5 Broad Spectrum Verification

Learning Objectives

After completing this lab, you should be able to:

- Build a Generator transactor class
- Build a Driver class
- Build a Receiver class
- Build a Scoreboard class
- Expand the testbench to drive and monitor all input and output ports concurrently

Lab Duration: 90 minutes

Getting Started

In lab 4, you created an encapsulated packet. But, because you had only one driver and monitor, you were only able to drive a single input and output port at a time.

In this lab, you will encapsulate the generator, driver, monitor and check routines into **`Generator`** class, **`Driver`** class, **`Receiver`** class and **`Scoreboard`** class respectively. You will then build a testbench architecture that is capable of exercising all ports simultaneously.

To facilitate passing of **`Packet`** object from transactor to transactor, you will use the SystemVerilog built-in **`mailbox`** class as the communication mechanism.

The resulting architecture is shown below:

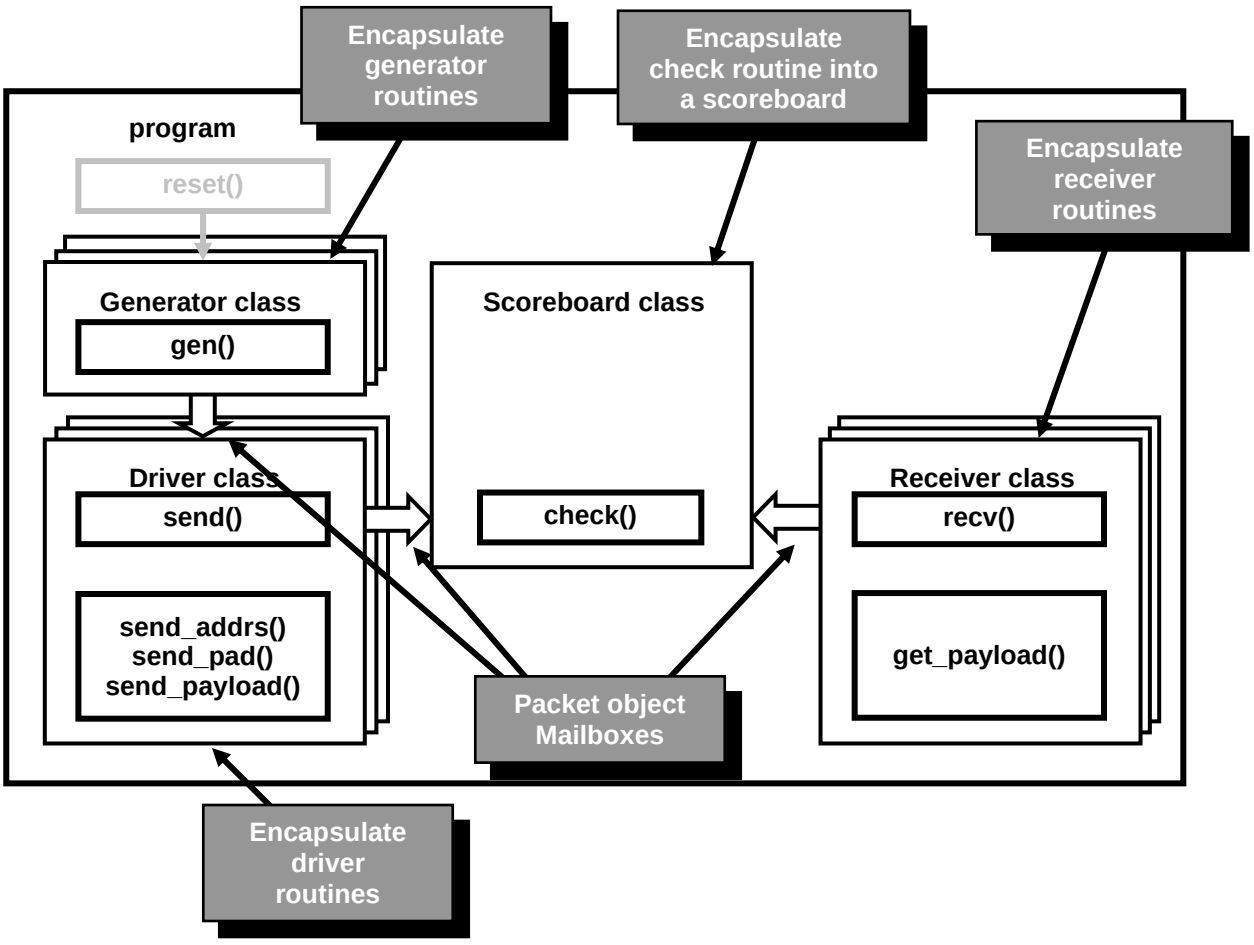


Figure 1. Lab 5 Encapsulate transactors for broad-spectrum verification

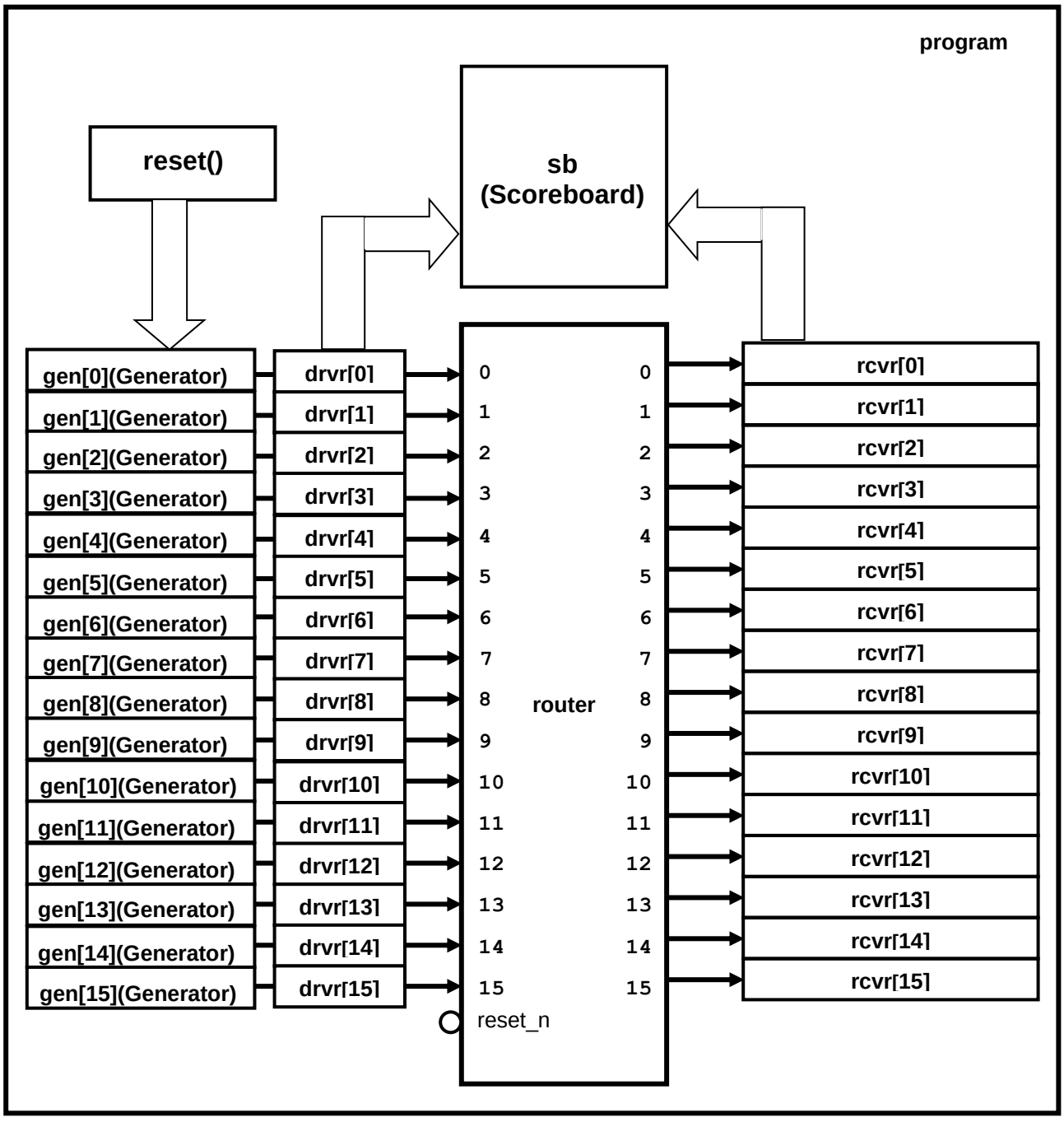


Figure 2. Lab 5 testbench architecture

Lab Overview

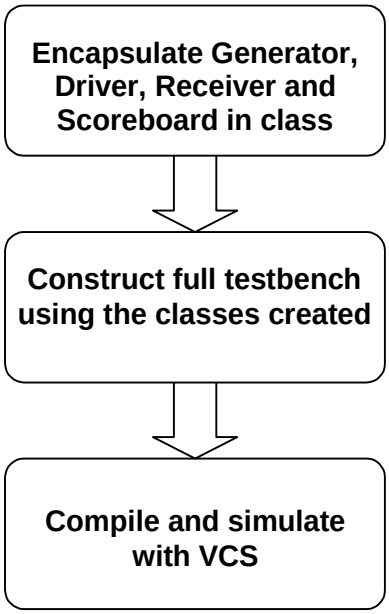


Figure 3. Diagram of Lab Exercise

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

Broad-Spectrum Verification

Task 1. Copy Files from Lab 4's solutions directory

1. Go into the lab5 directory.

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

