CUSTOMER EDUCATION SERVICES

SystemVerilog Assertions Workshop

Lab Guide

50-I-053-SLG-011 2019.06

Synopsys Customer Education Services
690 E. Middlefield Road
Mountain View, California 94043

Workshop Registration: https://training.synopsys.com

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Document Order Number: 50-I-053-SLG-011
SystemVerilog Assertions Workshop Lab Guide

1 Implement Immediate and Concurrent Assertions

Learning Objectives

After completing this lab, you should be able to:

- Write an immediate assertion in your test program
- Write a concurrent assertion
- Compile and simulate the DUT with assertions
- Debug assertions with Discovery Verification Environment (DVE) and Verdi

Lab Duration:
30 minutes

Getting Started

Once logged in, you will see four directories: `rtl_lib`, `rtl`, `labs` and `solutions`.

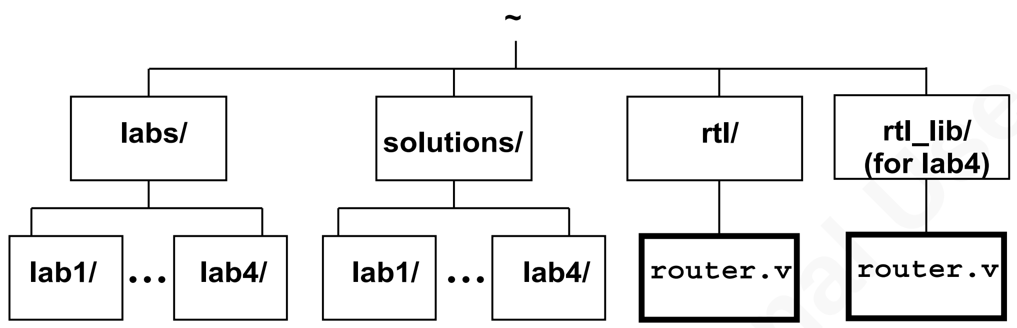


Figure 1. Lab Directory Structure

In PartA of this lab, you will implement immediate assertions in Part A and in Part B you will implement concurrent assertions.

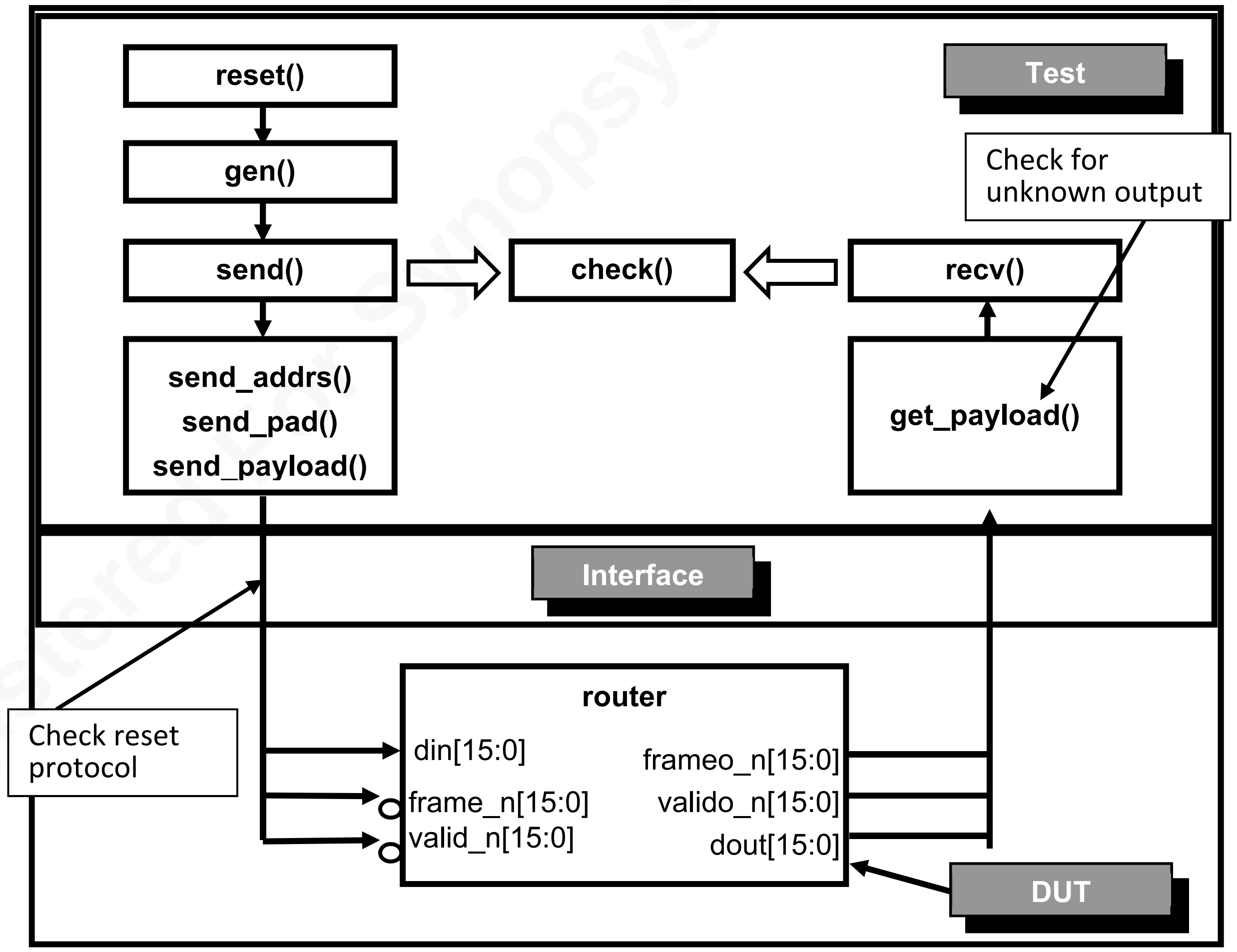
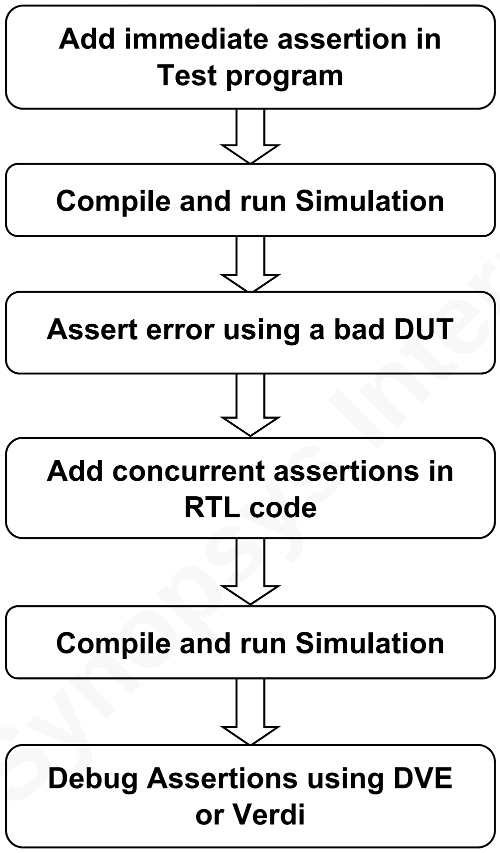


Figure 2. Lab Testbench Structure

Overview

This lab takes you through the process of developing, compiling, simulating and debugging assertions in your testbench.


Add immediate assertion in Test program
Compile and run Simulation
Assert error using a bad DUT
Add concurrent assertions in RTL code
Compile and run Simulation
Debug Assertions using DVE or Verdi

Lab 1 Flow Diagram

Answers & Solutions

Each lab contains answers to all questions and results or solutions.

You are encouraged to verify your results by checking the ***Answers/Solutions*** *section at the* ***end of each lab****.*

Part A: Write Immediate Assertions

For this lab, you will continue to use the same DUT and the same testbench that was developed in the SVTB workshop.

Task 1. Go into Lab1 Working Directory

1. Login to workstation if needed (use login/password provided by instructor)
2. CD into **lab1** directory.

```
> cd labs/lab1
```

The content of the directory is from `lab3` of the SVTB workshop.

Task 2. Add Immediate Assertion to Testbench

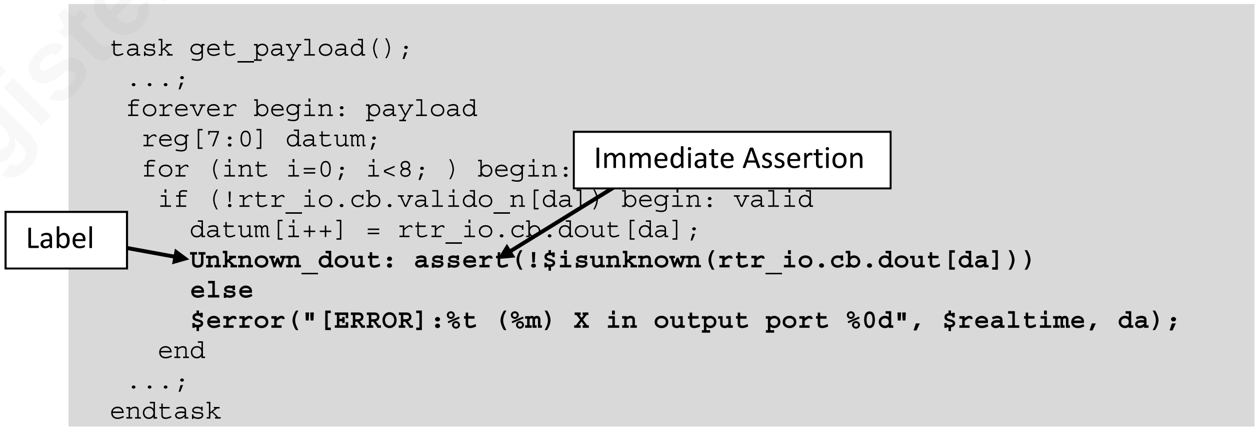
Within testbenches there are two distinct functional verfications that take place: correctness of data processing and correctness of protocol execution. Correctness of data processing is typically verified in a Scoreboard that accesses a reference model for validity check. Correctness of protocol execution is typically verified within a bus monitor that directly monitors the activities of the physical wires of the DUT.

Assertion in testbenches are usually embedded in bus monitors. In this test **get_payload()** method is the bus monitor. You will experiment with immediate assertions here.

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

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

2. In task **get_payload()**, add immediate assert statements to print error statements if the **dout** signal from the DUT contains unknowns.
3. Make sure you label this assertion.
4. When finished, your code should look like:



```
task get_payload();
 ...;
 forever begin: payload
  reg[7:0] datum;
  for (int i=0; i<8; ) begin:
   if (!rtr_io.cb.valido_n[da]) begin: valid
     datum[i++] = rtr_io.cb.dout[da];
     Unknown_dout: assert(!$isunknown(rtr_io.cb.dout[da]))
     else
     $error("[ERROR]:%t (%m) X in output port %0d", $realtime, da);
   end
 ...;
endtask
```

5. Save and close the file.

Task 3. Compile and Simulate

When compling and simulating with immediate assertions, no additional compile or run-time switch is required.

1. Compile and simulate the testbench.

```
> make
```

There should be no errors. Simulation should stop after 2,000 packets.

2. Compile and simulate a known bad DUT.

```
> make bad
```

You should see many error messages similar to the one below.

Message printed by user **`$error`** system call

```
[ERROR]:6944050.0ns
(router_test_top.t.get_payload.payload.get_byte.valid.Unknown_dout) X in port 8
"./test.sv", 90:
router_test_top.t.get_payload.payload.unnamed$$_0.get_byte.valid.Unknown_dout:
started at 6944150000ps failed at 6944150000ps
      Offending '(!$isunknown(rtr_io.cb.dout[da]))'
```

Default Message printed by $error

The immediate assertion caught all occurances of the **`dout`** signal containing X as the signal value.

However, the assertion failure did not stop simulation. If you wanted to stop simulation, you need to call the **`$fatal()`** system task. You will do this in Part B.

Continue on to Part B to experiment with concurrent assertions.

Part B: Write Concurrent Assertions

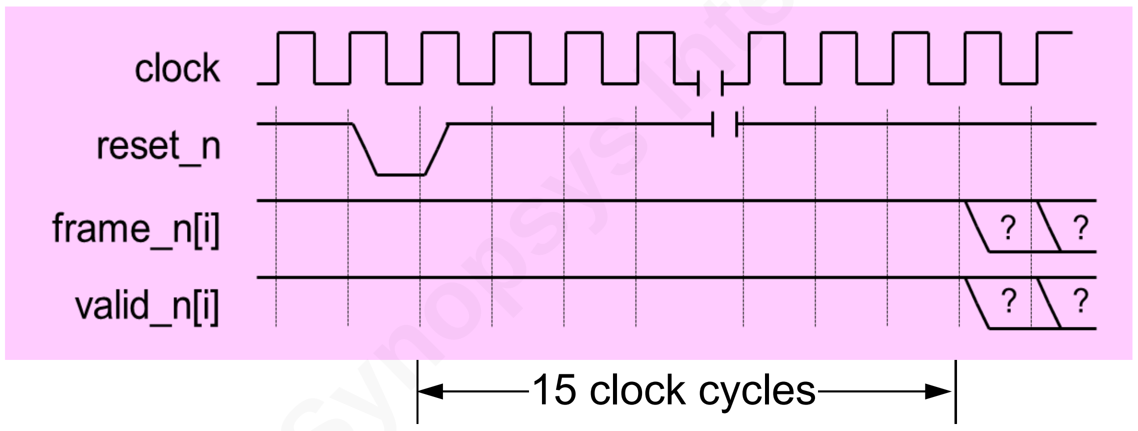
Concurrent assertions are typically embedded in-line within RTL code or coded in a separate module then bound to the RTL code.

In this lab, you will embed the concurrent assertions in-line within the RTL code. One convenient place to do this is within the interface block.

Task 1. Write a Simple Concurrent Assertion

1. Open **`router_io.sv`** file in an editor.

One of the specifications of the DUT is that the control signal **`router.frame_n`** must stay high for at least 15 clock cycles after the **`router.reset_n`** signal returns back to high.



2. Implement this reset protocol requirement with a concurrent assertion statement.

3. Add a failure action block with a **fatal** severity.

4. When finished, your code may look like the following:

Clocking event | Concurrent Assertion | Property Expression | Failure action block

```
reset: assert property (
    @(posedge clock) $rose(reset_n) |-> not(##[0:14] ~&(frame_n))
  ) else begin
    $fatal(1,"[FATAL]:%t (%m) reset violation!!!", $realtime);
  end
```

5. Save and close the file.

Task 2. Compile and Simulate

1. Compile and run the simulation with the following command:

```
> vcs -sverilog –debug_access+all –R –f files
```

Simulation should run successfully and end after 2,000 packets. The –debug_access+all option dumps waveform data for DVE and Verdi. There are two entries for Verdi dumping in the module **`router_test_top`**:

```
$fsdbDumpvars;
$fsdbDumpSVA;
```

Task 3. Create a Violation of the Assertion

To see that the assertion can actually catch an error, create a violation of the assertion.

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

2. Locate the **`reset()`** routine.

3. Reduce the time advancement statement (last statement of the routine) from 15 to 14.

```
repeat(15) @(rtr_io.cb);
```

Change To:

```
repeat(14) @(rtr_io.cb);
```

4. Recompile, run the simulation, and capture the results for DVE or Verdi debugging of assertions (using additional options shown in italics):

```
> setenv FSDB_SVA_SUCCESS 1
> vcs -sverilog –debug_access+all –f files –assert dve \
–assert enable_diag -kdb -lca
> ./simv -l simv.log +fsdb+functions
```

By default, Verdi suppresses recording and display of assertion successes for performance reasons. Set environment variable **`FSDB_SVA_SUCCESS`** to 1 to enable verdi to record and display assertion successes. Failures are always displayed.
The **`+fsdb+functions`** runtime option records assertions inside functions and necessary, since our assertion is inside a SystemVerilog function.

