

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) or Verdi Automated Debug Platform



Lab
Duration:

Lab 1

Getting Started

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

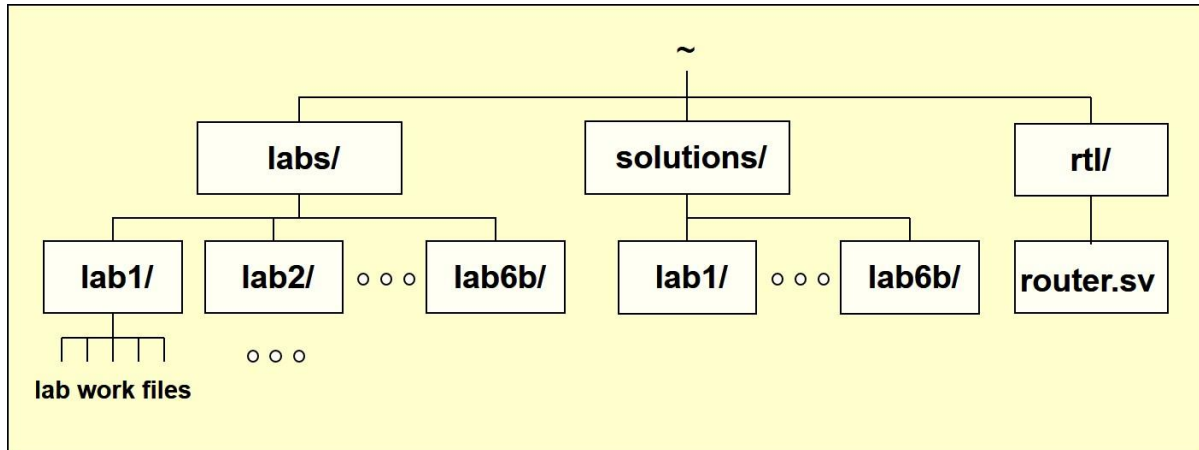


Figure 1. Lab Directory Structure

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

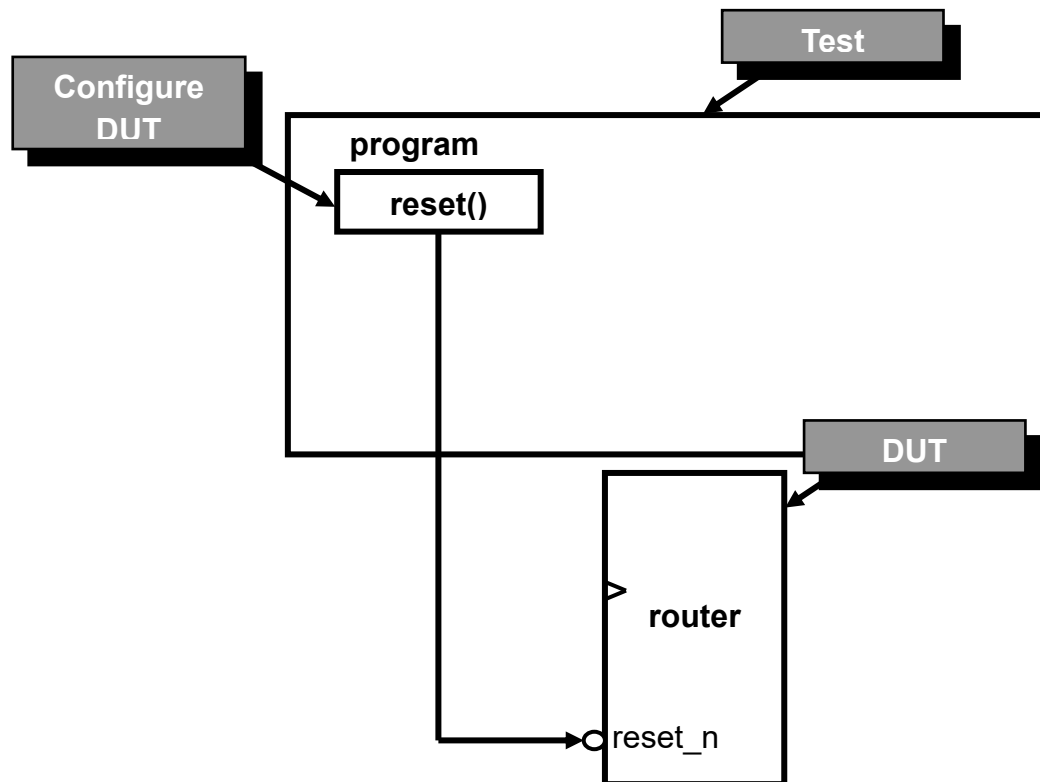


Figure 2. Testbench Architecture

Lab Overview

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

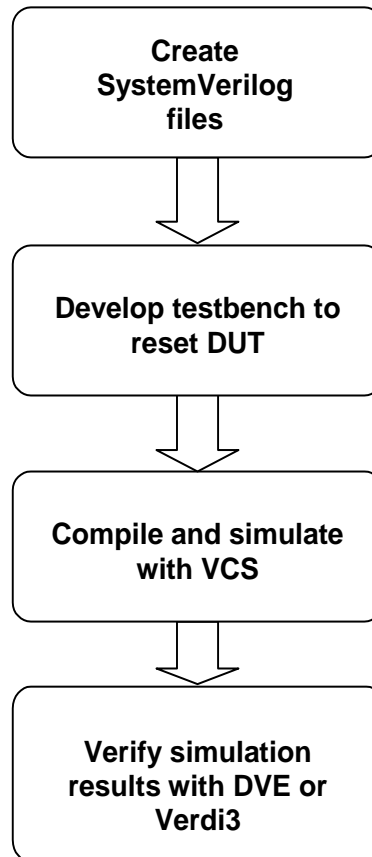
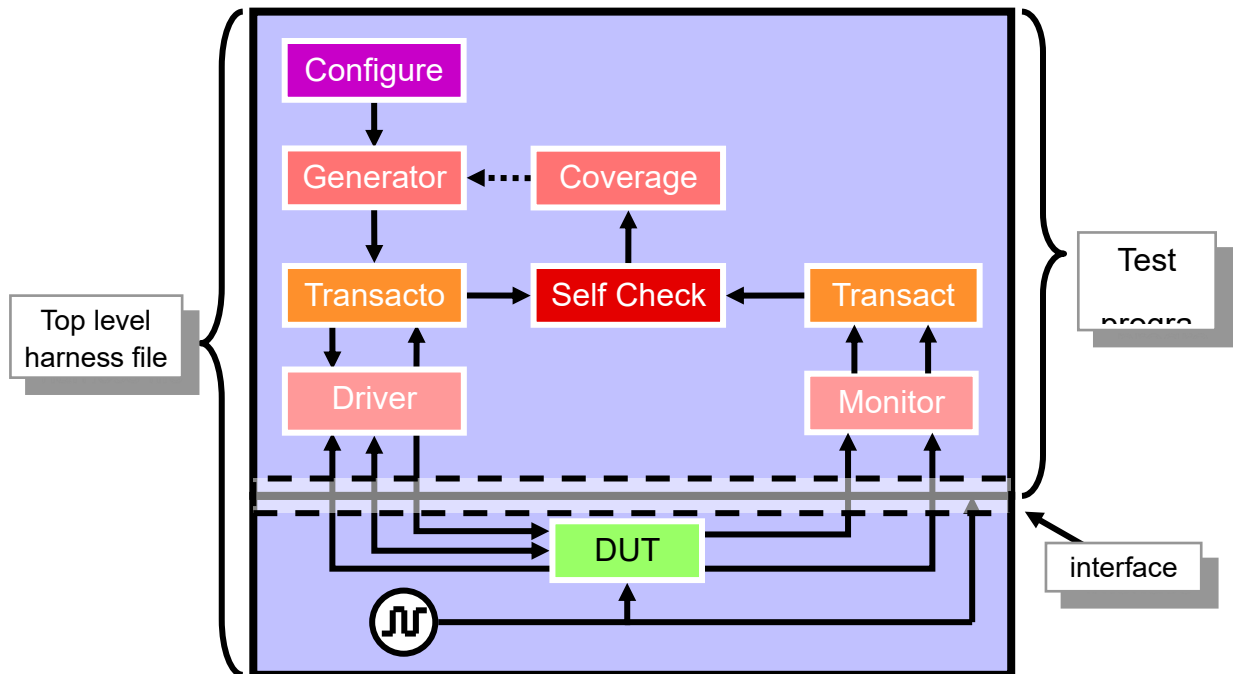


Figure 3. Lab 1 Flow Diagram

