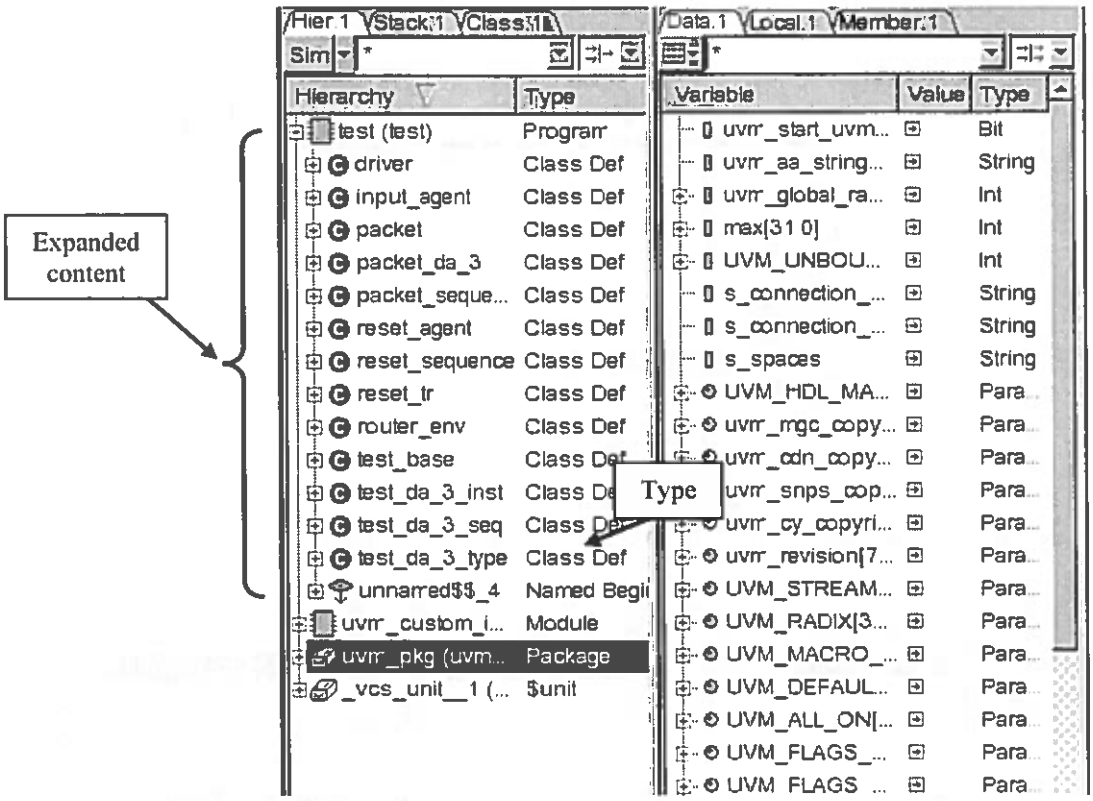


To expand and see contents of module/program/class or macro, simply click on the ⊞ symbol



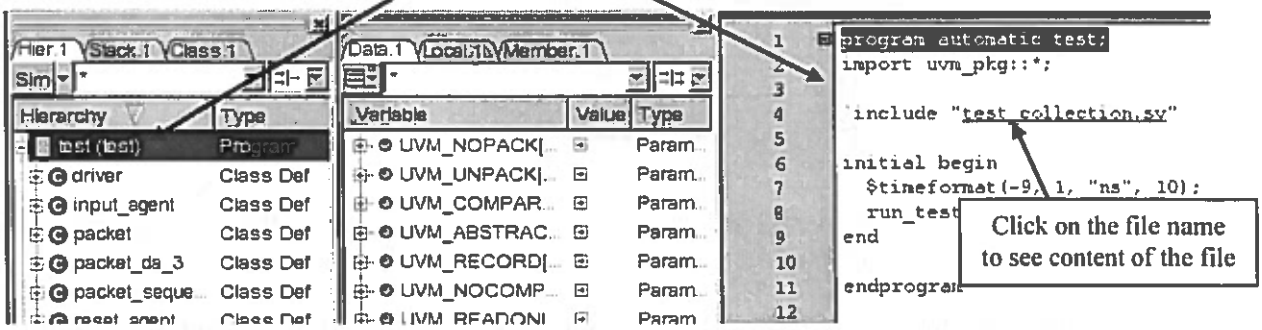
3. Click on the ⊞ symbol of test to expand it out

You should see something like the following:

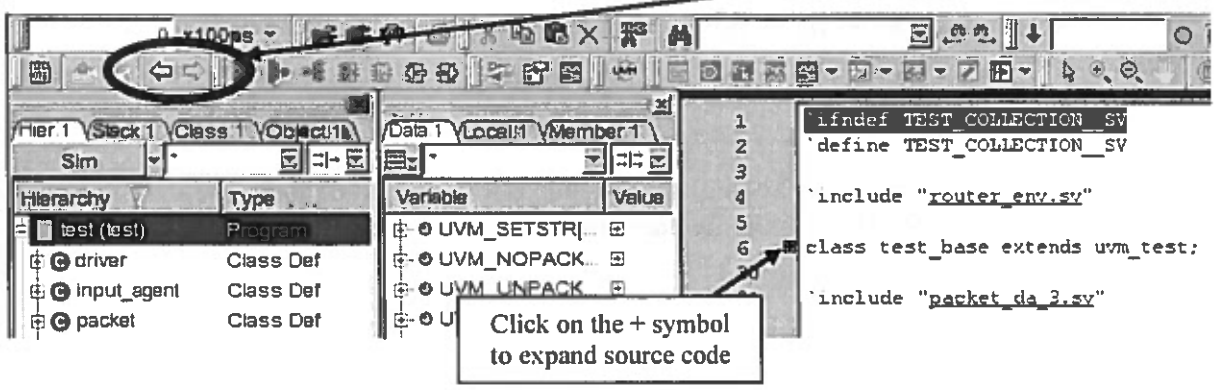


To see source code, double click on the entity of interest.

4. Double click on test to see the source code in source code pane

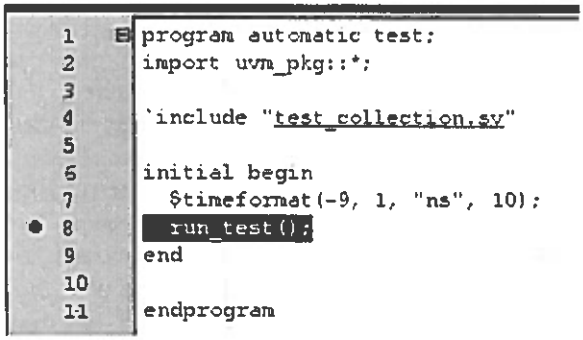


Within the source code pane, if you entered a source code, you can return back to where you were by clicking on the scope forward and backward buttons:

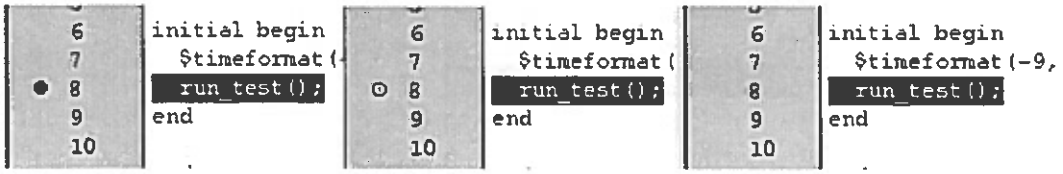


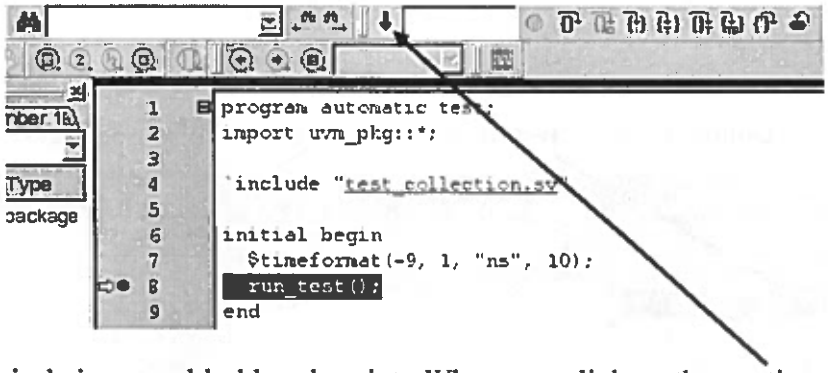
Source code debugging of simulation involves two general mechanisms: break point and single-stepping .

For break point, browse through the source code and click on the line number where you want to set the break point. On the first click, a solid circle will appear next to the line number.



On each subsequent clicks on the same line, the solid circle will change to a clear circle, then disappears, then back to a solid circle and so forth:





The solid circle is an enabled break point. When you click on the continue button, (or F5 key) simulation will execute up to that line (yellow arrow) and halt.

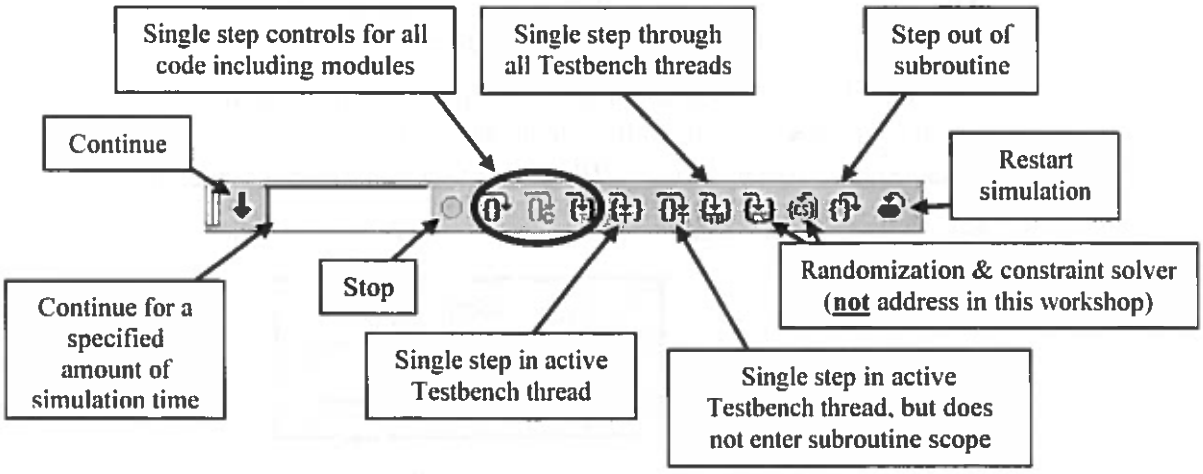
The clear circle is a disabled break point. It will have no effect when the continue button is clicked. It's a convenient reminder of where a break point can/should be for that debugging session.

No circle indicates break point is not set for that line.

5. Set a break point at the **`run_test()`** statement, then click on continue button

   You are now at the beginning of the UVM test execution. The next thing to try is the single-step mechanism.

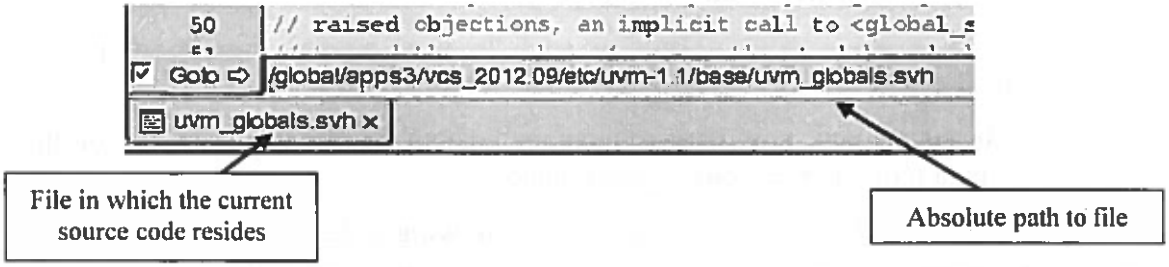
The source code simulation control buttons are group together:



6. Click on the {} button to single step through all threads in testbench

   You are now in **`run_test()`** subroutine scope.

7. Look at the bottom of the source code window, you should be able to see what file the source code is in and where to locate the file:

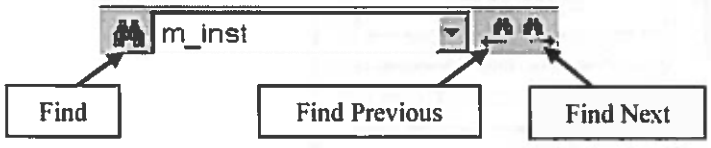


   The reason that **`run_test()`** subroutine is in the **`uvm_globals.svh`** file is because it is an independent declaration not inside any class. Thus, when the UVM package is imported into the **`program`** block, **`run_test()`** is a global subroutine within the **`program`** block scope.

8. Continue by clicking on {} to enter the **`uvm_root::get()`** scope

   The first line in the **`get()`** subroutine checks to see if **`m_inst`** is null. What is **`m_inst`**? Let's find out.

9. Locate the Find button (next to the simulation control buttons) and enter **`m_inst`** in the text field, then click on Find Previous button.



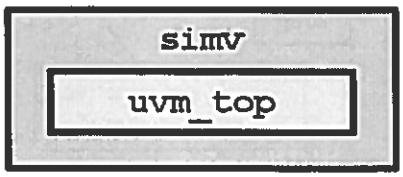
   You should see that **`m_inst`** is a **`static uvm_root`** handle. The find mechanism combined with the file location can be very helpful for you to quickly pin point where to examine and correct code when necessary.

10. Click on {} again, to go to next step.

    You will see that the source code execution has advanced to the return statement. Meaning that **m_inst** had already been constructed.

    How can this be? Look at the source code from the **m_inst** declaration down to the **get()** subroutine declaration. Do you see another **uvm_root::get()** call? When will this other **get()** execute?

    Recall from lecture that in the beginning of UVM simulation, the following structure is created:



    Do you see that **uvm_top** is created at the beginning of simulation via the **get()** subroutine?

    What about **uvm_test_top**? How is that created. Continue to find out.

11. Click on {} to leave the subroutine and go back to the caller scope

    What is top handle pointing to now?

12. Click on {} a couple of times to enter the **uvm_root**'s **run_test()** method

    The source code now starts to get complicated. So, we will focus just on the creation of the **uvm_test_top** object

13. Click on {} a few times to get to the following statement:

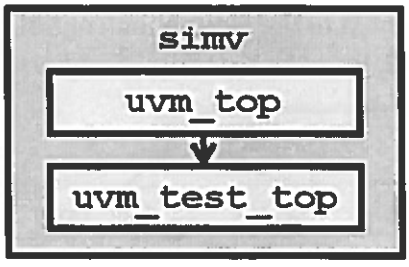
```
test_name_count = clp.get_arg_values("+UVM_TESTNAME=",test_name);
```

    Here is where your **+UVM_TESTNAME** run-time option is read and stored as the **test_name**.

14. Click on {} until you get to the following statement:

```
$cast(uvm_test_top, factory.create_component_by_name(test_name,
                                    "", "uvm_test_top", null));
```

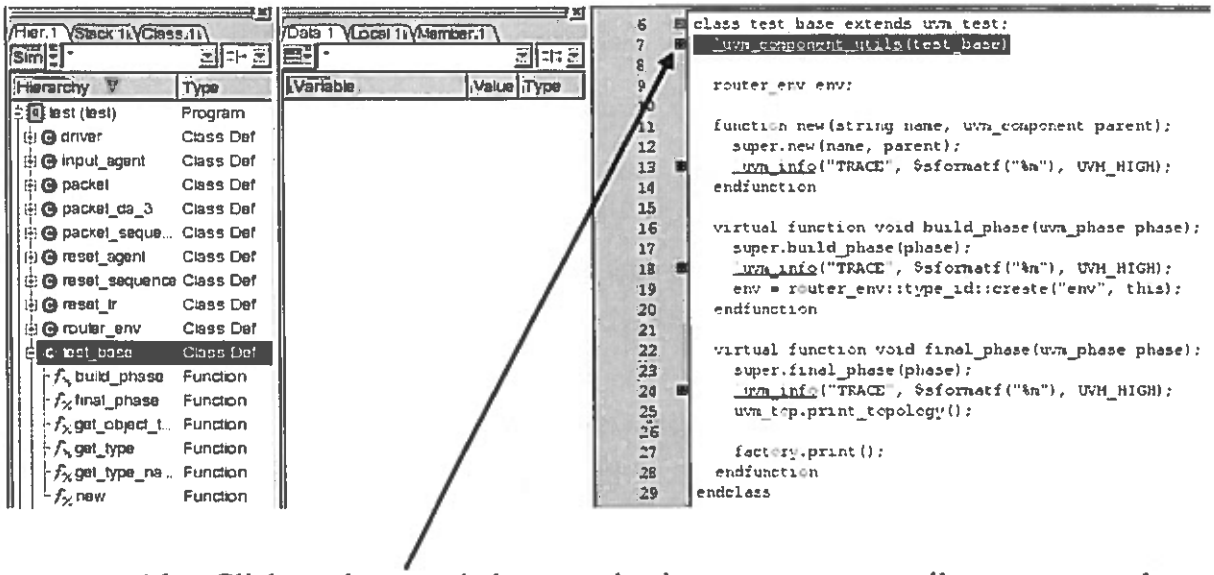
    At this point, you have the full initial structure!



Component with parent handle set to null is at the top of the test hierarchy

    You can step through the rest of the base class code if desired. However, you will find it overwhelming. So, you will better off to go to your own source code and see what's happening.

15. From the Hierarchy pane, locate and double click on **test_base** to see the source code:



16. Click on the + symbol to see what `uvm_component_utils macro expands to

The most important thing to recognize is that the macro creates a proxy type, **type_id**, which is used in UVM to create objects of the class.

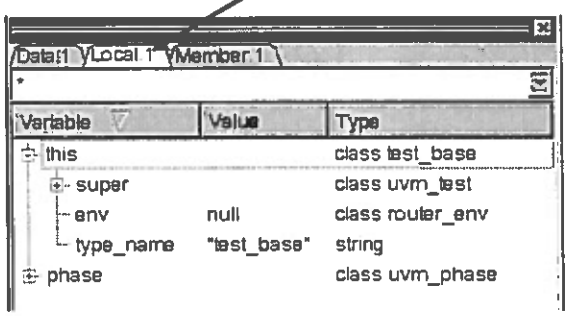
```
6  class test_base extends uvm_test;
7    `uvm_component_utils(test_base)
1    typedef uvm_component_registry #(test_base,"test_base") type_id;
2      static function type_id get_type();
3        return type_id::get();
4      endfunction
5      virtual function uvm_object_wrapper get_object_type();
6        return type_id::get();
7      endfunction
8      const static string type_name = "test_base";
9      virtual function string get_type_name ();
10       return type_name;
11     endfunction
```

The rest of the lab is left up to you for experimenting with DVE debugging.

Just a few more notes:

If you want to restart the simulation, just click on the Restart button: 

If you want to see the value of local properties of the scope that the source code pane is in, look under the Local tab:



You have completed the DVE interactive debugging session!

