6

Functional Coverage, Using Packages, Standardizing Environments

Learning Objectives

After completing this lab, you should be able to:

- Implement Functional Coverage to determine when you are done with simulation
- Define Packages for reuse
- Import Packages for use in a test
- Understand the use of Environments and Standardized Test Methodologies

Lab Duration:
45 minutes

Getting Started

The question that Lab 5 did not answer is - how many packets should be sent through the router in order to test all combinations of input and output ports? With the current random stimulus based code, the answer cannot be determined. You will need to implement functional coverage.

In the first part of this lab, you will add the functional coverage components in the scoreboard class. Within the scoreboard class, you will implement functional coverage to measure the progress of your testbench and end the simulation when the testbench has fully exercised all input and output port combinations.

In Lab 5, you expanded your testbench to do broad-spectrum verification. You constructed all the components like drivers, receivers etc., and then "started" them to run the simulation. Then you coordinated the end-of-simulation. This methodology is very common to almost all testbenches that perform simulation-based verification. Can we standardize some of the components of the structure and tasks of running the simulation? This is what most standard methodologies like VMM, UVM etc. do.

In the second part of this lab you will use an "Environment" that encapsulates the components and "runs" them in a standardized manner. You will also define a package that allows reuse of standard components and class libraries.

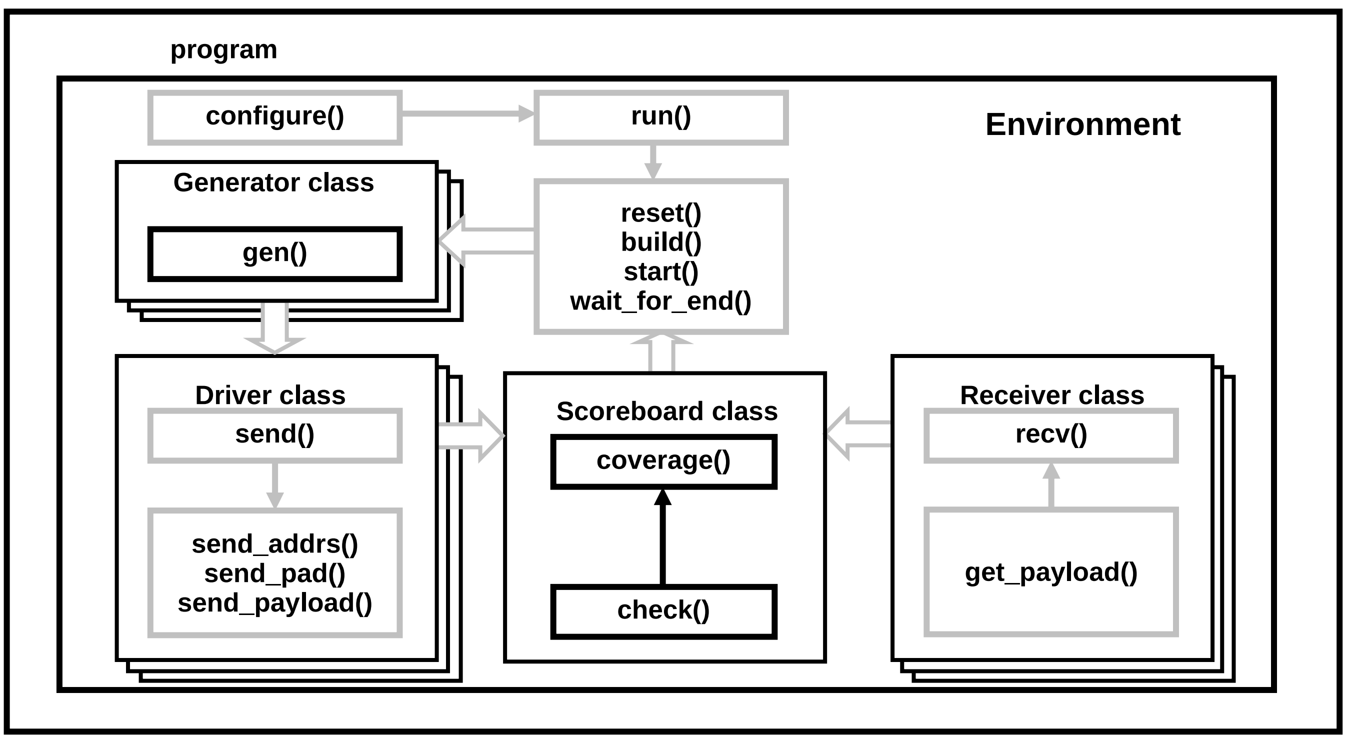

Figure 1. Lab 6 Standardized Environments and Functional Coverage

Lab Overview

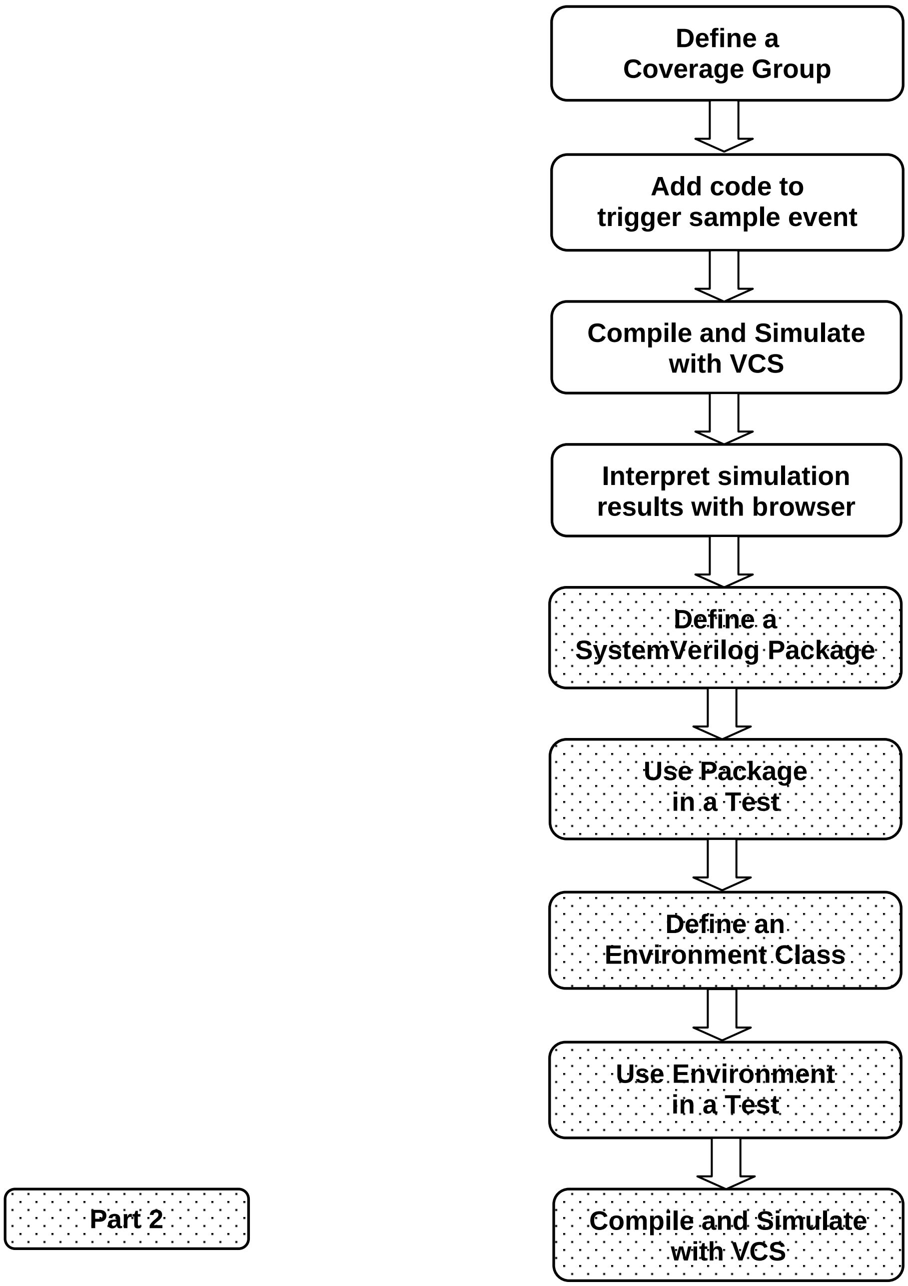

