SYNOPSYS®

CUSTOMER EDUCATION SERVICES

SystemVerilog Testbench Workshop

Lab Guide

50-I-052-SLG-008 2011.12

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

Workshop Registration: **1-800-793-3448**

www.synopsys.com

Copyright Notice and Proprietary Information

Destination Control Statement

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

Disclaimer

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

Registered Trademarks, Trademarks, and Service Marks of Synopsys, Inc.

Registered Trademarks (®)

Synopsys, AMPS, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, CoMET, Design Compiler, DesignWare, Formality, Galaxy Custom Designer, HAPS, HapsTrak, HDL Analyst, HSIM, HSPICE, Identify, Leda, MAST, METeor, ModelTools, NanoSim, OpenVera, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Syndicated, Synplicity, the Synplicity logo, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Confirma, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomExplorer, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclypse, Encore, EPIC, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIM^{plus}, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, StarRC, System Compiler, System Designer, Taurus, TotalRecall, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (℠)

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: 50-I-052-SLG-008
SystemVerilog Testbench Lab Guide

1 SystemVerilog Verification Flow

Learning Objectives

After completing this lab, you should be able to:

- Create the **SystemVerilog** testbench files for a Device Under Test (DUT)
- Write a SystemVerilog **task** to reset the DUT
- **Compile and simulate** the SystemVerilog test program
- Verify that the DUT signals are driven as specified with Discovery Verification Environment (dve)

Lab Duration: 30 minutes

Getting Started

Once logged in, you will see three directories: **rtl**, **labs** and **solutions**.

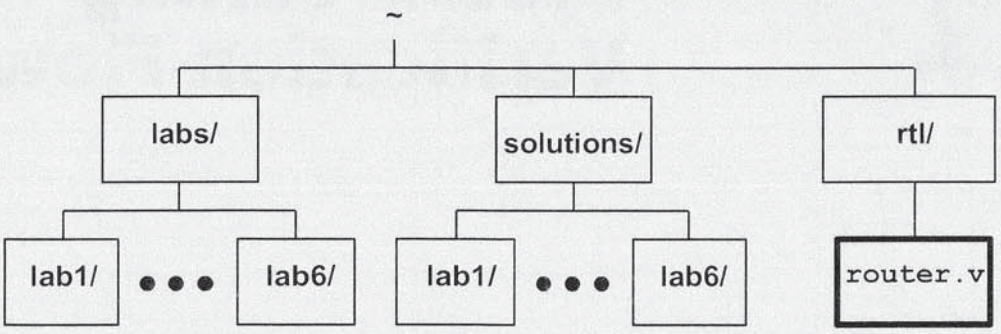


Figure 1. Lab Directory Structure

In this lab, you will develop a simple SystemVerilog test program to reset the DUT.

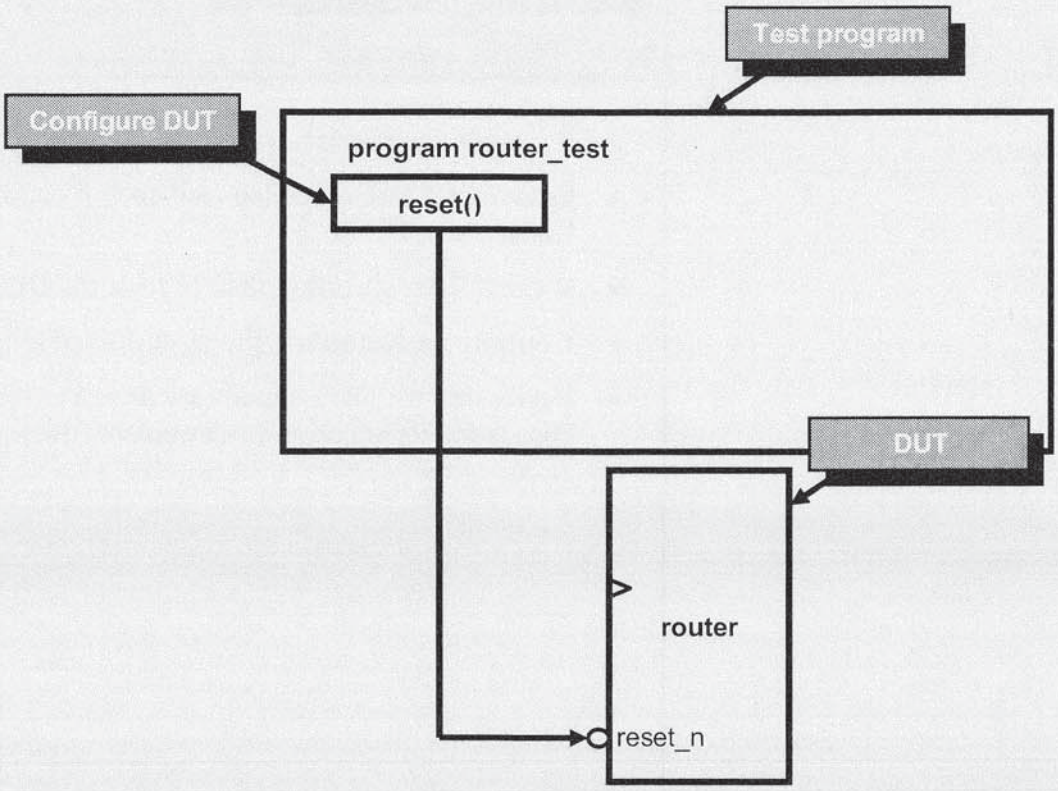


Figure 2. Testbench Architecture

Lab Overview

This lab takes you through the process of building, compiling, simulating and debugging the testbench:

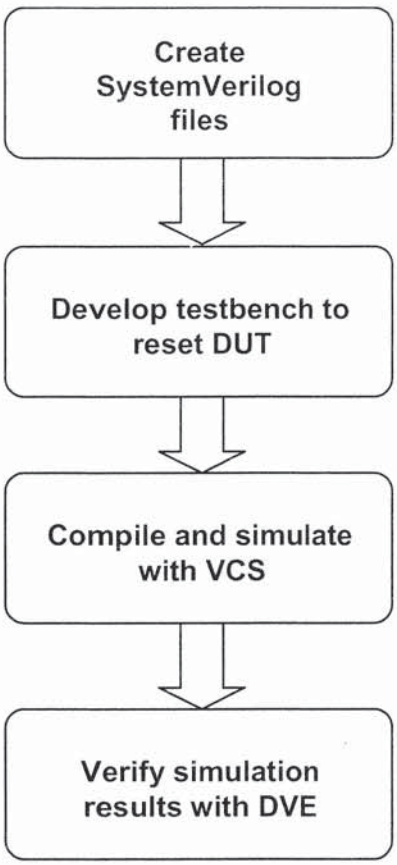


Figure 3. Lab 1 Flow Diagram

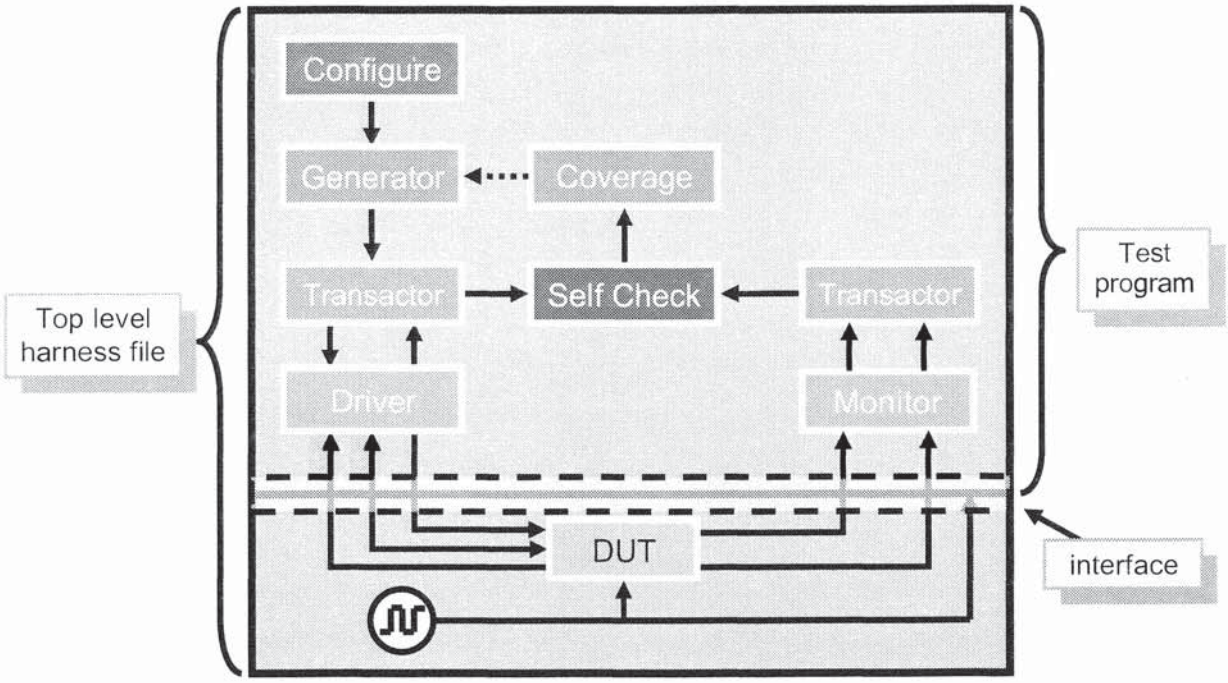
Note: You will find Answers for all questions and solutions in the Answers / Solutions at the end of this lab.

System Verilog 5 in 1.

- Verilog
- OOP
- Randomization / Constraints
- Coverage
- Assertion.

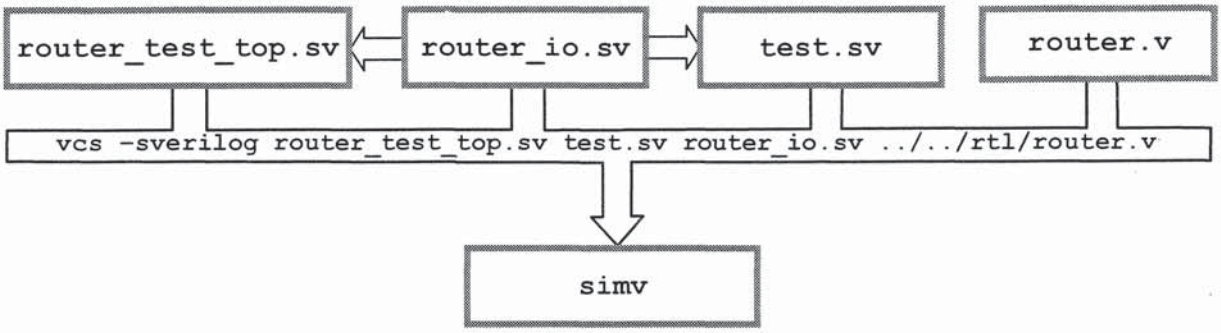
Constructing SystemVerilog Testbench

A typical architecture of a SystemVerilog testbench looks like the following:



The process with **VCS** in creating this SystemVerilog testbench is as follows:

- Create an interface to connect the test program and the DUT.
- Write test program(s).
- Connect the test and DUT using a harness file including the interface.



Task 1. Logging In

1. Log into workstation if needed (use login and password provided by instructor).
2. Go to **lab1** directory:

```
> cd labs/lab1
```

Task 2. Create SystemVerilog Interface File

1. There is a skeleton **router_io.sv** file. Open it with a text editor.

   Note: The skeleton files in the labs have comments as markers, indicated by a "ToDo" to guide you through the lab steps.

2. The signals needed to connect to the DUT are already entered for you:

```
interface router_io(input bit clock);
  logic reset_n ;
  logic [15:0] din ;
  ...
  logic [15:0] frameo_n ;
endinterface: router_io
```

   At this stage, all interface signals are asynchronous and without direction specification (i.e. input, output, inout). The direction can only be specified in a clocking block for synchronous signals or a modport for asynchronous signals.

   The next step is to create the set of synchronous signal for the test program to drive and sample the DUT signals. This is done with **clocking** block declarations.

3. Declare a clocking block driven by the posedge of the signal **clock**. This clocking block will be used by the test program to execute synchronous drives and samples. All directions for the signals in this clocking block must be with respect to the test program.

Create synchronous signals by placing signals into clocking block with direction specific to

```
interface router_io(input bit clock);
  logic  reset_n;
  logic [15:0]din;
  ...
  logic [15:0]frameo_n;
  clocking cb @(posedge clock);
    output reset_n;
    output din;     // no bit reference
    output frame_n; // no bit reference
    output valid_n; // no bit reference
    input dout;     // no bit reference
    input valido_n; // no bit reference
    input busy_n;   // no bit reference
    input frameo_n; // no bit reference
  endclocking: cb
endinterface: router_io
```

4. If desired, add specifications for input and output skews (optional):

```
interface router_io(input bit clock);
  ...
  clocking cb @(posedge clock);
    default input #1ns output #1ns;
    output reset_n;
    output din;     // no bit reference
    ...
  endclocking: cb
endinterface: router_io
```

5. Lastly, create a modport to be used for connection with the test program. In the argument list, there should be references to the clocking block created in the previous step and all other potential asynchrous signals.

```
interface router_io(input bit clock);
  ...
  clocking cb @(posedge clock);
    default input #1 output #1;
    output reset_n;
    output din;     // no bit reference
    ...
  endclocking: cb
  modport TB(clocking cb, output reset_n);
endinterface: router_io
```

6. Save and close the file.

Task 3. Create SystemVerilog Test Program File

1. Open the SystemVerilog test program file called **`test.sv`** with an editor
2. In this file, declare a test program block with arguments which connects to the TB modport in the interface block:

```
program automatic test(router_io.TB rtr_io);


endprogram: test
```

3. Within the program block, print a simple message to the screen:

```
program automatic test(router_io.TB rtr_io);
  initial begin
    $display("Hello World!");
  end
endprogram: test
```

4. Save and close the file.

Task 4. Create SystemVerilog Test Harness File

1. Open the skeleton **router_test_top.sv** provided with a text editor.
2. It looks like

```
module router_test_top;
  parameter simulation_cycle = 100;
  bit  SystemClock;
  router dut(
    .reset_n    (reset_n),
    .clock      (clock),
    .din        (din),
    .frame_n    (frame_n),
    .valid_n    (valid_n),
    .dout       (dout),
    .valido_n   (valido_n),
    .busy_n     (busy_n),
    .frameo_n   (frameo_n)
  );
  initial begin
    SystemClock = 0;
    forever begin
      #(simulation_cycle/2)
        SystemClock = ~SystemClock;
    end
  end
endmodule
```

3. Add an interface instance to the harness:

```
module router_test_top;
  parameter simulation_cycle = 100;
  bit  SystemClock;
  router_io top_io(SystemClock);
  router dut( … );
  initial begin
    SystemClock = 0;
    ...
  end
endmodule
```

Instantiate interface

4. Instantiate the test program (make I/O connection via interface instance):

```
module router_test_top;
  parameter simulation_cycle = 100;
  bit  SystemClock;
  router_io top_io(SystemClock);  // instantiating interface
  test t(top_io); // add program
  router dut( … );
  initial begin
    SystemClock = 0;
    ...
  end
endmodule
```

5. Modify DUT connection to connect via interface:

```
router dut(
  .reset_n   (top_io.reset_n),
  .clock     (top_io.clock),
  .din       (top_io.din),
  .frame_n   (top_io.frame_n),
  .valid_n   (top_io.valid_n),
  .dout      (top_io.dout),
  .valido_n  (top_io.valido_n),
  .busy_n    (top_io.busy_n),
  .frameo_n  (top_io.frameo_n)
);
```

Connect DUT via interface instance

6. Add

   a. `` `timescale ``

   b. `$timeformat`

```
`timescale 1ns/100ps
module router_test_top;
  ...
  router dut(...);
  initial begin
    $timeformat(-9, 1, "ns", 10);
    SystemClock = 0;
    ...
  end
endmodule
```

Set time (`%t`) to `ns` scale

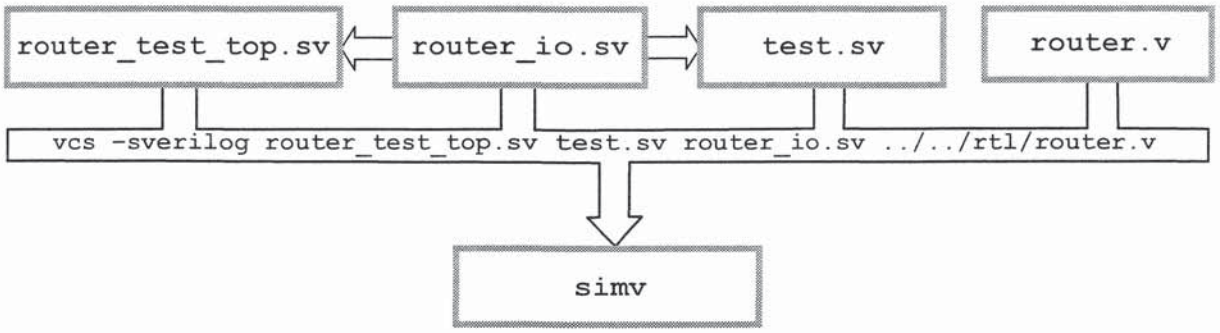
7. Save and close the file

Task 5. Compile & Simulate

1. **Compile** all files with the following command (using vcs):

```
> vcs -sverilog router_test_top.sv test.sv router_io.sv ../../rtl/router.v
```

VCS will compile the files and create an executable file called **simv**.