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

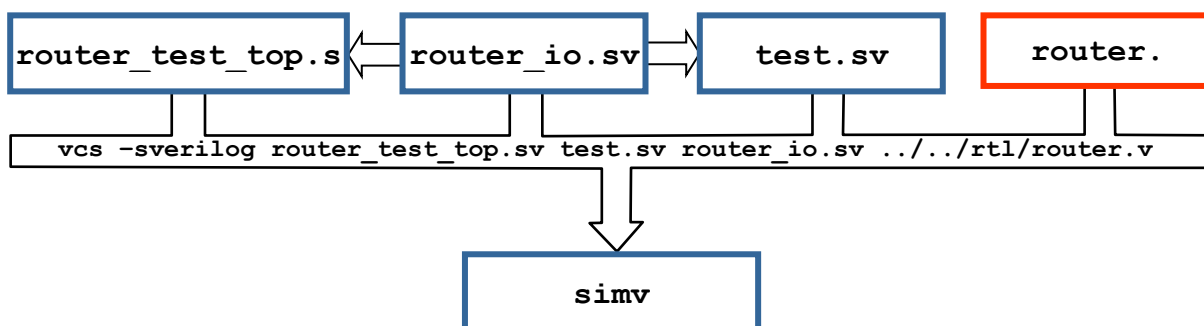
Constructing SystemVerilog Testbench

A typical architecture of a SystemVerilog testbench looks like the figure below:



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 your workstation if needed (use the login and password provided by the instructor).

2. Go to the **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 a 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 signals for the test program to drive and sample the DUT signals. This is done with **clocking** block declarations.

