
# QuickStart Workshop

# System Verilog for Verification –

# Lab Instructions

July 29, 2025

# Labs

- Lab 1: Data Structures
- Lab 2: Tasks and Interfaces
- Lab 3: Multi-threading
- Lab 4: Classes
- Lab 5: Inheritance
- Lab 6: Randomization
- Lab 7: Coverage
- Lab 8: Simple Assertions
- Lab 9: UART Sequences
- Lab 10: UART Transmit
- Appendix: Sample Solutions

# Lab Environment

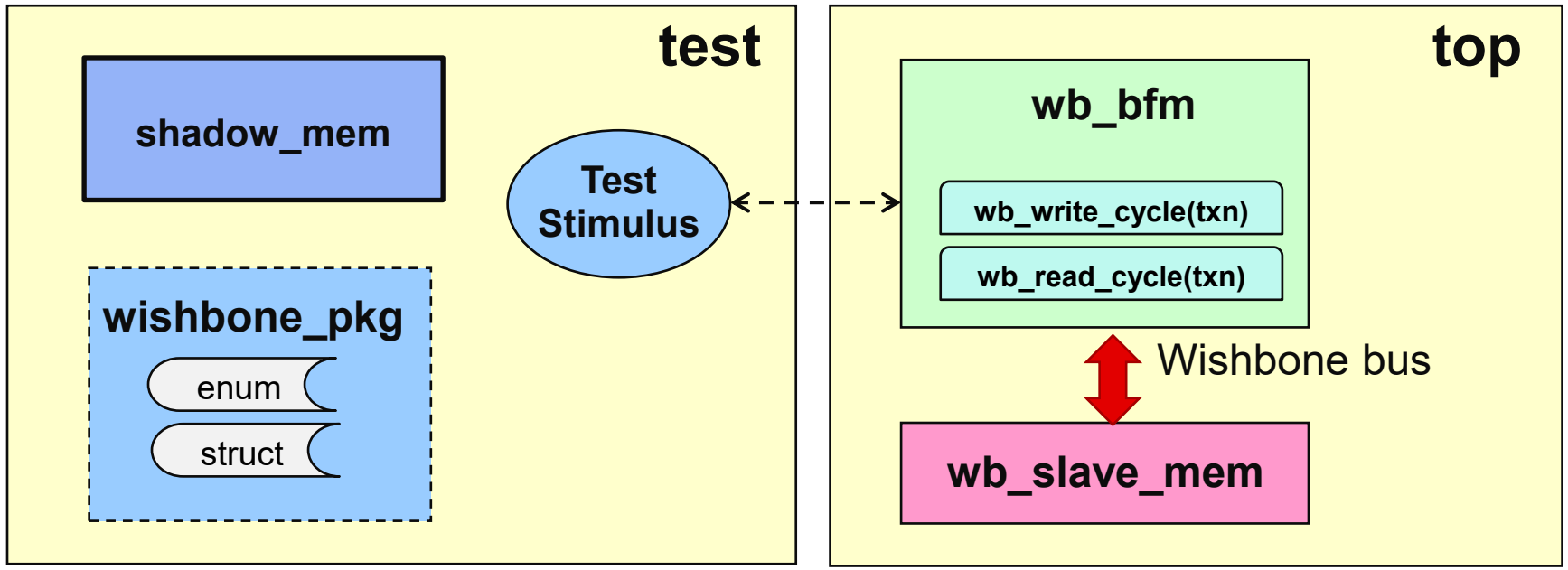
- Siemens Questa or Synopsys VCS

# Lab 1:
# Data Structures

2RTP

# Lab 1 – Data Structures: Overview

- The **`wb_bfm`** is a BFM style interface (more later) which has 2 tasks for generating Wishbone bus transactions to read and write the slave memory
- Dual top-level modules, one for testbench, the other for DUT (more later)
- A package for useful types we will use in our code
- A behavioral model of the memory for heuristic testing
- All stimulus in a single initial block



# Lab 1 – Data Structures: Instructions – 1

Working directory: **data_structures**

1. Edit the file ***wishbone_pkg.sv***
   - Create a package named **`wishbone_pkg`**
   - Declare two data types:
     - An enumerated type named **`wb_txn_t`** reflecting Wishbone operations:
       - **`NONE`** **`WRITE`** **`READ`** **`RMW`** **`WAIT_IRQ`**
     - A structure type named **`wb_txn`** to represent a Wishbone operation with the following fields:
       - **`txn_type`** (type is **`wb_txn_t`**)
       - **`adr`** (type is **`bit[31:0]`**)
       - **`data`** (type is **`logic[31:0]`**)
       - **`count`** (type is un-signed **`int`**)
       - **`byte_sel`** (type is **`bit[7:0]`**)

- Continued on next page -

# Lab 1 – Data Structures: Instructions – 2

Working directory: **data_structures**

2. Edit the the file **`test.sv`**
   - Import everything from the package **`wishbone_pkg`**
   - Declare an associative array **`shadow_mem`** to “shadow” the wishbone memory and be compared with it as the testbench runs
     - Both the array type and index type are **`bit[31:0]`**
   - In the initial block:
     - Declare an instance of your **`wb_txn`** struct called **`txn`**, and 32-bit variables for address and data
     - Wait for the reset signal (**`rst`**) on the wishbone bus interface (`top.wb_bfm`) to go low
       - Note that the Wishbone bus interface is inside the module top so we will do a hierarchical reference to it

- Continued on next page -

# Lab 1 – Data Structures: Instructions – 3

Working directory: **data_structures**

3. Notice the existing code which writes to memory 10 times

```
. . .
address = 0;
for(int i=0; i <10; i++) begin
  // setup txn for write
  txn.adr = address;
  txn.data = i;
  txn.txn_type = WRITE;
  txn.count = 1;
  txn.byte_sel[3:0] = 4'b1111;
  // Write Wishbone mem
  top.wb_bfm.wb_write_cycle(txn);
  shadow_mem[address] = i;  // write shadow
  // increment address
  address += 4;
end
...
```

- Starts at address 0 with ascending data values also starting at value 0
- Note that the Wishbone bus is byte addressable yet the interface to the bus is writing/reading 4 bytes (32 bits) at a time
  - Means the lower 2 bits of the address must be **`2'b00`**
  - Addresses increment by 4: **`'h0, 'h4, 'h8 'hc, 'h10, 'h14`** etc.
- Each write uses a wishbone transaction struct (**`txn`** of type **`wb_txn`**)
  - Using **`1`** for the `count`
  - Using **`4'b1111`** for the `byte_sel` value
- Writes to the wishbone memory are done by calling the write bus task ( **`wb_write_cycle(txn)`** where **`txn`** is the struct variable) on the Wishbone bus interface (**`top.wb_bfm`**)
- Writes the data to the shadow memory at the same address for checking purposes.

- Continued on next page -

# Lab 1 – Data Structures: Instructions – 4

Working directory: **data_structures**

4. After the memory writes, add code to check that the data in the Wishbone memory matches the data in the shadow memory

   - Loop through the shadow memory (hint: use **`foreach`**) and do a Wishbone bus read operation of the wishbone bus memory (use the **`wb_read_cycle(txn)`** task of **`top.wb_bfm`**) for each entry in the shadow memory

   - Check that the data observed from the bus matches the expected data in the shadow memory
     - Display a pass or fail message for each entry

   - The Wishbone read operation will read the data at the value of the **`adr`** field of **`txn`** and place the data read into the **`data`** field of **`txn`**

   - Compile and run

     **make**

# Lab 1 – Data Structures: Sample Output

Sample output:

```
# Memory passed at address 00000000
# Memory passed at address 00000004
# Memory passed at address 00000008
# Memory passed at address 0000000c
# Memory passed at address 00000010
# Memory passed at address 00000014
# Memory passed at address 00000018
# Memory passed at address 0000001c
# Memory passed at address 00000020
# Memory passed at address 00000024
# ** Note: $finish    : test.sv(55)
#    Time: 1310 ns  Iteration: 1  Instance: /test
```

# Lab 1 - Data Structures - conclusion

- Good
  - Using a package to hold common parameters and types
- Bad
  - Totally linear code in a single procedural block
    - **More complex DUTs will require pipelining (simultaneous in/out)**

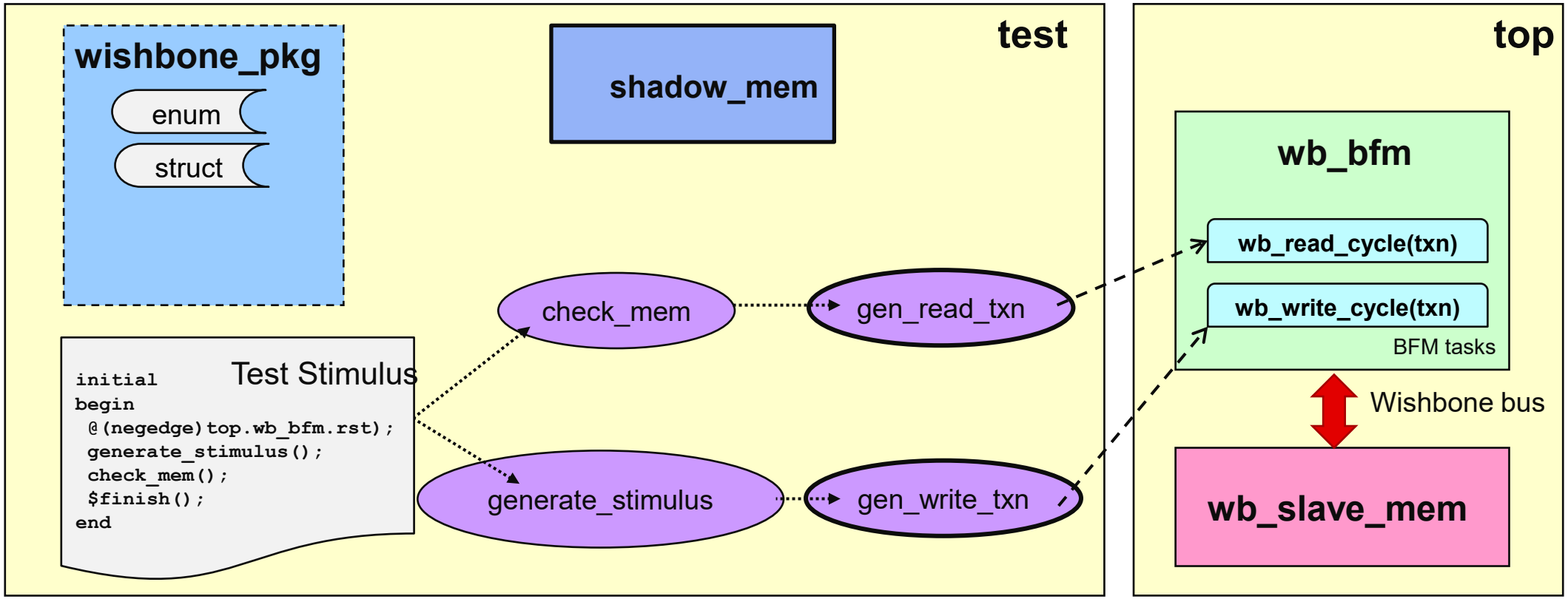
We can do better !