5. You should see the following failure reported:

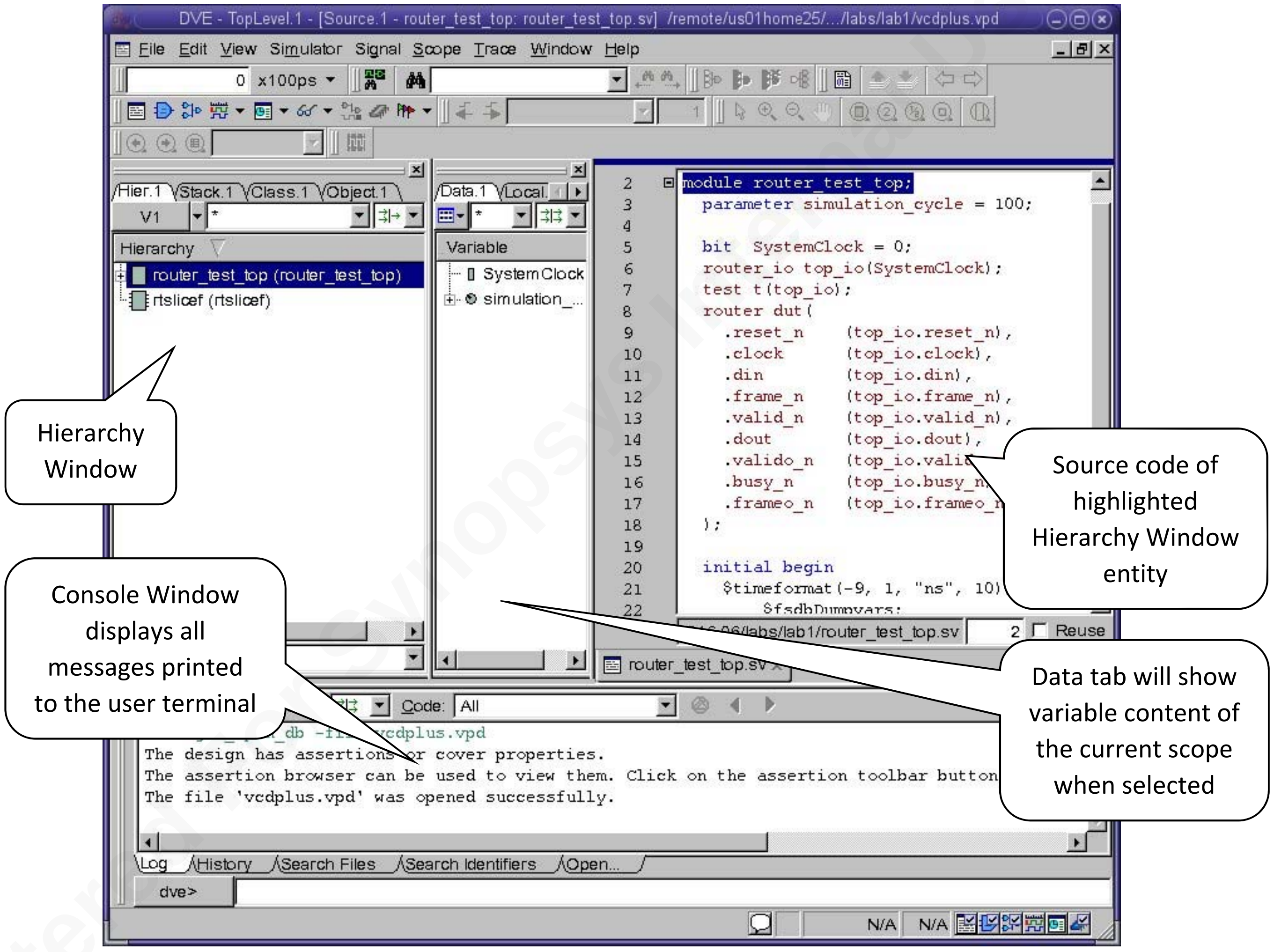
```
"./router_io.sv", 11: router_test_top.top_io.reset: started at 250000ps
failed at 1650000ps
Fatal: "./router_io.sv", 11: router_test_top.top_io.reset: at time 1650000 ps
[FATAL]:  1650.0ns (router_test_top.top_io.reset) reset violation!!!
$finish called from file "./router_io.sv", line 11.
$finish at simulation time   1650.0ns
```

Task 4. Graphical Debugging: Invoke DVE

1. If you wish to use Verdi, skip to Task 6.
2. Open the Discovery and Visualization Environment (DVE).

```
> dve -vpd vcdplus.vpd &
```

The following dve **`TopLevel`** window will open. From this window, you can access all elements of RTL and Testbench. By default, you do not see assertions.


DVE - TopLevel.1 - [Source.1 - router_test_top: router_test_top.sv] /remote/us01home25/.../labs/lab1/vcdplus.vpd
File Edit View Simulator Signal Scope Trace Window Help
Hier.1 Stack.1 Class.1 Object.1
Hierarchy
router_test_top (router_test_top)
rtslicef (rtslicef)
Data.1 Local.
Variable
SystemClock
simulation_...
router_test_top.sv
Code: All
Log History Search Files Search Identifiers Open...
dve>
Hierarchy Window
Console Window displays all messages printed to the user terminal
Source code of highlighted Hierarchy Window entity
Data tab will show variable content of the current scope when selected

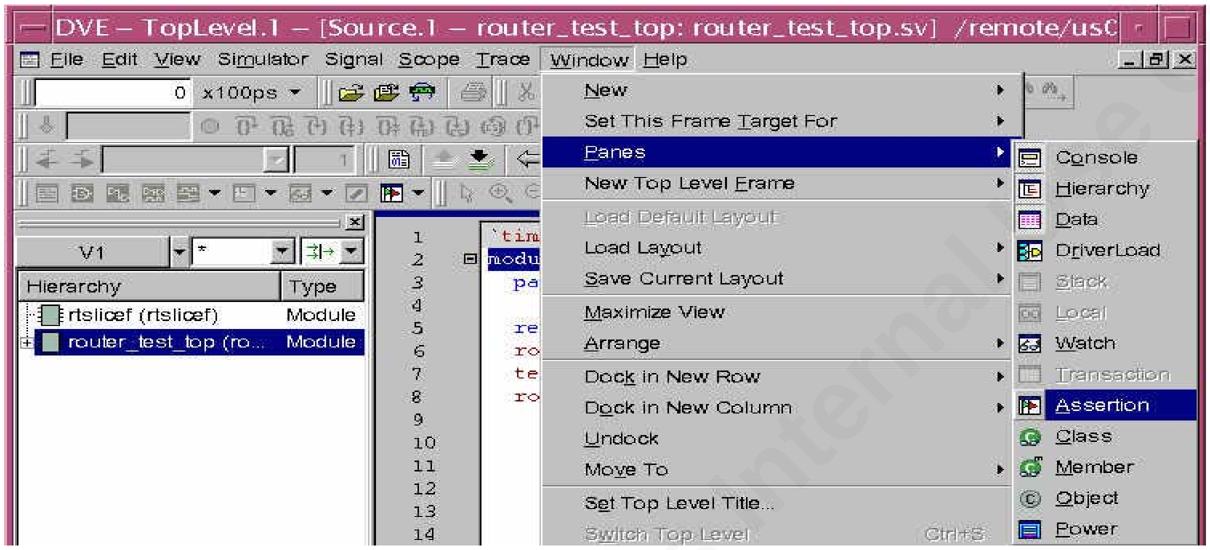
```
module router_test_top;
  parameter simulation_cycle = 100;

  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.vali
    .busy_n      (top_io.busy_n
    .frameo_n    (top_io.frameo_
  );

  initial begin
    $timeformat(-9, 1, "ns", 10)
      $fsdbDumpvars;
```

```
_db -fi          vcdplus.vpd
The design has assertions or cover properties.
The assertion browser can be used to view them. Click on the assertion toolbar button
The file 'vcdplus.vpd' was opened successfully.
```

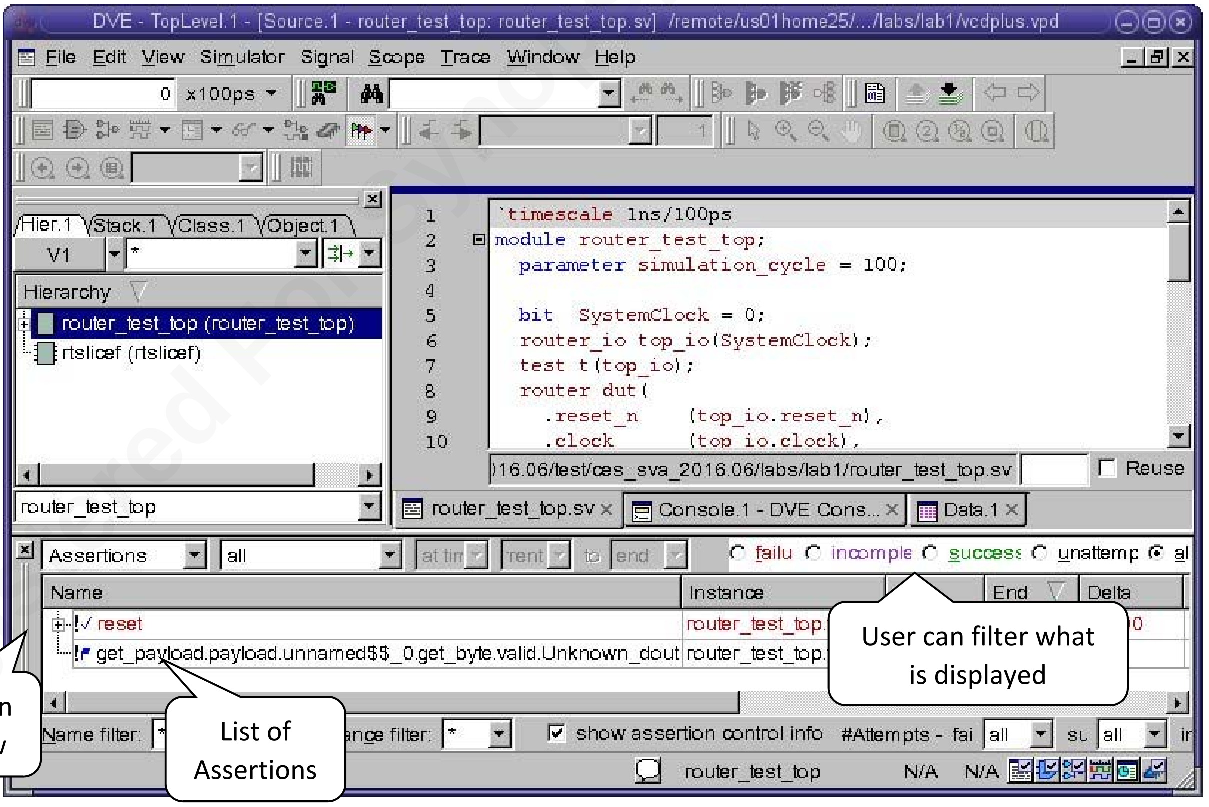
Task 5. View Assertions in TopLevel Window

1. To view assertions in the TopLevel window, try the following menu sequence:

   Window -> Panes -> Assertion



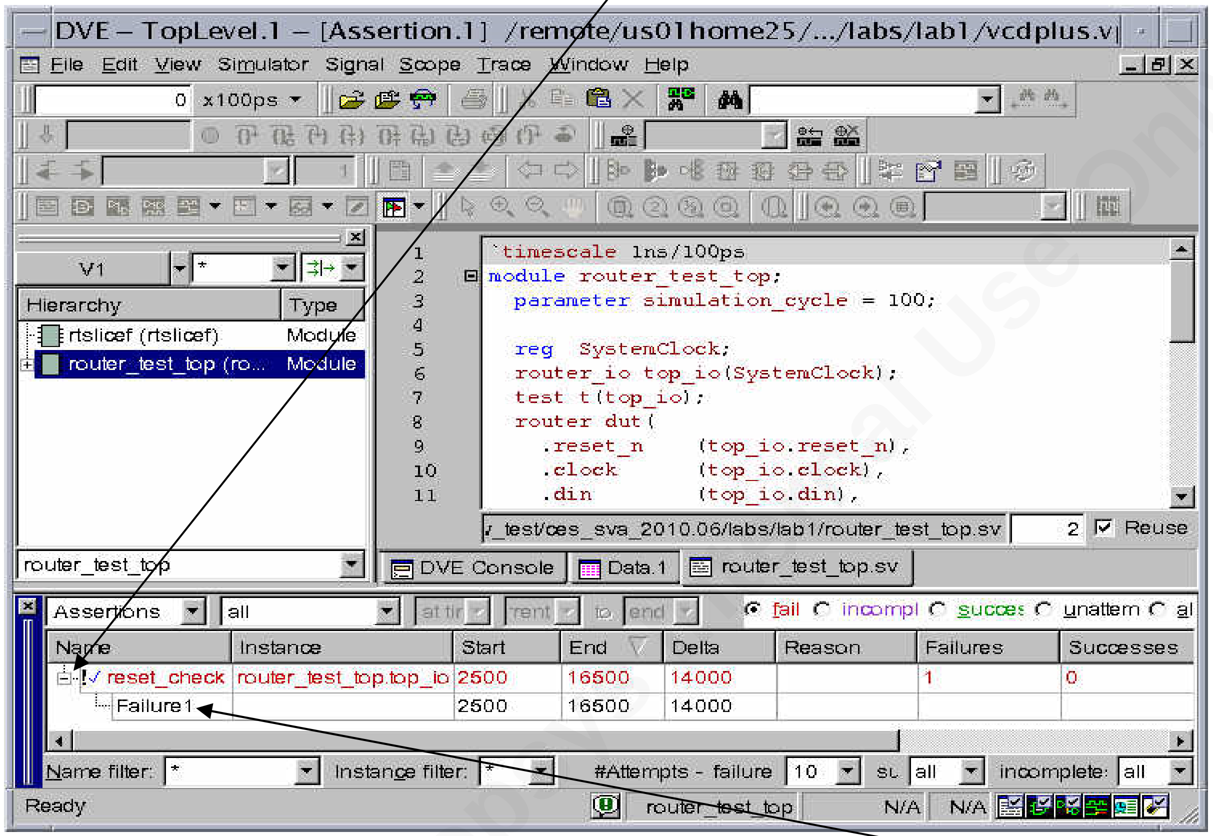
The Assertions window will open above Console window.


DVE - TopLevel.1 - [Source.1 - router_test_top: router_test_top.sv] /remote/us01home25/.../labs/lab1/vcdplus.vpd
File Edit View Simulator Signal Scope Trace Window Help
Hier.1 Stack.1 Class.1 Object.1
Hierarchy
router_test_top (router_test_top)
rtslicef (rtslicef)
router_test_top.sv Console.1 - DVE Cons... Data.1
Assertions all
failu incomple success unattemp al
Name filter: ance filter: show assertion control info #Attempts - fai all su all
Assertion Window
List of Assertions
User can filter what is displayed

```
`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),
```

Name	Instance	End	Delta
reset	router_test_top.		0
get_payload.payload.unnamed$$_0.get_byte.valid.Unknown_dout	router_test_top.		

2. Expand the assertion by clicking on ⊞ to see report list of success and failures.


DVE – TopLevel.1 – [Assertion.1] /remote/us01home25/.../labs/lab1/vcdplus.v
File Edit View Simulator Signal Scope Trace Window Help
0 x100ps
V1
Hierarchy Type
rtslicef (rtslicef) Module
router_test_top (ro... Module
router_test_top
DVE Console
Data.1
router_test_top.sv
Reuse
Ready
router_test_top
N/A N/A

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

  reg  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),
```

/_test/ces_sva_2010.06/labs/lab1/router_test_top.sv 2

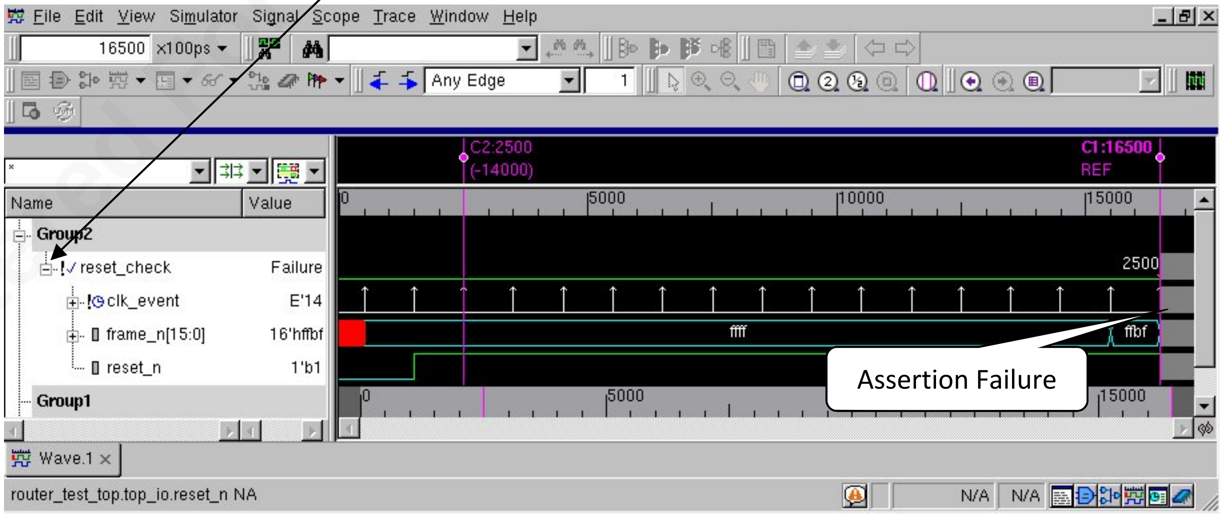
Assertions: all — (● fail ○ incompl ○ succes ○ unattem ○ al)

Name	Instance	Start	End	Delta	Reason	Failures	Successes
⊟ ! ✓ reset_check	router_test_top.top_io	2500	16500	14000		1	0
Failure1		2500	16500	14000			

Name filter: * Instance filter: * #Attempts - failure 10 su all incomplete: all

3. To view the assertions in a Waveform window, double click on the Failure.

The Waveform window will open displaying the assertion. In the Waveform window use the ⊞ (or ⊟) before the assertion name to expand (or collapse) to view(hide) the components of the assertion.



Name	Value
⊟ Group2	
⊟ ! ✓ reset_check	Failure
⊞ ! clk_event	E'14
⊞ frame_n[15:0]	16'hffbf
reset_n	1'b1
Group1	

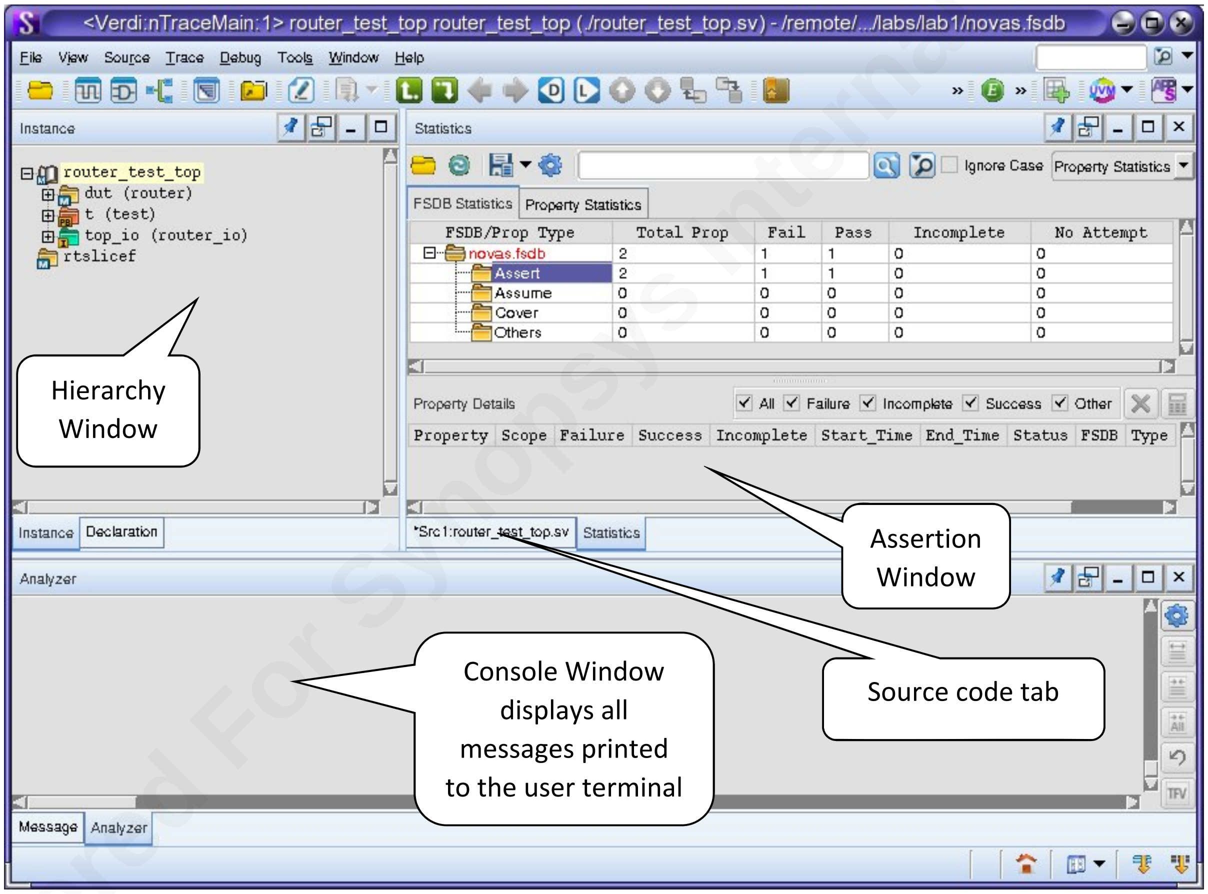
router_test_top.top_io.reset_n NA

Task 6. Graphical Debugging: Invoke Verdi

1. If you chose to use DVE in step 4, skip to step 8.
2. Invoke Verdi.