2. Copy the source files in the **solutions/lab4** directory into the current directory with the **make** script.

```
> make copy
```

Task 2. Develop `Driver` class

A **DriverBase** class which encapsulates the driver routines and the program global variables used in previous labs is already developed for you. You will extend from this base class to implement a new **Driver** class.

1. Open the existing **DriverBase.sv** file in an editor

2. Examine the **DriverBase** class.

In this base class, the following properties are declared:

- `virtual router_io.TB rtr_io;` // interface signal
- `string name;` // unique identifier
- `bit[3:0] sa, da;` // source and destination addresses
- `logic[7:0] payload[$];` // Packet payload
- `Packet pkt2send;` // stimulus Packet object

The **rtr_io** property defines the interface signals for the Drivers to drive. The **name** property uniquely identifies the object. Both properties will be set by the constructor **new()**.

The **sa**, **da**, **payload** and **pkt2send** properties are the program global variables that you had used in previous labs. You will set these properties in the **Driver** class to be developed in the next few steps. Since these are class properties, all methods in the class can access these properties directly.

For each of the methods in the class, there is an **if-$display()** combination at the beginning of the subroutine. It is often helpful during debugging to be able to see a trace of the subroutine execution sequences. This **if-$display()** combination will let you control whether or not to print subroutine sequence tracing to the terminal. The variable **TRACE_ON** is a program global variable that you will declare and control later in the **test.sv** file.

3. Close the file.

4. Open the existing **Driver.sv** file in an editor.

   The **Driver** class is derived from the **DriverBase** class. Three properties and two method prototypes are declared for you.

   The **in_box** will be used to pass **Packet** objects from the Generator to the Driver. The **out_box** will be used to pass **Packet** objects from the Driver to the Scoreboard. The **in_box** and **out_box** are of type **pkt_mbox**. This is a typed **mailbox** defined in the file **router_test.h** as shown below.

```
typedef class Packet;
typedef mailbox #(Packet) pkt_mbox;
```

   The **sem[]** array will be used as an arbitration mechanism for preventing multiple input ports trying to drive the same output port at the same time.

   The constructor method has five variables in the argument list. The **name** argument allows user to assign a unique identifier. The **port_id** argument sets the Driver to drive a specific input port. The **sem[]** argument is a set of semaphore bins for the Driver to self-arbitrate access to output ports. The **in_box** argument is a mailbox, which connects the Driver to the Generator. The **out_box** argument is a mailbox which connects the Driver to the Scoreboard.

