SYNOPSYS®

CUSTOMER EDUCATION SERVICES

SystemVerilog Verification UVM Workshop

Lab Guide

40-I-055-SLG-004 2017.03

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

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

www.synopsys.com

Copyright Notice and Proprietary Information

Copyright © 2017 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

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-055-SLG-004
SystemVerilog Verification UVM Lab Guide

1

UVM Test & Messages

Learning Objectives

After completing this lab, you should be able to:

- Create a simple UVM test
- Embed report messages
- Compile the test
- Run the simulation and observe results
- Add environment to test
- Compile and run simulation to observe results

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:

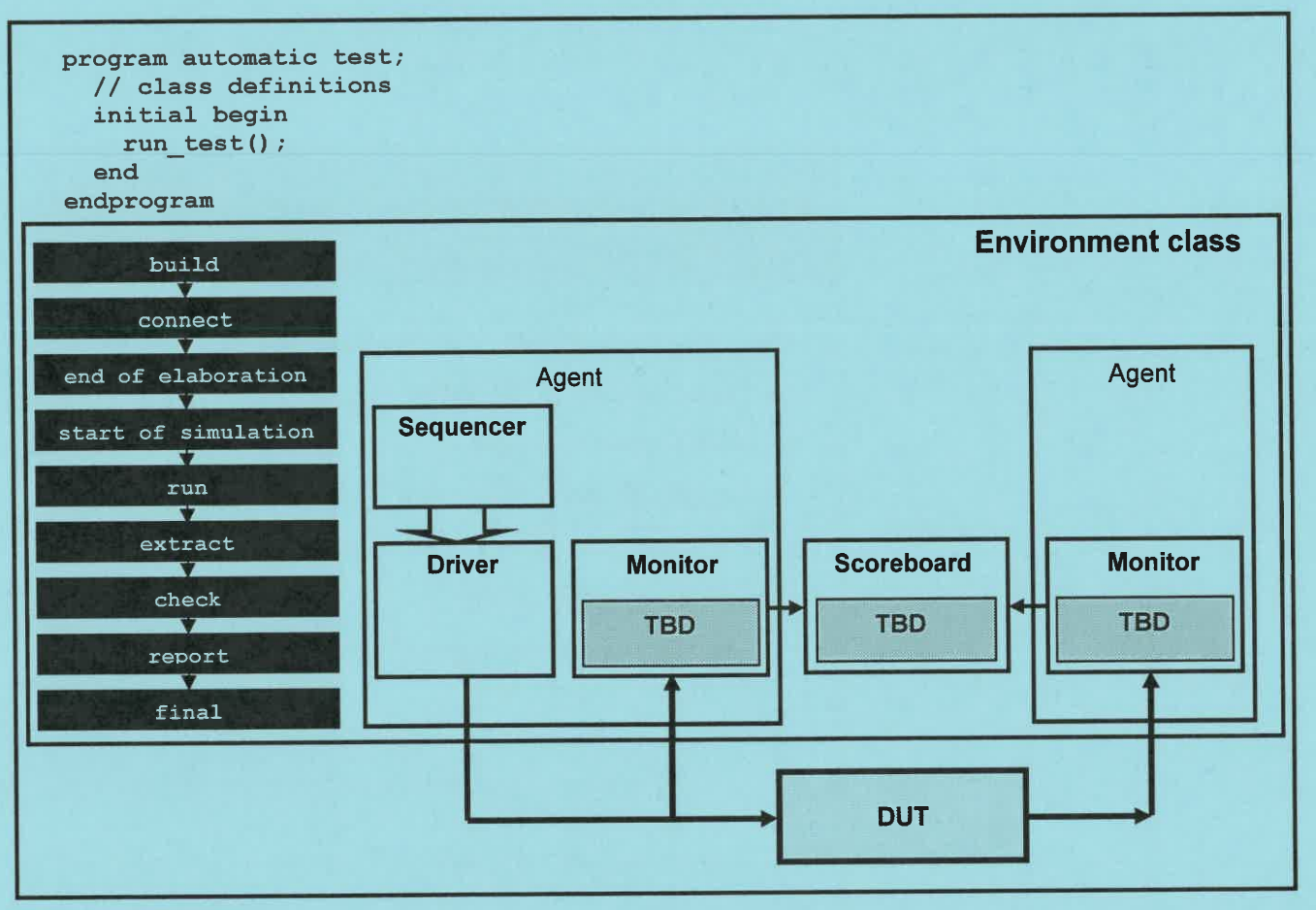


Figure 1. Lab 1 Testbench Architecture

Lab Overview

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

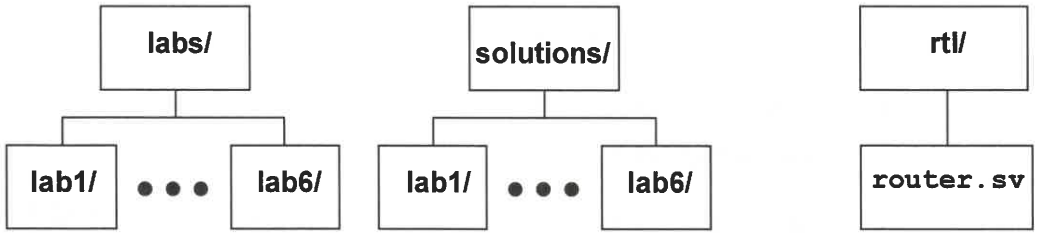


Figure 2. Lab Directory Structure

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 flip through to end of the lab to find solution code.

The work flow for this lab is illustrated as follows.

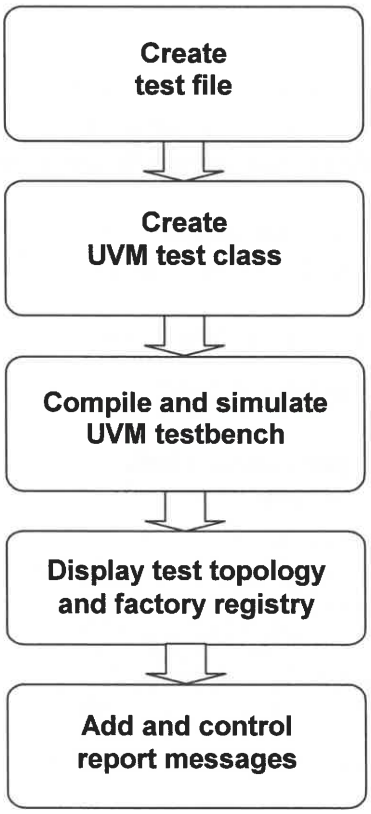


Figure 3. Lab 1 Flow Diagram

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
   ```

   Every test in Verilog has an entry point. This is done in an **initial** block.

   The **initial** block can be coded inside either a **program** or a **module**. Executing testbench code in **program** block will ensure better protection against race conditions than executing the code in **module**. For this reason, in the labs, we are going to run the testbench code in a **program**.

   Inside the **program** block, the **import uvm_pkg::*;** statement enables access to all the UVM features. The three most important elements of the **uvm_pkg** are:

   1 – the UVM execution manager (accessed via **uvm_root::get()**)
   2 – the UVM factory (accessed via **uvm_factory::get()**)
   3 – the UVM configuration database (**uvm_config_db**).

   To start UVM execution, you must start the UVM execution manager by calling the **run_test()** method in an **intial** block.

2. Open a new file, call it **test.sv**

3. Enter the following test code:

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

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

endprogram
```

4. Compile this simple UVM testbench using vcs:

   ```
   > vcs -sverilog -ntb_opts uvm-1.2 test.sv
   ```

   The **-sverilog** switch enables VCS to recognize SystemVerilog code. The **-ntb_opts uvm-1.2** switch enables VCS to look for the **uvm_pkg** in the VCS installation directory. The **test.sv** file is the file that you just created.

5. Simulate this UVM testbench and store the output in simv.log file:

```
> simv -l simv.log
```

You should see the following fatal error:

```
UVM_FATAL @ 0.0ns: reporter [NOCOMP] No components instantiated. You must
either instantiate at least one component before calling run_test or use
run_test to do so. To run a test using run_test, use +UVM_TESTNAME or
supply the test name in the argument to run_test(). Exiting simulation.
...
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :    1
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    1
** Report counts by id
[NOCOMP]     1
[UVM/RELNOTES]     1
```

There are two reason for this fatal error:

First, you don't have any UVM test class.

Second, even if you had a UVM test class, you did not specify the test to be executed via the **+UVM_TESTNAME** switch.

You will correct this by creating the test class first.

6. Open a new file, call it **test_collection.sv**. Enter the following code:

```
class test_base extends uvm_test;
  `uvm_component_utils(test_base)
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
endclass
```

Some explanations are needed for this code.

The test class (**test_base**) is created by extending from the **uvm_test** base class.

```
class test_base extends uvm_test;
  ...
endclass
```

As was mentioned earlier, there are three key elements of the UVM package that you must understand and apply them to your UVM testbench.

One of them is the UVM factory.

Every UVM class that you create must be registered into this UVM factory. The UVM package provides a macro to simplify this process. The macro for a component class is: **`uvm_component_utils**. The name that you enter into the argument of the macro must match the class name:

You will see the effects and benefits of registering user classes into the UVM factory in the next lab (lab2). For now, consider this as a requirement.

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

Once you registered a class into the UVM factory, you will be using the UVM factory to construct UVM testbench objects. The UVM factory requires that you define the constructor **new()** for component classes in a very specific way:

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

There must only be two arguments to the constructor – a **string** field for the instance name of the object and a **uvm_component** field for the object's structural parent. These two fields must be passed on to the base class via the **super.new()** call. Again, this is a requirement that you must obey in order to use the UVM factory.

7. Save and close the file.

   Once you have the test class defined, you will need to include this file to be compiled as part of your **test.sv** code. (A better alternative is to embed the code inside a SystemVerilog package, then import the package into the program file. This will be done for you in subsequent labs.)

8. Open the **test.sv** file in an editor.

9. Add the following include statement:

```
program automatic test;
import uvm_pkg::*;
`include "test_collection.sv"
initial begin
  ...
end
endprogram
```

Compile and simulate this code again. This time, use the **+UVM_TESTNAME** switch to specify the test for the UVM execution manager (one of the three key elements of the UVM package) to run.

10. Compile & simulate:

```
> vcs -sverilog -ntb_opts uvm-1.2 test.sv
> simv -l simv.log +UVM_TESTNAME=test_base
```

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

```
UVM_INFO @ 0.0ns: reporter [RNTST] Running test test_base...
--- UVM Report Summary ---
** Report counts by severity
UVM_INFO :    2
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
```

Test as specified by **+UVM_TESTNAME**

Before proceeding, let's make sure you understand what's happening here.

Question 1. Which one of the three key UVM elements of the UVM package is responsible for reading the **+UVM_TESTNAME** switch?

.....

Question 2. What code in the **test.sv** file caused this key element to run and look at the **+UVM_TESTNAME** switch?

.....

Question 3. How does this key element make use of the test class name provided via the **+UVM_TESTNAME** switch?

.....

The object in UVM that reads the **+UVM_TESTNME** switch is the UVM execution manager (**uvm_root**) which you started via **run_test()**. It makes use of the UVM factory to create the top test object from the class name provided.

This top test object is called **uvm_test_top**. The **uvm_test_top** object sits at the top of the entire UVM test hierarchy. It is the root parent of all your UVM test components.

Since the UVM execution manager is the creator of the test object, it is also aware of the entire UVM component structural hierarchy from test on down.

The UVM execution manager is a singleton object of the **uvm_root** class. You can retrieve the handle to this manger by calling **uvm_root::get()**. You can use this handle to print the test structural topology.

11. Open the **test_collection.sv** file in an editor.

12. Add the following start of simulation phase method to the **test_base** class:

```
class test_base extends uvm_test;
  `uvm_component_utils(test_base)
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
  virtual function void start_of_simulation_phase(uvm_phase phase);
    super.start_of_simulation_phase(phase);
    uvm_root::get().print_topology();
  endfunction
endclass
```

13. Compile & simulate this code (a **Makefile** is provided to simply typing):

```
> make
```

You should now see the test topology printed:

Topology printed via UVM execution manager

```
UVM_INFO … @ 0.0ns: reporter [UVMTOP] UVM testbench topology:
-----------------------------------
Name                Type          Size  Value
-----------------------------------
uvm_test_top        test_base     -     @344
 recording_detail   uvm_verbosity 32    UVM_FULL
-----------------------------------
```

Unique instance id

Recording verbosity manage with **uvm_config_db** Controls recording for viewing in debugger's Waveform window

Instance name – source is the first argument of the constructor method when the object is created.

Class type of the test instance

At this point, you only have the **test_base** class object called **uvm_test_top**. So, that's all you see.

As your test bench becomes more complex, you will find this set of information to be very helpful during debugging sessions.

It is also useful for debugging to see all the user classes registered in the UVM factory. You can get this information by doing the following steps.

14. Open the **test_collection.sv** file in an editor.

15. Add a statement to print factory registry:

```
virtual function void start_of_simulation_phase(uvm_phase phase);
  super.start_of_simulation_phase(phase);
  uvm_root::get().print_topology();
  uvm_factory::get().print();
endfunction
```

uvm_factory::get() returns the handle to the UVM factory singleton object. Calling the **print()** method of the singleton object displays the user registered classes.

16. Compile & simulate :

```
> make
```

You should now see the factory registry content:

```
#### Factory Configuration (*)

  No instance or type overrides are registered with this factory

All types registered with the factory: 57 total
  Type Name
  ---------
  snps_uvm_reg_bank_group
  snps_uvm_reg_map
  test_base
(*) Types with no associated type name will be printed as <unknown>

####
```

uvm_factory::get().print() output

Factory overrides will be covered in next lab

Registered classes without a type name are intentionally hidden

Synopsys enhancement classes (always start with **snps_**)

User registered classes

Task 2. Debugging Essentials

One common problem encountered when running a test is hung simulation. Simulation starts then hangs at some point during simulation. The cause is usually due to a zero-time infinite loop or a broken (unconnected) communication link or a dead locked code.

To debug this, it is helpful to find out at what point did the simulation hang and how did the simulation get to that point. You can embed a trace statement as the first executeable statement (after calling the super method) of a method to enable this.

1. Open the **test_collection.sv** file with an editor
2. Add the following UVM report messages:

```
class test_base extends uvm_test;
  `uvm_component_utils(test_base)
  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction
  virtual function void start_of_simulation_phase(uvm_phase phase);
    super.start_of_simulation_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    uvm_root::get().print_topology();
    uvm_factory::get().print();
  endfunction
endclass
```

In the UVM report message macro, the first argument is the **ID** of the message. Choose it carefully by picking a brief name that is likely to be unique. It will be very useful when you need to control the report messages to display. In this code, we want to clearly show that the message is intended for tracing the execution of all user methods. Thus, the **ID** is set to **TRACE**.

It is helpful in debugging to know the method name and the file context. The second argument, **$sformatf("%m")**, will give you this information.

The third argument of the report message macro is the verbosity of the message. By setting it to **UVM_HIGH**, this message will be filtered out in normal execution (only **UVM_NONE**, **UVM_LOW** and **UVM_MEDIUM** messages are displayed by default). But, you will be able to enable the message via a run-time switch during debugging sessions.

3. Compile & simulate this code

```
> make
```

You should see the same display as before. None of the added **TRACE** report messages are displayed.

4. Run the simulation with verbosity set to **UVM_HIGH**:

```
> make verbosity=UVM_HIGH
```

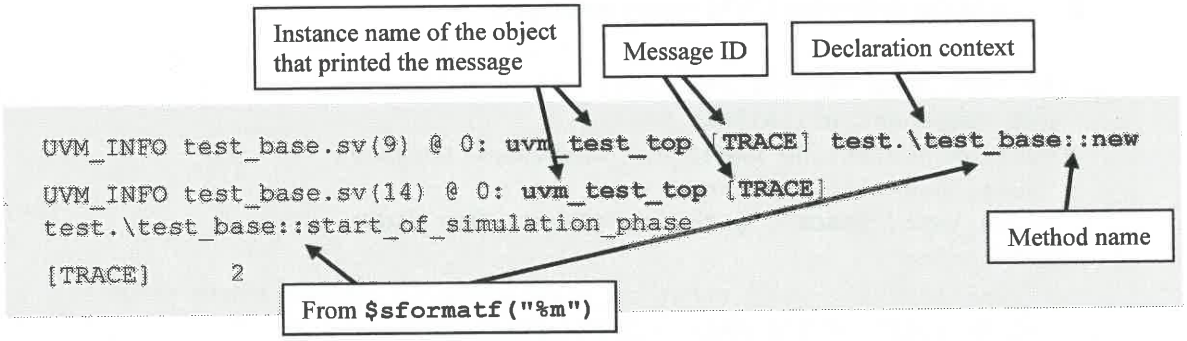
This sets the run-time switch **+UVM_VERBOSITY=UVM_HIGH**.

Do you see the messages? Hard to read?

5. Try this:

```
> grep TRACE simv.log
```

Now, you can clearly see that the only thing that happened during this test run was that the **test_base** instance **uvm_test_top** was constructed and its **start_of_simulation_phase** method was executed afterwards.



There are other ways you can make use of the **ID** field.

You may want to turn on **TRACE** report messages without affecting other report messages. This can be done via the **+uvm_set_verbosity** switch.

6. Run and save the simulation log:

```
> make verbosity=UVM_DEBUG log=simv.log.debug
```

Look at the simulation result in the **simv.log.debug** file. You will see that every report message is captured. This, in a typical debugging session, will overwhelm you with too much data. Debugging then becomes very difficult.

You can use the **+uvm_set_verbosity** switch to apply the verbosity to only the area of debugging that you are interested in.

7. Run the simulation with verbosity set to **UVM_DEBUG** only for the **TRACE** id:

```
> make plus=uvm_set_verbosity=\*,TRACE,UVM_DEBUG,time,0
```

Note: **plus** is the mechanism built into the **Makefile** to allow the user to specify any run-time switches.

8. Compare the two simulation results:

```
> tkdiff simv.log.debug simv.log
```

By using the **uvm_set_verbosity** switch and selecting the **ID** to display, you can control and manage the UVM report messages without being overwhelmed.

Recall these statements in the **start_of_simulation_phase** method:

```
uvm_root::get().print_topology();
uvm_factory::get().print();
```

You may want to control these two print statements also. Unfortunately, these print statements are not easily filtered out.

What you can do, though, is to wrap the code you want to control around a user created report message.

9. Open the **test_collection.sv** file with an editor

10. Wrap the print statements as user defined UVM report messages:

```
class test_base extends uvm_test;
  `uvm_component_utils(test_base)
  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction
  virtual function void start_of_simulation_phase(uvm_phase phase);
    super.start_of_simulation_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if (uvm_report_enabled(UVM_MEDIUM, UVM_INFO, "TOPOLOGY")) begin
      uvm_root::get().print_topology();
    end
    if (uvm_report_enabled(UVM_MEDIUM, UVM_INFO, "FACTORY")) begin
      uvm_factory::get().print();
    end
  endfunction
endclass
```

11. Compile & simulate the modified code:

```
> make
```

Again, you should see the same display as before. The topology and the factory registry are printed. The **TRACE** report messages are not displayed.

12. Simulate with **TOPOLOGY** verbosity at **UVM_NONE** everything else at **UVM_HIGH**

```
> make verbosity=UVM_HIGH plus=uvm_set_verbosity=\*,TOPOLOGY,UVM_NONE,time,0
```

Now, the **TOPOLOGY** message is filtered out, but you see everything else at the **UVM_HIGH** verbosity.

The impact of filtering will be much more obvious when you deal with a real testbench.

Task 3. Add in an Environment

An environment with an agent which contains a sequencer and driver has been created for you. These classes are the same as shown in the lecture.

1. Bring in the environment and transaction files

```
> make environment
```

Add these to the **test.sv** file.

2. Open the **test.sv** file with an editor
3. Add the following include statements (must be in the order shown!)

```
program automatic test;
import uvm_pkg::*;
`include "packet.sv"
`include "driver.sv"
`include "input_agent.sv"
`include "router_env.sv"
`include "test_collection.sv"
```

4. Open the **test_collection.sv** file with an editor
5. Instantiate and construct an environment object in the test class:

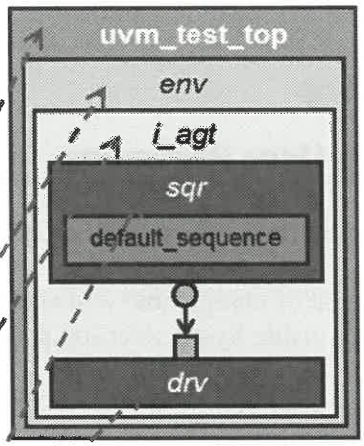
```
class test_base extends uvm_test;
  // Other code not shown
  router_env env;
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH) ;
    env = router_env::type_id::create("env", this);
  endfunction
endclass
```

Use factory to create object

6. Compile and simulate this simple UVM testbench:

```
> make
```

You should see a fuller topology and the factory content.
(You will see **`recording_details`** in additional to what's shown below)




```
UVM_INFO @ 0.0ns: reporter [UVMTOP] UVM testbench topology:
-------------------------------------------------------------
Name                      Type                      Size  Value
-------------------------------------------------------------
uvm_test_top              test_base                 -     @344
  env                     router_env                -     @356
    i_agt                 input_agent               -     @364
      drv                 driver                    -     @496
        rsp_port          uvm_analysis_port         -     @513
        seq_item_port     uvm_seq_item_pull_port    -     @504
      sqr                 uvm_sequencer             -     @373
        rsp_export        uvm_analysis_export       -     @381
        seq_item_export   uvm_seq_item_pull_imp     -     @487
        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: 58 total
  Type Name
  ---------
  driver
  input_agent
  router_env
  snps_uvm_reg_bank_group
  snps_uvm_reg_map
  test_base
```

7. Simulate with **TOPOLOGY** verbosity set to **UVM_NONE** everything else to **UVM_HIGH**

```
> make verbosity=UVM_HIGH plus=uvm_set_verbosity=\*,TOPOLOGY,UVM_NONE,time,0
```

You can begin to see the impact of the UVM report message filtering.