Figure 2. Diagram of Lab 6 Exercise

Note: You will find Answers for all questions and solutions in the Answers / Solutions at the end of this lab.

Part 1: Functional Coverage

Task 1. Copy Files from Lab 5's Solutions directory

1. Go into the lab6 directory.

> **cd ../lab6a**

2. Copy the source files in the **solutions/lab5** directory into the current directory with **make** script.

> **make copy**

(If you chose to use your own lab files from Lab5, type "**make mycopy**".)

Task 2. Create a `covergroup` in `Scoreboard` Class

First thing in implementing functional coverage in SystemVerilog is to define the coverage groups. Within the coverage groups, we can define coverage bins, bin update event timing and coverage goal.

Coverage bins should be created for each input port and output port. Then, cross coverage bins for all combinations of the input and output ports should be created.

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

2. Add two new class properties:

- **bit[3:0] sa, da;** // functional coverage properties

3. Declare a definition for a cover group (**router_cov**) immediately after the property declarations.

4. Inside the cover group

- Create coverpoint groups based on **sa** and **da**.
- Create cross bins based on the two sample groups (the cross coverage is the real coverage information we are looking for).

Task 3. Modify `new()` To Construct Coverage Object

1. In the constructor **`new()`**, construct **`router_cov.`** This is mandatory for covergroups defined inside classes. They must be constructed in the class constructor.

   When done, the **`covergroup`** definition should look like the following:

```
class Scoreboard;
  ...
  bit[3:0] sa, da; // functional coverage properties
  covergroup router_cov;
    coverpoint sa;
    coverpoint da;
    cross sa, da;
  endgroup
  ...
endclass