Answers / Solutions

test_collection.sv Solution:

```
class test_base extends uvm_test; ...

class test_da_3_seq extends test_base;
  `uvm_component_utils(test_da_3_seq)

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    uvm_config_db#(bit[15:0])::set(this,
"env.i_agent.seqr.packet_sequence", "da_enable", 16'h0008);
    uvm_config_db#(int)::set(this, "env.i_agent.seqr.packet_sequence",
"item_count", 20);
  endfunction
endclass
```

driver.sv Solution:

```
class driver extends uvm_driver #(packet);
  `uvm_component_utils(driver)

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual task run_phase(uvm_phase phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    forever begin
      seq_item_port.get_next_item(req);
      `uvm_info("DRV_RUN", {"\n", req.sprint()}, UVM_MEDIUM);
      seq_item_port.item_done();
    end
  endtask
endclass
```

packet_sequence.sv Solution:

```
class packet_sequence extends uvm_sequence_base;
  int       item_count = 10;
  int       port_id = -1;
  bit[15:0] da_enable  = '1;
  int       valid_da[$];

  `uvm_object_utils_begin(packet_sequence)
    `uvm_field_int(item_count, UVM_ALL_ON)
    `uvm_field_int(port_id, UVM_ALL_ON)
    `uvm_field_int(da_enable, UVM_ALL_ON)
    `uvm_field_queue_int(valid_da, UVM_ALL_ON)
  `uvm_object_utils_end

  task pre_start();
    super.pre_start();
    begin
      uvm_sequencer_base my_seqr = get_sequencer();
      uvm_config_db#(int)::get(null, get_full_name(), "item_count",
item_count);
      uvm_config_db#(bit[15:0])::get(null, get_full_name(), "da_enable",
da_enable);
      uvm_config_db#(int)::get(my_seqr.get_parent(), "",
"port_id",port_id);
      if (!(port_id inside {-1, [0:15]})) begin
        `uvm_fatal("CFGERR", $sformatf("Illegal port_id value of %0d",
port_id));
      end
      valid_da.delete();
      for (int i=0; i<16; i++) begin
        if (da_enable[i]) begin
          valid_da.push_back(i);
        end
      end
    end
  endtask

  function new(string name = "packet_sequence");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  task body();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

    repeat(item_count) begin
      `uvm_do_with(req, {if (port_id == -1) sa inside {[0:15]}; else sa
== port_id; da inside valid_da;});
    end

  endtask

endclass
```

reset_sequence.sv Solution:

```
class reset_sequence extends uvm_sequence#(reset_tr);
  `uvm_object_utils(reset_sequence)
  function new(string name = "reset_sequence");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction
  task pre_start();
   `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if ((get_parent_sequence() == null) && (starting_phase != null)) begin
      starting_phase.raise_objection(this);
    end
  endtask
  task body();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    `uvm_info("RESET", "Executing Reset", UVM_MEDIUM);
  endtask
  task post_start();
   `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if ((get_parent_sequence() == null) && (starting_phase != null)) begin
      starting_phase.drop_objection(this);
    end
  endtask
endclass
```

router_env.sv Solution:

```
class router_env extends uvm_env;
  input_agent i_agent;
  reset_agent r_agent;
  `uvm_component_utils(router_env)
  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    i_agent = input_agent::type_id::create("i_agent", this);
    uvm_config_db #(uvm_object_wrapper)::set(this, "i_agent.seqr.main_phase",
"default_sequence", packet_sequence::get_type());

    r_agent = reset_agent::type_id::create("r_agent", this);
    uvm_config_db #(uvm_object_wrapper)::set(this, "r_agent.seqr.reset_phase",
"default_sequence", reset_sequence::get_type());
  endfunction
endclass
```

4

Configure Drivers, Sequences, and Environment

Learning Objectives

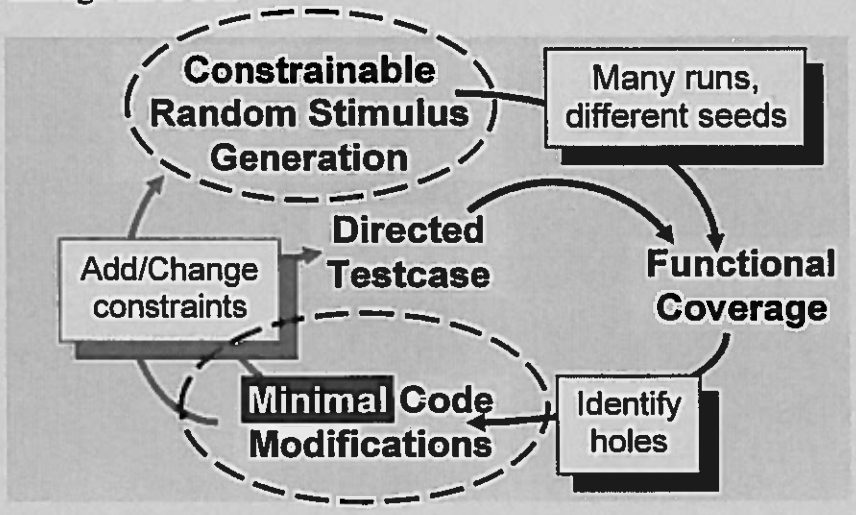
After completing this lab, you should be able to:

- Add DUT virtual interfaces to driver
- Add configuration fields to driver
- Add physical device drivers to driver
- Add DUT virtual interfaces to reset sequence
- Add physical device drivers to reset sequence
- Compile and simulate

Lab Duration: 60 minutes

Getting Started

In Lab 3, through embedding of configuration fields, you enabled minimal code modification and constrainable random stimulus generation capability. In this lab, you will add physical fields to the driver and reset sequence to actively process the stimulus through the DUT.



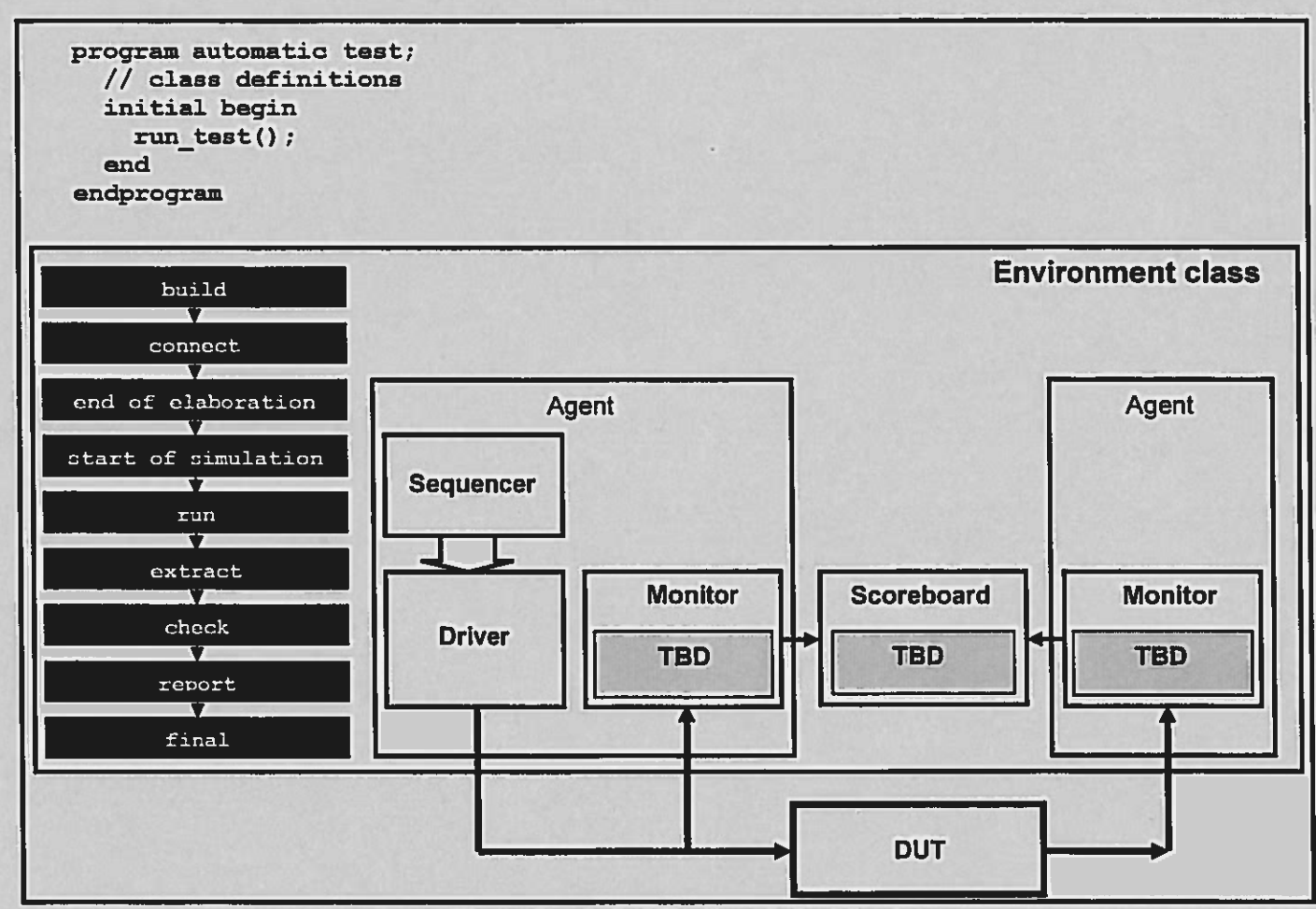


Figure 1. Lab 4 Testbench Architecture

Lab Overview

The work flow for this lab is illustrated as follows:

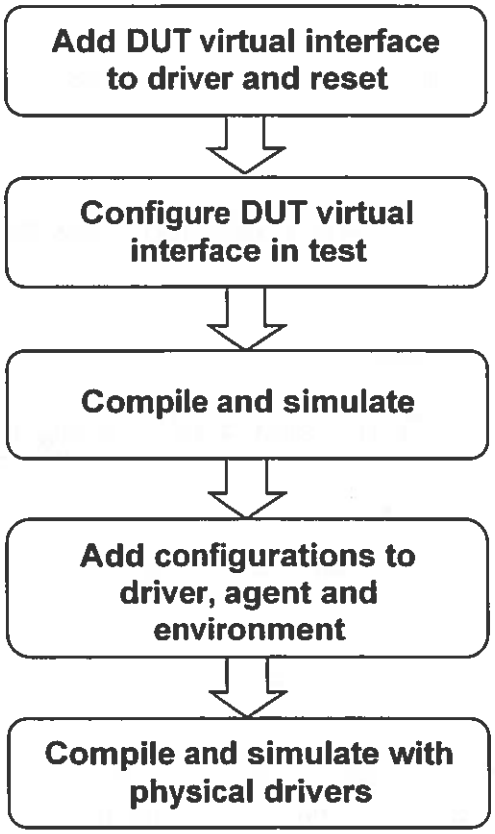


Figure 2. Lab 4 Flow Diagram

Implement Configurable Environment

Up to this point, you have not driven any test stimulus through the DUT. In order to do this, you need to embed and configure DUT virtual interfaces in physical drivers and implement the device drivers.

In this lab, you will add the DUT virtual interface and other configuration fields to the driver, reset sequence, agent and environment. The device drivers will be written for you.

Task 1. Go into lab4 Working Directory

1. CD into the lab4 directory

```
> cd ../lab4
```

Task 2. Add Interface and Configuration to Driver

1. Open **driver.sv** file in an editor
2. Create a DUT virtual interface (**router_tb_io**) handle, call it **sigs**:

```
class driver extends uvm_driver #(packet);
  virtual router_tb_io sigs;
```

3. Add the ability to designate the driver to only drive a chosen port with an **int** property called **port_id** with the default value of **-1**:

```
  int port_id = -1;
```

This **port_id** is meant to configure the driver to only drive packets of matching source address (**sa**). If the incoming packet's **sa** does not match the driver's **port_id**, that packet will be dropped.

If **port_id** is not set (**-1**), the driver will accept and drive all incoming packets. For this lab, you will leave the **port_id** at the default value of -1. In the next lab, you will make use of the **port_id**.

4. Add the **port_id** to **uvm_component_utils**:

```
  `uvm_component_utils_begin(driver)
    `uvm_field_int(port_id, UVM_DEFAULT | UVM_DEC)
  `uvm_component_utils_end
```

Add _begin to macro

Notice that the virtual interface handle (**sigs**) is not added to the **uvm_component_utils** macro list. This is because there is no support in the **uvm_component_utils** macro to accommodate virtual interfaces.

5. Add the following code to retrieve and check the configuration properties in build phase:

```
virtual function void build_phase(uvm_phase phase);
 super.build_phase(phase);
 `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
 uvm_config_db#(int)::get(this, "", "port_id", port_id);
 if (!(port_id inside {-1, [0:15]})) begin
  `uvm_fatal("CFGERR",$sformatf("port_id must be {-1, [0:15]}, not %0d!",port_id));
 end
 uvm_config_db#(virtual router_tb_io)::get(this, "", "sigs", sigs);
 if (sigs == null) begin
  `uvm_fatal("CFGERR", "Interface for Driver not set");
 end
endfunction
```

Get **port_id**

Retrieve DUT virtual interface and store handle in **sigs**

6. Locate the **run_phase()** method and add the following:

- Check **port_id** to see if the driver should accept or drop the packet. If **port_id** is **-1**, or if **port_id** matches **req** object's vi dri**sa** field, call **send()** method to drive the content of the **req** object through the DUT. Otherwise, drop the **req** object without processing.

Drive packet only if **port_id** matches **req.sa** or is **-1**

```
if (port_id inside { -1, req.sa }) begin
   send(req);
   `uvm_info("DRV_RUN", {"\n", req.sprint()}, UVM_MEDIUM);
 end
 seq_item_port.item_done();
end
endtask
```

To save lab time, the **start_of_simulation_phase()** method and the physical device driver methods are done for you.

7. Save and close the file

Task 3. Agent Configuration

The agent contains the sequencer, driver and monitor. The agent should be treated by the higher layer structural components (environement, test) as a black box. This means that the configuration of the sequencer, driver and monitors by the higher structural components should target the agent, not the individual components in the agent. The agent would then configure its own sub-components.

Since the configuration calls are the same as what you have already done, there is no learning point in going through a typing exercise to enter the code in the agent. All the configuration code have been done for you in **input_agent.sv**.

Take a look at the code if you are interested. Otherwise, continue to the next task.

Task 4. Compile and Simulate

1. Compile and simulate the testbench

```
> make
```

You should see the following error:

```
UVM_INFO @ 0.0ns: reporter [RNTST] Running test test_base...
UVM_FATAL driver.sv(77) @ 0.0ns: uvm_test_top.env.i_agent.drv
[CFGERR] Interface for Driver not set
--- UVM Report Summary ---
** Report counts by severity
UVM_INFO :    1
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    1
```

A fatal error occurred!
This is because the virtual interface was never set!

Even though the DUT virtual interface was added in the driver, you have not yet configured it in the test. This is a fatal error that you must correct.

Task 5. Add Virtual Interface Configuration in Test

1. Open **test_collection.sv** file in an editor
2. Locate the **build_phase** method in **test_base** class
3. Configure each agent's virtual interface in the environment as follows:

```
virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  env = router_env::type_id::create("env", this);
  uvm_config_db#(virtual router_tb_io)::set(this, "env.i_agent",
                 "sigs", router_test_top.router_if.agent_vif);
  uvm_config_db#(virtual reset_tb_io)::set(this, "env.r_agent",
                  "sigs", router_test_top.reset_if.agent_vif);
endfunction
```

Access the DUT interface with XMR

Note: The testbench harness file is locate in **~/rtl** directory. If you are interested, take a look at the **router_test_top.sv** file

4. Save and close the file

Task 6. Compile And Debug The Program

1. Compile and simulate the testbench.

```
> make
```

How many packets did the driver process? (Hint: **DRV_RUN** count)

But, did these packets propagate correctly through the DUT? Check it in the DVE waveform window.

2. Open DVE with the following command:

```
> make dve
```

You should see that all output values are red (unknown)! This is because the DUT needs to be reset before it can successfully process inputs.

3. Exit DVE

Task 7. Update the Reset Sequence

In lab 3, you configured a **reset_sequence** to execute at the reset phase. However, that **reset_sequence** code didn't do anything. In this task, you will update the **reset_sequence** to perform a DUT reset.

1. Open the **reset_sequence.sv** file in an editor
2. In **body()** task
   - De-assert reset for 2 cycles
   - Then, assert reset for 1 cycle
   - Followed by de-assert for 15 cycles

```
task body();
  `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

  `uvm_do_with(req, {kind == DEASSERT; cycles == 2;});
  `uvm_do_with(req, {kind == ASSERT; cycles == 1;});
  `uvm_do_with(req, {kind == DEASSERT; cycles == 15;});
 endtask
```

3. Save and close the file

Task 8. Compile And Simulate with Reset Sequence

1. Compile and simulate the testbench.

```
> make
```

Did it work? Check it in the DVE waveform window.

2. Open DVE with the following command:

```
> make dve
```

You should now see that the output signals no longer are red. However, the base test was designed to send 10 packets through the DUT. In the waveform window, this is not what you are seeing. There is still a problem.

3. Exit DVE

Task 9. Control Signal De-Assertion Sequence

With the reset agent and reset sequence, the only thing that happened was the assertion and de-assertion of the reset signal. How about the other control signals of the interfaces? The reset agent and the reset sequence should not be responsible for taking care of all signals of the DUT. Doing so will complicate the development of individual tests and integration.

There are two ways to resolve this problem. One, create and configure a sequence to de-assert the the control signals. Or, two, de-assert the control signals within the driver's **`reset_phase()`** method.

For flexible controls of reset priorities, you should de-assert the control signals with sequences.

A sequence to de-assert the control signals the driver objects are responsible for driving has been done for you in **`driver_reset_sequence.sv`**.

Take a look at the code if you are interested. Otherwise, configure this sequence to be executed at the reset phase.

1. Open the **`router_env.sv`** file in an editor
2. Bring in the **`driver_reset_sequence.sv`** file with **`` `include ``**
3. Configure the input agent's (**`i_agent`**) sequencer (**`seqr`**) to execute the **`driver_reset_sequence`** at the **`reset_phase`**.
4. Save and close the file

Task 10. Compile & Simulate with Driver Reset Phases

1. Compile and simulate the testbench.

   ```
   > make
   ```

2. Open DVE with the following command:

   ```
   > make dve
   ```

   You should now see the correct behavior: 10 packets came out of the DUT.

3. Try and run the destination address is 3 and 20 packets test:

   ```
   > make test=test_da_3_seq
   ```

4. Take a look at DVE again

   You should see that 20 packets came out of output port 3.

Congratulations, you have completed the regular portion of Lab 4!

Task 11. Optional: UVM Transaction Debugging

There are times when you need to trace the problem in simulation by conducting an interactive source code debugging session. The following steps will take you through a simple example.

1. Take a look at the run-time command:

```
> make -n run
```

You should see the following:

```
./simv -l simv.log +ntb_random_seed=1 +UVM_TESTNAME=test_base \
       +UVM_VERBOSITY=UVM_MEDIUM +UVM_TR_RECORD +UVM_LOG_RECORD
```

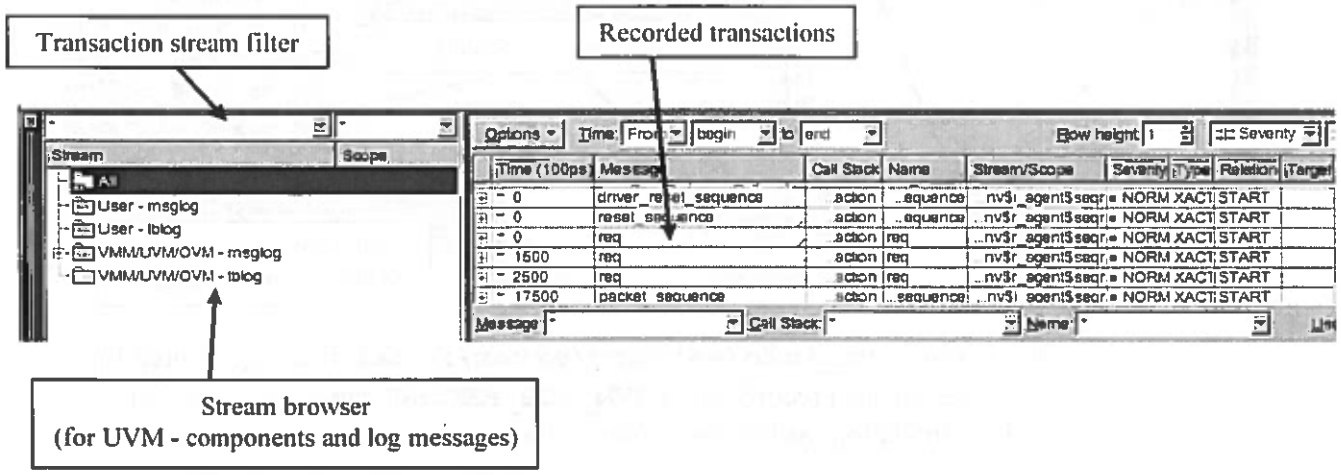
Enables UVM transaction recording

Enables UVM LOG message recording

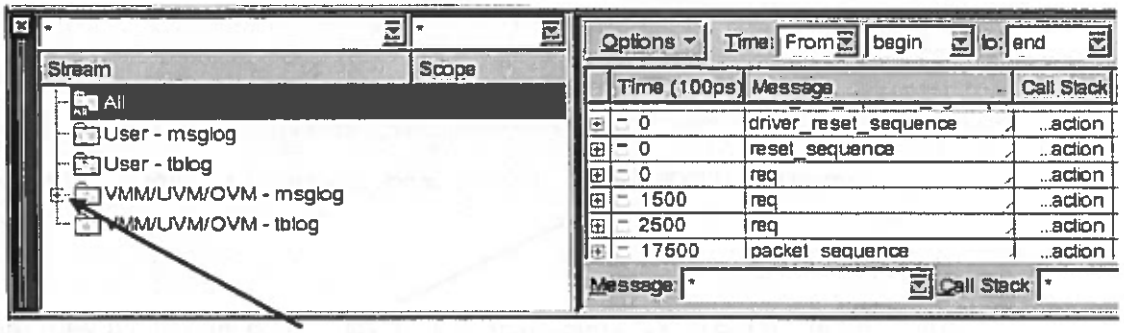
2. Start DVE in post-processing mode with transaction debugging:

```
> make dve_tr
```

You should see the following panes in the lower portion of the DVE window:



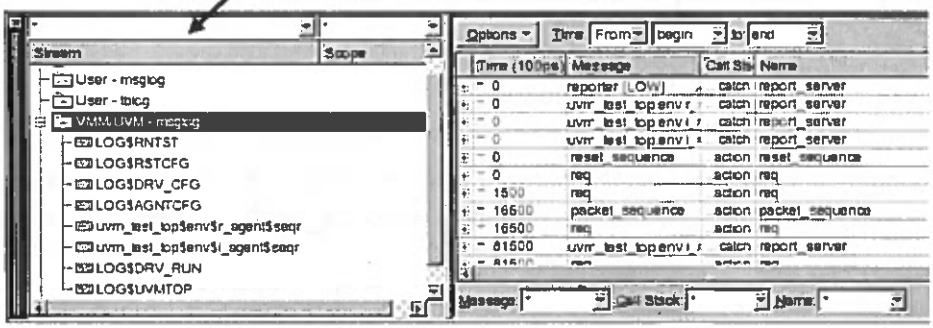
For UVM, the automatically recorded streams are the UVM sequence item transactions and the UVM log messages. For this lab, you will only be looking at the **VMM/UVM - mslog** folder.