The next step is to execute the simulation binary.

2. Run the simulation by executing the binary created by VCS:

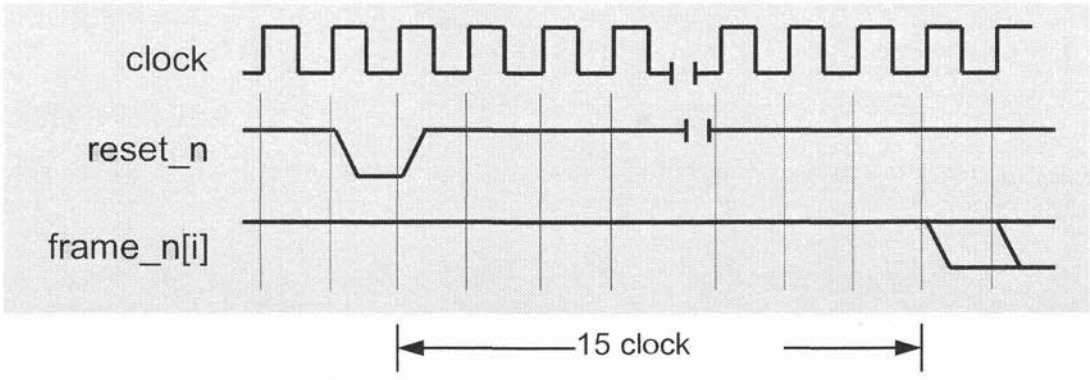
```
> simv
```

3. Check to see if you see the **$display()** message. If so, proceed to the next step. If not, debug your testbench to see why the message was not displayed.

Task 6. Reset the Router

You have successfully built, compiled and simulated a simple SystemVerilog testbench. The next step is to added needed functionality to the testbench. In this task, you will add a routine to intialize the router.

1. Open **test.sv** with a text editor.
2. In the **program** block define a task called **reset()** to reset the DUT per spec. as described in lecture: (bear in mind that **reset_n** can be either a synchronous or an asynchronous signal)



3. In the initial block, replace **$display()** with **$vcdpluson** to create a vcd+ dump file for waveform viewing.

4. Immediately afterwards, call the **reset()** task.

When completed, your program may look something like:

```
program automatic test(router_io.TB rtr_io);
  initial begin
    $vcdpluson;
    reset();
  end
  task reset();
    rtr_io.reset_n = 1'b0;          // asynchronous
    rtr_io.cb.frame_n <= '1;        // synchronous
    rtr_io.cb.valid_n <= '1;        // synchronous
    ##2 rtr_io.cb.reset_n <= 1'b1;  // synchronous
    repeat(15) @(rtr_io.cb);        // synchronous
  endtask: reset
endprogram: test
```

5. Save and close the file.

Task 7. Compile & Simulate

Once the test code is completed, compile and simulate the DUT code with the SystemVerilog testbench.

1. **Compile** all files with the following command (using vcs):

```
> vcs -sverilog -debug router_test_top.sv test.sv router_io.sv ../../rtl/router.v
```

The above compile added usage of one more switch: **-debug**.

This switch combined with the **$vcdpluson;** system task call in you test program will create a dump file "**vcdplus.vpd**" which stores the activities of signals during simulation. You will be using the VCS Discovery Visual Environment (dve) to examine the DUT signals in a waveform window.

The next step is to execute the simulation binary.

2. Run the simulation by executing the binary created by VCS:

```
> simv
```

Did the simulation execute correctly? Looking at the simulation print out, there is no way to know for sure. You will need to examine the simulation waveform with a waveform viewer to verify that your testbench was executed correctly.

For future compile/simulate iterations, there is a make file (**Makefile**) already written to simplify this process. To use it, just type "**make**" to recompile and run simulation.

The content of the **Makefile** is as follows:

(Type **make help** to get a summary description of content)

```
# Makefile for SystemVerilog Lab1
RTL= ../../rtl/router.v
SVTB = ./router_test_top.sv ./router_io.sv ./test.sv
SEED = 1

default: test

test: compile run

run:
  ./simv -l simv.log +ntb_random_seed=$(SEED)

compile:
  vcs -sverilog -debug_all $(SVTB) $(RTL)

dve:
  dve &

debug:
  ./simv -l simv.log -gui -tbug +ntb_random_seed=$(SEED)

solution: clean
  cp ../../solutions/lab1/*.sv .

clean:
  rm -rf simv* csrc* *.tmp *.vpd *.key *.log

nuke: clean
  rm -rf *.v* *.sv include .*.lock .*.old DVE* *.tcl *.h

help:
```

Take a look at the file content
Or, type **make help** to try it out

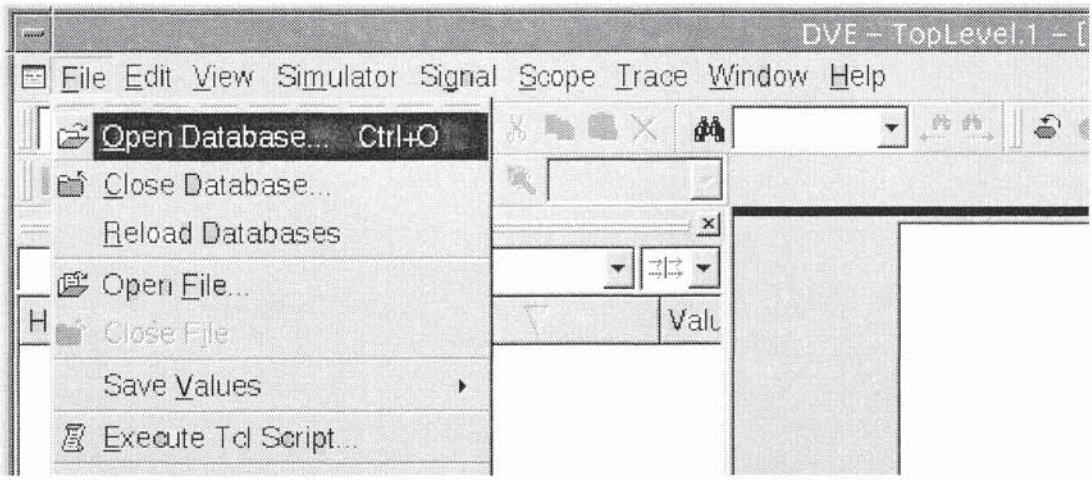
Task 8. Browse Simulation Hierachy in DVE

1. Bring up the waveform viewer using the UNIX command **dve**.

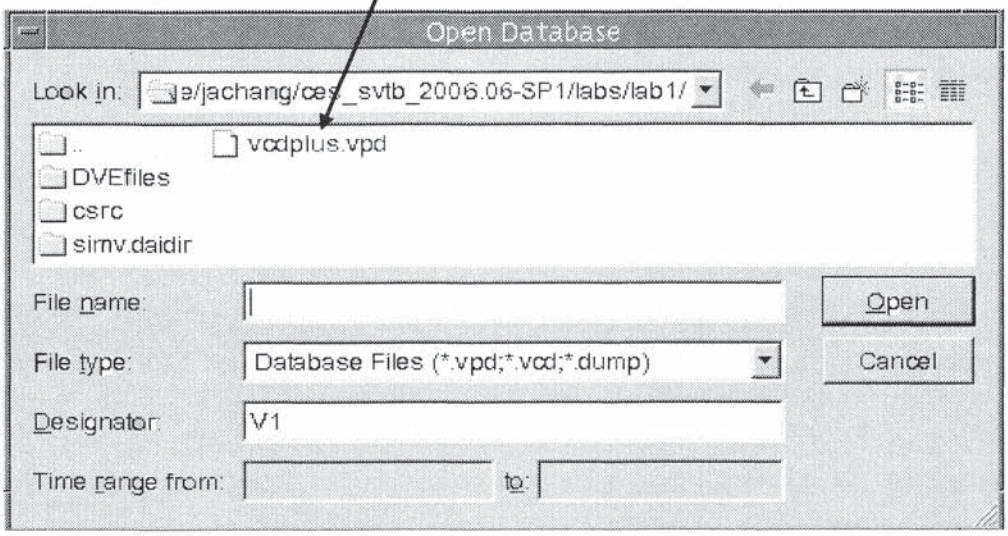
   ```
   > dve &
   ```

   It may take a while for the gui to appear.

2. Click on **File → Open Database.**



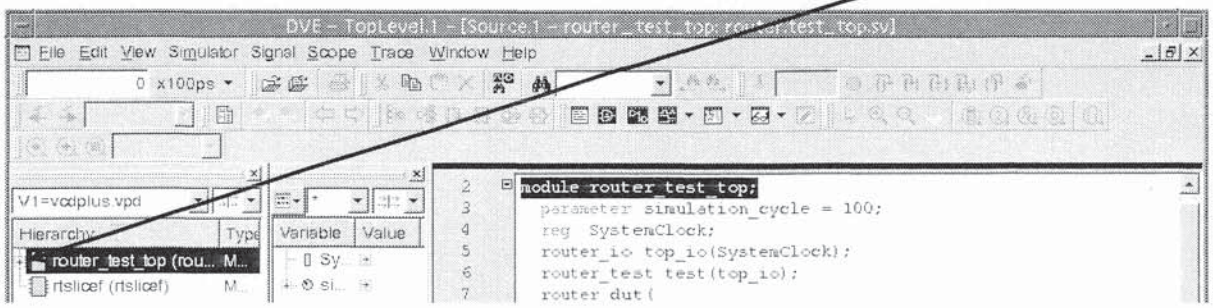
3. Double click on **vcdplus.vpd** file.



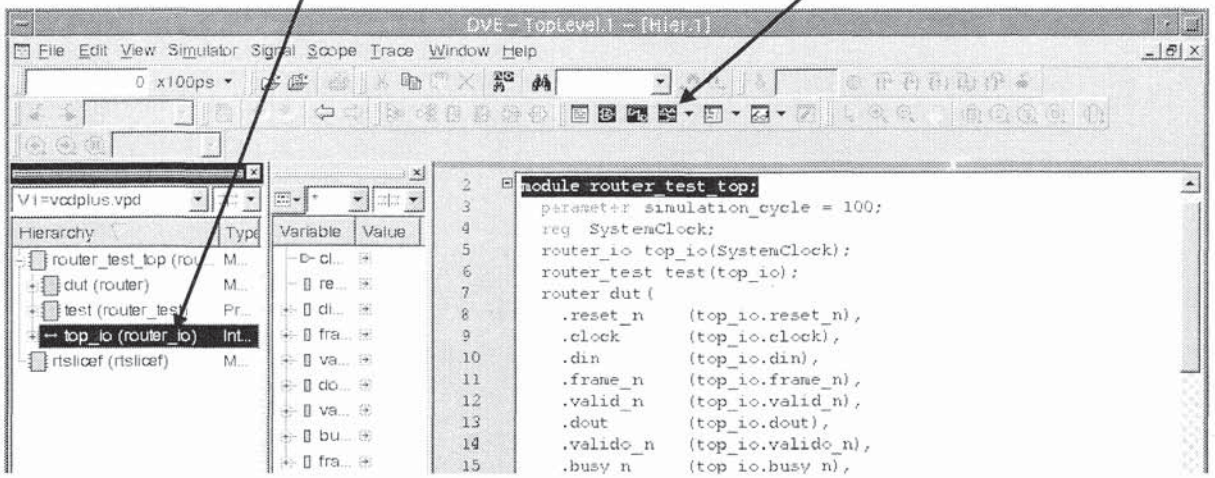
The DVE debugging windows should now be opened with the harness hierarchy and source code.

To see the waveforms you will need to open a Waveform window.

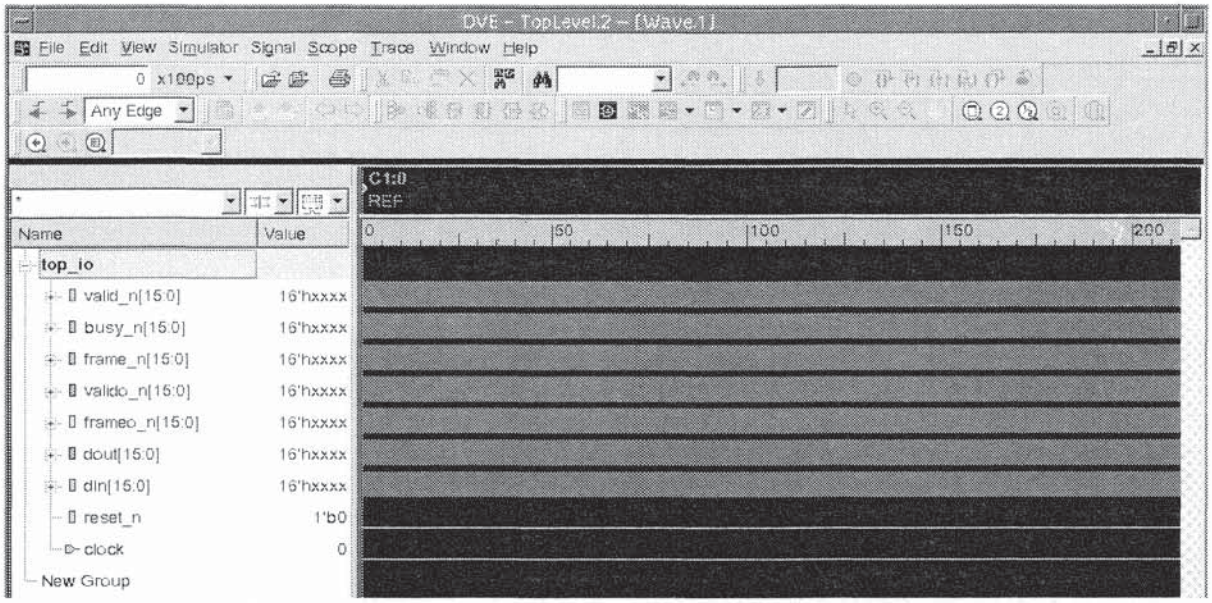
4. Expand harness to access interface signals by clicking on ⊞.