function Scoreboard::new(...);
  ...;
  router_cov = new();
endfunction
```

Task 4. Modify `check()` For Coverage

Modify the following in the **`check()`** function

1. Add a new **`real`** variable **`coverage_result`** (The data type must be **`real`** because the functional coverage results are returned as **`real`** values.)

   You will be storing the running functional coverage % in this variable.
2. Immediately after the successful comparison of **`pkt2send`** and **`pkt2cmp`**, set the class variables **`sa`** and **`da`** to values found in the **`pkt2send`** object.
3. Then, update the functional coverage bin with a call to **`router_cov.sample()`**.
4. Make a call to **`$get_coverage()`** to retrieve the updated functional coverage value and store this value in **`coverage_result.`**
5. Modify the **`$display()`** statement that follows to also print coverage %.
6. Modify the **`if`** statement that follows to also trigger the **`DONE`** event flag if coverage reaches 100%.
7. Save and close the file.

Task 5. Compile and Run

1. Use `make` script to compile and run your program.

   > **make**

   Make sure you reach 100% coverage. Increase the number of packets if necessary. Debug any error you find.

2. Run the simulation with different seeds (other than seed value of 0 or 1).

   > **make seed=<*seed_value*>**

   Note that you reach 100% coverage with a different number of packets from the previous run.

3. If there are no errors, open the functional coverage html or text file and verify that each port was driven and sampled.

   If the coverage report shows all combinations of input and output ports have been driven, you are done with the lab.

   To view the HTML report run

   > **firefox urgReport/dashboard.html &**

   Select the groups link and follow the links for the**router_test_top.t::Scoreboard::router_cov** covergroup.