UVM provides you with a great deal of control over what report messages to generate. Choose the report message **IDs** carefully. They can become very handy during debugging.

Task 4. Creating User Packages

(The following is information only. No action is required.)

To enable better management of class names and compiled binaries, it is highly useful to include these class codes inside SystemVerilog packages.

For the subsequent labs, the files you have been working with will be moved inside packages. This will be done for you.

Be aware that package coding guidelines vary from company to company and project to project. It is highly recommended that once you return back to your home environment, consult your project manager and include your class code inside packages as directly by the project and company coding guidelines.

You are done with Lab 1!

Answers / Solutions

test.sv Solution:

```
program automatic test;
import uvm_pkg::*;
`include "packet.sv"
`include "driver.sv"
`include "input_agent.sv"
`include "router_env.sv"
`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

  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

  virtual function void start_of_simulation_phase(uvm_phase phase);
    super.start_of_simulation_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if (uvm_report_enabled(UVM_MEDIUM, UVM_INFO, "TOPOLOGY")) begin
      uvm_root::get().print_topology();
    end
    if (uvm_report_enabled(UVM_MEDIUM, UVM_INFO, "FACTORY")) begin
      uvm_factory::get().print();
    end
  endfunction
endclass
```

This page was intentionally left blank.

2 Stimulus Generation

Learning Objectives

After completing this lab, you should be able to:

- Create a UVM transaction class
- Generate UVM transaction objects in a sequence
- Configure the sequencer to execute the sequence in a desired phase
- Override existing transaction objects in environment with modified transaction class objects

Lab Duration:
30 minutes

Getting Started

In Lab 1, you wrote a simple UVM test. But, it didn't do anything. You can experiment with the debugging switches, but nothing more.

In this lab, you will develop a transaction class and a sequence class so that the embedded environment can generate and process transaction objects.

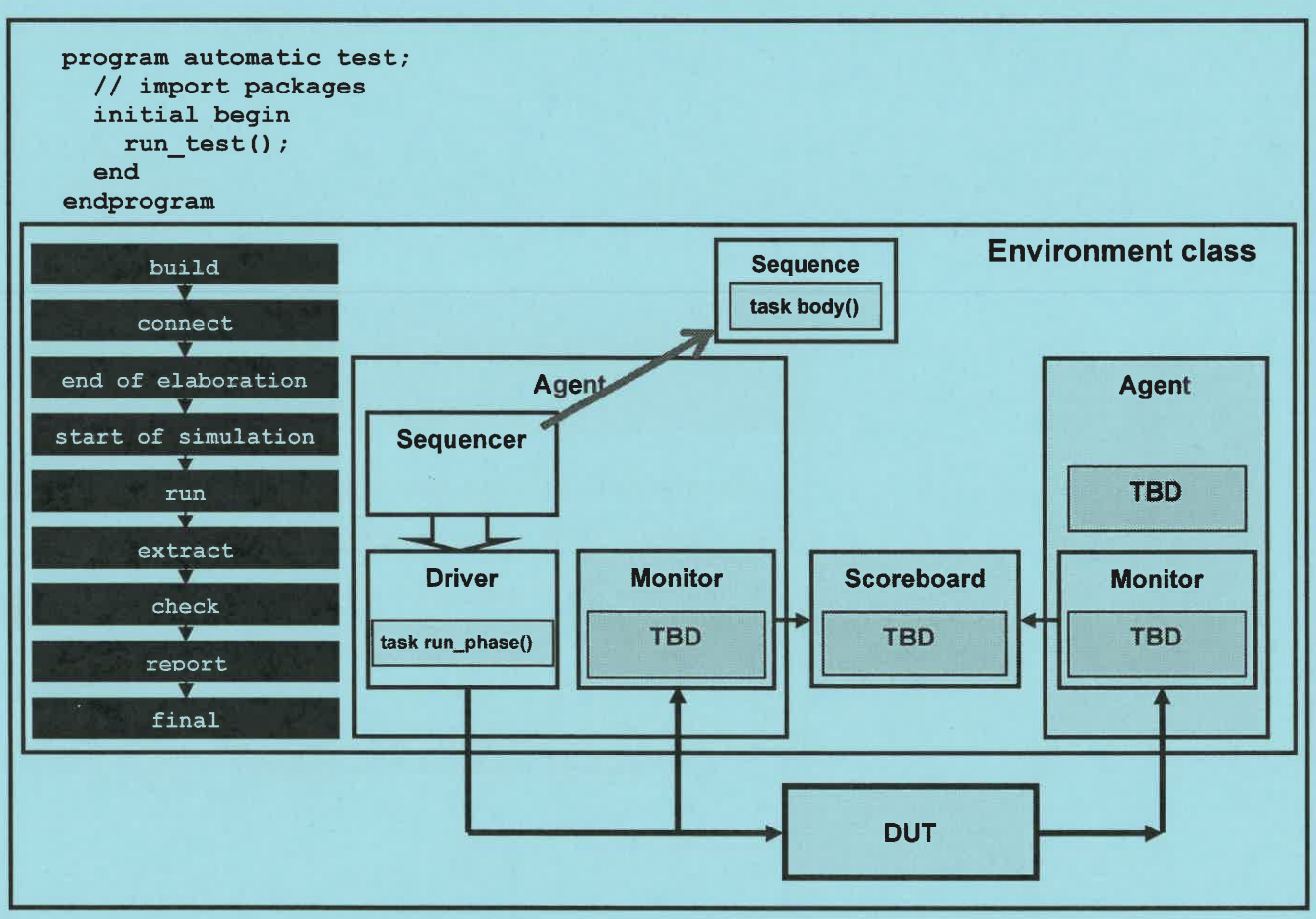


Figure 1. Lab 2 Testbench Architecture

Lab Overview

The work flow for this lab is illustrated as follows.

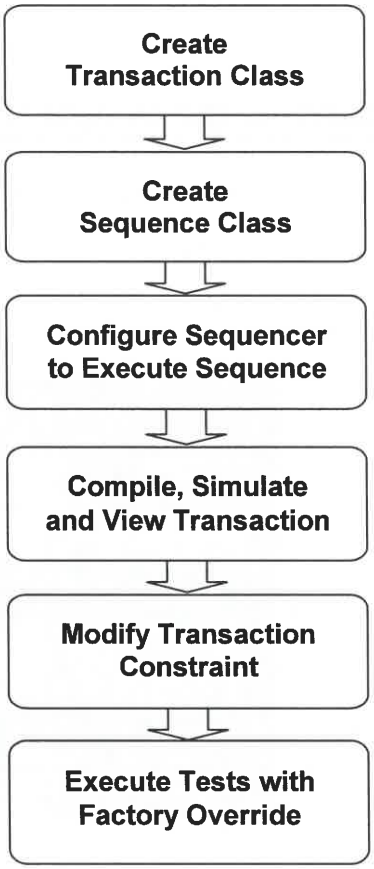


Figure 2. Lab 2 Flow Diagram

Generating UVM Transactions

Task 1. Go into lab2 Working Directory

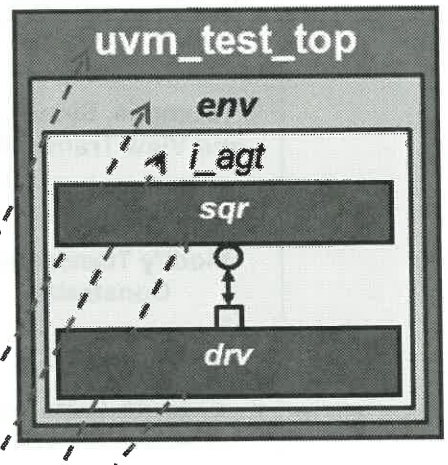
1. CD into the lab2 directory

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

Task 2. Review the Simple UVM Environment

In the previous lab, you constructed an environment in the test class.

This environment consists of a basic UVM component structure: environment (**`router_env`**), agent (**`input_agent`**), driver (**`driver`**) and sequencer (**`uvm_sequencer`**).




```
UVM_INFO @ 0.0ns: reporter [UVMTOP] UVM testbench topology:
-----------------------------------------------------------
Name                      Type                     Size  Value
-----------------------------------------------------------
uvm_test_top              test_base                -     @457
  env                     router_env               -     @465
    i_agt                 input_agent              -     @473
      drv                 driver                   -     @608
        rsp_port          uvm_analysis_port        -     @625
        seq_item_port     uvm_seq_item_pull_port   -     @616
      sqr                 uvm_sequencer            -     @485
        rsp_export        uvm_analysis_export      -     @493
        seq_item_export   uvm_seq_item_pull_imp    -     @599
        arbitration_queue array                    0     -
        lock_queue        array                    0     -
        num_last_reqs     integral                 32    'd1
        num_last_rsps     integral                 32    'd1
-----------------------------------------------------------
```

Task 3. Review Class Organization

Also, in the previous lab, all the class files were included into the program block.

```
program automatic test;
import uvm_pkg::*;
`include "packet.sv"
`include "driver.sv"
`include "input_agent.sv"
`include "router_env.sv"
`include "test_collection.sv"

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

endprogram
```

This is not an efficient way to manage class files. One downside of this type of implementation is that the **test.sv** file would need to be modified whenever a new test file is created. That's impractical and sometimes impossible.

A better way to manage these files is to organize them as packages. This is done for you. For the labs, you will be making use of three packages located in the **packages** directory: **router_stimulus_pkg**, **router_env_pkg** and **router_test_pkg**.

The **router_stimulus_pkg** contains the base transaction class.

```
package router_stimulus_pkg;
import uvm_pkg::*;
`include "packet.sv"
endpackage
```

The **router_env_pkg** contains the structural component classes.

```
package router_env_pkg;
import uvm_pkg::*;
import router_stimulus_pkg::*;
`include "driver.sv"
`include "input_agent.sv"
`include "router_env.sv"
endpackage
```

The **router_test_pkg** contains the test specific classes.

```
package router_test_pkg;
import uvm_pkg::*;
import router_stimulus_pkg::*;
import router_env_pkg::*;
`include "test_collection.sv"
endpackage
```

Once the files are organized this way, you will find your management of files becomes easier. If you also make use of partition compile in **vcs**, it can also reduce your compilation time.

Task 4. Update Program Block

With the packages provided, you will no longer need to include the class files into the program block.

1. Open the **test.sv** file with an editor.
2. Replace the include statements with import of the **router_test_pkg**

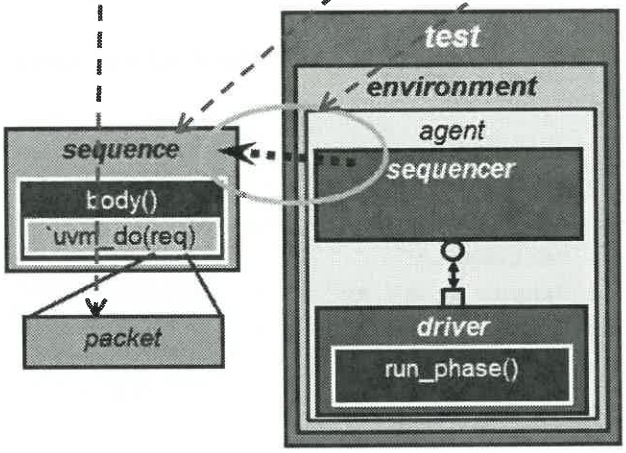
   The Makefile has been updated to make use of the packages.

3. Compile and run simulation.

   ```
   > make
   ```

   You should see the same result as the previous lab. The reorganization of class files does not change the behavior of the test.

   In the execution of the test, no stimulus was generated. You need a transaction class (**packet**) and a sequence class (**packet_sequence**) that constructs and randomizes the transaction objects. Then, configure the sequencer to execute the sequence in a desired phase so that transaction objects created in the sequence can be passed on to the driver in the desired task phase.



Task 5. Register the Packet Class into Factory

1. Open the **`packet.sv`** file with an editor.

2. Look for: "**`// Lab 2: Task 5, Step 2`**" and enter the following:
   - Register the class into the UVM factory
   - Enable all fields to be processed (except the **`sa`** field, see comment)
3. Look for: "**`// Lab 2: Task 5, Step 3`**"
   - Define the constructor method

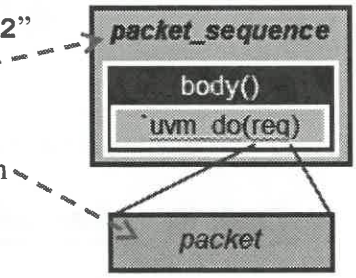
   Note: There is a major difference between transaction class constructor and component class constructor. In transaction class constructor, there is no parent component handle in the argument list.

4. Save and close the file

Task 6. Generate Stimulus Transaction

1. Open the **`packet_sequence.sv`** file with an editor
2. Look for: "**`// Lab 2: Task 6, Step 2`**"
   - Enter the **class declaration**

     You are required to parameterize the sequence class with the sequence_item class that the sequence is designed to generate for the driver to process.



When implementing the sequence class, one must define the **`body()`** task in which stimulus are generated.

3. Look for: "**`// Lab 2: Task 6, Step 3`**"
   - Create the **`body()`** method as described in the comment

One BIG caution before leaving the sequence class – UVM-1.2 was released on June 24, 2014. Unfortunately, some of the mechanisms implemented in UVM-1.1 have been deprecated and will not compile with UVM-1.2. One of these is the objection mechanism in sequence classes. Please read the comments as directed by the following two Steps.

4. Locate the constructor **`new()`** method and read the comments in the method.
5. Locate the **`pre_start()`** and **`post_start()`** methods and read the comments.

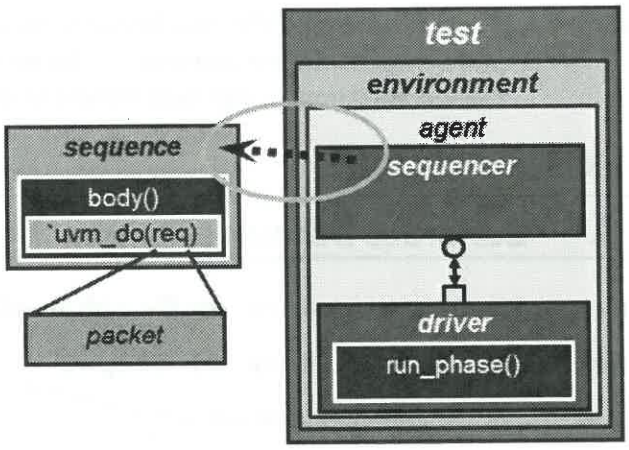
   Please make sure you understand what's going with the objection handling. This will save you a lot of headaches later.

6. Save and close the file.

Task 7. Configure Sequencer to execute Sequence

With the **packet** and the **packet_sequence** classes completed, you now have the mechanism for generating stimulus in UVM. The last step in this process is to configure the sequencer to execute the sequence.

1. Open the **router_env.sv** file with an editor
2. Look for "**// Lab 2: Task 7, Step 2**"
   - Set the **i_agt**'s sequencer (**sqr**) to execute **packet_sequence** as the default sequence in the **main phase**



3. Save and close the file

Task 8. Include Sequence Class in Stimulus Package

The sequence class needs to be compiled. The proper place for this is as part of the stimulus package.

1. Open **packages/router_stimulus_pkg.sv** file with an editor
2. Add an include statement for the **packet_sequence.sv** file
3. Save and close the file
4. Use **make** to compile and run simulation.

```
> make
```

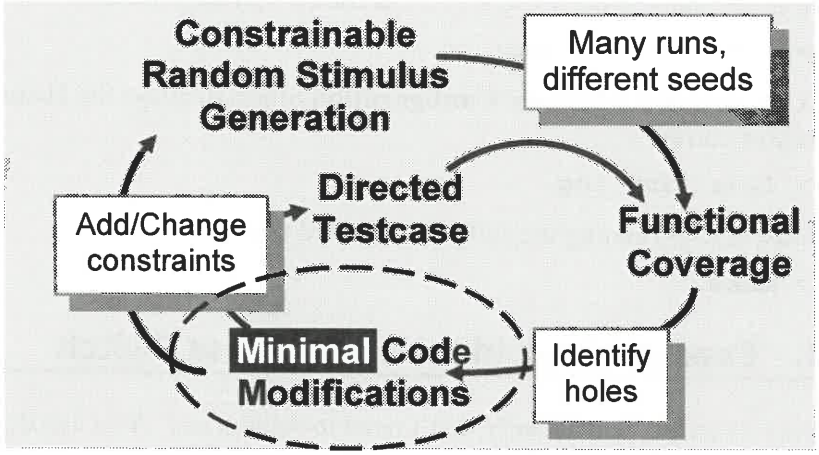
5. Take a look at the simulation result:

```
> less simv.log
```

   You should see that ten packets were printed. You are now able to generate stimulus for the driver to process.

Task 9. Execute Tests with Override

The goal of UVM is to simplify the development and execution of individual tests. One way to achieve this goal in UVM is to make use of the factory override feature.



In this task, you will constrain the destination address (**da**) for the test 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 10. Include Class in Stimulus Package

1. Open **packages/router_stimulus_pkg.sv** file with an editor
2. Add an include statement for the **packet_da_3.sv** file
3. Save and close the file

Task 11. Create a Test to Use the Modified Packet Class

1. Open **test_collection.sv** file in an editor
2. Look for "**// Lab 2: Task 11, Step 2**"
3. Use instance override to configure the sequence to use **packet_da_3** instead of **packet**
4. Save and close the file

Task 12. 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

```
> grep da simv.log
```

3. Check to see that the **Factory Configuration** report displays the **Instance Overrides** correctly

```
> less simv.log
```

4. Constrast it with running the fully randomized test:

```
> make
```

Task 13. Execute Override with run-time Switch

For the factory override, you actually don't need to write a test. You can do the same thing with a run-time switch.

1. Try instance override (you should type out the full path instead of wildcard):

```
> make plus=uvm_set_inst_override=packet,packet_da_3,\*.req
```

2. Try type override (no path required)

```
> make plus=uvm_set_type_override=packet,packet_da_3
```

So long as you have a path to an object creation, you can override any of the classes that you registered into the factory. For the user, it means that once you have created all the classes that you need, you will not need to write another test or re-compile. You can use the run-time switches to pick and choose the overrides to accomplish what you want to test.

You are done with Lab 2!

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: 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 final_phase(uvm_phase phase);
    super.final_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

    if (uvm_report_enabled(UVM_MEDIUM, UVM_INFO, "TOPOLOGY")) begin
      uvm_root::get().print_topology();
    end

    if (uvm_report_enabled(UVM_MEDIUM, UVM_INFO, "FACTORY")) begin
      uvm_factory::get().print();
    end
  endfunction: final_phase
endclass: test_base

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_agt.sqr.packet_sequence.req",
packet::get_type(), packet_da_3::get_type());
  endfunction
endclass
```

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
```

router_env.sv Solution:

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

  input_agent i_agt;

  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_agt = input_agent::type_id::create("i_agt", this);

    uvm_config_db #(uvm_object_wrapper)::set(this, "i_agt.sqr.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);

    `ifdef UVM_POST_VERSION_1_1
      set_automatic_phase_objection(1);
    `endif
  endfunction: new

  virtual task body();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    repeat (10) begin
      `uvm_do(req);
    end
  endtask: body

  `ifdef UVM_VERSION_1_1
  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: 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
  `endif

endclass: packet_sequence
```

3 Component Configuration

Learning Objectives

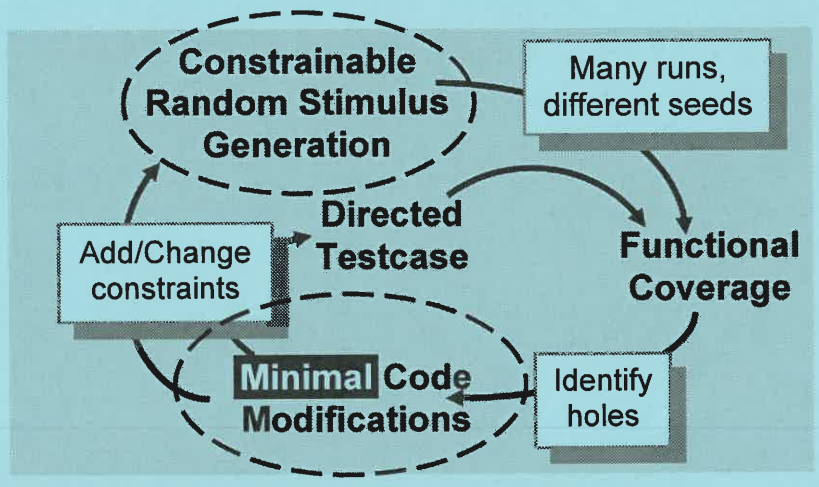
After completing this lab, you should be able to:

- 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 2, you were able to generate stimulus. In this lab, you will add physical interface to the driver and drive the stimulus through the DUT.