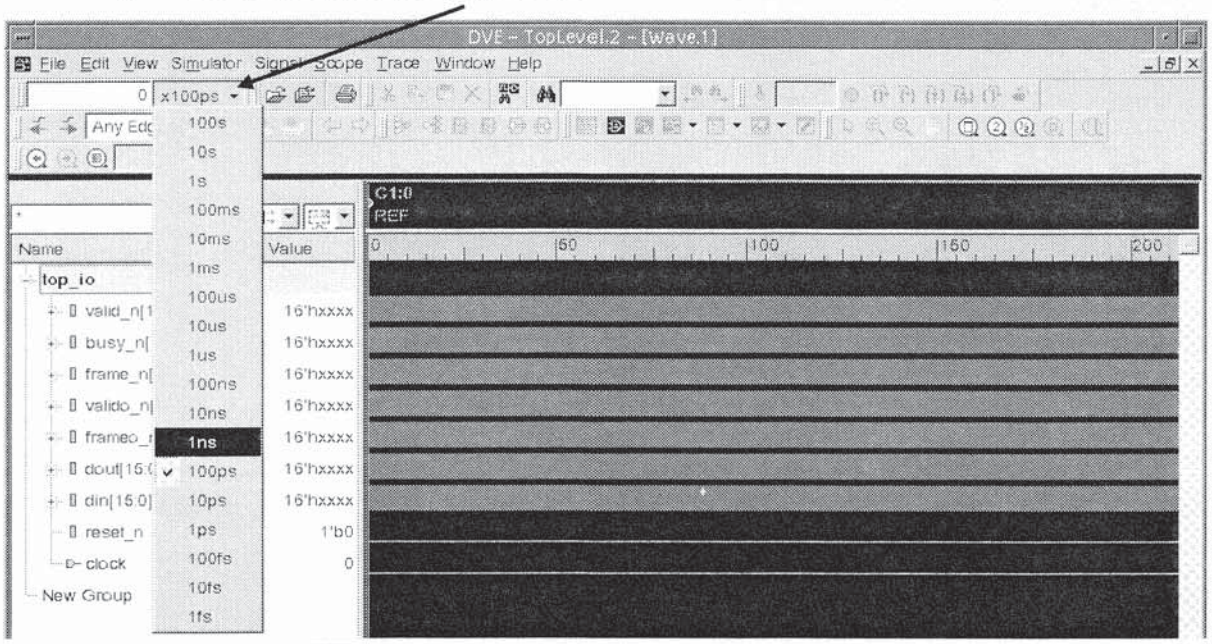
5. Click interface instance to highlight, then click on  to display waveform.



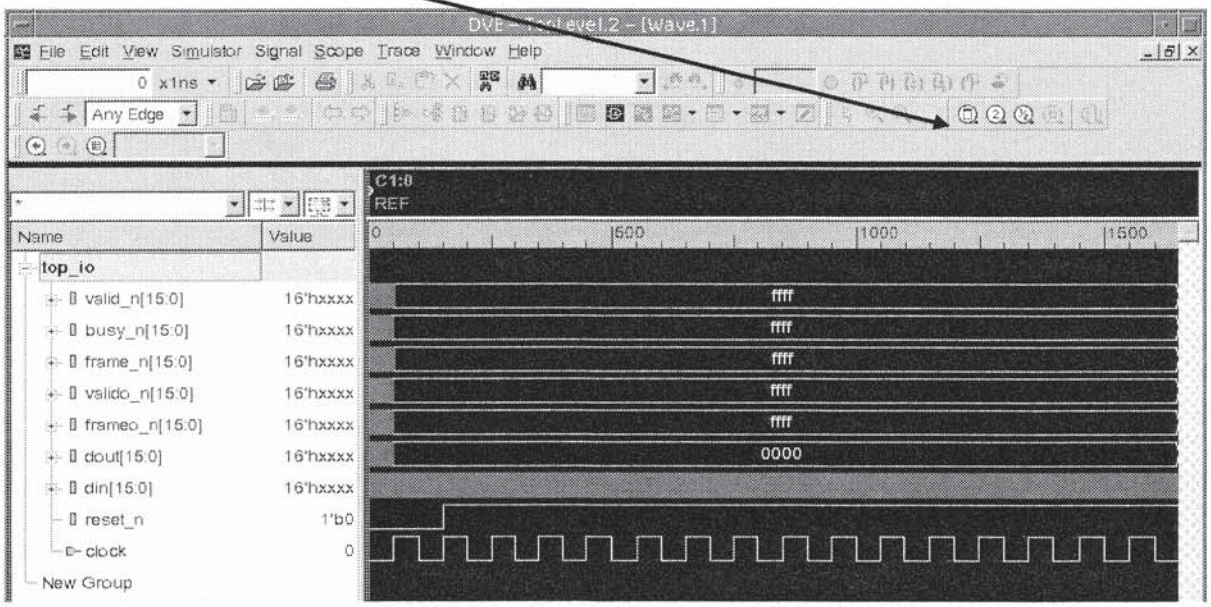
6. The **Waveform** window should open up looking like the following:



7. Set waveform time scale to 1ns



8. Click on  to expand window to see through all simulation time.



9. Verify the correct timing of the reset signal. If there are errors, correct the error then type "**make**" to re-compile and re-simulate the modified code.

10. You can save the waveform setting with File -> Save Session menu command. In the pop-up dialog box, save the session as **router_waves.tcl**. Make sure to save this file again whenever you change the waveform settings.

11. Reuse the setting with File -> Recent Session -> ../lab1/router_waves.tcl in future labs after loading the database as shown in step 2 above.

If there are no errors, you are done with lab1.

Answers / Solutions

router_io.sv Solution:

```
interface router_io(input bit clock);
  logic         reset_n;
  logic [15:0]  din;
  logic [15:0]  frame_n;
  logic [15:0]  valid_n;
  logic [15:0]  dout;
  logic [15:0]  valido_n;
  logic [15:0]  busy_n;
  logic [15:0]  frameo_n;

  clocking cb @(posedge clock);
    default input #1ns output #1ns;
    output reset_n;
    output din;
    output frame_n;
    output valid_n;
    input  dout;
    input  valido_n;
    input  busy_n;
    input  frameo_n;
  endclocking: cb

  modport TB(clocking cb, output reset_n);
endinterface: router_io
```

test.sv Solution:

```
program automatic test(router_io.TB router);

  initial begin
    $vcdpluson;
    reset();
  end

  task reset();
    router.reset_n <= 1'b0;
    router.cb.frame_n <= '1;
    router.cb.valid_n <= '1;
    ##2 router.cb.reset_n <= 1'b1;
    repeat(15) @(router.cb);
  endtask: reset

endprogram: test
```

router_test_top.sv Solution:

```
`timescale 1ns/100ps
module router_test_top;
  parameter simulation_cycle = 100;

  bit  SystemClock;
  router_io top_io(SystemClock);
  test t(top_io);
  router dut(
    .reset_n     (top_io.reset_n),
    .clock (top_io.clock),
    .din    (top_io.din),
    .frame_n     (top_io.frame_n),
    .valid_n     (top_io.valid_n),
    .dout  (top_io.dout),
    .valido_n    (top_io.valido_n),
    .busy_n      (top_io.busy_n),
    .frameo_n    (top_io.frameo_n)
  );

  initial begin
    $timeformat(-9, 1, "ns", 10);
    SystemClock = 0;
    forever begin
      #(simulation_cycle/2)
        SystemClock = ~SystemClock;
    end
  end

endmodule
```

This page was intentionally left blank.

2 Sending Packets Through Router

Learning Objectives

After completing this lab, you should be able to:

- Extend the SystemVerilog testbench from lab1 to send a packet from an input port through an output port
- Compile and simulate the router with the updated SystemVerilog testbench
- Verify stimulus generation and signal drives with DVE

Lab Duration: 60 minutes

Getting Started

In lab1, you have familiarized with the process of building the simulation environment for verifying a DUT with SystemVerilog.

In this lab, you will continue to build the next component of the testbench: stimulus generator, protocol transactors and device drivers. You will develop a set of routines to drive one packet into input port 3 and visually observe the payload of this packet coming out of output port 7.

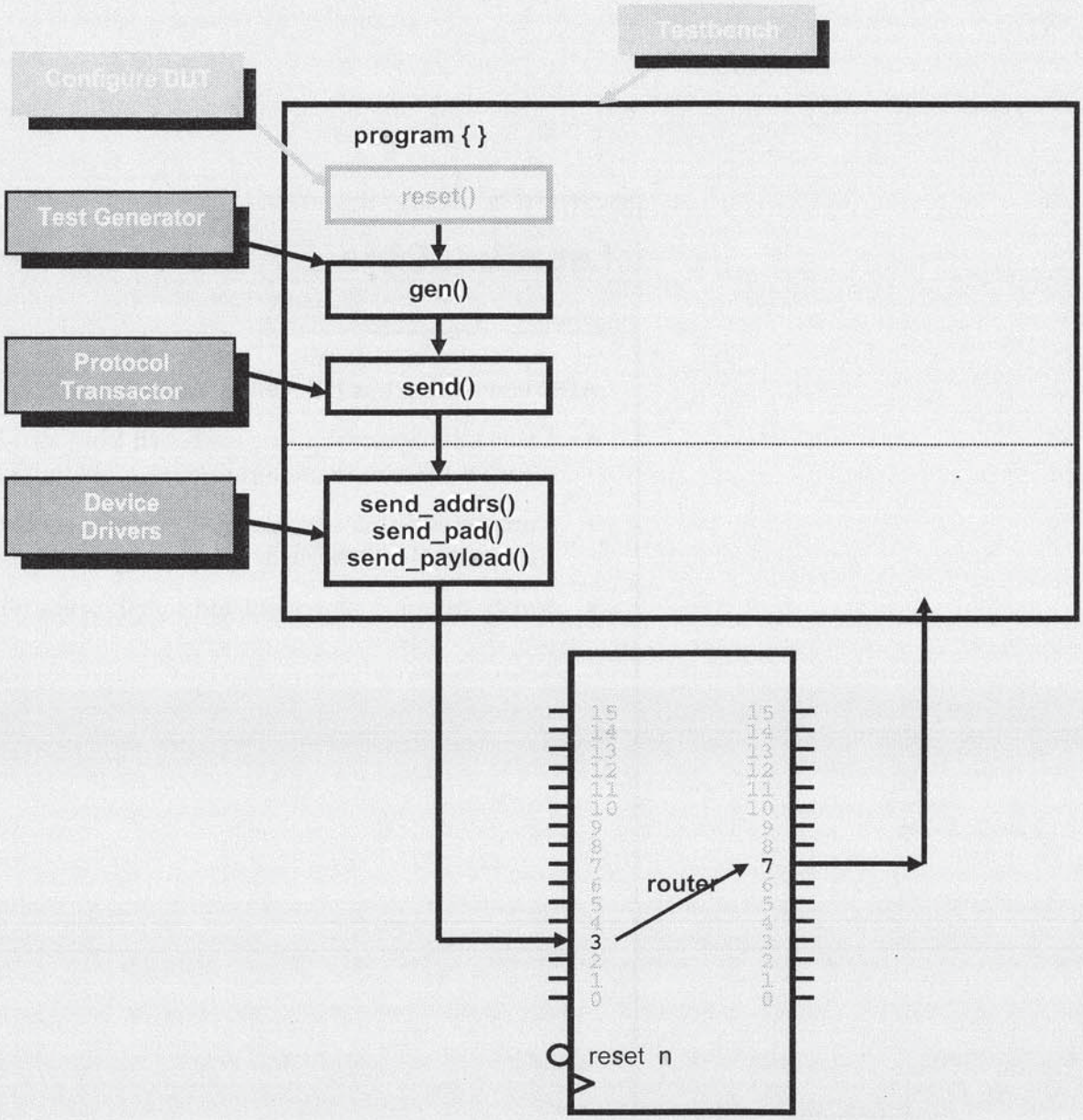


Figure 1. Lab 2 testbench architecture

Lab Overview

The process of updating, compiling, simulating and debugging the testbench:

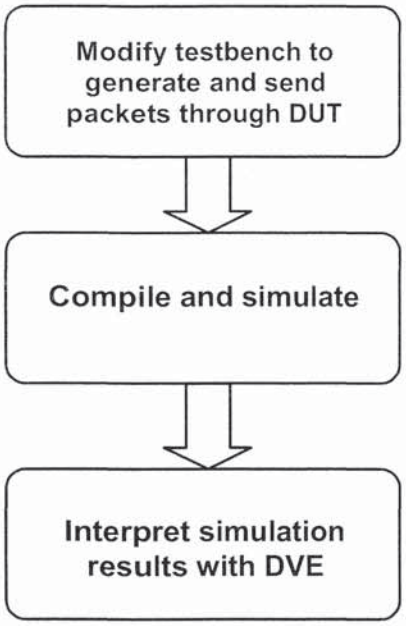


Figure 2. Lab 2 Flow Diagram

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

Sending a Packet through the Router

Task 1. Copy Files from the Previous Directory

To keep everyone in the workshop synchronized in developing the SystemVerilog testbench code, you will adopt a strategy of basing all lab work on standard solutions from previous labs. For lab 2, you will copy the files from the lab1 solutions directory, and develop your lab 2 testbench based on these files.

1. CD into the lab2 directory.

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

2. Copy the source files in the **solutions/lab1** directory into the current directory with **make** script.

```
> make copy
```

Note: If you chose to use your own lab files from lab1, type "**make mycopy**". However, your files will not have the ToDo markers to guide you.

Task 2. Declaring Program Global Variables

Sending a **packet** through the router requires specifying which **input port** and **output port** to use, and what **data** to send.

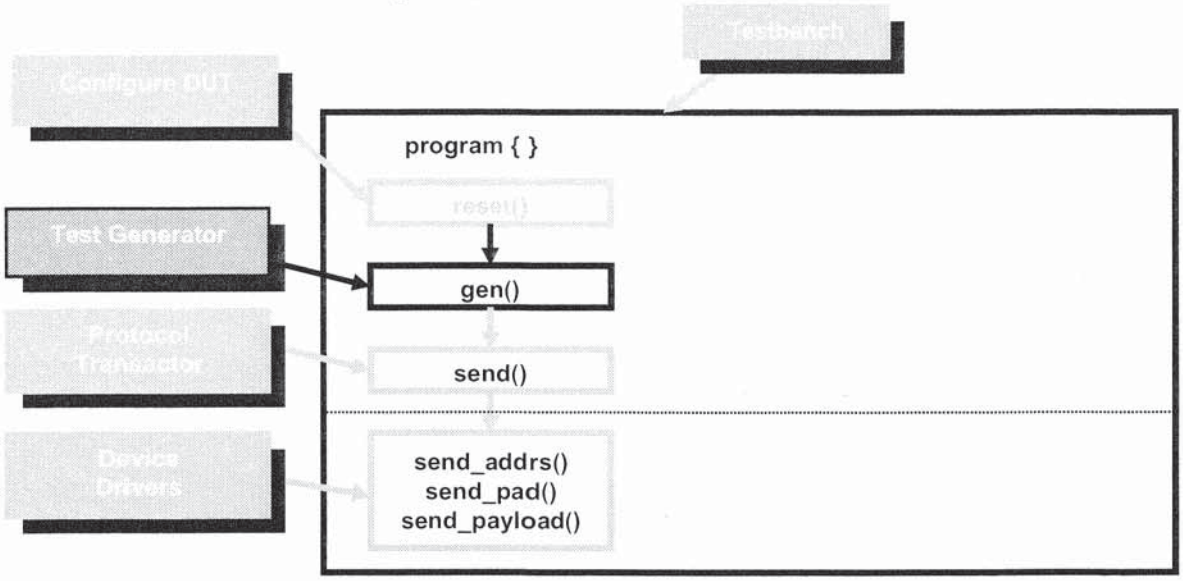
To make referencing these variables simple, you will declare them as program globals.