   The **start()** method retrieves **Packet** object from **in_box**. Within the method, drive the packet through the DUT with a call to **send()**, if the selected destination address port is not already in use by another driver.

```
`ifndef INC_DRIVER_SV
`define INC_DRIVER_SV
`include "DriverBase.sv"
class Driver extends DriverBase;
  pkt_mbox in_box;                  // Generator mailbox
  pkt_mbox out_box;                 // Scoreboard mailbox
  semaphore sem[];                  // output port arbitration
  extern function new(...);
  extern virtual task start();
endclass

function Driver::new(string name, int port_id, semaphore sem[],
pkt_mbox in_box, out_box, virtual router_io.TB rtr_io);
endfunction

task Driver::start();
endtask: start
`endif
```

Task 3. Fill in `Driver` class `new()` method

1. In the body of the externally declared constructor **`new()`**, call **`super.new()`** with the **`name`** and **`rtr_io`** arguments.
2. Add a tracing statement after the call to **`super.new()`** as follows:

```
if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, this.name);
```

3. Assign the class property **`sa`** (defined in base class) to the value passed in via **`port_id`**.
4. Complete the constructor development by assigning class properties **`sem[]`**, **`in_box`** and **`out_box`** with the values passed in via the argument list.

Task 4. Fill in `Driver` Class `start()` Method

Each transactor object you instantiate in the test program will need a mechanism to start operation. You will standardize the name of this method to be **`start()`**.

For the **`Driver`**, the **`start()`** method will execute an infinite loop. In each iteration of the loop, a **`Packet`** object will be retrieved from **`in_box`**. This **`Packet`** object content will then be driven through the DUT via a call to **`send()`**. Once the **`Packet`** object processing is completed, the **`Packet`** object is passed on to **`Scoreboard`** via the **`out_box`**.

Since the **`Driver`** object, when started, is expected to run concurrently with all other components of the testbench, all contents of the **`start()`** method with the exception of the trace statement must be inside a non-blocking **`fork-join`** construct.

1. In the existing **`start()`** method body, add a trace statement.
2. After the trace statement, create a **non-blocking `fork-join`** block.
3. Inside the **`fork-join`** construct, create a single infinite loop.
4. Each iteration through the loop do the following:
   - **a)** Retrieve a **`Packet`** object (**`pkt2send`**) from **`in_box`**.
   - **b)** If the **`sa`** property in the retrieved **`Packet`** object does not match **`this.sa`**, continue on to the next iteration of the loop.
   - **c)** If the retrieved **`Packet`** `sa` does match **`this.sa`**, update the **`da`** and **`payload`** class properties with the content of **`pkt2send`**.
   - **d)** Use the semaphore **`sem[]`** array to arbitrate for access to the output port specified by **`da`**.

e) Once the arbitration is successful, call **`send()`** to drive the packet through the DUT.

f) When **`send()`** completes, deposit **`Packet`** object into **`out_box.`**

g) Put the semaphore key back into its bin in the final step of the loop.

5. Save and close the file.

Task 5. Develop `Receiver` Class

1. Open the existing **`Receiver.sv`** skeleton file in an editor.

```
`ifndef INC_RECEIVER_SV
`define INC_RECEIVER_SV
`include "ReceiverBase.sv"
class Receiver extends ReceiverBase;
  pkt_mbox out_box;                    // Scoreboard mailbox
  extern function new(...);
  extern virtual task start();
endclass
function Receiver::new(string name, int port_id, pkt_mbox
out_box);
endfunction
task Receiver::start();
endtask: start
`endif
```

Task 6. Fill in `Receiver` Class `new()`

1. In the body of the externally declared constructor **`new()`**, call **`super.new()`** with the **`name`** and **`rtr_io`** argument.