```
program automatic test;
  // import packages
  initial begin
    run_test();
  end
endprogram
```

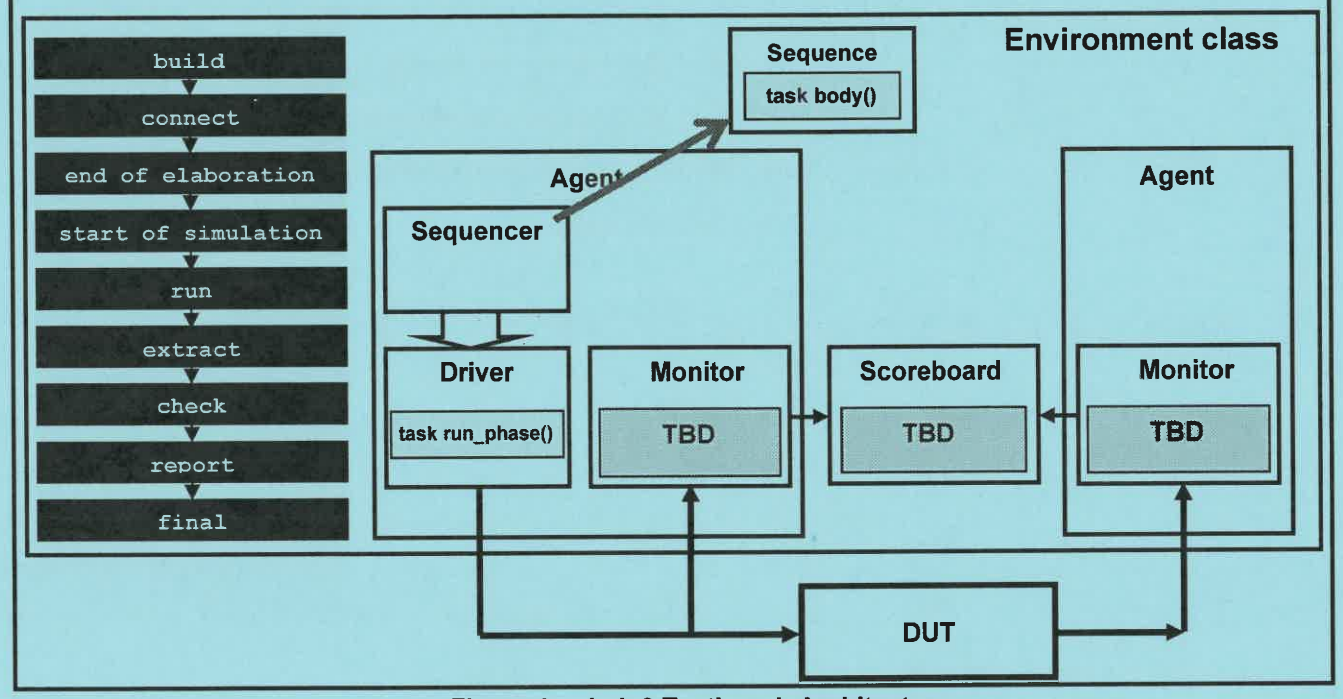


Figure 1. Lab 3 Testbench Architecture

Lab Overview

The work flow for this lab is illustrated as follows:

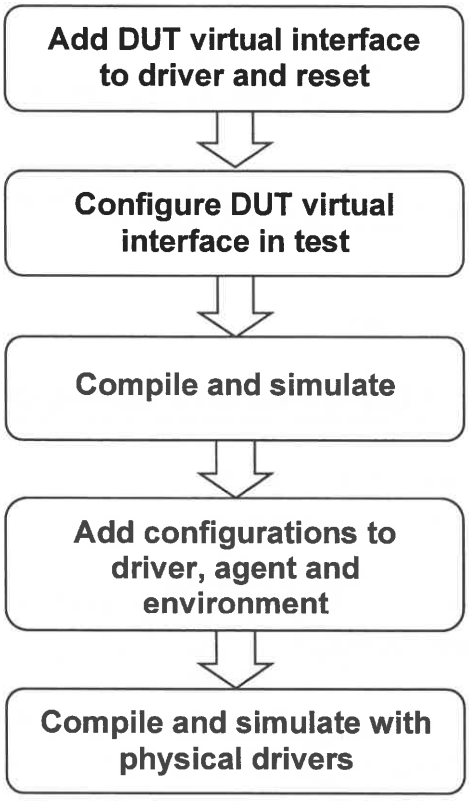


Figure 2. Lab 3 Flow Diagram

Implement Configurable 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 driver code.

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

Task 1. Go into lab3 Working Directory

1. CD into the lab3 directory

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

Task 2. Add Interface and Configuration to Driver

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

```
class driver extends uvm_driver #(packet);
  virtual router_io vif;
```

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 **port_id** field to **uvm_component_utils**:

Add _begin to macro

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

Notice that the virtual interface handle (**vif**) 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 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);
 uvm_config_db#(virtual router_io)::get(this, "", "vif", vif);
endfunction
```

Get **port_id**

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

6. Check the configuration properties in end_of_elaboration phase:

```
virtual function void end_of_elaboration_phase(uvm_phase phase);
 super.end_of_elaboration_phase(phase);
 if (!(port_id inside {-1, [0:15]})) begin
  `uvm_fatal("CFGERR",$sformatf("port_id must be {-1, [0:15]}, not %0d!",port_id));
 end
 if (vif == null) begin
  `uvm_fatal("CFGERR", "Interface for Driver not set");
 end
endfunction
```

check

7. 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();
```

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

8. 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 configuration of the sequencer, driver and monitors inside the agent should be done by the agent. The higher structural components should configure only the agent, not the individual components within the agent. The agent would then configure its own children 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 fatal error:

```
UVM_INFO @ 0.0ns: reporter [RNTST] Running test test_base...
UVM_FATAL driver.sv(94) @ 0.0ns: uvm_test_top.env.i_agt.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. Store Interfaces in Resource Database

The interfaces of the DUT are static, the proper place to store the handles to these interfaces into the resource database is the **`program/module`** where **`run_test()`** is called. There are two interfaces that the test need access to: the router port io and the router reset signal.

1. Open **`test.sv`** file in an editor
2. Populate the resoure data base with the interfaces needed by the test
3. Save and close the file

Task 6. Configuration Agents in Test

The proper place to configure the physical agents to use these interfaces is in the base test.

1. Open **test_collection.sv** file in an editor
2. Declare the two virtual interface handles to the interfaces

```
class test_base extends uvm_test;
  `uvm_component_utils(test_base)
  router_env env;
  virtual router_io router_vif;
  virtual reset_io  reset_vif;
```

3. Locate the **build_phase** method in **test_base** class
4. Retrieve the interfaces from resource database
5. Then configure the agent with these interfaces
6. Save and close the file

Task 7. 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. Check the behavior in debugger window

   > make dve

   Or,

   > make verdi

   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 debugger

Task 8. Generate Reset Transactions

In this task, you will develop 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

To save lab time, this file is included in the router stimulus package for you.

Task 9. 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 the debugger

   > make dve

   Or,

   > make verdi

   You should now see that the output signals no longer are red. However, the sequence was designed to send 10 packets through the DUT. In the waveform window, this is not what you are seeing (you may need to adjust the waveform window to see all 16 frame signals). There is still a problem.

3. Exit debugger

Task 10. 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 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 complicates 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 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. De-asserting the control signals in the driver's **`reset_phase()`** is discouraged.

A sequence to de-assert the router input control signals has been done for you in **`router_input_port_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. Configure the input agent's (**`i_agt`**) sequencer (**`sqr`**) to execute the **`router_input_port_reset_sequence`** at the **`reset_phase`**.
3. Save and close the file

Task 11. Compile & Simulate with Driver Reset Phases

1. Compile and simulate the testbench.

   ```
   > make
   ```

2. Open the debugger

   ```
   > make dve
   ```

   Or,

   ```
   > make verdi
   ```

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

   A test has been written for you that executes a sequence which sets the da to 3 and generates 20 packets.

3. Run this test:

   ```
   > make test=test_da_3_seq
   ```

4. Take a look at the debugger waveform window again

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

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

If desired, you can continue with UVM transaction debugging in the following pages.

Optional: DVE Transaction Debugging

It is very helpful to view the transaction being processed alongside the DUT waveforms. The following steps will take you through a simple example.

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

```
> make dve_tr -n
```

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
```

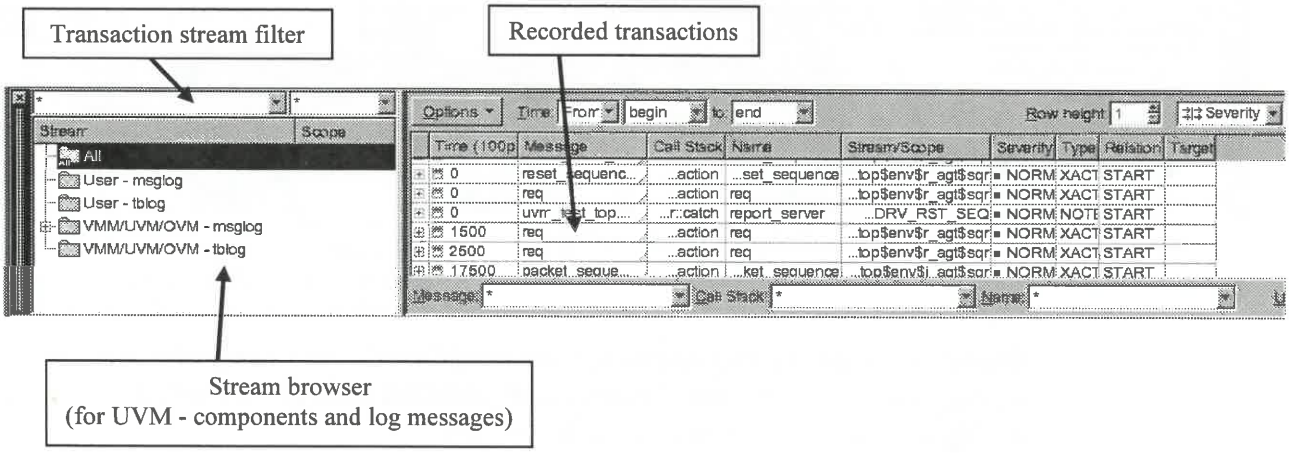
Enables UVM transaction recording

Enables UVM LOG message recording

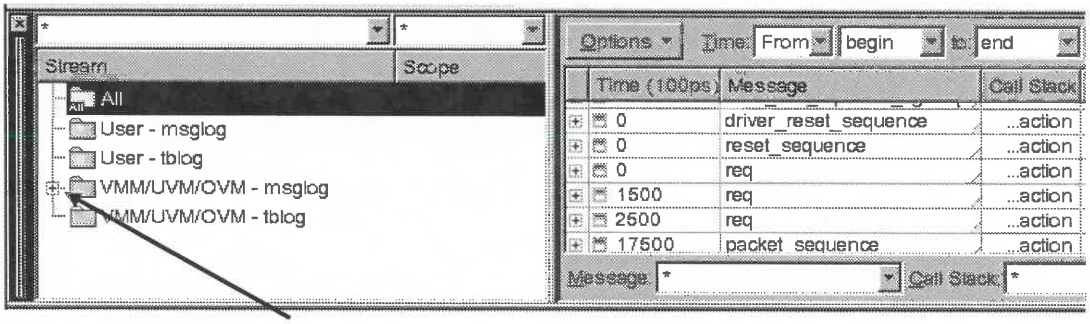
2. Start DVE in post-processing mode with transaction debugging:

```
> make dve_tr
```

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



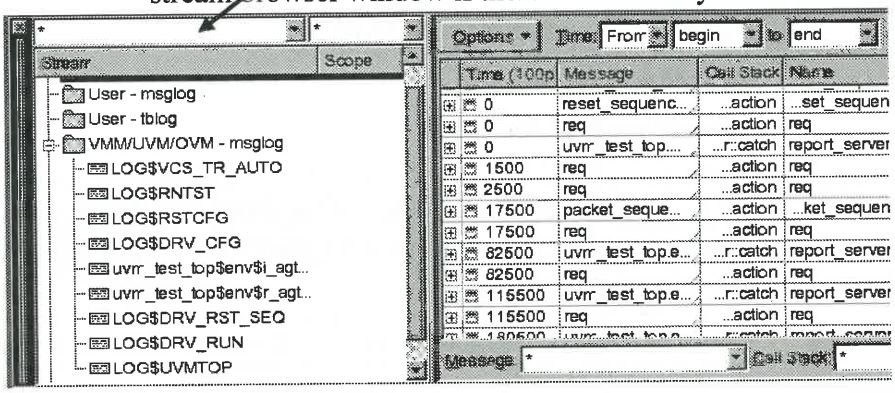
For UVM, the automatically recorded streams are the UVM sequence item transactions and the UVM log messages. For this lab, you will only be looking at the **VMM/UVM/OVM – mslog** folder.



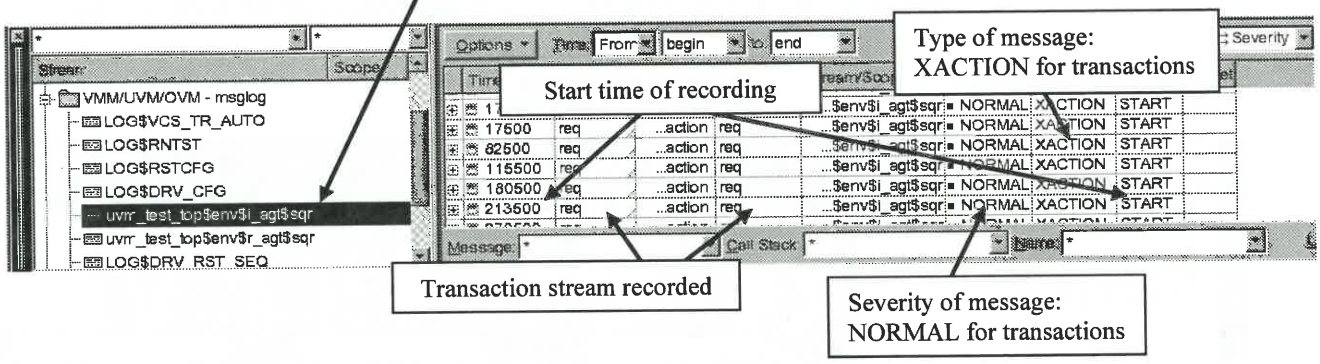
3. Click on the + symbol of the **VMM/UVM/OVM – msglog** folder to expand it

In the expanded window, 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_top$env$i_agt$sqr`** to see the **sequence** and **sequence_items** transaction stream that the sequencer generated

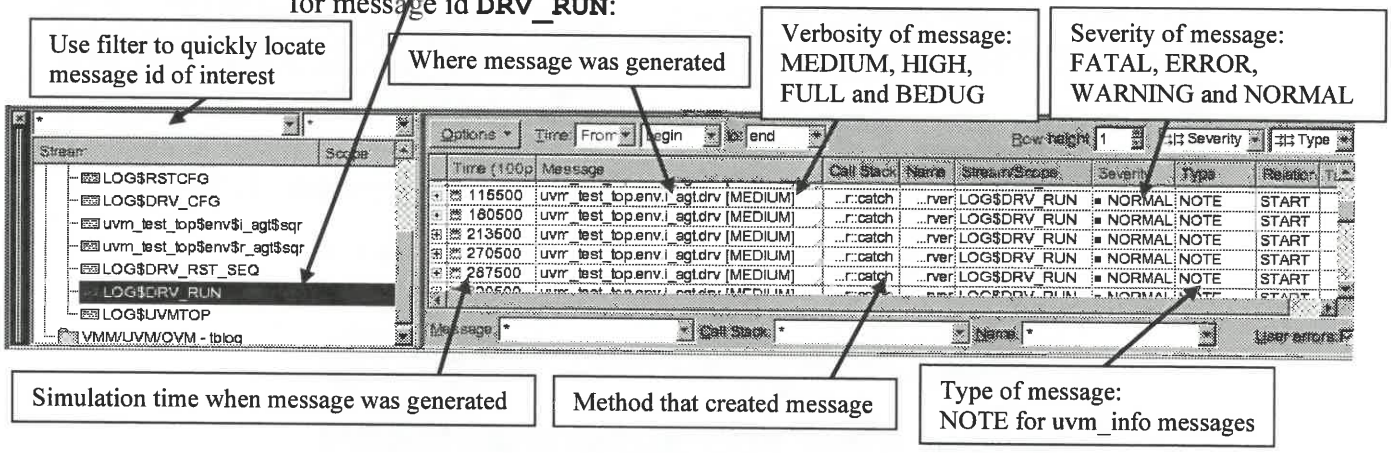


The UVM **`uvm_info/warning/error/fatal** macro generated log messages are also recorded if **+UVM_LOG_RECORD** run-time is applied during simulation (as has been done in the lab).

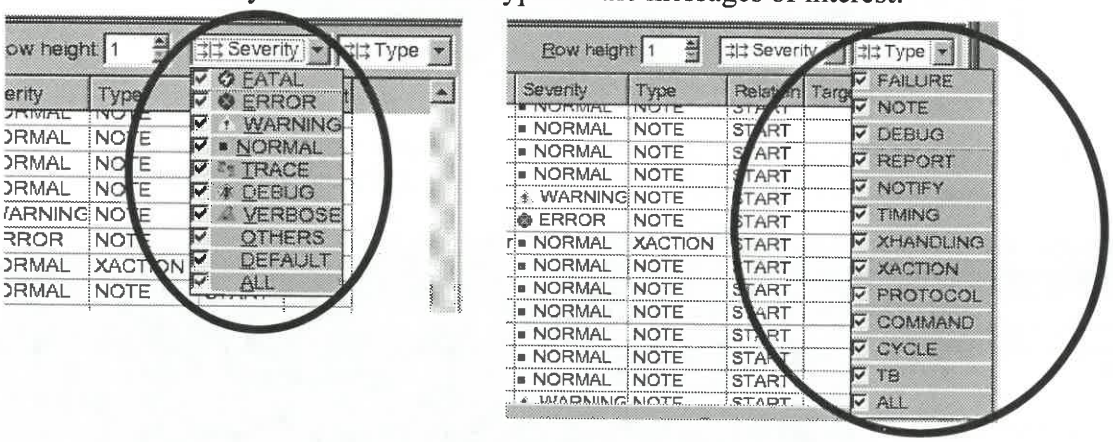
```
class driver extends uvm_driver #(packet);
  // other code not shown
  virtual task run_phase(uvm_phase phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    forever begin
      seq_item_port.get_next_item(req);
      if (port_id inside { -1, req.sa }) begin
        send(req);
        `uvm_info("DRV_RUN",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

For example, in the driver code, a **`uvm_info** is embedded with message id of **DRV_RUN**. This shows 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**:

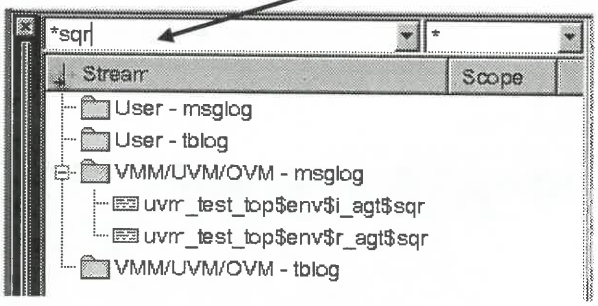


At the upper right hand corner of the transaction pane, there are two drop-down tabs for severity and type. Clicking on them will enable you to display only the severities and types of the messages of interest:



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 ***sqr**

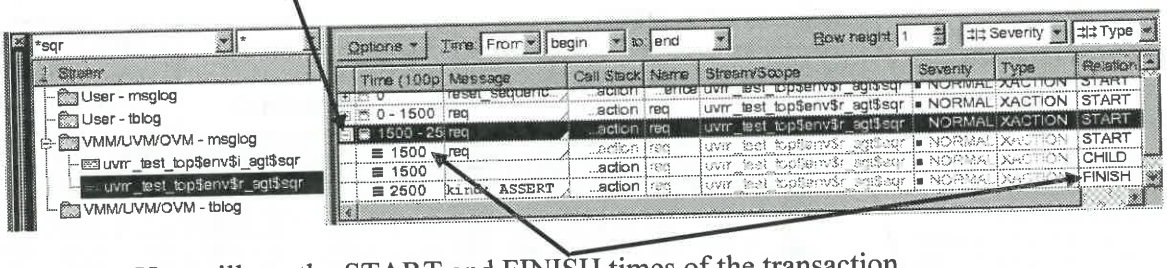


As you type, you should notice that the stream source browser window is updated dynamically to show you the result of the filter.

7. Click on **uvm_test_topenvr_agt$sqr**

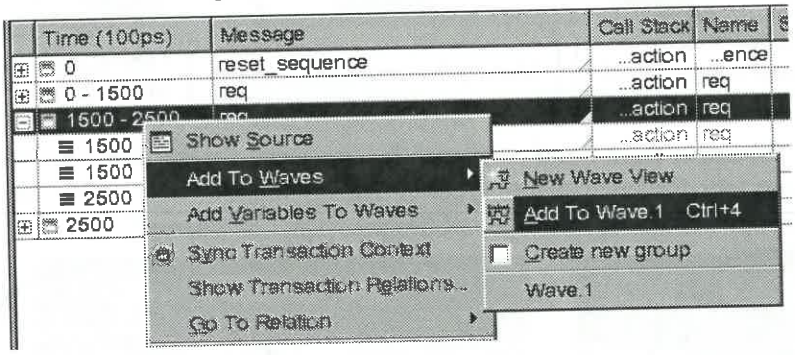
This will display the **reset_sequence** and reset_tr object **req**.