1. Open **test.sv** with text editor.
2. Declare the **program globals** for the packet:

```
bit[3:0] sa;              // source address (input port)
bit[3:0] da;              // destination address (output port)
logic[7:0] payload[$];    // packet data array
                          // logic stores 0, 1, x and z values
```

Task 3. Generate Packet Data

In Lab 1, you configured the DUT by calling the **reset()** subrouinte. You will continue the testbench development by creating a subroutine to generate the test stimulus in a task called "**gen()**".



1. In the **program intial** block, call the generator, **gen()**, after **reset()**
2. Following the task **reset()** code block, declare task **gen()**:

```
program automatic test(router_io.TB rtr_io);
  bit[3:0] sa;           // source address
  bit[3:0] da;           // destination address
  logic[7:0] payload[$]; // packet data array

  initial begin
    $vcdpluson;
    reset();
    gen();
  end

  task reset();
    ...
  endtask: reset

  task gen();

  endtask: gen
endprogram: test
```

3. In the body of the **gen()** task:
   a) Set **sa** to 3 and **da** to 7.
   b) Fill the **payload** queue with 2 to 4 random bytes.

   When completed, your code should look something like:

```
task gen();
  sa = 3;
  da = 7;
  payload.delete();
  repeat($urandom_range(2,4))
    payload.push_back($urandom);
endtask: gen
```

Task 4. Create Send Packet Routines

With the packet information generated, you are now ready to develop the transactor and device drivers routines to send a packet through the router.

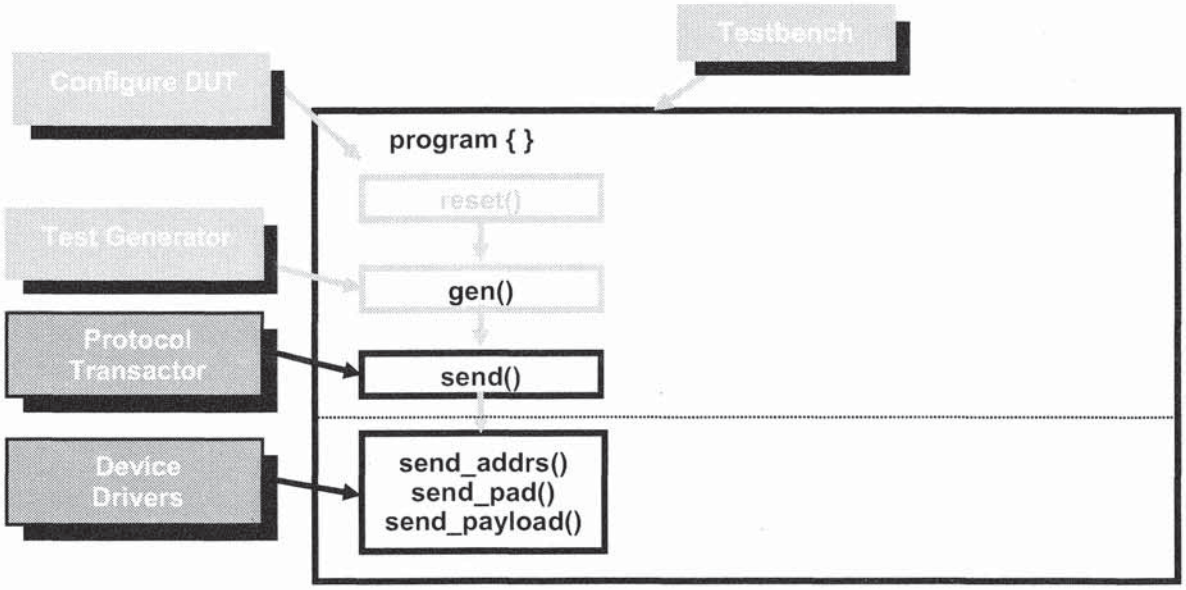
Sending a packet through the router is done using three processes:

Send the:

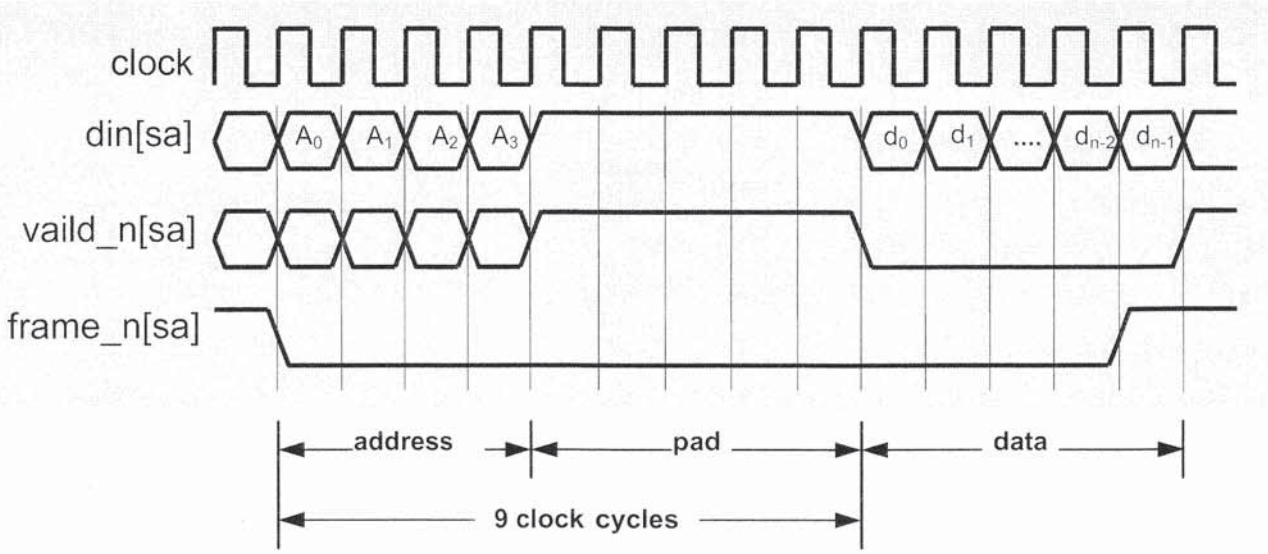
- Destination Address
- Padding bits
- Payload

Each process will be developed as an individual device driver routine. A transactor routine will call these device driver routines to execute a complete transaction.

The distinction between device drivers and transactor is that device drivers interact with the hardware signals directly while the transactor calls the device drivers. These layers of abstraction make the testbench routines more manageable, portable and reliable.



The following steps take you through the development of these routines.



1. In the initial block, call **`send()`** task to send a packet immediately after the **`gen()`** statement.
2. Then, advance simulation time by 10 additional clock cycles after the **`send()`** call.

   This will allow the data coming out of the router to be observed. Otherwise, when you view the waveform in DVE, the output will appear to be cut off.
3. Add a declaration of **`send()`** task in the program block.
4. In the body of the **`send()`** task, code the following operations:
   - Call **`send_addrs()`**
   - Call **`send_pad()`**
   - Call **`send_payload()`**
5. Create the **`send_addrs()`** task.

   This is the device driver that drive the four bits of address into the router.
6. In the body of the **`send_addrs()`** task, code the following operations:
   - Drive the **`frame_n`** signal as per router specification
   - Drive the **`din`** signal with the destination address (LSB first)

   Recall that **`din`** is a single-bit serial signal. Use a loop construct to drive **`din`** one bit per clock cycle for four cycles.

Note: Each port is represented by an individual bit in **frame_n**, **din**, and **valid_n**. Make sure you specify the correct bit.

Driving the **frame_n** signal for input port 3 to "1" takes the form of: `rtr_io.cb.frame_n[3] <= 1'b1;`

Question 1. What will this driving statement do?
`rtr_io.cb.frame_n <= 1;`

.....

.....

7. Create the **send_pad()** task.

   This is the device driver that drives the five padding bits into the router.

8. In the body of the **send_pad()** task, code the following operations:

   - Drive the **frame_n**, **valid_n** and **din** signals as per router specification

9. Create the **send_payload()** task.

   This is the device driver that sends the payload (data) into the router.

10. In the body of the **send_payload()** task, code the following operations:

    - Write a loop to execute "**payload.size()**" number of times.
    - Within this loop, each 8-bit datum of the **payload[$]** array should be transmited one bit per clock cycle starting with the lsb.
    - Also remember to drive the valid bit, **valid_n**, as per the router specification.
    - For the last valid bit of the packet, make sure to set the **frame_n** signal to 1'b1 as per router specification.
    - Return the **valid_n** signal back to 1'b1 after the packet completes.

*** ***Warning*** ***

Pay particular attention to how you advance the clock. If in one subroutine you advance the clock upon exiting the subroutine, then in another subroutine you advance the clock upon entering the subroutine, erroneous timing may result.

11. Save and close the file.

Task 5. Compile And Debug The Program

1. Compile and simulate the SystemVerilog testbench with **`make`**.
2. If you see any runtime errors, you must correct them now.
3. If there are no runtime errors, check the operation using DVE.

   When you use DVE from this point onwards, you will want to look at individual port signals. The easiest way to do this is to drag the individual bit you want to see into a new group in the Waveform window.

Task 6. Extend The Program To Send 21 Packets

1. Modify the program block to send 21 packets through the router using the same sa = 3 & da =7.
2. Compile, simulate & verify via DVE.

Congratulations, you have completed lab2!

Answers / Solutions

Task 1. Create Send Packet Routines

Question 1. What will this driving statement do?

```
rtr_io.cb.frame_n <= 1;
```

This will drive all 16 input port's frame_n signal. Port 0's frame_n signal will be driven to "1". Ports 1 – 15's frame_n signal will be driven to "0".

If the intention is to drive all 16 input ports to "1" the correct statement would be:

```
rtr_io.cb.frame_n <= '1;
```

If the intention is to drive an individual port, then the input port number must be specified as a bit selection in the rtr_io.frame_n signal.

e.g. driving input port 5's frame_n signal to "0" takes the following form:

```
rtr_io.cb.frame_n[5] <= 1'b0;
```

test.sv Solution:

```
program automatic test(router_io.TB rtr_io);
  int run_for_n_packets;      // number of packets to test
    bit[3:0] sa;              // source address
    bit[3:0] da;              // destination address
    logic[7:0] payload[$];    // packet data array

    initial begin
      $vcdpluson;
      run_for_n_packets = 21;
      reset();
      repeat(run_for_n_packets) begin
        gen();
        send();
      end
      repeat(10) @(rtr_io.cb);
    end

    task reset();
      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

    task gen();
      sa = 3;
      da = 7;
      payload.delete();
      repeat($urandom_range(2,4))
        payload.push_back($urandom);
    endtask: gen

    task send();
      send_addrs();
      send_pad();
      send_payload();
    endtask: send

    task send_addrs();
      rtr_io.cb.frame_n[sa] <= 1'b0;
      for(int i=0; i<4; i++) begin
        rtr_io.cb.din[sa] <= da[i];
        @(rtr_io.cb);
      end
    endtask: send_addrs
```

```
    task send_pad();
      rtr_io.cb.frame_n[sa] <= 1'b0;
      rtr_io.cb.valid_n[sa] <= 1'b1;
      rtr_io.cb.din[sa] <= 1'b1;
      repeat(5) @(rtr_io.cb);
    endtask: send_pad

    task send_payload();
      foreach(payload[index]) begin
        for(int i=0; i<8; i++) begin
          rtr_io.cb.din[sa] <= payload[index][i];
          rtr_io.cb.valid_n[sa] <= 1'b0;
          rtr_io.cb.frame_n[sa] <= (index == (payload.size()-1)) &&
(i==7);
          @(rtr_io.cb);
        end
      end
      rtr_io.cb.valid_n[sa] <= 1'b1;
    endtask: send_payload

  endprogram: test
```

3 Self-Checking

Learning Objectives

After completing this lab, you should be able to:

- Develop a monitor to sample the output of the router
- Develop a checker to verify the output of the router
- Run driver and monitor routines concurrently
- Verify the self-checking mechanism by executing the testbench against a faulty DUT

Lab Duration: 90 minutes

Getting Started

In Lab 3, you will add in the monitors and self-check mechanisms. The architecture is shown below:

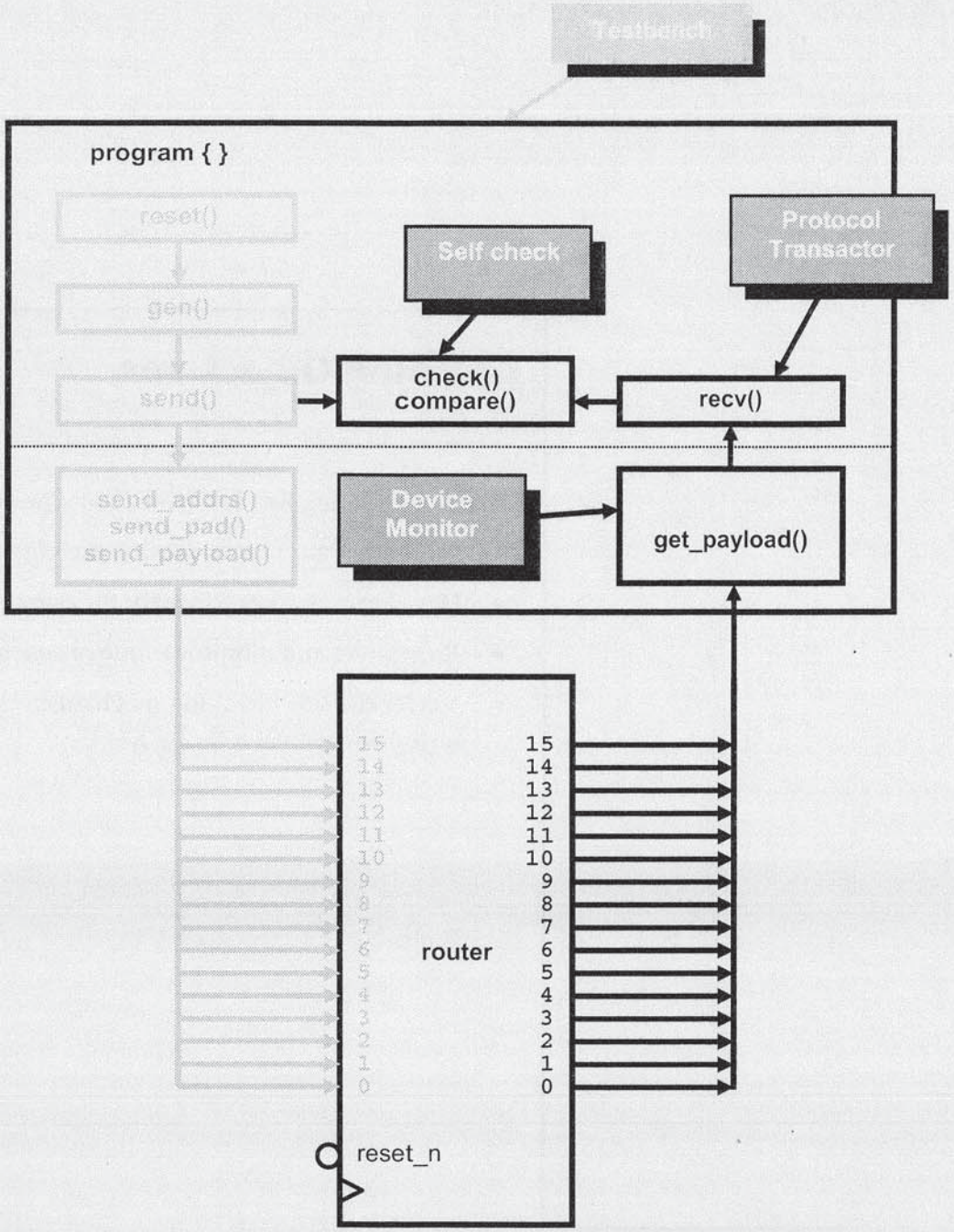


Figure 1. Lab 3 testbench architecture

Lab Overview

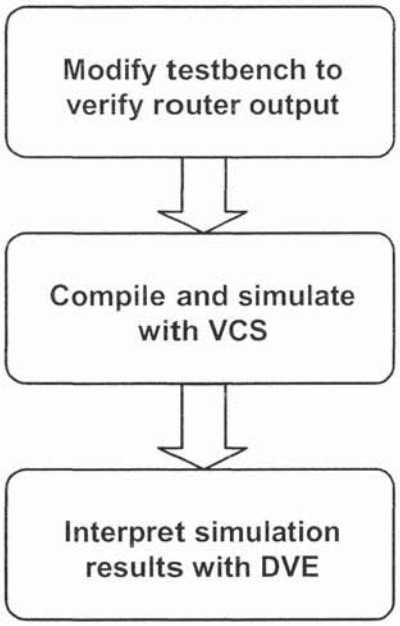


Figure 2. Lab 3 Flow Diagram

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

Self-Checking & Concurrency

Task 1. Copy Files From Lab 2's Solutions Directory

For consistency, copy the files from `../../solutions/lab2` into the `lab3` directory.

