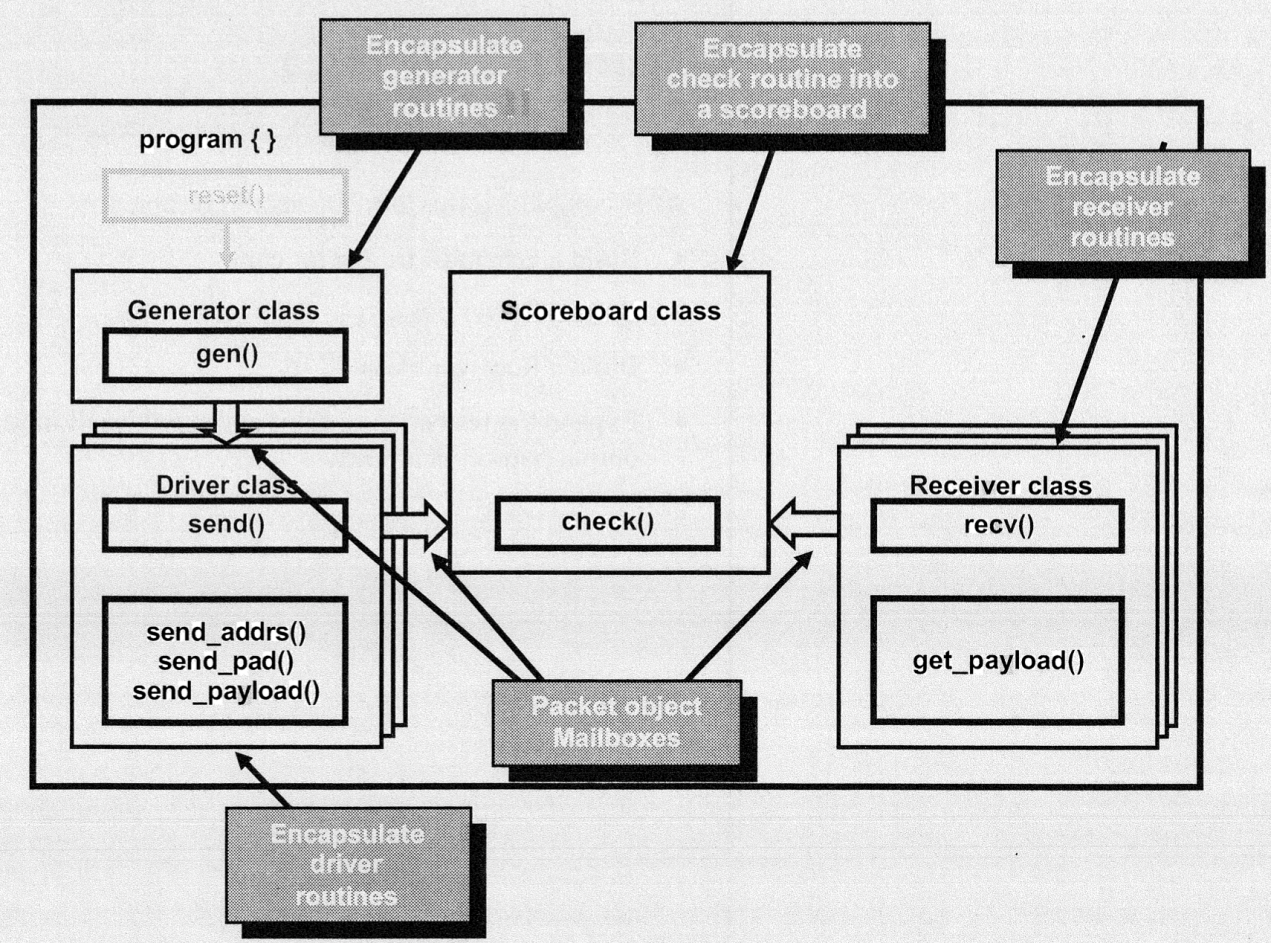
Extra-lab: Verification Components

The architecture is shown below:



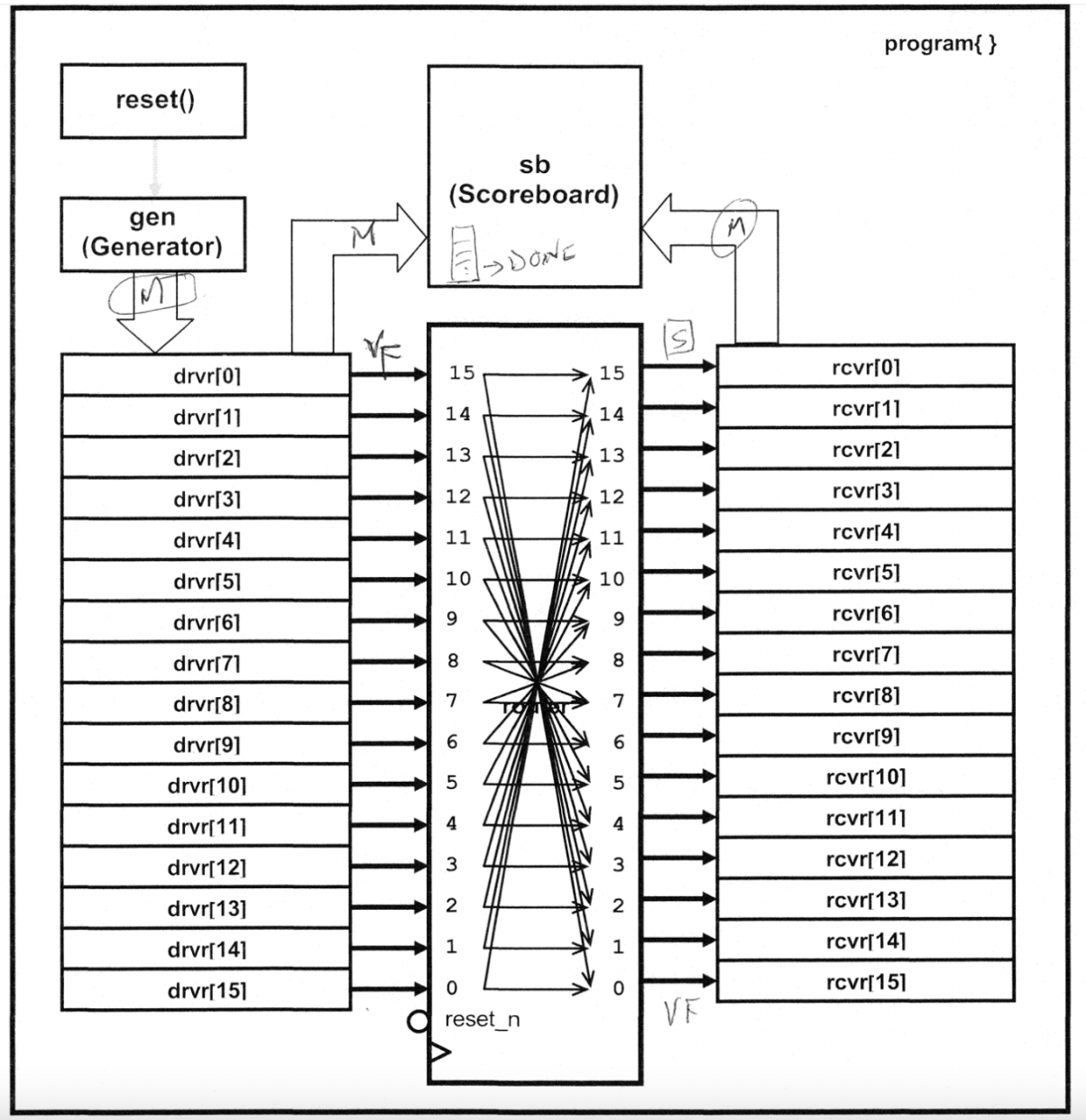


Figure 1. Testbench architecture

Task 1. Create a Packet Class File

Create a Packet class to encapsulate the packet information.

1. Open the Packet.sv file in an editor.
2. Declare a class definition for Packet as follows:

```
class Packet;

endclass: Packet
```

3. Use macros as a guard against multiple compilation before and after the class statements.

```
`ifndef INC_PACKET_SV
`define INC_PACKET_SV
class Packet;

endclass: Packet
`endif
```

4. *In* the body *of the Packet class, create the following properties:*

- **`rand bit[3:0] sa, da;`** // random port selection
- **`rand logic[7:0] payload[$];`** // random payload array
- **`string name;`** // (see description below)

Individual test **`Packet`** objects will be created by the **`gen`** () routine. For these **`Packet`** objects, you will want to tag each one uniquely to identify the object.

The string property **`name`** is the unique identifier. Displaying the **`name`** property as part of the printed message will be very helpful during the debugging process.