Lab 1

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 directions specific to test

```
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 (output recommended):

```
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 asynchronous 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 = 0;
    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)
    );
    ...
    always begin
        #(simulation_cycle/2) SystemClock =
~SystemClock;
    end
endmodule
```

3. Add an interface instance to the harness:

```
module router_test_top;
    parameter simulation_cycle = 100;
    bit  SystemClock = 0;
    router_io top_io(SystemClock);
    router dut( ... );
    ...
    always begin
        #(simulation_cycle/2) SystemClock =
~SystemClock;
    end
endmodule
```

Instantiate

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

```
module router_test_top;
  parameter simulation_cycle = 100;
  bit SystemClock = 0;
  router_io top_io(SystemClock); // interface
instance
  test t(top_io); // program instance
  router dut( ... );
  ...
  always begin
    #(simulation_cycle/2) SystemClock =
    ~SystemClock;
  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
- ``timescale`
 - `$timeformat` (in initial block)

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

Set time (%t)
to ns scale

7. Save and close the file

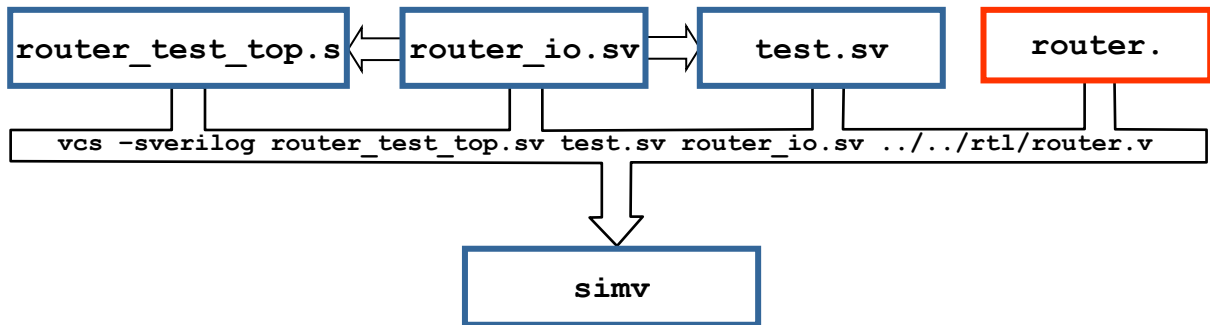
Lab 1

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:

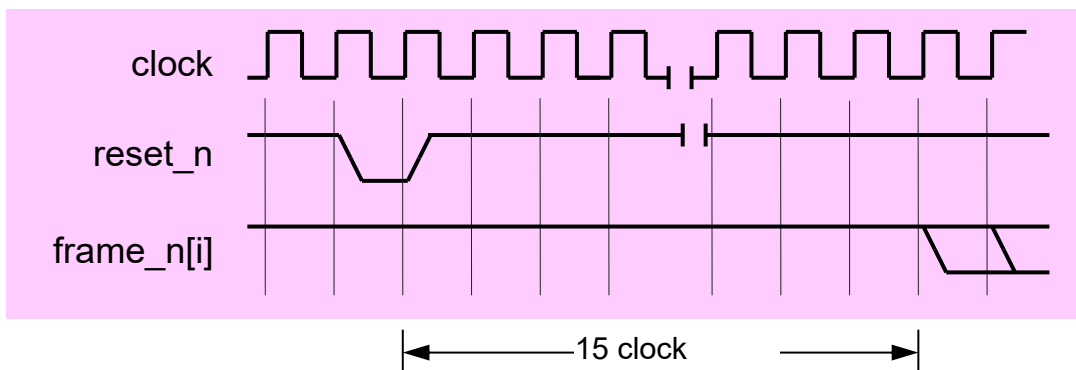
```
> simv
```

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 add needed functionality to the testbench. In this task, you will add a routine to reset 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 a call to **reset()**

When completed, your program may look something like:

```
program automatic test(router_io.TB rtr_io);
  initial begin
    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
    repeat(2) @(rtr_io.cb);         // synchronous
    rtr_io.cb.reset_n <= 1'b1;      // synchronous
    repeat(15) @(rtr_io.cb);        // synchronous
  endtask: reset
endprogram: test
```

4. 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. If you wish to use the Verdi debugging environment, add the instrumentation for Verdi in `router_test_top.sv` **initial** block. You can use Verdi as detailed later in Task 9. This requires the **-kdb** and **-lca** switches. It creates the dump file “**novas.fsd**” which is used by Verdi waveform viewer.

```
initial begin
    $fsdbDumpvars;    // Only for Verdi debugging
```

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

```
> vcs -sverilog -debug_access+all +vcs+vcdpluson router_test_top.sv \
test.sv router_io.sv ../../rtl/router.v -kdb -lca
```

The above compile added usage of some more switches:

-debug_access+all, +vcs+vcdpluson, -kdb, -lca

These first two switches will create a dump file “**vcdplus.vpd**” which stores the activities of signals during simulation. This file is used by VCS Discovery Visual Environment (DVE) to examine the DUT signals in a waveform window.