1. Go into the lab3 directory.

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

2. Copy the source files in the `solutions/lab2` directory into the current directory with `make` script.

```
> make copy
```

(If you chose to use your own lab files from lab2, type "`make mycopy`".)

Task 2. Build The Top-Level Test Environment

In the following steps, you will complete building of the top-level test environment.

1. Edit **`test.sv`** file.
2. Add a global declaration for an 8-bit (**`logic[7:0]`**) queue named: **`pkt2cmp_payload[$]`**

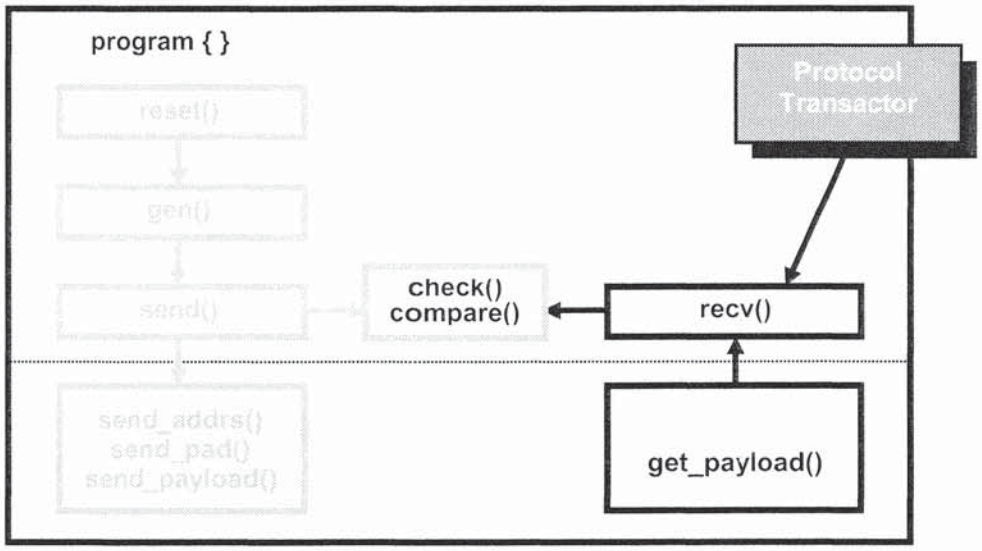
   This queue will be used to store the data sampled from the DUT.
3. For self-checking, modify the program to execute a receive routine **`recv()`** concurrently with **`send()`** followed by a self-checking routine **`check()`**.

```
program automatic router_test(router_io.TB router);
  ...
  logic[7:0] pkt2cmp_payload[$];

  initial begin
    ...
    repeat(run_for_n_packets) begin
      gen();
      fork
        send();
        recv();
      join
      check();
    end
  end
  ...
endprogram
```

Task 3. Develop Monitor Routines

In this step, you will build the monitor transactor called **recv()**.



This **recv()** transactor will call device monitor **get_payload()** to retrieve a packet payload from the router. This payload shall be stored in **pkt2cmp_payload[$]** queue by the **get_payload()** routine.

1. Declare the **recv()** task.
2. In the body of **recv()** call **get_payload()** to retrieve the payload.

   For now, this is the only content of the **recv()** routine. More will be added in later labs.
3. Declare the **get_payload()** task.
4. In **get_payload()**, delete content of **pkt2cmp_payload[$]**.

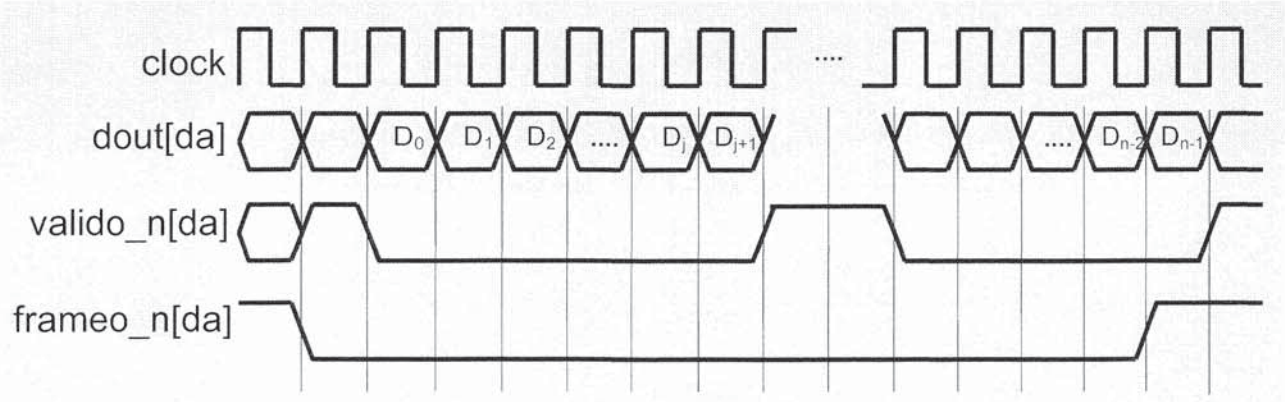
   This is necessary to remove potential residues from previous packet.
5. Continuing in **get_payload()**, wait for the falling edge of the output frame signal.

   Question 1. How do you implement a watchdog timer for **@(negedge router.cb.frameo_n[da])**?
   Hint: Remember to put it in an enclosing fork-join to prevent disable fork problems discussed in the class slides.

   .....

   .....

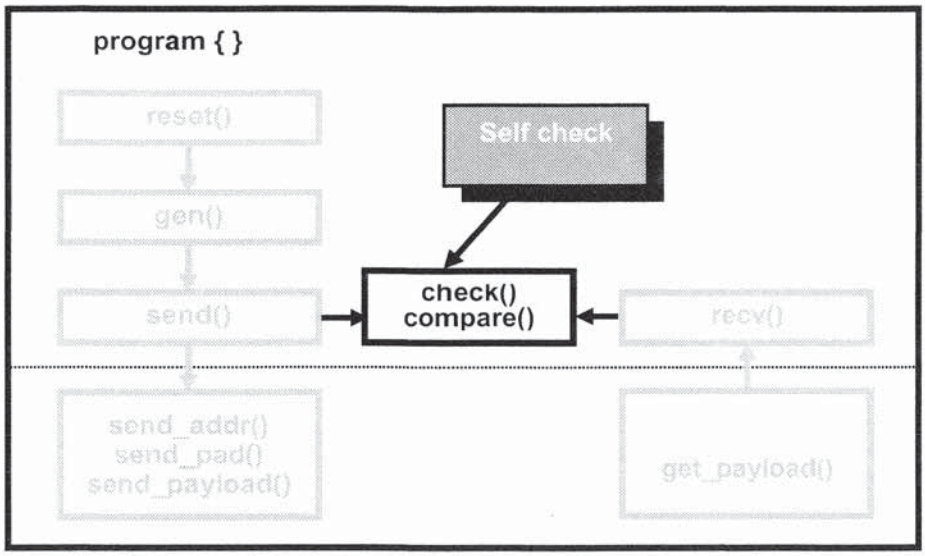
   .....



6. Continue to develop **get_payload()** routine by sampling the output of router:
   - Loop until end of frame is detected.
   - Within the loop, assemble a byte of data at a time (minimum of 8 clock cycles). Then, store each 8-bit datum into the `pkt2cmp_payload[$]` queue.
7. If payload is not byte aligned, print a message to terminal and end simulation.

Task 4. Develop The Checker

The last step in completing this basic testbench is to develop the checker for checking the output of the router.



1. Create a function called `compare()` which returns a single bit and has a pass-by-reference string argument.

2. In the body of `compare()`, compare the data in `payload[$]` (reference packet data) and `pkt2cmp_payload[$]` (sampled packet data) to verify that the payload is received correctly.
   - If the sizes of the `payload[$]` and `pkt2cmp_payload[$]` arrays do not match, set the string argument with a description of the error, return a value of `0` and terminate the subroutine.
   - If the data in `payload[$]` and `pkt2cmp_payload[$]` arrays do not match, set the string argument with a description of the error, return a value of `0` and terminate the subroutine. You can directly compare arrays in SystemVerilog using the == operator.
   - If the data in `payload[$]` and `pkt2cmp_payload[$]` arrays are the same, set the string argument with a description of the success, return a value of 1 and terminate the subroutine.
3. Create a task called `check()`.
4. In the body of check(), declare a string variable message and a counter for packets pkts_checked.
5. In the body of `check()`, call the `compare()` function to check the packet received.
   - If an error is detected, print the error message to teminal then end simulation.
   - If check is successful, print a message indicating the number of packets successfully checked to terminal.
6. Save and close the file.

Task 5. Compile and Run

1. Use `make` script to compile and run your program.
2. Make sure the simulation completes successfully. Correct all errors.

Task 6. Test All Ports

1. Modify your test program to randomly generate `sa` and `da`.
2. Extend your testbench to send **2000** packets.
3. Use `make` script to compile and run your program.
4. Make sure the simulation still completes successfully.

Task 7. Expand To Detect RTL Error

1. Run your testbench against a bad rtl code

   A bad RTL code has been included in the rtl directory. The script to run your testbench is already embedded in the Makefile. To run the script, type the following command.

   Compile and run simulation

   ```
   > make bad
   ```

   If your simulation stops on an error you are done. If not, your testbench is not working properly. Correct your testbench and try again.

Congratulations, you completed Lab 3!

Answers / Solutions

Task 3. Develop Monitor Routines

5. Continuing in **get_payload()**, wait for the falling edge of the output frame signal.

Question 1. How do you implement a watchdog timer for **@(negedge router.cb.frameo_n[da])**?

There is a potential that simulation may run forever if there is an error in testbench coding.

Consider the following code:

```
for (i = 0; i < run_for_n_packets; i++)
  gen();
  fork
    begin
      send();
      recv();
    end
  join
```

The **send()** and **recv()** routines were supposed to be called concurrently inside a fork-join construct. The coder mistakenly put both inside a begin-end construct making execution of **send()** and **recv()** serial rather than concurrent.

If you use @(negedge router.cb.frameo_n[da]) to detect the beginning of a new packet at the router's output the simulation would run forever, you would never know that there was a mistake or what the mistake was.

To prevent this type of run-away simulation, you can add a watchdog timer and have the routines timeout if a problem similar to the example occurs. The following code sample illustrates just such a watchdog timer implementation:

```
fork
  begin: wd_timer_fork
  fork: frameo_wd_timer
    @(negedge router.cb.frameo_n[da]);
    begin
      repeat(1000) @(router.cb);
      $display("\n%m\n[ERROR]%t Frame signal timed out!\n", $realtime);
      $finish;
    end
  join_any: frameo_wd_timer
  disable fork;
  end: wd_timer_fork
join
```

test.sv Solution:

```
program automatic test(router_io.TB router);
  int run_for_n_packets;     // number of packets to test
  bit[3:0] sa;               // source address
  bit[3:0] da;               // destination address
  logic[7:0] payload[$];           // expected packet data array
  logic[7:0] pkt2cmp_payload[$];   // actual packet data array

  initial begin
    $vcdpluson;
    run_for_n_packets = 2000;
    reset();
    repeat(run_for_n_packets) begin
      gen();
      fork
        send();
        recv();
      join
      check();
    end
    repeat(10) @(router.cb);
  end

  task reset();
    router.reset_n <= 1'b0;
    router.cb.frame_n <= '1;
    router.cb.valid_n <= '1;
    ##2 router.cb.reset_n <= 1'b1;
    repeat(15) @(router.cb);
  endtask: reset

  task gen();
    sa = $urandom;
    da = $urandom;
    payload.delete();
    repeat($urandom_range(2,4))
      payload.push_back($urandom);
  endtask: gen

  task send();
    send_addrs();
    send_pad();
    send_payload();
  endtask: send

  task send_addrs();
    router.cb.frame_n[sa] <= 1'b0;
    for(int i=0; i<4; i++) begin
      router.cb.din[sa] <= da[i];
      @(router.cb);
    end
  endtask: send_addrs

  task send_pad();
    router.cb.frame_n[sa] <= 1'b0;
    router.cb.valid_n[sa] <= 1'b1;
    router.cb.din[sa] <= 1'b1;
    repeat(5) @(router.cb);
  endtask: send_pad
```

```
  task send_payload();
    foreach(payload[index]) begin
      for(int i=0; i<8; i++) begin
        router.cb.din[sa] <= payload[index][i];
        router.cb.valid_n[sa] <= 1'b0;
        router.cb.frame_n[sa] <= (index == (payload.size() - 1)) && (i == 7);
        @(router.cb);
      end
    end
    router.cb.valid_n[sa] <= 1'b1;
  endtask: send_payload

  task recv();
    get_payload();
  endtask: recv

  task get_payload();
    pkt2cmp_payload.delete();
    fork
      begin: wd_timer_fork
      fork: frameo_wd_timer
        @(negedge router.cb.frameo_n[da]);
        begin
          repeat(1000) @(router.cb);
          $display("\n%m\n[ERROR]%t Frame signal timed out!\n", $realtime);
          $finish;
        end
      join_any: frameo_wd_timer
      disable fork;
      end: wd_timer_fork
    join
    forever begin
      logic[7:0] datum;
      for (int i=0; i<8; ) begin
        if (!router.cb.valido_n[da])
          datum[i++] = router.cb.dout[da];
        if (router.cb.frameo_n[da])
          if (i == 8) begin
            pkt2cmp_payload.push_back(datum);
            return;
          end
          else begin
            $display("\n%m\n[ERROR]%t Packet payload not byte aligned!\n",
$realtime);
            $finish;
          end
        @(router.cb);
      end
      pkt2cmp_payload.push_back(datum);
    end
  endtask: get_payload
```

```
  function bit compare(ref string message);
    if (payload.size() != pkt2cmp_payload.size()) begin
      message = "Payload Size Mismatch:\n";
      message = { message, $sformatf("payload.size() = %0d,
pkt2cmp_payload.size() = %0d\n", payload.size(), pkt2cmp_payload.size()) };
      return(0);
    end
    if (payload == pkt2cmp_payload) ;
    else begin
      message = "Payload Content Mismatch:\n";
      message = { message, $sformatf("Packet Sent:  %p\nPkt Received: %p",
payload, pkt2cmp_payload) };
      return(0);
    end
    message = "Successfully Compared";
    return(1);
  endfunction: compare

  task check();
    string message;
    static int pkts_checked = 0;
    if (!compare(message)) begin
      $display("\n%m\n[ERROR]%t Packet #%0d %s\n", $realtime, pkts_checked,
message);
      $finish;
    end
    $display("[NOTE]%t Packet #%0d %s", $realtime, pkts_checked++, message);
  endtask: check

endprogram: test
```

4 OOP Encapsulation

Learning Objectives

After completing this lab, you should be able to:

- Encapsulate packet information into a **`Packet`** class
- Utilize **randomization** in `Packet` class to randomly generate source address, destination address and payload
- Create two `Packet` objects, one for the input into the DUT, the other for reconstructing the output of the DUT
- Use the `compare()` method embedded in the Packet objects to verify the correctness of DUT operation

Lab Duration: 90 minutes

Getting Started

In Lab 3, you added the monitor and self-check. In this lab, you will encapsulate the packet information into a class structure. You will create random `Packet` objects in the generator then send, receive and check the correctness of the DUT using these `Packet` objects.