# Lab 2:
# Tasks and Interfaces

2RTP

# Lab 2 – Tasks & Interfaces: Overview

- Objective:
  - Create tasks  to read and write the slave memory calling the tasks in the BFM



# Lab 2 – Tasks and Interfaces: Instructions – 1

- Working directory: **tasks**

1. Edit file **`test.sv`**
   - Write a task named **`gen_write_txn`** to perform a wishbone write operation
     - Takes two input arguments (32-bit) for address and data
     - Generates a Wishbone bus write cycle
       - Call **`wb_write_cycle()`** task of the wishbone bus interface
         - Look at the task in the **`wishbone_bus_syscon_if`** interface for the arguments to the task
         - Use a hierarchical reference to access the Wishbone bus interface inside of **`top`**
     - Write the shadow memory (**`shadow_mem`**) with the data written to the Wishbone bus

# Lab 2 – Tasks and Interfaces: Instructions – 2

2. Write a task named **`gen_read_txn`** to perform a wishbone read operation
   - Takes one input argument (32-bit) for address and one reference argument (32-bit) for data
   - Generates a Wishbone bus read cycle
     - Calls the **`wb_read_cycle()`** task of the wishbone bus interface (called **`wishbone_bus_syscon_if`** )
       - Look at the task in the interface to get the arguments to the task
       - Use a hierarchical reference to access the Wishbone bus interface inside of **`top`**
     - Returns the data from the Wishbone bus read through the ref argument
       - (Hint) remember the requirement for a reference argument??
3. Compile/run

   **make**

# Lab 2 – Tasks and Interfaces: Sample Output

Sample output:

```
# Memory passed at address 00000000
# Memory passed at address 00000004
# Memory passed at address 00000008
# Memory passed at address 0000000c
# Memory passed at address 00000010
# Memory passed at address 00000014
# Memory passed at address 00000018
# Memory passed at address 0000001c
# Memory passed at address 00000020
# Memory passed at address 00000024
```

# Lab 2 – Tasks & Interfaces Conclusions

- Good
  - At the top level, divided into two modules, **test** and **top**
    - Good for many reasons, but especially as we add OOP shortly
  - Interface contains everything related to the I/O of the DUT
    - Including BFM-style methods to write/read the DUT
    - Supports reuse

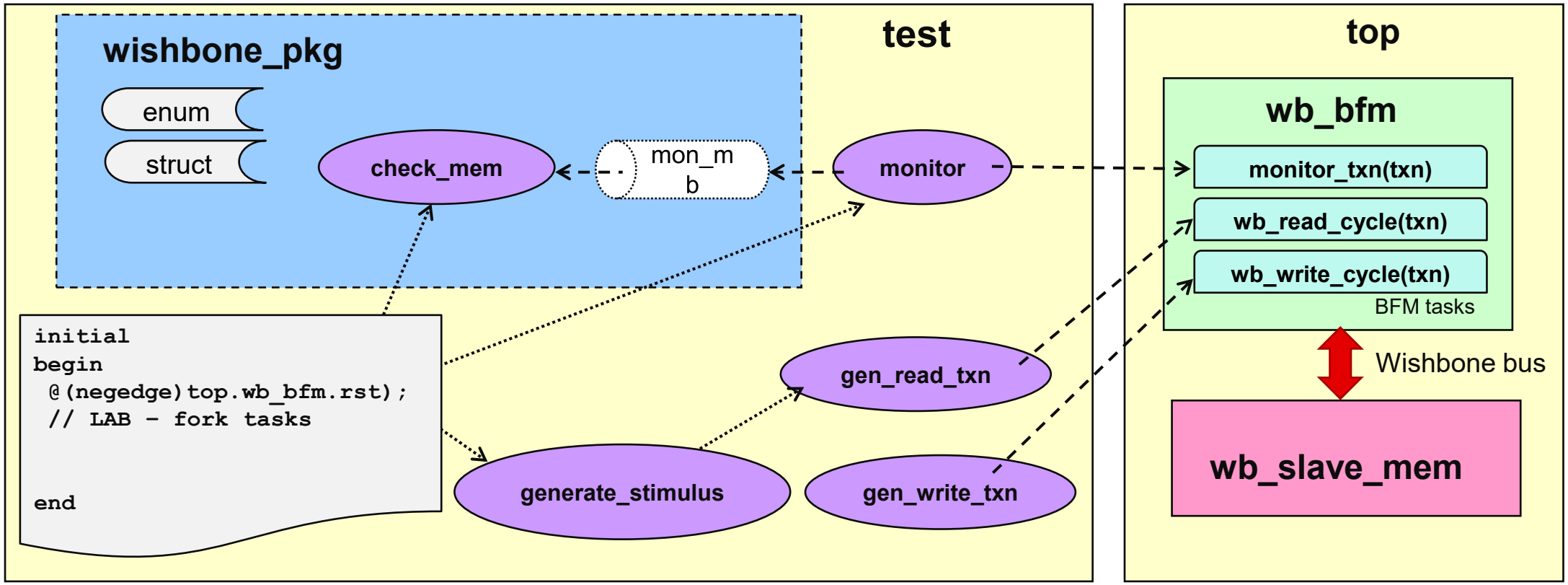
- Bad
  - Accessing the interface via hierarchical reference

# Lab 3: Multi-threading

2RTP

# Lab 3 – Multi-threading: Overview

Working directory: **concurrency**



# Lab 3 – Multi-threading: Instructions – 1

Working directory: **concurrency**

1. Edit the file ***wishbone_pkg.sv***
   - Declare a mailbox named **`mon_mb`**, parameterized on **`wb_txn`**
   - Declare an **`int`** named **`in_flight`** and initialize to 0
     - This variable keeps track of how many outstanding transactions there are
   - Add code to the **`check_mem()`** task
     - Get a transaction (**`txn`**) from the **`mon_mb`** mailbox and decrement **`in_flight`** variable
2. Edit the file ***test.sv***
   - Write a task called **`monitor`**
     - Add a forever loop:
       - Get a transaction from the the Wishbone bus Interface by calling its **`monitor_txn`** task
       - Put the received transaction into the **`mon_mb`** mailbox
   - In the initial block, after the reset is over
     - Spawn the **`generate_stimulus`**, **`check_mem`**, and **`monitor`** tasks in parallel with a **`fork-join_none`**
3. Compile and run

- Continued on next page -

# Lab 3 – Multi-threading: Instructions – 2

Sample output:

```
# Memory passed at address 00000000
# Memory passed at address 00000004
# Memory passed at address 00000008
# Memory passed at address 0000000c
# Memory passed at address 00000010
# Memory passed at address 00000014
# Memory passed at address 00000018
# Memory passed at address 0000001c
# Memory passed at address 00000020
# Memory passed at address 00000024
```

# Lab 3 - Multi-threading Lab - Conclusions

- Good
  - Refactoring code into tasks/functions is good practice
    - Aids reuse and understanding
  - Multi-threading is useful for system-level testing
    - Sometimes hard to stop the testbench though, hence **in_flight** variable
  - Mailboxes are an excellent way to synchronize threads

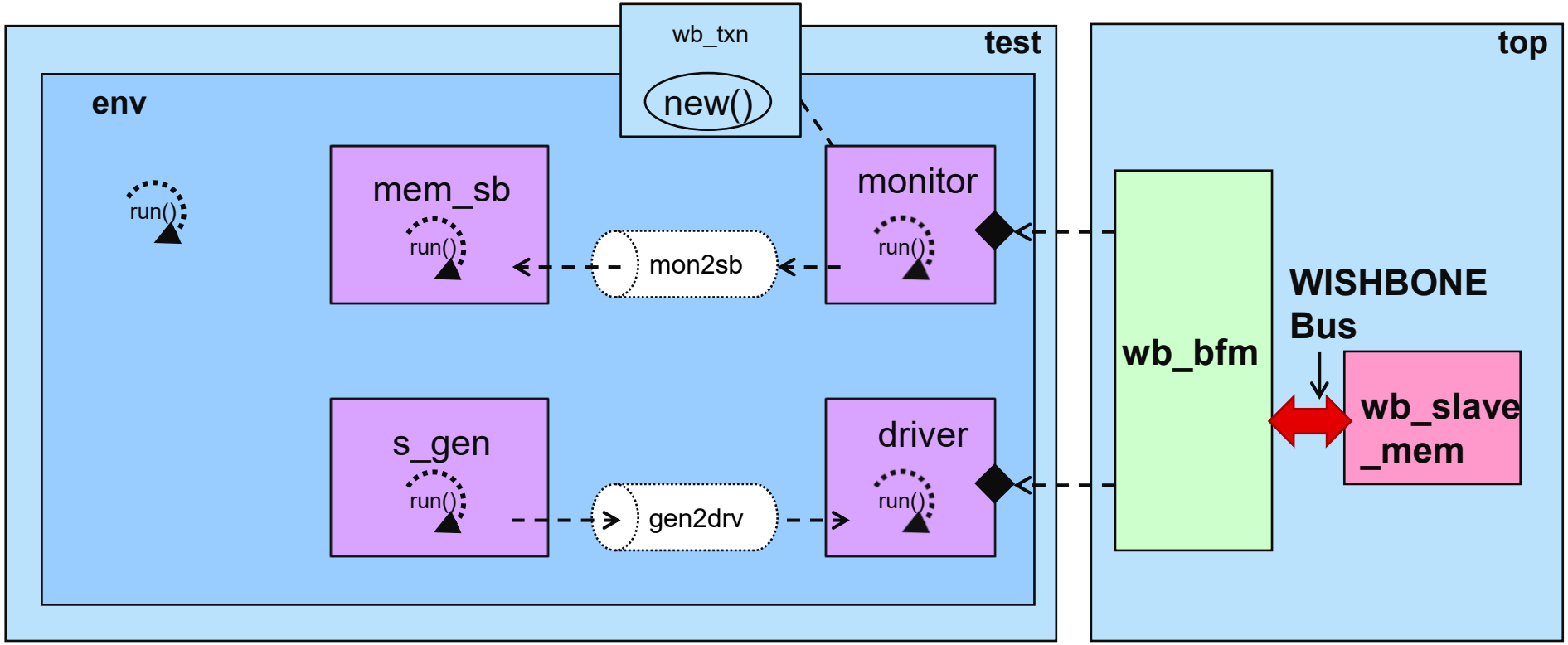
- Bad
  - Testbench is STILL pretty low level code
    - Accessing the interface via hierarchical reference ( `top.a_bfm_if.write()` )
    - Need to move to a higher level of abstraction for efficiency and to support:
  - Reuse reuse reuse… this testbench is still not very reusable