Lab 1

3. 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 Testbench Lab1
LAB_DIR = lab1
RTL= ../../rtl/router.v
SVTB = ./router test top.sv ./router io.sv ./test.sv
seed = 1
run_opts =
comp_opts =

default: test

test: compile run

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

compile:
    vcs -l vcs.log -sverilog -kdb -debug access+all $(SVTB) $(RTL)
    +vcs+vcdpluson -lca -full64 $(comp_opts)

verdi:
    verdi -ssf novas.fsdb -undockWin -nologo &

dve:
    dve -vpd vcdplus.vpd &

verdi_debug:
    ./simv -l simv.log -gui=verdi +ntb_random_seed=$(seed) $(run_opts)

debug:
    ./simv -l simv.log -gui=dve +ntb_random_seed=$(seed) $(run_opts)

solution: clean
    cp -f ../../solutions/$(LAB_DIR)/*.sv .

clean:
    rm -rf simv* csrc* *.tmp *.vpd *.key *.log *hdrs.h *.fsdb verdiLog
    elabcomLog novas.* *.dat

original: nuke
    cp -f ../../solutions/$(LAB_DIR)/router io.sv.orig router io.sv
    cp -f ../../solutions/$(LAB_DIR)/router test top.sv.orig
router test top.sv
    cp -f ../../solutions/$(LAB_DIR)/test.sv.orig test.sv

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

help:

Task 8. Browse Simulation Hierachy in DVE

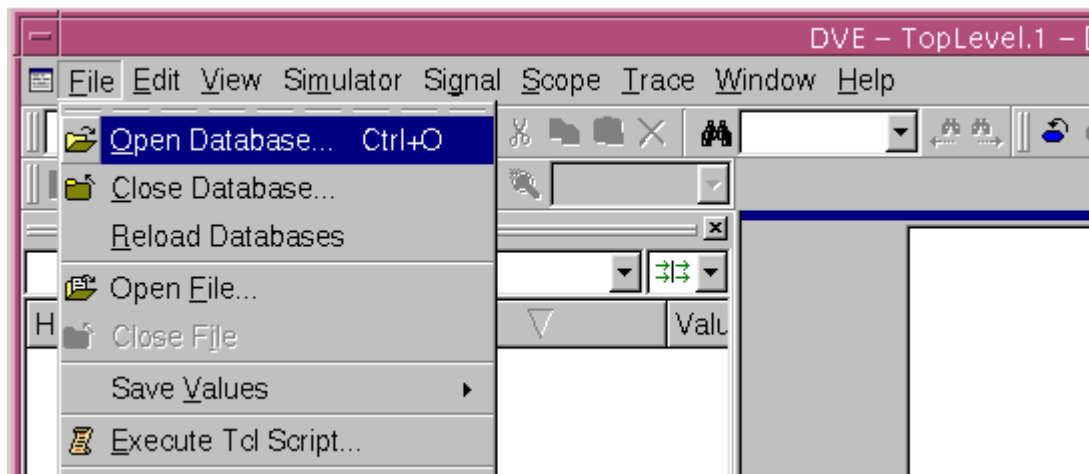
Note: If you wish to use Verdi instead of DVE for viewing waveforms, please skip to Task 9. **Do so only if you are familiar with Verdi Automated Debug Platform.**

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

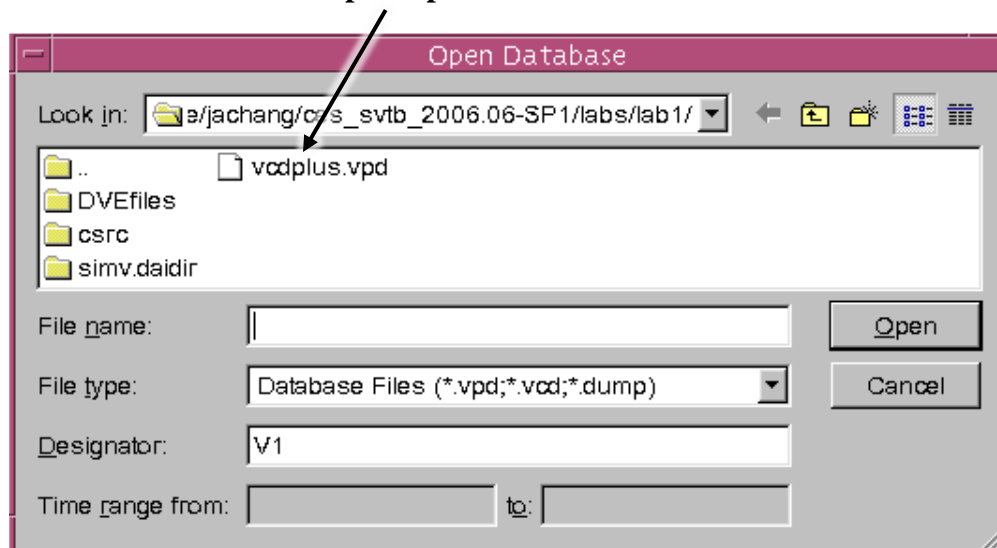
> **dve &**

The GUI may not look exactly like the images shown below.