The architecture is shown below:

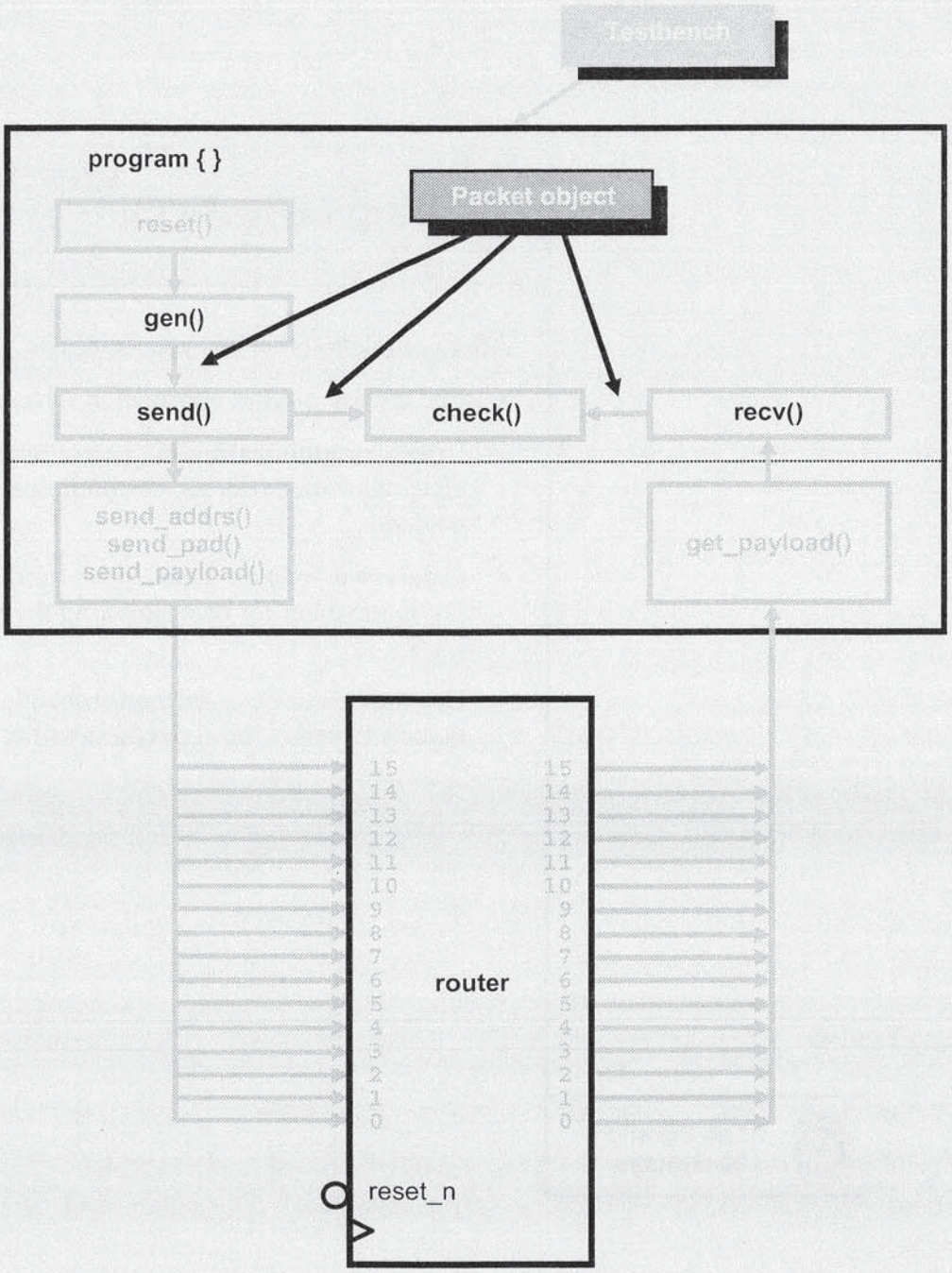


Figure 1. Lab 4 testbench architecture

Lab Overview

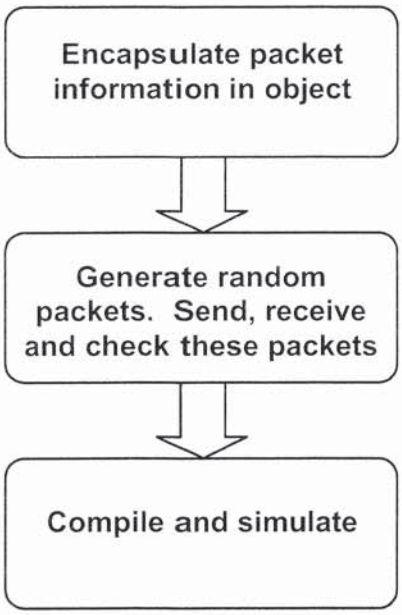


Figure 2. Diagram of Lab Exercise

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

Working with Objects

Task 1. Copy Files from Lab 3's Solutions directory

1. Go into the lab4 directory.

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

2. Copy the source files in the `solutions/lab3` directory into the current directory with `make script`.

```
> make copy
```

(If you chose to use your own lab files from lab3, type "**`make mycopy`**".)

Task 2. 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.

Task 3. Define Packet Property Constraints

1. Immediately after the property declarations, add a constraints block to limit **sa** and **da** to **0:15**, and the **payload.size()** to **2:4**.

Task 4. Define Packet Class Method Prototypes

1. 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
```

Task 5. 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 of the class body, create the constructor `new()` method. Make sure to reference the method back to the class via the `Packet::` 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
```

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

Task 6. 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 of **`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`**.

Task 7. 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.

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

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

2. 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, WARNING, DEBUG, etc.

   Or, you can just use the display routine saved in the **`.display`** file in solutions directory (`` `endif `` must be after the inserted code):

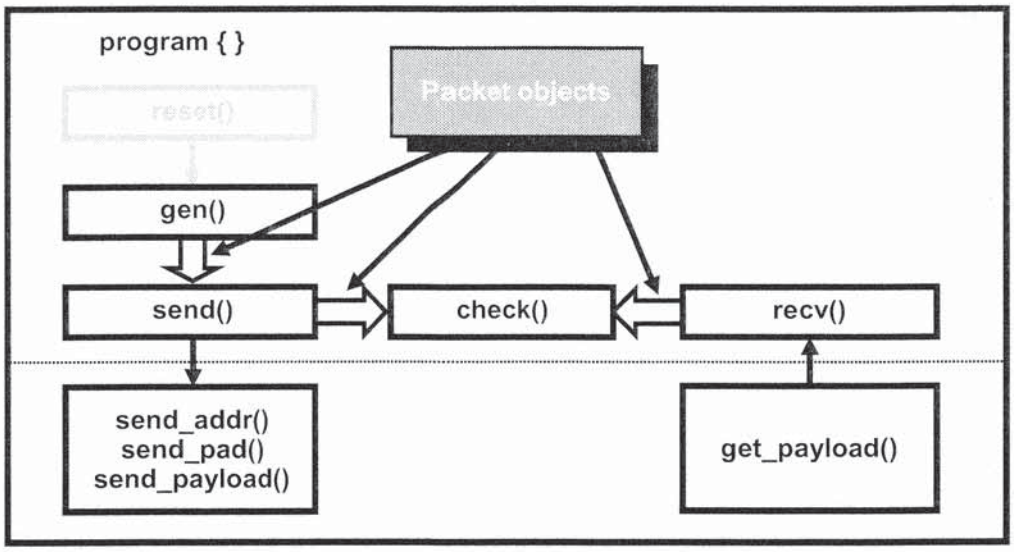
```
> cat ../../solutions/lab4/.display >> Packet.sv
```

3. Save and close the file.

Task 8. Modify `test.sv` to use Packet class

The Packet class now encapsulates the router packet information.

You will now generate random Packet objects, then send and receive packets through the router based on these random Packet objects.



The following steps take you through this process.

1. Open `test.sv` in an editor.
2. Inside the program block, add an include statement for the **`Packet`** class file.
3. **Create** and **construct** two program global **`Packet`** objects **`pkt2send`** and **`pkt2cmp`**.

Task 9. Modify `gen()` Task To Generate Packet objects

You will now use the `Packet` objects just created to generate random stimulus.

1. In the **`gen()`** task, delete all existing code.
2. Declare a static int **`pkts_generated`** variable (initialize to 0) to keep track of how many packets were generated by the generator
3. Set the **`name`** property of **`pkt2send`** to a unique string (use the **`pkts_generated`** variable value as part of the string)
4. Randomize the `Packet` object **`pkt2send`** (print an error message to terminal and end simulation if randomization fails)
5. Update all program global variables, **`sa`**, **`da`** and **`payload`** with values from the randomized **`pkt2send`** object.

This makes the content of **pkt2send** object visible to all components of the program. These program global variables are only an interim solution to simplify development of subroutines in the program block. In the next lab, these variables will all be deleted and migrated inside individual testbench component objects.

When done, your code might look something like the following:

```
task gen();
  static int pkts_generated = 0;
  pkt2send.name = $sformatf("Packet[%0d]", pkts_generated++);
  if (!pkt2send.randomize()) begin
    $display("\n%m\n[ERROR]%t Randomization Failed!", $realtime);
    $finish;
  end
  sa = pkt2send.sa;
  da = pkt2send.da;
  payload = pkt2send.payload;
endtask: gen
```

Task 10. Modify `recv()` task

In the **recv()** task, the payload sampled from the output of the router needs to be assembled into a Packet object (**pkt2cmp**). This Packet object will then be used in the checking process against the **pkt2send** object.

In **recv()** task:

Before calling the `get_payload()` ***task do the following:***

1. Create a static int variable **pkt_cnt** to track the number of packets received (should be initialized to 0)

After the `get_payload()` ***task do the following:***

2. Assign **pkt2cmp.da** with the value of program global variable **da**.
3. Assign **pkt2cmp.payload** with the values from **pkt2cmp_payload** array
4. Set a unique **name** for the **pkt2cmp** object. (use **pkt_cnt** value as part of the string)

When finished, the **recv()** task should look like:

```
task recv();
  static int pkt_cnt = 0;
  get_payload();
  pkt2cmp.da = da;
  pkt2cmp.payload = pkt2cmp_payload;
  pkt2cmp.name = $sformatf("rcvdPkt[%0d]", pkt_cnt++);
endtask: recv
```

Task 11. Modify the `check()` task

In the **check()** task, you will use the **compare()** method built into the `Packet` object to verify the content of the `Packet` object sent and received. You should use the **display()** method in the `Packet` object to assist in debugging errors.

1. Replace the **compare()** call with a call to the **compare()** method within the `Packet` object. (The two objects you want to compare are the program global objects **pkt2send** and **pkt2cmp**)
2. Make use of the **display()** method when an error is detected.

When finished, the **check()** routine should look like:

```
task check();
  string message;
  static int pkts_checked = 0;
  if (!pkt2send.compare(pkt2cmp, message)) begin
    $display("\n%m\n[ERROR]%t Packet#%0d %s\n", $realtime, pkts_checked, message);
    pkt2send.display("ERROR");
    pkt2cmp.display("ERROR");
    $finish;
  end
  $display("[NOTE]%t Packet#%0d %s", $realtime, pkts_checked++, message);
endtask: check
```

Task 12. Check and Save file

1. Make sure you have deleted the `compare()` routine in `test.sv`.
2. Save and close the file.

Task 13. 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, execute the following script which runs the testbench on a bad RTL code.

   ```
   > make bad
   ```

   If the simulation finds an error, you are done.

   If the simulation does not find an error, you have a problem with your testbench. You must debug the error in your testbench.

Congratulations, you completed Lab 4!

test.sv Solution:

```
program automatic test(router_io.TB router);
  // The following program variables will be seen by the included files without extern
  int run_for_n_packets;      // number of packets to test

  `include "Packet.sv"

  // The following program variables can be seen by the included files with extern
  bit[3:0] sa;              // source address
  bit[3:0] da;              // destination address
  logic[7:0] payload[$];          // expected packet data array
  logic[7:0] pkt2cmp_payload[$];     // actual packet data array
  Packet   pkt2send = new();      // expected Packet object
  Packet   pkt2cmp = new();       // actual Packet object

  initial begin
    $vcdpluson;
    run_for_n_packets = 2000;
    reset();
    repeat(run_for_n_packets) begin
      gen();
      fork
        send();
        recv();
      join
      check();
    end
    repeat(10) @(router.cb);
  end

  task reset();
    router.reset_n <= 1'b0;
    router.cb.frame_n <= '1;
    router.cb.valid_n <= '1;
    ##2 router.cb.reset_n <= 1'b1;
    repeat(15) @(router.cb);
  endtask: reset

  task gen();
    static int pkts_generated = 0;
    pkt2send.name = $sformatf("Packet[%0d]", pkts_generated++);
    if (!pkt2send.randomize()) begin
      $display("\n%m\n[ERROR]%t gen(): Randomization Failed!", $realtime);
      $finish;
    end
    sa = pkt2send.sa;
    da = pkt2send.da;
    payload = pkt2send.payload;
  endtask: gen

  task send();
    send_addrs();
    send_pad();
    send_payload();
  endtask: send

  task send_addrs();
    router.cb.frame_n[sa] <= 1'b0;
    for(int i=0; i<4; i++) begin
```

```
    router.cb.din[sa] <= da[i];
    @(router.cb);
  end
endtask: send_addrs

task send_pad();
  router.cb.frame_n[sa] <= 1'b0;
  router.cb.valid_n[sa] <= 1'b1;
  router.cb.din[sa] <= 1'b1;
  repeat(5) @(router.cb);
endtask: send_pad

task send_payload();
  foreach(payload[index]) begin
    for(int i=0; i<8; i++) begin
      router.cb.din[sa] <= payload[index][i];
      router.cb.valid_n[sa] <= 1'b0;
      router.cb.frame_n[sa] <= (index == (payload.size() - 1)) && (i == 7);
      @(router.cb);
    end
  end
  router.cb.valid_n[sa] <= 1'b1;
endtask: send_payload

task recv();
  static int pkt_cnt = 0;
  get_payload();
  pkt2cmp.da = da;
  pkt2cmp.payload = pkt2cmp_payload;
  pkt2cmp.name = $sformatf("rcvdPkt[%0d]", pkt_cnt++);
endtask: recv

task get_payload();
  pkt2cmp_payload.delete();
  fork
    begin: wd_timer_fork
    fork: frameo_wd_timer
      @(negedge router.cb.frameo_n[da]);
      begin
        repeat(1000) @(router.cb);
        $display("\n%m\n[ERROR]%t Frame signal timed out!\n", $realtime);
        $finish;
      end
    join_any: frameo_wd_timer
    disable fork;
    end: wd_timer_fork
  join
  forever begin
    logic[7:0] datum;
    for (int i=0; i<8; ) begin
      if (!router.cb.valido_n[da])
        datum[i++] = router.cb.dout[da];
      if (router.cb.frameo_n[da])
        if (i == 8) begin
          pkt2cmp_payload.push_back(datum);
          return;
        end
        else begin
          $display("\n%m\n[ERROR]%t Packet payload not byte aligned!\n", $realtime);
          $finish;
        end
```

```
      @(router.cb);
    end
    pkt2cmp_payload.push_back(datum);
  end
endtask: get_payload

task check();
  string message;
  static int pkts_checked = 0;
  if (!pkt2send.compare(pkt2cmp, message)) begin
    $display("\n%m\n[ERROR]%t Packet #%0d %s\n", $realtime, pkts_checked, message);
    pkt2send.display("ERROR");
    pkt2cmp.display("ERROR");
    $finish;
  end
  $display("[NOTE]%t Packet #%0d %s", $realtime, pkts_checked++, message);
endtask: check

endprogram: test
```