# Lab 4: Classes

2RTP

# Lab 4 – Classes: Overview

- Lab objective is to write the environment class and then create and run the environment in the test module



# Lab 4 – Classes: Instructions – 1

Working directory: **classes**

1. Edit the file **`env/wb_env.svh`**
   - Declare a class named **`wb_env`**
   - Declare 2 mailbox handles of type **`wb_txn`** named **`gen2drv, mon2sb`**
   - Declare handles for types **`stim_gen, wb_bfm_driver, wb_bfm_monitor`** and **`mem_checker`**
     - IMPORTANT! name the handles per the lab block diagram
   - Write a function called **`build()`** with a return type of **`void`**
     - Create objects for the handles you just declared
     - NOTE: declare **`gen2drv`** with a depth of 1 (or else sim won't run correctly)

# Lab 4 – Classes: Instructions – 2

Working directory: **classes**

2. Continue editing the file **`env/wb_env.svh`**
   - Write a function called **`connect()`** with a return type of **`void`**
     - Connect up the components to the mailboxes
       - Assign the **`gen2drv`** and **`mon2sb`** mailbox handles to the mailbox handles in the components per the lab block diagram
         - Look inside the components for a *method* to call to set the mailbox handles
     - Connect up the **`driver`** and **`monitor`** virtual interface properties using the **`v_wb_bfm`** property of the **`wishbone_pkg`**
       - Look inside the components for a method to call to set the virtual interfaces
   - Write a **`run()`** task that
     - Forks (using **`fork-join_none`**) the the **`run()`** tasks in the **`driver`**, **`monitor`**, and **`mem_sb`**
     - Calls the **`run()`** task in the **`s_gen`** (after the **`fork-join_none`** of the other **`run()`** processes)
   - Write a **`report()`** function with a return type of **`void`**
     - Calls the **`report()`** method of the **`mem_sb`**

# Lab 4 – Classes: Instructions – 3

3. Edit the file **`top_modules/test.sv`**
   - Declare a **`wb_env`** handle in the module
   - In the initial block:
     - Create the **`wb_env`** object
     - After the wait for the wishbone bus reset, call the wb_env **`build()`, `connect()`, `run()`, `report()`** methods in the order listed
4. Compile and run

   make

# Lab 4 – Classes: Sample Output

```
# WB_MEM_SLAVE:
#         INFO: WISHBONE Slave Memory instantiated
#         WISHBONE bus 0
#         WISHBONE bus slave  0
#         Memory Size = 1048576 bytes
#         Address range(hex) is from 00000000 to 000fffff
#         Memory is word addressable only - lower 2 address bits are ignored
#
# MEM_CHECK:  Memory passed at address 000b41d1
# MEM_CHECK:  Memory passed at address 00018c2a
# MEM_CHECK:  Memory passed at address 00008a40
# MEM_CHECK:  Memory passed at address 0007313d
# MEM_CHECK:  Memory passed at address 000266cb
# MEM_CHECK:  Memory passed at address 0005de73
# MEM_CHECK:  Memory passed at address 000cf783
# MEM_CHECK:  Memory passed at address 000400cc
# MEM_CHECK:  Memory passed at address 00010df9
# MEM_CHECK:  Memory passed at address 000e2d94
#
# ********************************
#   total transactions: 20
#   total errors:       0
# ********************************
```

# Lab 4 - Conclusions

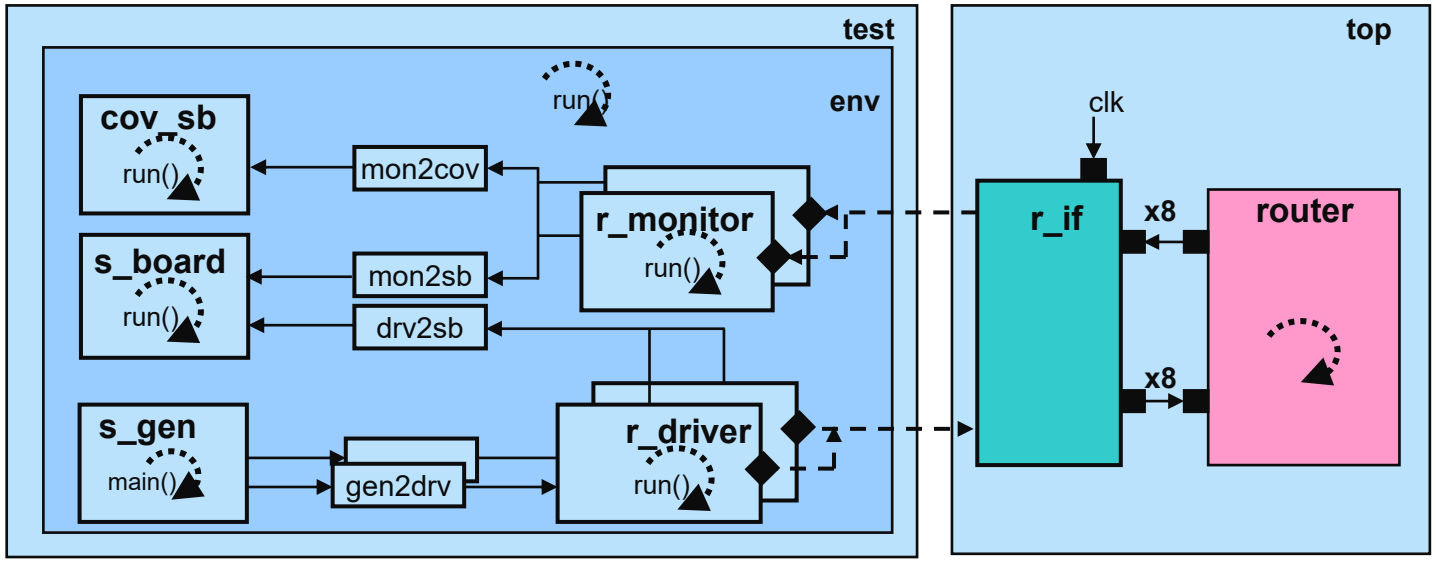
- Changes we made
  - Truly multi-threaded, system level testbench
  - Classes encapsulate functional blocks of the TB – ease of reuse
  - Classes encourage good coding style e.g. helper functions **`set_mb_handle()`**

# Lab 5: Inheritance

2RTP

# Lab 5 - Meet a new example Testbench and DUT

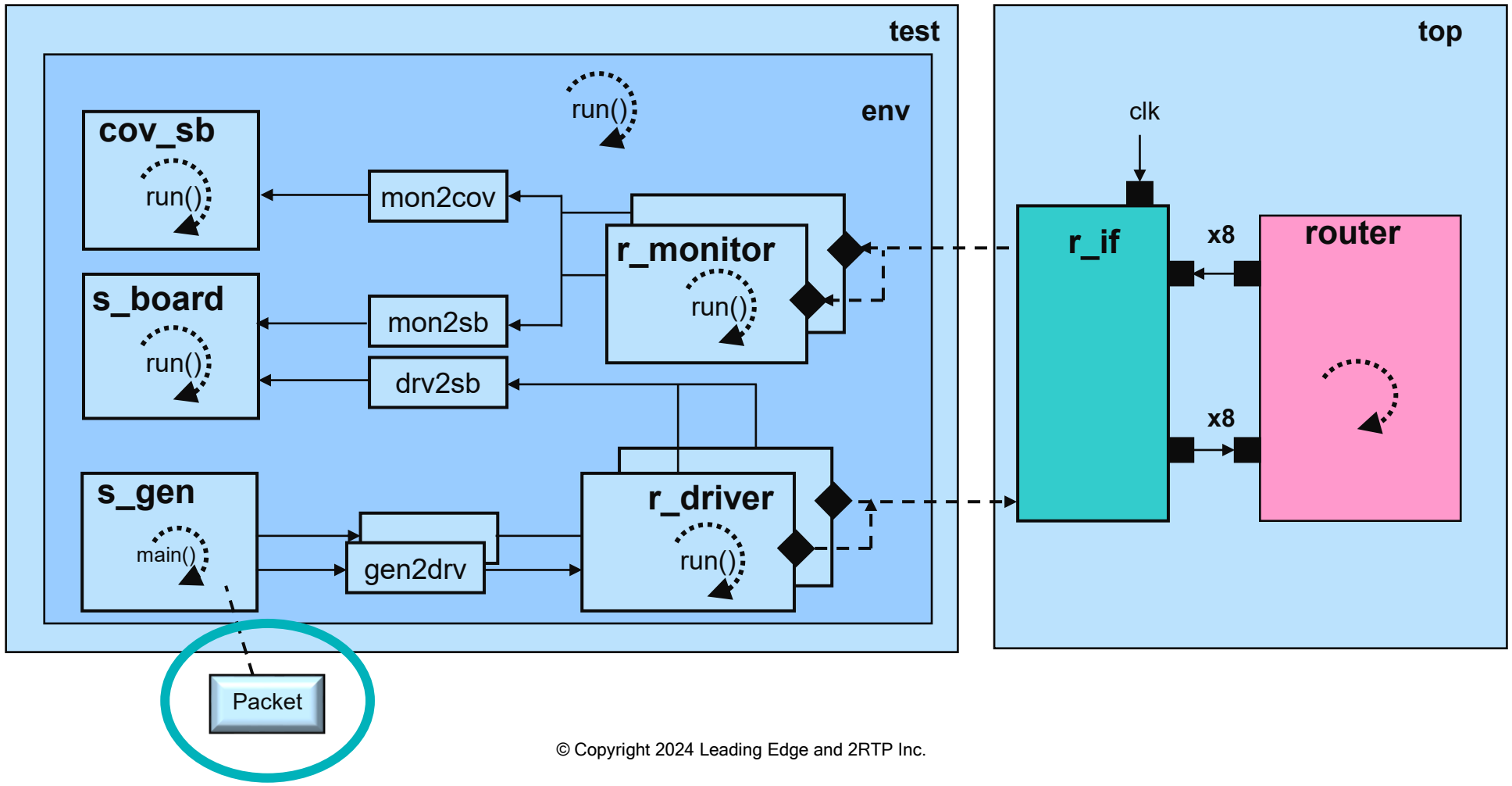
- Introducing a more sophisticated testbench with concurrent stimulus/checking




- The router **DUT** has 8 i/o channels, we must drive and monitor all 16 ports at the same time
  - Class **env** encapsulates a group of classes that implement the testbench and run concurrently
  - All **run(), main()** methods in the diagram are executed concurrently
  - Instance **s_gen** drives 8 DUT inputs using 8 **r_driver** instances via 8 **mailbox** instances
  - Scoreboards cover the DUT behavior via 8 **r_monitor** instances, one per **DUT** output
- We will implement key parts of this testbench as we learn new test techniques