2. Add a tracing statement after the call to **`super.new()`**.

3. Assign the class property **`da`** to the value passed in via **`port_id.`**

4. Assign class property **`out_box`** with values passed in via the argument list.

Task 7. Fill in `Receiver` Class `start()` Method

The **`start()`** method will execute a non-blocking infinite loop. In each iteration of the infinite loop, reconstuct a **`Packet`** object（**`pkt2cmp`**) monitored from DUT. Once, retrieved, this **`Packet`** object will be passed to Scoreboard via an out mailbox.

1. In the **`start()`** method body, add a trace statement.
2. After the trace statement, create a **non-blocking** concurrent process thread.
3. Inside the **`fork-join`** construct, create a single infinite loop code block.
4. Each iteration through the loop do the following:
   a) Call **`recv()`** to retrieve a **`Packet`** object from DUT
   b) Deposit a **copy** of the **`Packet`** object (**`pkt2cmp`**) retrieved from DUT into **`out_box`**. Use the **`copy()`** method of the **`Packet`** class.
5. Save and close the file.

Task 8. Examine the `Generator` class

In the interest of saving time during lab, a **`Generator.sv`** file is written for you. It encapsulates the **`gen()`** routine that you had completed earlier and includes a **`start()`** method similar to what you have done for the Drivers and Receivers.

There are two significant differences in the **`start()`** method you should know about. First, the **`start()`** method loop is controlled by the program global **`run_for_n_packets`** variable. If **`run_for_n_packets`** is <= 0, then the loop will be infinite. If it is > 0, then, the loop will stop after **`run_for_n_packets`** iterations. Second, after **`gen()`** method is called, a copy of the randomized **`Packet`** object (**`pkt2send`**) is created and sent to the **`Drivers`** via **`out_box`** mailboxes. Note that the **`gen()`** routine has a static variable **`pkts_generated`** to count the packets generated across all generators.

The Generator generates packets for the particular driver it is connected to, by constraining packets to source address values matching its **`port_id`** variable.

Examine the content of the **`Generator.sv`** file if you are interested.

Task 9. Examine The `Scoreboard` Class

A **`Scoreboard.sv`** file has also been written for you. It is mainly an encapsulation of the **`check()`** routine you have already writen.

The main new feature is the implementation of mailboxes to allow communication between the **`Scoreboard`** and the **`Drivers`** and **`Receivers`**.

A **`Driver`** will deposit the **`Packet`** objects it has just sent to the DUT into the **`driver_mbox`** mailbox. A **`Receiver`** will deposit the **`Packet`** object it has just retrieved from the DUT into the **`receiver_mbox`** mailbox.