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

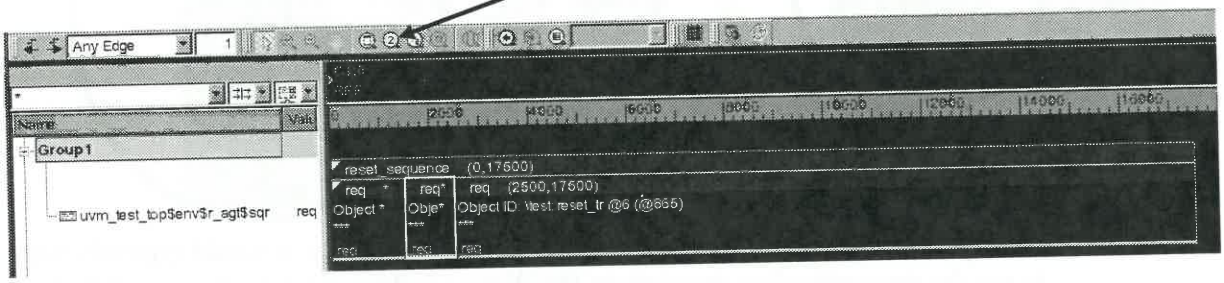


You will see the START and FINISH times of the transaction

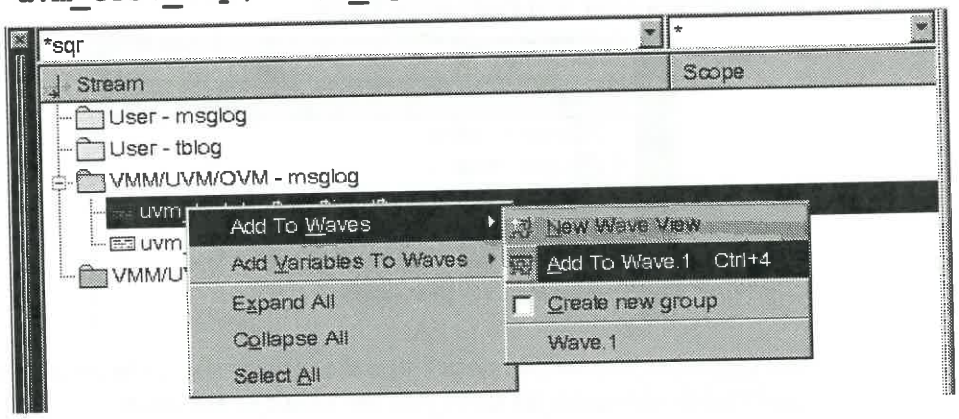
9. Click on the Right Mouse Button on the transaction, add it to Wave window



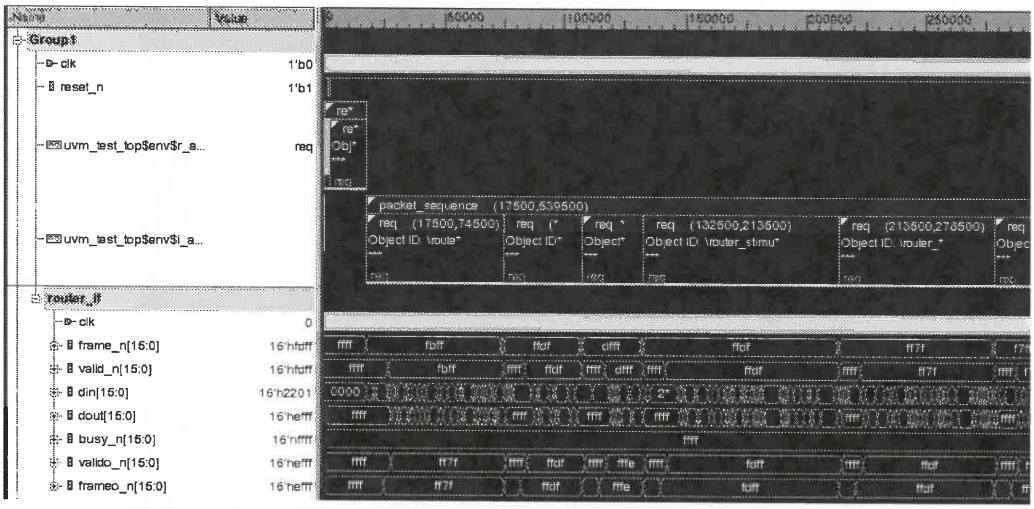
10. In the Wave window, click on the zoom in button a few time to see the reset_sequence fully displayed at the start time and clear at the finish time.



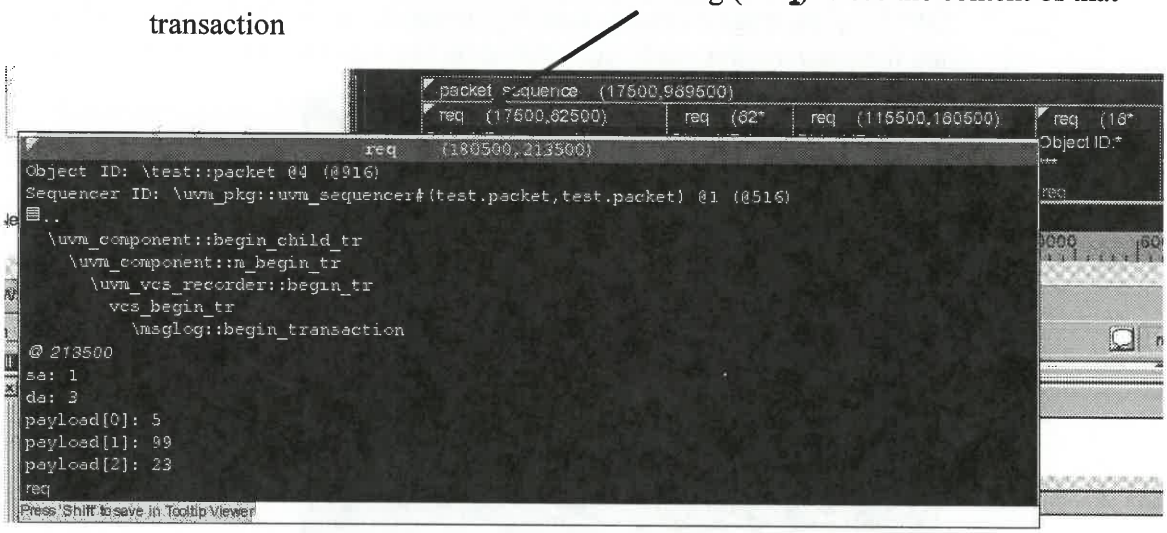
11. Go back to DVE Hierarchy window. This time, in Stream browser pane, add **`uvm_test_top$env$i_agt$sqr`** to the Wave window.



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



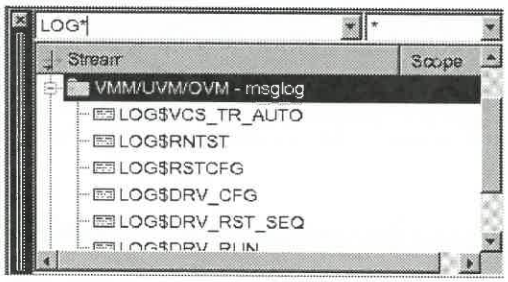
12. Place the cursor over each transaction heading (**`req`**) to see the content of that transaction



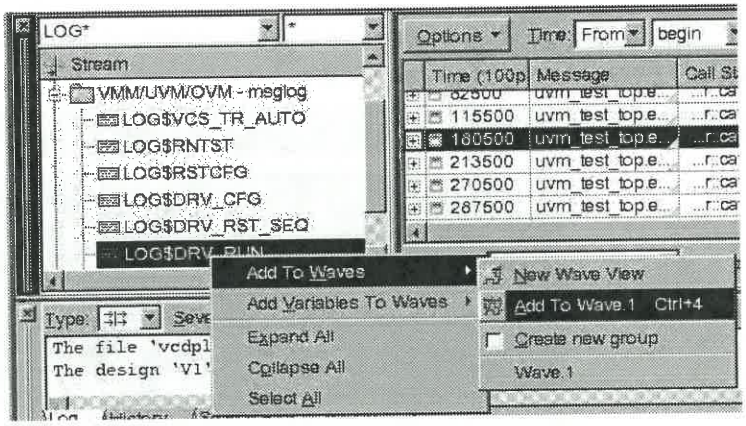
For the UVM log messages, you can do the same thing.

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

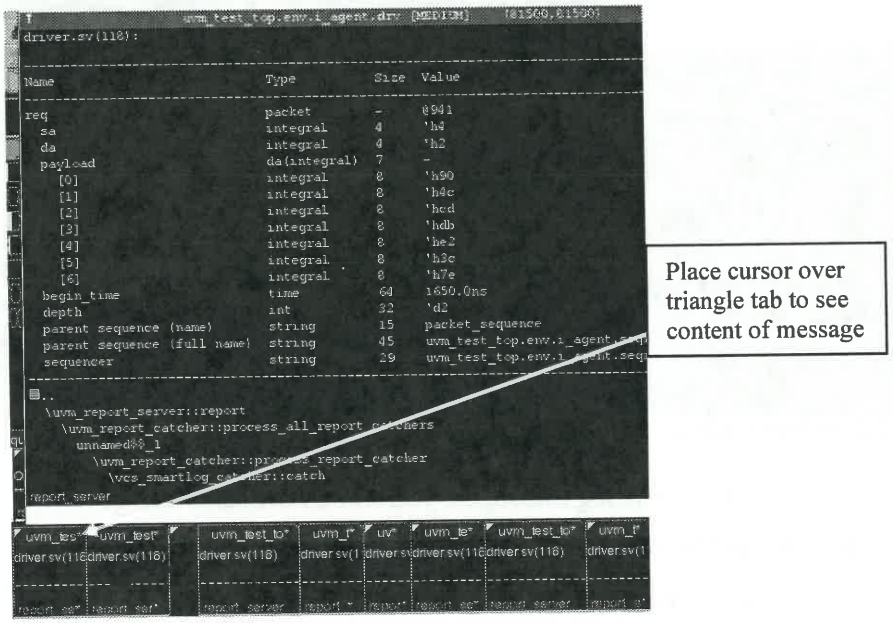
You will now see only the log message streams.



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



Once again, you can place the cursor over the message heading 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 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.

You have completed the optional dve portion of Lab 3!

Optional: Verdi Transaction Debugging

Caution: The features show below are lca features and not part of general release. If you have any questions on the availability of these features in your environment, please consult with your Synopsys Sales Rep.

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

```
> make compile -n
```

You should see the following:

```
vcs -sverilog -lca -debug_access+all [other switch not shown]
```

Requires lca switch to enable transaction debugging

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

```
> make verdi_tr -n
```

You should see the following:

```
./simv [other switches not shown] \
      +UVM_TR_RECORD +UVM_LOG_RECORD +UVM_VERDI_TRACE
```

Enables UVM LOG message recording

Enables UVM transaction recording

Enables UVM message recording

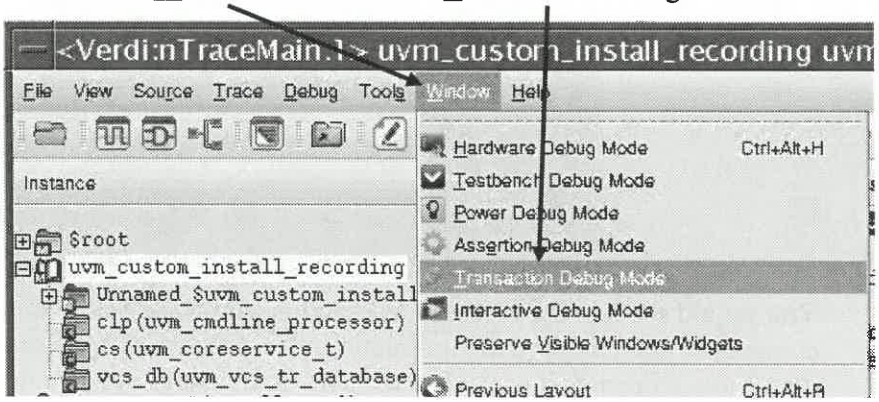
Enables Verdi transaction recording

3. Start Verdi in post-processing mode with transaction debugging:

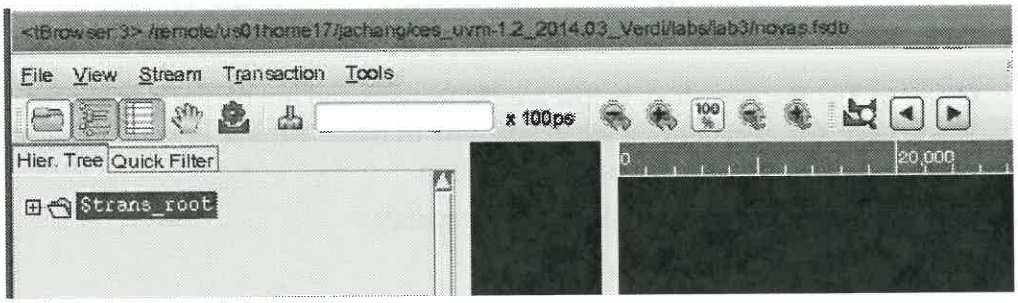
```
> make verdi_tr
```

4. Open the transaction debugging window:

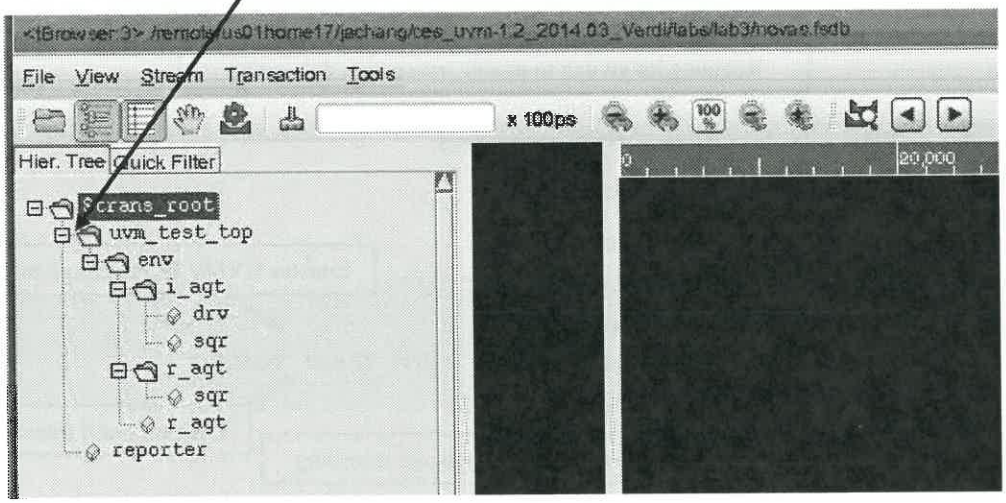
Click the Window tab and select Transaction Debug Mode



In the **`Hier. Tree`** pane you should see **`$trans_root:`**

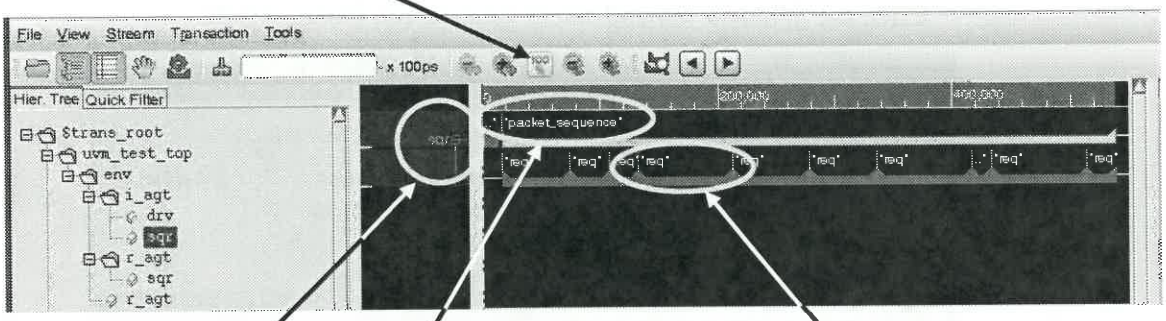


5. Click on all the ⊞ symbols to fully expand out the transaction hierarchy. What you see is the complete collection of components in which uvm transactions and uvm report messages are captured.



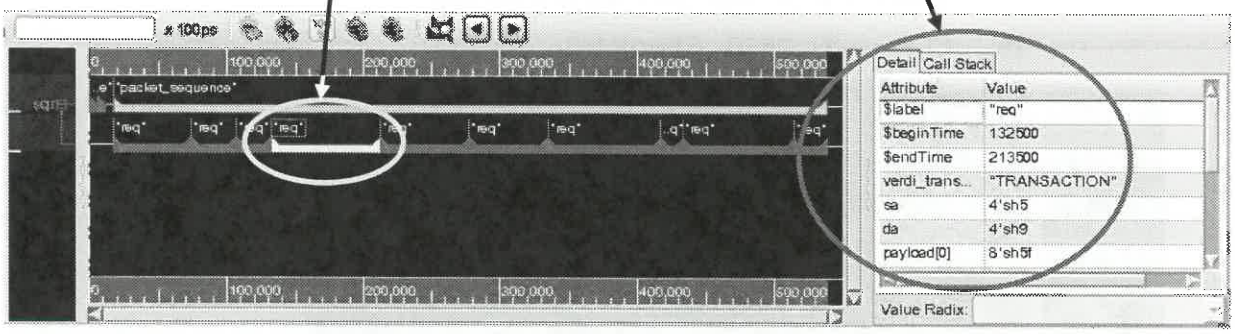
6. Using the middle mouse button, click, drag and drop the **`i_agt`**'s **`sqr`** content into the waveform window

7. Then, click on to zoom out to see transactions recorded by the sequencer



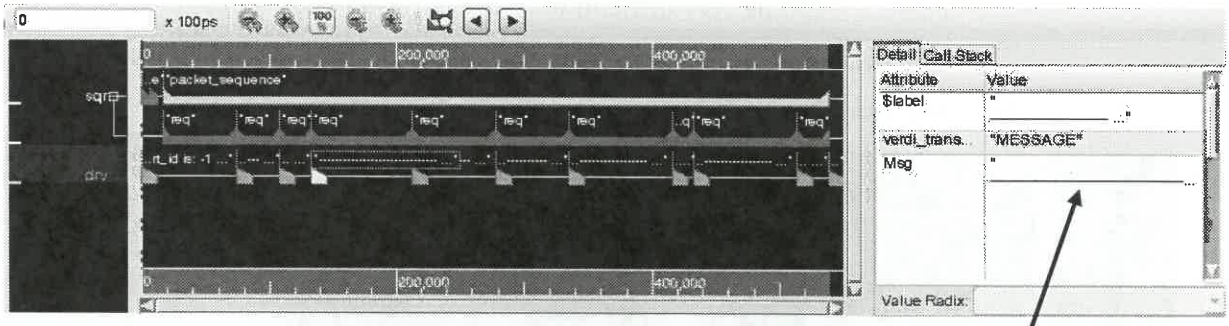
You should see that the recorded transaction shows the name of the component that recorded the transaction. If the transaction recording was done via a sequence, it not only shows the individual sequence item, it also shows the sequence that generated the sequence item.

8. Click on a transaction highlights the transaction. In the **`Detail`** pane, the properties of the transactions are displayed.



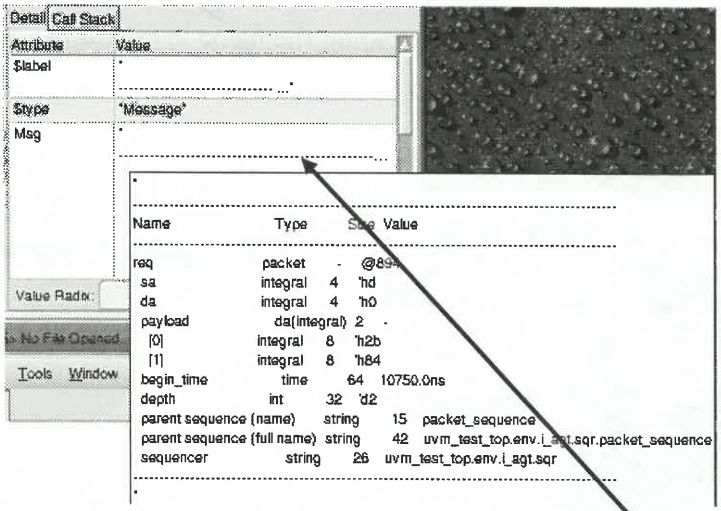
9. Drag and drop the driver (**`drv`**)'s report message recordings into the waveform window

You should see something a little different.



Instead of spanning time like you see . The difference is that transaction processing typically spans time, whereas report messages are issued at an instance of time.

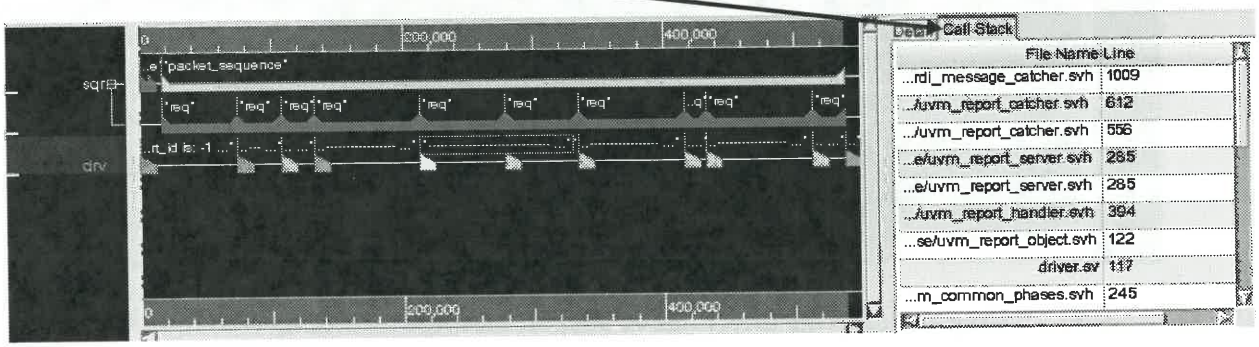
If the message is too big, you will see a line of dashes ----------.