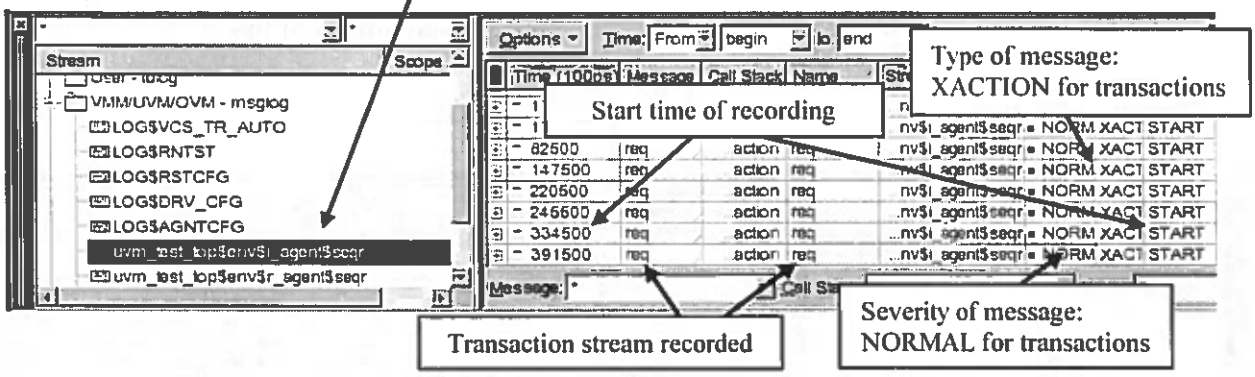
3. Click on the + symbol of the **VMM/UVM - msglog** folder to expand it

In the expanded **VMM/UVM - mslog** folder, you should see that recorded streams are displayed. There are two typical streams: sequencer transaction streams and UVM log (message id) streams. Each stream may contain a series of transaction over time associated with the source of the stream.

Note: Use the filter text field to narrow down what get displayed in the stream browser window if there are too many streams.



4. Click on **uvm_test_topenvi_agent$seqr** to see the **sequence** and **sequence_items** transaction stream that the sequencer generated

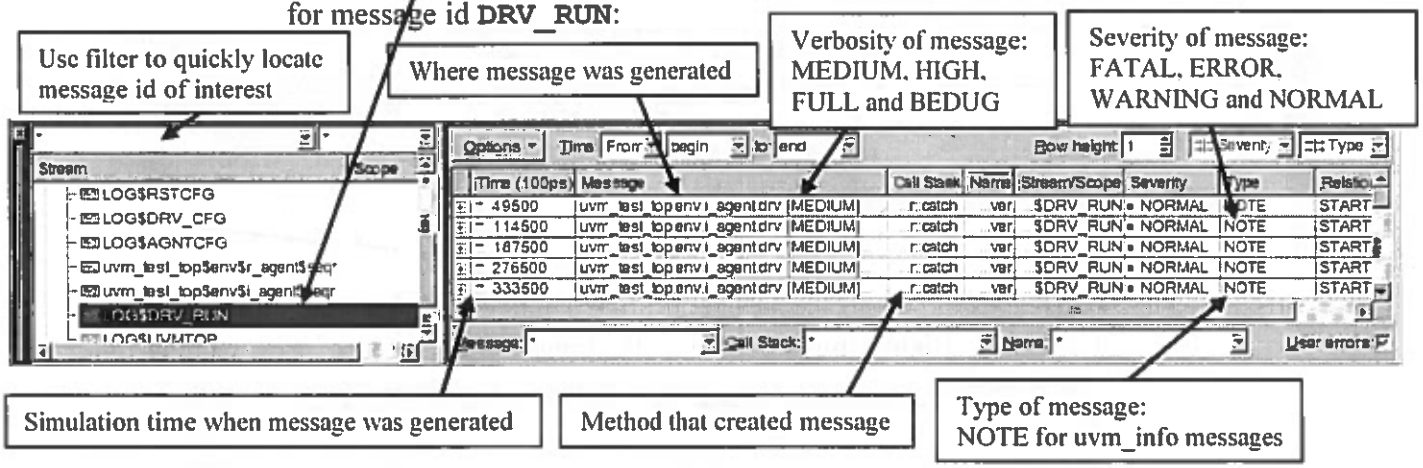


The UVM **`uvm_info/warning/error/fatal** macro generated log messages are also recorded if **+UVM_LOG_RECORD** run-time is applied during simulation (as has been done in the lab).

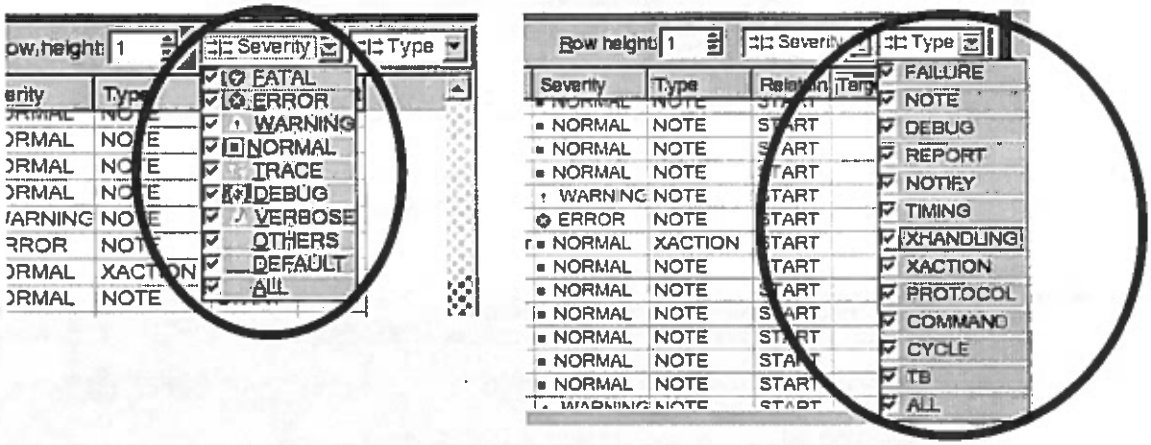
```
class driver extends uvm_driver #(packet);
  // other code not shown
  virtual task run_phase(uvm_phase phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    forever begin
      seq_item_port.get_next_item(req);
      if (port_id inside { -1, req.sa }) begin
        send(req);
        `uvm_info("DRV_RUN", {"\n", req.sprint()}, UVM_MEDIUM);
      end
...
```

For example, in the driver code, there is a **`uvm_info** embedded with the message id of **DRV_RUN**. This will show up in the browser window under **LOG$DRV_RUN**.

5. Click on **LOG$DRV_RUN** in the browser window to see all the log messages for message id **DRV_RUN**:

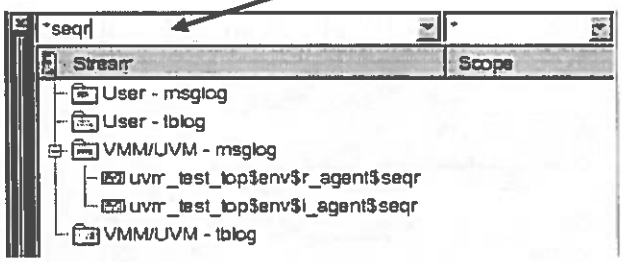


At the upper right hand corner of the transaction pane, there are two drop-down tabs for severity and type. Clicking on them will enable you to display only the severities and types of the messages of interest:



Once you located the transaction and UVM log messages, you would typically want to see the transaction/log message on the Wave window. One quick way of finding a stream is to make use of the stream filter.

6. In the stream filter text field enter ***seqr**

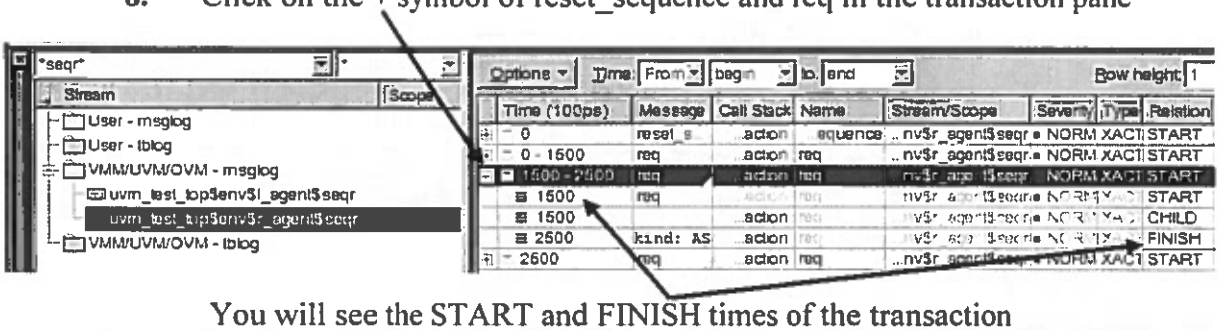


As you type, you should notice that the stream source browser window is updated dynamically to show you the result of the filter.

7. Click on **uvm_test_topenvr_agent$seqr**

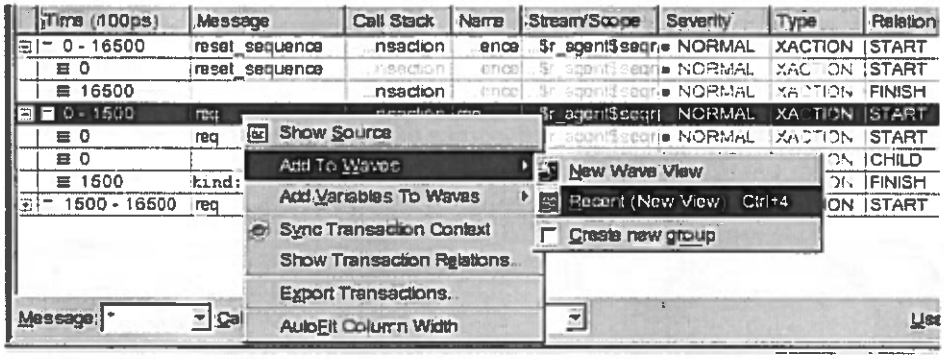
This will display the **reset_sequence** and reset_tr object **req**.

8. Click on the + symbol of reset_sequence and req in the transaction pane

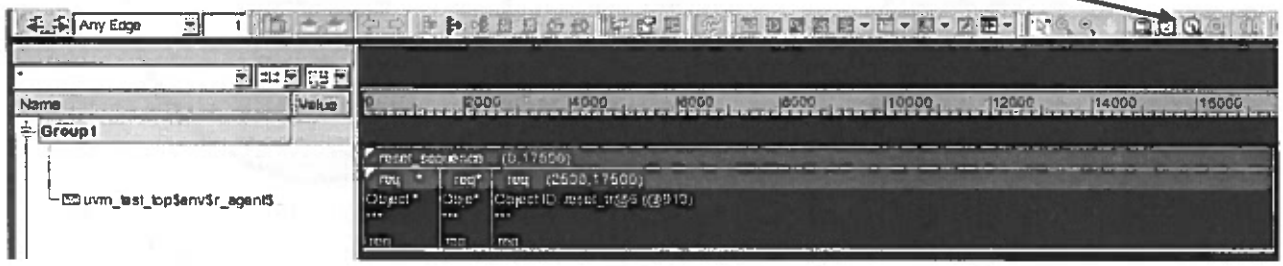


You will see the START and FINISH times of the transaction

9. Click on the Right Mouse Button on the transaction, add it to Wave window

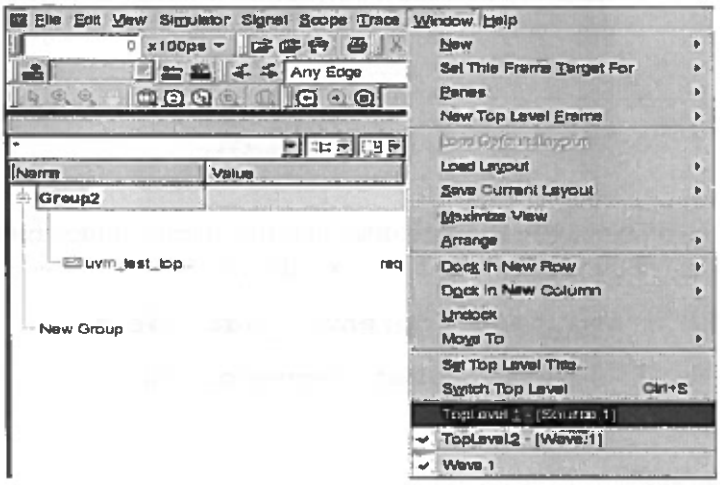


10. In the Wave window, click on the zoom out button a few time to see the reset_sequence fully displayed at the start time and clear at the finish time.

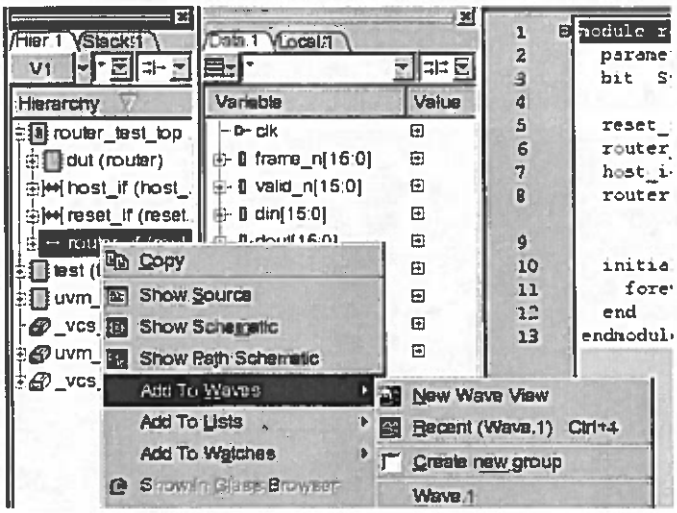


With only transaction displayed, it's hard to tell what's going on. You need to see the DUT signals to start to be able make sense of things.

11. Click on Window -> TopLevel.1 to go back to the Hierarchy pane

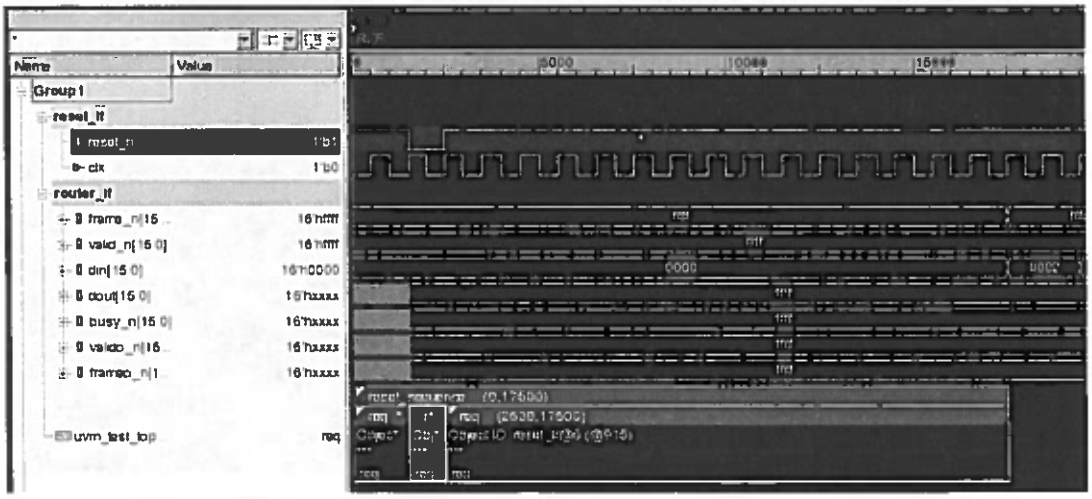


12. In the Herarchy pane, expand router_test_top, right-click on **`router_if`** and add it to the recent Wave window:

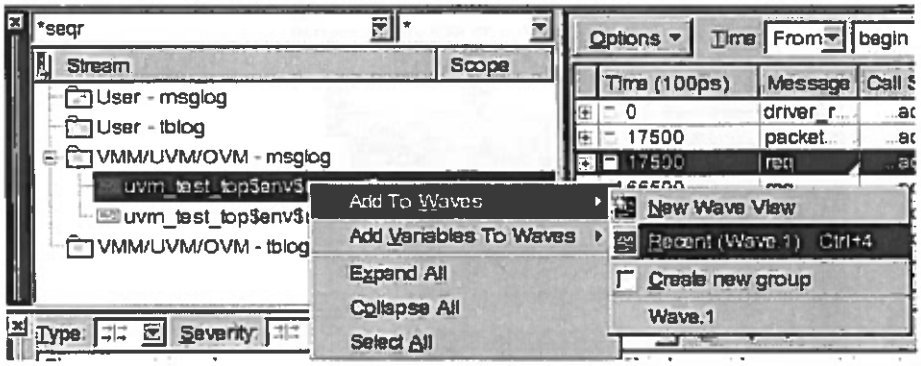


13. Do the same for **`reset_if`**

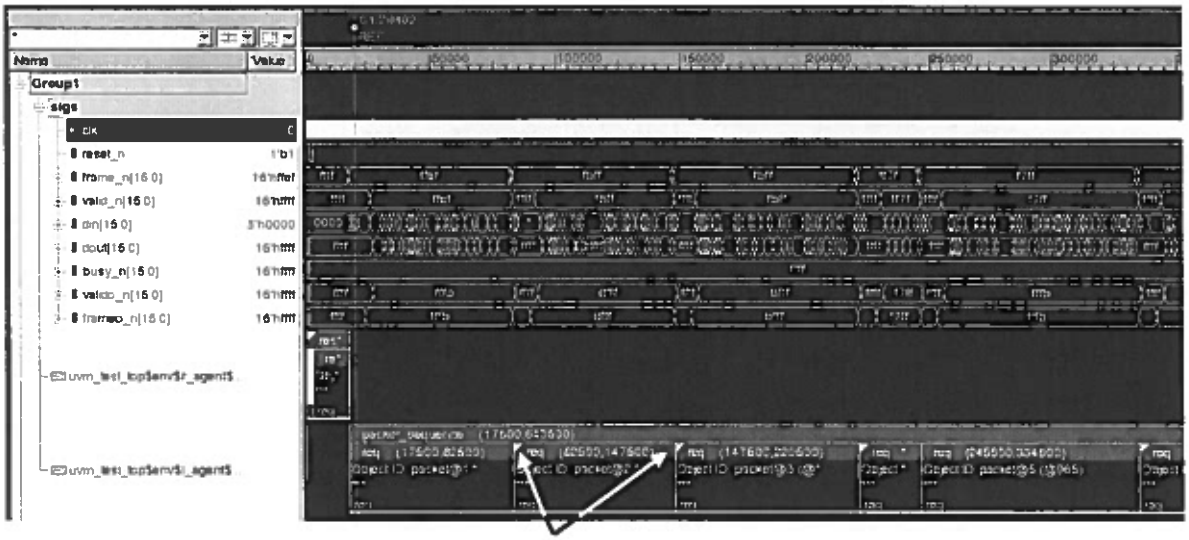
14. Go back to the Wave window, you should be able to see the reset_sequence timing with respect to the actual DUT interface signals (delete the signals that you don't to see. E.g. $unit and the extra clock)



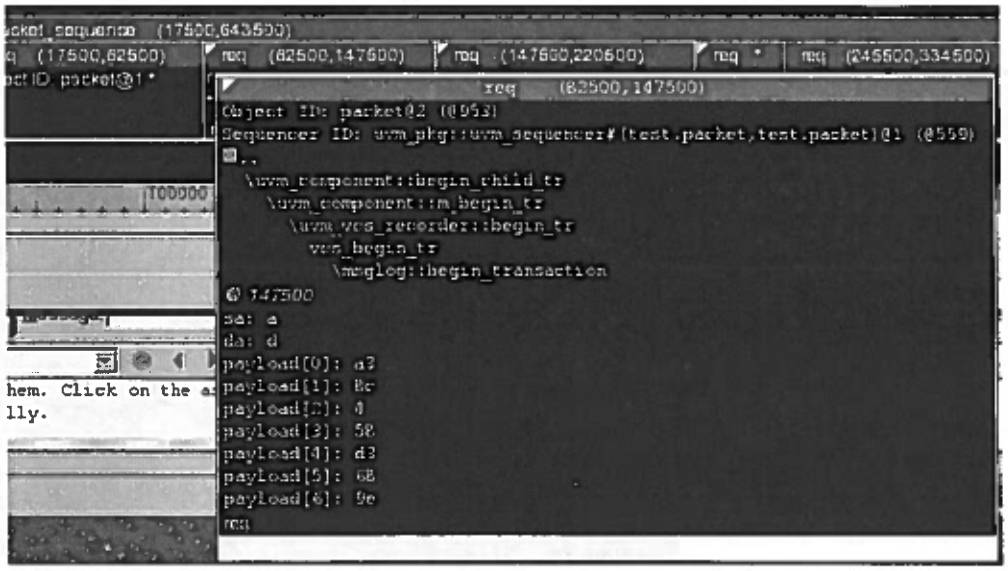
15. Go back to DVE TopLevel. This time, in Stream browser pane, add **`uvm_test_top$env$i_agent$seqr`** to the Wave window.