   To view the report in text format run

   > **<editor> urgReport/grpinfo.txt**

   where **<editor>** is any text editor of your choice.

Part 2: Packages and Environments

Task 6. Copy Files from Labs 5/6a's Solutions directory

1. Go into the lab6b directory.

   > **cd ../lab6b**

2. Copy the source files in the **solutions/lab5** and **lab6a** directories into the current directory with **make** script.

   > **make copy**

   (If you choose to use your own lab files from Labs 5 and 6a, type "**make mycopy**".)

Task 7. Create a Package for Reuse

Since many components and other classes of the testbench are reused in many tests, you can put them in a library called a **package** in SystemVerilog. Let us see how you can refine the Lab 5 test to create a reusable environment using packages.

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

Note the sections of the file. First there are some global variables – **run_for_n_packets** and **TRACE_ON**. Then there are include directives for the component files. Since these variables and components are likely to be used in different tests, for example running the same simulation with different seeds, you can put them in a reusable package.

```
program automatic test(router_io.TB rtr_io);
  int run_for_n_packets; // number of packets to test
  int TRACE_ON = 0;         // subroutine tracing control

  `include "router_test.h"
  `include "Packet.sv"
  `include "Driver.sv"
  `include "Receiver.sv"
  `include "Generator.sv"
  `include "Scoreboard.sv"

  semaphore  sem[];      // prevent output port collision
  Driver     drvr[];     // driver objects
  Receiver   rcvr[];     // receiver objects
  Generator  gen;    // generator object
  Scoreboard sb;     // scoreboard object
  //continued...
```

Global Variables and common components – Create Reusable package

```
//test.sv continued...
initial begin
    run_for_n_packets = 2000;
    sem = new[16];
    drvr = new[16];
    rcvr = new[16];
    gen = new();
    sb = new();
    foreach (sem[i])
      sem[i] = new(1);
    for (int i=0; i<drvr.size(); i++)
      drvr[i] = new($psprintf("drvr[%0d]", i), i, sem,
                       gen.out_box[i], sb.driver_mbox, rtr_io);
    for (int i=0; i<rcvr.size(); i++)
      rcvr[i] = new($psprintf("rcvr[%0d]", i), i,
                       sb.receiver_mbox, rtr_io);

    reset();

    gen.start();
    sb.start();
    foreach(drvr[i])
      drvr[i].start();
    foreach(rcvr[i])
      rcvr[i].start();

    wait(sb.DONE.triggered);

  end

  task reset();
    if (TRACE_ON) $display("[TRACE]%t %m", $realtime);
    rtr_io.reset_n = 1'b0;
    rtr_io.cb.frame_n <= '1;
    rtr_io.cb.valid_n <= '1;
    repeat(2) @rtr_io.cb;
    rtr_io.cb.reset_n <= 1'b1;
    repeat(15) @(rtr_io.cb);
  endtask
endprogram
```

Build the Environment (construct Testbench components)

Reset DUT

Start the TB components running concurrently

Wait for end-of-test

2. Close the file.

3. Edit the file **router_test_pkg.sv** in an editor.

4. Declare a new package **router_test_pkg**.

- **package router_test_pkg;**

5. Add an int property **`run_for_n_packets`** and initialize it to 0.
   - **`int run_for_n_packets = 0;`**
6. Add the **`TRACE_ON`** variable and initialize it to 0. Packages are top-level name spaces in SystemVerilog and hence global variables should go in packages.
   - **`int TRACE_ON = 0;`**
7. Include the following files to create the package.
   - **`` `include "router_test.h" ``**
   - **`` `include "Packet.sv" ``**
   - **`` `include "Driver.sv" ``**
   - **`` `include "Receiver.sv" ``**
   - **`` `include "Generator.sv" ``**
   - **`` `include "Scoreboard.sv" ``**
   - **`` `include "Environment.sv" ``**

   We will use the Environment class after Task 9.
8. Complete the package definition.
9. Save and close the file.