Packet.sv Solution:

```
`ifndef INC_PACKET_SV
`define INC_PACKET_SV
class Packet;
  rand bit[3:0] sa, da;  // random port selection
  rand logic[7:0] payload[$];    // random payload array
       string   name;    // unique identifier

  constraint Limit {
    sa inside {[0:15]};
    da inside {[0:15]};
    payload.size() inside {[2:4]};
  }

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

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

function bit Packet::compare(Packet pkt2cmp, ref string message);
  if (payload.size() != pkt2cmp.payload.size()) begin
    message = "Payload Size Mismatch:\n";
    message = { message, $sformatf("payload.size() = %0d, pkt2cmp.payload.size() =
%0d\n", payload.size(), pkt2cmp.payload.size()) };
    return(0);
  end
  if (payload == pkt2cmp.payload) ;
  else begin
    message = "Payload Content Mismatch:\n";
    message = { message, $sformatf("Packet Sent:  %p\nPkt Received: %p", payload,
pkt2cmp.payload) };
    return(0);
  end
  message = "Successfully Compared";
  return(1);
endfunction: compare

function void Packet::display(string prefix);
  $display("[%s]%t %s sa = %0d, da = %0d", prefix, $realtime, name, sa, da);
  foreach(payload[i])
    $display("[%s]%t %s payload[%0d] = %0d", prefix, $realtime, name, i,
payload[i]);
endfunction: display

`endif
```

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
- 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. 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 **`mailbox`** class as the communication mechanism.

The resulting architecture is shown below:

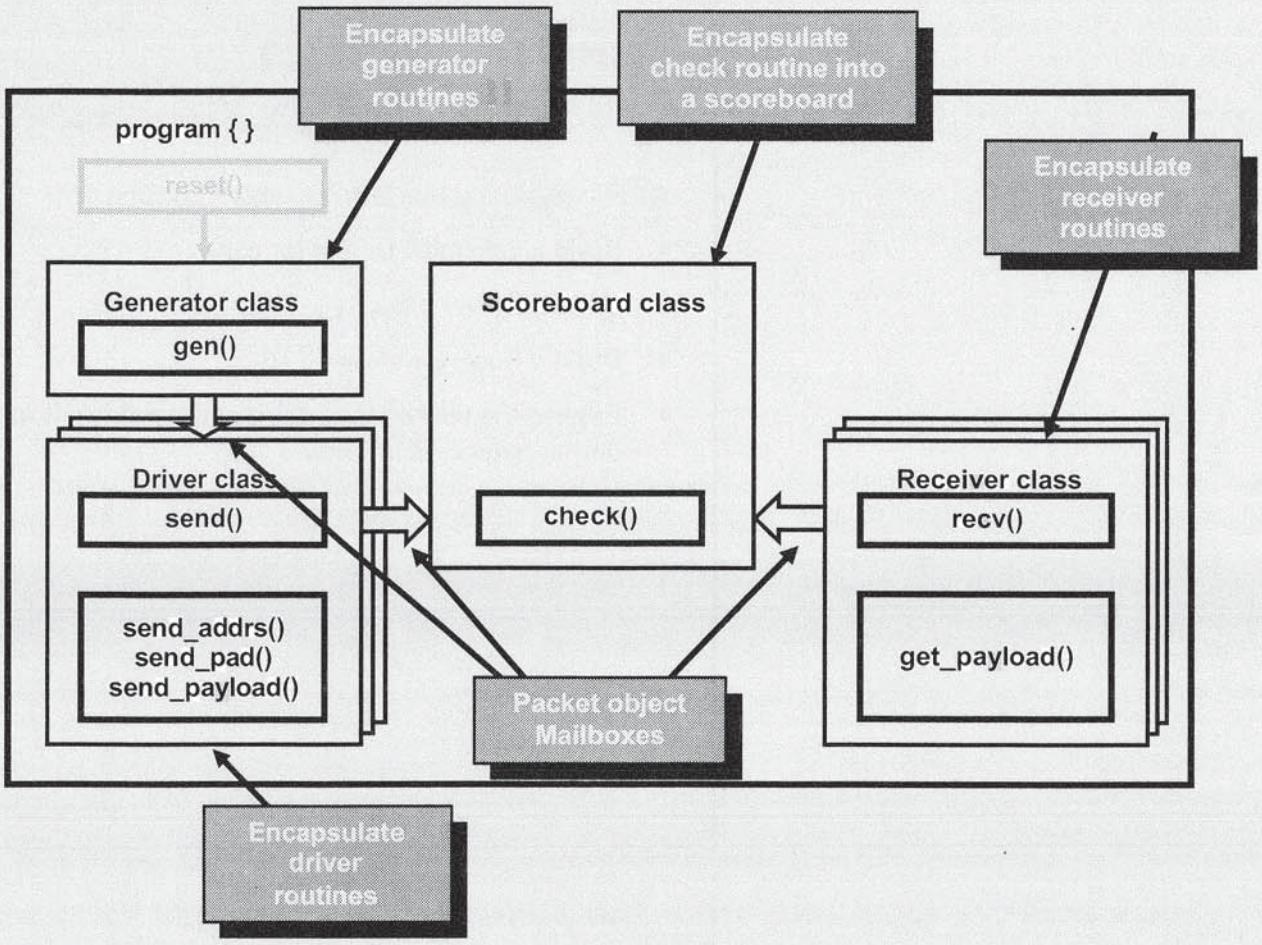


Figure 1. Lab 5 Encapsulate transactors for broad-spectrum verification

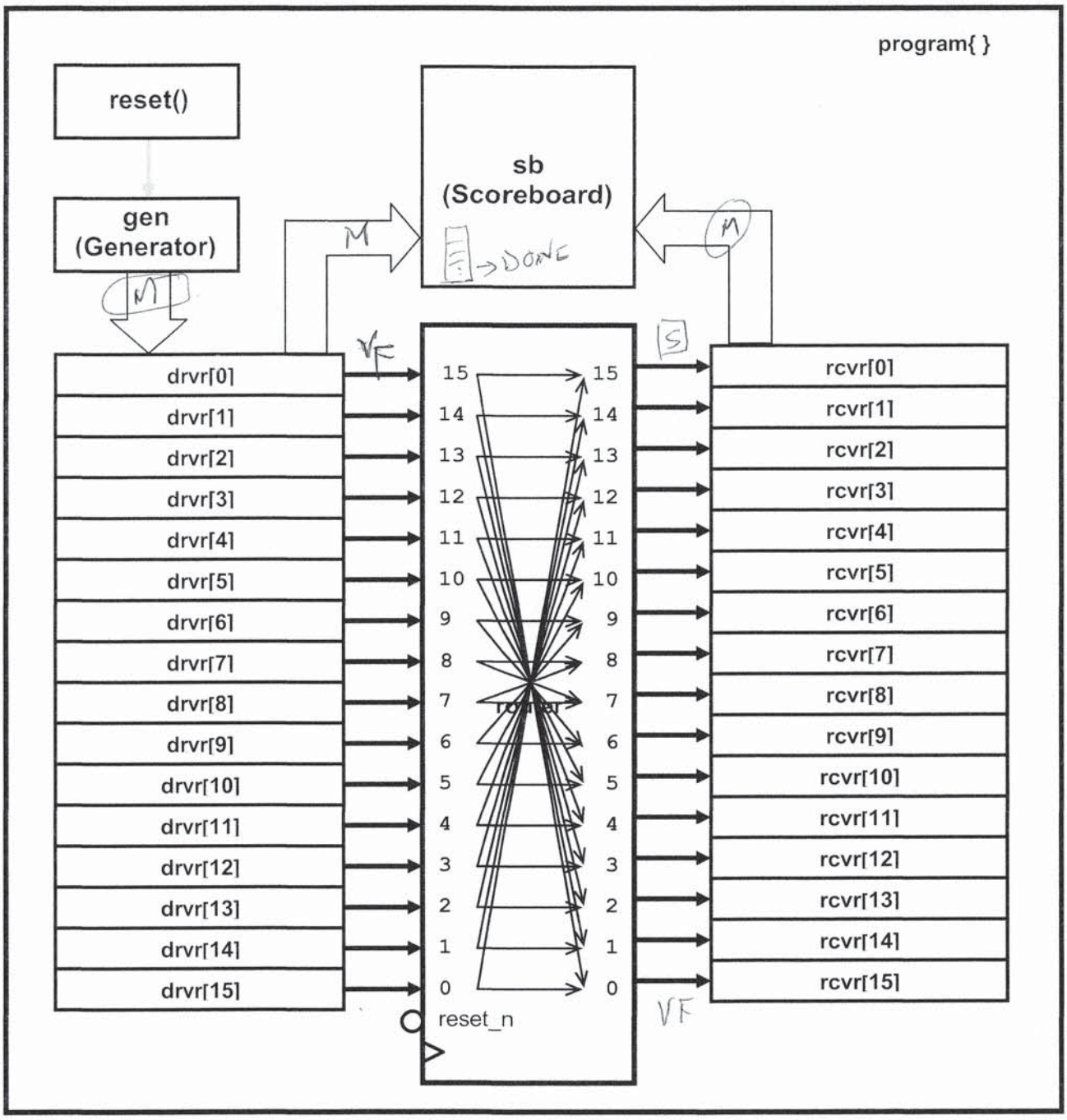


Figure 2. Lab 5 testbench architecture

Lab Overview

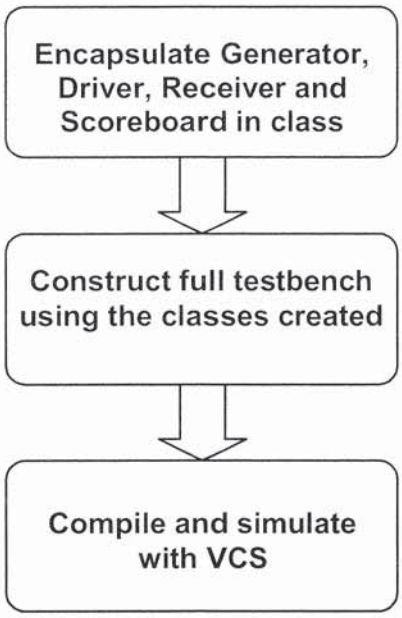


Figure 3. Diagram of Lab Exercise

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.

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-$display() combination at the beginning of the subroutine. It is often helpful during debugging to be able to see a printout of the subroutine execution sequences. This if-$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.h 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 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
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 `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

The `start()` method will execute a non-blocking infinite loop. In each iteration of the infinite loop, reconstuct a `Packet` object (`pkt2cmp`) 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 processthread.
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`.
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.

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.

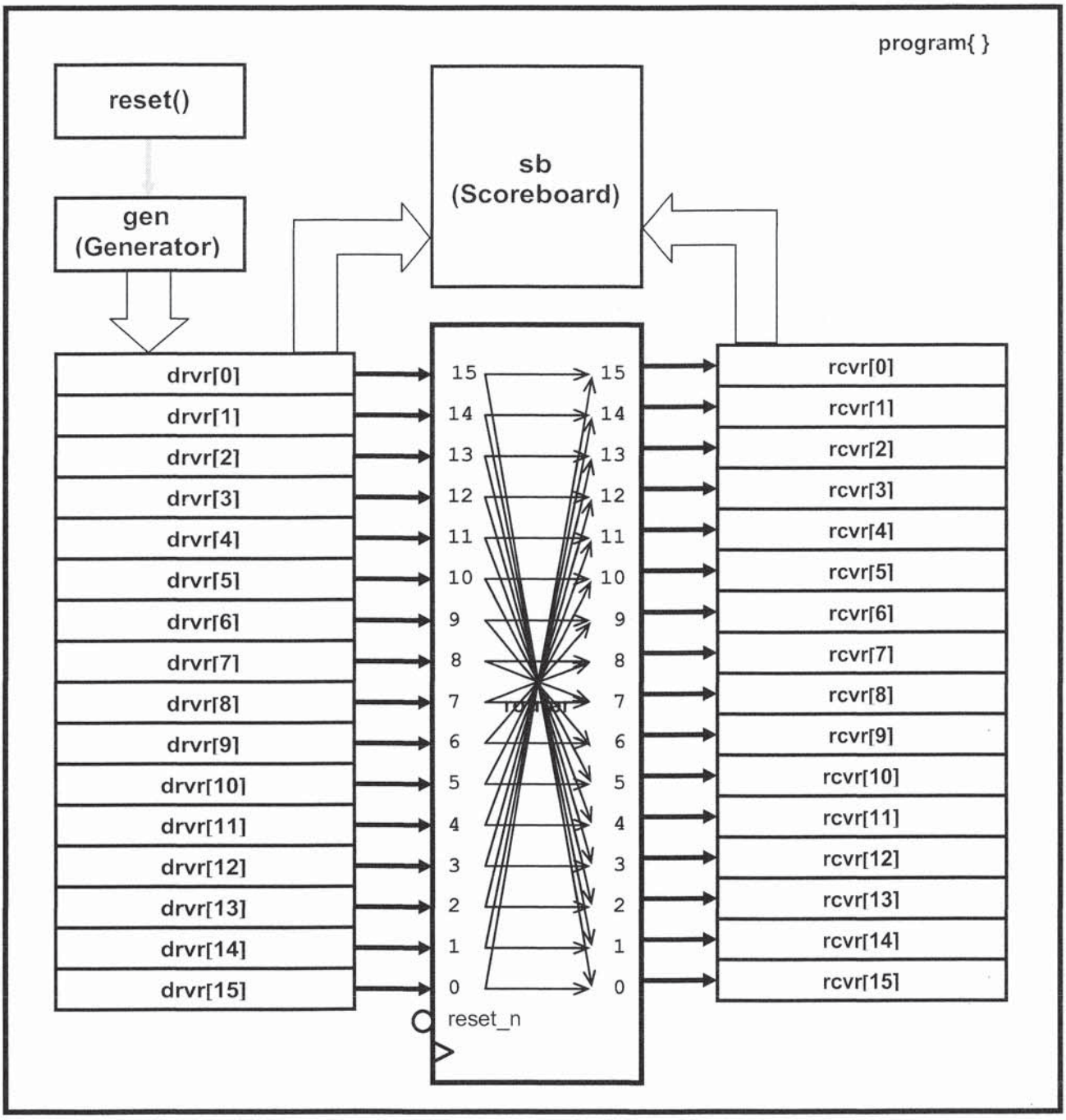
When the Scoreboard finds a `Packet` object in the `receiver_mbox`, it will first save this object handle as `pkt2cmp`. Then, it will push all `Packet` objects found in the `driver_mbox` onto a `refPkt[$]` queue. Afterwards, on the basis of the output port address (`da`) in `pkt2cmp` object, it will try to locate the corresponding reference `Packet` in `refPkt[$]` 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 one generator, 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 object
   - `Scoreboard sb;` // scoreboard object
6. Delete all content of the initial block except for two:

```
initial begin
  $vcdpluson;
  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: Generator to all Drivers (gen.out_box[i]); 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 (gen, sb, drivers and receivers)
10. Finally, before the end of the program, block until sb'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 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
```

