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 driver signals with DVE or Verdi3

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.

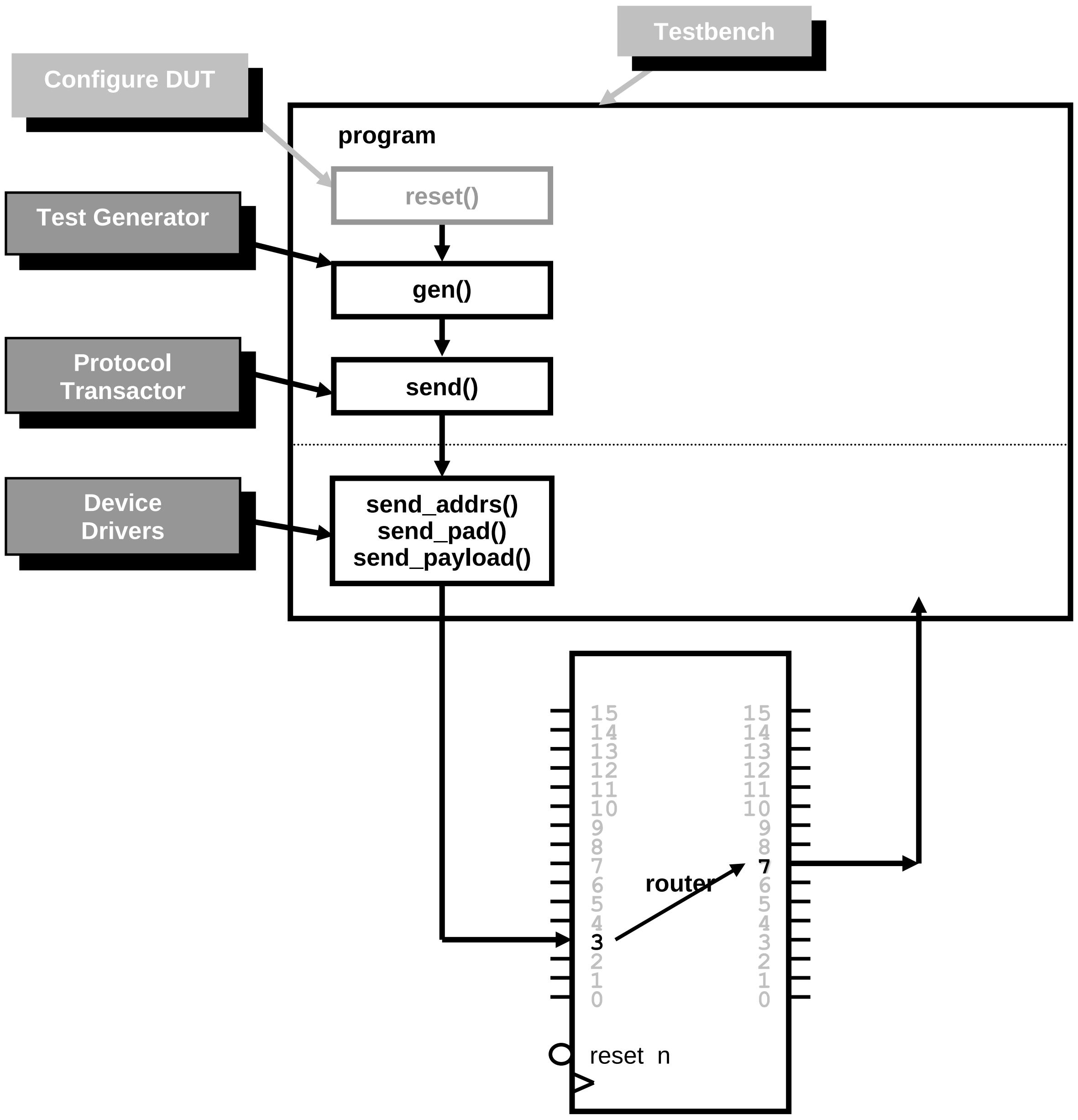


Figure 1. Lab 2 testbench architecture

Lab Overview

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