To see the message, simply move the cursor over the line of dash.

During debugging, there are generally two things that you want to see with respect to the transaction and message – where was it generated and how is it related to DUT behavior (timing diagram).

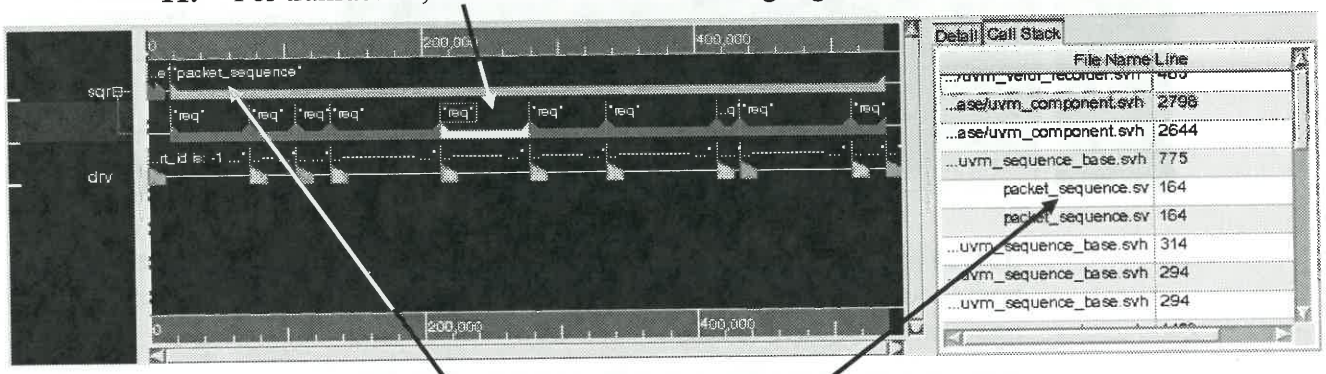
10. Click on the **`Call Stack`** tab



You will see the call stack trace of how the transaction/message was generated. But, because the actual recording is done by the UVM base classes, it can be very difficult to figure out what to do here.

The strategy is different for tracing transaction and report messages.

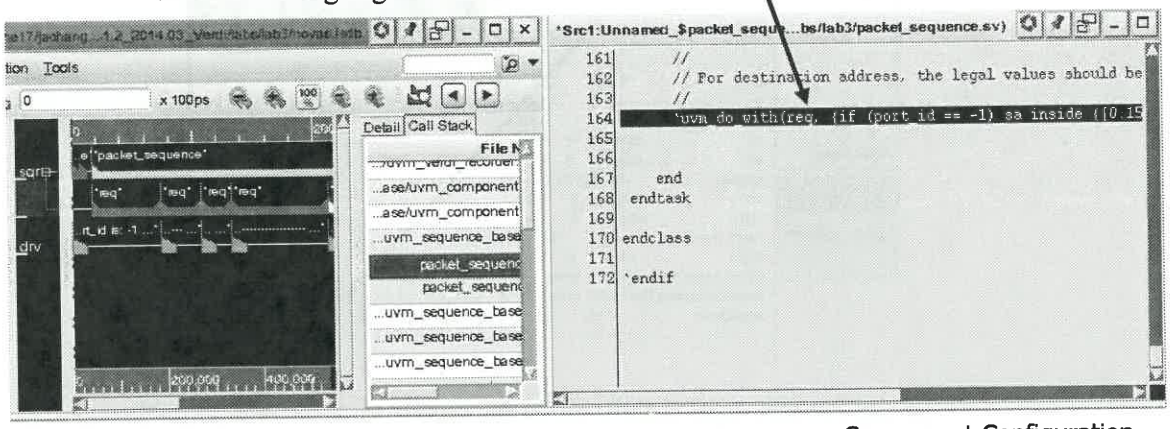
11. For transaction, click on a transaction to highlight it



Look for the sequence name in the wavename.

12. Double click on the corresponding file name in the **`Call Stack`** pane.

This will open a source code window with the line that generated the transaction highlighted.



Typically, the window is too small to be useful for debugging. You can undock this window.

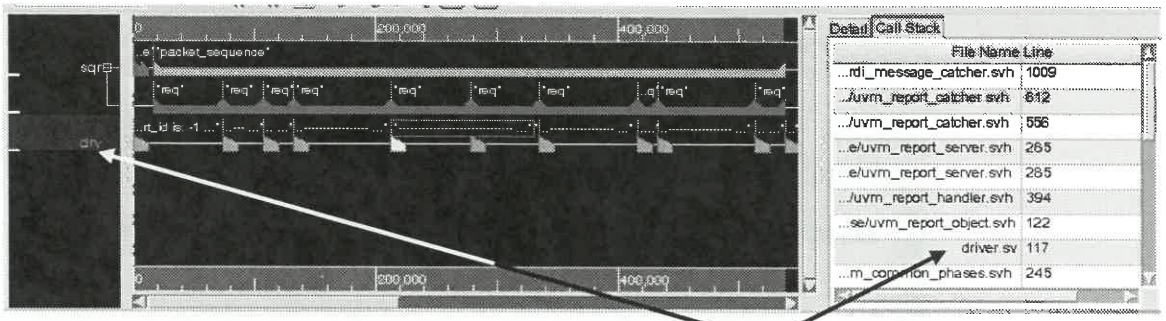
13. Click on to undock the window

The source code window should now be undocked and can be re-sized to meet your debugging needs.

```
*Src1:Unnamed_$packet_sequence_sv_164(/remote/us01home17/jachang/ces_uvm-1.2_2014.03_Verdi/labs/lab3/packet_sequence.sv)
149     //
150     // Instead of hard coding the number of item to be generated, replace the
151     // hard-coded value 10 in the repeat() statement to use item_count.
152     //
153     repeat(item_count) begin
154
155
156       //
157       // As stated in the comment at the beginning of the file.  If the port_id is unconfigured (-1)
158       // then the legal values for the source address shall be in the range of {[0:15]}.  If the
159       // port_id is configured, then the source address shall be port_id.  This will give the test
160       // the ability to test whether or not the driver drops the packet it is not configured to drive.
161       //
162       // For destination address, the legal values should be picked out of the valid_da array
163       //
164       `uvm_do_with(req, {if (port_id == -1) sa inside {[0:15]}; else sa == port_id; da inside valid_da;});
165
```

To trace a report message, the mechanism is similar.

14. Click on a report message to highlight it

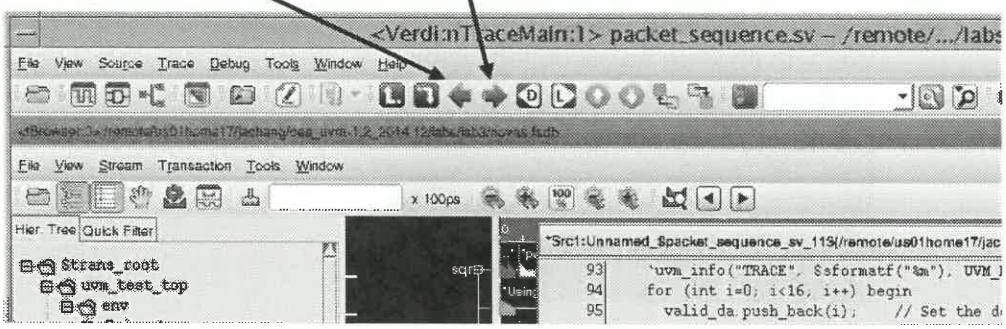


For the report message, you will need to relate it to the component that generated the message.

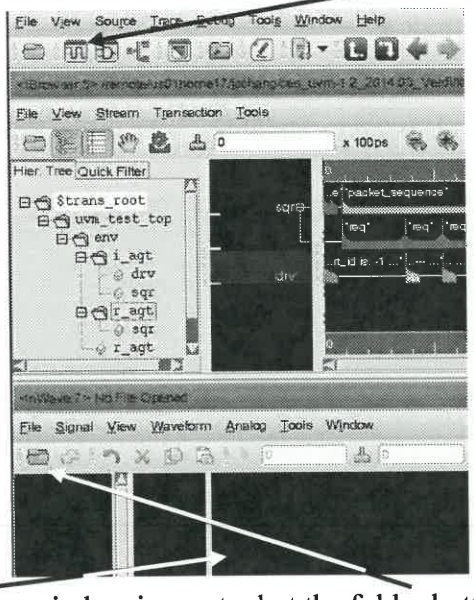
15. Double click on the corresponding file name in the **`Call Stack`** pane.

The source code window now shows the generating code.

You can switch back and forth between the two source codes by clicking on the Backward History and Forward History buttons.

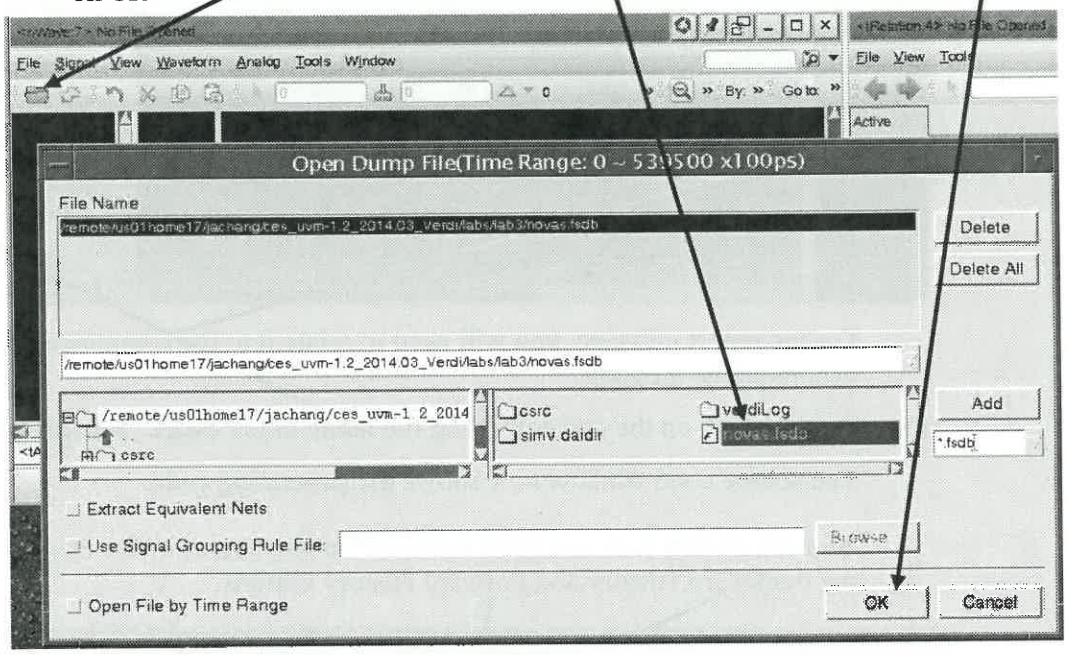


16. Open a waveform window by clicking on the button

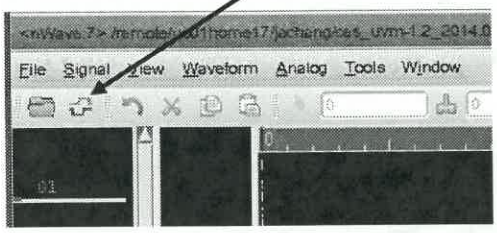


The waveform window is empty, but the folder button is now enabled.

17. Click on the folder button, double-click on the **novas.fsdb** file, then click on **OK**

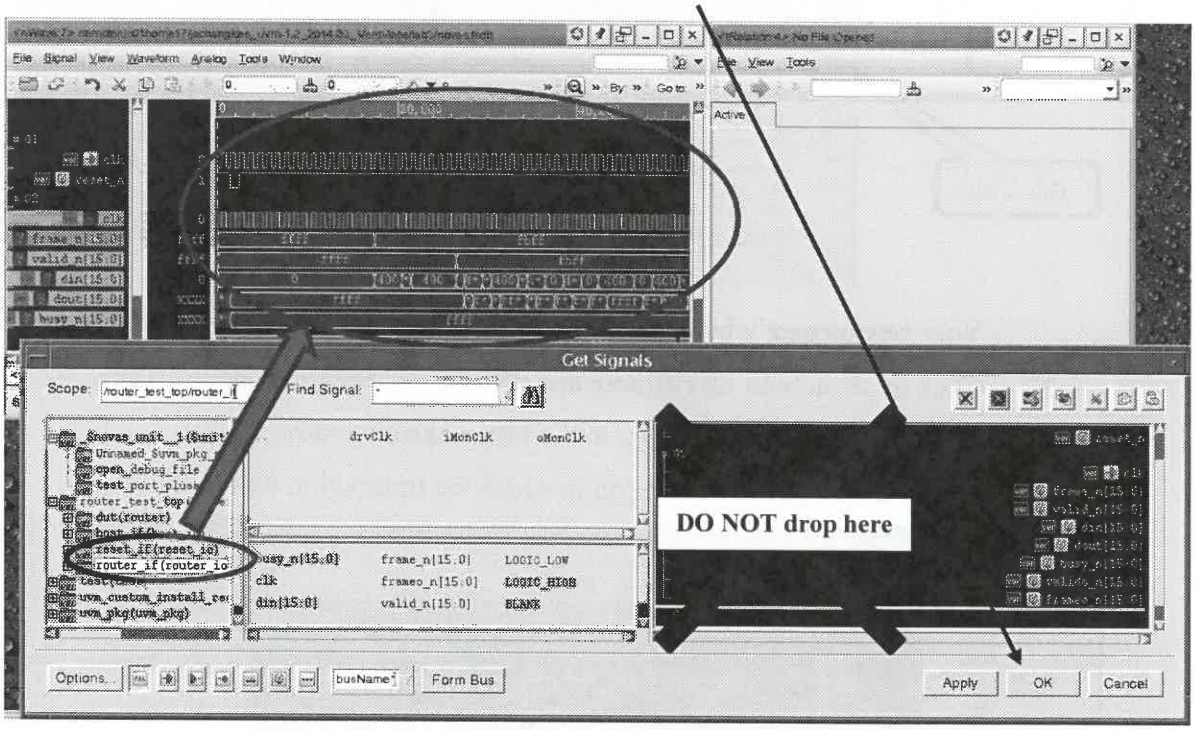


You will now see waves button enabled.



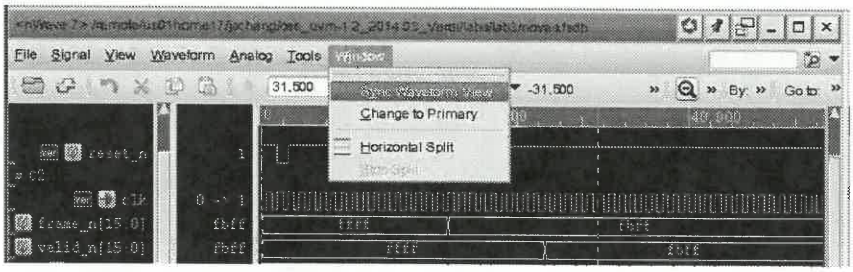
18. Click on the waves button

19. Drag and drop **reset_if** and **router_if** in the **Get Signals** window into the **nWave** window then click on **OK**



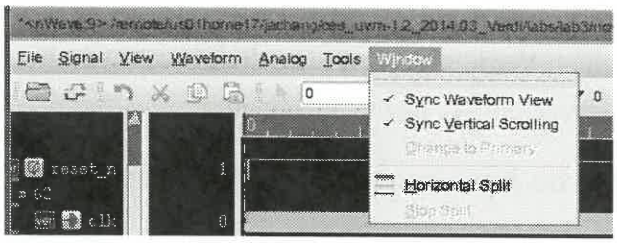
What will be very useful for debugging is to visually see the relationship between the DUT waveform and the transaction begin processed.

20. In **nWave** window, click on **Window** and select **Sync Waveform View**

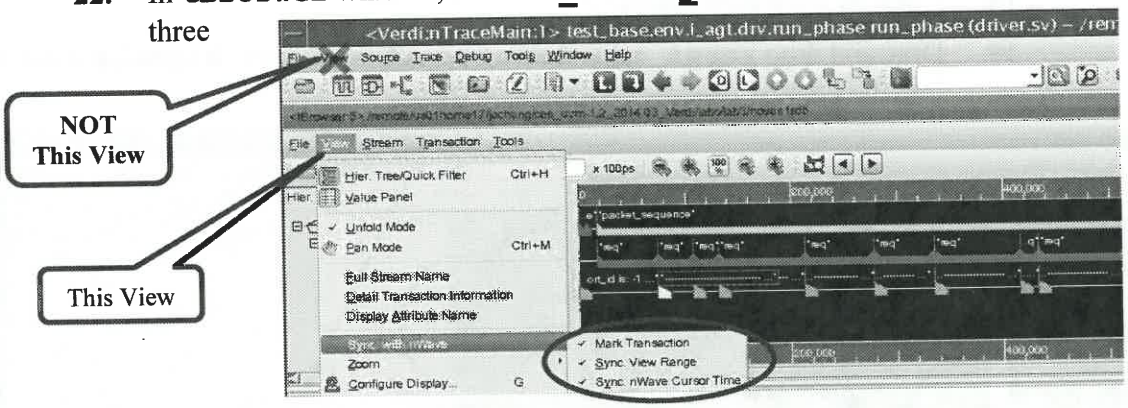


21. In the **Window** tab again, enable **Sync Vertical Scrolling** and **Change to Primary**

The Change to Primary selection will grey out. This is what you will see when these actions complete.



22. In **tBroswer** window, click on **View->Sync with nWave** and enable all three

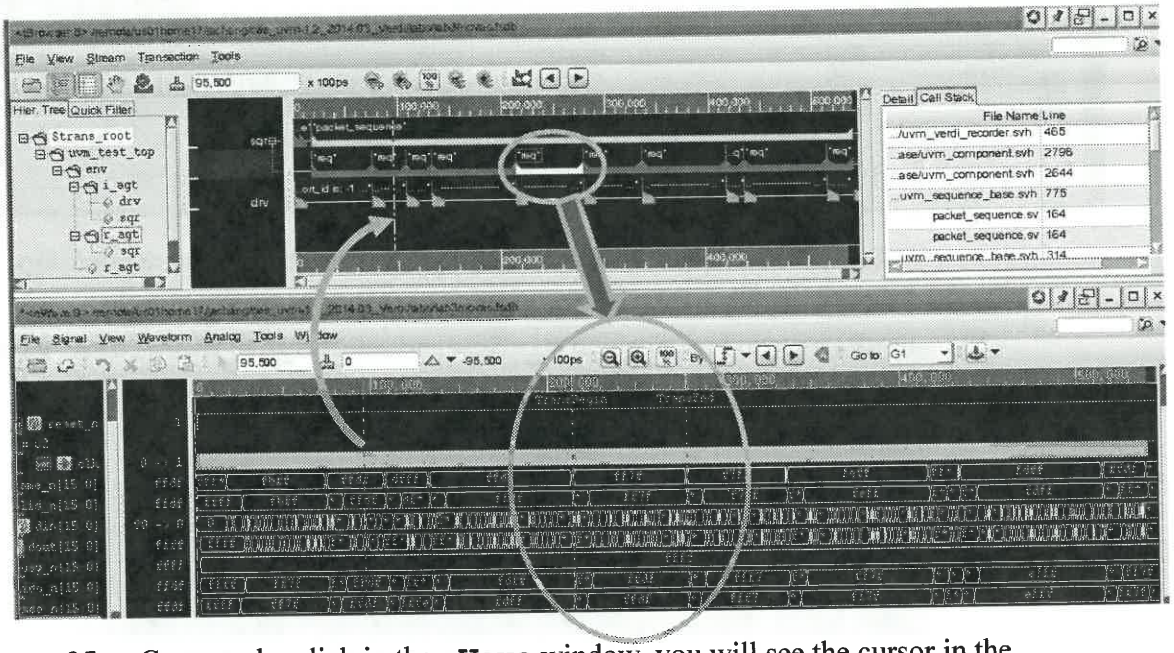


Your **tBrowser** window and the **nWave** windows are now sync'ed.

23. Click on in both **tBrowser** and **nWave** windows to zoom to 100%

24. Then click on any transaction in the **tBrowser** window

You will now see the time span in which the transaction was in-flight.



25. Conversely, click in the **nWave** window, you will see the cursor in the **tBrowser** window is sync'ed to the same simulation time.

By making using of the ability to see transactions and report messages alongside the DUT waveform, you will find debugging to be less of a chore.

For more information on Verdi, consult the Verdi documents.