You should see **packet_sequence** and its associated **sequence_items** displayed. (You may need to zoom out)



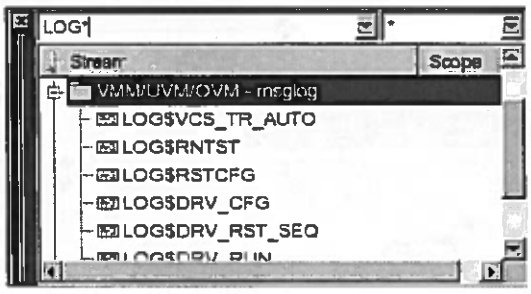
16. Place the cursor over the triangle table for each transaction to see the content of that transaction



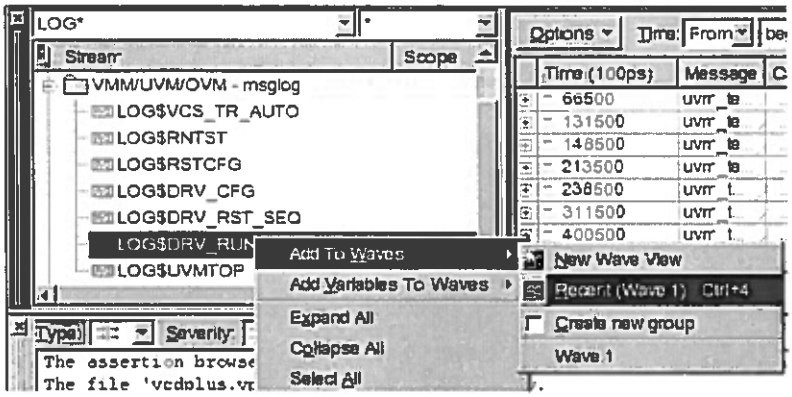
For the UVM log messages, you can do the same thing.

17. In the Stream browser pane, change to filter to LOG*

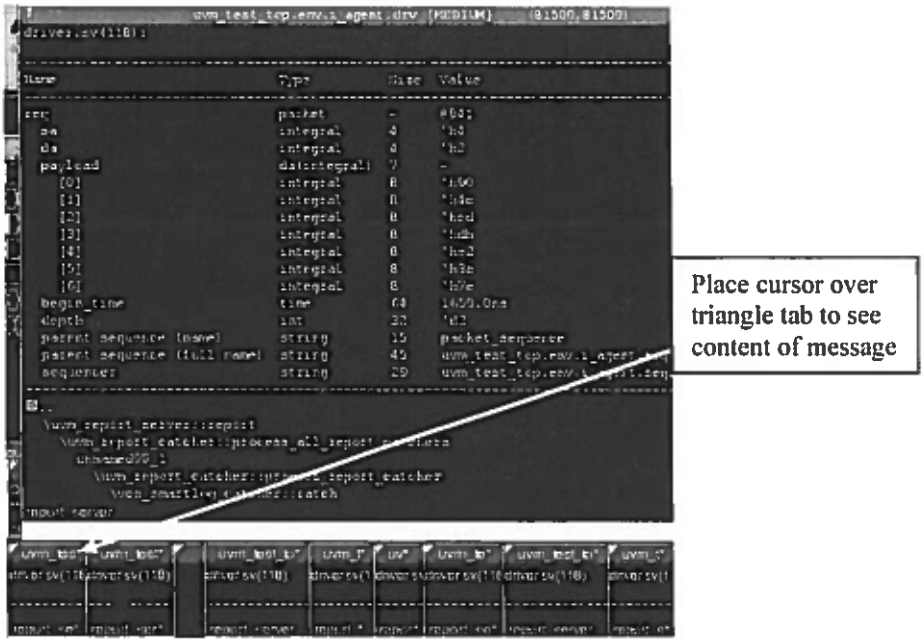
You will now see only the log message streams.



18. Right-click on LOG$DRV_RUN and add the stream to the Wave window



Once again, you can place the cursor over the triangle tab to see the content of the message. One thing you should notice, UVM log messages do not have start and finish time. Yet, in the display, the UVM log messages span over a period of time. This is not entirely accurate, but is done for visualization. The simulation time at which the UVM log message is generated is treated by DVE as the start time. The UVM log message then persists until the next UVM log message of that stream is generated. So, in DVE, once the UVM log message stream starts, you will never see a gap in between the messages.



The knowledge that you gained in this exercise may help you to debug your code in future labs. The rest of the lab is left for you to do your own exploration.

For more information on the DVE debugger, consult the user guide for DVE. (see $VCS_HOME/doc/UserGuide/pdf/dve_ug.pdf)

Congratulations, you have completed the optional portion of Lab 4!

Answers / Solutions

test_collection.sv Solution:

```
class test_base extends uvm_test;
  `uvm_component_utils(test_base)

  router_env env;

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    env = router_env::type_id::create("env", this);
    uvm_config_db#(virtual router_tb_io)::set(this, "env.i_agent",
"sigs", router_test_top.router_if.agent_vif);
    uvm_config_db#(virtual reset_tb_io)::set(this, "env.r_agent", "sigs",
router_test_top.reset_if.agent_vif);
  endfunction

  virtual function void final_phase(uvm_phase phase);
    super.final_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    uvm_top.print_topology();
    factory.print();
  endfunction
endclass
```

router_env.sv Solution:

```
class router_env extends uvm_env;
  input_agent i_agent;
  reset_agent r_agent;

  `uvm_component_utils(router_env)

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

    i_agent = input_agent::type_id::create("i_agent", this);

    uvm_config_db #(uvm_object_wrapper)::set(this,
"i_agent.seqr.reset_phase", "default_sequence",
driver_reset_sequence::get_type());

    uvm_config_db #(uvm_object_wrapper)::set(this,
"i_agent.seqr.main_phase", "default_sequence",
packet_sequence::get_type());

    r_agent = reset_agent::type_id::create("r_agent", this);
    uvm_config_db #(uvm_object_wrapper)::set(this,
"r_agent.seqr.reset_phase", "default_sequence",
reset_sequence::get_type());
  endfunction
endclass
```

driver.sv Solution:

```
class driver extends uvm_driver #(packet);
  virtual router_io sigs;          // DUT virtual interface
  int               port_id = -1;  // Driver's designated port

  `uvm_component_utils_begin(driver)
    `uvm_field_int(port_id, UVM_DEFAULT | UVM_DEC)
  `uvm_component_utils_end

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction: new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    uvm_config_db#(int)::get(this, "", "port_id", port_id);
    if (!(port_id inside {-1, [0:15]})) begin
      `uvm_fatal("CFGERR", $sformatf("port_id must be {-1, [0:15]}, not
%0d!", port_id));
    end
    uvm_config_db#(virtual router_io)::get(this, "", "sigs", sigs);
    if (sigs == null) begin
      `uvm_fatal("CFGERR", "Interface for Driver not set");
    end
  endfunction: build_phase

  virtual function void start_of_simulation_phase(uvm_phase phase);
    super.start_of_simulation_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    `uvm_info("DRV_CFG", $sformatf("port_id is: %0d", port_id),
UVM_MEDIUM);
  endfunction: start_of_simulation_phase

  virtual task run_phase(uvm_phase phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

    forever begin
      seq_item_port.get_next_item(req);
      if (port_id inside { -1, req.sa }) begin
        send(req);
        `uvm_info("DRV_RUN", {"\n", req.sprint()}, UVM_MEDIUM);
      end
      seq_item_port.item_done();
    end
  endtask: run_phase

  //
  // See file for device drivers
  //

endclass: driver
```

driver_reset_sequence.sv Solution:

```
class driver_reset_sequence extends uvm_sequence #(packet);
  virtual router_tb_io sigs;          // DUT virtual interface
  int                  port_id = -1;  // Driver's designated port

  `uvm_object_utils_begin(driver_reset_sequence)
    `uvm_field_int(port_id, UVM_DEFAULT | UVM_DEC)
  `uvm_component_utils_end

  function new(string name="driver_reset_sequence");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction: new

  virtual task pre_start();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if ((get_parent_sequence() == null) && (starting_phase != null)) begin
      starting_phase.raise_objection(this);
    end
    uvm_config_db#(int)::get(get_sequencer(), "", "port_id", port_id);
    if (!(port_id inside {-1, [0:15]})) begin
      `uvm_fatal("CFGERR", $sformatf("port_id must be {-1, [0:15]}, not
%0d!", port_id));
    end
    uvm_config_db#(virtual router_tb_io)::get(get_sequencer(), "", "sigs",
sigs);
    if (sigs == null) begin
      `uvm_fatal("CFGERR", "Interface for the Driver Reset Sequence not
set");
    end
  endtask: pre_start

  virtual task post_start();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if ((get_parent_sequence() == null) && (starting_phase != null)) begin
      starting_phase.drop_objection(this);
    end
  endtask: post_start

  virtual task body();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if (port_id == -1) begin
      sigs.frame_n = '1;
      sigs.valid_n = '1;
      sigs.din = '0;
    end else begin
      sigs.frame_n[port_id] = '1;
      sigs.valid_n[port_id] = '1;
      sigs.din[port_id] = '0;
    end
  endtask: body
endclass: driver_reset_sequence
```

reset_sequence.sv Solution:

```
class reset_tr extends uvm_sequence_item;
  typedef enum {ASSERT, DEASSERT} kind_e;
  rand kind_e kind;
  rand int unsigned cycles = 1;

  `uvm_object_utils_begin(reset_tr)
    `uvm_field_enum(kind_e, kind, UVM_ALL_ON)
    `uvm_field_int(cycles, UVM_ALL_ON)
  `uvm_object_utils_end

  function new(string name = "reset_tr");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction: new
endclass

class reset_sequence extends uvm_sequence #(reset_tr);
  `uvm_object_utils(reset_sequence)

  function new(string name = "reset_sequence");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual task pre_start();
    if ((get_parent_sequence() == null) && (starting_phase != null)) begin
      starting_phase.raise_objection(this);
    end
  endtask

  virtual task body();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

    `uvm_do_with(req, {kind == DEASSERT; cycles == 2;});
    `uvm_do_with(req, {kind == ASSERT; cycles == 1;});
    `uvm_do_with(req, {kind == DEASSERT; cycles == 15;});
  endtask

  virtual task post_start();
    if ((get_parent_sequence() == null) && (starting_phase != null)) begin
      starting_phase.drop_objection(this);
    end
  endtask
endclass
```

5 Monitors and Scoreboard

Learning Objectives

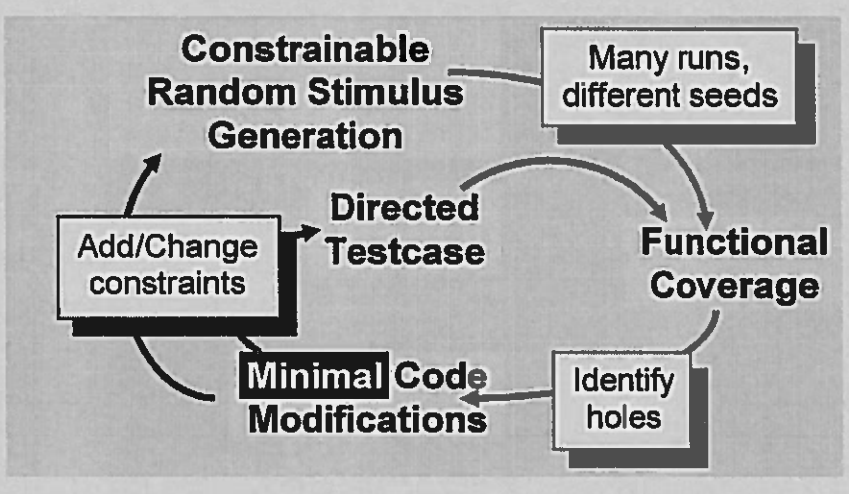
After completing this lab, you should be able to:

- Develop a bus monitor with TLM analysis port
- Add the bus monitor to agent
- Implement a scoreboard using uvm_in_order_class_comparator
- Add scoreboard and an array of agents to the environment
- Compile and simulate

Lab Duration: 60 minutes

Getting Started

Through Lab 4, you have begun to drive the stimulus sequence through the DUT. You now need to add monitors and scoreboards into the environment to enable self-check.



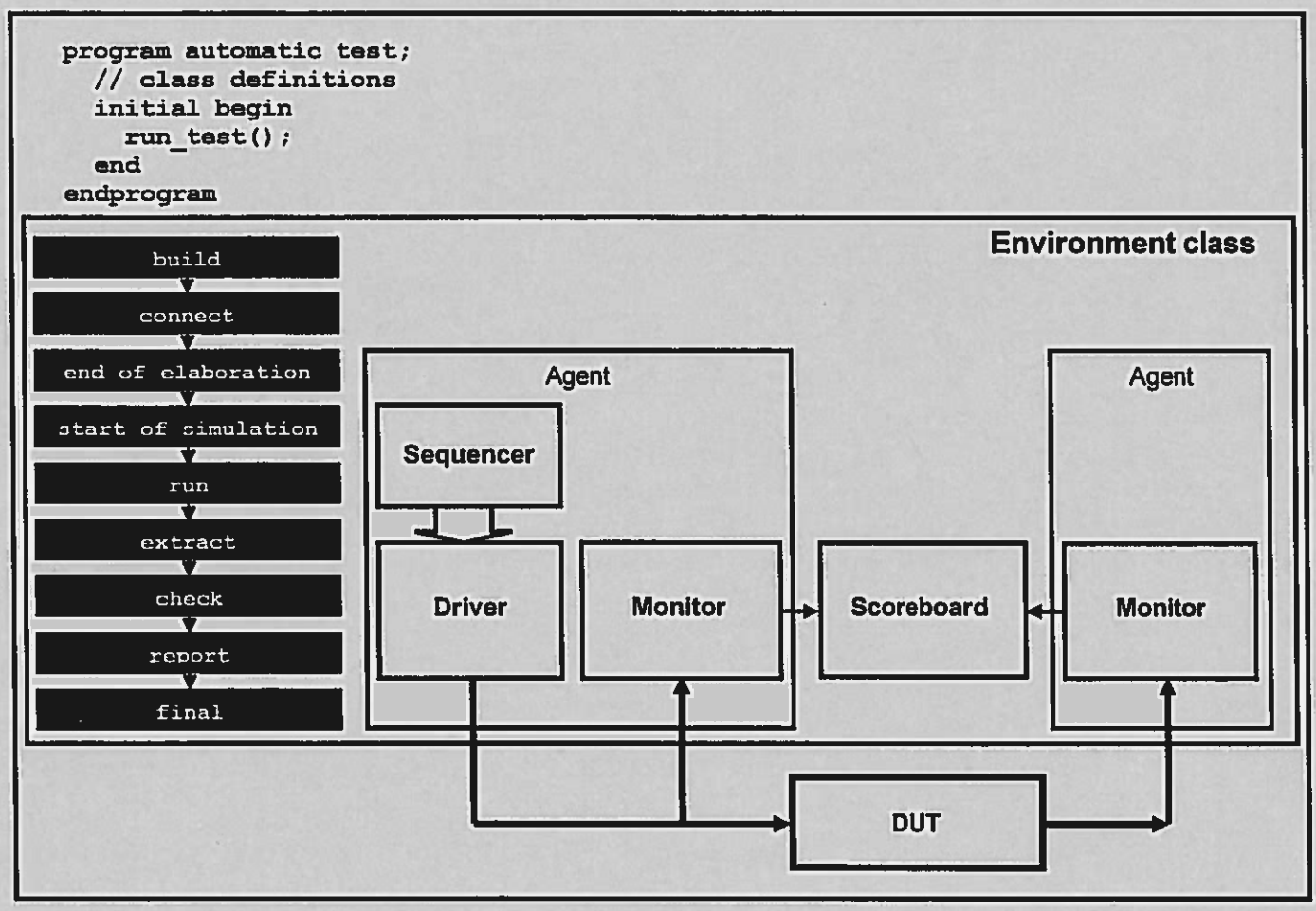


Figure 1. Lab 5 Testbench Architecture

Lab Overview

The work flow for this lab is illustrated as follows:

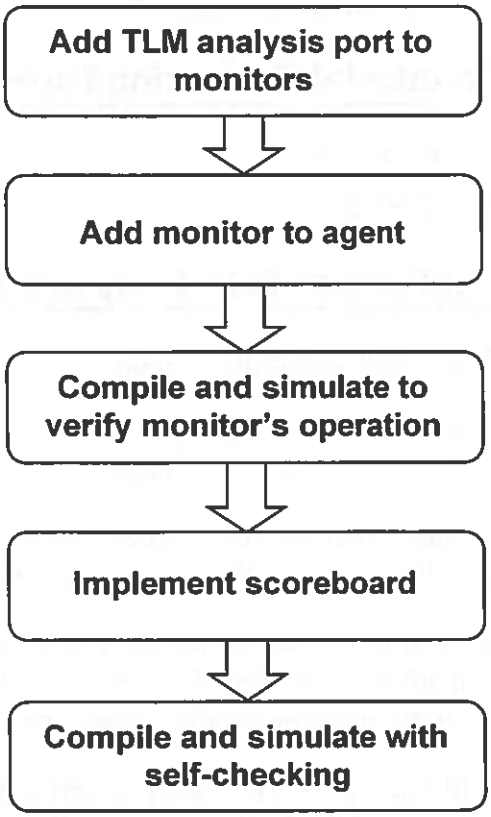


Figure 2. Lab 5 Flow Diagram

Implement Monitors and Scoreboard

You have now driven the test stimulus sequence through the DUT.

In this lab, you will add self-checking into the environment.

Task 1. Go into lab5 Working Directory

1. CD into the lab5 directory

```
> cd ../lab5
```

Task 2. Implement TLM Analysis Port in Monitor

Monitors and drivers have similarities and differences.

Similarities: both need access to DUT virtual interface. For the labs in this workshop, both need to be configured to handle a specified port of the DUT.

Differences: driver has a built-in TLM port for communication with the sequencer, but monitors require the implementer to create the required TLM port(s).

Two monitor types will be needed for the lab, one for monitoring the input to the DUT and one for monitoring the output of the DUT. The input monitor is called **`iMonitor`**. The output monitor is called **`oMonitor`**.

In this task, you will implement TLM analysis port in the input monitor. The physical device drivers and the fields identical to the driver are coded for you.

1. Open **`iMonitor.sv`** file in an editor
2. Inside the class, add a TLM analysis port object handle typed to **`packet`**

```
class iMonitor extends uvm_monitor;
  virtual router_tb_io sigs;
  int                  port_id = -1;
  uvm_analysis_port #(packet) analysis_port;
```

3. In the build phase, construct the anaylsis port object

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

Note: TLM ports in UVM do not have factory support. You cannot construct the TLM port with the proxy **`create()`** method. You must construct it with the constructor **`new()`**.

4. Locate the forever loop in the run phase

   Inside the forever loop, you need to call the device driver to re-construct the packet that was observed on the DUT physical signals. Then, use the TLM analysis port to pass it on to all other components requiring the observed transaction.

5. Do the following inside the forever loop:
   - Construct the packet object (**`tr`**) to store the observed transaction
   - Set the source address (**`sa`**) field of the packet object to **`port_id`** (The destination address (**`da`**) field will be populated by the device driver)
   - Call **`get_packet()`** method (device driver) to retrieve the observed transaction
   - Use **`uvm_info`** macro to display the content of the observed transaction
   - Pass the observed transaction to all interested components via the TLM analysis port

```
virtual task run_phase(uvm_phase phase);
  ...
  forever begin
    tr = packet::type_id::create("tr", this);
    tr.sa = this.port_id;
    get_packet(tr);
    `uvm_info("Got Input Packet", {"\n", tr.sprint()}, UVM_MEDIUM);
    analysis_port.write(tr);
  end
endtask
```

6. Save and close the file.

Task 3. Update Agent

The **`input_agent`** class currently only contains a sequencer and a driver. You need to add a monitor to the agent to complete the class definition.

1. Open **`input_agent.sv`** file in an editor
2. Include the **`iMonitor.sv`** file
3. Add the following to the class:
   - An instance of **`iMonitor`** called **`mon`**
   - An instance of **`uvm_analysis_port #(packet)`** called **`analysis_port`**

Agents can operate in one of two possible modes: active or passive. When configured to operate in the active mode, all three members (sequencer, driver and monitor) must be constructed. If the agent was configured to operate in the passive mode, only the monitor will be constructed. In the active mode, the sequencer's TLM port also need to be connected the driver's TLM port.

In the **uvm_agent** base class, the **is_active** flag is built in for this purpose. You will construct and connect the sub-component based on the state of this flag.

4. In the **build_phase()** method check the state of the **is_active** flag
   - If flag is **UVM_ACTIVE**, create the sequencer (**seqr**) and driver (**drv**) objects
5. Regardless of the state of **is_active** flag, construct the monitor (**mon**) object.
6. Construct the **analysis_port** object
7. In the **connect_phase()** method, again, check the **is_active** flag
   - If **is_active** flag is **UVM_ACTIVE**, connect the driver's TLM port (**seq_item_port**) to sequencer's TLM port (**seq_item_export**)
8. Connect the monitor's analysis port to the agent's pass-through analysis port
9. Save and close the file

Task 4. Update Environment & Test to Enable All Ports

The existing environment only has one instance of the input agent. Whereas the DUT has 16 ports that need to be tested. In this task, you will change the single instance of the agent in the environment to an array of 16 agents. Because there are 16 individual agents, they will each need to be configured to a dedicated port. Each will also need to handle the de-assertion of the control signals.