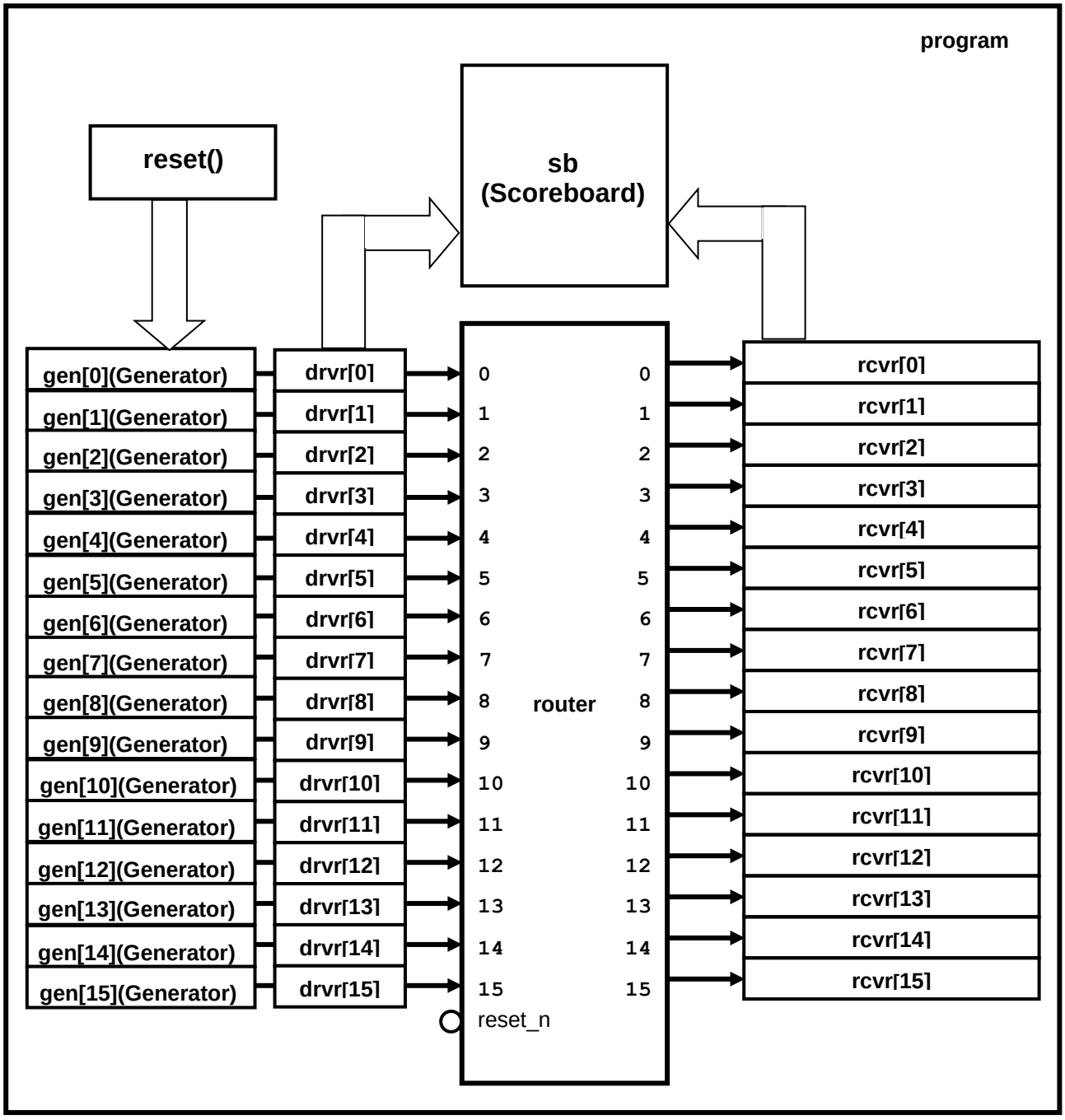
When the Scoreboard finds a **`Packet`** object in the **`receiver_mbox`**, it will first save this object handle as **`pkt2cmp`**. Then, it will push all **`Packet`** objects found in the **`driver_mbox`** onto a **`refPkt[$]`** queue. Afterwards, on the basis of the output port address (**`da`**) in **`pkt2cmp`** object, it will try to locate the corresponding reference **`Packet`** in **`refPkt[$]`** and compare the content. If no corresponding reference **`Packet`** is found, an error is reported.

When the number of **`Packet`** objects checked matches the global variable **`run_for_n_packets`**, an `event` flag called **`DONE`** is triggered. This **`DONE`** flag will allow the simulation to terminate gracefully at the appropriate time.

Examine the content of the **`Scoreboard.sv`** file if you are interested.

Task 10. Modify `test.sv` To Use These New Classes

You will now modify the testbench to have 16 generators, one scoreboard, 16 drivers and 16 receivers.



1. Open **`test.sv`** in an editor.
2. Delete all program global variables except **`run_for_n_packets`**.
3. Create an int variable **`TRACE_ON`** and intialize it to 0 (change to 1 if subroutine execution tracing is desired for debugging).

4. Following the two program global variables, add include statements for all header files and the new class files (**router_test.h**, **Driver.sv**, **Receiver.sv**, **Generator.sv**, **Scoreboard.sv**).