```
> verdi -ssf novas.fsdb -undockWin -workMode assertionDebug &
```

The following Verdi nTraceMain window will open. From this window, you can access all elements of RTL and Testbench. The command above also opened the Assertion debugging window for you. A Waveform nWave window will also open.



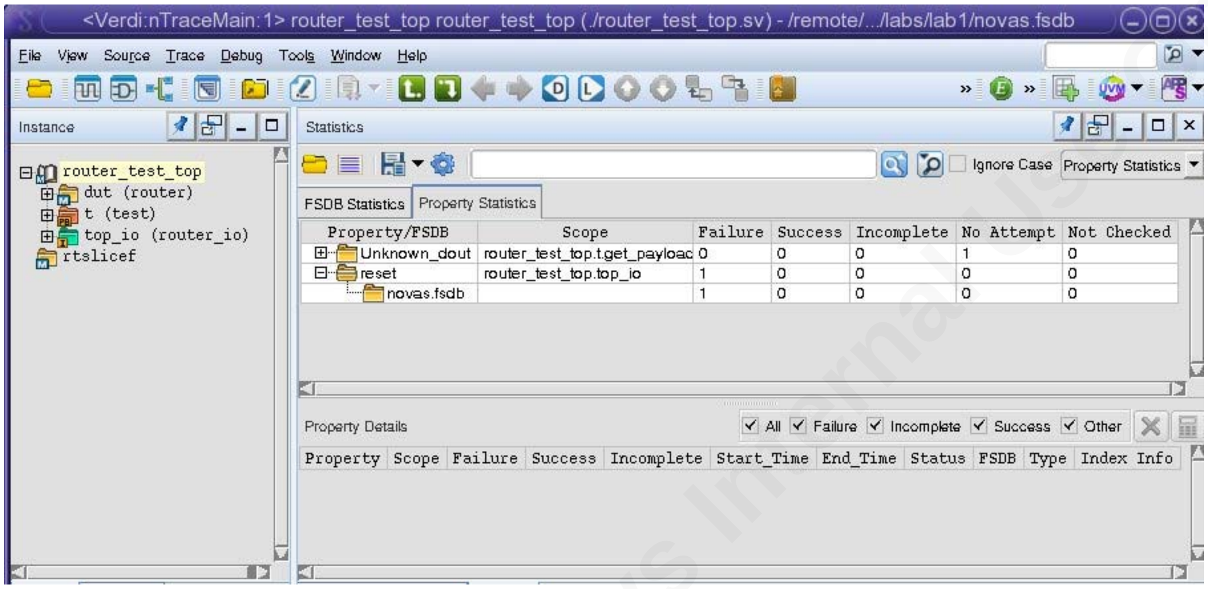
<Verdi:nTraceMain:1> router_test_top router_test_top (./router_test_top.sv) - /remote/.../labs/lab1/novas.fsdb
File View Source Trace Debug Tools Window Help
Instance
router_test_top
dut (router)
t (test)
top_io (router_io)
rtslicef
Instance Declaration
Statistics
Ignore Case
Property Statistics
FSDB Statistics Property Statistics
Property Details
All Failure Incomplete Success Other
*Src1:router_test_top.sv Statistics
Analyzer
Message Analyzer
Hierarchy Window
Assertion Window
Console Window displays all messages printed to the user terminal
Source code tab

FSDB/Prop Type	Total Prop	Fail	Pass	Incomplete	No Attempt
novas.fsdb	2	1	1	0	0
Assert	2	1	1	0	0
Assume	0	0	0	0	0
Cover	0	0	0	0	0
Others	0	0	0	0	0

Property	Scope	Failure	Success	Incomplete	Start_Time	End_Time	Status	FSDB	Type

Task 7. View Assertions

1. Select the Property Statistics tab and expand the reset property. You will see a novas.fsdb entry.

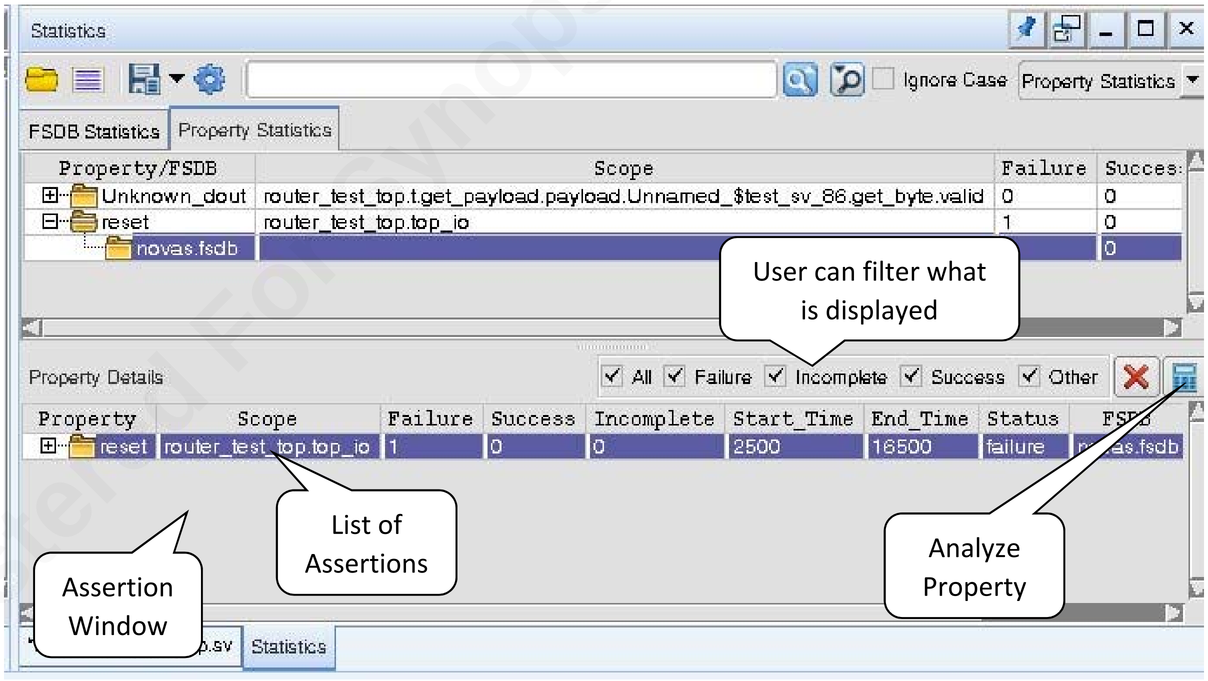


<Verdi:nTraceMain:1> router_test_top router_test_top (./router_test_top.sv) - /remote/.../labs/lab1/novas.fsdb
File View Source Trace Debug Tools Window Help
Instance
router_test_top
dut (router)
t (test)
top_io (router_io)
rtslicef
Statistics
Ignore Case
Property Statistics
FSDB Statistics
Property Statistics
Property Details
All Failure Incomplete Success Other

Property/FSDB	Scope	Failure	Success	Incomplete	No Attempt	Not Checked
Unknown_dout	router_test_top.t.get_payloac	0	0	0	1	0
reset	router_test_top.top_io	1	0	0	0	0
novas.fsdb		1	0	0	0	0

Property	Scope	Failure	Success	Incomplete	Start_Time	End_Time	Status	FSDB	Type	Index	Info

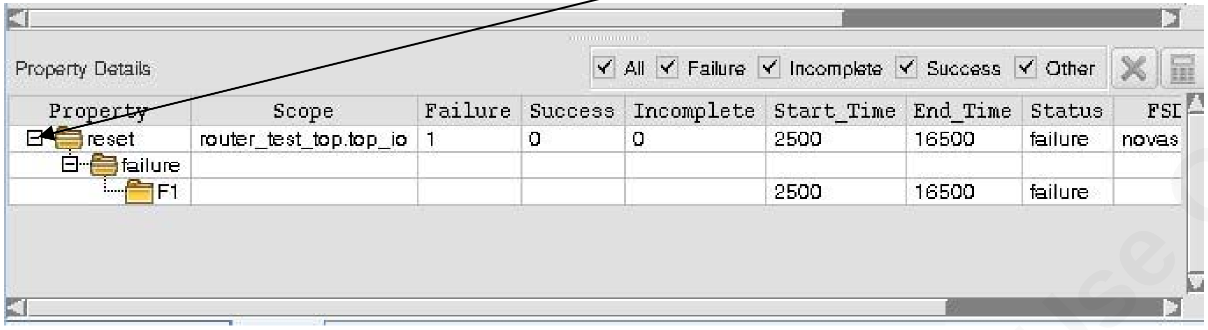
2. Double-click the novas.fsdb folder icon.



Property/FSDB	Scope	Failure	Succes:
Unknown_dout	router_test_top.t.get_payload.payload.Unnamed_$test_sv_86.get_byte.valid	0	0
reset	router_test_top.top_io	1	0
novas.fsdb			0

Property	Scope	Failure	Success	Incomplete	Start_Time	End_Time	Status	FSDB
reset	router_test_top.top_io	1	0	0	2500	16500	failure	novas.fsdb

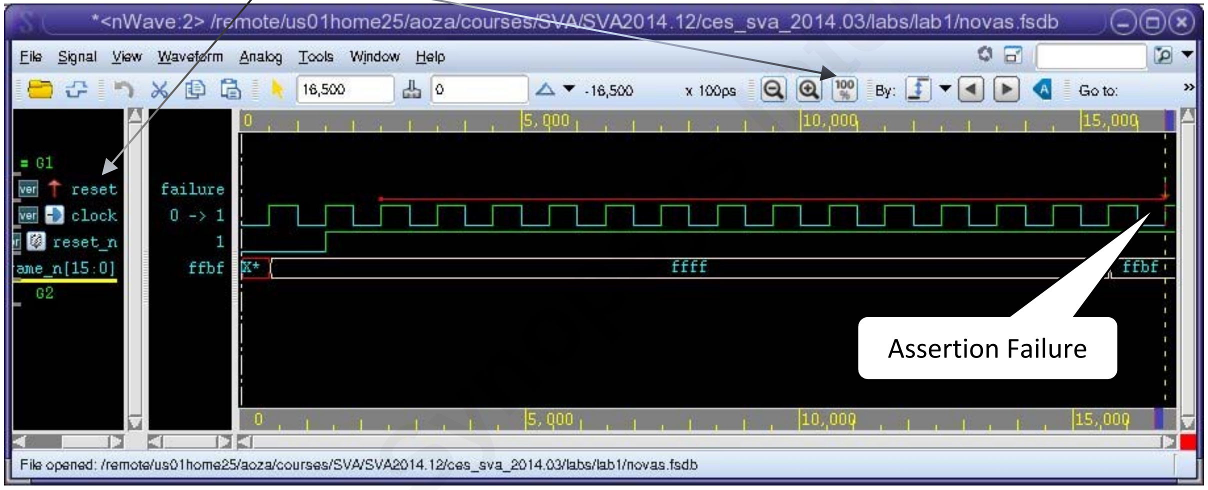
3. Expand the assertion by clicking on ⊞ to see report list of success and failures.



Property Details ✓ All ✓ Failure ✓ Incomplete ✓ Success ✓ Other

Property	Scope	Failure	Success	Incomplete	Start_Time	End_Time	Status	FSI
reset	router_test_top.top_io	1	0	0	2500	16500	failure	novas
failure								
F1					2500	16500	failure	

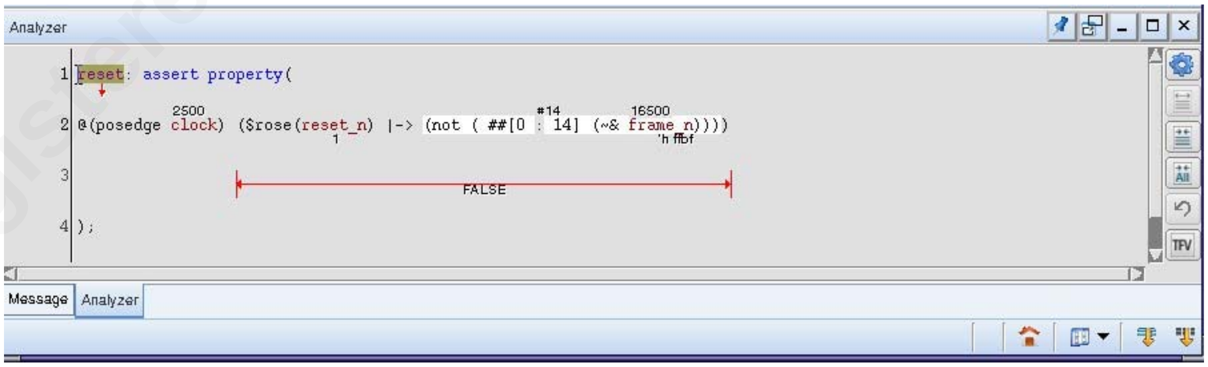
4. To view the assertions in a Waveform window, drag Failure F1 using Middle Mouse Button (MMB) into the open nWave (Waveform) Window and expand the reset waveform by double clicking on the name in the nWave window. Select the zoom button on the nWave window.



*<nWave:2> /remote/us01home25/aoza/courses/SVA/SVA2014.12/ces_sva_2014.03/labs/lab1/novas.fsdb

File opened: /remote/us01home25/aoza/courses/SVA/SVA2014.12/ces_sva_2014.03/labs/lab1/novas.fsdb

5. Note the Analyzer at the bottom of the nTraceMain window. This can be used to see the sequence that caused the failure. You can select any property failure or success in the list above (e.g. F1 above) and use the button to display in the Analyzer. The Analyzer works only for properties, not immediate assertions.



Analyzer

```
reset: assert property(
@(posedge clock) ($rose(reset_n) |-> (not ( ##[0 : 14] (~& frame_n))))
);
```

Task 8. Remove the Injected Error

1. Open the **`test.sv`** file with an editor.
2. In the **`reset()`** routine, remove the error:

   Change from:

```
repeat(14) @(rtr_io.cb);
```

   Back to:

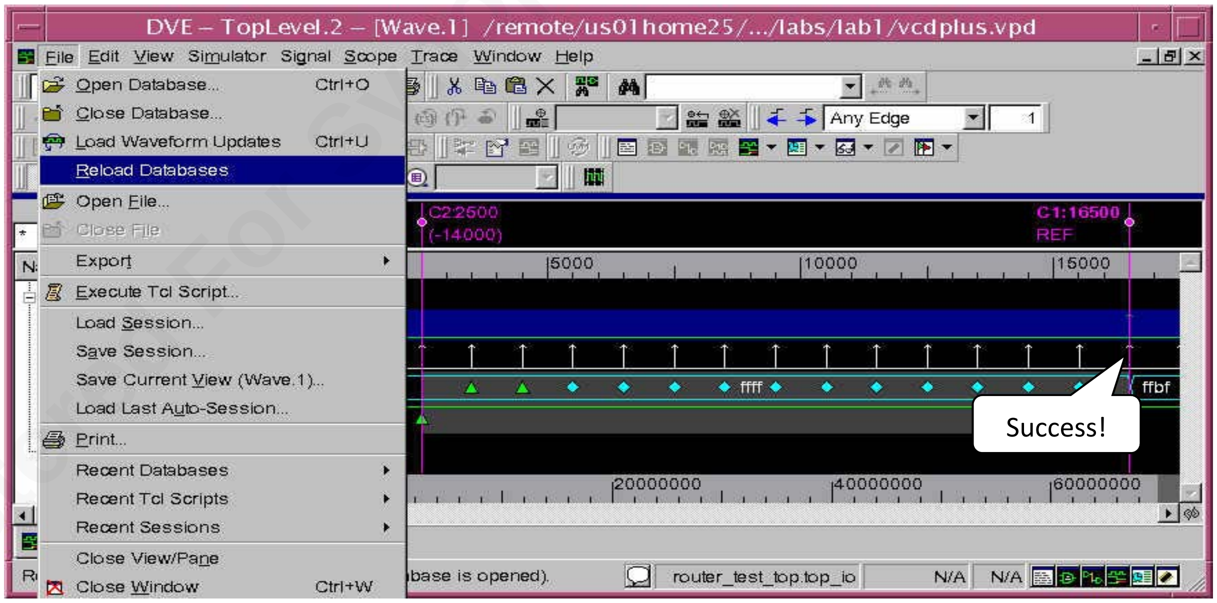
```
repeat(15) @(rtr_io.cb);
```

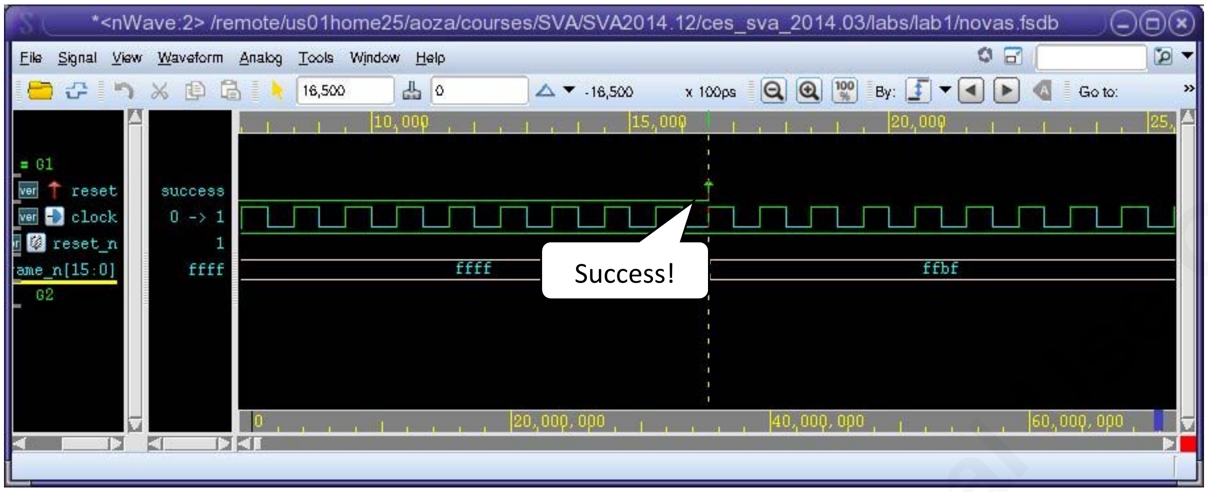
3. Recompile, run the simulation, capturing the results in log file and enable DVE/Verdi debugging of assertions:

```
> make
```

4. At the completion of the simulation, go back to the DVE Waveform window and select **`File -> Reload Databases`** (or **`File -> Reload`** in Verdi nWave).

   You should now see a success for the reset assertion.




*<nWave:2> /remote/us01home25/aoza/courses/SVA/SVA2014.12/ces_sva_2014.03/labs/lab1/novas.fsdb
File Signal View Waveform Analog Tools Window Help
16,500
0
-16,500
x 100ps
By:
Go to:
G1
reset
clock
reset_n
ame_n[15:0]
G2
success
0 -> 1
1
ffff
ffff
ffbf
Success!

The Analyzer in Verdi will also clear since the result has changed. You can also analyze successes if you wish.

Task 9. View Assertion Errors

The immediate assertions can also be viewed in DVE or Verdi.

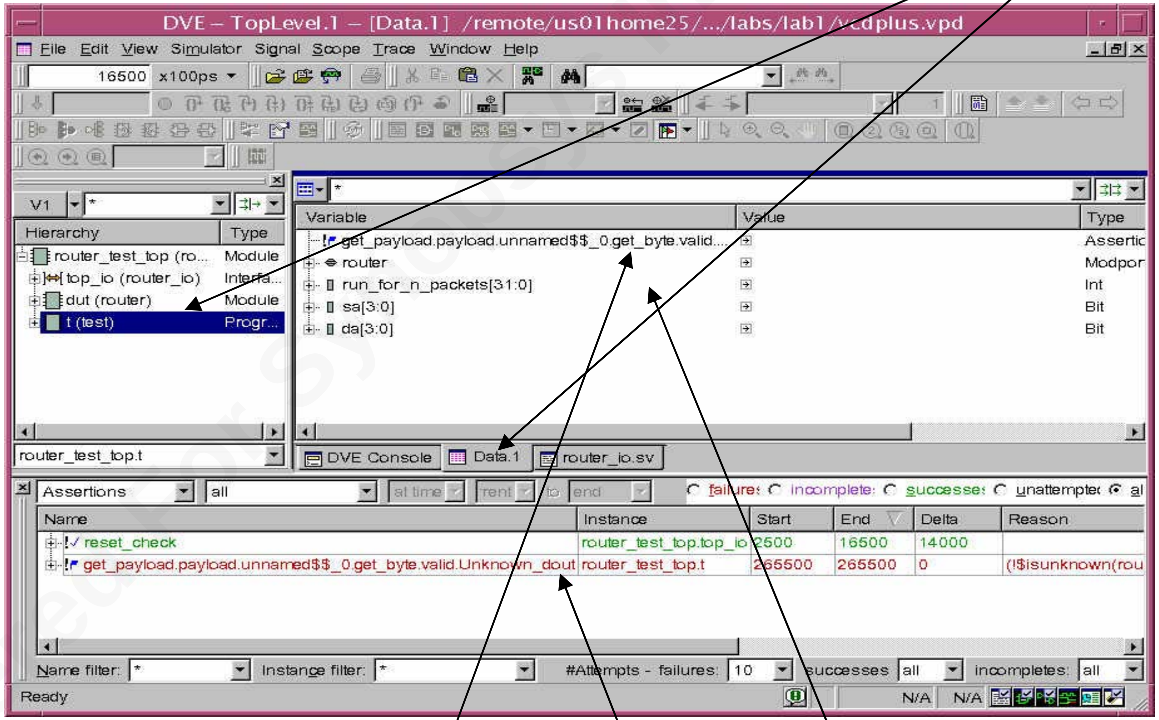
1. Recompile, run the simulation, capturing the results in log file and enable dve debugging of assertions but with a bad DUT:

```
> make bad
```

2. Go to Task 12 if using Verdi.
At the completion of simulation, go to the TopLevel window and refresh the simulation results with: **`File -> Reload Databases`**

Assertions can also be accessed via the data hierarchy.

3. An alternate way of accessing the embedded assertion is to click on the **`Data`** tab. Then click on the instance you want to view (in this case **`test`**)


DVE – TopLevel.1 – [Data.1] /remote/us01home25/.../labs/lab1/vcdplus.vpd

Hierarchy	Type
router_test_top (ro...	Module
top_io (router_io)	Interfa...
dut (router)	Module
t (test)	Progr...

router_test_top.t

Variable	Value	Type
get_payload.payload.unnamed$$_0.get_byte.valid....		Assertic
router		Modpor
run_for_n_packets[31:0]		Int
sa[3:0]		Bit
da[3:0]		Bit

Name	Instance	Start	End	Delta	Reason
reset_check	router_test_top.top_io	2500	16500	14000	
get_payload.payload.unnamed$$_0.get_byte.valid.Unknown_dout	router_test_top.t	265500	265500	0	(!$isunknown(rou

Name filter: * Instance filter: * #Attempts - failures: 10 successes: all incompletes: all

Ready

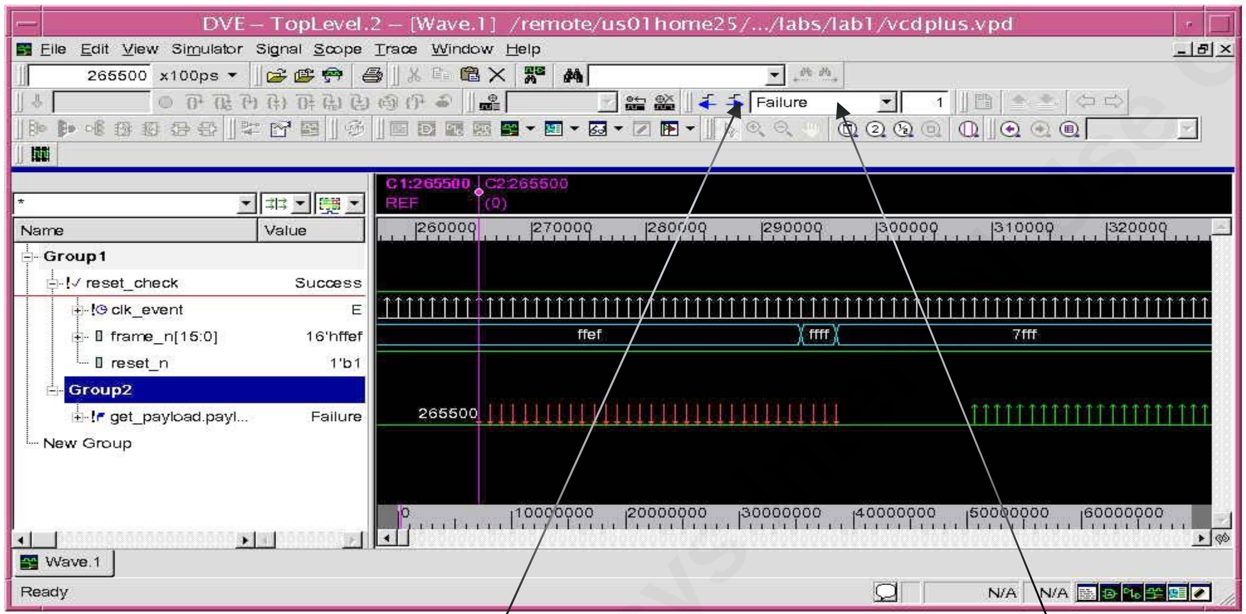
You should see all variables of **`test`**, including the assertions.

4. There are two ways to add the assertions to the existing Waveform window:

- Double clicking on the assertion in the Assertion pane
- Highlight the assertion in the Data pane, click on right button of the mouse and select
  `Add To Waves` followed by **`Add To Wave.1`**(number may differ)

Execute one of these to add the immediate assertion to the Waveform window.

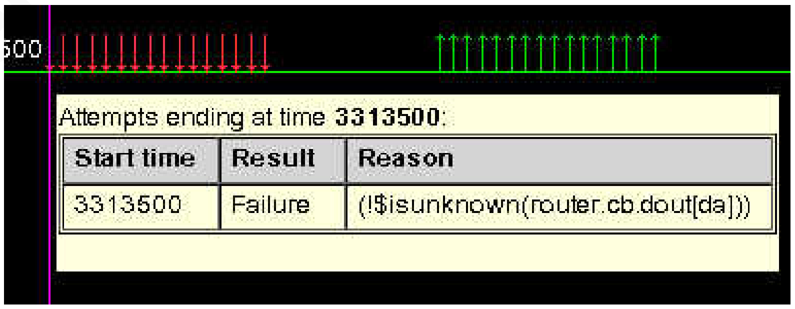
Task 10. Searching for Assertion Results (DVE)



Note: The white **`clk_event`** arrows tell you where the assertions were fired. Successes and failures of assertions are denoted by the green and red arrows respectively.

1. To find points of failure click on the search criteria field and select **`Failure`**.
2. Click on the search direction button to bring you to the point of failure for the highlighted assertion.
3. Place your cursor over a specific failure (or success) in the waveform window.

You will see a box showing the starting and ending times of the success and failure assertions. In complex sequences, more than one assertion may end at the same simulation time.



Attempts ending at time 3313500:

Start time	Result	Reason
3313500	Failure	(!$isunknown(router.cb.dout[da]))

Task 11. Tracing Assertions Back to Source Code (DVE)

You can also trace assertion failures(or successes) in the source code window.

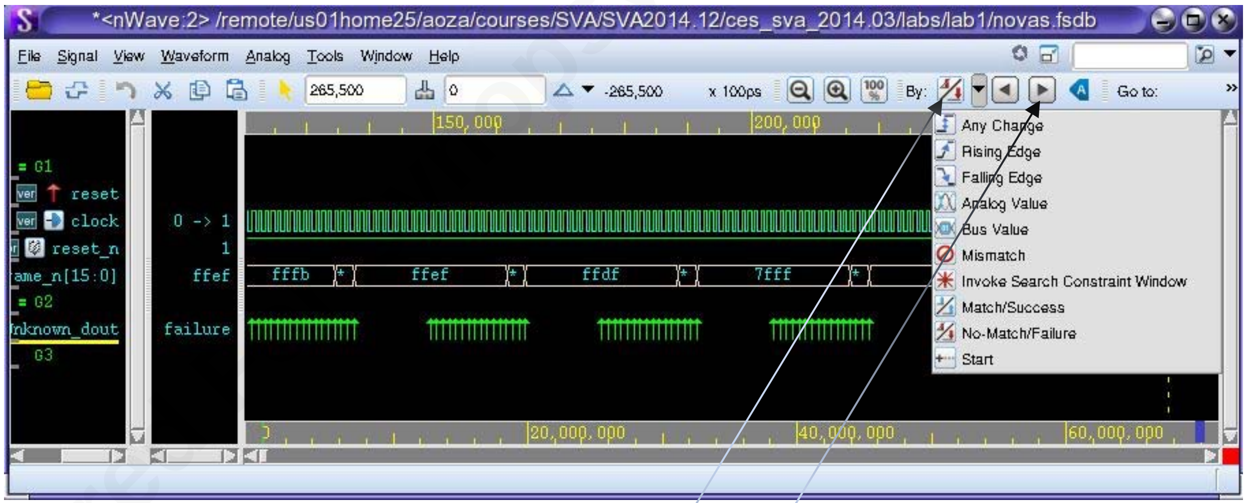
1. To do this, go back to the TopLevel Window.
2. In the Assertion Pane, expand the assertions to show all failures (or successes).
3. Double click on the failure (or success) that you want to trace.

   This will set both the Source Code Window and the Waveform Window to show the point of failure (or success). You can then start the debugging process.

   Note: This feature has not yet been implemented for assertions in a Testbench (`program` construct).

Task 12. Searching for Assertion Results (Verdi)

1. Go to the nWave window and refresh the simulation results with: **`File -> Reload`**. Using the MMB drag the **`Unknown_dout`** assertion from the Property Statistics pane of the nTraceMain window to the nWave window.



Note: For properties, the starting point of the assertion evaluation is a green (red) dot. A green (red) straight line follows the cycles of the assertion till a success (failure). Successes and failures of assertions are denoted by green and red arrows respectively.

2. Note that the cursor is automatically placed at the first failure.
3. To find other points of failure(success) click on the search criteria drop-down and select **`No-Match/Failure(Match/Success)`**
4. Clicking on the search direction button brings you to the next or previous point of failure(success) for the highlighted assertion.

Task 13. Tracing Assertions Back to Source Code (Verdi)

You can also trace assertion failures (or successes) in the source code window.

1. Double click on the failure (or success) that you want to trace in the nWave window.

This will set both, the Source Code Window and the Waveform Window to show the point of failure (or success). You can then start the debugging process. If this is a property it will also show in the Analyzer Window.

Note: This feature has not yet been implemented for assertions in a SystemVerilog `program` construct.

Note: The assertion in the function `get_payload()` in the test cannot access the interface signals when debugging in Verdi. Hence a local variable `dout` was used which also shows up in the Verdi Analyzer Window.

Congratulations! You have successfully added immediate and concurrent assertions to a test program and a DUT respectively. You compiled and simulated the testbenches with VCS, using switches necessary for assertions. You navigated DVE or Verdi to view assertion details and display specific assertion successes and failures. You also traced the source of these errors in the source code.

Congratulations! You have completed Lab 1.

Answers / Solutions

Part A: Write Immediate Assertion

test.sv (task get payload()) Solution:

```
task get_payload();
   automatic int brk = 0;
    pkt2cmp_payload.delete();
    @(negedge rtr_io.cb.frameo_n[da]);
    forever begin: payload
      logic[7:0] datum;
      for (int i=0; i<8; ) begin: get_byte
        if (!rtr_io.cb.valido_n[da]) begin: valid
          datum[i++] = rtr_io.cb.dout[da];

          begin logic dout;  dout = rtr_io.cb.dout[da];
          Unknown_dout: assert(!$isunknown(dout))
          else
          $error("[ERROR]:%t (%m) X in output port %0d", $realtime, da);
          end
        end
        if (rtr_io.cb.frameo_n[da])
          if (i == 8) begin
            pkt2cmp_payload.push_back(datum);
            brk = 1;
            break;
          end
          else begin
            $display("\n%m\n[ERROR]%t Payload not byte aligned!\n", $realtime);
            $finish;
          end
        @(rtr_io.cb);
      end: get_byte
      if (brk) break;
      pkt2cmp_payload.push_back(datum);
    end
endtask
```

Part B: Write Concurrent Assertion

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;

  reset: assert property (
    @(posedge clock) $rose(reset_n) |-> not (##[0:14] ~&(frame_n))
    ) else begin
    $fatal(1,"[FATAL]:%t (%m) reset violation!!!", $realtime);
    end

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

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

2

Implement Sequences & Properties in Assertions

Learning Objectives

After completing this lab, you should be able to:

- Create Named Property
- Create Named Sequence
- Use named sequence in properties for protocol checking
- Implement and assert properties in assertion modules
- Supplement SVA with behavioral Verilog code
- Bind modules to RTL modules

Lab Duration:
60 minutes

Getting Started

In Lab 1 you implemented assertions in the Testbench and the DUT.

In this lab you will define your assertions inside a separate module in a separate file. You will then bind this module to the Device Under Test via SystemVerilog bind construct. This method of implemening assertions brings additional flexibility into play for assertions. With this style of assertion implementation, you can add and modify assertions in your verification flow without modifying your RTL or Testbench code.

The DUT being monitored continues to be the 16x16 Router from the SVTB workshop.

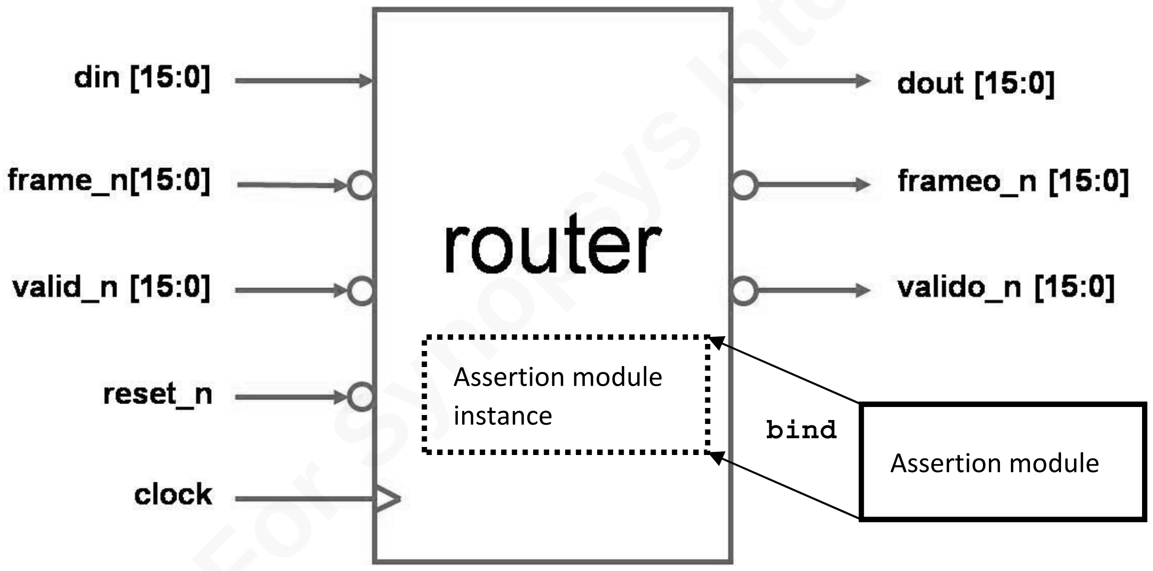


Figure 1. 16x16 Router

Overview

This lab flow steps you through binding assertion modules, defining properties in the assertion module, defining sequences and using them inside properties and reading the error messages. Then it shows how to add behavioral procedural code to assertions in action blocks.

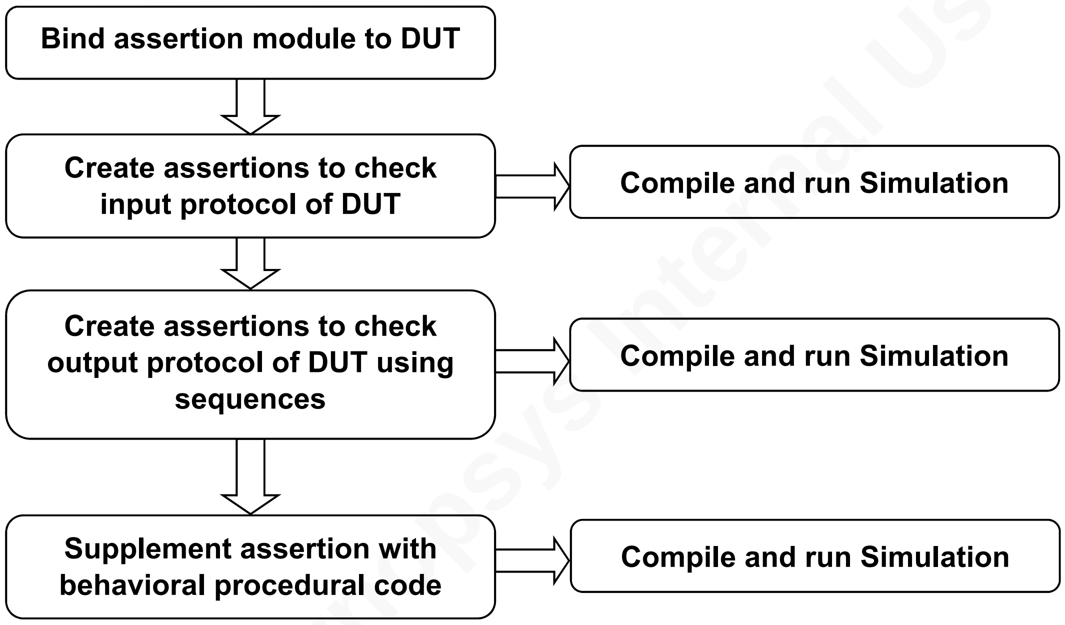

Bind assertion module to DUT
Create assertions to check input protocol of DUT
Compile and run Simulation
Create assertions to check output protocol of DUT using sequences
Compile and run Simulation
Supplement assertion with behavioral procedural code
Compile and run Simulation

Figure 2. Lab 2 Flow Diagram

Input Protocol Check Assertion

Task 1. Go into Lab2 Working Directory

1. CD into **`lab2`** directory

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

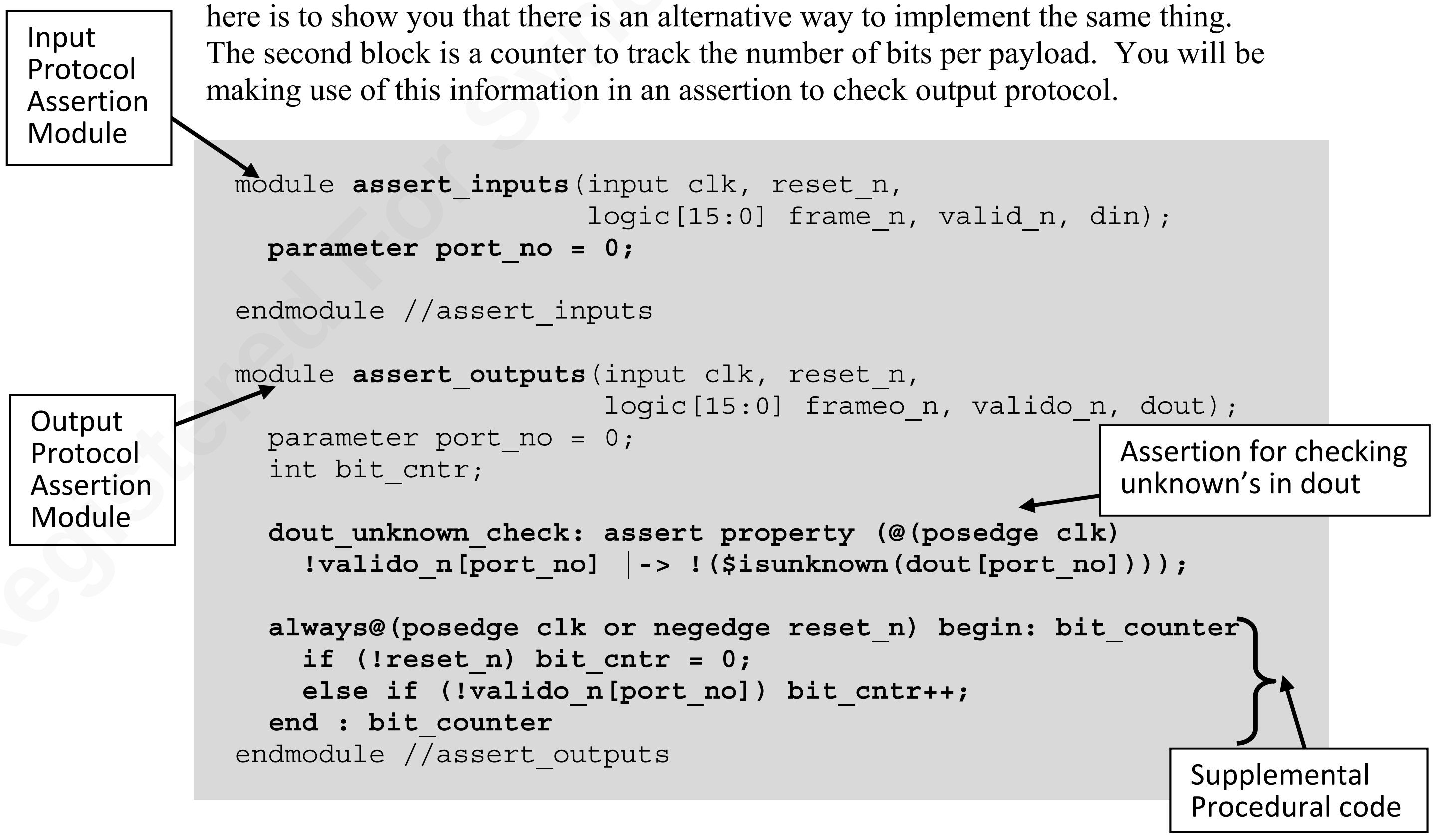
Task 2. Get Familiar with Assertion Module

1. Open the existing **`assert.sva`** file in an editor

Assertions are typically encapsulated within assertion modules, which are then bound to DUT using a SystemVerilog **`bind`** statement.

This file contains two such modules: **`assert_inputs`** for input protocol assertion checks and **`assert_outputs`** for output protocol checks. The port list of **`assert_inputs`** module consists of the input signals of the DUT. The port list of the **`assert_outputs`** module consists of the output signals of the DUT.

The **`assert_inputs`** module is left blank for you to fill in over the course of this lab. The **`assert_outputs`** module contains two blocks already written. The first block is a concurrent assertion check for unknown's in the **`dout`** signal. In Lab 1, you embedded this check with an immediate assertion in the testbench. The code here is to show you that there is an alternative way to implement the same thing. The second block is a counter to track the number of bits per payload. You will be making use of this information in an assertion to check output protocol.


Input Protocol Assertion Module
Output Protocol Assertion Module
Assertion for checking unknown's in dout
Supplemental Procedural code

```
module assert_inputs(input clk, reset_n,
                     logic[15:0] frame_n, valid_n, din);
  parameter port_no = 0;

endmodule //assert_inputs

module assert_outputs(input clk, reset_n,
                      logic[15:0] frameo_n, valido_n, dout);
  parameter port_no = 0;
  int bit_cntr;

  dout_unknown_check: assert property (@(posedge clk)
    !valido_n[port_no] |-> !($isunknown(dout[port_no])));

  always@(posedge clk or negedge reset_n) begin: bit_counter
    if (!reset_n) bit_cntr = 0;
    else if (!valido_n[port_no]) bit_cntr++;
  end : bit_counter
endmodule //assert_outputs
```

Notice that in **`assert_inputs`** module, there is a parameter called **`port_no`**. This parameter will need to be set to indicate which of the sixteen input ports is being monitored by the assertions.

```
parameter port_no = 0;
```

Similarly, there is a **`port_no`** parameter in the **`assert_outputs`** module to select which of the sixteen output ports to monitor.

2. Close the file

Task 3. Create an Assertion Bind Module

A common way to simplfy maintainance of assertion bindings is to create a module specifically to bind assertions.

In this task, you will create such a module.

1. Open the existing **`assert_binds.sva`** file in an editor
2. Create sixteen bind statements to bind sixteen instances of **`assert_inputs`** module each to its respective DUT input port signal.
3. Create sixteen bind statements to bind sixteen instances of **`assert_outputs`** module each to its respective DUT output port signal.

   When completed, your code should look like the following:

```
module assert_binds;

bind router_test_top.dut assert_inputs #(.port_no(0))
     a_iport0(clock, reset_n, frame_n, valid_n, din);
bind router_test_top.dut assert_inputs #(.port_no(1))
     a_iport1(clock, reset_n, frame_n, valid_n, din);
bind router_test_top.dut assert_inputs #(.port_no(2))
     a_iport2(clock, reset_n, frame_n, valid_n, din);
(continued for the rest of the input ports)

bind router_test_top.dut assert_outputs #(.port_no(0))
     a_oport0(clock, reset_n, frameo_n, valido_n, dout);
bind router_test_top.dut assert_outputs #(.port_no(1))
     a_oport1(clock, reset_n, frameo_n, valido_n, dout);
bind router_test_top.dut assert_outputs #(.port_no(2))
     a_oport2(clock, reset_n, frameo_n, valido_n, dout);
(continued for the rest of the output ports)

endmodule //assert_binds
```

Task 4. Compile and Simulate

1. Compile and run the simulation

```
> make
```

The simulation should complete without errors.

To make sure that the binding of the assertion module was done correctly and is able to catch errors, compile and simulate against a faulty DUT.

2. Compile and run simulation on a faulty DUT:

```
> make bad
```

You should see many errors reported. These errors are from asserting that the output is not unknown.

Task 5. Create a Named Property

In the next step you will create a property that checks the input protocol for the **pad** phase of the input packet as shown below.

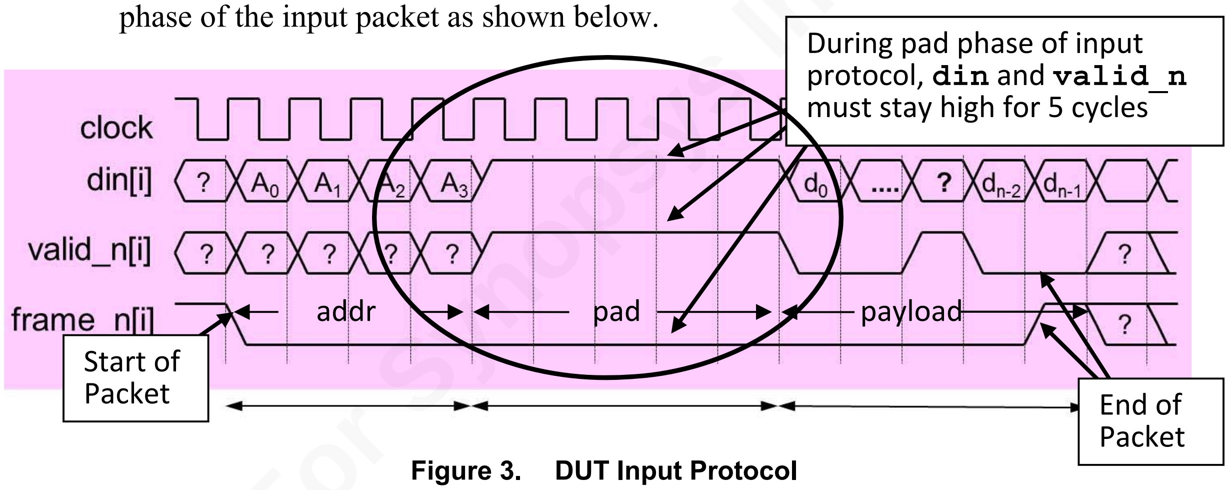


Figure 3. DUT Input Protocol

1. Open **assert.sva** file in an editor
2. Inside the **assert_inputs module**, create a named property called p**_valid_during_pad**

   Pass in arguments **fr_n**, **vld_n**, and **data** as the port list for this property. These represent the frame, valid_n and din input signals respectively of the DUT.

```
property p_valid_during_pad(fr_n, vld_n, data);
endproperty
```

3. Inside the named property, write the property expression which checks the pad protocol

```
@(posedge clk) $fell(fr_n) |-> ##4 (vld_n && data) [*5];
```

Task 6. Assert the Property

1. Assert the named property you just completed. Label this assertion as **`a_vld_hi_in_pad`**.

```
a_vld_hi_in_pad: assert property (
  disable iff(!reset_n)
  p_valid_during_pad(frame_n[port_no], valid_n[port_no],
                                                    din[port_no])
)
```

2. Add an **`else`** clause to the assertion.

```
else begin
 $fatal(1,"[FATAL]%m:
 Pad protocol violation on port %0d at %t", port_no, $realtime);
end
```

Note: An **`else`** clause can only be added in an assertion. It cannot be added in a named property or a named sequence

3. Save and close the file

Task 7. Compile and Simulate

1. Compile and simulate

```
> make
```

2. Check for any errors. There should be no violations of the protocol.

Task 8. Emulate an Error

To make sure that the assertion is working, once again, you want to test it against an error. This time around, you will emulate the error in the **`test.sv`** file.

1. Open **`test.sv`** in an editor
2. Locate the **`send_pad()`** task
3. Change the duration of the pad cycles to 4:

```
repeat(4) @(router.cb);
```

4. Save and close the file
5. Re-run simulation and make sure that you see the error reported

   Note: The first violation reports a fatal error. The assertions on the other input ports do not complete. Hence you will see messages noting the "Antecedent of the implication never satisfied." as shown below.