1. Open the **router_env.sv** file in an editor
2. Change the single instance of input agent to an array of 16 agents
3. In the build phase, change the construction of the single instance of input agent to construct each input agent within the array
4. And, configure each agent as follows:
   - For each agent, set a dedicated **port_id** value
   - Configure the sequencer's **default_sequence** for **reset_phase** to execute **driver_reset_sequence**
   - Configure the sequencer's **default_sequence** for **main_phase** to execute **packet_sequence**
5. Save and close the file

6. Open the **test_collections.sv** file in an editor
7. In build phase of **test_base** class, configure all input agents to use the interface instance in the harness (**router_test_top**) module.
8. Save and close the file.

Task 5. Compile and Simulate

1. Compile and simulate the testbench

```
> make
```

You should see that 160 packets were processed (16 ports X 10 items each)

Task 6. Implement Basic Scoreboard

UVM has minimal scoreboard support. Here you will try out the bare bone scoreboard implemenation.

1. Open the **scoreboard.sv** file in an editor
2. Inside the class, use typedef to simplify the name of the comparator class paramertized with packet:

```
typedef uvm_in_order_class_comparator #(packet) packet_cmp;
```

3. Then add an instance of the comparator to the class, call it comparator

```
class scoreboard extends uvm_scoreboard;
  typedef uvm_in_order_class_comparator #(packet) packet_cmp;
  packet_cmp comparator;
```

In-order comparator typed to check packet objects

You will need two TLM analysis exports to bring in packets from the input monitor and the output monitor.

4. Add the following TLM exports:

```
  uvm_analysis_export #(packet) before_export;
  uvm_analysis_export #(packet) after_export;
```

Passing iMonitor packet to comparator

Passing oMonitor packet to comparator

5. In build phase, construct the comparator and the pass-through analysis exports

```
  virtual function void build_phase(uvm_phase phase);  ...;
    comparator = packet_cmp::type_id::create("comparator", this);
    before_export = new("before_export", this);
    after_export  = new("after_export", this);
  endfunction
```

6. In connect phase, connect the pass-through exports to the comparator

```
this.before_export.connect(comparator.before_export);
this.after_export.connect(comparator.after_export);
```

7. Save and close the file

Task 7. Update the Environment Class

1. Open **router_env.sv** file in an editor.

   For the scoreboard to work, you will also need an agent at the output of the DUT. This is done for you. It is the **output_agent** class.

2. Add include statements for the **scoreboard.sv** and **output_agent.sv** files

```
`include "scoreboard.sv"
`include "output_agent.sv"
```

3. Inside the environment class, add an instance of **scoreboard** and an array of **output_agents**

```
class router_env extends uvm_env;
  ...
  scoreboard sb;
  output_agent o_agent[16];
```

4. In the build phase do the following:
   - Construct the **scoreboard** and the **output_agent** objects
   - Configure the **output_agent** objects to dedicated port

```
virtual function void build_phase(uvm_phase phase);
 ...
 sb = scoreboard::type_id::create("sb", this);
 foreach (o_agent[i]) begin
  o_agent[i] = output_agent::type_id::create($sformatf("o_agent[%0d]",i),this);
  uvm_config_db #(int)::set(this, o_agent[i].get_name(), "port_id", i);
 end
endfunction
```

5. In the connect phase, connect the scoreboard to the agents' analysis ports:

```
virtual function void connect_phase(uvm_phase phase);
  `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  foreach (i_agent[i]) begin
    i_agent[i].analysis_port.connect(sb.before_export);
  end
  foreach (o_agent[i]) begin
    o_agent[i].analysis_port.connect(sb.after_export);
  end
endfunction
```

6. Save and close the file
7. Open the **test_collection.sv** file in an editor
8. In build phase of **test_base** class, configure all output agents to use the interface instance in the harness (**router_test_top**) module.
9. Save and close the file.

Task 8. Compile And Debug The Program

1. Compile and run the simulation

```
> make
```

You should see something like the following message at end of simulation:

```
** Report counts by id
[Comparator Match]   153
[Comparator Mismatch]     6
[DRV_RUN]   160
[Got_Input_Packet]   160
[Got_Output_Packet]   159
[MISCMP]    12
[RNTST]     1
[Scoreboard_Report]     1
[UVMTOP]     1
```

The simulation is reporting that 160 packets were detected in the input but only 159 packets were observed at the output.

(Ignore the Mismatches for now. It will be solved in a coming Step.)

This is the expected behavior. The cause of the missing output is due to the latency of the transaction flowing through the DUT. Since the objection were only raised and dropped on the input side, as soon as the input is done, as far as the UVM simulation is concerned, everything is done because there are no existing objections.

This is a common problem in UVM testbench. There are multiple ways to solve this problem.

One way to correct this is to take care of the expected latency on the input side by implementing an objection drain time. If an objection drain time is set, then when the objection count reaches 0, the phase must wait for the drain time to elapse before terminating the phase. If another objection is raised during the drain time, the phase objection mechanism starts over and waits for objection count to reach 0 again.

Other ways to solve the problem will be address in Day 3 lecture.

For now, use the drain time mechanism.

2. Open the **test_collection.sv** file
3. Locate **test_base** class
4. In the **main_phase()** method, retrieve the objection handle for the phase, then set drain time to 1us. This should be sufficient for the lab DUT.

```
virtual task main_phase(uvm_phase phase);
  uvm_objection objection;
  super.main_phase(phase);
  objection = phase.get_objection();
  objection.set_drain_time(this, 1us);
endtask
```

5. Save and close the file
6. Compile and run the simulation again:

   ```
   > make
   ```

   You should see all 160 packets on the output observed.

   However, due to the basic nature of the UVM comparator and scoreboard, you should also see that there are mismatches.

   ```
   ** Report counts by id
   [Comparator Match]    140
   [Comparator Mismatch]     20
   [DRV_RUN]    160
   [Got Input Packet]    160
   [Got Output Packet]    160
   [MISCMP]    20
   ```

   Is this a testbench problem or a DUT problem? One way to isolate the problem is to just test one port.

7. Compile and run the simulation on destination address 3:

   ```
   > make test=test_da_3_seq
   ```

   You should see that 320 packets (20 packets per input port) were successfully matched. If you see other error, then you have made a mistake in your testbench code that needs to be corrected.

If it passes, what's the problem? The problem is that UVM only provides a mechanism for checking in-order sequence comparison. The DUT you are testing can have simultaneous input and output packet observed at different ports. Which one of these output packet will be checked against the input packet? It is non-deterministic. Thus the lab problem arises.

Again, this is a common problem in UVM testbenches.

How can one solve this problem? If the sequence is truly out-of-order, then one must write an out-of-order scoreboard from scratch. If the sequence is in-order, but, there are multiple streams of in-order sequences at the same time, then the UVM's **`uvm_in_order_class_comparator`** class can still be used to implement a multi-stream scoreboard.

A multi-stream scoreboard has been coded for you. Try it out in the next step.

8. Open the **`router_env.sv`** file in an editor
9. Replace the include statement for **`scoreboard.sv`** with an include statement for **`ms_scoreboard.sv`**
10. Save and close the file

Task 9. Compile And Simulate

1. Compile and simulate the testbench.

```
> make
```

All should now match!

You can verify that there are activities in the DUT with DVE.

2. Open DVE with the following command:

```
> make dve
```

You should see that packets were driven through the DUT.

3. Run the destination address is 3 and 20 packets test:

```
> make test=test_da_3_seq
```

4. Take a look at DVE again

You should see that 320 packets were driven through port 3.

Congratulations, you have completed Lab 5!

Answers / Solutions

test_collection.sv Solution:

```
class test_base extends uvm_test;
  `uvm_component_utils(test_base)
  router_env env;
  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    env = router_env::type_id::create("env", this);

    uvm_config_db#(virtual router_tb_io)::set(this, "env.i_agent[*]",
"sigs", router_test_top.router_if.agent_vif);
    uvm_config_db#(virtual router_tb_io)::set(this, "env.o_agent[*]",
"sigs", router_test_top.router_if.agent_vif);
    uvm_config_db#(virtual reset_tb_io)::set(this, "env.r_agent", "sigs",
router_test_top.reset_if.agent_vif);
  endfunction

  virtual task main_phase(uvm_phase phase);
    uvm_objection objection;
    super.main_phase(phase);
    objection = phase.get_objection();
    objection.set_drain_time(this, 1us);
  endtask

  virtual function void final_phase(uvm_phase phase);
    super.final_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    uvm_top.print_topology();
    factory.print();
  endfunction
endclass

// Other code left out.  See file for content.
```

router_env.sv Solution:

```
class router_env extends uvm_env;
  reset_agent r_agent;
  input_agent i_agent[16];
  scoreboard sb;
  output_agent o_agent[16];

  `uvm_component_utils(router_env)

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

    r_agent = reset_agent::type_id::create("r_agent", this);
    uvm_config_db #(uvm_object_wrapper)::set(this,
"r_agent.seqr.reset_phase", "default_sequence",
reset_sequence::get_type());

    foreach (i_agent[i]) begin
      i_agent[i] = input_agent::type_id::create($sformatf("i_agent[%0d]",
i), this);
      uvm_config_db #(int)::set(this, i_agent[i].get_name(), "port_id", i);
      uvm_config_db #(uvm_object_wrapper)::set(this,
{i_agent[i].get_name(), ".", "seqr.reset_phase"}, "default_sequence",
driver_reset_sequence::get_type());
      uvm_config_db #(uvm_object_wrapper)::set(this,
{i_agent[i].get_name(), ".", "seqr.main_phase"}, "default_sequence",
packet_sequence::get_type());
    end

    sb = scoreboard::type_id::create("sb", this);
    foreach (o_agent[i]) begin
      o_agent[i] = output_agent::type_id::create($sformatf("o_agent[%0d]",
i), this);
      uvm_config_db #(int)::set(this, o_agent[i].get_name(), "port_id", i);
    end
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    foreach (i_agent[i]) begin
      i_agent[i].analysis_port.connect(sb.before_export);
    end
    foreach (o_agent[i]) begin
      o_agent[i].analysis_port.connect(sb.after_export);
    end
  endfunction
endclass
```

input_agent.sv Solution:

```
typedef uvm_sequencer #(packet) packet_sequencer;

class input_agent extends uvm_agent;
  packet_sequencer     seqr;
  driver               drv;
  virtual router_tb_io sigs;
  int                  port_id = -1;
  iMonitor             mon;
  uvm_analysis_port #(packet) analysis_port;

  `uvm_component_utils(input_agent)

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction: new

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

    if (is_active == UVM_ACTIVE) begin
      seqr = packet_sequencer::type_id::create("seqr", this);
      drv  = driver::type_id::create("drv", this);
    end

    mon = iMonitor::type_id::create("mon", this);

    analysis_port = new("analysis_port", this);

    uvm_config_db#(int)::get(this, "", "port_id", port_id);
    uvm_config_db#(virtual router_tb_io)::get(this, "", "sigs", sigs);

    uvm_config_db#(int)::set(this, "*", "port_id", port_id);
    uvm_config_db#(virtual router_tb_io)::set(this, "*", "sigs", sigs);
  endfunction: build_phase

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

    if (is_active == UVM_ACTIVE) begin
      drv.seq_item_port.connect(seqr.seq_item_export);
    end

    mon.analysis_port.connect(this.analysis_port);
  endfunction: connect_phase
endclass: input_agent
```

iMonitor.sv Solution:

```
class iMonitor extends uvm_monitor;
  virtual router_tb_io sigs;
  int                  port_id = -1;
  uvm_analysis_port #(packet) analysis_port;

  `uvm_component_utils_begin(iMonitor)
    `uvm_field_int(port_id, UVM_DEFAULT | UVM_DEC)
  `uvm_component_utils_end

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    uvm_config_db#(int)::get(this, "", "port_id", port_id);
    if (!(port_id inside {-1, [0:15]})) begin
      `uvm_fatal("CFGERR", $sformatf("port_id must be {-1, [0:15]}, not
%0d!", port_id));
    end
    uvm_config_db#(virtual router_tb_io)::get(this, "", "sigs", sigs);
    if (sigs == null) begin
      `uvm_fatal("CFGERR", "iMonitor DUT interface not set");
    end

    analysis_port = new("analysis_port", this);
  endfunction

  virtual task run_phase(uvm_phase phase);
    packet tr;
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

    forever begin
      tr = packet::type_id::create("tr", this);
      tr.sa = this.port_id;
      get_packet(tr);
      `uvm_info("Got_Input_Packet", {"\n", tr.sprint()}, UVM_MEDIUM);
      analysis_port.write(tr);
    end
  endtask

  //
  // See file see device driver code
  //

endclass
```

oMonitor.sv Solution:

```
class oMonitor extends uvm_monitor;
  int port_id = -1;
  virtual router_tb_io sigs;
  uvm_analysis_port #(packet) analysis_port;

  `uvm_component_utils_begin(oMonitor)
    `uvm_field_int(port_id, UVM_DEFAULT | UVM_DEC)
  `uvm_component_utils_end

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    uvm_config_db#(int)::get(this, "", "port_id", port_id);
    if (!(port_id inside {-1, [0:15]})) begin
      `uvm_fatal("CFGERR", $sformatf("port_id must be {-1, [0:15]}, not
%0d!", port_id));
    end
    uvm_config_db#(virtual router_tb_io)::get(this, "", "sigs", sigs);
    if (sigs == null) begin
      `uvm_fatal("CFGERR", "oMonitor DUT interface not set");
    end
    analysis_port = new("analysis_port", this);
  endfunction

  virtual task run_phase(uvm_phase phase);
    packet tr;
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    forever begin
      tr = packet::type_id::create("tr", this);
      tr.da = this.port_id;
      this.get_packet(tr);
      `uvm_info("Got_Output_Packet", {"\n", tr.sprint()}, UVM_MEDIUM);
      analysis_port.write(tr);
    end
  endtask

  // See file for the device driver code
  //

endclass
```

scoreboard.sv Solution:

```
class scoreboard extends uvm_scoreboard;
  typedef uvm_in_order_class_comparator #(packet) packet_cmp;
  packet_cmp comparator;

  uvm_analysis_export #(packet) before_export;
  uvm_analysis_export #(packet) after_export;

  `uvm_component_utils(scoreboard)

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

    comparator = packet_cmp::type_id::create("comparator", this);
    before_export = new("before_export", this);
    after_export  = new("after_export", this);
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);

    this.before_export.connect(comparator.before_export);
    this.after_export.connect(comparator.after_export);
  endfunction

  virtual function void report_phase(uvm_phase phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    `uvm_info("Scoreboard_Report",
      $sformatf("Comparator Matches = %0d, Mismatches = %0d",
comparator.m_matches, comparator.m_mismatches), UVM_MEDIUM);
  endfunction
endclass
```

ms_scoreboard.sv Solution:

```
`uvm_analysis_imp_decl(_before)
`uvm_analysis_imp_decl(_after)

class scoreboard extends uvm_scoreboard;
  uvm_analysis_imp_before #(packet, scoreboard) before_export;
  uvm_analysis_imp_after  #(packet, scoreboard) after_export;
  uvm_in_order_class_comparator #(packet) comparator[16];

  `uvm_component_utils(scoreboard)

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    before_export = new("before_export", this);
    after_export  = new("after_export", this);
    for (int i=0; i < 16; i++) begin
      comparator[i]       = new($sformatf("comparator_%0d", i), this);
    end
  endfunction

  virtual function void write_before(packet pkt);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    comparator[pkt.da].before_export.write(pkt);
  endfunction

  virtual function void write_after(packet pkt);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    comparator[pkt.da].after_export.write(pkt);
  endfunction

  virtual function void report();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    foreach (comparator[i]) begin
      `uvm_info("Scoreboard_Report",
       $sformatf("Comparator[%0d] Matches = %0d, Mismatches = %0d",
         i, comparator[i].m_matches, comparator[i].m_mismatches),
UVM_MEDIUM);
    end
  endfunction

endclass
```

6 Virtual Sequence

Learning Objectives

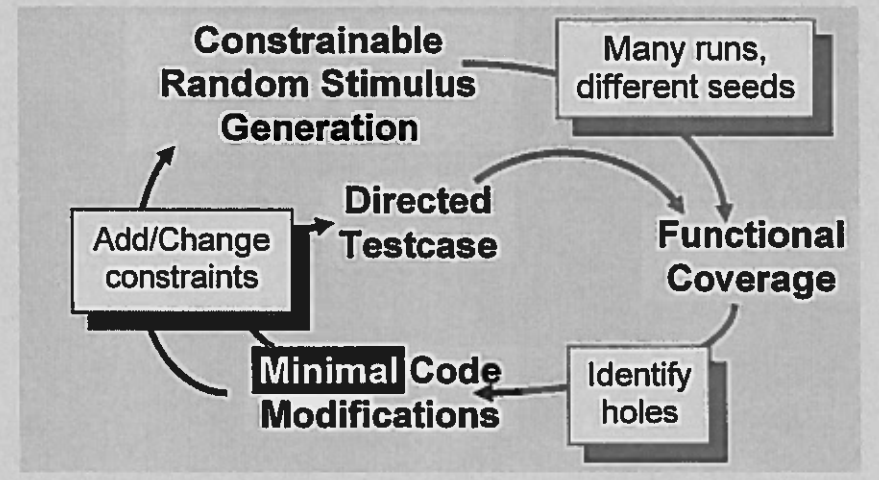
After completing this lab, you should be able to:

- Develop a virtual sequence to control reset sequence execution order
- Implement test to manage sequencers via virtual sequence

Lab Duration: 45 minutes

Getting Started

Through the first five labs, you have developed a complete verification platform to exercise the DUT. In the next two labs, you will expand the testbench to add greater flexibility for even more productivity.




```
program automatic test;
  // class definitions
  initial begin
    run_test();
  end
endprogram
```

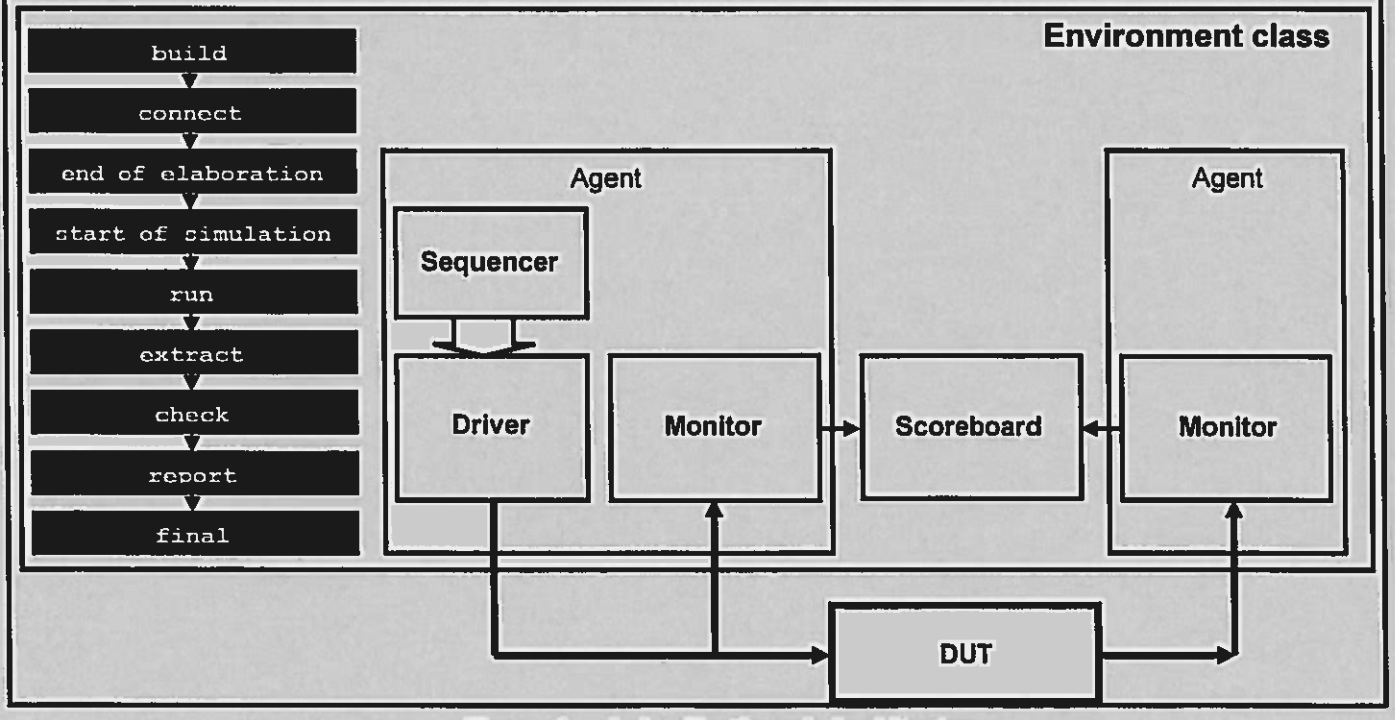


Figure 1. Lab 6 Testbench Architecture

Lab Overview

The work flow for this lab is illustrated as follows:

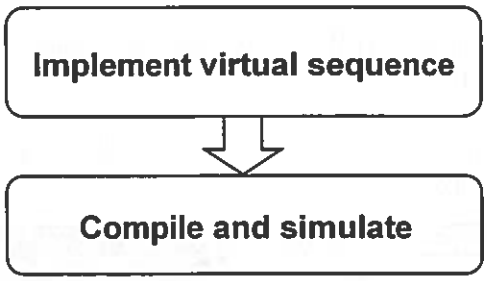


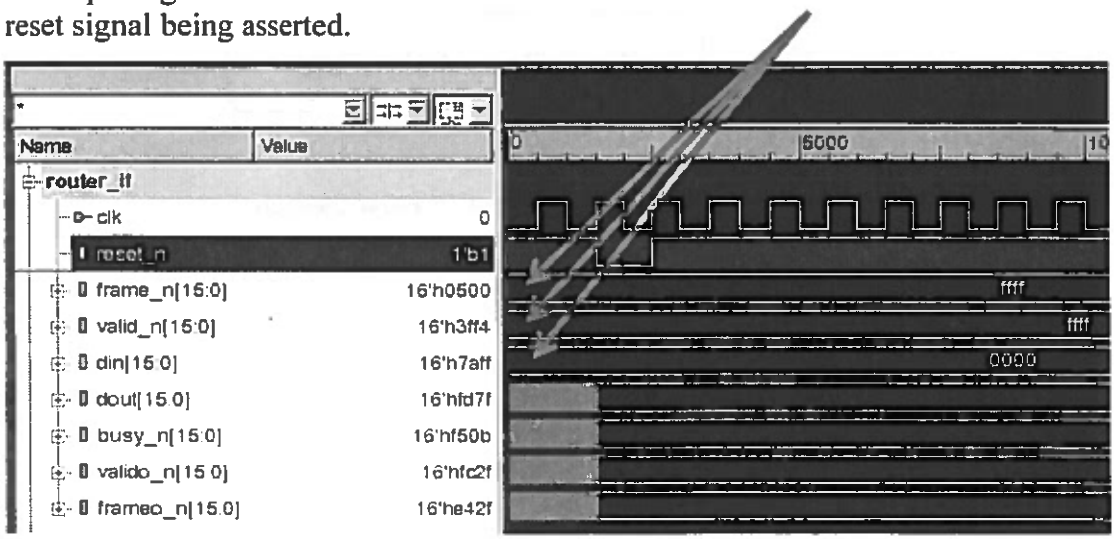
Figure 2. Lab 6 Flow Diagram

Virtual Sequence

In lab 5, you have sucessfully driven all input and output ports. All packets were verified to have been processed correctly by the DUT.