(type **verdi -doc** to see the available documents)

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

Answers / Solutions

test_collection.sv Solution:

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

  virtual router_io router_vif;
  virtual reset_io  reset_vif;

  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);

    uvm_resource_db#(virtual router_io)::read_by_type("router_vif",
router_vif, this);
    uvm_resource_db#(virtual reset_io)::read_by_type("reset_vif",
reset_vif, this);

    uvm_config_db#(virtual router_io)::set(this, "env.i_agt", "vif",
router_vif);
    uvm_config_db#(virtual reset_io)::set(this, "env.r_agt", "vif",
reset_vif);
  endfunction: build_phase

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

    if (uvm_report_enabled(UVM_MEDIUM, UVM_INFO, "TOPOLOGY")) begin
      uvm_root::get().print_topology();
    end

    if (uvm_report_enabled(UVM_MEDIUM, UVM_INFO, "FACTORY")) begin
      uvm_factory::get().print();
    end
  endfunction: final_phase
endclass: test_base

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: new

  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_agt.sqr.*", packet::get_type(),
packet_da_3::get_type());
  endfunction: build_phase
endclass: test_da_3_inst

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: new

  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: build_phase
endclass: test_da_3_type

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: new

  virtual function void build_phase(uvm_phase phase);
    packet_sequence::int_q_t valid_da = {3};
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    uvm_config_db#(packet_sequence::int_q_t)::set(this,
"env.i_agt.sqr.packet_sequence", "valid_da", valid_da);
    uvm_config_db#(int)::set(this, "env.i_agt.sqr.packet_sequence",
"item_count", 20);
  endfunction: build_phase
endclass: test_da_3_seq
```

router_env.sv Solution:

```
class router_env extends uvm_env;
  input_agent i_agt;
  reset_agent r_agt;

  `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_agt = input_agent::type_id::create("i_agt", this);

    uvm_config_db #(uvm_object_wrapper)::set(this,
"i_agt.sqr.reset_phase", "default_sequence",
router_input_port_reset_sequence::get_type());

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

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

driver.sv Solution:

```
class driver extends uvm_driver #(packet);
  virtual router_io vif;            // 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

  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);
    uvm_config_db#(virtual router_io)::get(this, "", "vif", vif);
  endfunction: build_phase

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

router_input_port_reset_sequence.sv Solution:

```
class router_input_port_reset_sequence extends uvm_sequence #(packet);
  virtual router_io vif;            // DUT virtual interface
  int               port_id = -1;   // Driver's designated port
  `uvm_object_utils_begin(router_input_port_reset_sequence)
    `uvm_field_int(port_id, UVM_DEFAULT | UVM_DEC)
  `uvm_component_utils_end
  function new(string name="router_input_port_reset_sequence");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    `ifdef UVM_POST_VERSION_1_1
     set_automatic_phase_objection(1);
    `endif
  endfunction: new
  virtual task pre_start();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    `ifdef UVM_VERSION_1_1
    if ((get_parent_sequence() == null) && (starting_phase != null)) begin
      starting_phase.raise_objection(this);
    end
    `endif
    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_info("DRV_RST_SEQ", $sformatf("Using port_id = %0d", port_id),
UVM_MEDIUM);
    uvm_config_db#(virtual router_io)::get(get_sequencer(), "", "vif", vif);
    if (vif == null) begin
      `uvm_fatal("CFGERR", "Interface for the Driver Reset Sequence not
set");
    end
  endtask: pre_start
  `ifdef UVM_VERSION_1_1
  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
  `endif
  virtual task body();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if (port_id == -1) begin
      vif.frame_n = '1;
      vif.valid_n = '1;
      vif.din = '0;
    end else begin
      vif.frame_n[port_id] = '1;
      vif.valid_n[port_id] = '1;
      vif.din[port_id] = '0;
    end
  endtask: body
endclass: router_input_port_reset_sequence
```

reset_tr.sv

```
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
```

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);

    `ifdef UVM_POST_VERSION_1_1
     set_automatic_phase_objection(1);
    `endif
  endfunction

  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

  `ifdef UVM_VERSION_1_1
  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 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
  `endif
endclass
```

4

Monitors and Scoreboard

Learning Objectives

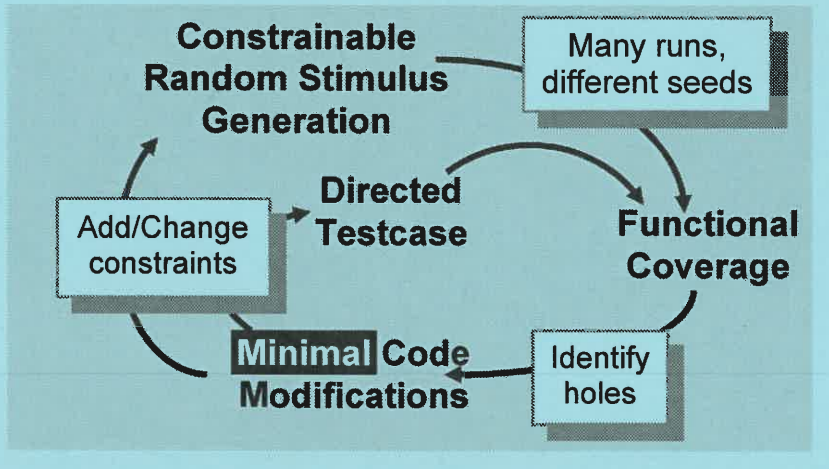
After completing this lab, you should be able to:

- Implement TLM analysis port in monitor
- Add the 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 3, 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.



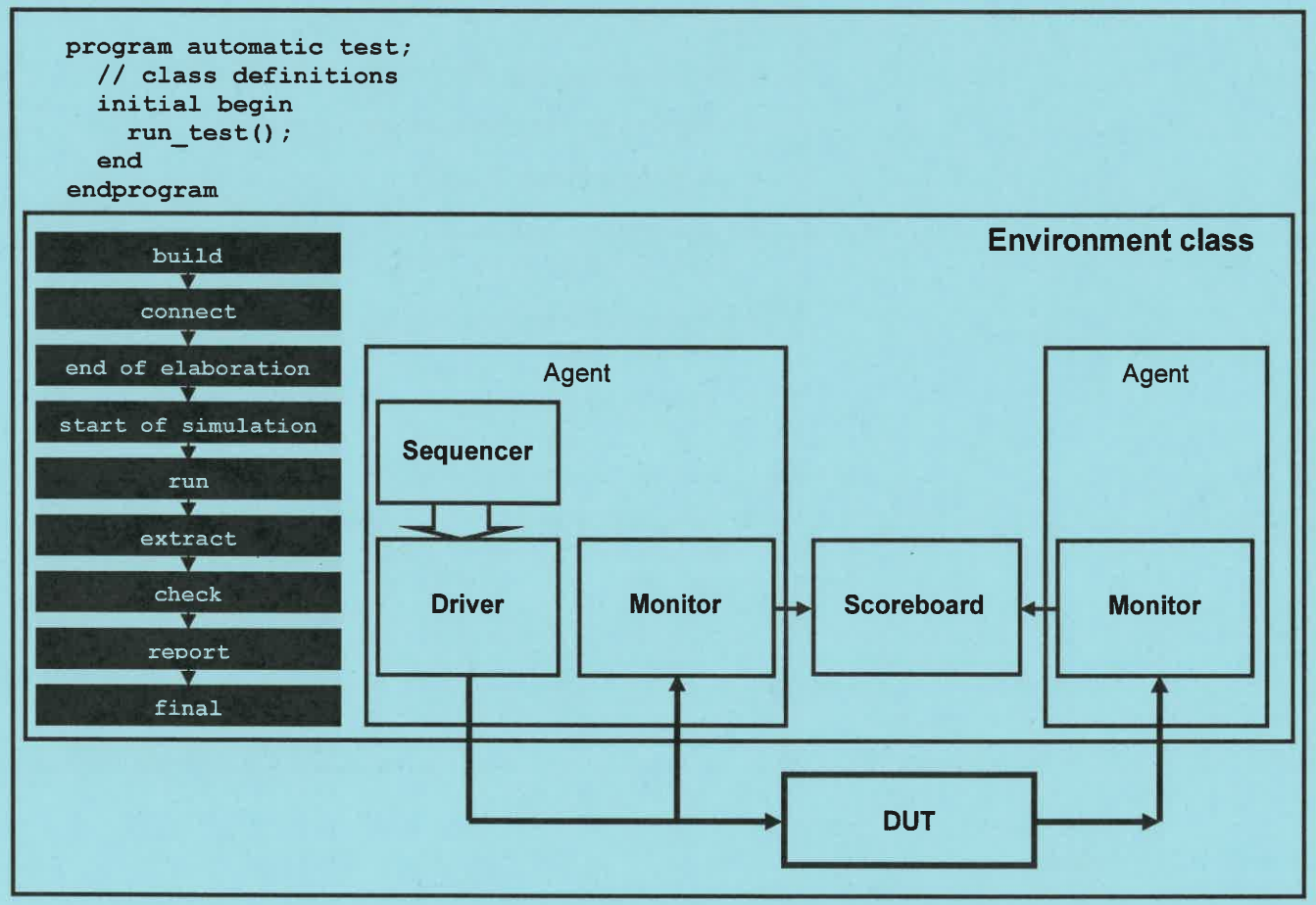


Figure 1. Lab 4 Testbench Architecture

Lab Overview

The work flow for this lab is illustrated as follows:

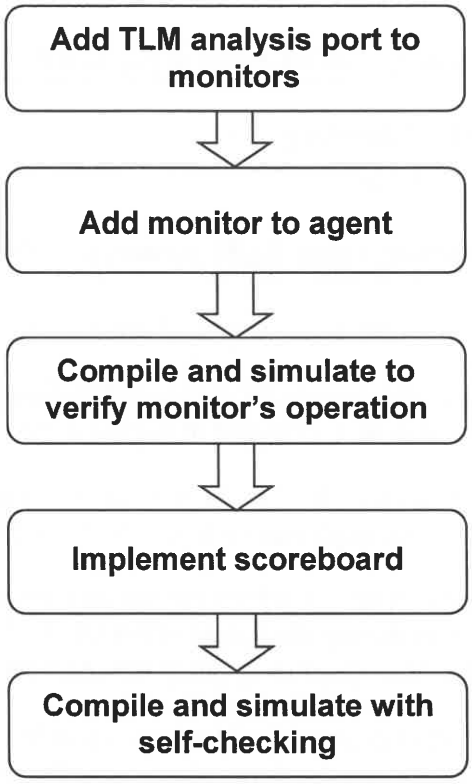


Figure 2. Lab 4 Flow Diagram

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 lab4 Working Directory

1. CD into the lab4 directory

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

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_io vif;
  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 monitor's 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. Step 5 will take you through this.

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. 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.

3. In the **build_phase()** method check the state of the **is_active** flag
   - If flag is **UVM_ACTIVE**, create the sequencer (**sqr**) and driver (**drv**) objects
4. Regardless of the state of **is_active** flag, construct the monitor (**mon**) object.
5. Construct the **analysis_port** object
6. 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**)
7. Connect the monitor's analysis port to the agent's pass-through analysis port
8. 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 **router_input_port_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_collection.sv** file in an editor
7. In build phase of **test_base** class, configure all input agents to use the **router_vif** interface.
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 TLM Port in Scoreboard

A very basic scoreboard has been created for you using the in-order comparator.

1. Open the **scoreboard.sv** file in an editor

An in-order comparator has been implemented for you.

In-order comparator typed to check packet objects

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

You need to add two TLM pass-through exports to connect the comparator to the input monitor and the output monitor.

2. Add the following TLM pass-through exports:

Passing iMonitor packet to comparator

Passing oMonitor packet to comparator

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

3. 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
```

4. 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);
```

5. 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. 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_agt[16];
```

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

4. 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_agt[i]) begin
      i_agt[i].analysis_port.connect(sb.before_export);
    end
    foreach (o_agt[i]) begin
      o_agt[i].analysis_port.connect(sb.after_export);
    end
  endfunction
```

5. Save and close the file
6. Open the **test_collection.sv** file in an editor
7. In build phase of **test_base** class, configure all output agents to use the **router_vif** interface
8. In report phase of **test_base** class, print the scoreboard content
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]    141
[Comparator Mismatch]      18
[DRV_RUN]    160
[Got_Input_Packet]    160
[Got_Output_Packet]    159
[MISCMP]     36
[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.

A better way to address the issue is to detect the expected queue in the comparator is empty. You will try out both ways.

Because the UVM comparator does not support detection of expected queue being empty, you will apply the drain time mechanism first.

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]    142
[Comparator Mismatch]     18
[DRV_RUN]    160
[Got_Input_Packet]    160
[Got_Output_Packet]    160
[MISCMP]    36
```

   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.

   This is another problem related to the in-order comparator.

How can one solve this problem? If the transactions are truly out-of-order, then one must write an out-of-order scoreboard from scratch. If the transactions are in-order on a port by port basis, but runs into race conditions among multiple ports, then the **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_pkg.sv** file in the **packages** directory
9. Add the **ms_scoreboard.sv** file to the package

```
`include "ms_scoreboard.sv"
```

10. Open the **test_collection.sv** file in an editor
11. Override the instance of the **scoreboard** to be an instance of the **ms_scoreboard**:

```
set_type_override_by_type(scoreboard::get_type(),ms_scoreboard::get_type());
```

12. Compile and simulate the testbench.

    > make

    All should now match!

Task 9. End Test Properly!

Ending a test based on the drain time should bother you. The reason that you were shown the drain time mechanism is that it can be useful during debugging when there are uncertainties of what may be going wrong. But, it is not to be used for production worthy tests.

The most common way to detect when to end the test is to depend on the scoreboard. Within the scoreboard, there is typically a mechanism built-in to detect when the expected queue reaches empty state. This state is when the test should terminate. Such a mechanism is provided in the ms_scoreboard class. You can enable the mechanism by calling the **wait_for_done()** method.

Take a look at the class code if interested. Otherwise, modify the test to use this mechanism

1. Open the **test_collection.sv** file in an editor
2. Comment out the entire **main_phase()** method where the drain time code resides
3. In the shutdown phase, call the scoreboard's **wait_for_done()** method

4. Compile and simulate the testbench.

   > make

   You should see that only 159 packets on the output observed. Test terminated too early again! What happened?

```
** Report counts by id
...
[Comparator Match]    159
...
[Got_Input_Packet]    160
[Got_Output_Packet]    159
```

   Remember the way that UVM task phases work. If one does not raise objection in a task phase, that phase may terminate in 0 time.

   This is what's happening with your code. Even though you called the scoreboard's **wait_for_done()** method to wait for the expected queue to empty, without raising the phase objection, the shutdown phase terminated in 0 time. Thus, the wait never happened.

5. Edit the **test_collection.sv** file

6. Add raise and drop phase objection around the call to **wait_for_done()**

7. Compile and simulate the testbench.

   > make

   All should now match!

8. Check the waveform

   > make dve

   Or,

   > make verdi

   You should see that packets were driven through the DUT.

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

   > make test=test_da_3_seq

10. Take a look at the waveform again

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

Congratulations, you have completed Lab 4!

Answers / Solutions

test_collection.sv Solution:

```
class test_base extends uvm_test;
  `uvm_component_utils(test_base)
  router_env env;
  virtual router_io router_vif;
  virtual reset_io  reset_vif;

  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);

    uvm_resource_db#(virtual router_io)::read_by_type("router_vif",
router_vif, this);
    uvm_resource_db#(virtual reset_io)::read_by_type("reset_vif",
reset_vif, this);
    uvm_config_db#(virtual router_io)::set(this, "env.i_agt[*]", "vif",
router_vif);
    uvm_config_db#(virtual router_io)::set(this, "env.o_agt[*]", "vif",
router_vif);
    uvm_config_db#(virtual reset_io)::set(this, "env.r_agt", "vif",
reset_vif);
set_type_override_by_type(scoreboard::get_type(),ms_scoreboard::get_type(
));
  endfunction: build_phase

/*
  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: main_phase
*/
  virtual task shutdown_phase(uvm_phase phase);
    super.shutdown_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);

    phase.raise_objection(this);
    env.sb.wait_for_done();
    phase.drop_objection(this);
  endtask: shutdown_phase

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

endclass: test_base
```

router_env.sv Solution:

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

  reset_agent r_agt;
  input_agent i_agt[16];
  scoreboard sb;
  output_agent o_agt[16];

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

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

    sb = scoreboard::type_id::create("sb", this);
    foreach (o_agt[i]) begin
      o_agt[i] = output_agent::type_id::create($sformatf("o_agt[%0d]", i),
this);
      uvm_config_db #(int)::set(this, o_agt[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_agt[i]) begin
      i_agt[i].analysis_port.connect(sb.before_export);
    end
    foreach (o_agt[i]) begin
      o_agt[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     sqr;
  driver               drv;
  virtual router_io    vif;
  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
      sqr = packet_sequencer::type_id::create("sqr", 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_io)::get(this, "", "vif", vif);

    uvm_config_db#(int)::set(this, "*", "port_id", port_id);
    uvm_config_db#(virtual router_io)::set(this, "*", "vif", vif);
  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(sqr.seq_item_export);
    end

    mon.analysis_port.connect(this.analysis_port);
  endfunction: connect_phase

  virtual function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if (!(port_id inside {-1, [0:15]})) begin
      `uvm_fatal("CFGERR", $sformatf("port_id must be {-1, [0:15]}, not
%0d!", port_id));
    end
    if (vif == null) begin
      `uvm_fatal("CFGERR", "Interface for input agent not set");
    end
  endfunction: end_of_elaboration_phase

endclass: input_agent
```

iMonitor.sv Solution:

```
class iMonitor extends uvm_monitor;
  virtual router_io vif;
  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);
    uvm_config_db#(virtual router_io)::get(this, "", "vif", vif);

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

  virtual function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if (!(port_id inside {-1, [0:15]})) begin
      `uvm_fatal("CFGERR", $sformatf("port_id must be {-1, [0:15]}, not
%0d!", port_id));
    end
    if (vif == null) begin
      `uvm_fatal("CFGERR", "Interface for iMonitor not set");
    end
  endfunction: end_of_elaboration_phase

  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: iMonitor
```

oMonitor.sv Solution:

```
class oMonitor extends uvm_monitor;
  int port_id = -1;
  virtual router_io vif;
  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);
    uvm_config_db#(virtual router_io)::get(this, "", "vif", vif);
    analysis_port = new("analysis_port", this);
  endfunction

  virtual function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if (!(port_id inside {-1, [0:15]})) begin
      `uvm_fatal("CFGERR", $sformatf("port_id must be {-1, [0:15]}, not
%0d!", port_id));
    end
    if (vif == null) begin
      `uvm_fatal("CFGERR", "Interface for oMonitor not set");
    end
  endfunction: end_of_elaboration_phase

  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: oMonitor
```

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: new

  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: build_phase

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

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

  virtual function string convert2string();
    return $sformatf("Comparator Matches = %0d, Mismatches = %0d",
comparator.m_matches, comparator.m_mismatches);
  endfunction: convert2string

  // The following are supplemental methods for detecting end of test and
reporting results.
  // They will be implemented in the derived classes.
  virtual task wait_for_done(); endtask
  virtual function void set_timeout(realtime timeout); endfunction
  virtual function realtime get_timeout(); endfunction

endclass: scoreboard
```

ms_scoreboard.sv Solution:

```
class ms_scoreboard extends scoreboard;
  `uvm_analysis_imp_decl(_before)
  `uvm_analysis_imp_decl(_after)

  uvm_analysis_imp_before #(packet, ms_scoreboard) ms_before_export;
  uvm_analysis_imp_after  #(packet, ms_scoreboard) ms_after_export;
  packet_cmp comparator[16];
  int count = 0;
  realtime timeout = 10us;

  `uvm_component_utils(ms_scoreboard)

  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", $sformatf("%m"), UVM_HIGH);
    ms_before_export = new("ms_before_export", this);
    ms_after_export  = new("ms_after_export", this);
    for (int i=0; i < 16; i++) begin
      comparator[i] = uvm_in_order_class_comparator
#(packet)::type_id::create($sformatf("comparator_%0d", i), this);
    end
  endfunction: build_phase

  virtual function void connect_phase(uvm_phase phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    before_export.connect(ms_before_export);
    after_export.connect(ms_after_export);
  endfunction: connect_phase

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

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

  virtual task wait_for_done();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    fork
      begin
        fork
          wait(count == 0);
          begin
            #timeout;
```