2. Click on **File → Open Database**.




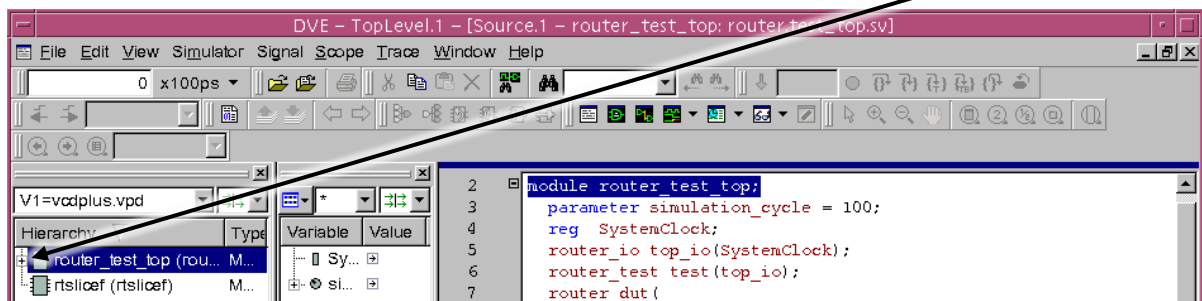
3. Double click on **vcdplus.vpd** file.



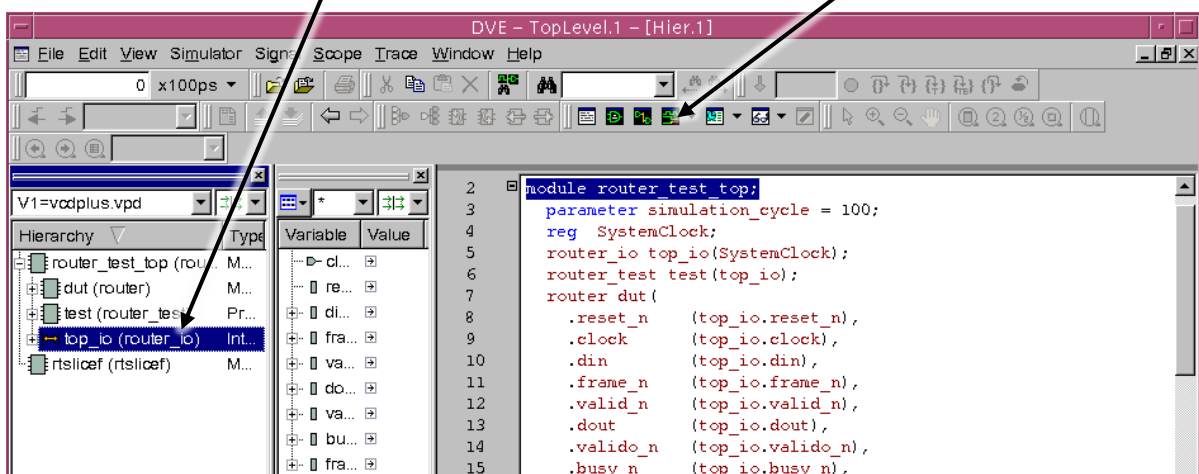
The DVE debugging windows should now be opened with the harness hierarchy and source code.

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

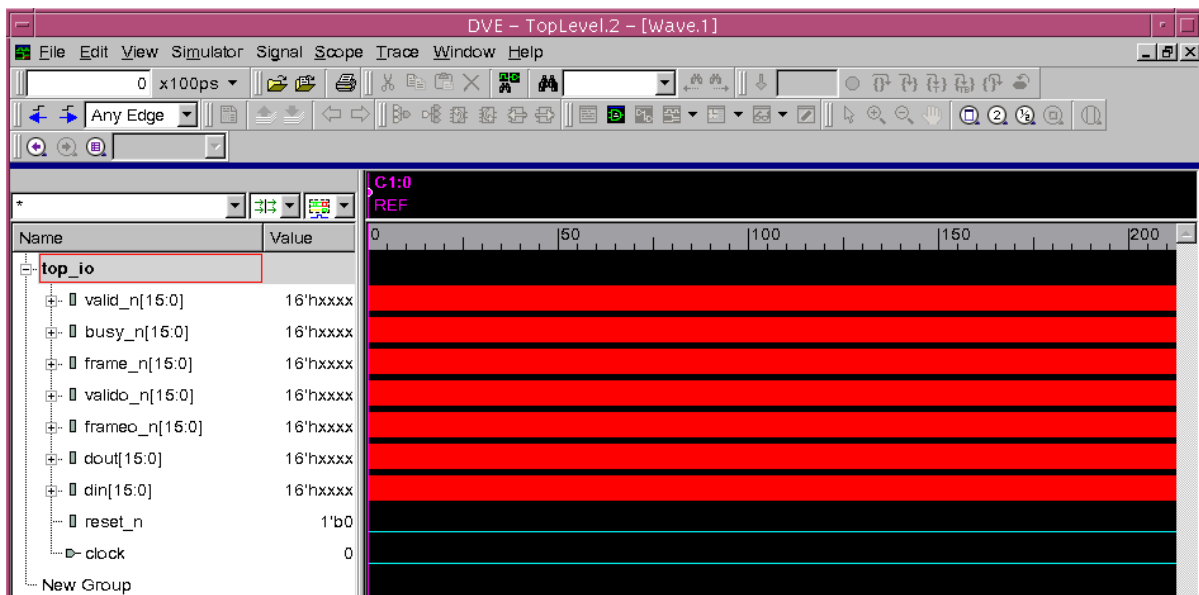
4. Expand harness to access interface signals by clicking on .