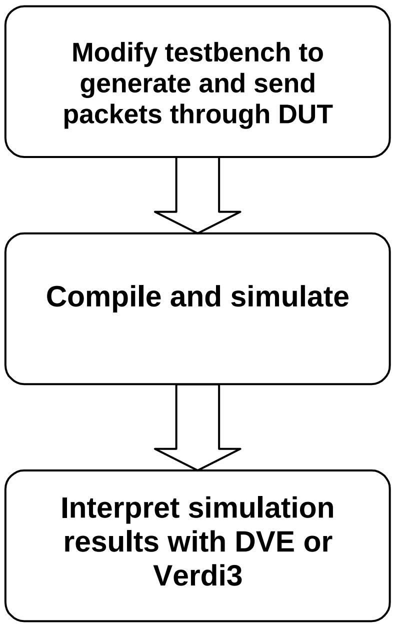

Modify testbench to generate and send packets through DUT
Compile and simulate
Interpret simulation results with DVE or Verdi3

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 global variables.

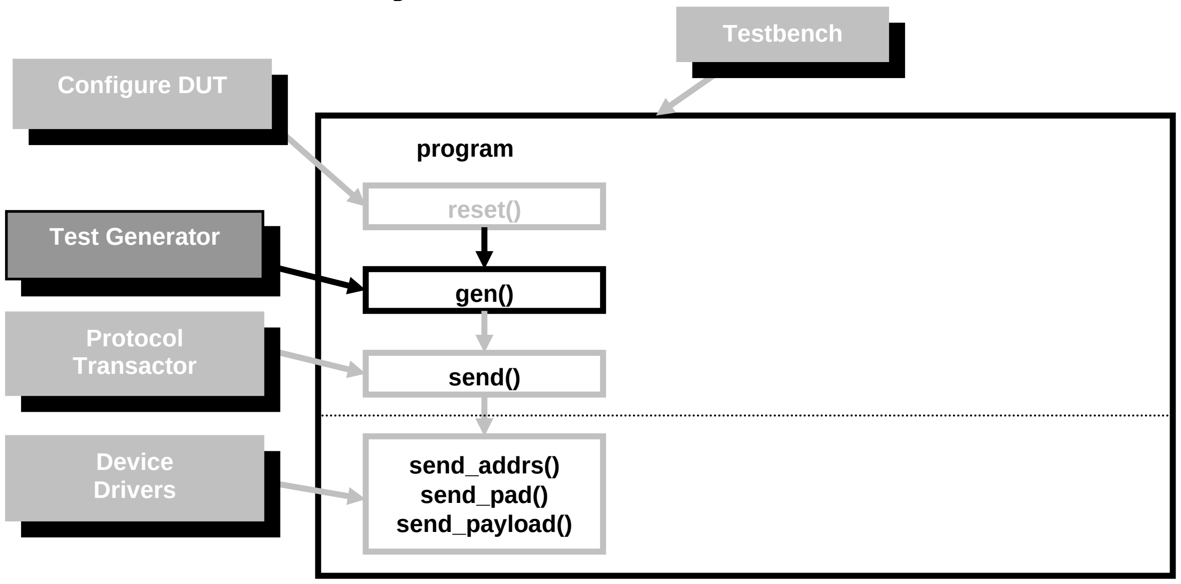
1. Open **`test.sv`** with text editor.