Task 8. Use a Package in the Test Program

1. Open the file **`test.sv`** in an editor.
2. Delete the variable definitions and the include directives. Do not delete the component declarations and the initial block.
3. Import the package created in Task 2. Refer to your slides for the syntax. By importing the package everything in the package is visible in the program scope.
4. Save and close the file.

Task 9. Compile and Run

1. Use **`make`** script to compile and run your program.

```
> make
```

This simulation run is similar to the one you ran in Task 5. You should see the simulation stop after reaching 100% coverage or after 2000 packets have been transmitted. You may need to increase the number of packets transmitted to reach 100% coverage.

Task 10. Create a Test Environment for Reuse

The test environment is made up of components and their interconnections. The idea of an environment is to provide a test infrastructure with "knobs" that can be changed to create individual tests. These knobs can consist of extended classes to change behavior of existing classes, configuration variables that can be modified, randomized, etc. This also takes advantage of SystemVerilog's built-in randomization capability. The idea is to be able to set and control these knobs without having to directly modify the code in the environment and it's components. By creating random configurations you can test many more configurations and test conditions than you can with directed tests. The components like drivers, monitors etc. can also be constructed, started and stopped in a standardized manner.

1. Open the **`Environment.sv`** file in an editor. The complete environment has been coded for you.
2. Note the random variable **`run_for_n_packets`**. By putting random variables in the environment, or in a configuration object inside the environment, you can randomize the test configuration. Note the **`constraint`** block definition.
3. Note the methods called **`build()`**, **`reset()`**, **`configure()`**, **`start()`**, **`wait_for_end()`** . These correspond to the usual steps of most simulations for functional verification.
4. The **`configure()`** method is used to configure the test environment. Here it simply randomizes the **`run_for_n_packets`** variable. In a more complex verification environment, you would have many "knobs" in your test to configure. For example, knobs that control how many drivers or monitors to run the test with, or, what types of packets to drive. All these could be randomized during the test configuration step.
5. The **`run()`** method calls **`build()`**, then **`reset()`**, followed by **`start()`** and **`wait_for_end()`**. In the **`build()`** you construct the components needed by the test. The **`reset()`** is used to reset the DUT. The **`start()`** method will start all the components by calling the **`start()`** methods of each component constructed in the **`build()`** method. The **`wait_for_end()`** method waits for end-of-test conditions. You may think of these methods that **`run()`** calls as implementing the phases of the test run.

```
class Environment;
  string name;
  rand int run_for_n_packets;// number of packets to test
  virtual router_io.TB rtr_io;

  semaphore  sem[];    // prevent output port collision
  Driver     drvr[];   // driver objects
  Receiver   rcvr[];   // receiver objects
  Generator  gen[];    // generator objects
  Scoreboard sb;    // scoreboard object

  constraint valid {
   this.run_for_n_packets inside { [1500:2500] };
  }

  extern function new(string name = "Env",
                      virtual router_io.TB rtr_io);
  extern virtual task run();
  extern virtual function void configure();
  extern virtual function void build();
  extern virtual task start();
  extern virtual task wait_for_end();
  extern virtual task reset();

endclass: Environment

function Environment::new(string name = "Env", virtual
router_io.TB rtr_io);
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, name);
  this.name = name;
  this.rtr_io = rtr_io;
endfunction: new

task Environment::run();
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime,
this.name);
  this.build();
  this.reset();
  this.start();
  this.wait_for_end();
endtask: run

function void Environment::configure();
   if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, this.name);
this.randomize();
endfunction: configure
```