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

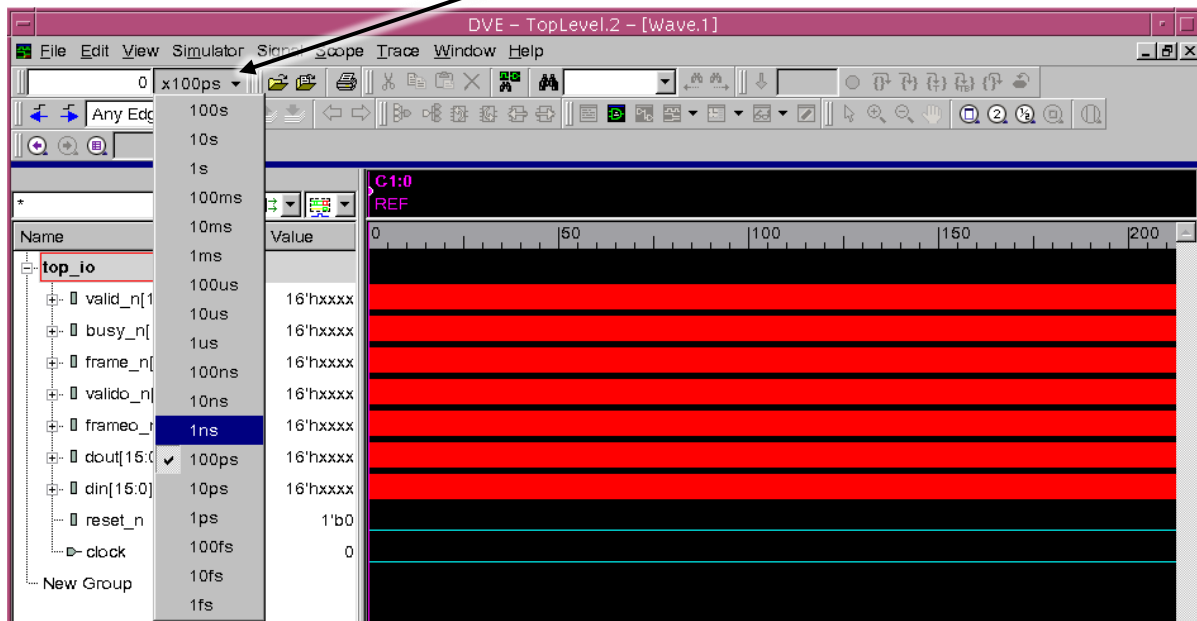



6. The **Waveform** window should open looking like the following:

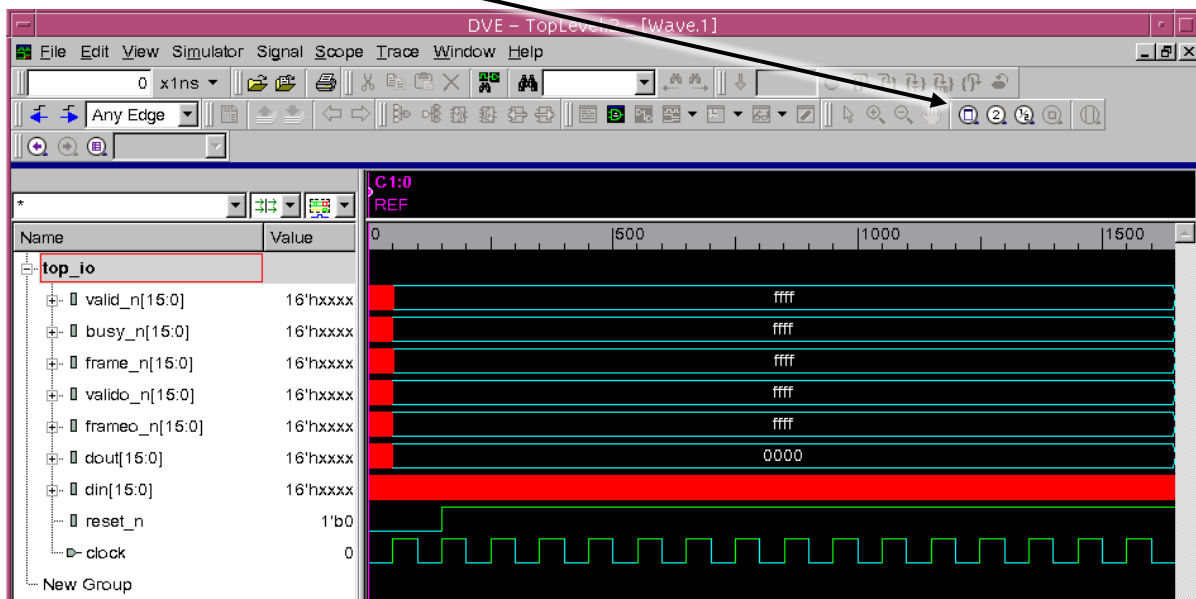


Lab 1

7. Set waveform time scale to 1ns



8. Click on  to span signal window to see through all simulation time.



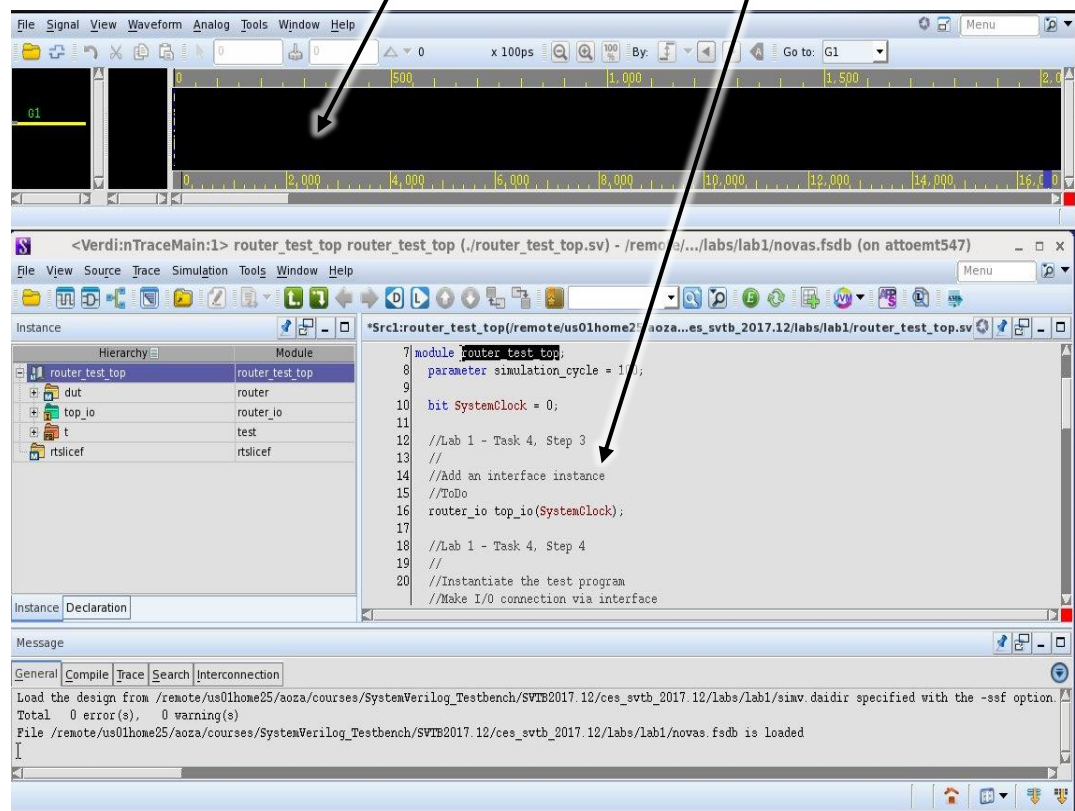
9. Verify the correct timing of the reset signal. If there are errors, correct the error then type “**make**” to re-compile and re-simulate the modified code.