```
VCD+ Writer K-2015.09-1 Copyright (c) 1991-2015 by Synopsys
Inc.
"./assert.sva", 14:
router_test_top.dut.a_iport7.a_vld_hi_in_pad: started at
1750000ps failed at 2550000ps
        Offending '(valid_n[port_no] && din[port_no])'
Fatal: "./assert.sva", 14:
router_test_top.dut.a_iport7.a_vld_hi_in_pad: at time 2550000
ps
[FATAL]router_test_top.dut.a_iport7.a_vld_hi_in_pad: pad
protocol violation on port 7 at   2550.0ns
$finish called from file "./assert.sva", line 14.
$finish at simulation time   2550.0ns
"./assert.sva", 14:
router_test_top.dut.a_iport6.a_vld_hi_in_pad: Antecedent of
the implication never satisfied.
"./assert.sva", 14:
router_test_top.dut.a_iport5.a_vld_hi_in_pad: Antecedent of
the implication never satisfied.
…
```

6. Edit **`test.sv`** and remove the emulated error

Output Protocol Check Assertions

The specifications of the router outputs are as follows:

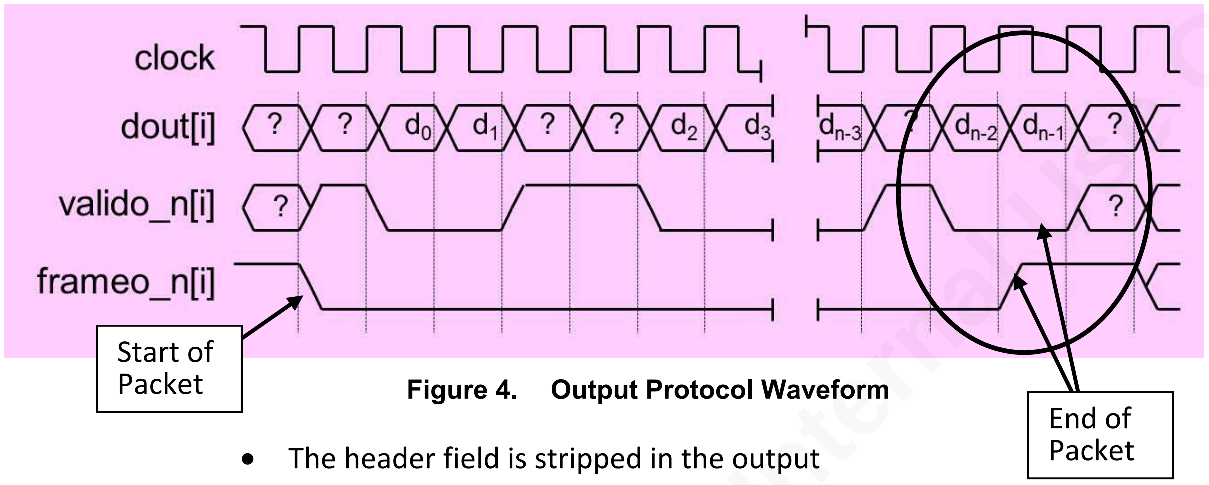


Figure 4. Output Protocol Waveform

- The header field is stripped in the output

The specification for the output of the router requires that **`valido_n`** be active when the last bit of the last byte of the payload comes out of the DUT. The last bit of the payload is indicated by rising **`frameo_n`** signal on the output port.

In this section you will create a property to check for this specification.

Task 9. Create a General Error Message Handler

For convenience of error message display, create a general error message handler that prints the error message along with simulation time and port of failure.

1. Open the **`assert.sva`** file in an editor
2. In **`assert_outputs`** module, create a **`protocol_error_handler`** function with a pass-by-value argument of type **`string`**.
3. In the function, display an error message which includes the port number, simulation time and string argument:

```
function void protocol_error_handler(string msg);
  $display("[ERROR]%t Port %0d: %s",$realtime, port_no, msg);
endfunction
```

This function will be called in the error action blocks of your assertions later.

Task 10. Create a Named Sequence

To make development of assertion properties easier to manage, you can encapsulate fine-grained event sequences within named sequences then use these named sequences in either named properties or directly in **`assert property`** statements.

The following step will ask you to develop such a named sequence.

1. Create a named sequence called **`s_frame`**
2. In this named sequence, write a sequence to detect the period in which the signal **`frameo_n`** stays low.

Hint: Look for **`$fell`** followed by **`$rose`** for the **`frameo_n`** signal. Use the **`->`** (goto operator) to look for **`$rose`**.

Task 11. Create Named Property with Named Sequence

You will now put together a named property to detect the correctness of the last bit.

To make the detection of the correctness of the last bit easier to manage, there is an **`always`** block labeled **`bit_counter`** in the **`assert_outputs`** module that increments a **`int unsigned bit_cntr`** for every valid bit received. You can use this count to verify that the last bit when **`frameo_n`** rises is bit 7.

1. Create a named property called **`p_vld_lo_data_aligned_rsng_frm.`**
2. In the property, use the named sequence **`s_frame`** to detect the end of a packet coming out of DUT
3. In the same cycles that **`s_frame`** ends, check that **`valido_n`** is active and that **`bit_cntr%8`** is 7, indicating that the data is byte-aligned.

Task 12. Assert the Output Protocol Property

1. Write a concurrent assertion using the property **`p_vld_lo_data_aligned_rsng_frm`**. Label this concurrent assertion **`a_vld_lo_data_aligned_rsng_frm`**.
2. In the success action block,
   - Display "End of frame successfully checked" with the simulation time
   - Set **`bit_cntr`** back to 0
3. In the failure action block,
   - Set **`bit_cntr`** back to 0
   - Call the **`protocol_error_handler`**`()` task with an error message
   - Exit simulation with a **`$fatal`** system call
4. Save and close file.

Task 13. Compile and Simulate.

1. Compile the testbench and run the simulation.

   ```
   > make
   ```

   Debug any compile errors.

Task 14. Emulate an Error

To make sure that the assertion is working, once again, you want to test it against an error. You will emulate the error in the **`test.sv`** file.

1. Open **`test.sv`** in an editor.
2. Locate the **`send_payload()`** task.
3. Return the **`frame_n`** signal back to high at bit 6 rather than 7:

```
router.cb.frame_n[sa] <= (index == (payload.size - 1)) && (i == 6);
```

4. Save and close the file.
5. Re-compile and re-run simulation to make sure that you see the correct error reported.

Congratulations! You have completed Lab 2

Answers / Solutions

assert_binds.sva Solution:

```
module assert_binds;
  bind router_test_top.dut assert_inputs #(.port_no(0))
    a_iport0(clock, reset_n, frame_n, valid_n, din);
  bind router_test_top.dut assert_inputs #(.port_no(1))
    a_iport1(clock, reset_n, frame_n, valid_n, din);
  bind router_test_top.dut assert_inputs #(.port_no(2))
    a_iport2(clock, reset_n, frame_n, valid_n, din);
  bind router_test_top.dut assert_inputs #(.port_no(3))
    a_iport3(clock, reset_n, frame_n, valid_n, din);
  bind router_test_top.dut assert_inputs #(.port_no(4))
    a_iport4(clock, reset_n, frame_n, valid_n, din);
  bind router_test_top.dut assert_inputs #(.port_no(5))
    a_iport5(clock, reset_n, frame_n, valid_n, din);
  bind router_test_top.dut assert_inputs #(.port_no(6))
    a_iport6(clock, reset_n, frame_n, valid_n, din);
  bind router_test_top.dut assert_inputs #(.port_no(7))
    a_iport7(clock, reset_n, frame_n, valid_n, din);
  bind router_test_top.dut assert_inputs #(.port_no(8))
    a_iport8(clock, reset_n, frame_n, valid_n, din);
  bind router_test_top.dut assert_inputs #(.port_no(9))
    a_iport9(clock, reset_n, frame_n, valid_n, din);
  bind router_test_top.dut assert_inputs #(.port_no(10))
    a_iport10(clock, reset_n, frame_n, valid_n, din);
  bind router_test_top.dut assert_inputs #(.port_no(11))
    a_iport11(clock, reset_n, frame_n, valid_n, din);
  bind router_test_top.dut assert_inputs #(.port_no(12))
    a_iport12(clock, reset_n, frame_n, valid_n, din);
  bind router_test_top.dut assert_inputs #(.port_no(13))
    a_iport13(clock, reset_n, frame_n, valid_n, din);
  bind router_test_top.dut assert_inputs #(.port_no(14))
    a_iport14(clock, reset_n, frame_n, valid_n, din);
  bind router_test_top.dut assert_inputs #(.port_no(15))
    a_iport15(clock, reset_n, frame_n, valid_n, din);

  bind router_test_top.dut assert_outputs #(.port_no(0))
    a_oport0(clock, reset_n, frameo_n, valido_n, dout);
  bind router_test_top.dut assert_outputs #(.port_no(1))
    a_oport1(clock, reset_n, frameo_n, valido_n, dout);
  bind router_test_top.dut assert_outputs #(.port_no(2))
    a_oport2(clock, reset_n, frameo_n, valido_n, dout);
  bind router_test_top.dut assert_outputs #(.port_no(3))
    a_oport3(clock, reset_n, frameo_n, valido_n, dout);
  bind router_test_top.dut assert_outputs #(.port_no(4))
    a_oport4(clock, reset_n, frameo_n, valido_n, dout);
```

continued…

```
  bind router_test_top.dut assert_outputs #(.port_no(5))
    a_oport5(clock, reset_n, frameo_n, valido_n, dout);
  bind router_test_top.dut assert_outputs #(.port_no(6))
    a_oport6(clock, reset_n, frameo_n, valido_n, dout);
  bind router_test_top.dut assert_outputs #(.port_no(7))
    a_oport7(clock, reset_n, frameo_n, valido_n, dout);
  bind router_test_top.dut assert_outputs #(.port_no(8))
    a_oport8(clock, reset_n, frameo_n, valido_n, dout);
  bind router_test_top.dut assert_outputs #(.port_no(9))
    a_oport9(clock, reset_n, frameo_n, valido_n, dout);
  bind router_test_top.dut assert_outputs #(.port_no(10))
    a_oport10(clock, reset_n, frameo_n, valido_n, dout);
  bind router_test_top.dut assert_outputs #(.port_no(11))
    a_oport11(clock, reset_n, frameo_n, valido_n, dout);
  bind router_test_top.dut assert_outputs #(.port_no(12))
    a_oport12(clock, reset_n, frameo_n, valido_n, dout);
  bind router_test_top.dut assert_outputs #(.port_no(13))
    a_oport13(clock, reset_n, frameo_n, valido_n, dout);
  bind router_test_top.dut assert_outputs #(.port_no(14))
    a_oport14(clock, reset_n, frameo_n, valido_n, dout);
  bind router_test_top.dut assert_outputs #(.port_no(15))
    a_oport15(clock, reset_n, frameo_n, valido_n, dout);

endmodule //assert_binds
```

`assert.sva` Solution:

```
module assert_inputs(input clk, reset_n,
                     logic[15:0] frame_n, valid_n, din);
  parameter port_no = 0;

  property p_valid_during_pad(fr_n, vld_n, data);
    @(posedge clk) $fell(fr_n) |-> ##4 (vld_n && data) [*5];
  endproperty

  a_vld_hi_in_pad:
  assert property (
    disable iff(!reset_n)
    p_valid_during_pad(frame_n[port_no], valid_n[port_no], din[port_no])
  ) else begin
    $fatal(1,"[FATAL] %m: Pad protocol violation on port %0d at %t",
             port_no, $realtime);
  end

endmodule //assert_inputs
```

```
module assert_outputs(input clk, reset_n,
                      logic[15:0] frameo_n, valido_n, dout);
  parameter port_no = 0;

  int unsigned bit_cntr;

  dout_unknown_check:
  assert property (
    @(posedge clk)
    !valido_n[port_no] |-> (!$isunknown(dout[port_no]))
  );

  always@(posedge clk or negedge reset_n) begin: bit_counter
    if (!reset_n) bit_cntr = 0;
    else if (!valido_n[port_no]) bit_cntr++;
  end : bit_counter

  a_vld_lo_data_aligned_rsng_frm:
  assert property (
    @(posedge clk)
    disable iff (!reset_n) p_vld_lo_data_aligned_rsng_frm
  ) begin
      $display("[NOTE]%t End of frame successfully checked", $realtime);
      bit_cntr = 0;
    end else begin
      bit_cntr = 0;
      protocol_error_handler("Bit Alingment Error Found");
      $fatal;
    end

  property p_vld_lo_data_aligned_rsng_frm;
    s_frame |-> ##0 ((bit_cntr%8 == 7) && (!valido_n[port_no]));
  endproperty

  sequence s_frame;
    $fell(frameo_n[port_no]) ##1 $rose(frameo_n[port_no])[->1];
  endsequence

  function void protocol_error_handler(string msg);
    $display("[ERROR]%t Port %0d: %s", $realtime, port_no, msg);
  endfunction

endmodule //assert_outputs
```

3 Implement Assertion Coverage

Learning Objectives

After completing this lab, you should be able to:

- Use compile switches to turn on assertion coverage
- View coverage with DVE or Verdi
- Write cover properties
- Enhance cover properties with covergroups
- Bind assertion modules to the test program
- Create and read HTML coverage reports

Lab Duration:
30 minutes

Getting Started

In Lab 1 and 2 you implemented immediate and concurrent assertions in the Testbench and the DUT. Since functional coverage is an important part of any verification methodology, you will experiment with SVA coverage in this lab.

The DUT being monitored continues to be the 16x16 Router from the SVTB workshop.

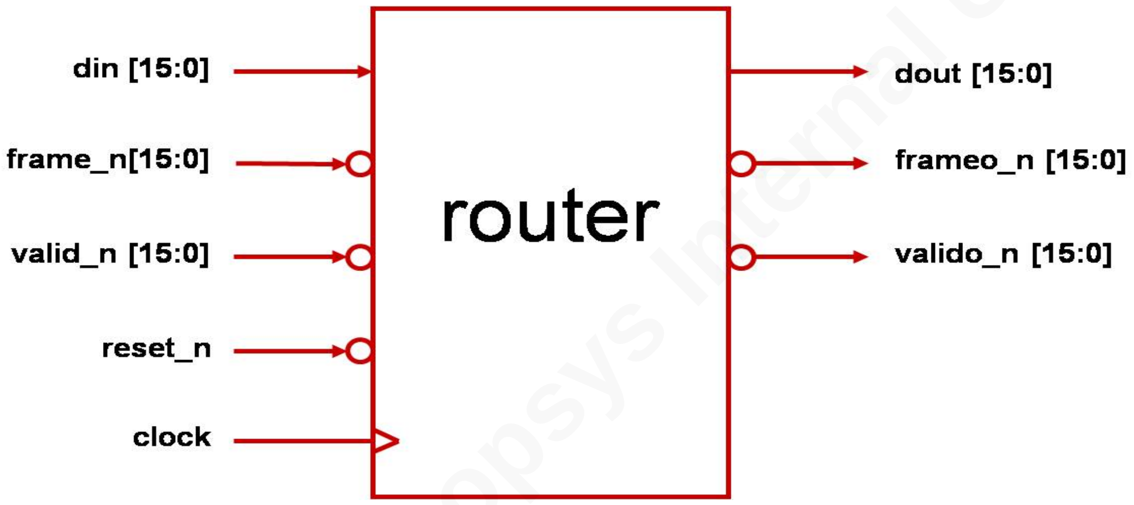


Figure 1. 16 X 16 Router

Overview

This lab flow steps you through viewing assertion coverage, developing and viewing covergroups embedded in assertions and defining cover properties.

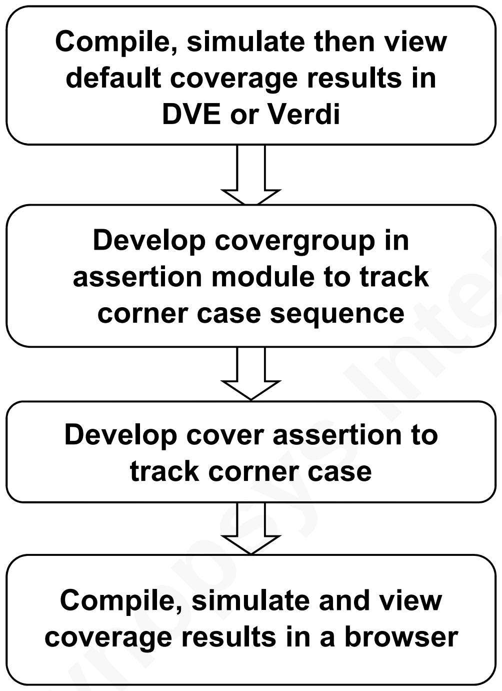

Compile, simulate then view default coverage results in DVE or Verdi
Develop covergroup in assertion module to track corner case sequence
Develop cover assertion to track corner case
Compile, simulate and view coverage results in a browser

Figure 2. Lab 3 Flow Diagram

Assertion Coverage

For this lab, you will continue to use the same DUT and the same testbench that was developed in the SVTB workshop.

Task 1. Go into Lab3 Working Directory

1. CD into **lab3** directory

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

Task 2. Turn On Assertion Coverage

All immediate and concurrent assertions can be tracked via Assertion Coverage. The required switch is **-cm assert** for both compile and run-time.

Try this switch on the assertions that you have already developed for lab2.

Note: If you chose to use your own lab files from lab2, type "**make my_copy**"

1. Compile and run simulation with the **-cm assert** switch:

```
> vcs -debug_access+all -R -f files -cm assert
```

(All other compile-time switches are embedded within **files**)

2. At the completion of simulation, use DVE or Verdi to view coverage.