# Lab 5 – Inheritance: Overview

- In this lab, you will complete the router testbench by extending the **`Packet`** base class to add functionality



# Lab 5 – Inheritance: Instructions – 1

Lab directory: inheritance

1. Edit the file **`txn/Packet.svh`**
   - Declare the class **`Packet`**
     - Derive it from the base class **`Packet_base`**
       - Note what is inherited!
     - Add the following properties:
       - **`payload`** which is a dynamic array of type **`bit[7:0]`**
       - **`src_id`**, **`dest_id`** both of which are of type **`bit[ROUTER_SIZE-1:0]`**
         - NOTE: **`ROUTER_SIZE`** is a parameter set to the size of the router
       - Add the type qualifier `rand` (put it in front of the type) to the **`src_id`**, **`dest_id`**, and **`payload`** properties
         - This is for randomization - we will cover it later

- Continued on next page -

# Lab 5 – Inheritance: Instructions – 2

2. Still editing the class **`Packet`** (file: **`txn/Packet.svh`**)
   - Override the the following methods:
     - **`function bit compare(Packet_base rhs);`**
       - Compare the properties of the **`rhs`** argument to the properties in (**`this`**) **`Packet`** class
       - Note: be sure and cast **`rhs`** from the **`Packet_base`** type to **`Packet`** type
       - Note: Remember to include the inherited property as well
       - Return 1 for success, 0 for fail
     - **`function void copy(Packet_base rhs);`**
       - Copies the properties of the **`rhs`** argument to the properties in (**`this`**) **`Packet`** class
       - Note: be sure and cast **`rhs`** from the **`Packet_base`** type to **`Packet`** type
       - Note: Remember to include the inherited property as well
   - Compile & simulate

     make

# Lab 5 – Inheritance: Partial Sample Output

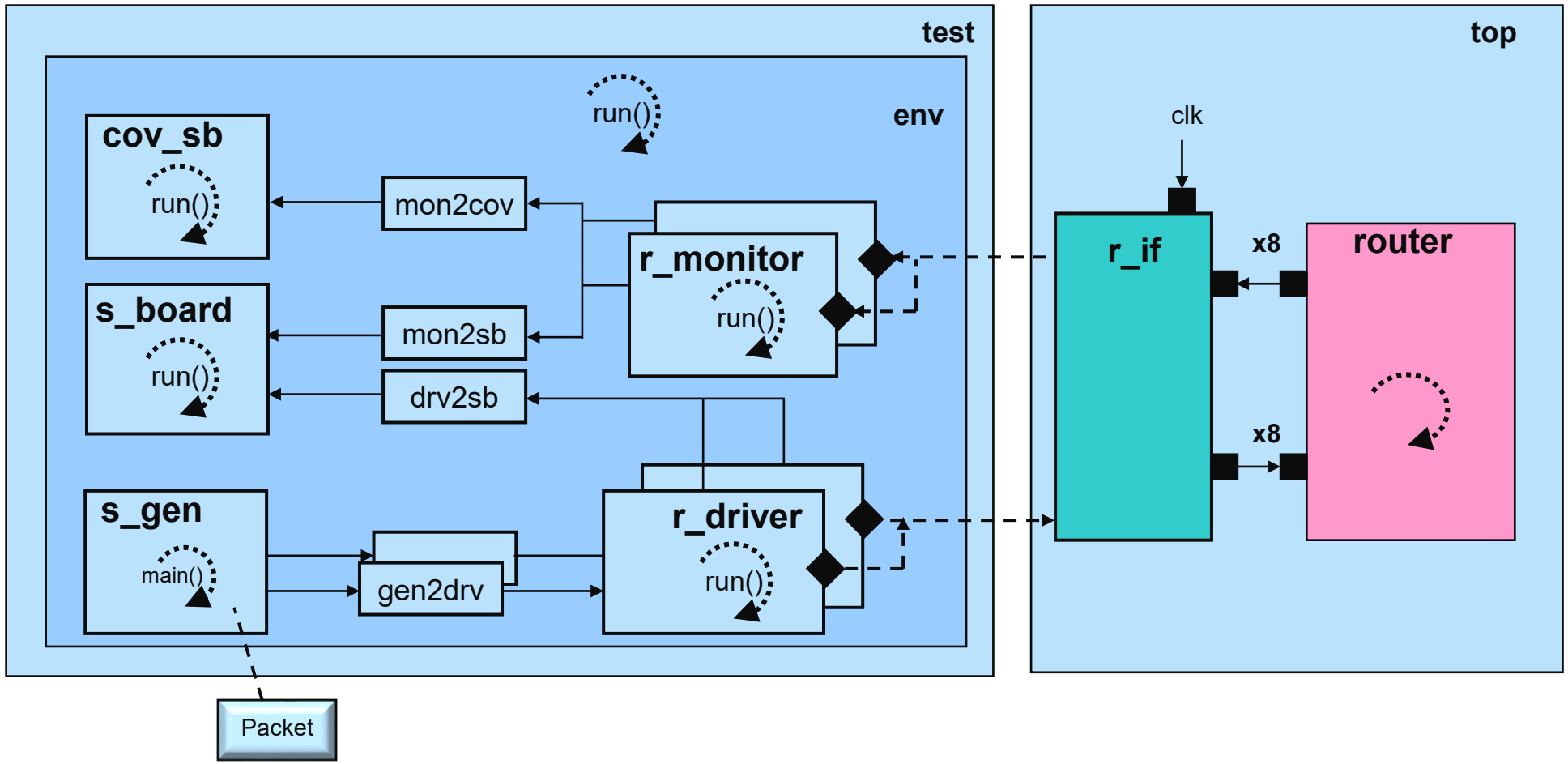
```
# Router size = 8x8
# stim_gen:  Creating 1000 Packets
#
#
# ********************************
# Matches:    989
# Mismatches: 0
# Missing:    1
#
# ********************************
#
#
# ********************************
#  Final Coverage = 100.000000%
#  100% Coverage met with 320 transactions
# ********************************
#
```

# Lab 6: Randomization

2RTP

# Lab 6 – Randomization: Overview

- Add randomization to the router testbench



# Lab 6 – Randomization: Instructions

Working directory: **randomization**

1. Edit the file **`txn/Packet.svh`**
   - Declare these properties randomizable
     - **`src_id`**
     - **`dest_id`**
     - **`payload`**
   - Write constraint expressions
     - **`src_id`** and **`dest_id`** should be non-negative values less than the size of the router (**`ROUTER_SIZE`**)
     - **`payload`** size should be between `1` and `5` (inclusive)
     - **`payload`** values should not repeat

# Lab 6 – Randomization: Instructions

Working directory: **randomization**

2. Edit the file **`stim/stim_gen.svh`**
   - Add a task called **`run()`**
     - Purpose is to generate **`num_to_send`** packets (perhaps a for loop?)
     - Randomize the transaction (**`txn`**), and be sure to check for failure
     - Set **`pkt_id`** to a unique value per transaction (use the loop variable?)
     - Write the **`txn`** into the **`gen2drv`** mailbox
3. Compile & run

# Lab 6 – Randomization: Partial Sample Output

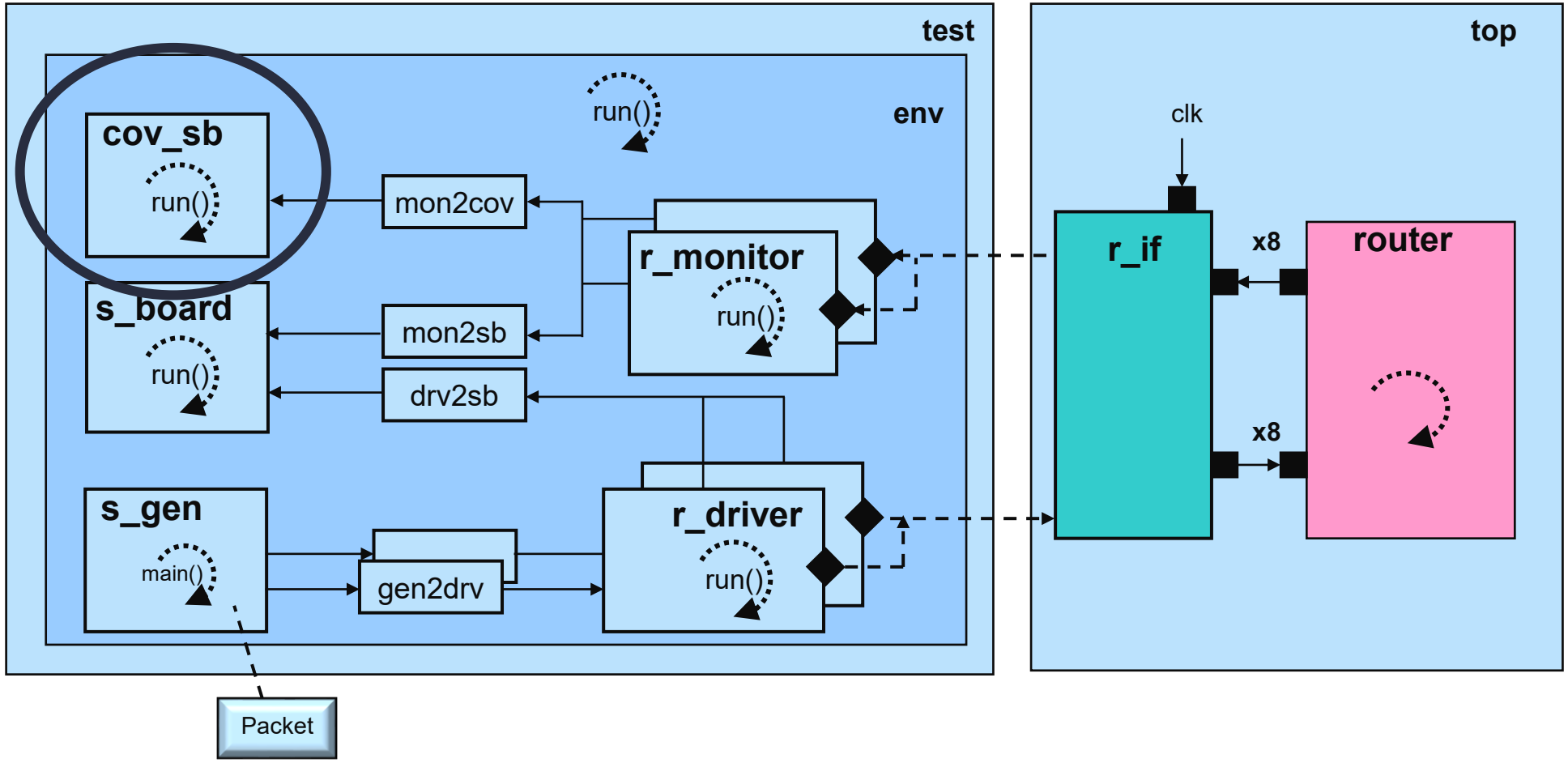
```
# Router size = 8x8
# stim_gen:  Creating 1000 Packets
#
#
# ********************************
# Matches:    989
# Mismatches: 0
# Missing:    1
#
# ********************************
#
#
# ********************************
#  Final Coverage = 100.000000%
#  100% Coverage met with 320 transactions
# ********************************
#
```