```
            `uvm_warning("TIMEOUT", $sformatf("Scoreboard has %0d
unprocessed expected objects", count));
          end
        join_any
        disable fork;
      end
    join
  endtask: wait_for_done

  virtual function void set_timeout(realtime timeout);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    this.timeout=timeout;
  endfunction: set_timeout

  virtual function realtime get_timeout();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    return (timeout);
  endfunction: get_timeout

  virtual function string convert2string();
    foreach (comparator[i]) begin
      convert2string = {convert2string, $sformatf("Comparator[%0d]
Matches = %0d, Mismatches = %0d\n", i, comparator[i].m_matches,
comparator[i].m_mismatches)} ;
    end
  endfunction: convert2string

endclass: ms_scoreboard
```

5 Managing Sequences

Learning Objectives

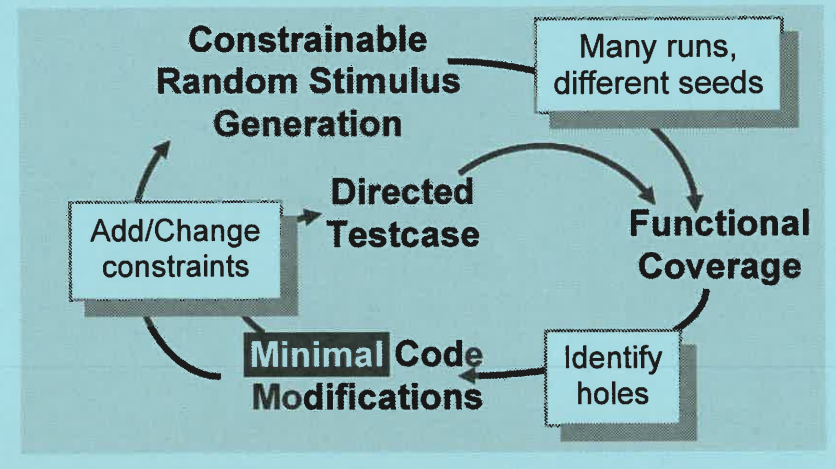
After completing this lab, you should be able to:

- Develop a top level sequence to control reset sequence execution order
- Implement test to execute the next sequence

Lab Duration:
30 minutes

Getting Started

Through the first four 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.



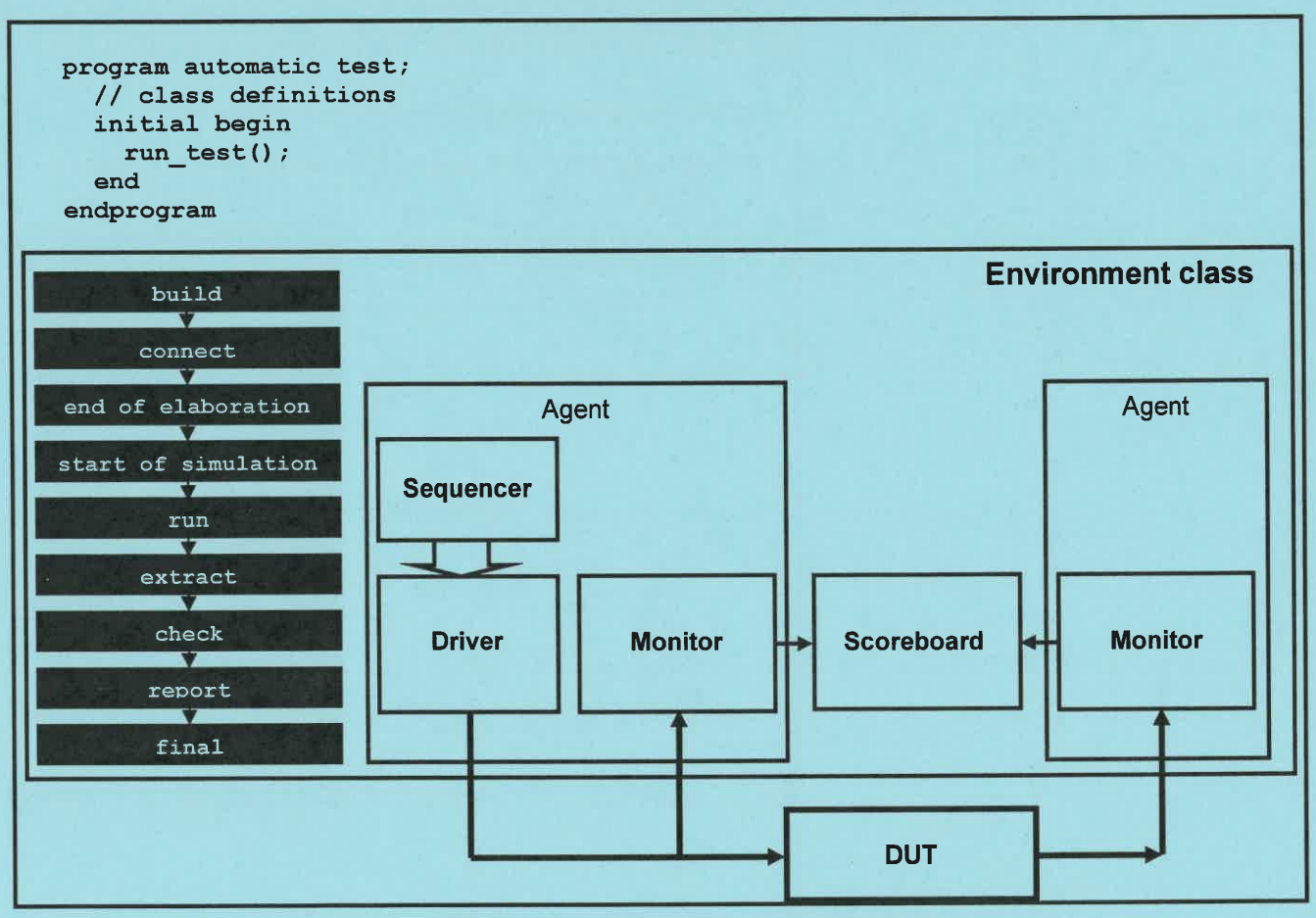


Figure 1. Lab 5 Testbench Architecture

Lab Overview

The work flow for this lab is illustrated as follows:

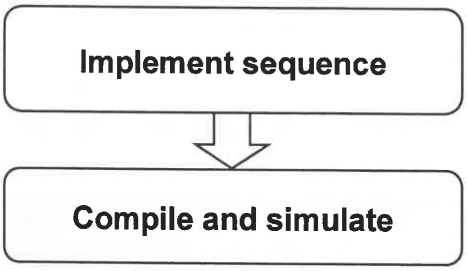


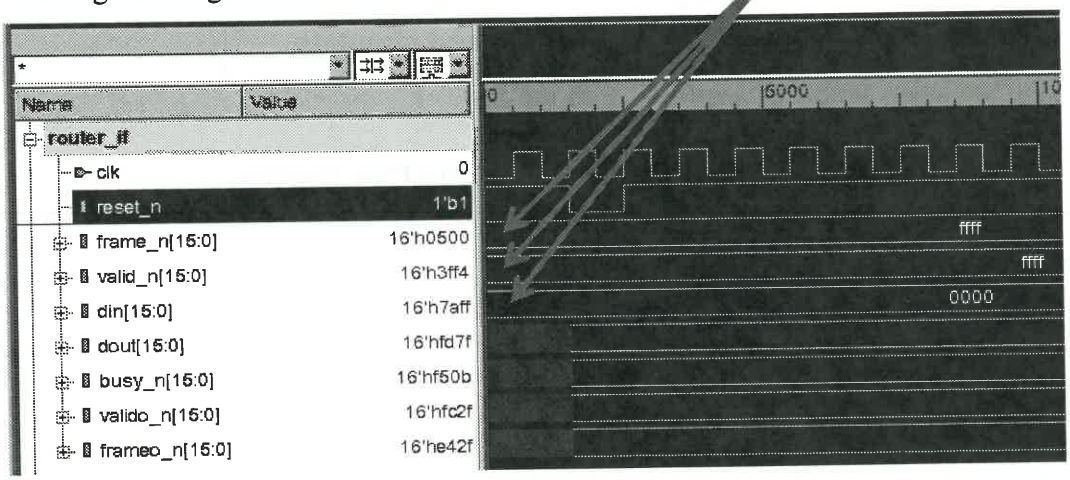
Figure 2. Lab 5 Flow Diagram

Reset Sequence

In lab 4, you sucessfully drove 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 4:

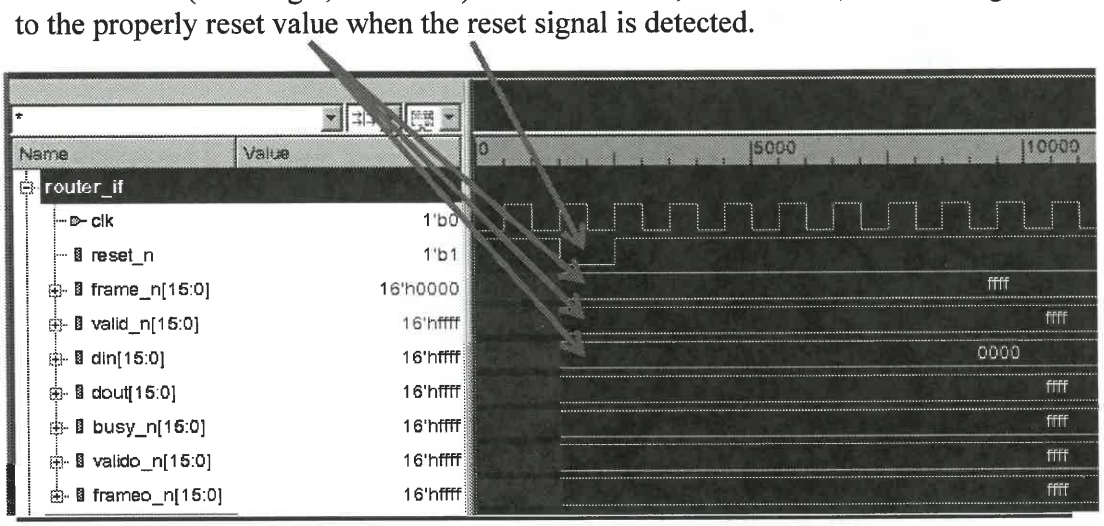
```
class router_env extends uvm_env;
 // A lot of code left off.  Only relevant code is shown

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

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


  foreach (i_agt[i]) begin
   uvm_config_db#(uvm_object_wrapper)::set(this, {i_agt[i].get_name(),
    ".sqr.reset_phase"}, "default_sequence",
router_input_port_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 value when the reset signal is detected.



The way to manage this in UVM is to implement a top reset sequence and a top reset sequencer.

Task 1. Implement Top Reset Sequence

For this task, you will create a top reset sequence to manage the execution of the existing **`reset_sequence`** and **`router_input_port_reset_sequence`**.

To reduce lab time, the top reset sequencer is created for you.

1. CD into the lab5 directory and open **`top_reset_sequencer.sv`** to take a look and make sure you understand why the sequencer is declared as it is
2. Exit the file
3. Open **`top_reset_sequence.sv`** file in an editor
4. Inside the class, use the **`` `uvm_declare_p_sequencer ``** macro to specify the associated sequencer type
5. Continuing in the class, create a **`reset_sequence`** handle called **`r_seq`** and a **`router_input_port_reset_sequence`** handle called **`i_seq`**
6. Inside the **`body()`** method do the following:
   - Execute the **`reset_sequence`** instance with the parent sequencer's **`r_sqr`**
   - Iterate through the parent sequencer's **`pkt_sqr`** queue and execute the the **`router_input_port_reset_sequence`**
7. Save and close the file

Task 2. Execute Top Sequence in Test

Because top sequence executions are test specific, you will manage them in the test class.

1. Open **test_collection.sv** file in an editor
2. Inside the **test_base** class, create a **top_reset_sequencer** handle called **top_reset_sqr**
3. In the **build_phase**, construct this sequencer

   Because the top reset sequence will control all reset sequence executions, you will need to turn off the existing reset sequence executions in the environment. Otherwise, both sequence execution will occur.
4. Turn off the **r_agt**'s sequencer execution at the reset phase by setting **"default_sequence"** to **null**
5. And, turn off all the **i_agt**'s sequencer execution at the reset phase by setting **"default_sequence"** to **null**
6. Then, configure the top reset sequencer to execute the top reset sequence at the reset phase

```
virtual function void build_phase(uvm_phase phase);
  ...;
  uvm_config_db #(uvm_object_wrapper)::set(this, "env.r_agt.sqr.reset_phase",
                                                 "default_sequence", null);
  uvm_config_db #(uvm_object_wrapper)::set(this, "env.i_agt[*].sqr.reset_phase"},
                                                 "default_sequence", null);
  uvm_config_db #(uvm_object_wrapper)::set(this, "top_reset_sqr.reset_phase",
                                "default_sequence", top_reset_sequence::get_type());
  ...;
endfunction
```

7. In the **connect_phase**, push the input agent's sequencers onto the top reset sequencer's **pkt_sqr** queue.
8. Continuing in the **connect_phase**, set the top sequencer's **r_sqr** handle to reference the reset agent's sequencer (**r_agt.sqr**)
9. Compile and simulate the testbench to see if this works

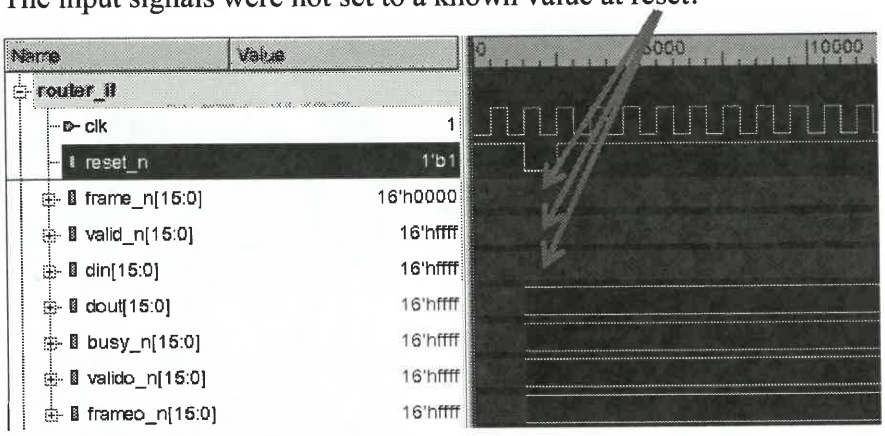
   > make

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

10. Bring up debugger to see what's happening

    > make dve or **make verdi**

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



The reason is because, in the top 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_sqr);
  foreach (p_sequencer.pkt_sqr[i]) begin
    `uvm_do_on(i_seq, p_sequencer.pkt_sqr[i]);
  end
endtask
```

Let's correct this.

Task 3. Execute Sequences Concurrently

1. Open **top_reset_sequence.sv** file in an editor
2. Comment out the code from Task 1, Step 6 and un-comment the following code:

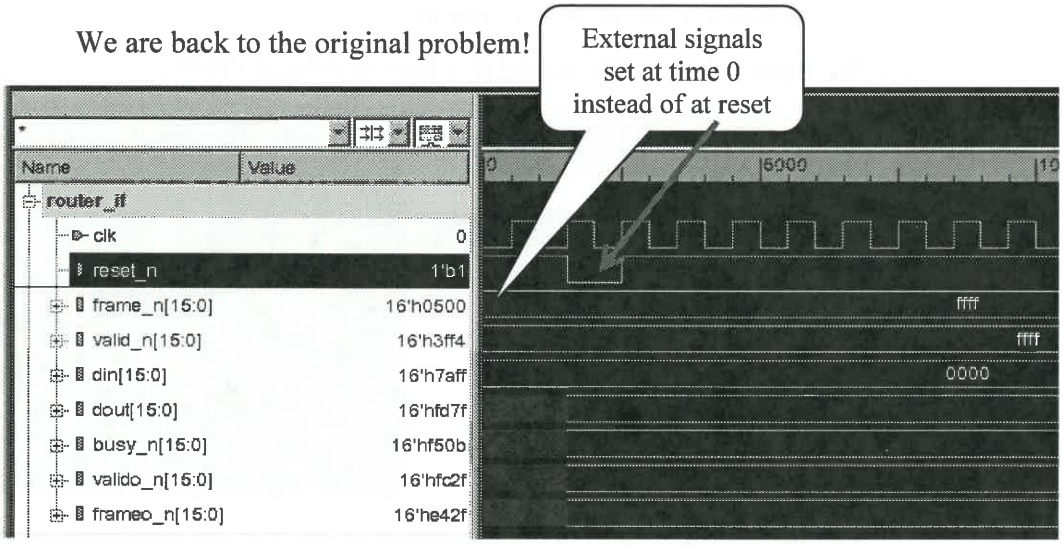
```
fork
  `uvm_do_on(r_seq, p_sequencer.r_sqr);
  foreach (p_sequencer.pkt_sqr[i]) begin
    fork
      int j = i;
      `uvm_do_on(i_seq, p_sequencer.pkt_sqr[j]);
    join_none
  end
join
```

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? View it in debugger to make sure

We are back to the original problem!



This is a 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_io vif;   // 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);
  @(vif.reset_n);
  assert(!$isunknown(vif.reset_n));
  if (vif.reset_n == 1'b0) begin
    tr.kind = reset_tr::ASSERT;
    reset_event.trigger();
  end else begin
    tr.kind = reset_tr::DEASSERT;
    reset_event.reset(.wakeup(1));
  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 **top_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 un-comment the following code:

```
fork
  `uvm_do_on(r_seq, p_sequencer.r_sqr);
  foreach (p_sequencer.pkt_sqr[i]) begin
    fork
      int j = i;
      begin
        reset_event.wait_on();
        `uvm_do_on(i_seq, p_sequencer.pkt_sqr[j]);
      end
    join_none
  end
join
```

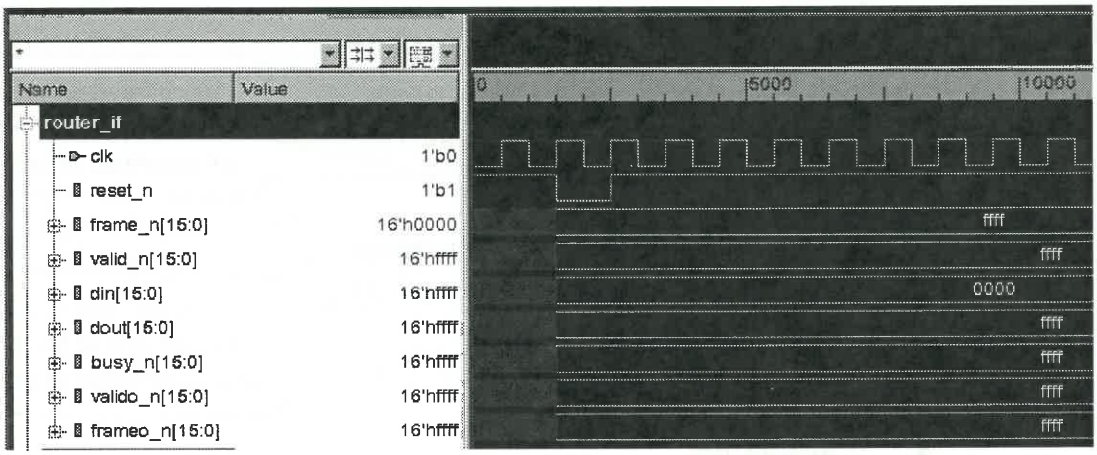
Wait for the "reset" uvm_event to turn on before executing the driver reset sequence

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

```
> make
```

10. Did it work? Again bring up debugger to make sure

You should now see the following correct timing:



Congratulations, you have completed Lab 5!

Answers / Solutions

top_reset_sequencer.sv Solution:

```
class top_reset_sequencer extends uvm_sequencer;
  `uvm_component_utils(top_reset_sequencer)
  typedef uvm_sequencer#(reset_tr) reset_sequencer;
  typedef uvm_sequencer#(packet) packet_sequencer;
  reset_sequencer  r_sqr;
  packet_sequencer pkt_sqr[$];

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

top_reset_sequence.sv Solution:

```
class top_reset_sequence extends uvm_sequence;
  `uvm_object_utils(top_reset_sequence)
  `uvm_declare_p_sequencer(top_reset_sequencer)

  reset_sequence                    r_seq;
  router_input_port_reset_sequence  i_seq;
  uvm_event reset_event = uvm_event_pool::get_global("reset");

  function new(string name="top_reset_sequence");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    `ifdef UVM_POST_VERSION_1_1
     set_automatic_phase_objection(1);
    `endif
  endfunction
  virtual task body();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    fork
      `uvm_do_on(r_seq, p_sequencer.r_sqr);
      foreach (p_sequencer.pkt_sqr[i]) begin
        fork
          int j = i;
          begin
            reset_event.wait_on();
            `uvm_do_on(i_seq, p_sequencer.pkt_sqr[j]);
          end
        join_none
      end
    join
  endtask

  `ifdef UVM_VERSION_1_1
  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 post_start();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
```

```
    if ((get_parent_sequence() == null) && (starting_phase != null))
      starting_phase.drop_objection(this);
  endtask
  `endif
endclass
```

test_collection.sv Solution:

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

  router_env env;
  virtual router_io router_vif;
  virtual reset_io  reset_vif;
  top_reset_sequencer  top_reset_sqr;

  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);

    uvm_resource_db#(virtual router_io)::read_by_type("router_vif", router_vif,
this);
    uvm_resource_db#(virtual reset_io)::read_by_type("reset_vif", reset_vif,
this);
    uvm_config_db#(virtual router_io)::set(this, "env.i_agt[*]", "vif",
router_vif);
    uvm_config_db#(virtual router_io)::set(this, "env.o_agt[*]", "vif",
router_vif);
    uvm_config_db#(virtual reset_io)::set(this, "env.r_agt", "vif", reset_vif);

    top_reset_sqr = top_reset_sequencer::type_id::create("top_reset_sqr", this);

    uvm_config_db #(uvm_object_wrapper)::set(this, "env.r_agt.sqr.reset_phase",
"default_sequence", null);
    uvm_config_db #(uvm_object_wrapper)::set(this,
"env.i_agt[*].sqr.reset_phase", "default_sequence", null);

    uvm_config_db #(uvm_object_wrapper)::set(this, "top_reset_sqr.reset_phase",
"default_sequence", top_reset_sequence::get_type());
  endfunction: build_phase