```
> make dve_cov
```

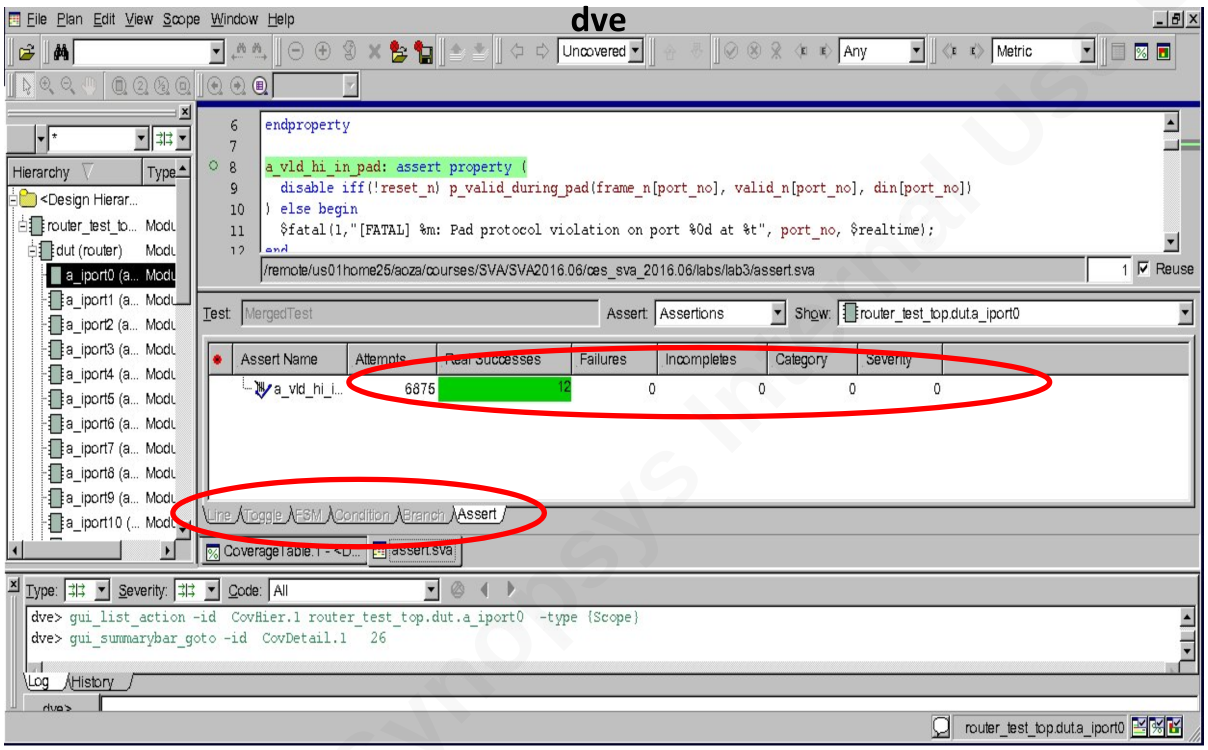
or

```
> make verdi_cov
```

These **Makefile** targets execute the following:

```
dve_cov:
   dve -cov -dir simv.vdb &

 verdi_cov
   verdi -cov -covdir simv.vdb
```

In the invoked dve or verdi window, you can navigate to see a summary of all the assertions including: **attempts**, **real successes**, **failures** and **incompletes**. You can navigate by module (all assertions in a module) or function (all assertions of same name in different modules). Play around to see the different views. Note the tabs for different types of coverage. We are interested in the Assert tab.


dve

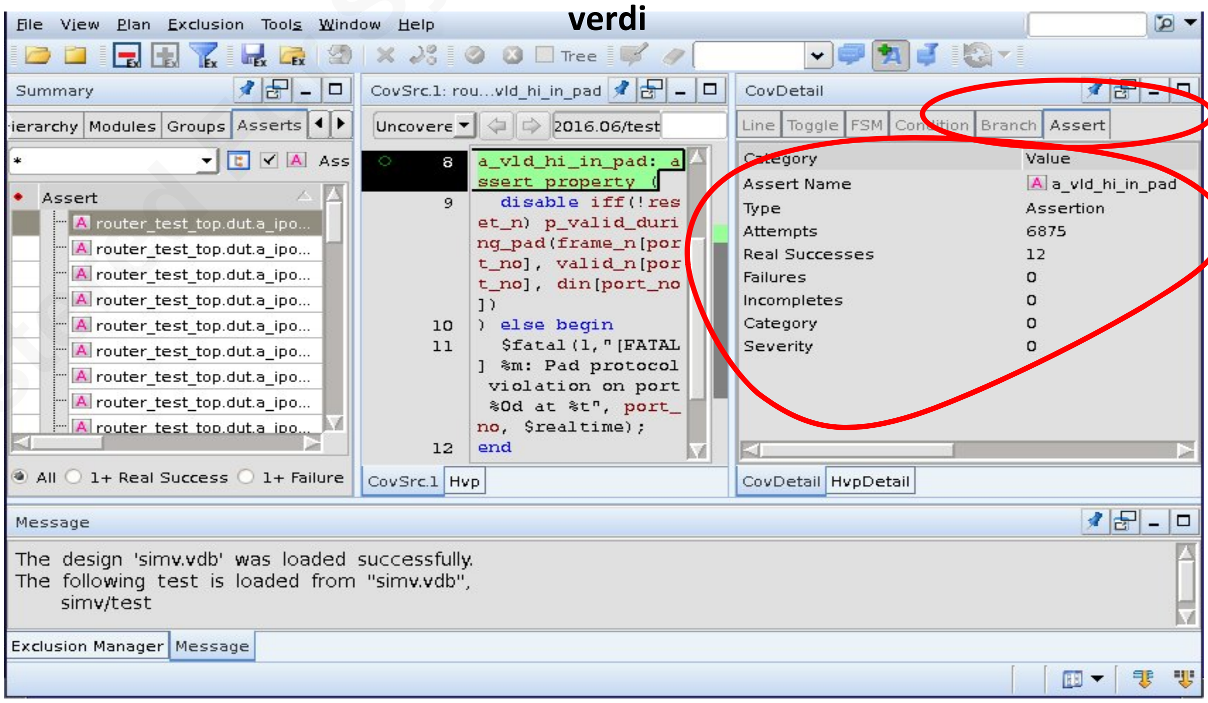
```
endproperty

a_vld_hi_in_pad: assert property (
  disable iff(!reset_n) p_valid_during_pad(frame_n[port_no], valid_n[port_no], din[port_no])
) else begin
  $fatal(1,"[FATAL] %m: Pad protocol violation on port %0d at %t", port_no, $realtime);
end
```

/remote/us01home25/aoza/courses/SVA/SVA2016.06/ces_sva_2016.06/labs/lab3/assert.sva

Test: MergedTest Assert: Assertions Show: router_test_top.dut.a_iport0

Assert Name	Attempts	Real Successes	Failures	Incompletes	Category	Severity
a_vld_hi_i...	6875	12	0	0	0	0

```
dve> gui_list_action -id CovHier.1 router_test_top.dut.a_iport0 -type {Scope}
dve> gui_summarybar_goto -id CovDetail.1 26
```


verdi

```
a_vld_hi_in_pad: assert property (
  disable iff(!reset_n) p_valid_during_pad(frame_n[port_no], valid_n[port_no], din[port_no])
) else begin
  $fatal(1,"[FATAL] %m: Pad protocol violation on port %0d at %t", port_no, $realtime);
end
```

Category	Value
Assert Name	a_vld_hi_in_pad
Type	Assertion
Attempts	6875
Real Successes	12
Failures	0
Incompletes	0
Category	0
Severity	0

```
The design 'simv.vdb' was loaded successfully.
The following test is loaded from "simv.vdb",
    simv/test
```

File View Plan Exclusion Tools Window Help
Tree
Summary
CovSrc.1: rou...vld_hi_in_pad
CovDetail
Hierarchy Modules Groups Asserts
Uncovered
2016.06/test
Line Toggle FSM Condition Branch Assert
Assert
router_test_top.dut.a_ipo...
router_test_top.dut.a_ipo...
router_test_top.dut.a_ipo...
router_test_top.dut.a_ipo...
router_test_top.dut.a_ipo...
router_test_top.dut.a_ipo...
router_test_top.dut.a_ipo...
router_test_top.dut.a_ipo...
router_test_top.dut.a_ipo...
All
1+ Real Success
1+ Failure
CovSrc.1
Hvp
CovDetail
HvpDetail
Message
Exclusion Manager
Message

This set of assertion coverage may be too coarse to be useful. To get a finer grain of assertion coverage, you may need to write a targeted cover property or embed covergroups within an assertion module.

Task 3. Develop Cover Assertion

On an as-needed basis, you can create cover properties to only track coverage.

In this task, you will develop a cover property to track the number of bytes per payload going through the DUT.

1. Open the **`assert.sva`** file in an editor.
2. In **`assert_outputs`** module create cover property called **`cover_bytes.`**

```
module assert_outputs(input clk, reset_n, logic[15:0] frameo_n,
                      valido_n, dout)

  cover_bytes: cover property (
  );
  …
endmodule //assert_outputs
```

3. Inside the property create a sequence for detecting a payload containing exactly two bytes.

```
cover_bytes: cover property(
  @(posedge clk)
  ($fell(frameo_n[port_no]) ##1 $rose(frameo_n[port_no]) [->1])
  intersect
  ($fell(frameo_n[port_no]) ##1 (!valido_n[port_no]) [->(8*2)])
);
```

4. Save and close the file.
5. Compile and simulate.

```
> make
```

6. Check the messages at the end of simulation in the simulation log file simv.log. You should have detected a few 2-byte packets. If not, you will need to debug your code.

7. Modify the cover property to cover 1 – 5 bytes as follows:

```
genvar i;
generate
for(i = 1; i <=5; i++) begin: cover_bytes
  cover property ( @(posedge clk)
  ($fell(frameo_n[port_no]) ##1 $rose(frameo_n[port_no])[->1])
  intersect
  ($fell(frameo_n[port_no]) ##1 (!valido_n[port_no]) [->(8*i)])
);
end
endgenerate
```

Make sure you change 2 to i

8. Save and close the file.

9. Compile and simulate.

```
> make
```

10. Check the message at the end of simulation.

```
> make
```

You should see an array of cover properties covering 1 – 5 bytes of payload. You should also see that 1- and 5-byte payload size coverage bins have no matches as shown below.

If not, you will need to debug your code.

```
…
"./assert.sva", 37:
router_test_top.dut.a_oport0.cover_bytes[4].unnamed$$_0
, 6875 attempts, 4 match
"./assert.sva", 37:
router_test_top.dut.a_oport0.cover_bytes[5].unnamed$$_0
, 6875 attempts, 0 match
```

Task 4. Create an Assertion Module Covergroup

Cover properties do not easily lend themselves to cover values when a sequence is defined over a range of values. In Task 3, to get around this limitation, you used the generate mechanism to create one cover property per required value.

If you already have a concurrent assertion looking at the protocol, an easier way to do this is to embed covergroups in the assertion module and use the existing concurrent assertion to update the covergroup bins.

In this task, you will add a covergroup to your assertion module.

1. Open the **`assert.sva`** file in an editor.
2. In **`assert_outputs`** module, create a covergroup to track the **`bit_cntr`** variable:

```
covergroup cov_packet_size ;
 num_bytes: coverpoint (bit_cntr/8) iff (bit_cntr%8 == 0) {
   bins num_bytes[] = {[1:5]};  //no of bytes in payload
   illegal_bins not_legal_size = default;
 }
 misaligned: coverpoint (bit_cntr/8) iff (bit_cntr%8);
endgroup
```

3. Immediately after the definition of the covergroup instantiate the covergroup.

```
cov_packet_size cov_pkt_sz = new();
```

Task 5. Use Assertion to Trigger Covergroup

1. Locate the **`a_vld_lo_data_aligned_rsng_frm`** assertion.
2. In the success action block, before the variable **`bit_cntr`** is set to 0, add a **`sample()`** call for the covergroup instantiated in the previous Task.
3. Save and close the file.

Task 6. Compile and Simulate

1. Compile and run your test.

```
> make
```

   Debug any compile errors.

Task 7. View Coverage Results

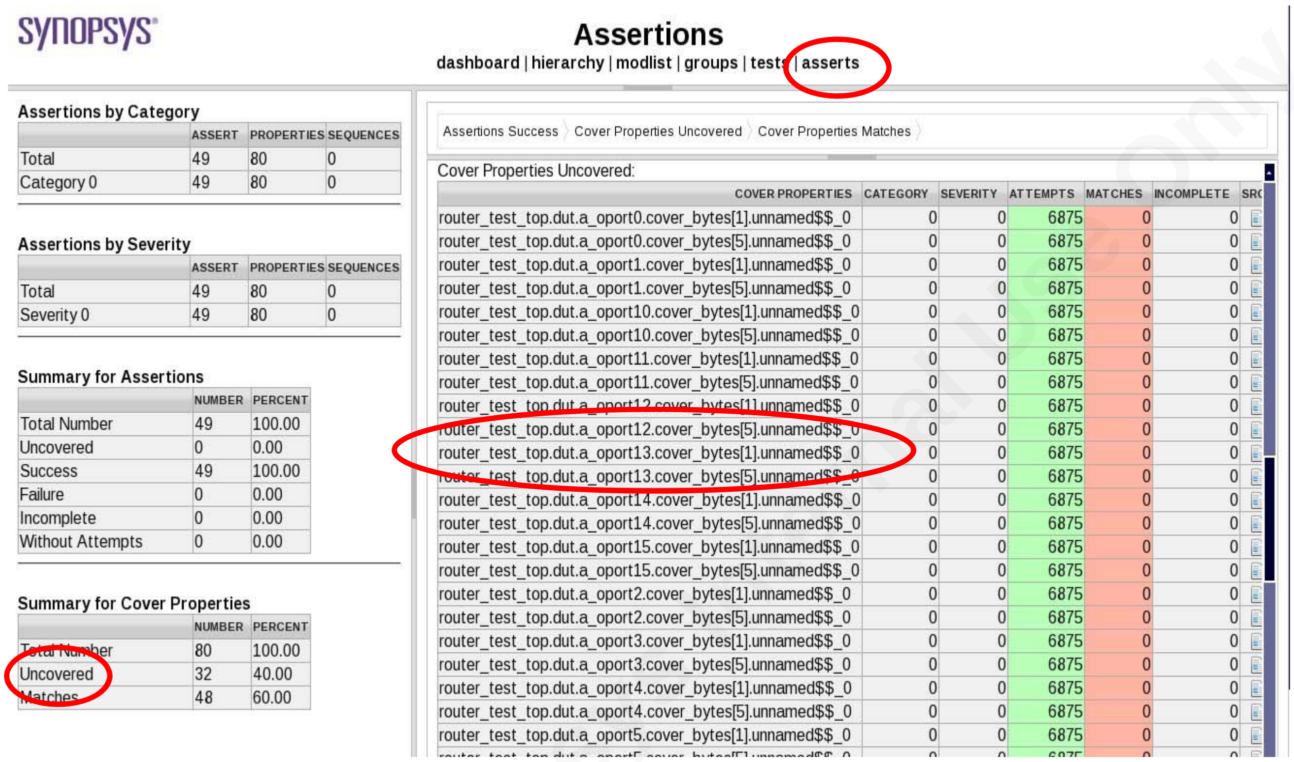
1. Format and view coverage results in a browser. Text reports are also generated.

```
> make urg
```

   The make target looks like:

```
urg:
 urg -dir simv.vdb
 urg -dir simv.vdb -format text
 /usr/bin/firefox urgReport/asserts.html &
```

The covergroup results should match the cover property results as seen below.



Assertions

dashboard | hierarchy | modlist | groups | tests | asserts

Assertions by Category

	ASSERT	PROPERTIES	SEQUENCES
Total	49	80	0
Category 0	49	80	0

Assertions by Severity

	ASSERT	PROPERTIES	SEQUENCES
Total	49	80	0
Severity 0	49	80	0

Summary for Assertions

	NUMBER	PERCENT
Total Number	49	100.00
Uncovered	0	0.00
Success	49	100.00
Failure	0	0.00
Incomplete	0	0.00
Without Attempts	0	0.00

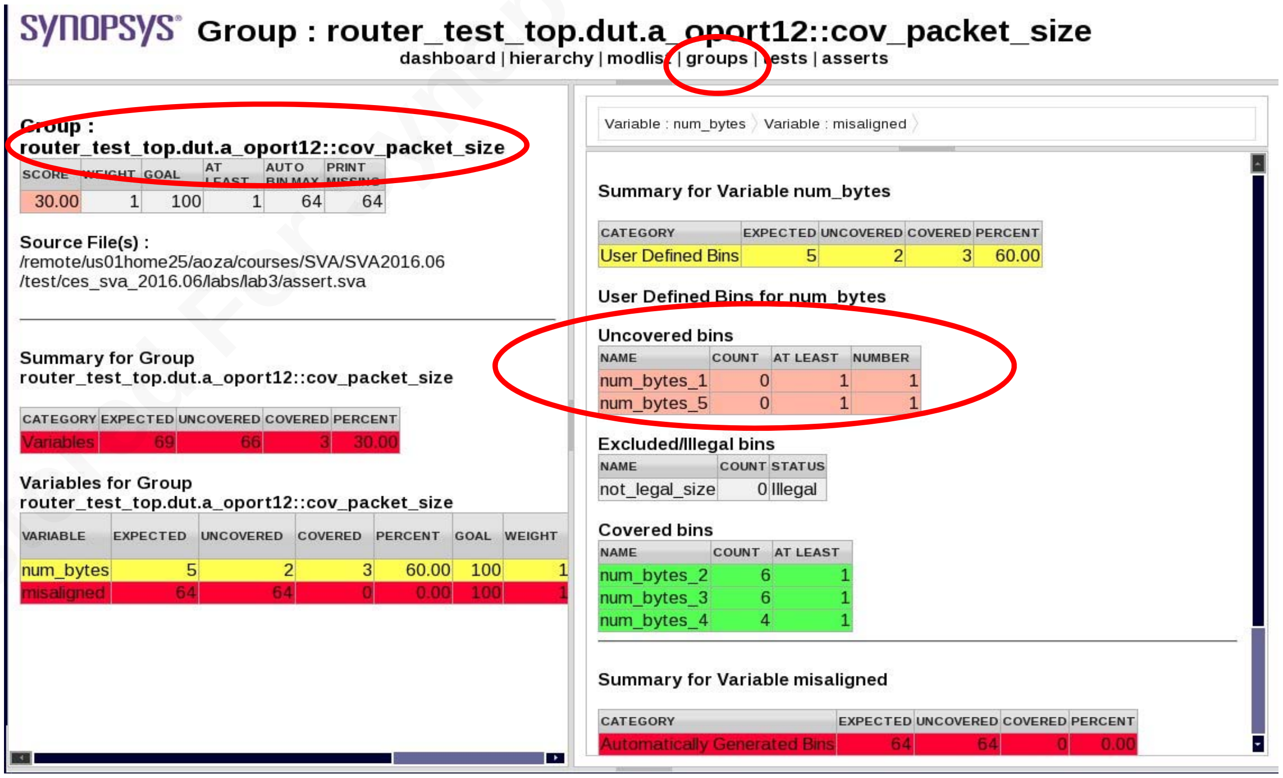
Summary for Cover Properties

	NUMBER	PERCENT
Total Number	80	100.00
Uncovered	32	40.00
Matches	48	60.00

Assertions Success 〉 Cover Properties Uncovered 〉 Cover Properties Matches 〉

Cover Properties Uncovered:

COVER PROPERTIES	CATEGORY	SEVERITY	ATTEMPTS	MATCHES	INCOMPLETE	SRC
router_test_top.dut.a_oport0.cover_bytes[1].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport0.cover_bytes[5].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport1.cover_bytes[1].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport1.cover_bytes[5].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport10.cover_bytes[1].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport10.cover_bytes[5].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport11.cover_bytes[1].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport11.cover_bytes[5].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport12.cover_bytes[1].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport12.cover_bytes[5].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport13.cover_bytes[1].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport13.cover_bytes[5].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport14.cover_bytes[1].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport14.cover_bytes[5].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport15.cover_bytes[1].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport15.cover_bytes[5].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport2.cover_bytes[1].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport2.cover_bytes[5].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport3.cover_bytes[1].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport3.cover_bytes[5].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport4.cover_bytes[1].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport4.cover_bytes[5].unnamed$$_0	0	0	6875	0	0	
router_test_top.dut.a_oport5.cover_bytes[1].unnamed$$_0	0	0	6875	0	0	



Group : router_test_top.dut.a_oport12::cov_packet_size

dashboard | hierarchy | modlist | groups | tests | asserts

Group : router_test_top.dut.a_oport12::cov_packet_size

SCORE	WEIGHT	GOAL	AT LEAST	AUTO BIN MAX	PRINT MISSING
30.00	1	100	1	64	64

Source File(s) :
/remote/us01home25/aoza/courses/SVA/SVA2016.06
/test/ces_sva_2016.06/labs/lab3/assert.sva

Summary for Group router_test_top.dut.a_oport12::cov_packet_size

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Variables	69	66	3	30.00

Variables for Group router_test_top.dut.a_oport12::cov_packet_size

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT
num_bytes	5	2	3	60.00	100	1
misaligned	64	64	0	0.00	100	1

Variable : num_bytes 〉 Variable : misaligned 〉

Summary for Variable num_bytes

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	5	2	3	60.00

User Defined Bins for num_bytes

Uncovered bins

NAME	COUNT	AT LEAST	NUMBER
num_bytes_1	0	1	1
num_bytes_5	0	1	1

Excluded/Illegal bins

NAME	COUNT	STATUS
not_legal_size	0	Illegal

Covered bins

NAME	COUNT	AT LEAST
num_bytes_2	6	1
num_bytes_3	6	1
num_bytes_4	4	1

Summary for Variable misaligned

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Automatically Generated Bins	64	64	0	0.00

Congratulations! You have completed Lab 3.

Answers / Solutions

assert.sva Solution:

```
module assert_inputs(input clk, reset_n,
                     logic[15:0] frame_n, valid_n, din);
parameter port_no = 0;

property p_valid_during_pad(fr_n, vld_n, data);
  @(posedge clk) $fell(fr_n) |-> ##4 (vld_n && data) [*5];
endproperty

a_vld_hi_in_pad:
assert property (
  disable iff(!reset_n)
  p_valid_during_pad(frame_n[port_no], valid_n[port_no], din[port_no])
) else begin
  $fatal(1,"[FATAL] %m: Pad protocol violation on port %0d at %t",
            port_no, $realtime);
end

endmodule //assert_inputs

module assert_outputs(input clk, reset_n,
                      logic[15:0] frameo_n, valido_n, dout);

parameter port_no = 0;
int unsigned bit_cntr;

covergroup cov_packet_size;
  num_bytes: coverpoint (bit_cntr/8) iff ((bit_cntr%8) == 0) {
    bins num_bytes[] = {[1:5]};
    illegal_bins not_legal_size = default;
  }
  misaligned: coverpoint (bit_cntr/8) iff (bit_cntr%8);
endgroup

cov_packet_size cov_pkt_sz = new();

dout_unknown_check:
assert property (
  @(posedge clk)
  !valido_n[port_no] |-> (!$isunknown(dout[port_no]))
);
```

```
for(genvar i=1; i<=5; i++) begin: cover_bytes
     cover property (
       @(posedge clk)
       ($fell(frameo_n[port_no]) ##1 $rose(frameo_n[port_no]) [->1])
       intersect
       ($fell(frameo_n[port_no]) ##1 (!valido_n[port_no]) [->(8*i)])
     );
end

always@(posedge clk or negedge reset_n) begin: bit_counter
  if (!reset_n) bit_cntr = 0;
  else if (!valido_n[port_no]) bit_cntr++;
end : bit_counter

a_vld_lo_data_aligned_rsng_frm:
assert property (
  @(posedge clk)
  disable iff (!reset_n) p_vld_lo_data_aligned_rsng_frm
) begin
    cov_pkt_sz.sample();
    $display("[NOTE]%t End of frame successfully checked", $realtime);
    bit_cntr = 0;
  end else begin
    bit_cntr = 0;
    protocol_error_handler("Bit Alingment Error Found");
    $fatal;
  end

property p_vld_lo_data_aligned_rsng_frm;
  s_frame |-> ##0 ((bit_cntr%8 == 7) && (!valido_n[port_no]));
endproperty

sequence s_frame;
  $fell(frameo_n[port_no]) ##1 $rose(frameo_n[port_no])[->1];
endsequence

function void protocol_error_handler(string msg);
  $display("[ERROR]%t Port %0d: %s", $realtime, port_no, msg);
endfunction

endmodule //assert_outputs
```

This page was intentionally left blank.

4

Implement Assertions Using SVA Library

Learning Objectives

After completing this lab, you should be able to:

- Use Assertion Libraries
- Bind assertion library modules to DUT
- Debug error found with Assertion Libraries

Lab Duration: 30 minutes

Getting Started

In Labs 1, 2 and 3 you wrote your own assertion code. This works well when the protocol you are tracking is non-standard.

If the protocol you are tracking is a common standard, you may be able to use assertions already embedded in the SVA libraries that ship with VCS.

In this lab you will use the arbiter assertion module (**`assert_arbiter.sv`**) in **`${VCS_HOME}/packages/sva_cg`**.

The DUT being monitored continues to be the 16x16 Router from the SVTB workshop. The test bench however now uses the SVTB lab5 solution.

Within this DUT, there is a round-robin arbiter per output port. You will use the arbiter assertion module in the VCS SVA library to check for the correct operation of all 16 arbiters.

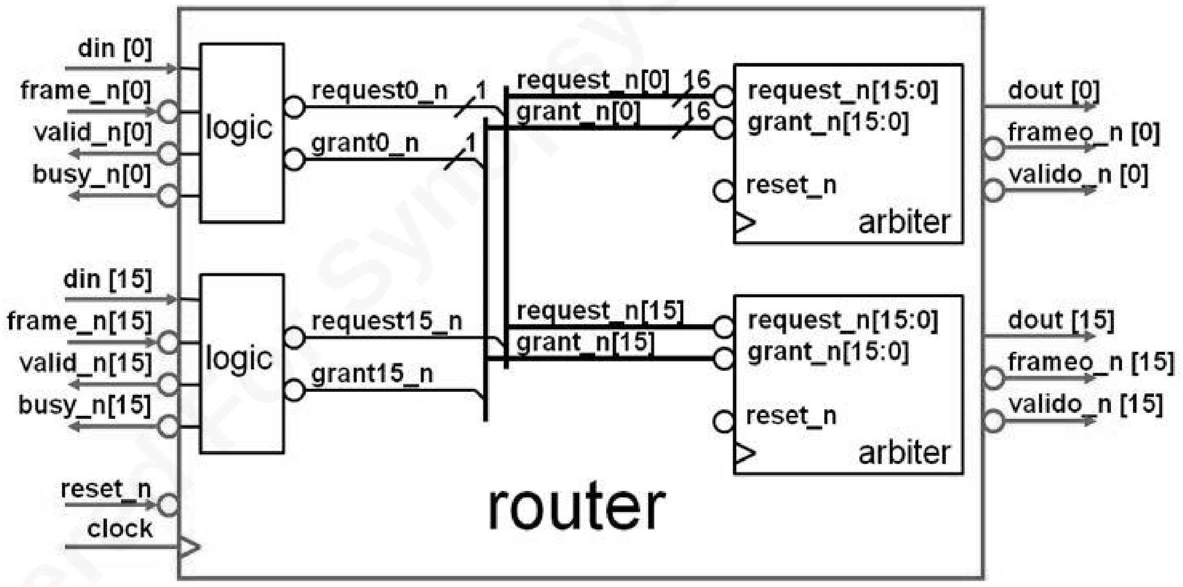


Figure 1. 16 X 16 Router

Overview

This lab flow steps you through using a checker from the assertion library provided by Synopsys. It simulates an injected design protocol error being caught by the checker.

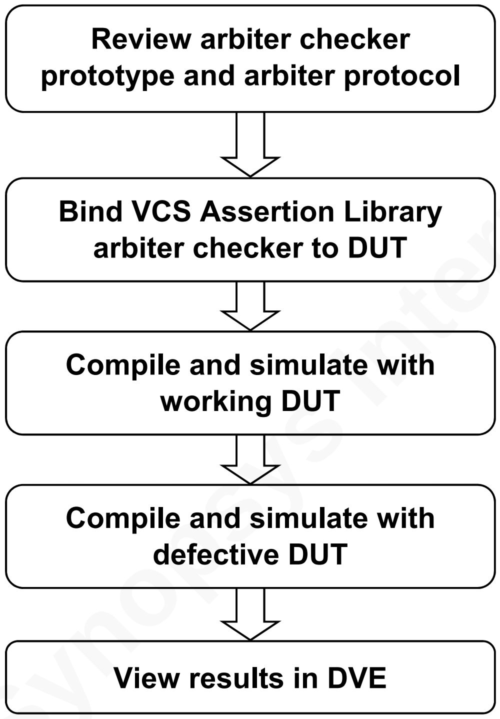

Review arbiter checker prototype and arbiter protocol
Bind VCS Assertion Library arbiter checker to DUT
Compile and simulate with working DUT
Compile and simulate with defective DUT
View results in DVE

Figure 2. Lab 4 Flow Diagram

Assertion Library

In this lab you will use an arbiter assertion module from the VCS SVA library.

This lab uses the same 16x16 router you used in the other labs as the device under test. For this test there are 16 drivers and 16 receivers. Each path of the router is driven. The test generates a random data packet for a randomly selected source and destination pair. Two packets could have the same destination port. Each output channel has a round-robin arbiter that arbitrates requests for that output. You will check the arbiter design for correctness using a checker from the SVA library.

Task 1. Go into Lab 4 Working Directory

1. CD into **`lab4`** directory

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

Task 2. Aribiter Protocol Review

The arbiter assertion module (**`assert_arbiter.sv`**) you will use to verify the correctness of the DUT arbiter resides in **`${VCS_HOME}/packages/sva_cg`**, where **`${VCS_HOME}`** is the install directory of the vcs binaries.

This lab will only focus on a few of the assertion module parameters and ports: **`no_chnl`**, **`arbitration_rule`**, **`reqs`** and **`grants`**. The rest will be left at default. If interested, you can open the file and examine how to control the other parameters and ports.

Prototype:

```
assert_arbiter [#(severity_level, no_chnl, bw_prio, grant_one_chk,
                  req_priority_chk, arbitration_rule, min_lat, max_lat,
                  edge_expr, msg, category, coverage_level_1,
                  coverage_level_2, coverage_level_3, property_type)]
                  inst_name (clk, reset_n, reqs, req_priority, grants);
```

```
reqs and grants are vectors of size [no_chnl-1:0] where the bits
correspond to the corresponding channels in reqs and grants.

arbitration_rule = 0 - none of the rules is checked
arbitration_rule = 1 - fairness (round-robin)
arbitration_rule = 2 – fifo
arbitration_rule = 3 - LRU
```

Protocol:

```
A grant is expected to be one clock cycle wide.

It is assumed that a request holds asserted until
granted. It is assumed that a request is removed on the
clock tick the grant is sampled.
```

Task 3. Bind Arbiter Checker to DUT Port 0

1. Open the **`assert.sva`** file in an editor.

This file contains an empty module that you need to fill in with bindings.

There are 16 arbiters in the DUT. In this lab, you will bind each of 16 arbiters individually to an instance of the Assertion Library module **`assert_arbiter`**.

To make sure that the Assertion Library module is used correctly, bind only port 0 arbiter to an instance of the library module as a starting point.

2. Bind an instance of **`assert_arbiter`** assertion module to port 0 arbiter:

```
bind router_test_top.dut assert_arbiter
    #(.no_chnl(16), .arbitration_rule(1)) arbiter
     (clock, reset_n, ~request_n[0], , ~grant_n[0]);
```

- There are 16 channels through the DUT, therefore the **`.no_chnl`** parameter is set to 16
- The arbitration scheme in the DUT is round robin, therefore the **`.arbitration_rule`** parameter is set to 1
- The assertion libarary module expects the request and grant signals to be high true, but the DUT implements low true, therefore these signals are inverted as they are connected to the library module

3. Save and close the file.

Task 4. Compile and Simulate with Good and Bad DUT

1. Compile and simulate.

```
> make
```

Simulation should complete without errors.

2. To make sure that the assertion can catch errors, try the following:

```
> make bad
```

3. Open DVE or Verdi^3 and examine the failure (use the techniques you learned in lab 1).

You should see violations of the following requirement:

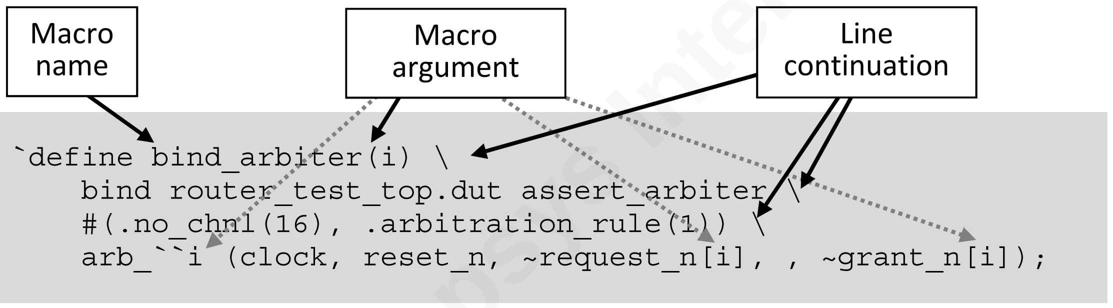
```
It is assumed that a request holds asserted until granted.
```

Task 5. Expand to Cover All Ports

There are different ways of doing binding when you need multiple binds of the same assertion module. In this lab, you will make use the Verilog marcro mechanism.

1. Open the **`assert.sva`** file in an editor

2. Change the bind statement to a macro definition:



```
`define bind_arbiter(i) \
    bind router_test_top.dut assert_arbiter \
    #(.no_chnl(16), .arbitration_rule(1)) \
    arb_``i (clock, reset_n, ~request_n[i], , ~grant_n[i]);
```

3. Create 16 individual bindings using the macro:

```
`bind_arbiter(0)
`bind_arbiter(1)
 ...
`bind_arbiter(15)
```

4. Save and close the file.

5. Compile and simulate with good DUT.

```
> make
```

Simulation should complete without errors.

6. Try the bad DUT:

```
> make bad
```

You should see failures at every port. If you are interested, you can open dve and examine each of the errors. Or, look for the string "VIOLATION" in simv.log file.

Congratulations! You have completed Lab 4! You are done with all the labs.

Answers / Solutions

`assert.sva` Solution:

```
module assert_router_arbiter(input logic clock, reset_n,
                         input logic [15:0] request, grant);

`define bind_arbiter(i) \
    bind router_test_top.dut assert_arbiter \
    #(.no_chnl(16), .arbitration_rule(1)) \
    arb_``i (clock, reset_n, ~request_n[i],, ~grant_n[i]);

`bind_arbiter(0)
`bind_arbiter(1)
`bind_arbiter(2)
`bind_arbiter(3)
`bind_arbiter(4)
`bind_arbiter(5)
`bind_arbiter(6)
`bind_arbiter(7)
`bind_arbiter(8)
`bind_arbiter(9)
`bind_arbiter(10)
`bind_arbiter(11)
`bind_arbiter(12)
`bind_arbiter(13)
`bind_arbiter(14)
`bind_arbiter(15)

endmodule
```

This page is left blank intentionally.