# Lab 7: Coverage

2RTP

# Lab 7 – Coverage: Overview

- Add functional coverage to the router testbench



# Lab 7 – Coverage: Instructions 1

Working directory: **coverage**

1. Edit the file **`analysis/router_coverage.svh`**
   - Declare a covergroup called **`cov1`**
     - Add coverpoints on properties **`src_id`** and **`dest_id`** of **`temp_txn`**
     - Add cross between the two coverpoints you just defined
   - Create the **`cov1`** covergroup in **new()**
   - Note the convenience methods **`sample()`** and **`get_coverage()`** at the bottom of the file

# Lab 7 – Coverage: Instructions 2

Working directory: **coverage**

2. Edit the file **`analysis/coverage_sb.svh`**
   - Create an instance of wrapper class **router_coverage** (previous slide) called **`r_cov`**
   - Edit the task called **`run()`**
     - Call the **`sample()`** task of the in **`r_cov`** with packet **`txn`**
     - Check coverage
       - Use the **`get_coverage()`** method in **`r_cov`**
       - Set the **`current_coverage`** variable to the coverage value (it is used to display the current coverage)
     - When 100% coverage is achieved…

       set the **`percentage_100_met`** bit to 1 and the **`percentage_100_cnt`** to **`txn_cnt`**
3. Compile & run

   **make**

# Lab 7 – Coverage: Partial Sample Output

```
# Router size = 8x8
# stim_gen:  Creating 1000 Packets
#
#
# ********************************
# Matches:    989
# Mismatches: 0
# Missing:    1
#
# ********************************
#
#
# ********************************
#  Final Coverage = 100.000000%
#  100% Coverage met with 320 transactions
# ********************************
#
```

**HINT**

Not seeing 100% coverage?
+ Check your coverpoint variables

# Lab 7 - Functional Coverage - Conclusions

- Good
  - We updated the testbench to include functional coverage
    - Measures metrics we defined
    - Tells us when we have reached those metrics

- Bad
  - We've come to the end of this part of the course

# Lab 8:
# Simple Assertions

2RTP

# Lab 8 – SVA Simulation example

## Lab Example Directory: sva/sva_simple_ex

```
module sva_ex;
logic [2:0] cnt;
logic clk;

initial
  begin
    clk = 0; cnt = 0;
    forever #20 clk = !clk;
  end

initial
  begin
    wait(cnt == 2) cnt = 4;
    #240 $stop;
  end

always @(posedge clk)
  cnt <= #1 cnt +1;
```

```
sequence s_count3;
   (cnt == 3'h1) ##1 (cnt == 3'h2)
                 ##1 (cnt == 3'h3);
endsequence


property p_count3;
  @(posedge clk) (cnt == 3'h0) |=> s_count3;
endproperty


assert_count3: assert property (p_count3);

endmodule
```

Sequence checks count 1:3

Property with cnt==0 antecedent

Inject deliberate error

Counter being checked

# Lab 8 – Run Interactive Simulation (Questa)

1. Invoke the simulation using the Makefile and target for Questa: **make**

2. The Assertions pane should open, if not, do so now using: View / Coverage / Assertions

3. In the Assertions pane right click on the assertion **sva_ex/assert_count3** and select Add to Wave

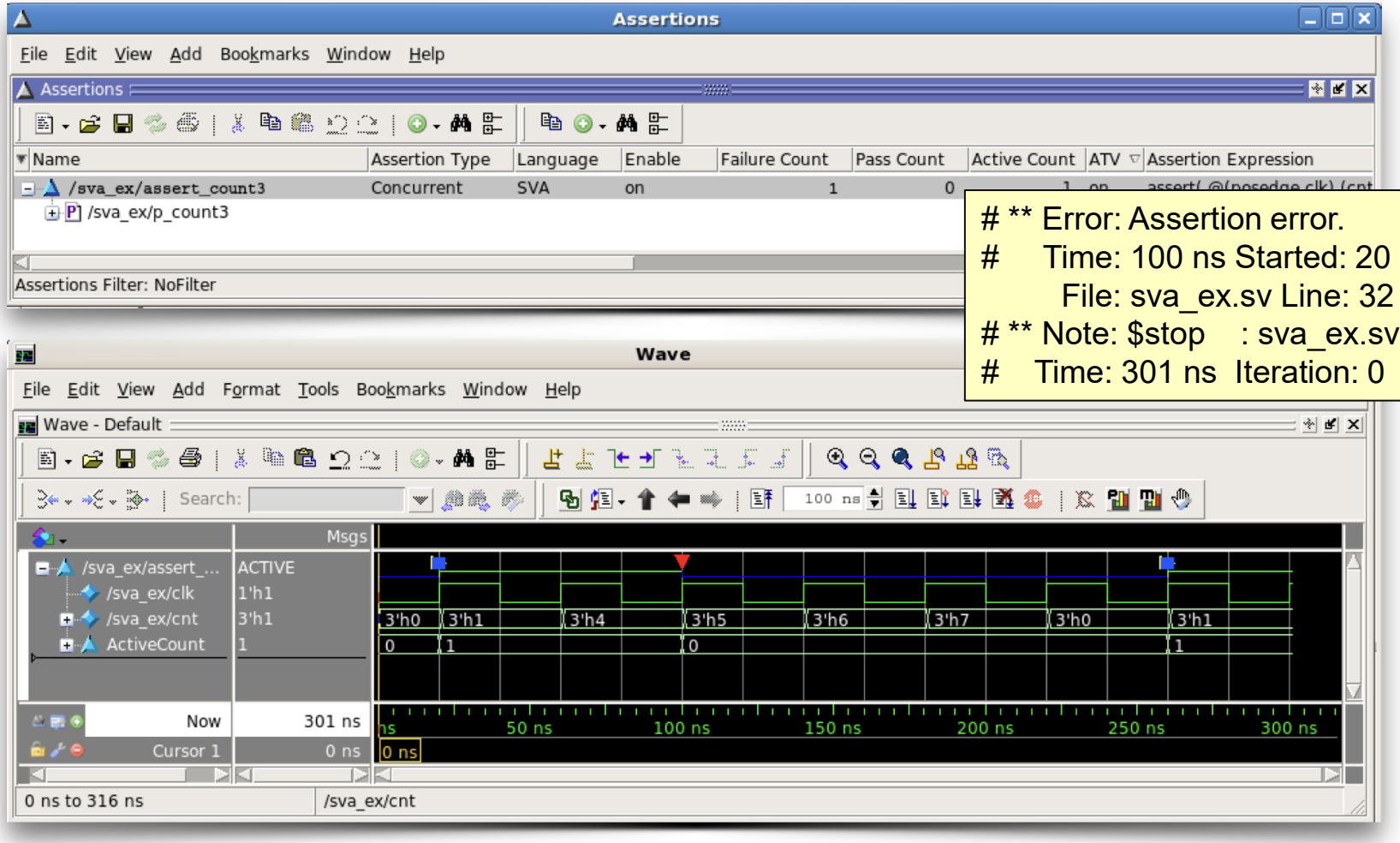
4. Repeat the right click a second time to select "Enable ATV"

5. Run simulation : **`run -all`**

6. Notice that assertion ***assert_count3*** fails at time 100ns.
   ( *This is reported in the Wave pane, the Analysis pane and the Transcript* pane)

7. At this point your Simulator should look as shown on the next slide…

# Lab 8 – Simulation Output (Questa)




*Simulator transcript*

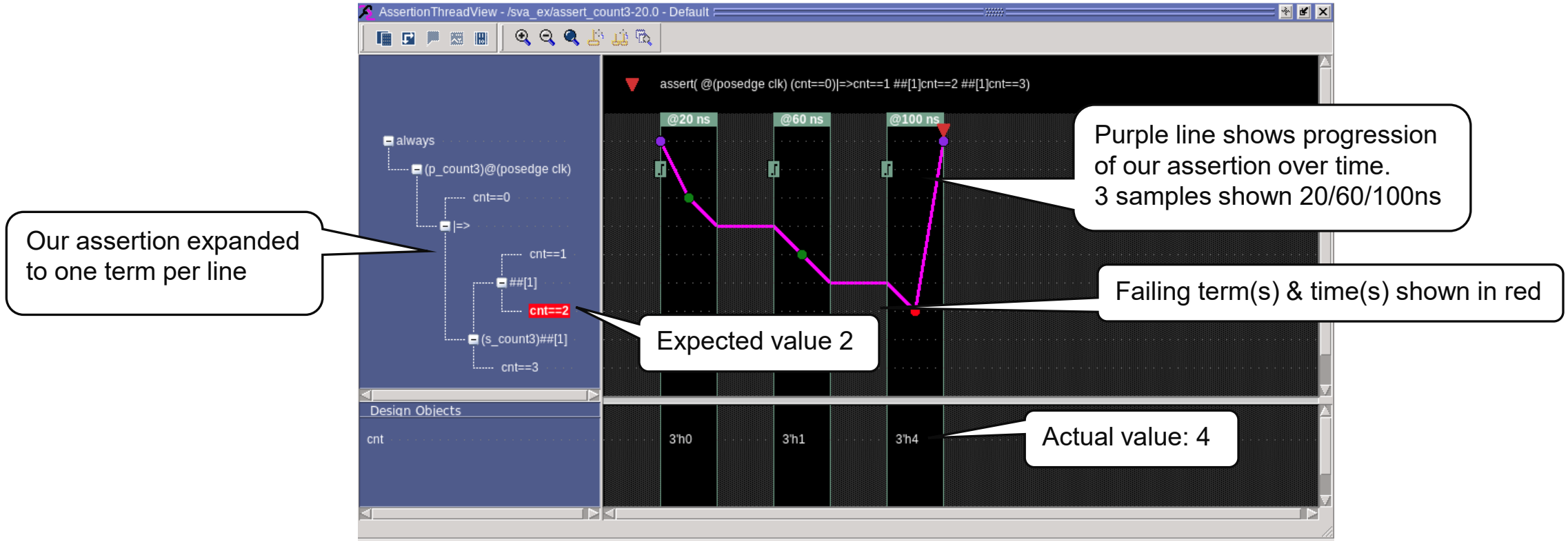
```
# ** Error: Assertion error.
#    Time: 100 ns Started: 20 ns  Scope: sva_ex.assert_count3
        File: sva_ex.sv Line: 32 Expr: cnt==2
# ** Note: $stop    : sva_ex.sv(17)
#    Time: 301 ns  Iteration: 0  Instance: /sva_ex
```