"Run" the Environment

```
//Class Environment continued...
function void Environment::build();
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, this.name);
  if(this.run_for_n_packets == 0) this.run_for_n_packets = 2000;
    this.sem = new[16];
    this.drvr = new[16];
    this.rcvr = new[16];
    this.gen = new[16];
    this.sb = new();
    foreach (this.sem[i])
      this.sem[i] = new(1);
    foreach (this.gen[i])
      this.gen[i] = new($sformatf("gen[%0d]", i));
    foreach (this.drvr[i])
      this.drvr[i] = new($psprintf("drvr[%0d]", i), i, this.sem,
          this.gen.out_box[i], this.sb.driver_mbox, this.rtr_io);
    foreach (this.rcvr[i])
      this.rcvr[i] = new($psprintf("rcvr[%0d]", i), i,
          this.sb.receiver_mbox, this.rtr_io);
endfunction: build
```

Build the Environment (construct components)

```
task Environment::reset();
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, this.name);
  this.rtr_io.reset_n <= 1'b0;
  this.rtr_io.cb.frame_n <= '1;
  this.rtr_io.cb.valid_n <= '1;
  repeat(2) @rtr_io.cb;
  this.rtr_io.cb.reset_n <= 1'b1;
  repeat(15) @(this.rtr_io.cb);
endtask: reset
```

Reset DUT

```
task Environment::start();
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, this.name);
    this.sb.start();
    foreach(this.gen[i])
      this.gen[i].start();
    foreach(this.drvr[i])
      this.drvr[i].start();
    foreach(this.rcvr[i])
      this.rcvr[i].start();
endtask: start
```

Start the components

```
task Environment::wait_for_end();
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, this.name);
    wait(this.sb.DONE.triggered);
endtask: wait_for_end
```

Wait for end-of-test

Task 11. Use Environment in the Test Program

1. Open the file **`test.sv`** in an editor.

Note that in the **`initial`** block you perform the following steps to run your test.

- The variables are set (configured).
- The components constructed (built).
- The DUT is reset.
- The components are started. This starts the simulation traffic.
- Finally the test waits for the end-of-test condition – the DONE event.

These are common test tasks or **phases** that you will now encapsulate into an Environment class.

2. Delete the handle declarations for the components.
3. Immediately after the import statement declare a handle to the Environment class. Call it **`env`**.
4. Delete all code in the **`initial`** block.
5. In the **`initial`** block construct the Environment object. Remember to pass the correct arguments to the constructor.
6. Call the **`configure()`** method of the Environment. This is one way to control the knobs of the test. In our test this randomizes the Environment's **`run_for_n_packets`** variable.
7. Update the **`run_for_n_packets`** (defined in the package, available in the program) to environment's **`run_for_n_packets`**.
8. "Run" the test by calling the **`run()`** method of the Environment. The Environment will now handle all the phases of running the simulation that you performed in the **`program`** in Lab 5.
9. Delete the **`reset()`** task.

10. When done, the test program should look like the following:

```
program automatic test(router_io.TB rtr_io);
import router_test_pkg::*;
Environment env;
initial begin
    env = new("env", rtr_io);
    env.configure();
    run_for_n_packets = env.run_for_n_packets;
    env.run();
end
endprogram: test
```

Task 12. Compile and Run

1. Use `make` script to compile and run your program.

> **make**

Question 1. How is this run different from the run in Task 8? Why?

....

....

....

2. Rerun simulation with a different seed.

> **make run seed=<*seed_value*>**

Question 2. How did the seed affect the results? Why?

....

....

Congratulations, you completed Lab 6!

Answers / Solutions

Task 12. Compile and Run

Question 1. How is this run different from the run in Task 8? Why?

- The number of packets transmitted in the two tests was not the same. The Environment configuration that was performed when you called **`env.configure()`**, randomized the **`run_for_n_packets`** variable. This was used by the test. The constraint for this random variable is defined in the **`Environment`** class.
- The number of packets required to reach 100% coverage also changed. Since the structure of the two test environments is different, the randomization of packets was not the same between the two tests. So the number of packets needed to reach 100% coverage was also different. This is a good example of how changing the test environment affects the constraint solver. If you change the environment after fixing a bug in the DUT you may not be able to reproduce the bug.

Question 2. How did the seed affect the results? Why?