10. You can save the waveform setting. In the Wave window select File -> Save Current View menu command. In the pop-up dialog box, save the session 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.

Lab 1-16

Task 9. Browse Simulation Hierarchy in Verdi



Note: Do this step only if you wish to use Verdi Automated Debug Platform waveform viewer to debug the workshop labs.

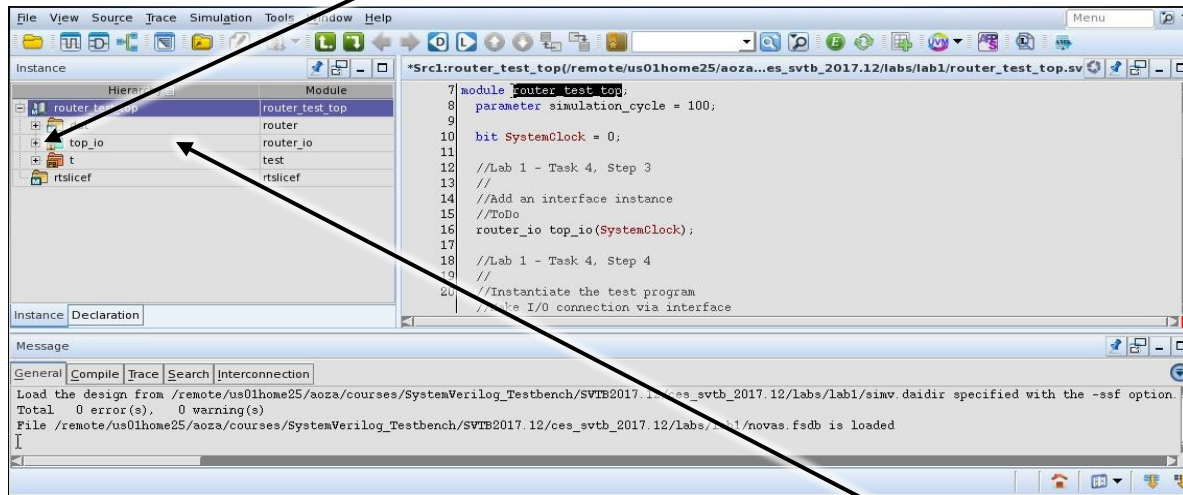
1. Open the Verdi waveform viewer. It may take a while for the GUI to appear. The **novas.fsdb** file is loaded in the waveform tool by the command
> make verdi
2. You should see two windows – the main Verdi console nTrace and the waveform window nWave - as shown below.