The Blue square shows where an assertion starts evaluation

The red down-arrow indicates assertion failure

# Lab 8 – Assertion Thread Viewer (ATV) - Questa

8. In the Wave pane right-click on a red-down-arrow and select : ***View ATV / 20ns***
9. The Assertion Thread Viewer opens
10. Right click near the word "always" on the left hand side and select : "Expand All Terms"



# Lab 8 – ATV - Questa

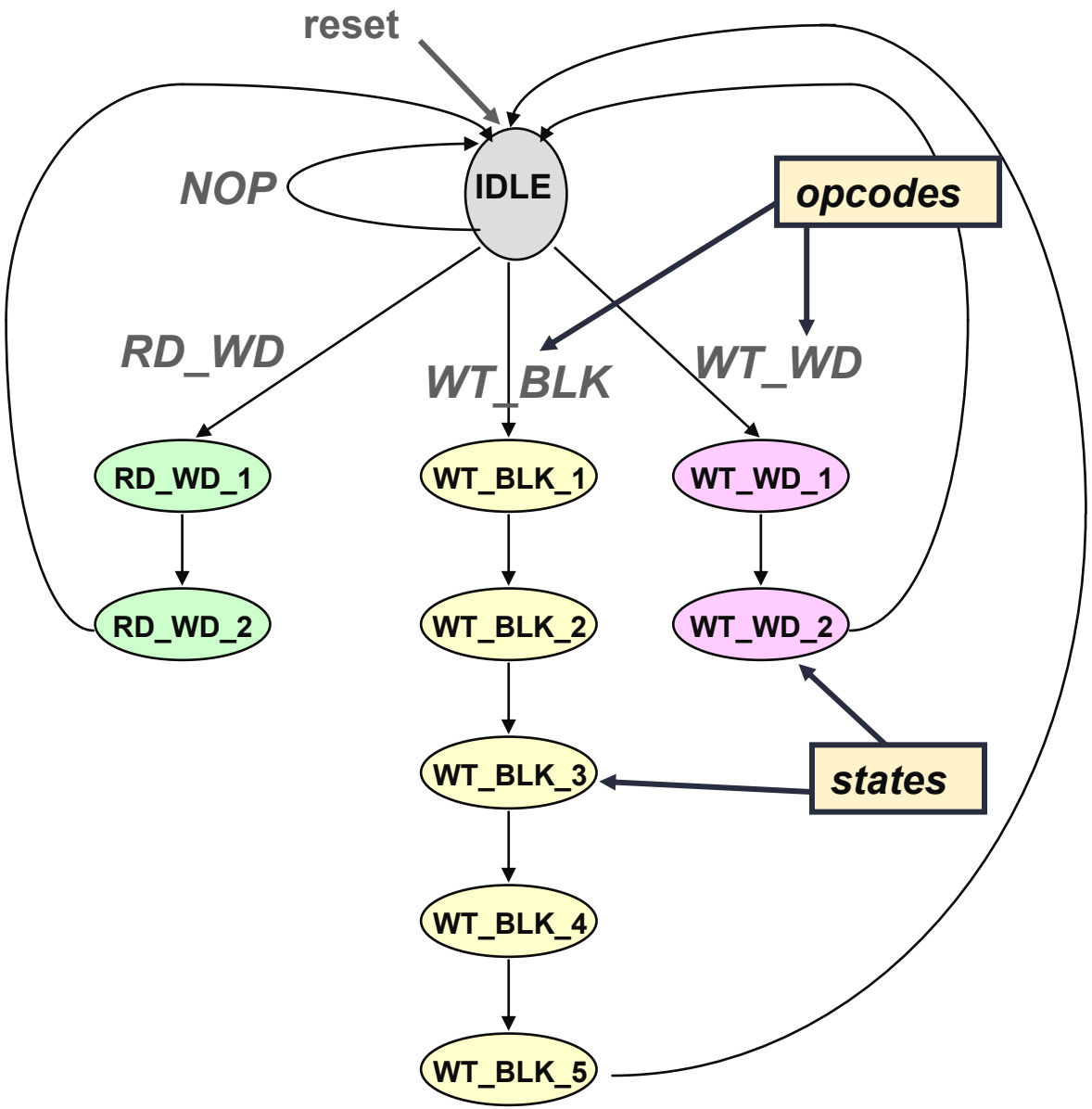
- Very useful for debug, we will use it shortly to explain more operators
- Synopsys Verdi has similar capabilities (not discussed here)

# Lab 8 – Simple Assertions: Overview

Lab directory: **sva_q/fsm**

1. Objective:
   - Write simple assertions to verify the state flow of a state machine
2. Edit the file **sva_container.sv**
   - Add assertions to module **`sva_container`** to verify that the state machine transitions from state to state correctly
     - Add sequences, properties and assert statements as needed
     - State diagram is on the next slide
   - The module **`sva_container`** is "bound" to the **sm** module such that all the signals you need are visible inside this module
     - All needed signals are inputs to this module
     - We will talk more about binding and how it works later
   - The enumeration of the state values are in a package in the file **types_pkg.sv**
3. Run "make" or "make gui" to compile and run
   - You should get no assertion failures
4. Run "make bad" to verify your assertions catch errors

# Lab 8 – Simple Assertions: State Transitions



# Lab 8 – Simple Assertions: Sample Output No Errors

```
5  state machine:  illegal op received
   135 Testbench:  Did a single write:  Address = 00000100, data = 000000aa
   195 Testbench:  Did a single write:  Address = 00000030, data = 000000bb
   315 Testbench:  Did a block  write:  Start address = 00000040, start data = 00000a10
   450 Testbench:  Read passed: Address = 00000100, data = 000000aa
   590 Testbench:  Read passed: Address = 00000030, data = 000000bb
   730 Testbench:  Read passed: Address = 00000040, data = 00000a10
   870 Testbench:  Read passed: Address = 00000041, data = 00000a11
  1010 Testbench:  Read passed: Address = 00000042, data = 00000a12
  1150 Testbench:  Read passed: Address = 00000043, data = 00000a13
```

This is the normal output of the state machine when there are no errors (including the “illegal op received”)

# Lab 8 – Simple Assertions: Sample Output with Errors

You won't receive all these errors unless you wrote assertions for every path

```
# 5 state machine:  illegal op received
#    135 Testbench:  Did a single write:  Address = 00000100, data = 000000aa
# ** Error: Assertion error.
#    Time: 170 ns Started: 130 ns  Scope: test_sm.sm_seq0.sm_0.sva_1.assert_p_wt_wd File: sva_container.sv Line: 44 Expr: state==WT_WD_2
#    195 Testbench:  Did a single write:  Address = 00000030, data = 000000bb

# ** Error: Assertion error.
#    Time: 230 ns Started: 190 ns  Scope: test_sm.sm_seq0.sm_0.sva_1.assert_p_wt_wd File: sva_container.sv Line: 44 Expr: state==WT_WD_2
#    315 Testbench:  Did a block  write:  Start address = 00000040, start data = 00000a10

# ** Error: Assertion error.
#    Time: 350 ns Started: 250 ns  Scope: test_sm.sm_seq0.sm_0.sva_1.assert_p_wt_blk File: sva_container.sv Line: 46 Expr: state==WT_BLK_5

# ** Error: Assertion error.
#    Time: 410 ns Started: 370 ns  Scope: test_sm.sm_seq0.sm_0.sva_1.assert_p_rd_wd File: sva_container.sv Line: 45 Expr: state==RD_WD_2
#    450 Testbench:  Read ERROR:  Address = 00000100, data = xxxxxxxx, should have been = 000000aa

# ** Error: Assertion error.
#    Time: 550 ns Started: 510 ns  Scope: test_sm.sm_seq0.sm_0.sva_1.assert_p_rd_wd File: sva_container.sv Line: 45 Expr: state==RD_WD_2
#    590 Testbench:  Read ERROR:  Address = 00000030, data = xxxxxxxx, should have been = 000000bb

# ** Error: Assertion error.
#    Time: 690 ns Started: 650 ns  Scope: test_sm.sm_seq0.sm_0.sva_1.assert_p_rd_wd File: sva_container.sv Line: 45 Expr: state==RD_WD_2
#    730 Testbench:  Read ERROR:  Address = 00000040, data = xxxxxxxx, should have been = 00000a10

# ** Error: Assertion error.
#    Time: 830 ns Started: 790 ns  Scope: test_sm.sm_seq0.sm_0.sva_1.assert_p_rd_wd File: sva_container.sv Line: 45 Expr: state==RD_WD_2
#    870 Testbench:  Read ERROR:  Address = 00000041, data = xxxxxxxx, should have been = 00000a11

# ** Error: Assertion error.
#    Time: 970 ns Started: 930 ns  Scope: test_sm.sm_seq0.sm_0.sva_1.assert_p_rd_wd File: sva_container.sv Line: 45 Expr: state==RD_WD_2
#   1010 Testbench:  Read ERROR:  Address = 00000042, data = xxxxxxxx, should have been = 00000a12

# ** Error: Assertion error.
#    Time: 1110 ns Started: 1070 ns  Scope: test_sm.sm_seq0.sm_0.sva_1.assert_p_rd_wd File: sva_container.sv Line: 45 Expr: state==RD_WD_2
#   1150 Testbench:  Read ERROR:  Address = 00000043, data = xxxxxxxx, should have been = 00000a13
```

# Lab 8 - Simple Assertions - Conclusions

- Good
  - We wrote simple assertion(s)
  - We learned the component parts of an assertion
  - Tip: Remember to check for same signal values at start and end of a sequence
    - Example: Assertions start in idle state and then test for correct return to idle