- The number of packets required to reach 100% coverage changed. This is a good example of how changing the simulation random seed affects the constraint solver and thus affects the time to coverage closure.

Scoreboard.sv Solution Lab6a:

```
`ifndef INC_SCOREBOARD_SV
`define INC_SCOREBOARD_SV
class Scoreboard;
  string   name;      // unique identifier
  event    DONE;      // flag to indicate goal reached
  Packet   refPkt[$];     // reference Packet array
  Packet   pkt2send;      // Packet object from Drivers
  Packet   pkt2cmp;       // Packet object from Receivers
  pkt_mbox  driver_mbox; // mailbox for Packet objects from Drivers
  pkt_mbox  receiver_mbox;// mailbox for Packet objects from Receivers
  bit[3:0] sa, da;       // functional coverage properties
  int run_for_n_packets; //how many packets

  covergroup router_cov;
    coverpoint sa ;
    coverpoint da ;
    cross sa, da;
  endgroup

  extern function new(string name = "Scoreboard",
                   pkt_mbox driver_mbox =null, receiver_mbox = null);
  extern virtual task start();
  extern virtual function void check();
endclass

function Scoreboard::new(string name, pkt_mbox driver_mbox,
receiver_mbox);
  if (TRACE_ON) $display("[TRACE]%0t %s:%m", $time, name);
  this.name = name;
  if (driver_mbox == null) driver_mbox = new();
  this.driver_mbox = driver_mbox;
  if (receiver_mbox == null) receiver_mbox = new();
  this.receiver_mbox = receiver_mbox;
  router_cov = new();
endfunction
```

Continued…

```
task Scoreboard::start();
  if (TRACE_ON) $display("[TRACE]%0t %s:%m", $time, name);
  fork
    forever begin
      receiver_mbox.get(pkt2cmp);
      while (driver_mbox.num()) begin
        Packet pkt;
        driver_mbox.get(pkt);
        refPkt.push_back(pkt);
      end
      check();
    end
  join_none
endtask

function void Scoreboard::check();
  int    index[$];
  string message;
  static int  pkts_checked = 0;
  real   coverage_result;

  if (TRACE_ON) $display("[TRACE]%0t %s:%m", $time, name);
  index = refPkt.find_first_index() with (item.da == pkt2cmp.da);
  if (index.size() <= 0) begin
    $display("\n%m\n[ERROR]%0t %s not found in Reference Queue\n",
$time, pkt2cmp.name);
    pkt2cmp.display("ERROR");
    $finish;
  end
  pkt2send = refPkt[index[0]];
  refPkt.delete(index[0]);
  if (!pkt2send.compare(pkt2cmp, message)) begin
    $display("\n%m\n[ERROR]%0t Packet #%0d %s\n", $time, pkts_checked,
message);
    pkt2send.display("ERROR");
    pkt2cmp.display("ERROR");
    $finish;
  end
  this.sa = pkt2send.sa;
  this.da = pkt2send.da;
  router_cov.sample();
  coverage_result = $get_coverage();
  $display("[NOTE]%0t Packet #%0d %s coverage = %3.2f", $time,
pkts_checked++, message, coverage_result);
  if ((pkts_checked >= run_for_n_packets) || (coverage_result == 100))
    ->DONE;
endfunction: check
`endif
```

`router_test_pkg.sv` Solution Lab 6b:

```
package router_test_pkg;

int run_for_n_packets = 0;
int TRACE_ON = 0;

`include "router_test.h"
`include "Packet.sv"
`include "Driver.sv"
`include "Receiver.sv"
`include "Generator.sv"
`include "Scoreboard.sv"
`include "Environment.sv"

endpackage: router_test_pkg
```

`test.sv` Solution Lab 6b:

```
program automatic test(router_io.TB rtr_io);
import router_test_pkg::*;

Environment env;

initial begin
    env = new("env", rtr_io);
    env.configure();
    run_for_n_packets = env.run_for_n_packets;
    env.run();
end

endprogram: test
```