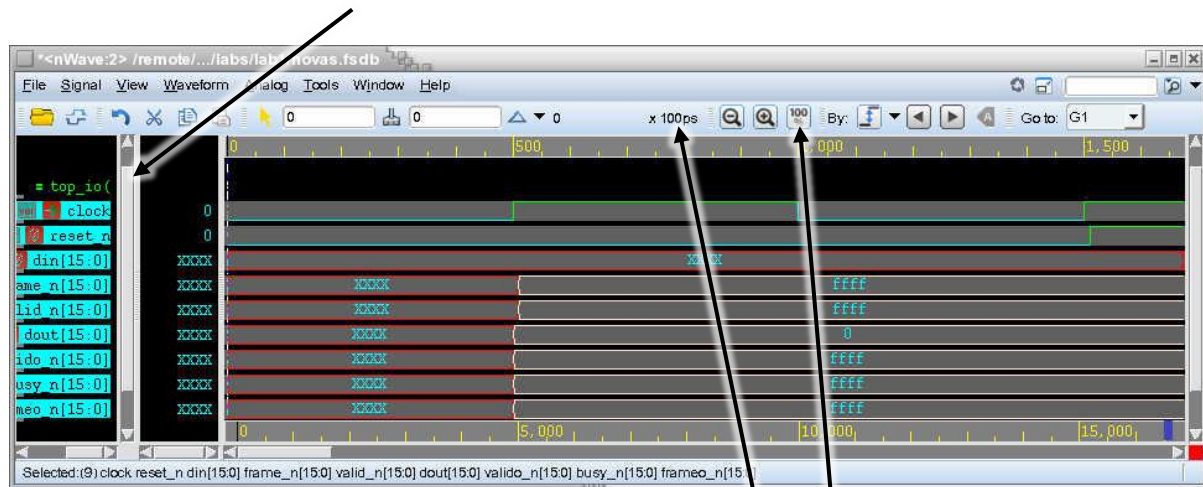
Continued...

Lab 1

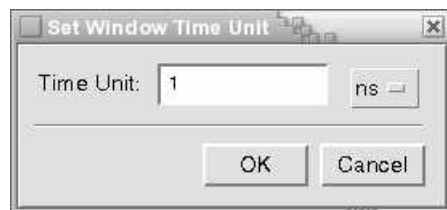
- The testbench hierarchy is shown on the top left of the nTrace window. You can use the  to expand and the  to collapse the hierarchy.




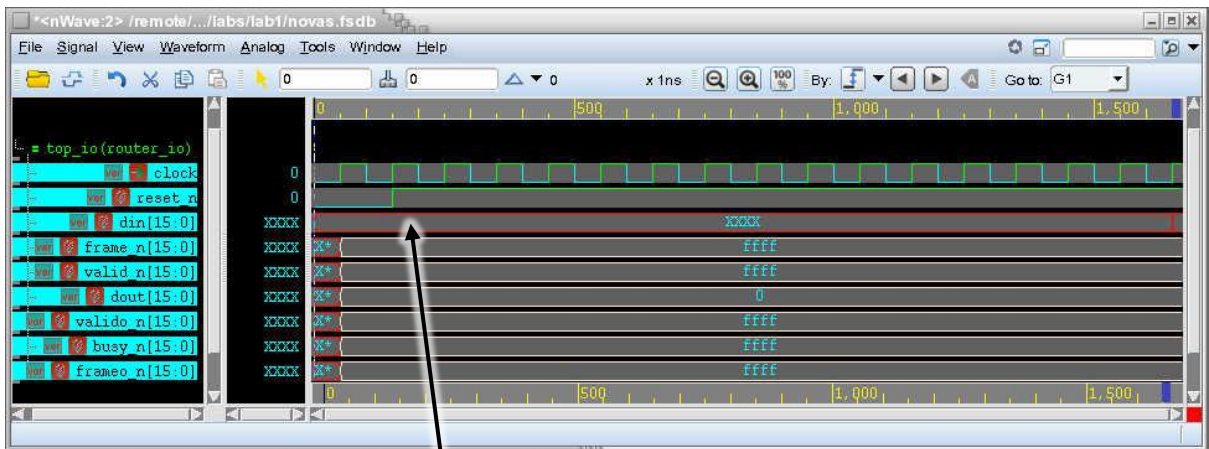
- Using the Middle Mouse Button (MMB), drag the top_io from the Instance pane to the nWave Window. This opens the interface signals in the nWave window as shown below.
- You can resize the different sections of the window as desired.



- Change the time unit to 1ns. Click on the 100 ps. A separate box pops up to let you set the time unit. Set it to 1ns as shown.



To display complete simulation time, click on the  icon.



7. Verify the correct timing of the reset signal. If there are errors, correct the error then type the correct “**make**” commands to re-compile and re-simulate the modified code. Zoom in on the area where signal **reset_n** goes inactive and check the 1 ns drive delay you coded in the clocking block.

If there are no errors, you are done with lab 1.

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 rtr_io);

    initial begin
        reset();
    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

endprogram: test
```

router test top.sv Solution:

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

    bit  SystemClock = 0;
    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);
        $fsdbdumpvars //only for Verdi debugging
    end

    always begin
        #(simulation_cycle/2) SystemClock = ~SystemClock;
    end

endmodule
```