However, if you look carefully at the reset execution, you will see that it is not technically correct.

The input signal into the DUT at time 0 should not be at a known value without the reset signal being asserted.



The cause of this incorrect timing is due to each agent's sequencer being configured to execute their own reset sequence in the reset phase without any knowledge or dependency on any other sequence that may be going on at the same time.

From lab 5:

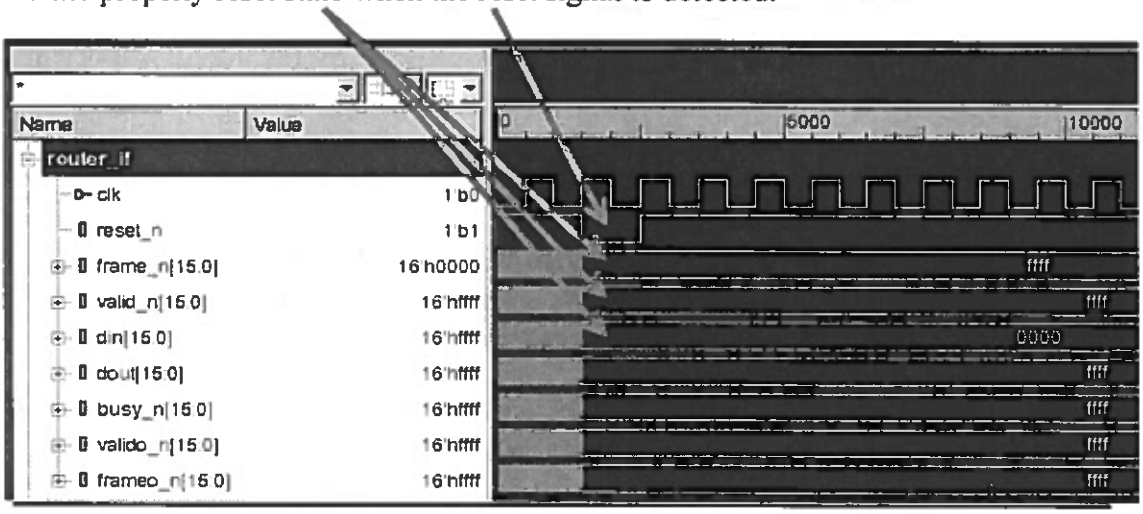
```
class router_env extends uvm_env;
  // A lot of code left off.  Only relevant code is shown

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    uvm_config_db#(uvm_object_wrapper)::set(this, "r_agent.seqr.reset_phase",
                                            "default_sequence", reset_sequence::get_type());

    foreach (i_agent[i]) begin
      uvm_config_db#(uvm_object_wrapper)::set(this, {i_agent[i].get_name(),
      ".seqr.reset_phase"}, "default_sequence",
driver_reset_sequence::get_type());
    end
  endfunction
endclass
```

The correct way to handle the external signals is to have the signals default to their default states (x for logic, z for wire) at time 0. Then, in testbench, set these signals to the properly reset state when the reset signal is detected.



The way to manage this in UVM is to implement a virtual sequence and a virtual sequencer to execute this virtual sequence.

Task 1. Implement Virtual Reset Sequence

For this task, you will create a virtual sequence to manage the execution of the existing reset_sequence and driver_reset_sequence.

1. Open **`virtual_reset_sequence.sv`** file in an editor
2. Locate the **`virtual_reset_sequencer`** class at the top of the file.

   For the sake of reducing lab time, the virtual sequencer is created for you. Take a quick look and make sure you understand why the virtual sequencer is declared as it is.
3. Next, locate the **`virtual_reset_sequence`** class
4. Inside the class, use the **`` `uvm_declare_p_sequencer``** macro to specify the parent sequencer type
5. Continuing in the class, create a **`reset_sequence`** handle called **`r_seq`** and a **`driver_reset_sequence`** handle called **`d_seq`**
6. Inside the **`body()`** method do the following:
   - Execute the **`reset_sequence`** with the parent sequencer's **`r_seqr`**
   - Iterate through the parent sequencer's **`pkt_seqr`** queue and execute the the **`driver_reset_sequence`**
7. Save and close the file

Task 2. Execute Virtual Sequence in Environment

1. Open **router_env.sv** file in an editor
2. Bring the **virtual_reset_sequence.sv** file into the class with a **`include** statement
3. Inside the **router_env** class, create a **virtual_reset_sequencer** handle called **v_reset_seqr**
4. In the **build_phase**, construct this virtual sequencer
5. Turn off the **r_agent**'s sequencer execution at the reset phase by setting **"default_sequence"** to **null**
6. And, turn off all the **i_agent**'s sequencer execution at the reset phase by setting **"default_sequence"** to **null**
7. Then, configure the virtual sequencer to execute the virtual reset sequence at the reset phase

```
virtual function void build_phase(uvm_phase phase);
  ...;
  uvm_config_db #(uvm_object_wrapper)::set(this, "r_agent.seqr.reset_phase",
                                           "default_sequence", null);
  uvm_config_db #(uvm_object_wrapper)::set(this, {i_agent[i].get_name(),
                          ".", "seqr.reset_phase"}, "default_sequence", null);
  uvm_config_db #(uvm_object_wrapper)::set(this, "v_reset_seqr.reset_phase",
                     "default_sequence", virtual_reset_sequence::get_type());
  ...;
endfunction
```

8. In the **connect_phase**, push the input agent's sequencers onto the virtual sequencer's **pkt_seqr** queue.
9. Continuing in the **connect_phase**, set the virtual sequencer's **r_seqr** handle to reference the reset agent's sequencer (**r_agent.seqr**)

   Because the virtual reset sequence will control all reset sequence executions, you will need to turn off the existing reset sequence executions. Otherwise, both sequence execution will occur.
10. Save and close the file
11. Compile and simulate the testbench to see if this works

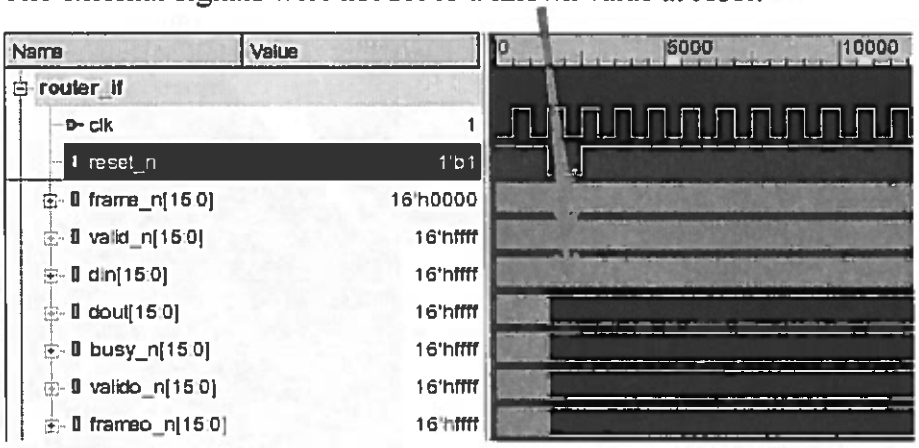
    > make

    You should see a bunch of scoreboard mismatches! What happened?

12. Bring up dve to see what's happening

```
> make dve
```

The external signals were not set to a known value at reset!



The reason is because, in the virtual sequence, each of the sequences was executed sequentially not concurrently as they needed to be.

```
virtual task body();
  `uvm_do_on(r_seq, p_sequencer.r_seqr);
  foreach (p_sequencer.pkt_seqr[i]) begin
    `uvm_do_on(d_seq, p_sequencer.pkt_seqr[i]);
  end
endtask
```

Let's correct this.

Task 3. Execute Sequences Concurrently

1. Open **virtual_reset_sequence.sv** file in an editor
2. Comment out the code from Task 1, Step 6 and replace it with the following:

```
fork
  `uvm_do_on(r_seq, p_sequencer.r_seqr);
  foreach (p_sequencer.pkt_seqr[i]) begin
    int j = i;
    fork
      `uvm_do_on(d_seq, p_sequencer.pkt_seqr[j]);
    join_none
  end
join
```

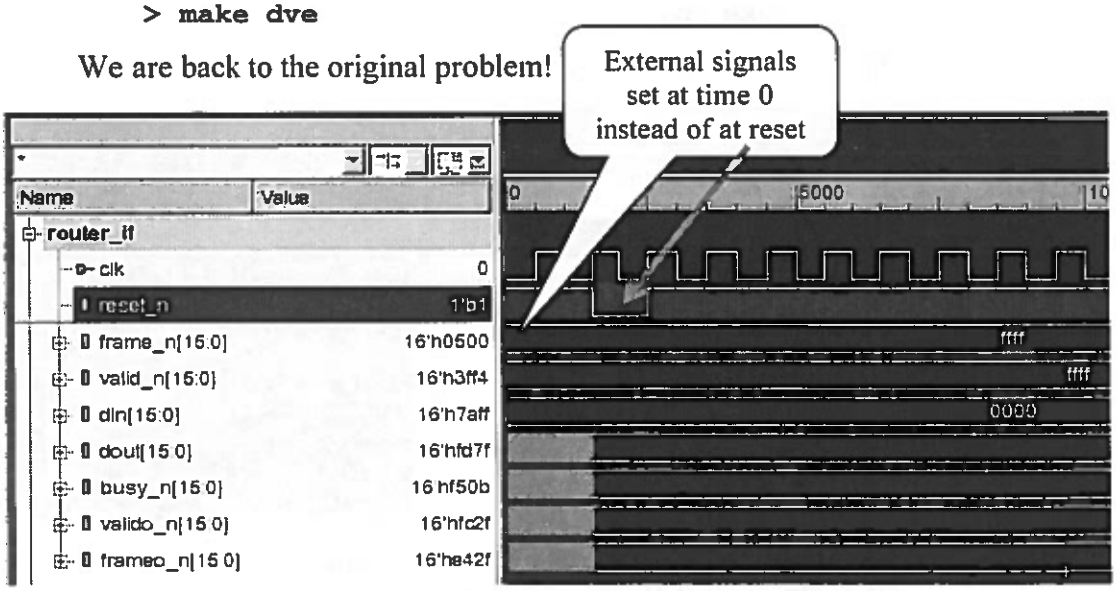
3. Save and close the file
4. Compile and simulate the testbench to see if this works

```
> make
```

5. Did it work? Are you sure? Bring up dve to make sure

```
> make dve
```

We are back to the original problem!



This is actually a fairly common issue that test developer face. Within a given phase, multiple sequences need to execute concurrently, yet each sequence may need to wait for a specific condition to happen before execution starts.

One way to handle this is to make use of the **uvm_event** mechanism.

Task 4. Implementing uvm_event for Synchronization

An **uvm_event** called "**reset**" is already embedded in the **reset_monitor**. Take a quick look at how it is done.

1. Open **reset_agent.sv** file in an editor
2. Locate the **reset_monitor** class

Inside the class, you will see the following:

```
class reset_monitor extends uvm_monitor;
  virtual reset_tb_io sigs;   // DUT virtual interface
  uvm_analysis_port #(reset_tr) analysis_port;
  uvm_event reset_event = uvm_event_pool::get_global("reset");
```

The last line uses the **uvm_event_pool** class to get the **uvm_event** singleton object called "**reset**".

If the "**reset**" **uvm_event** singleton object doesn't already exist, it is created now.

The intent of creating this "**reset**" **uvm_event** singleton object is to let all observers who are interested in the occurrence of a system reset use this singleton object and watch for reset occurrences.

3. Locate the **detect()** method and take a look at how the event is handled

```
virtual task detect(reset_tr tr);
  @(sigs.reset_n);
  assert(!$isunknown(sigs.reset_n));
  if (sigs.reset_n == 1'b0) begin
    tr.kind = reset_tr::ASSERT;
    reset_event.trigger();
  end else begin
    tr.kind = reset_tr::DEASSERT;
    reset_event.reset();
  end
endtask: detect
```

"reset" event is triggered (turned on) at the detection of assertion of reset signal

"reset" event is reset (turned off) at the detection of de-assertion of reset signal

4. Close the file
5. Open **virtual_reset_sequence.sv** file in an editor
6. Add the "**reset**" **uvm_event** singleton object to the sequence class
7. Comment out the code from Task 3, Step 2 and replace it with the following:

```
fork
  `uvm_do_on(r_seq, p_sequencer.r_seqr);
  foreach (p_sequencer.pkt_seqr[i]) begin
    int j = i;
    fork
      begin
        reset_event.wait_on();
        `uvm_do_on(d_seq, p_sequencer.pkt_seqr[j]);
      end
    join_none
  end
join
```

Wait for the "reset" uvm_event to turn on before executing the driver reset sequence

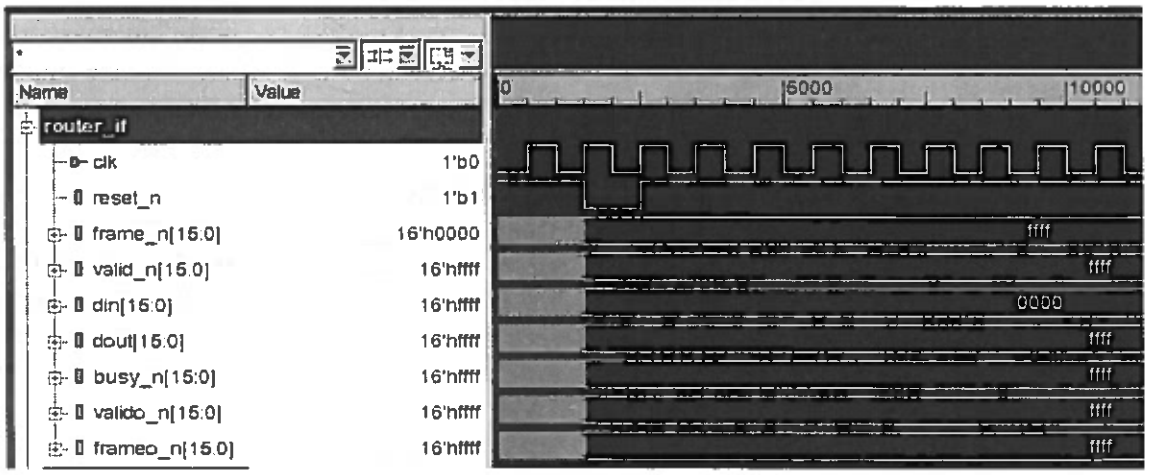
8. Save and close the file
9. Compile and simulate the testbench to see if this works

```
> make
```

10. Did it work? Bring up dve to make sure

```
> make dve
```

You should now see the following correct timing:



Congratulations, you have completed Lab 6!

Answers / Solutions

virtual_reset_sequencer.sv Solution:

```
class virtual_reset_sequencer extends uvm_sequencer;
  `uvm_component_utils(virtual_reset_sequencer)
  reset_sequencer  r_seqr;
  packet_sequencer pkt_seqr[$];

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction
endclass
```

virtual_reset_sequence.sv Solution:

```
class virtual_reset_sequence extends uvm_sequence;
  `uvm_object_utils(virtual_reset_sequence)
  `uvm_declare_p_sequencer(virtual_reset_sequencer)

  reset_sequence         r_seq;
  driver_reset_sequence  d_seq;
  uvm_event reset_event = uvm_event_pool::get_global("reset");

  function new(string name="virtual_reset_sequence");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual task pre_start();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if ((get_parent_sequence() == null) && (starting_phase != null))
      starting_phase.raise_objection(this);
  endtask

  virtual task body();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    fork
      `uvm_do_on(r_seq, p_sequencer.r_seqr);
      foreach (p_sequencer.pkt_seqr[i]) begin
        int j = i;
        fork
          begin
            reset_event.wait_on();
            `uvm_do_on(d_seq, p_sequencer.pkt_seqr[j]);
          end
        join_none
      end
    join
  endtask

  virtual task post_start();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if ((get_parent_sequence() == null) && (starting_phase != null))
      starting_phase.drop_objection(this);
  endtask
endclass
```

router_env.sv Solution:

```
class router_env extends uvm_env;
  reset_agent r_agent;
  input_agent i_agent[16];
  output_agent o_agent[16];
  scoreboard sb;
  virtual_reset_sequencer  v_reset_seqr;
  `uvm_component_utils(router_env)
  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    v_reset_seqr = virtual_reset_sequencer::type_id::create("v_reset_seqr",
this);
    r_agent = reset_agent::type_id::create("r_agent", this);
    uvm_config_db #(uvm_object_wrapper)::set(this, "r_agent.seqr.reset_phase",
"default_sequence", null);
    foreach (i_agent[i]) begin
      i_agent[i] = input_agent::type_id::create($sformatf("i_agent[%0d]", i),
this);
      uvm_config_db #(int)::set(this, i_agent[i].get_name(), "port_id", i);
      uvm_config_db #(uvm_object_wrapper)::set(this, {i_agent[i].get_name(),
".", "seqr.main_phase"}, "default_sequence", packet_sequence::get_type());
      uvm_config_db #(uvm_object_wrapper)::set(this, {i_agent[i].get_name(),
".", "seqr.reset_phase"}, "default_sequence", null);
    end
    uvm_config_db #(uvm_object_wrapper)::set(this, "v_reset_seqr.reset_phase",
"default_sequence", virtual_reset_sequence::get_type());
    sb = scoreboard::type_id::create("sb", this);
    foreach (o_agent[i]) begin
      o_agent[i] = output_agent::type_id::create($sformatf("o_agent[%0d]", i),
this);
      uvm_config_db #(int)::set(this, o_agent[i].get_name(), "port_id", i);
    end
  endfunction
  virtual function void connect_phase(uvm_phase phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    foreach (i_agent[i]) begin
      i_agent[i].analysis_port.connect(sb.before_export);
      v_reset_seqr.pkt_seqr.push_back(i_agent[i].seqr);
    end
    foreach (o_agent[i]) begin
      o_agent[i].analysis_port.connect(sb.after_export);
    end
    v_reset_seqr.r_seqr = this.r_agent.seqr;
  endfunction
endclass
```

7 UVM Register Abstraction Layer (RAL)

Learning Objectives

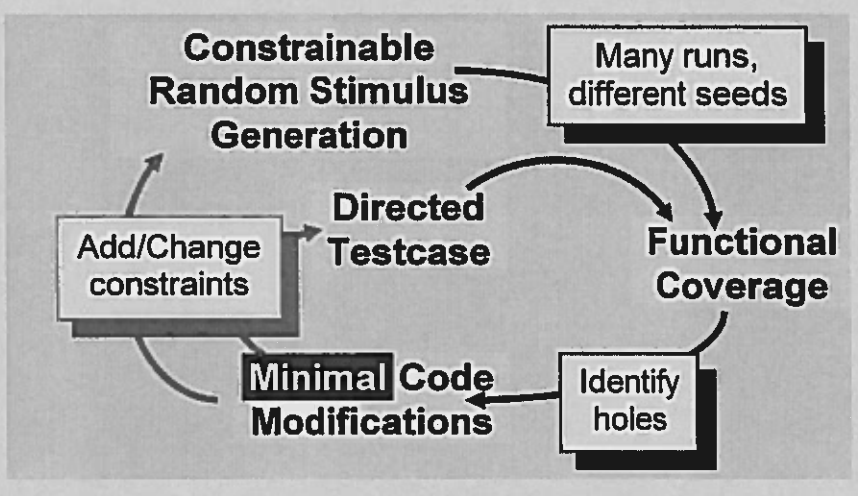
After completing this lab, you should be able to:

- Write a register access sequence without UVM register abstraction
- Compile and verify front door access for registers are working correctly
- Describe registers in .ralf format
- Convert .ralf format to UVM register abstraction
- Create adopter class
- Add UVM register model and adopt to environment
- Develop a register access sequence using UVM register abstraction
- Compile and simulate with UVM register sequence

Lab Duration: 60 minutes

Getting Started

The testbench is now reasonably complete. The only major thing that is lacking is accessing registers within the DUT. In this lab, you will implement the UVM register abstraction to access the DUT registers.