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!

test.sv Solution:

```
program automatic test(router_io.TB rtr_io);
  // The 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
    $vcdpluson;
    run_for_n_packets = 2000;
    sem = new[16];
    drvr = new[16];
    rcvr = new[16];
    gen = new();
    sb = new();
    foreach (sem[i])
      sem[i] = new(1);
    for (int i=0; i<drvr.size(); i++)
      drvr[i] = new($sformatf("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($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
```

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, 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, 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 = new this.pkt2cmp;
        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

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

function Generator::new(string name);
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, name);
  this.name = name;
  this.pkt2send = new();
  this.out_box = new[16];
  foreach(this.out_box[i])
    this.out_box[i] = new();
endfunction: new

task Generator::gen();
  static int pkts_generated = 0;
  if (TRACE_ON) $display("[TRACE]%t %s:%m", $realtime, this.name);
  this.pkt2send.name = $sformatf("Packet[%0d]", pkts_generated++);
  if (!this.pkt2send.randomize()) 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
    for (int i=0; i<run_for_n_packets || run_for_n_packets <= 0; i++)
begin
      this.gen();
      begin
        Packet pkt = new this.pkt2send;
        this.out_box[pkt.sa].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 task 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
```

```
task 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;
endtask: check
`endif
```

This page was intentionally left blank.

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.

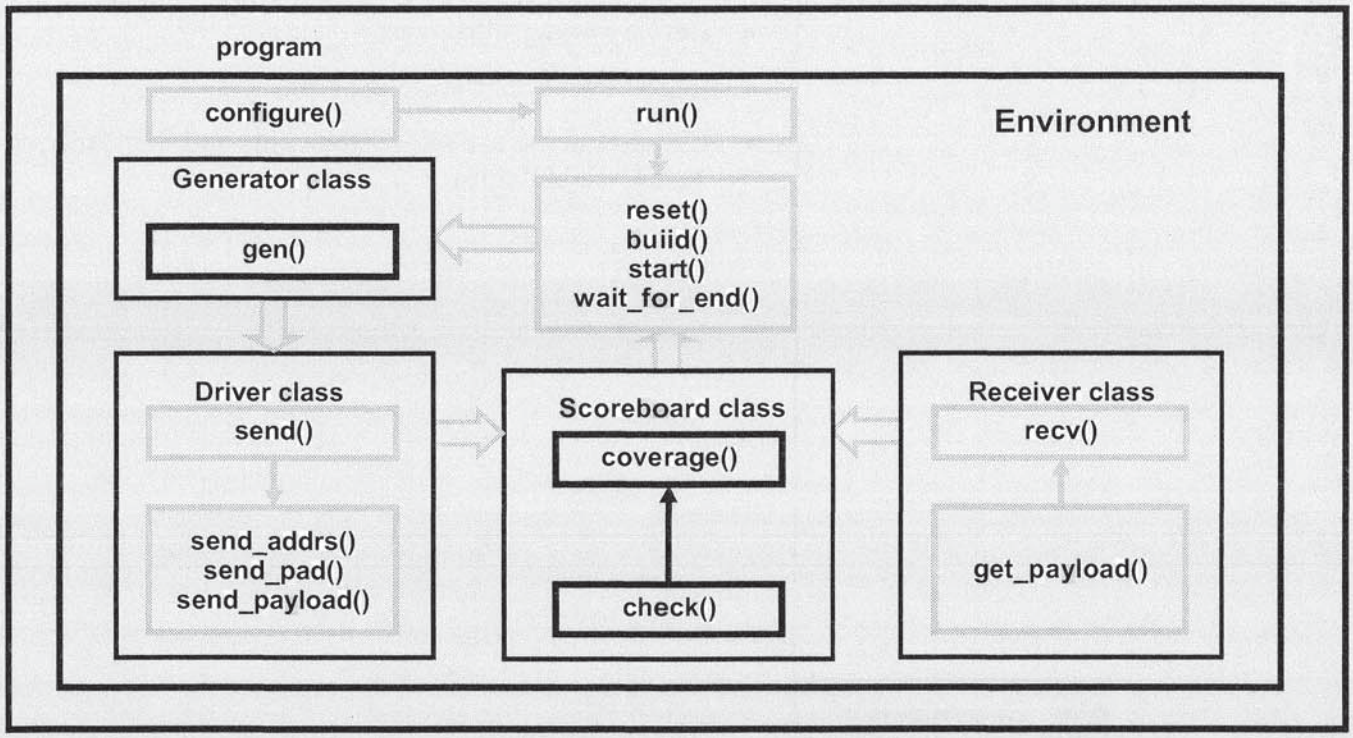


Figure 1. Lab 6 Standardized Environments and Functional Coverage

Lab Overview

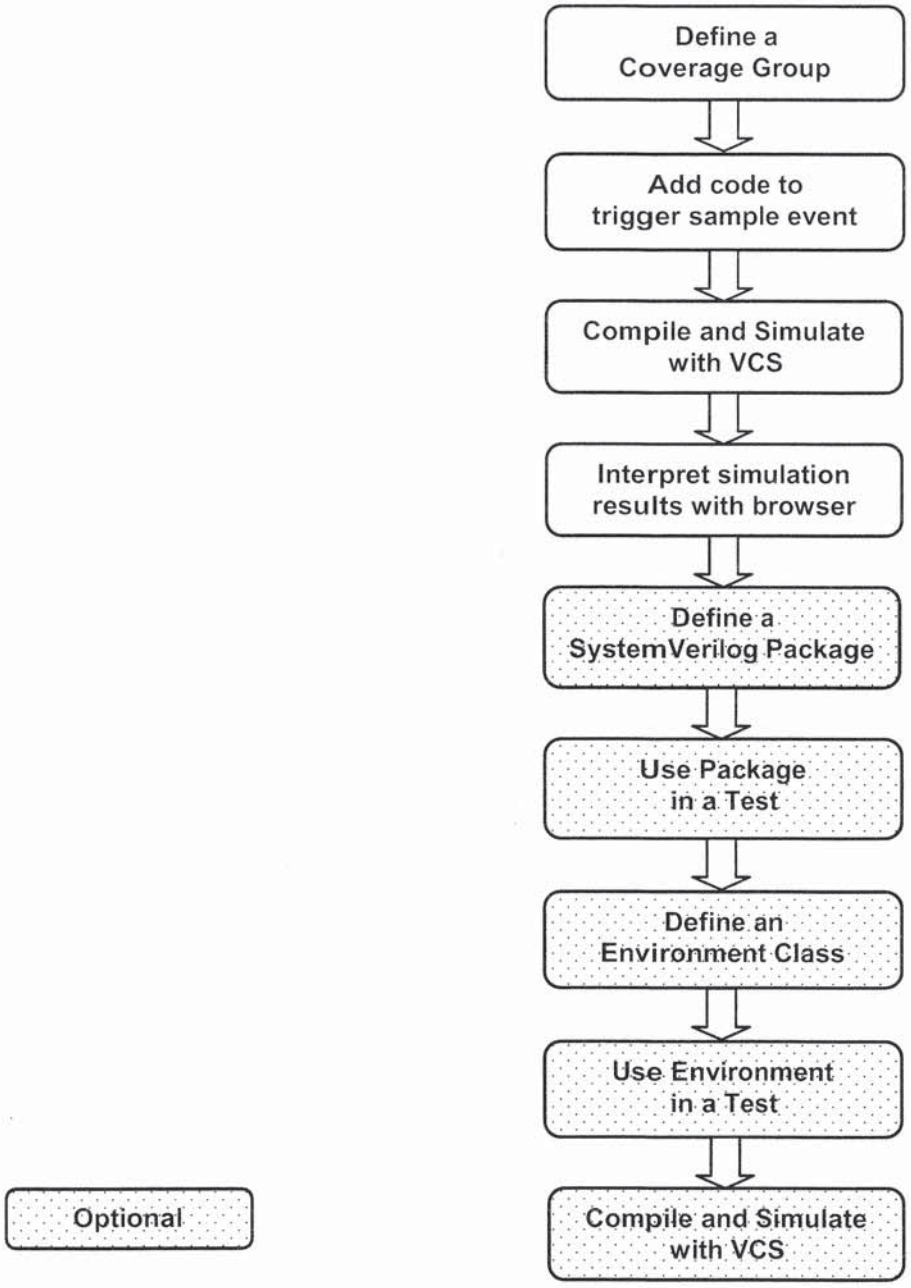


Figure 2. Diagram of **Lab** 6 Exercise

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

Functional Coverage

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

1. Go into the lab6 directory.

   > `cd ../lab6`

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 group. Within the coverage group, coverage bins, bin update event timing and coverage goal will be defined.

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`**

   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()`** task

1. In the `check()` method, 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, trigger the functional coverage bin update with `router_cov.sample()` call

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. 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 $cwd/urgReport/dashboard.html &
   ```

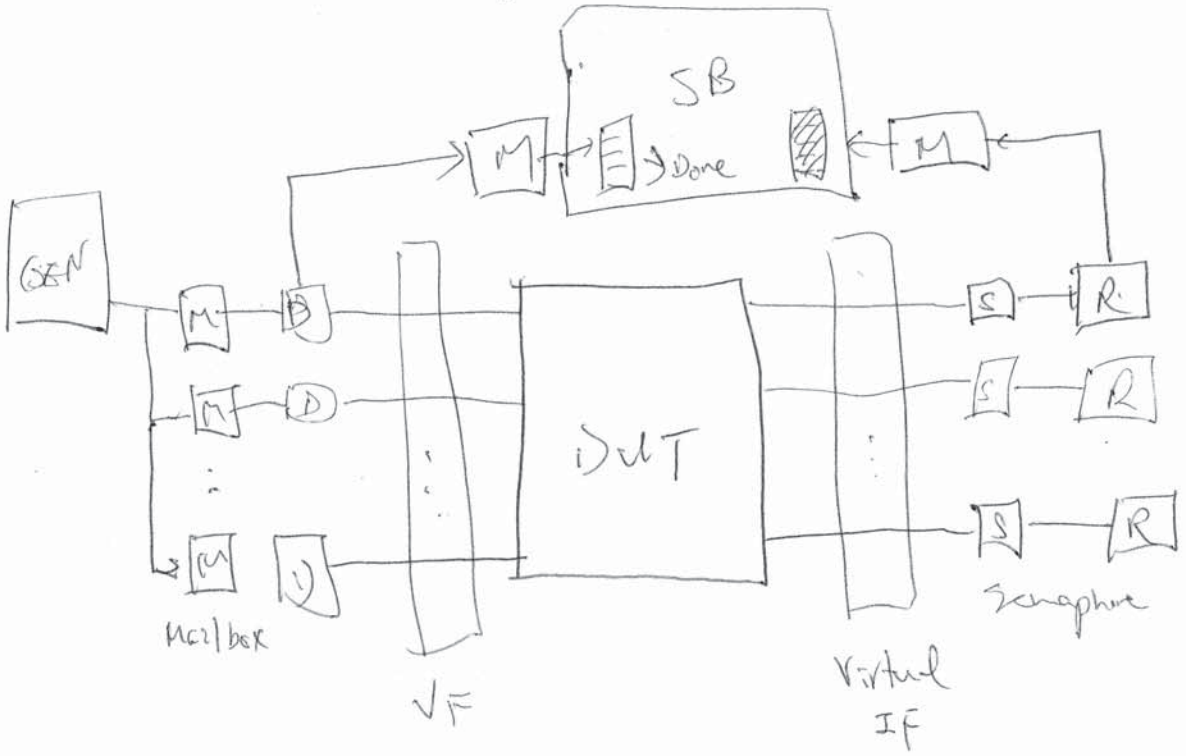
   Select the groups link and follow the links for the **`router_cov`** covergroup.

   To view the report in text format run

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

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

The next part of the lab is optional. Proceed if you wish to learn about packages and environments. If not, congratulations, you have completed all labs successfully!



Task 6. 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
    $vcdpluson;
    run_for_n_packets = 2000;
    sem = new[16];
    drvr = new[16];
    rcvr = new[16];
    gen = new();
    sb = new();
    foreach (sem[i])
      sem[i] = new(1);
    for (int i=0; i<drvr.size(); i++)
      drvr[i] = new($psprintf("drvr[%0d]", 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), .,
                    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

endprogram
```

Build the Environment (construct components)

Reset DUT

Start the components

Wait for end-of-test

2. Save and 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" ``
8. Complete the package definition.
9. Save and close the file.

Task 7. 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.
4. Save and close the file.

Task 8. Compile and Run

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

```
> make package_run
```

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 9. 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. 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 components. By creating random configurations you can test many more configurations and test conditions than you can with directed tests. The components 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()`** and **`run()`**.
4. The **`configure()`** method is used to set the configuration of the test. Here it simply randomizes the **`run_for_n_packets`** variable.
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.

```
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 object
  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", $realtime, this.name);
  if(this.run_for_n_packets == 0) this.run_for_n_packets = 2000;
    this.sem = new[16];
    this.drvr = new[16];
    this.rcvr = new[16];
    this.gen = new();
    this.sb = new();
    foreach (sem[i])
      this.sem[i] = new(1);
    for (int i=0; i<drvr.size(); i++)
      this.drvr[i] = new($psprintf("drvr[%0d]", i), i, this.sem,
          this.gen.out_box[i], this.sb.driver_mbox, this.rtr_io);
    for (int i=0; i<rcvr.size(); i++)
      this.rcvr[i] = new($psprintf("rcvr[%0d]", i), i,
          this.sb.receiver_mbox, this.rtr_io);
endfunction: build

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;
  ##2 this.rtr_io.cb.reset_n <= 1'b1;
  repeat(15) @(this.rtr_io.cb);
endtask: reset

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

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
```

Build the Environment (construct components)

Reset DUT

Start the components

Wait for end-of-test

Task 10. Use Environment in the Test Program

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

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.
- 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 except the **$vcdpluson** line.
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_pkts** (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
    $vcdpluson;
    env = new("env", rtr_io);
    env.configure();
    run_for_n_packets = env.run_for_n_packets;
    env.run();
end
endprogram: test
```

Task 11. Compile and Run

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

```
> make package_run
```

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

.....

.....

2. Change the seed to see how it affects coverage results.

```
> make package_run SEED=<seed_value>
```

Task 11. 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 in the test by you. 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. Hence the number of packets needed to reach 100% coverage was also different. This is a good example of how changing test environment structure may affect the constraint solver.

router_test_pkg.sv Solution:

```
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:

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

Environment env;

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

endprogram: test
```

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
  bit[3:0] sa, da;        // functional coverage properties
  int run_for_n_packets; //how many packets

  covergroup router_cov;
    coverpoint sa { type_option.weight = 0; }
    coverpoint da { type_option.weight = 0; }
    cross sa, da;
  endgroup

  extern function new(string name = "Scoreboard", pkt_mbox driver_mbox
=null, receiver_mbox = null);
  extern virtual task start();
  extern virtual task 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
task Scoreboard::start();
  if (TRACE_ON) $display("[TRACE]%0t %s:%m", $time, name);
  fork
    while (1) 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
```

```
task 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;
endtask
`endif
```