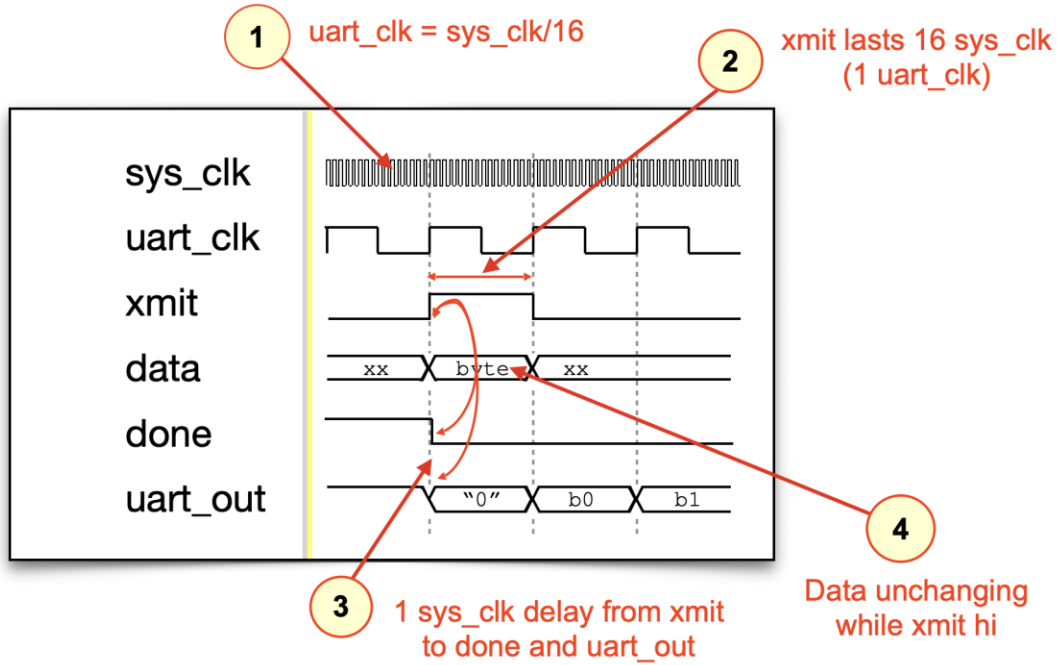
# Lab 9:
# UART Sequences

2RTP

# Lab 9 - UART Sequences

Working Directory: **sva_uart_xmit**

- Write 4 sequences per the timing diagram below
  - 4 properties and assert directives are already provided




Why is this here?

```
property p_uart_sys16;
  @(posedge sys_clk)
  $rose(uart_clk) && !sys_rst_l |-> s_uart_sys16;
endproperty

property p_xmit_hi16;
  @(posedge sys_clk) $rose(xmit) |-> s_xmit_hi16;
endproperty

property p_xmit_done;
  @(posedge sys_clk) $rose(xmit) |-> s_xmit_done;
endproperty

property p_xmit_nc_data;
  @(posedge sys_clk) ($rose(xmit)) |=> s_xmit_nc_data;
endproperty
```

# Lab 9 - UART Sequences

Working Directory: **sva_uart_xmit**

1. Edit file: **test_u_xmit.sv**

   - Look for the properties/assertions provided
   - Write your 4 sequences BEFORE those properties (see instructions in file)

     **s_uart_sys16**: Verify **uart_clk** == **sys_clk** / 16
     **s_xmit_hi16**: **xmit** stays hi for 16 **sys_clk** cycles
     **s_xmit_done**: One **sys_clk** after **xmit** asserts, look for **done**(lo) and **uart_out**(lo)
     **s_xmit_nc_data**: While **xmit** is asserted **data** value does not change

2. Compile and run using the appropriate Makefile target

| | | | |
|---|---|---|---|
| **make** | *<use Questa>* | **make vcs** | *<use VCS>* |
| **make gui** | *<use Questa GUI>* | **make vcs_gui** | *< use VCS with GUI)* |
| **make bad** | *<uses a bad DUT>* | **make vcs _bad** | *<uses a bad DUT>* |

# Lab 9 - UART Sequences – Example Output

```
#      27330 Success! Data transmitted: 0a
#      38850 Success! Data transmitted: 0b
#      50370 Success! Data transmitted: 0c
#      61890 Success! Data transmitted: 0d
#      73410 Success! Data transmitted: 0e
#      84930 Success! Data transmitted: 0f
#      96450 Success! Data transmitted: 10
#     107970 Success! Data transmitted: 11
#     119490 Success! Data transmitted: 12
#     131010 Success! Data transmitted: 13
#     142530 Success! Data transmitted: 14
```

No errors ( make )

With errors (make bad)

```
#       4290 *****Forcing uart_clk to not be sys_clk/16 - should see an assertion error
# test_u_xmit.U1.A1.assert_uart_sys16 : uart_clk should = sys_clk/16
# test_u_xmit.U1.A1.assert_uart_sys16 : uart_clk should = sys_clk/16
#
#      26370 Success! Data transmitted: 0a
#      37890 Success! Data transmitted: 0b
#      49410 Success! Data transmitted: 0c
#      60930 Success! Data transmitted: 0d
#      72450 Success! Data transmitted: 0e
#      83970 Success! Data transmitted: 0f
#      95490 Success! Data transmitted: 10
#     107010 Success! Data transmitted: 11
#     118530 Success! Data transmitted: 12
#     130050 Success! Data transmitted: 13
#     141570 Success! Data transmitted: 14
#
#     144930 *****Changing data too soon - should see an assertion error
# test_u_xmit.U1.A1.assert_xmit_nc_data : data should not change while xmit asserted
#
#     161250 *****Clearing xmit too soon - should see an assertion error
# test_u_xmit.U1.A1.assert_xmit_hi16 : Signal xmit should stay hi for 16 sys_clks
```

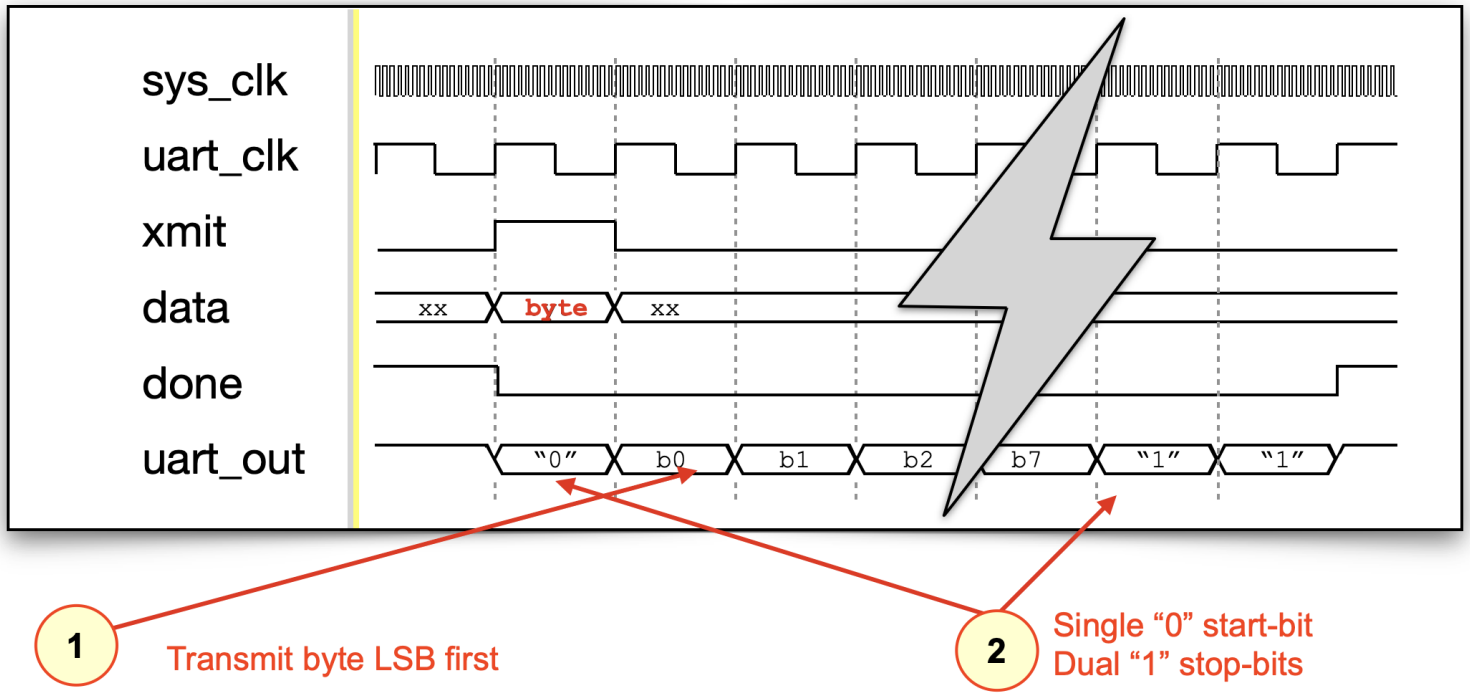
# Lab 10:
# UART Transmit

2RTP

# Lab 10 - UART Transmit Lab

Working Directory: **sva_uart_xmit**

- Write an assertion to verify the transmitted bitstream



# Lab 10 - UART Transmit Lab

Working Directory: **sva_uart_xmit**

1. Edit file: **test_u_xmit.sv**

   - Write an assertion called **p_val_bit_stream**
     - Verifies the correct serial bitstream is transmitted
     - Data byte transmitted LSB first
     - One "0" start bit, two "1" stop bits
   - Move all assertions in the file to a new module called **my_assertions**
     - Keep both modules in the same file, to avoid having to edit the Makefile
     - Use bind to bind **my_assertions** to the **U1** instance of **u_xmit**

2. Compile and run using the appropriate Makefile target

| | | | |
|---|---|---|---|
| **make** | *<use Questa>* | **make vcs** | *<use VCS>* |
| **make gui** | *<use Questa GUI>* | **make vcs_gui** | *< use VCS with GUI)* |
| **make bad** | *<uses a bad DUT>* | **make vcs _bad** | *<uses a bad DUT>* |

# Lab 10 - UART Transmit Lab – Example Output

```
#       27330 Success! Data transmitted: 0a
#       38850 Success! Data transmitted: 0b
#       50370 Success! Data transmitted: 0c
#       61890 Success! Data transmitted: 0d
#       73410 Success! Data transmitted: 0e
#       84930 Success! Data transmitted: 0f
#       96450 Success! Data transmitted: 10
#      107970 Success! Data transmitted: 11
#      119490 Success! Data transmitted: 12
#      131010 Success! Data transmitted: 13
#      142530 Success! Data transmitted: 14
```

No errors ( make )

With errors (make bad)

```
#        4290 *****Forcing uart_clk to not be sys_clk/16 - should see an assertion error
# test_u_xmit.U1.A1.assert_uart_sys16 : uart_clk should = sys_clk/16
# test_u_xmit.U1.A1.assert_uart_sys16 : uart_clk should = sys_clk/16
#
#       26370 Success! Data transmitted: 0a
#       37890 Success! Data transmitted: 0b
#       49410 Success! Data transmitted: 0c
#       60930 Success! Data transmitted: 0d
#       72450 Success! Data transmitted: 0e
#       83970 Success! Data transmitted: 0f
#       95490 Success! Data transmitted: 10
#      107010 Success! Data transmitted: 11
#      118530 Success! Data transmitted: 12
#      130050 Success! Data transmitted: 13
#      141570 Success! Data transmitted: 14
#
#      144930 *****Changing data too soon - should see an assertion error
# test_u_xmit.U1.A1.assert_xmit_nc_data : data should not change while xmit asserted
# test_u_xmit.U1.A1.assert_val_bit_stream : uart_out bitstream incorrect

#      161250 *****Clearing xmit too soon - should see an assertion error
# test_u_xmit.U1.A1.assert_xmit_hi16 : Signal xmit should stay hi for 16 sys_clks
```