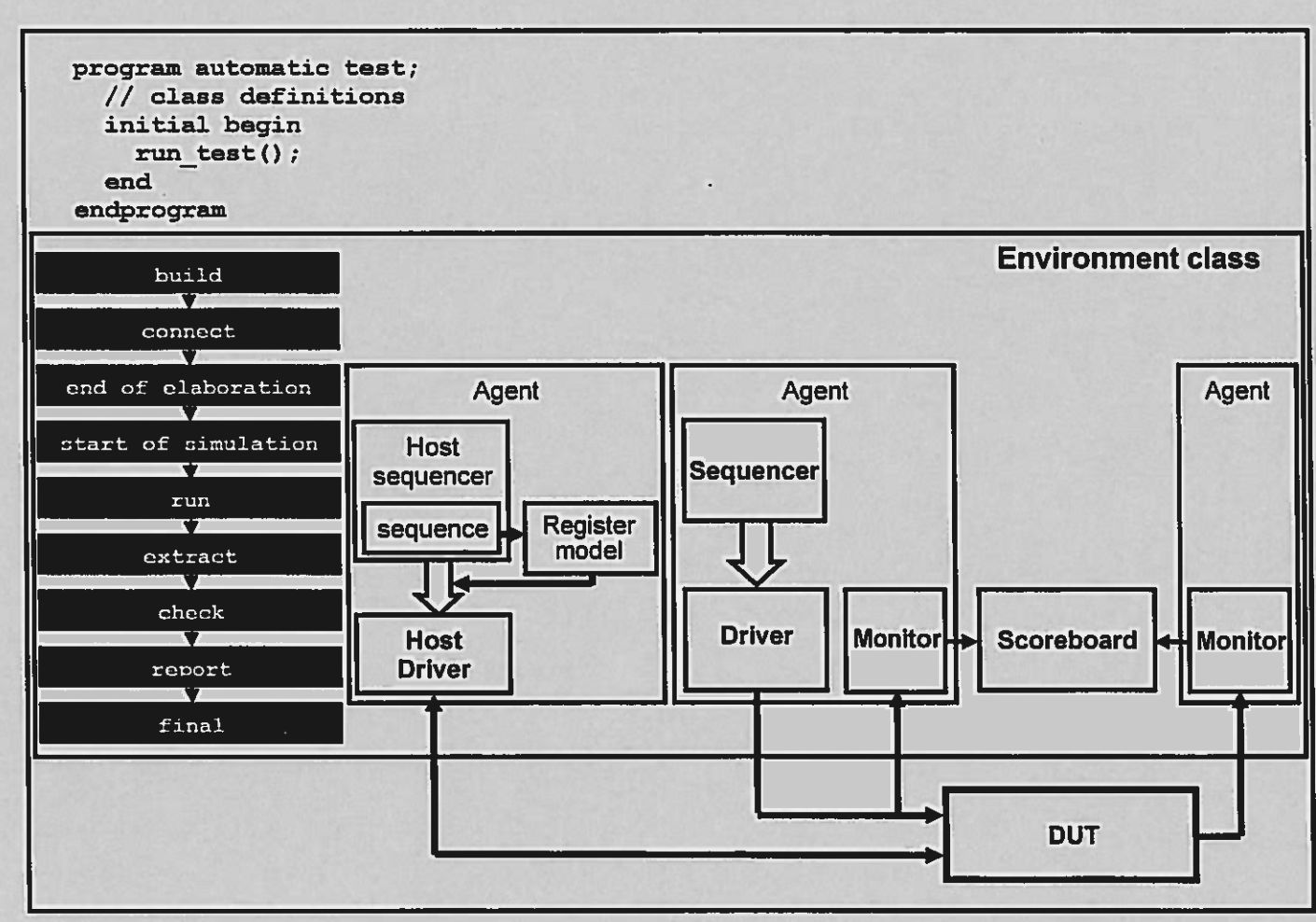


Figure 1. Lab 7 Testbench Architecture

Lab Overview

The work flow for this lab is illustrated as follows:

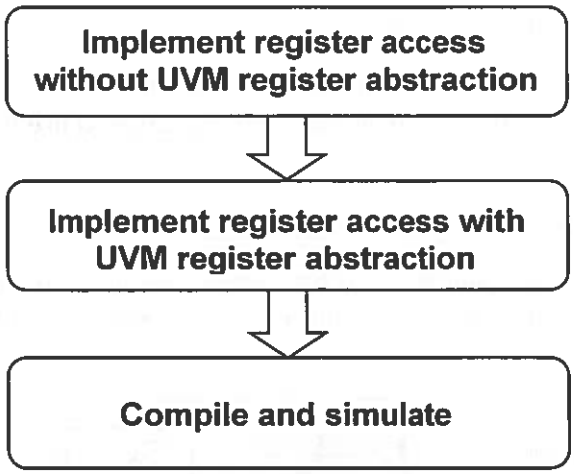


Figure 2. Lab 7 Flow Diagram

Implement UVM Register Abstraction

Task 1. Go into lab7 Working Directory

1. CD into the lab7 directory

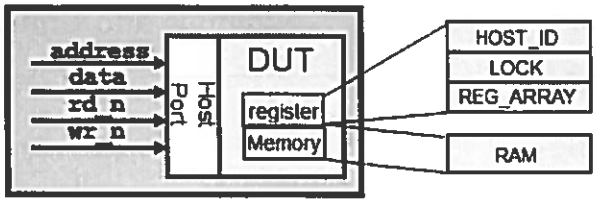
```
> cd ../lab7
```

The router RTL has been modified with registers and memories. For this lab, you will be developing an environment and sequence to use these registers and memories.

The following are the specs for the registers:

- The word width is 16 bits for each register/memory
- The address is word address based (not byte address).

Address Map	
HOST_ID	0x0000
LOCK	0x0100
R_ARRAY	0x1000-0x10FF
RAM	0x4000-0x4FFF



At address 0x0000, two read-only fields exist: chip id (CHIP_ID) and revsion id (REV_ID). The static values are as shown in the following table.

HOST_ID Register		
Field	CHIP_ID	REV_ID
Bits	15-8	7-0
Mode	ro	ro
Reset	0x5A	0x03

At address 0x0100, there is a port locking field (LOCK). If a bit has the value of 1, the corresponding port of the router will be disabled. To enable a port, the LOCK bit must be cleared with a write-one-to-clear. (writing a one to that bit location will clear the bit). The default value is 16'hffff, meaning that all ports are disabled at reset. The reading and writing of the field is word based.

LOCK Register	
Field	LOCK
Bits	15-0
Mode	w1c
Reset	0xff

At address 0x1000 - 0x10FF, there is an array of registers (R_ARRAY[256]). These registers can be written and read. The reset values are 16'h0000.

R_ARRAY[256] Registers	
Field	H_REG
Bits	15-0
Mode	rw
Reset	0x00

There is also 4K words of RAM located at address 0x4000 – 0x4FFFF.

RAM (4K)	
Bits	15-0
Mode	rw

Task 2. Create Sequence without Register Abstraction

The first step in implementing DUT register access is to implement all components supporting the access without UVM register abstraction.

The host data class is created for you in **host_data.sv**. It's pretty simple. It contains data and address field with two possible kinds of operation: read and write.

```
class host_data #(int base_address=0) extends uvm_sequence_item;
  typedef enum {READ, WRITE} kind_e;
  typedef enum {IS_OK, NOT_OK, HAS_X} status_e;
  rand status_e  status;
  rand kind_e    kind;
  rand bit[15:0] addr;
  rand bit[15:0] data;
  ...
endclass
```

A host agent class is also created for you in **host_agent.sv**. If interested, you can examine the file. However, aside from the device drivers, it contain the same things that you have already done in the previous labs. So, there is not much to gain in going through it in detail.

What's more important is to develop a sequence which is able to read and write the registers. You will do this in the next step.

1. Open **host_sequence.sv** file in an editor
2. In **host_bfm_sequence** class, create a **body()** method to do the following:
   - Read and check the content of the **HOST_ID** register at address **'h0** (The value must be **'h5A03**)
   - Read and check the content of the **LOCK** register at address **'h0100** (The value must be **'hffff** after reset)
   - Write all one's (**'1**) to the register to enable all ports
   - Read and check the content of the **LOCK** register to verify it is now **'0**
   - Write gray code pattern to the **R_ARRAY**
   - Read back and check the content of the **R_ARRAY** for gray code pattern
   - Write walking one's to the **RAM**
   - Read back and check the content of the **RAM** for walking one's
3. Save and close the file

Task 3. Add Host Reset to Virtual Reset Sequence

Other than the user sequences, you also need a reset sequence for the host interface. To conserve lab time, this sequence is written for you. If interested, you can take a look at the **host_reset_sequence** class in the **host_sequence.sv** file.

Otherwise, just add this **host_reset_sequence** to the virtual reset sequence.

1. Open **virtual_reset_sequence.sv** file in an editor
2. Locate the **virtual_reset_sequencer** class at the top of the file.
3. Add a **host_sequencer** handle, call it **h_seqr**
4. Next, locate the **virtual_reset_sequence** class
5. In the class, create a **host_reset_sequence** handle called **h_seq**
6. In the **body()** method, within the **fork-join** structure, add a thread to execute the **host_reset_sequence** with the parent sequencer's **h_seqr** when reset is detected
7. Save and close the file

Task 4. Add Host Agent to Environment

1. Open **router_env.sv** file in an editor
2. Include the **host_agent.sv** file
3. In the class, declare a **host_agent** handle, call it **h_agent**
4. In build phase, construct the **h_agent** object

5. In the connect phase, assign the virtual reset sequencer's **h_seqr** handle to reference the host_agent's (**h_agent**) sequencer (**seqr**)
6. Save and close the file

Task 5. Execute the Host BFM Sequence

1. Open **test_collection.sv** in an editor
2. In **test_base** class, locate the **build_phase()** method
3. In the **build_phase()** method
   - Configure the virtual interface (**"host_io"**) in **h_agent**'s component to use **router_test_top.host_if.agent_vif** interface

   Before you attempt to send packets through the DUT, make sure you can read and write the DUT registers.
4. Locate the **test_host_bfm** class
5. In the **build_phase()** method
   - Turn off all seqencer execution at configure phase and main phase
   - Configure the host agent to execution the **host_bfm_sequence** at the main phase.
6. Save and close the file
7. Verify the DUT registers can be accessed:

```
> make test=test_host_bfm
```

Task 6. Attempt to Drive Packets through DUT

Once the DUT register access is verified, try to execute **test_base** and see if packets can be driver through the DUT.

1. Execute test_base:

```
> make
```

   The expected result is: input packets are seen, but no output packets are observed. This is because the router has been modified to disable all output ports by default. You will need to write to DUT control register to unlock the ports.

   Rather than using a non-RAL sequence to unlock the ports, implement RAL. Then use the RAL abstraction to unlock the ports.

Task 7. Create the Register Abstraction (`.ralf`) File

The first step in RAL implementation is to create an abstraction file to describe the registers. When using Synopsys' **`ralgen`**, the file is a **`.ralf`** file.

1. Open **`host.ralf`** file in an editor
2. Create the RAL register definitions to represent the following set of registers and memory in the **`host.ralf`** file

   Hint: reference the lecture slides

   For this task, just populate the register and memory field definitions. The block definition is done for you in the file. The system definition is not necessary for this lab. (You are verifying the DUT at the block level)

Mode	
ro	Read Only
rw	Read/Write
w1c	Write 1 clears field

Address Map	
HOST_ID	0x0000
PORT_LOCK	0x0100
REG_ARRAY	0x1000-0x10FF
RAM	0x4000-0x4FFF

HOST_ID Register		
Field	CHIP_ID	REV_ID
Bits	15-8	7-0
Mode	ro	ro
Reset	0x5A	0x03

PORT_LOCK Register	
Field	LOCK
Bits	15-0
Mode	w1c
Reset	0xff

REG_ARRAY[256] Registers	
Field	HOST_REG
Bits	15-0
Mode	rw
Reset	0x00

RAM (4K)	
Bits	15-0
Mode	rw

3. Close the file when done

Task 8. Generate UVM Register Classes

1. Convert the RAL file content into UVM register classes with ralgen:

```
> ralgen -uvm -t host_regmodel host.ralf
```

   You should see a file called **`ral_host_regmodel.sv`** created by **`ralgen`**.

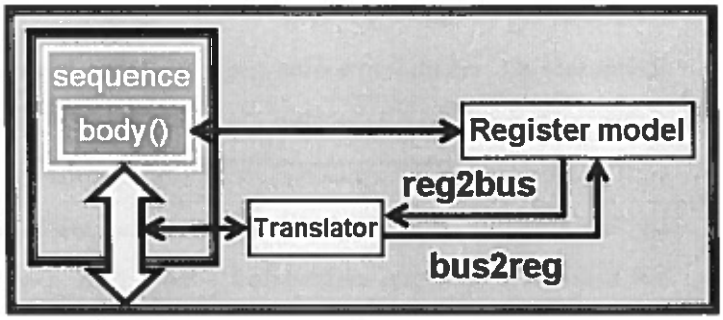
   If interested, take a look at the content. Otherwise, continue on to the next task.

Task 9. Create Sequence Using UVM Registers

1. Open **host_sequence.sv** file in an editor
2. Include the **ral_host_regmodel.sv** file
3. Locate the RAL sequence base class called **host_ral_sequence_base**
4. Inside the class, create an instance of **ral_block_host_regmodel** called **regmodel**
5. In the **pre_start()** method, use **uvm_config_db** to retrieve the register model
6. Locate the RAL test sequence class called **host_ral_test_sequence**
7. Inside the class, define a **body()** task that configures the DUT register with the exact same information as **host_bfm_sequence**, except use UVM register representation rather than direct access.
8. When done, save and close the file.

Task 10. Implement UVM Register Translator

In order for the UVM Register content to be processed correctly by the host driver, you must implement a UVM register to host bus and host bus to UVM register translator.



1. Open the **host_data.sv** file in an editor
2. Locate **class reg_adapter**
3. In the **reg2bus()** method, copy the generic UVM register object (**rw** argument of method) content into a **host_data** object and return the **host_data** handle.
4. In the **bus2reg()** method, check to see that the **uvm_sequence_item** (**bus_item**) from the argument is a **host_data** type. Then, copy the **host_data** object content into the UVM register object (**rw**).

   Reference the lecture slides for exact syntax.
5. When done, save and close the file.

Task 11. Instantiate RAL Model in Environment

1. Open **router_env.sv** in an editor
2. Inside the class
   - Declare a **ral_block_host_regmodel** handle call it **regmodel**
   - Declare a **reg_adapter** handle, call it **adapter**

```
class router_env extends uvm_env;
  `uvm_component_utils(router_env)
  reset_agent              r_agent;
  host_agent               h_agent;
  ral_block_host_regmodel  regmodel;
  reg_adapter              adapter;
```

3. In the build phase construct the adapter object

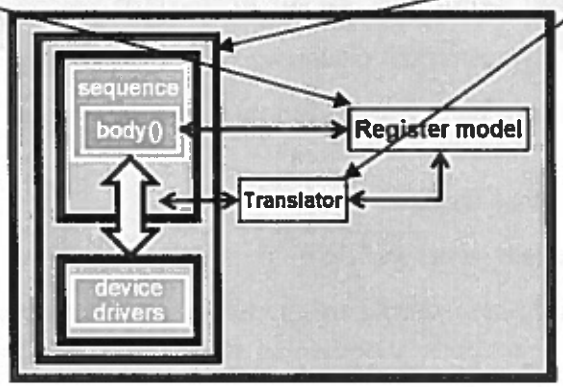
```
  adapter = reg_adapter::type_id::create("adapter", this);
```

4. Continuing in the build phase, use **uvm_config_db** to retrieve regmodel
5. If **regmodel** is null do the following:
   - Add a string called **hdl_path**
   - Retrieve **hdl_path** from **uvm_config_db**. If not found issue warning.
   - Contruct the **regmodel** object
   - Call the **regmodel**'s **build()** method to build the RAL representation
   - Lock **regmodel** and create the address map with **lock_model()** method
   - Set hdl root path using **regmodel**'s **set_hdl_path_root()** method.
6. In all cases, configure the host agent to use the regmodel

```
virtual function void build_phase(uvm_phase phase);  ...
  uvm_config_db #(ral_block_host_regmodel)::get(this,"","regmodel",regmodel);
  if (regmodel == null) begin
    string hdl_path;
    if (!uvm_config_db #(string)::get(this, "", "hdl_path", hdl_path))
      `uvm_warning("HOST_CFG", "HDL path for backdoor not set!");
    regmodel = ral_block_host_regmodel::type_id::create("regmodel", this);
    regmodel.build();
    regmodel.lock_model();
    regmodel.set_hdl_path_root(hdl_path);
  end
  uvm_config_db #(ral_block_host_regmodel)::set(this, h_agent.get_name(),
                                                "regmodel", regmodel);
endfunction
```

7. In connect phase, tie the **regmodel's** address map to a specific sequencer by calling the **set_sequencer()** method. In the argument, pass in sequencer handle (**h_agent.seqr**) and RAL translator handle (**adapter**).

```
virtual function void connect_phase(uvm_phase phase);
  ...
  regmodel.default_map.set_sequencer(h_agent.seqr, adapter);
endfunction
```



8. When done, save and close the file.

Task 12. Enable DPI Backdoor and Create RAL Test

Execute the RAL sequence in a test.

1. Open **test_collection.sv** file in an editor
2. In **test_base** class, set the remaining **hdl_path** for DPI backdoor access
3. Locate the **test_host_ral** class
4. In the build phase, set the host agent's sequencer at the **main_phase** to execute **host_ral_test_sequence**
5. Save and close the file

Task 13. Compile And Simulate

To verify that the RAL implementation works, execute the **host_ral_test_sequence** with **test_host_ral**.

1. Compile and simulate the testbench.

```
> make test=test_host_ral
```

You should see the same results as **test_host_bfm**.

Task 14. Unlock Ports with RAL Sequence

Now that the RAL access is working, use RAL to enable all ports. The sequence is already written for you. It is called **ral_port_unlock_sequence**. You just need to uncomment the code and run this sequence during configure phase to unlock all the ports.

1. Open **host_sequence.sv** file in an editor
2. Locate and uncomment class **ral_port_unlock_sequence**

   Take a look at the content and make sure you understand what this sequence is doing.
3. Save and close the file
4. Open **router_env.sv** file in an editor
5. In the build phase, configure the **host_agent** to execute the **ral_port_unlock_sequence** at the configure phase.
6. Save and close the file

Task 15. Compile And Simulate

1. Compile and simulate the testbench.

   ```
   > make
   ```

   You should see that all packets are successfully processed by the DUT once again.

Task 16. Implement Explicit Predictor

Once you have verified the RAL operation, you should also set up the predictor. The predictor will be needed for the next to last step – self test.

1. Open **router_env.sv** in an editor
2. Inside the class, use **typedef** to create a **uvm_reg_predictor** parameterized to **host_data** called **hreg_predictor**
3. Declare a handle of **hreg_predictor** called **hreg_predict**

```
typedef uvm_reg_predictor #(host_data) hreg_predictor;
hreg_predictor hreg_predict;
```

4. In build phase, construct the predictor object

```
hreg_predict = hreg_predictor::type_id::create("h_reg_predict", this);
```

5. In connect phase
   - Set the map of the predictor to the regmodel's map
   - Set the predictor's adapter to the adapter
   - Connect host agent's analysis port ot the predictor's bus_in analysis port

```
hreg_predict.map = regmodel.get_default_map();
hreg_predict.adapter = adapter;
h_agent.analysis_port.connect(hreg_predict.bus_in);
```

6. Save and close the file
7. Compile and simulate the testbench

   ```
   > make
   ```

   Everything should pass. Now you can also run self-tests in the next task.

Task 17. Executing RAL Self-Test

1. Open **test_collection.sv** in an editor
2. Locate the **test_ral_selftest** class at the bottom of the file
3. Uncomment the code for the RAL self test

```
class test_ral_selftest extends test_base;
 `uvm_component_utils(test_ral_selftest)
 string                  seq_name="uvm_reg_bit_bash_seq";
 uvm_reg_sequence        selftest_seq;
 virtual_reset_sequence v_reset_seq;
 function new(string name, uvm_component parent);
   super.new(name, parent);
 endfunction
 virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  uvm_config_db #(uvm_object_wrapper)::set(this,"*","default_sequence",null);
 endfunction
 virtual task run_phase(uvm_phase phase);
  phase.raise_objection(this, "Starting reset tests");
  v_reset_seq = virtual_reset_sequence::type_id::create("v_reset_seq", this);
  v_reset_seq.start(env.v_reset_seqr);
  clp.get_arg_value("+seq=", seq_name);
  $cast(selftest_seq, factory.create_object_by_name(seq_name));
  selftest_seq.model = env.regmodel;
  selftest_seq.start(env.h_agent.seqr);
  phase.drop_objection(this, "Done with register tests");
 endtask
endclass
```

4. Save and close the file
5. Compile and simulate the testbench

```
> make test=test_ral_selftest
```

You should see that each bit of the registers is verified.