Define Packet Property Constraints

Immediately after the property declaraions add a constraints block to limit **`sa`** and **`da`** to**`0:15,`** and the **`payloadsize`**() to**`2:4.`**

Define Packet Class Method Prototypes

Add the following method prototype declarations in the body of Packet class:

```
class Packet;
  ...
  extern function new(string name = "Packet");
  extern function bit compare(Packet pkt2cmp, ref string message);
  extern function void display(string prefix = "NOTE");
endclass: Packet
```

Define Packet Class `new ()` Constructor

The class constructor new () is used to initialize object properties. For Packet objects, most properties will be set with a call to randomize () . The one property that needs to be initialized in the constructor is **`name.`**

1. Outside o f the class body, create the constructor new () method. Make sure to reference the method back to the class via the P a c k e t: : notation.
2. Inside the constructor body, assign the class property `name` with the string passed in via the argument:

```
function Packet::new(string name);
   this.name = name;
endfunction: new
```

3. This mechanism makes the string passed in through the constructor argument accessible to all other methods of the Packet class.

Define Packet compare () Method

For self checking, comparing contents of two data objects is a common need. It is a good idea to build the compare method within the data object.

1. Cut and paste **`compare()`** from test.sv into the Packet class at the end of the file.

Note: An alternative to cut and paste is to concatenate (cat) content o f **test. sv** into `Packet.sv` then keep only `compare()` in`Packet.sv.`

2. Reference the method to `Packet` class with : : notation.
3. Modify the argument list to contain a Packet handle:

```
function bit Packet::compare(Packet pkt2cmp, ref string message)
```

4. Inside the `compare()` method, change `pkt2cmp_payload` to reference the class property `pkt2cmp.payload.`

Define Packet `display ()` Method

It is helpful during the debugging process to print out the contents of a packet. To ease this effort, a display method should be defined for the Packet class to print the content of the Packet object to the console.

Outside of the class body, create the **`display`**() method

function void Packet::display(string prefix = "NOTE");

Inside the method body, print a formated content of the object to terminal.

In the printed message, you should also include the string passed via the argument list. This string can be set by user to differentiate between different types of message: ERROR, W ARNING, DEBUG, etc.

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 `DriverBaser.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 of these 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 drive methods, there is an if-$display() combination at the beginning o f the subroutine. It is often helpful during debugging to be able to see a printout of the subroutine execution sequences. This if-$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 Generator to Driver. The `out_box` will be used to pass Packet objects from Driver to Scoreboard. The `in_box` and `out_box` are of type `pkt_mbox.` This is a typed mailbox defined in the file router_test.il shown below.

```
typedcef 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.` W ithin 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()`
2. with the `name` and `rtr_io` arguments.
   Add a tracing statement after the `super.new()` call as follows:

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

Task 5. Develop `Receiver` Class

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 `super.new ()` call
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

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.`

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.

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[$] queue. Afterwards, on the basis ofthe output port address (da) in pkt2cmp object, it will try to locate the corresponding reference Packet in refPkt [$] 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 o f the Scoreboard.sv file if you are interested.

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

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 [];`
- `Driverdrvr[];`
  `Receiverrcvr[];`
- `Generator gen;`
- `Scoreboard sb;`

6. Construct all objects declared in the program block. Make sure the mailboxes are connected correctly: Generator to all Drivers (gen.out_box[i]); all Drivers to one Scoreboard mailbox (sb.driver_m box); all Receivers to one Scoreboard mailbox (sb.receiver_mbox)

7. After all objects are constructed, call `reset ()` to reset the DUT Then, start all transactors (gen, sb, drivers and receivers)
8. Finally, before the end of the program, block until sb's DONE event flag is set

Task 11. Misc checks

The following subroutines should be deleted from t e s t . sv.

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

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 object
  Scoreboard sb;          // scoreboard object
  initial begin
    $vcdpluson;
    run_for_n_packets = 2000;
    sem = new[16];
    drvr = new[16];
    rcvr = new[16];
    gen = new();
    sb = new();
    foreach (sem[i])
      sem[i] = new(1);
    foreach (drvr[i])
      drvr[i] = new($sformatf("drvr[%0d]", i), i, sem,
gen.out_box[i], sb.driver_mbox, rtr_io);
    foreach (rcvr[i])
      rcvr[i] = new($sformatf("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;
    ##2 rtr_io.cb.reset_n <= 1'b1;
    repeat(15) @(rtr_io.cb);
  endtask: reset
endprogram: test
```

Task 12. Compile and Run