5. Following the include statements, add the following program global variables:

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

6. Delete all content of the initial block except for one:

```
initial begin
  run_for_n_packets = 2000;
end
```

Add the following:

7. Construct all objects declared in the program block. Make sure the mailboxes are connected correctly:
   Each Driver to corresponding Generator (**gen[i].out_box**)
   All Drivers to one Scoreboard mailbox (**sb.driver_mbox**)
   All Receivers to one Scoreboard mailbox (**sb.receiver_mbox**)

8. After all objects are constructed, call **reset()** to reset the DUT.

9. Then, start all transactors (Generators, Scoreboad, Drivers and Receivers).

10. Finally, before the end of the program, block until Scoreboard's **DONE** event flag is set.

Task 11. Misc checks

1. The following subroutines should be deleted from **test.sv**.

   - **gen()**
   - **send()**
   - **send_addr()**
   - **send_pad()**
   - **send_payload()**
   - **recv()**
   - **get_payload()**
   - **check()**

(Make sure **reset()** is NOT deleted)

When done, your **`test.sv`** file should look like:

```
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 objects
  Scoreboard sb;          // scoreboard object
  initial begin
    $vcdpluson;
    run_for_n_packets = 2000;
    sem = new[16];
    drvr = new[16];
    rcvr = new[16];
    gen = new[16];
    sb = new("sb");
    foreach (sem[i]) sem[i] = new(1);
    foreach (gen[i]) gen[i] = new($sformatf("gen[%0d]",
i),i);
    foreach (drvr[i])
      drvr[i] = new($sformatf("drvr[%0d]", i), i, sem,
gen[i].out_box, sb.driver_mbox, rtr_io);
    foreach (rcvr[i])
      rcvr[i] = new($sformatf("rcvr[%0d]", i), i,
sb.receiver_mbox, rtr_io);
    reset();
    sb.start();
    foreach(gen[i]) gen[i].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: reset
endprogram: test
```

2. Save and close the file.

Task 12. Compile and Run

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

   > **make**

   Debug any error you find.

2. If the testbench runs successfully again, execute the following script which runs the testbench on a bad RTL code.

   > **make bad**

   If the simulation finds an error, you are done.

   Congratulations, you completed Lab 5!

Answers / Solutions

`test.sv` Solution:

```
program automatic test(router_io.TB rtr_io);
  // following program variables will be seen by the included files without extern
  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"

  // The following program variables can be seen by the included files withextern
  semaphore  sem[];        // prevent output port collision
  Driver     drvr[];       // driver objects
  Receiver   rcvr[];       // receiver objects
  Generator  gen[];        // generator object
  Scoreboard sb;           // scoreboard object

  initial begin
    $vcdpluson;
    run_for_n_packets = 2000;
    sem = new[16];
    drvr = new[16];
    rcvr = new[16];
    gen = new[16];
    sb = new("sb");
    foreach (sem[i]) sem[i] = new(1);
    foreach (gen[i]) gen[i] = new($sformatf("gen[%0d]", i, i);
    foreach (drvr[i])
      drvr[i] = new($sformatf("drvr[%0d]", i), i, sem, gen[i].out_box, sb.driver_mbox,
rtr_io);
    foreach (rcvr[i])
      rcvr[i] = new($sformatf("rcvr[%0d]", i), i, sb.receiver_mbox, rtr_io);
    reset();
    sb.start();
    foreach(gen[i]) gen[i].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: reset
endprogram: test
```

Driver.sv Solution:

```
`ifndef INC_DRIVER_SV
`define INC_DRIVER_SV
`include "DriverBase.sv"
class Driver extends DriverBase;
  pkt_mbox in_box;   // Generator mailbox
  pkt_mbox out_box;  // Scoreboard mailbox
  semaphore sem[];   // output port arbitration

  extern function new(string name = "Driver", int port_id, semaphore
sem[], pkt_mbox in_box, out_box, virtual router_io.TB rtr_io);
  extern virtual task start();
endclass: Driver

function Driver::new(string name, int port_id, semaphore sem[],
pkt_mbox in_box, out_box, virtual router_io.TB rtr_io);
  super.new(name, rtr_io);
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, this.name);
  this.sa = port_id;
  this.sem = sem;
  this.in_box = in_box;
  this.out_box = out_box;
endfunction: new

task Driver::start();
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, this.name);
  fork
    forever begin
      this.in_box.get(this.pkt2send);
      if (this.pkt2send.sa != this.sa) continue;
      this.da = this.pkt2send.da;
      this.payload = this.pkt2send.payload;
      this.sem[this.da].get(1);
      this.send();
      this.out_box.put(this.pkt2send);
      this.sem[this.da].put(1);
    end
  join_none
endtask: start
`endif
```