Note that through the seq run-time switch, you can pick any RAL test sequence to execute without re-compilation like the following:

```
> make test=test_ral_selftest seq=uvm_reg_hw_reset_seq
```

Task 18. Turn on Functional Coverage

The **ralgen** utility is capable of creating functional coverage for the DUT registers. The user guide is at: $VCS_HOME/doc/UserGuide/pdf/uvm_ralgen_ug.pdf. There are three types of functional coverage that ralgen can create: Register Bits coverage, Address Map coverage and Field Values coverage. The switches are: -c b for Register Bits coverage; -c a for Address Map coverage; -c f for Field Value coverage.

For this lab, try the Register Bits Coverage.

1. Re-run ralgen to add Register Bits coverage:

```
> ralgen -uvm -c b -t host_regmodel host.ralf
```

ralgen created functional coverage. You need to enable them in the test.

2. Open **test_collection.sv** in an editor
3. In **test_base**'s **end_of_elaboration_phase()** method (to allow for all structural changes to be completed), turn on coverage for the **regmodel**:

```
env.regmodel.set_coverage(UVM_CVR_ALL);
```

4. Open **test.sv** in an editor
5. In the **initial** block the following is done for you to enable RAL coverage:

```
uvm_reg::include_coverage("*", UVM_CVR_ALL);
```

6. Compile and simulate the testbench

```
> make
```

7. Generate coverage report:

```
> make cover
```

8. Take a look at the coverage report:

```
> firefox urgReport/groups.html &
```

Congratulations, you have completed Lab 7!

Answers / Solutions

test.sv Solution:

```
program automatic test;
import uvm_pkg::*;
`include "test_collection.sv"
initial begin
  $timeformat(-9, 1, "ns", 10);
  uvm_reg::include_coverage("*", UVM_CVR_ALL); // Required for RAL
  run_test();
end
endprogram
```

test_collection.sv Solution:

```
class test_base extends uvm_test;
  `uvm_component_utils(test_base)

  router_env env;
  // For convenience, access to the command line processor is done for
you
  uvm_cmdline_processor clp = uvm_cmdline_processor::get_inst();

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    env = router_env::type_id::create("env", this);

    uvm_config_db#(virtual router_tb_io)::set(this, "env.i_agent[*]",
"sigs", router_test_top.router_if.agent_vif);
    uvm_config_db#(virtual router_tb_io)::set(this, "env.o_agent[*]",
"sigs", router_test_top.router_if.agent_vif);
    uvm_config_db#(virtual reset_tb_io)::set(this, "env.r_agent", "sigs",
router_test_top.reset_if.agent_vif);

    uvm_config_db#(virtual host_tb_io)::set(this, "env.h_agent", "sigs",
router_test_top.host_if.agent_vif);

    uvm_config_db #(string)::set(this, "env", "hdl_path",
"router_test_top.dut");
  endfunction

  virtual function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    env.regmodel.set_coverage(UVM_CVR_ALL);
  endfunction

  virtual task main_phase(uvm_phase phase);
```

```
    uvm_objection objection;
    super.main_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

    objection = phase.get_objection();
    objection.set_drain_time(this, 1us);
  endtask

  virtual function void final_phase(uvm_phase phase);
    super.final_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    uvm_top.print_topology();
    factory.print();
  endfunction
endclass

class test_host_bfm extends test_base;
  `uvm_component_utils(test_host_bfm)
  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

    uvm_config_db #(uvm_object_wrapper)::set(this,
"env.*.configure_phase", "default_sequence", null);
    uvm_config_db #(uvm_object_wrapper)::set(this, "env.*.main_phase",
"default_sequence", null);
    uvm_config_db #(uvm_object_wrapper)::set(this,
"env.h_agent.seqr.main_phase", "default_sequence",
host_bfm_sequence::get_type());
  endfunction
endclass

class test_host_ral extends test_base;
  `uvm_component_utils(test_host_ral)
  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

    uvm_config_db #(uvm_object_wrapper)::set(this,
"env.*.configure_phase", "default_sequence", null);
    uvm_config_db #(uvm_object_wrapper)::set(this, "env.*.main_phase",
"default_sequence", null);
    uvm_config_db #(uvm_object_wrapper)::set(this,
"env.h_agent.seqr.main_phase", "default_sequence",
host_ral_test_sequence::get_type());
  endfunction
endclass
```

```
class test_ral_selftest extends test_base;
  `uvm_component_utils(test_ral_selftest)
  string                  seq_name="uvm_reg_bit_bash_seq";
  uvm_reg_sequence        selftest_seq;
  virtual_reset_sequence v_reset_seq;

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    uvm_config_db
#(uvm_object_wrapper)::set(this,"*","default_sequence",null);
  endfunction

  virtual task run_phase(uvm_phase phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    phase.raise_objection(this, "Starting reset tests");
    v_reset_seq = virtual_reset_sequence::type_id::create("v_reset_seq",
this);
    v_reset_seq.start(env.v_reset_seqr);
    clp.get_arg_value("+seq=", seq_name);
    $cast(selftest_seq, factory.create_object_by_name(seq_name));
    selftest_seq.model = env.regmodel;
    selftest_seq.start(env.h_agent.seqr);
    phase.drop_objection(this, "Done with register tests");
  endtask
endclass
```

host_data.sv Solution:

```
class host_data extends uvm_sequence_item;
  typedef enum {READ, WRITE} kind_e;
  typedef enum {IS_OK, NOT_OK, HAS_X} status_e;
  rand kind_e    kind;
  rand status_e  status;
  rand bit[15:0] addr;
  rand bit[15:0] data;
  `uvm_object_utils_begin(host_data)
    `uvm_field_int(addr, UVM_ALL_ON)
    `uvm_field_int(data, UVM_ALL_ON)
    `uvm_field_enum(kind_e, kind, UVM_ALL_ON)
    `uvm_field_enum(status_e, status, UVM_ALL_ON)
  `uvm_object_utils_end
  constraint valid { addr inside {'h0, 'h100, ['h1000:'h10ff],
['h4000:'h4ffff]}; }
  function new(string name="host_data");
    super.new(name);
    status.rand_mode(0);
  endfunction
endclass

class reg_adapter extends uvm_reg_adapter;
  `uvm_object_utils(reg_adapter)
  function new(string name="reg_adapter");
    super.new(name);
  endfunction
  virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
    host_data tr;
    tr = host_data::type_id::create("tr");
    tr.kind = (rw.kind inside {UVM_READ, UVM_BURST_READ}) ? host_data::READ :
host_data::WRITE;
    tr.addr = rw.addr;
    tr.data = rw.data;
    return tr;
  endfunction

  virtual function void bus2reg(uvm_sequence_item bus_item, ref
uvm_reg_bus_op rw);
    host_data tr;
    if (!$cast(tr, bus_item)) begin
      `uvm_fatal("NOT_HOST_REG_TYPE", "bus_item is not correct type");
    end
    rw.kind = (tr.kind == host_data::READ) ? UVM_READ : UVM_WRITE;
    rw.addr = tr.addr;
    rw.data = tr.data;
    case (tr.status)
      host_data::IS_OK:  rw.status = UVM_IS_OK;
      host_data::NOT_OK: rw.status = UVM_NOT_OK;
      host_data::HAS_X:  rw.status = UVM_HAS_X;
      default: begin `uvm_fatal("RAL_STATUS", $sformatf("Unsupported status:
%p", tr.status)); end
    endcase
  endfunction
endclass
```

host_sequence.sv Solution:

```
class host_sequence_base extends uvm_sequence #(host_data);
  `uvm_object_utils(host_sequence_base)
  virtual host_tb_io sigs;
  uvm_sequencer_base p_seqr;
  function new(string name = "host_sequence_base");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction
  virtual task pre_start();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    p_seqr = get_sequencer();
    if ((get_parent_sequence() == null) && (starting_phase != null)) begin
      starting_phase.raise_objection(this);
    end
    if (uvm_config_db#(virtual host_tb_io)::get(p_seqr.get_parent(), "",
"sigs", sigs)) begin
      `uvm_info("HOST_SEQ_CFG", "Has access to host interface", UVM_HIGH);
    end
  endtask
  virtual task post_start();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if ((get_parent_sequence() == null) && (starting_phase != null)) begin
      starting_phase.drop_objection(this);
    end
  endtask
endclass
class host_reset_sequence extends host_sequence_base;
  `uvm_object_utils(host_reset_sequence)
  function new(string name = "host_reset_sequence");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction
  virtual task body();
    sigs.wr_n = 1'b1;
    sigs.rd_n = 1'b1;
    sigs.address ='z;
    sigs.data = 'z;
  endtask
endclass
class host_bfm_sequence extends host_sequence_base;
  `uvm_object_utils(host_bfm_sequence)
  function new(string name = "host_bfm_sequence");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction
   virtual task body();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    `uvm_do_with(req, {addr == 'h0; kind == host_data::READ;});
    if (req.data != 'h5a03) begin
```

```
      `uvm_fatal("BFM_ERR", $sformatf("HOST_ID is %4h instead of 'h5a03",
req.data));
    end else begin
      `uvm_info("BFM_TEST", $sformatf("HOST_ID is %4h the expected value is
'h5a03", req.data), UVM_MEDIUM);
    end

    `uvm_do_with(req, {addr == 'h100; kind == host_data::READ;});
    if (req.data != '1) begin
      `uvm_fatal("BFM_ERR", $sformatf("LOCK is %4h instead of 'hffff",
req.data));
    end
    `uvm_do_with(req, {addr == 'h100; data == '1; kind ==
host_data::WRITE;});
    `uvm_do_with(req, {addr == 'h100; kind == host_data::READ;});
    if (req.data != '0) begin
      `uvm_fatal("BFM_ERR", $sformatf("LOCK is %4h instead of 'h0000",
req.data));
    end else begin
      `uvm_info("BFM_TEST", $sformatf("LOCK is %4h the expected value is
'h0000", req.data), UVM_MEDIUM);
    end

    for (int i=0; i<256; i++) begin
      `uvm_do_with(req, {addr == 'h1000+i; data == (i ^ (i >> 1)); kind ==
host_data::WRITE;});
    end
    for (int i=0; i<256; i++) begin
      `uvm_do_with(req, {addr == 'h1000+i; kind == host_data::READ;});
      if (req.data != (i ^ (i >> 1))) begin
        `uvm_fatal("BFM_ERR", $sformatf("R_ARRAY is %4h instead of %4h",
req.data, i ^ (i >> 1)));
      end
    end
    `uvm_info("BFM_TEST", "R_ARRAY contains the expected values",
UVM_MEDIUM);

    for (int i=0; i<4096; i++) begin
      `uvm_do_with(req, {addr == 'h4000+i; data == 16'b1 << i%16; kind ==
host_data::WRITE;});
    end
    for (int i=0; i<4096; i++) begin
      `uvm_do_with(req, {addr == 'h4000+i; kind == host_data::READ;});
      if (req.data != 16'b1 << i%16) begin
        `uvm_fatal("BFM_ERR", $sformatf("R_ARRAY is %4h instead of %4h",
req.data, 16'b1 << i%16));
      end
    end
    `uvm_info("BFM_TEST", "RAM contains the expected values", UVM_MEDIUM);
  endtask
endclass

class host_ral_sequence_base extends uvm_reg_sequence #(host_sequence_base);
  `uvm_object_utils(host_ral_sequence_base)
  ral_block_host_regmodel regmodel;

  function new(string name = "host_ral_sequence_base");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction
```

```
  virtual task pre_start();
    super.pre_start();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if (!uvm_config_db#(ral_block_host_regmodel)::get(p_seqr.get_parent(),
"", "regmodel", regmodel)) begin
      `uvm_info("RAL_CFG", "regmodel not set through configuration.  Make
sure it is set by other mechanisms", UVM_MEDIUM);
    end
    if (regmodel == null) begin
      `uvm_fatal("RAL_CFG", "regmodel not set");
    end
  endtask
endclass

class host_ral_test_sequence extends host_ral_sequence_base;
  `uvm_object_utils(host_ral_test_sequence)
  function new(string name = "host_ral_test_sequence");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction
  virtual task body();
    uvm_status_e status;
    uvm_reg_data_t data;
    regmodel.HOST_ID.read(.status(status), .value(data), .path(UVM_BACKDOOR),
.parent(this));
    if (data != 'h5a03) begin
      `uvm_fatal("RAL_ERR", $sformatf("HOST_ID is %4h instead of 'h5a03",
data));
    end else begin
      `uvm_info("RAL_TEST", $sformatf("HOST_ID is %4h the expected value is
'h5a03", data), UVM_MEDIUM);
    end
    regmodel.LOCK.read(.status(status), .value(data), .path(UVM_BACKDOOR),
.parent(this));
    if (data != 'hffff) begin
      `uvm_fatal("RAL_ERR", $sformatf("LOCK is %4h instead of 'hffff",
data));
    end
    regmodel.LOCK.write(.status(status), .value('1), .path(UVM_FRONTDOOR),
.parent(this));
    regmodel.LOCK.read(.status(status), .value(data), .path(UVM_BACKDOOR),
.parent(this));
    if (data != '0) begin
      `uvm_fatal("RAL_ERR", $sformatf("LOCK is %4h instead of 'h0000",
data));
    end else begin
      `uvm_info("RAL_TEST", $sformatf("LOCK is %4h the expected value is
'h0000", data), UVM_MEDIUM);
    end
    for (int i=0; i<256; i++) begin
      regmodel.R_ARRAY[i].write(.status(status), .value(i ^ (i >> 1)),
.path(UVM_FRONTDOOR), .parent(this));
    end
```

```
    for (int i=0; i<256; i++) begin
      regmodel.R_ARRAY[i].read(.status(status), .value(data),
.path(UVM_BACKDOOR), .parent(this));
      if (data != (i ^ (i >> 1))) begin
        `uvm_fatal("RAL_ERR", $sformatf("R_ARRAY is %4h instead of %4h",
data, i ^ (i >> 1)));
      end
    end
    `uvm_info("RAL_TEST", "R_ARRAY contains the expected values",
UVM_MEDIUM);
    for (int i=0; i<4096; i++) begin
      regmodel.RAM.write(.status(status), .offset(i), .value(16'b1 << i%16),
.path(UVM_FRONTDOOR), .parent(this));
    end
    for (int i=0; i<4096; i++) begin
      regmodel.RAM.read(.status(status), .offset(i), .value(data),
.path(UVM_BACKDOOR), .parent(this));
      if (data != 16'b1 << i%16) begin
        `uvm_fatal("RAL_ERR", $sformatf("RAM is %4h instead of %4h", data,
16'b1 << i%16));
      end
    end
    `uvm_info("RAL_TEST", "RAM contains the expected values", UVM_MEDIUM);
  endtask
endclass

class ral_port_unlock_sequence extends host_ral_sequence_base;
  `uvm_object_utils(ral_port_unlock_sequence)
  function new(string name = "ral_port_unlock_sequence");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction
  virtual task body();
    uvm_status_e status;
    uvm_reg_data_t data;
    regmodel.LOCK.write(.status(status), .value('1), .path(UVM_FRONTDOOR),
.parent(this));
  endtask
endclass
```

host.ralf Solution:

```
register HOST_ID {
  field REV_ID {
    bits 8;
    access ro;
    reset 'h03;
  }
  field CHIP_ID {
    bits 8;
    access ro;
    reset 'h5A;
  }
}

register PORT_LOCK {
  field LOCK {
    bits 16;
    access wlc;
    reset 'hffff;
  }
}

register REG_ARRAY {
  field USER_REG {
    bits 16;
    access rw;
    reset 'h0;
  }
}

memory RAM {
  size 4k;
  bits 16;
  access rw;
}

block host_regmodel {
  bytes 2;
  register HOST_ID          (host_id)       @'h0000;
  register PORT_LOCK        (lock)          @'h0100;
  register REG_ARRAY[256] (host_reg[%d])  @'h1000; # array must specify HDL
index
  memory   RAM              (ram)           @'h4000;
}
```

router_env.sv Solution:

```
class router_env extends uvm_env;
  reset_agent r_agent;
  input_agent i_agent[16];
  output_agent o_agent[16];
  scoreboard sb;
  virtual_reset_sequencer  v_reset_seqr;

  host_agent  h_agent;
  ral_block_host_regmodel regmodel;
  reg_adapter adapter;

  typedef uvm_reg_predictor #(host_data) hreg_predictor;
  hreg_predictor hreg_predict;

  `uvm_component_utils(router_env)

  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

    r_agent = reset_agent::type_id::create("r_agent", this);
    uvm_config_db #(uvm_object_wrapper)::set(this,
"r_agent.seqr.reset_phase", "default_sequence", null);

    foreach (i_agent[i]) begin
      i_agent[i] = input_agent::type_id::create($sformatf("i_agent[%0d]", i),
this);
      uvm_config_db #(int)::set(this, i_agent[i].get_name(), "port_id", i);
      uvm_config_db #(uvm_object_wrapper)::set(this, {i_agent[i].get_name(),
".", "seqr.reset_phase"}, "default_sequence", null);
      uvm_config_db #(uvm_object_wrapper)::set(this, {i_agent[i].get_name(),
".", "seqr.main_phase"}, "default_sequence", packet_sequence::get_type());
    end

    uvm_config_db #(uvm_object_wrapper)::set(this,
"v_reset_seqr.reset_phase", "default_sequence",
virtual_reset_sequence::get_type());

    sb = scoreboard::type_id::create("sb", this);

    foreach (o_agent[i]) begin
      o_agent[i] = output_agent::type_id::create($sformatf("o_agent[%0d]",
i), this);
      uvm_config_db #(int)::set(this, o_agent[i].get_name(), "port_id", i);
    end

    v_reset_seqr = virtual_reset_sequencer::type_id::create("v_reset_seqr",
this);
```

```
    h_agent = host_agent::type_id::create("h_agent", this);
    adapter = reg_adapter::type_id::create("adapter", this);
    uvm_config_db #(ral_block_host_regmodel)::get(this, "", "regmodel",
regmodel);

    if (regmodel == null) begin
      string hdl_path;
      `uvm_info("HOST_CFG", "Self constructing regmodel", UVM_MEDIUM);
      if (!uvm_config_db #(string)::get(this, "", "hdl_path", hdl_path))
begin
        `uvm_warning("HOST_CFG", "HDL path for DPI backdoor not set!");
      end
      regmodel = ral_block_host_regmodel::type_id::create("regmodel", this);
      regmodel.build();
      regmodel.lock_model();
      regmodel.set_hdl_path_root(hdl_path);
    end

    uvm_config_db #(ral_block_host_regmodel)::set(this, h_agent.get_name(),
"regmodel", regmodel);

    uvm_config_db #(uvm_object_wrapper)::set(this, {h_agent.get_name(), ".",
"seqr.configure_phase"}, "default_sequence",
ral_port_unlock_sequence::get_type());

    hreg_predict = hreg_predictor::type_id::create("h_reg_predict", this);
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    foreach (i_agent[i]) begin
      i_agent[i].analysis_port.connect(sb.before_export);
      v_reset_seqr.pkt_seqr.push_back(i_agent[i].seqr);
    end

    foreach (o_agent[i]) begin
      o_agent[i].analysis_port.connect(sb.after_export);
    end

    v_reset_seqr.r_seqr = this.r_agent.seqr;
    v_reset_seqr.h_seqr = this.h_agent.seqr;

    regmodel.default_map.set_sequencer(h_agent.seqr, adapter);
    hreg_predict.map = regmodel.get_default_map();
    hreg_predict.adapter = adapter;
    h_agent.analysis_port.connect(hreg_predict.bus_in);
  endfunction
endclass
```

virtual_reset_sequence.sv Solution:

```
class virtual_reset_sequencer extends uvm_sequencer;
  `uvm_component_utils(virtual_reset_sequencer)
  reset_sequencer  r_seqr;
  packet_sequencer pkt_seqr[$];
  host_sequencer h_seqr;
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
endclass

class virtual_reset_sequence extends uvm_sequence;
  `uvm_object_utils(virtual_reset_sequence)
  `uvm_declare_p_sequencer(virtual_reset_sequencer)

  reset_sequence          r_seq;
  driver_reset_sequence   d_seq;
  uvm_event reset_event = uvm_event_pool::get_global("reset");
  host_reset_sequence     h_seq;

  function new(string name="virtual_reset_sequence");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction

  virtual task pre_start();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if ((get_parent_sequence() == null) && (starting_phase != null)) begin
      starting_phase.raise_objection(this);
    end
  endtask

  virtual task body();
    fork
      `uvm_do_on(r_seq, p_sequencer.r_seqr);
      foreach (p_sequencer.pkt_seqr[i]) begin
        int j = i;
        fork
          begin
            reset_event.wait_on();
            `uvm_do_on(d_seq, p_sequencer.pkt_seqr[j]);
          end
        join_none
      end
      begin
        reset_event.wait_on();
        `uvm_do_on(h_seq, p_sequencer.h_seqr);
      end
    join
  endtask

  virtual task post_start();
    if ((get_parent_sequence() == null) && (starting_phase != null)) begin
      starting_phase.drop_objection(this);
    end
  endtask
endclass
```