2. Declare the **program global variables** 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()** subroutine. 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
    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($urandom_range(4,2))
    payload.push_back($urandom);
endtask: gen
```

Task 4. Create Send Packet Routines

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

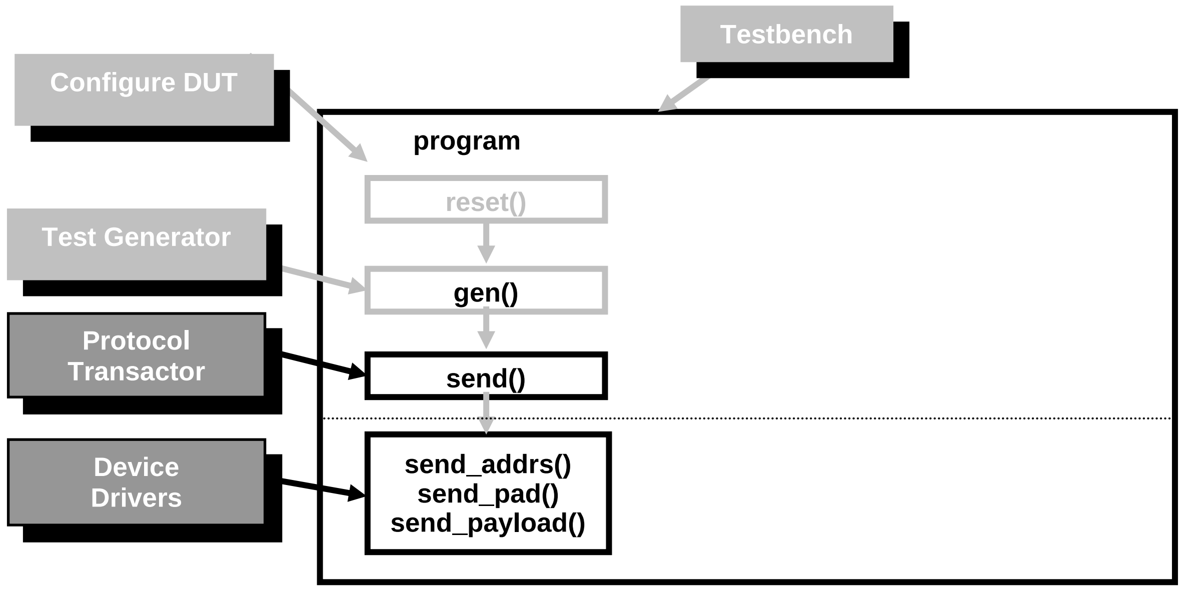
Sending a packet through the router is done using three processes(waveform on next page):

Send the:

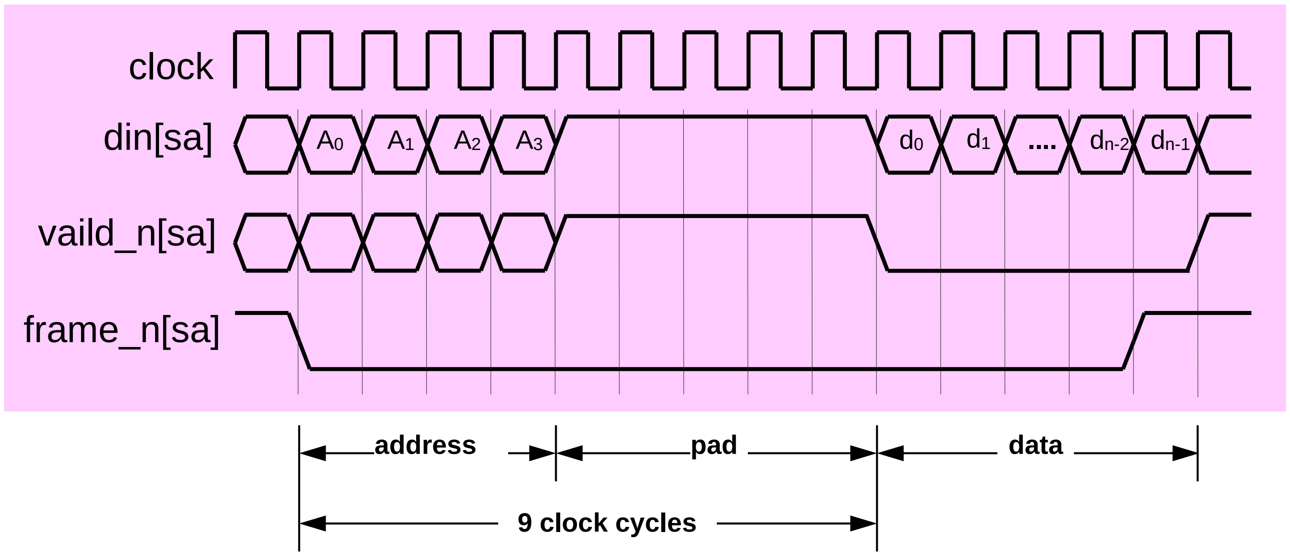
- Destination Address
- Padding bits
- Payload

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. Verdi3 automatically extends the last set of signal waveforms to enhance viewability.
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. 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 for each element of **`payload`**.
- 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 or Verdi3.

   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.

   To open DVE Automated Debug use

   ```
   > make dve
   ```

   To open Verdi Automated Debug use

   ```
   > make verdi
   ```

Task 6. Extend The Program To Send 21 Packets

1. Modify the program block to send 21 packets through the router using the same values for sa and da of 3 and 7 respectively.
2. Compile, simulate & verify using DVE or Verdi3.

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

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

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

Continued…

```
  …Continued from previous page
  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
```