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    foreach (env.i_agt[i]) begin
      top_reset_sqr.pkt_sqr.push_back(env.i_agt[i].sqr);
    end

    top_reset_sqr.r_sqr = env.r_agt.sqr;
  endfunction: connect_phase

// Other code not shown
endclass: test_base
```

6

UVM Register Abstraction Layer (RAL)

Learning Objectives

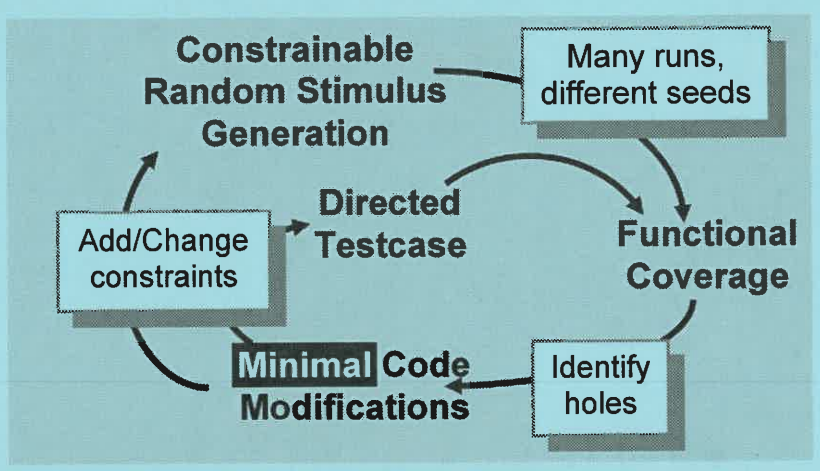
After completing this lab, you should be able to:

- 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 adopter 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.



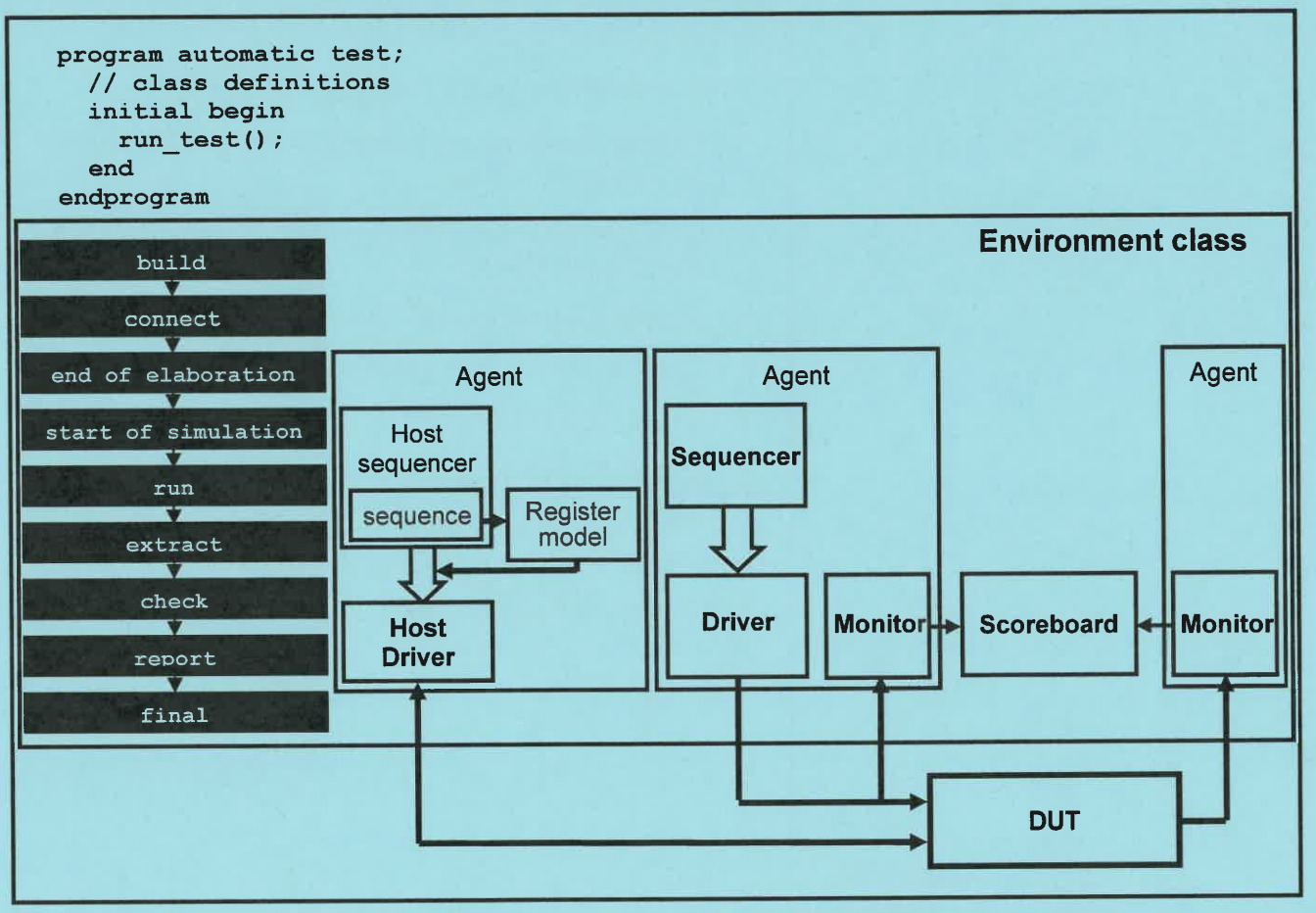


Figure 1. Lab 6 Testbench Architecture

Lab Overview

The work flow for this lab is illustrated as follows:

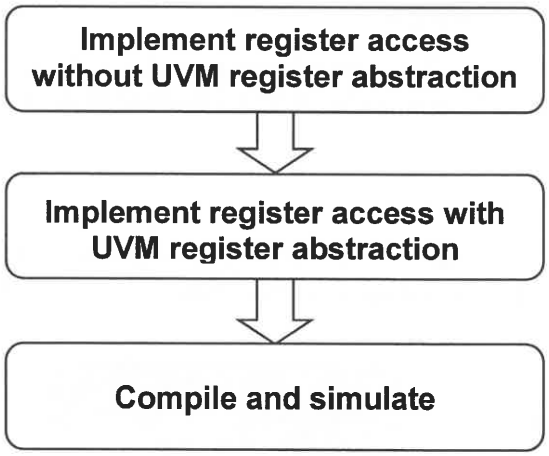


Figure 2. Lab 6 Flow Diagram

Implement UVM Register Abstraction

Task 1. Go into lab6 Working Directory

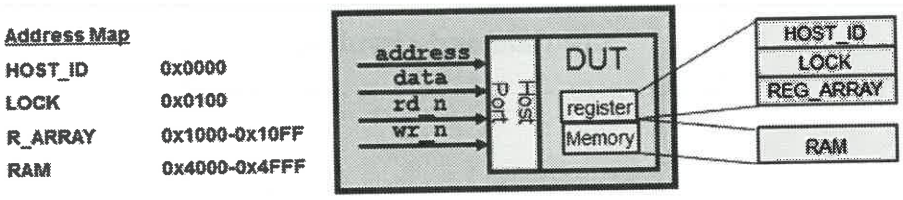
1. CD into the lab6 directory

```
> cd ../lab6
```

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).



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	0xffff

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	0x0000

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

RAM (4K)	
Bits	15-0
Mode	rw

Task 2. Add Host Agent to Environment

The host agent class has been created for you. (Just like all other agents with driver, monitor and sequencer) Add it to the router environment.

1. Open **`router_env.sv`** file in an editor
2. In the class, declare a **`host_agent`** handle, call it **`h_agt`**
3. In build phase, construct the **`h_agt`** object
4. Save and close the file

Task 3. Configure Host Agent with Interface

The configuration of the host interface is done for you. Take a look at the **`test.sv`** and **`test_collection.sv`** files if interested. Otherwise, continue on to Task 4.

Task 4. Add Host Reset to Top Reset Sequence

Like all other agents, you need to execute a reset sequence for the host interface at the reset phase. To conserve lab time, this sequence is written for you. If interested, take a look at the **`host_reset_sequence`** class in the **`host_sequence.sv`** file. Otherwise, add this **`host_reset_sequence`** to the top reset sequence.

1. Open **`top_reset_sequencer.sv`** file in an editor
2. Add a **`host_sequencer`** handle, call it **`h_sqr`**
3. Save and close the file
4. Open **`top_reset_sequence.sv`** file in an editor

5. 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_sqr** when reset is detected
7. Save and close the file

   You need to make the host agent accessible via the top reset sequencer.
8. Open **test_collection.sv** in an editor
9. Locate the **connect_phase()** method
10. Assign the host sequencer handle in the top reset sequencer to reference the host sequencer in the environment
11. Save and close the file
12. Compile and run simulation

    > make

    You will see a UVM_WARNING message. This is expected at this juncture of the testbench development. Ignore this for now.
13. Open up a debugger window

    > make dve or **make verdi**
14. Check the waveform and make sure that the reset set the host control signals (wr_n, rd_n) are de-asserted as a result of the reset.

    Once the reset works, DUT register access development can begin.

Task 5. Create Sequence without Register Abstraction

The first step in implementing DUT register access is to implement all classes 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 extends uvm_sequence_item;
  rand uvm_access_e kind;
  rand uvm_status_e status;
  rand bit[15:0]    addr;
  rand bit[15:0]    data;
  ...
endclass
```

Generate a few transaction to verify that the interface works.

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 6. Execute the Host BFM Sequence

1. Open **test_collection.sv** in an editor
2. Locate the **test_host_bfm** class
3. 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.
4. Save and close the file
5. Verify the DUT registers can be accessed:

```
> make test=test_host_bfm
```

Task 7. Attempt to Drive Packets through DUT

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

1. Execute test_base:

```
> make
```

   The expected result is: input packets are seen, but no output packets are observed. It 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 8. 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 (or the solution page)

   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
LOCK	0x0100
R_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

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

R_ARRAY[256] Registers	
Field	H_REG
Bits	15-0
Mode	rw
Reset	0x0000

RAM (4K)	
Bits	15-0
Mode	rw

3. Close the file when done

Task 9. 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`**.

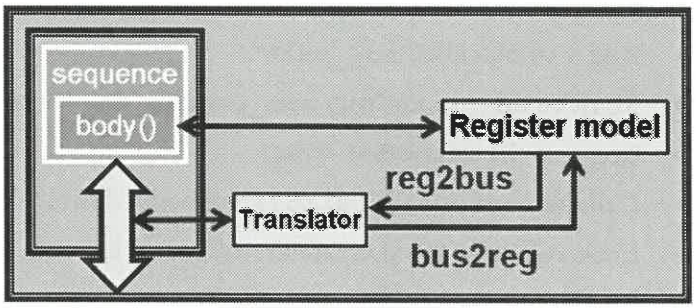
   If interested, take a look at the content. Otherwise, continue on to the next task.

Task 10. Create Sequence Using UVM Registers

1. Open **`host_sequence.sv`** file in an editor
2. Locate the RAL sequence base class called **`host_ral_sequence_base`**
3. Inside the class, create an instance of **`ral_block_host_regmodel`** called **`regmodel`**
4. In the **`pre_start()`** method, use **`uvm_config_db`** to retrieve the register model
5. Locate the RAL test sequence class called **`host_ral_test_sequence`**
6. 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.
7. When done, save and close the file.

Task 11. 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 (**`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 12. 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_agt;
  host_agent               h_agt;
  ral_block_host_regmodel  regmodel;
  reg_adapter              adapter;
```

3. In the build phase construct the adapter object

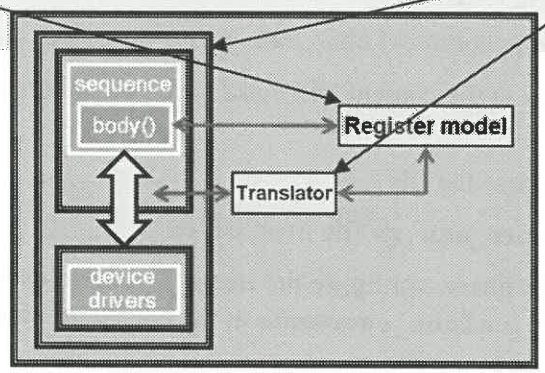
```
  adapter = reg_adapter::type_id::create("adapter", this);
```

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_agt.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_agt.sqr**) and RAL translator handle (**adapter**).

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



8. When done, save and close the file.

Task 13. 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, turn off sequences at the configure and main phase
5. Then, set the host agent's sequencer at the **main_phase** to execute **host_ral_test_sequence**
6. Save and close the file

Task 14. 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 15. 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 16. 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 17. 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
   - Turn off auto predict
   - 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

```
regmodel.default_map.set_auto_predict(0);
hreg_predict.map = regmodel.get_default_map();
hreg_predict.adapter = adapter;
h_agt.analysis_port.connect(hreg_predict.bus_in);
```

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

   > make

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

Task 18. 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_sqr);
  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_agt.sqr);
  phase.drop_objection(this, "Done with register tests");
 endtask
endclass
```

4. Save and close the file
5. Compile and simulate the testbench to run the bit bash test

```
> 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 19. 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 6!

Answers / Solutions

test.sv Solution:

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

initial begin
  uvm_resource_db#(virtual router_io)::set("router_vif", "",
router_test_top.router_if);
  uvm_resource_db#(virtual reset_io)::set("reset_vif", "",
router_test_top.reset_if);
  uvm_resource_db#(virtual host_io)::set("host_vif", "",
router_test_top.host_if);

  $timeformat(-9, 1, "ns", 10);
  uvm_reg::include_coverage("*", UVM_CVR_ALL);
  run_test();
end
endprogram
```

test_collection.sv Solution:

```
class test_base extends uvm_test;
  `uvm_component_utils(test_base)
  uvm_cmdline_processor clp = uvm_cmdline_processor::get_inst();

  router_env env;
  virtual router_io    router_vif;
  virtual reset_io     reset_vif;
  virtual host_io      host_vif;

  top_reset_sequencer  top_reset_sqr;

  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);

    uvm_resource_db#(virtual router_io)::read_by_type("router_vif",
router_vif, this);
    uvm_resource_db#(virtual reset_io)::read_by_type("reset_vif",
reset_vif, this);

    uvm_config_db#(virtual router_io)::set(this, "env.i_agt[*]", "vif",
router_vif);
    uvm_config_db#(virtual router_io)::set(this, "env.o_agt[*]", "vif",
router_vif);
    uvm_config_db#(virtual reset_io)::set(this, "env.r_agt", "vif",
reset_vif);

    top_reset_sqr = top_reset_sequencer::type_id::create("top_reset_sqr",
this);

    uvm_resource_db#(virtual host_io)::read_by_type("host_vif", host_vif,
this);
    uvm_config_db#(virtual host_io)::set(this, "env.h_agt", "vif",
host_vif);

    uvm_config_db #(uvm_object_wrapper)::set(this,
"env.*.sqr.reset_phase", "default_sequence", null);

    uvm_config_db #(uvm_object_wrapper)::set(this,
"top_reset_sqr.reset_phase", "default_sequence",
top_reset_sequence::get_type());

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

set_type_override_by_type(scoreboard::get_type(),ms_scoreboard::get_type(
));
  endfunction: build_phase
```

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

    foreach (env.i_agt[i]) begin
      top_reset_sqr.pkt_sqr.push_back(env.i_agt[i].sqr);
    end

    top_reset_sqr.r_sqr = env.r_agt.sqr;

    top_reset_sqr.h_sqr = env.h_agt.sqr;
  endfunction: connect_phase

  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: end_of_elaboration_phase

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

    phase.raise_objection(this);
    env.sb.wait_for_done();
    phase.drop_objection(this);
  endtask: shutdown_phase

  virtual function void report_phase(uvm_phase phase);
    super.report_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    `uvm_info("SB_REPORT", {"\n", env.sb.convert2string()}, UVM_MEDIUM);
  endfunction: report_phase

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

    if (uvm_report_enabled(UVM_DEBUG, UVM_INFO, "TOPOLOGY")) begin
      uvm_root::get().print_topology();
    end

    if (uvm_report_enabled(UVM_DEBUG, UVM_INFO, "FACTORY")) begin
      uvm_factory::get().print();
    end
  endfunction: final_phase
endclass: test_base

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: new
```

```
  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_agt[*].sqr.*", packet::get_type(),
packet_da_3::get_type());
  endfunction: build_phase
endclass: test_da_3_inst

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: new

  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: build_phase
endclass: test_da_3_type

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: new

  virtual function void build_phase(uvm_phase phase);
    packet_sequence::int_q_t valid_da = {3};
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    uvm_config_db#(packet_sequence::int_q_t)::set(this,
"env.i_agt[*].sqr.packet_sequence", "valid_da", valid_da);
    uvm_config_db#(int)::set(this, "env.i_agt[*].sqr.packet_sequence",
"item_count", 20);
  endfunction: build_phase
endclass: test_da_3_seq

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: new

  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_agt.sqr.main_phase", "default_sequence",
host_bfm_sequence::get_type());

  endfunction: build_phase
endclass: test_host_bfm

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: new

  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_agt.sqr.main_phase", "default_sequence",
host_ral_test_sequence::get_type());

  endfunction: build_phase
endclass: test_host_ral


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;
  top_reset_sequence      top_reset_seq;

  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);
    uvm_config_db
#(uvm_object_wrapper)::set(this,"*","default_sequence",null);
  endfunction: build_phase

  virtual task run_phase(uvm_phase phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    phase.raise_objection(this, "Starting reset tests");
    top_reset_seq = top_reset_sequence::type_id::create("top_reset_seq",
this);
    top_reset_seq.start(top_reset_sqr);
    clp.get_arg_value("+seq=", seq_name);
```

```
    $cast(selftest_seq,
uvm_factory::get().create_object_by_name(seq_name));
    selftest_seq.model = env.regmodel;
    selftest_seq.start(env.h_agt.sqr);
    phase.drop_objection(this, "Done with register tests");
  endtask: run_phase
endclass: test_ral_selftest
```

host_data.sv Solution:

```
class host_data extends uvm_sequence_item;
  rand uvm_access_e kind;
  rand uvm_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(uvm_access_e, kind, UVM_ALL_ON)
    `uvm_field_enum(uvm_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);
    `uvm_info("Trace", $sformatf("%m"), UVM_HIGH);
    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);
    `uvm_info("Trace", $sformatf("%m"), UVM_HIGH);
  endfunction
  virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
    host_data tr;
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    tr = host_data::type_id::create("tr");
    tr.addr = rw.addr;
    tr.data = rw.data;
    tr.kind = rw.kind;
    return tr;
  endfunction
  virtual function void bus2reg(uvm_sequence_item bus_item, ref
uvm_reg_bus_op rw);
    host_data tr;
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if (!$cast(tr, bus_item)) begin
      `uvm_fatal("NOT_HOST_REG_TYPE", "bus_item is not correct type");
    end
    rw.addr = tr.addr;
    rw.data = tr.data;
    rw.kind   = tr.kind;
    rw.status = tr.status;
  endfunction
endclass
```

host_sequence.sv Solution:

```
class host_sequence_base extends uvm_sequence #(host_data);
  `uvm_object_utils(host_sequence_base)

  virtual host_io    vif;
  uvm_sequencer_base p_sqr;

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

    `ifdef UVM_POST_VERSION_1_1
     set_automatic_phase_objection(1);
    `endif
  endfunction

  virtual task pre_start();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    p_sqr = get_sequencer();

    `ifdef UVM_VERSION_1_1
    if ((get_parent_sequence() == null) && (starting_phase != null)) begin
      starting_phase.raise_objection(this);
    end
    `endif

    if (uvm_config_db#(virtual host_io)::get(p_sqr.get_parent(), "", "vif",
vif)) begin
      `uvm_info("HOST_SEQ_CFG", "Has access to host interface", UVM_HIGH);
    end
  endtask

  `ifdef UVM_VERSION_1_1
  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
  `endif
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();
    vif.wr_n = 1'b1;
    vif.rd_n = 1'b1;
    vif.address ='z;
    vif.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 == UVM_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 == UVM_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 == UVM_WRITE;});
    `uvm_do_with(req, {addr == 'h100; kind == UVM_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 ==
UVM_WRITE;});
    end
    for (int i=0; i<256; i++) begin
      `uvm_do_with(req, {addr == 'h1000+i; kind == UVM_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);

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_sqr.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);
  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 LOCK {
  field LOCK {
    bits 16;
    access w1c;
    reset 'hffff;
  }
}

register R_ARRAY {
  field H_REG {
    bits 16;
    access rw;
    reset 'h0000;
  }
}

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;
  `uvm_component_utils(router_env)

  reset_agent   r_agt;
  input_agent   i_agt[16];
  output_agent  o_agt[16];
  scoreboard    sb;
  host_agent  h_agt;
  ral_block_host_regmodel regmodel;
  reg_adapter adapter;

  typedef uvm_reg_predictor #(host_data) hreg_predictor;
  hreg_predictor hreg_predict;

  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);

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

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

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

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

    h_agt = host_agent::type_id::create("h_agt", 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_agt.get_name(),
"regmodel", regmodel);
    uvm_config_db #(uvm_object_wrapper)::set(this, {h_agt.get_name(), ".",
"sqr.configure_phase"}, "default_sequence",
ral_port_unlock_sequence::get_type());

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

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

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

    regmodel.default_map.set_sequencer(h_agt.sqr, adapter);

    regmodel.default_map.set_auto_predict(0);
    hreg_predict.map = regmodel.get_default_map();
    hreg_predict.adapter = adapter;
    h_agt.analysis_port.connect(hreg_predict.bus_in);

  endfunction: connect_phase

endclass: router_env
```

top_reset_sequence.sv Solution:

```
class top_reset_sequence extends uvm_sequence;
  `uvm_object_utils(top_reset_sequence)
  `uvm_declare_p_sequencer(top_reset_sequencer)
  reset_sequence                    r_seq;
  router_input_port_reset_sequence  i_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);

    `ifdef UVM_POST_VERSION_1_1
     set_automatic_phase_objection(1);
    `endif
  endfunction: new

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

    fork
      `uvm_do_on(r_seq, p_sequencer.r_sqr);
      foreach (p_sequencer.pkt_sqr[i]) begin
        int j = i;
        fork
          begin
            reset_event.wait_on();
            `uvm_do_on(i_seq, p_sequencer.pkt_sqr[j]);
          end
        join_none
      end
      begin
        reset_event.wait_on();
        `uvm_do_on(h_seq, p_sequencer.h_sqr);
      end
    join
  endtask: body

  `ifdef UVM_VERSION_1_1
  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: 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
  `endif
endclass: top_reset_sequence
```

This page was intentionally left blank.