Receiver.sv Solution:

```
`ifndef INC_RECEIVER_SV
`define INC_RECEIVER_SV
`include "ReceiverBase.sv"
class Receiver extends ReceiverBase;
  pkt_mbox out_box;  // Scoreboard mailbox

  extern function new(string name = "Receiver", int port_id,
pkt_mbox
        out_box, virtual router_io.TB rtr_io);
  extern virtual task start();
endclass: Receiver

function Receiver::new(string name, int port_id, pkt_mbox out_box,
virtual router_io.TB rtr_io);
  super.new(name, rtr_io);
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, this.name);
  this.da = port_id;
  this.out_box = out_box;
endfunction: new

task Receiver::start();
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, name);
  fork
    forever begin
      this.recv();
      begin
        Packet pkt = this.pkt2cmp.copy();
        this.out_box.put(pkt);
      end
    end
  join_none
endtask: start
`endif
```

Generator.sv Solution:

```
`ifndef INC_GENERATOR_SV
`define INC_GENERATOR_SV
class Generator;
  string  name;      // unique identifier
  Packet  pkt2send;  // stimulus Packet object
  pkt_mbox out_box;  // mailbox to Drivers
  int port_id = -1; // port_id of connected Driver
  static int pkts_generated = 0; //packet count across all generators

  extern function new(string name = "Generator", int port_id);
  extern virtual task gen();
  extern virtual task start();
endclass: Generator

function Generator::new(string name, int port_id);
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, name);
  this.name = name;
  this.pkt2send = new();
  this.out_box = new(1); //1-deep mailbox
  this.port_id = port_id;
endfunction: new

task Generator::gen();
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, this.name);
  this.pkt2send.name = $sformatf("Packet[%0d]",
this.pkts_generated++);
  if (!this.pkt2send.randomize()
       with {if (port_id != -1) sa == port_id;})
  begin
    $display("\n%m\n[ERROR]%t Randomization Failed!\n", $realtime);
    $finish;
  end
endtask: gen

task Generator::start();
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, this.name);
  fork
    while (this.pkts_generated <run_for_n_packets ||
run_for_n_packets <= 0) begin
      this.gen();
      begin
        Packet pkt = this.pkt2send.copy();
        this.out_box.put(pkt);
      end
    end
  join_none
endtask: start
`endif
```

Scoreboard.sv Solution:

```
`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

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

function Scoreboard::new(string name, pkt_mbox driver_mbox,
receiver_mbox);
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, 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;
endfunction: new

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

Continued…

```
…Continued from previous page
function void Scoreboard::check();
  int    index[$];
  string message;
  static int  pkts_checked = 0;
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, this.name);
  index = this.refPkt.find_first_index() with (item.da ==
                                                this.pkt2cmp.da);
  if (index.size() <= 0) begin
$display("\n%m\n[ERROR]%t %s not found in Reference Queue\n",
         $realtime, this.pkt2cmp.name);
    this.pkt2cmp.display("ERROR");
    $finish;
  end
  this.pkt2send = this.refPkt[index[0]];
  this.refPkt.delete(index[0]);
  if (!this.pkt2send.compare(this.pkt2cmp, message)) begin
    $display("\n%m\n[ERROR]%t Packet #%0d %s\n",
         $realtime, pkts_checked, message);
    this.pkt2send.display("ERROR");
    this.pkt2cmp.display("ERROR");
    $finish;
  end
  $display("[NOTE]%t Packet #%0d %s", $realtime, pkts_checked++,
message);
  if (pkts_checked >= run_for_n_packets)
    ->this.DONE;
endfunction: check
`endif
```