# Sample Solutions

2RTP

# Sample Solution Data Structures: wishbone_pkg.sv

```
// LAB - create a package named wishbone_pkg

package wishbone_pkg;

  // LAB - declare an enumeration of wishbone operation types
  typedef enum  {NONE, WRITE, READ, RMW, WAIT_IRQ } wb_txn_t;

  // LAB - declare a struct

  typedef struct {
    wb_txn_t  txn_type;
    bit[31:0] adr;
    logic[31:0] data;
    int unsigned count;
    bit[7:0] byte_sel;
  } wb_txn;

endpackage
```

# Sample Solution Data Structures: test.sv

```
// test module
//-----------------

module test;
import wishbone_pkg::*;

  // LAB - Declare the shadow mem associative array
  bit[31:0] shadow_mem[bit[31:0]];

  initial begin
    // LAB - Declare transaction struct variable, address, and data variables
    wb_txn txn;
    bit[31:0] address;
    bit[31:0] data;

    // LAB - Wait for reset signal to go low
    @(negedge top.wb_bfm.rst);

    // LAB - Write 10  data values to 10  addresses,
    // also storing into the shadow mem.

    //initialize address
    address = 0;
    for(int i=0; i <10; i++) begin
      shadow_mem[address] = i;  // write shadow mem
      // setup txn for write
      txn.adr = address;
      txn.data = i;
      txn.txn_type = WRITE;
      txn.count = 1;
      txn.byte_sel[3:0] = 4'b1111;
      // Write Wishbone mem
      top.wb_bfm.wb_write_cycle(txn);
      // increment address
      address += 4;
    end
      foreach (shadow_mem[addr]) begin
        // setup txn for read
        txn.adr = addr;
        txn.data = 32'bx;
        txn.txn_type = READ;
        txn.count = 1;
        txn.byte_sel[3:0] = 4'b1111;
        // Read Wishbone mem
        top.wb_bfm.wb_read_cycle(txn);
        // check read result against shadow mem
        if (txn.data === shadow_mem[addr]) begin
          $display("Memory passed at address %x",addr);
        end else begin
          $display("Memory failed at address %x -
expected %x, got %x",addr, shadow_mem[addr], txn.data[0]);
        end
      end
      $finish;
    end


endmodule
```

# Sample Solution Tasks & Interfaces: test.sv (Pt 1)

```
// test module
//------------------

module test();
import wishbone_pkg::*;


 initial begin
   @(negedge top.wb_bfm.rst);
   generate_stimulus();
   check_mem();
   $finish();
 end

  bit[31:0] shadow_mem[bit[31:0]];

  // LAB - gen_write_txn
  task automatic gen_write_txn(bit[31:0] address, bit[31:0] data);
    logic [31:0] temp_data[1]; // temp data array
    temp_data[0] = data;
    shadow_mem[address] = data;
    top.wb_bfm.wb_write_cycle(temp_data, address);
  endtask

  // LAB - gen_read_txn
  task automatic gen_read_txn(bit[31:0] address, ref bit[31:0] data);
    logic [31:0] temp_data[1]; // temp data array
    top.wb_bfm.wb_read_cycle(temp_data, address);
    data = temp_data[0];
  endtask
```

# Sample Solution Tasks & Interfaces: test.sv (Pt 2)

```
  task automatic generate_stimulus();
    bit[31:0] address = 0;
    bit[31:0] data;
    for(int i=0; i <10; i++) begin
      data = i;
      // increment address
      gen_write_txn(address, data);
      address += 4;
    end
  endtask

  task automatic check_mem();
    bit[31:0] data;
    foreach (shadow_mem[addr]) begin
        gen_read_txn(addr, data);
        if (data === shadow_mem[addr]) begin
          $display("Memory passed at address %x",addr);
        end else begin
          $display("Memory failed at address %x -- expected %x, got %x",
                   addr, shadow_mem[addr], data);
        end
    end
  endtask
endmodule
```

# Sample Solution Concurrency: wishbone_pkg.sv

```
package wishbone_pkg;

  typedef enum  {NONE, WRITE, READ, RMW, WAIT_IRQ } wb_txn_t;

  typedef struct packed{
    wb_txn_t  txn_type;
    bit[31:0] adr;
    logic[31:0] data;
    int unsigned count;
    bit[7:0] byte_sel;
  } wb_txn;

  mailbox #(wb_txn) mon_mb = new();
  int in_flight = 0;

  task automatic ready_to_finish();
    wait (in_flight == 0) $finish();
  endtask
```

```
 task check_mem();
  bit[31:0] shadow_mem[bit[31:0]];
  wb_txn txn;
  bit[31:0] data;
  forever begin
   mon_mb.get(txn);
   in_flight--;
   case (txn.txn_type)
     WRITE: shadow_mem[txn.adr] = txn.data;
     READ: begin
       if (txn.data === shadow_mem[txn.adr]) begin
         $display("Memory passed at address %x",txn.adr);
       end else begin
         $display("Memory failed at address %x -- expected %x,
got %x",txn.adr, shadow_mem[txn.adr], txn.data[0]);
       end
     end
    endcase
   end // forever
  endtask
endpackage
```

# Sample Solution Concurrency: test.sv (Pt 1)

```
// test module
//------------------

module test();
import wishbone_pkg::*;

  task automatic gen_write_txn(bit[31:0] address, bit[31:0] data);
    top.wb_bfm.wb_write_cycle(data, address);
  endtask

  task automatic gen_read_txn(bit[31:0] address, ref bit[31:0] data);
    top.wb_bfm.wb_read_cycle(data, address);
  endtask

  task monitor();
    wb_txn txn;
    forever begin
      top.wb_bfm.monitor_txn(txn.data,txn.adr, txn.txn_type);
      mon_mb.put(txn);
    end
  endtask
```

# Sample Solution Concurrency: test.sv (Pt 2)

```
  task automatic generate_stimulus();
    bit[31:0] address = 0;
    bit[31:0] data;
    for(int i=0; i <10; i++) begin
      data = i;
      // increment address
      in_flight++;
      gen_write_txn(address, data);
      in_flight++;
      gen_read_txn(address, data);
      address += 4;
    end
    wishbone_pkg::ready_to_finish();
  endtask

 initial begin
   @(negedge top.wb_bfm.rst);
    // LAB - fork tasks
    fork
      generate_stimulus();
      check_mem();
      monitor();
    join_none;
  end

endmodule
```

# Sample Solution Classes: env/wb_env.svh

```
class wb_env;

  mailbox #(wb_txn) gen2drv;
  mailbox #(wb_txn) mon2sb;

  stim_gen       s_gen;
  wb_bfm_driver  driver;
  wb_bfm_monitor monitor;
  mem_checker    mem_sb;

  function void build();
    gen2drv = new(1);
    mon2sb  = new();

    s_gen   = new();
    driver  = new();
    mem_sb  = new();
    monitor = new();
endfunction
```

```
function void connect();
    s_gen.set_mb_handle(gen2drv);
    driver.set_mb_handle(gen2drv);
    monitor.set_mb_handle(mon2sb);
    mem_sb.set_mb_handle(mon2sb);
    driver.set_v_if(v_wb_bfm);
    monitor.set_v_if(v_wb_bfm);
endfunction

  task run();
    fork
      driver.run();
      monitor.run();
      mem_sb.run();
    join_none
    s_gen.run();
    disable fork;
  endtask

  function void report();
    mem_sb.report();
  endfunction
endclass
```

# Sample Solution Classes: top_modules/test.sv

```
// test module
//---------------------

module test;
import env_pkg::wb_env;

 wb_env env;

 initial  begin
   env = new();  // create environment

   // wait for the bus reset
   wishbone_pkg::v_wb_reset.wait_wb_bus_reset();

   // call env methods
   env.build();
   env.connect();
   env.run();
   env.report();
   $finish();
 end

endmodule
```

# Sample Solution Inheritance: txn/Packet.svh (Pt 1)

```
class Packet extends Packet_base;

 rand bit[ROUTER_SIZE-1:0] src_id;
 rand bit[ROUTER_SIZE-1:0] dest_id;
 rand bit[7:0] payload[];

 constraint srce {src_id  < ROUTER_SIZE ;}
 constraint dest {dest_id < ROUTER_SIZE ;}

 constraint pyld {
   payload.size() inside {[1:5]};
 }

 function bit compare(Packet_base rhs);
   Packet tmp;
   compare = (
     $cast(tmp, rhs)          &&
     src_id  == tmp.src_id    &&
     dest_id == tmp.dest_id   &&
     payload == tmp.payload   &&
     pkt_id  == tmp.pkt_id
   );
 endfunction
```

```
 function void copy(Packet_base rhs);
   Packet tmp;
   $cast(tmp, rhs);
   src_id    = tmp.src_id ;
   dest_id   = tmp.dest_id;
   payload   = tmp.payload;
   pkt_id    = tmp.pkt_id;
 endfunction


// function to init Packet properties
 function void init_txn(
                bit[7:0] s_id,
                bit[7:0] d_id,
                bit[7:0] p_load[],
                int p_id );
   src_id = s_id;
   dest_id = d_id;
   payload = p_load;
   pkt_id = p_id;
 endfunction
```

# Sample Solution Inheritance: txn/Packet.svh (Pt 2)

```
function string convert2string();
  string str1;
  str1 = {    "\n-------------------- Start Packet txn --------------------\n",
              "Packet txn:\n",
    $sformatf("  src_id          : 'h%h\n", src_id),
    $sformatf("  dest_id         : 'h%h\n", dest_id),
    $sformatf("  pkt_id          :  %0d\n", pkt_id),
              "  payload:\n"};
    if(payload.size() < 11 )
      for(int i=0; i<payload.size(); i++)
        str1 = {str1,$sformatf("    payload[%0d] = 'h%h\n", i, payload[i])};
    else  begin
      for(int i=0; i<6; i++)
        str1 = {str1,$sformatf("    payload[%0d] = 'h%h\n", i, payload[i])};
      str1 = {str1, ". . .\n"};
      for(int i=payload.size()-5; i<= payload.size(); i++)
        str1 = {str1,$sformatf("    payload[%0d] = 'h%h\n", i, payload[i])};
    end

    str1 = {str1,"-------------------- End Packet txn ---------------------\n"};
  return (str1);
 endfunction

 function void print();
  $display(convert2string());
 endfunction


endclass
```

# Sample Solution Randomization: txn/Packet.svh (Pt 1)

```
class Packet extends Packet_base;

 rand bit[ROUTER_SIZE-1:0] src_id;
 rand bit[ROUTER_SIZE-1:0] dest_id;
 rand bit[7:0] payload[];

 constraint srce {src_id  < ROUTER_SIZE ;}
 constraint dest {dest_id < ROUTER_SIZE ;}

 constraint pyld {
   payload.size() inside {[1:5]};
 }

 constraint uniq {unique { payload };}

 function bit compare(Packet_base rhs);
   Packet tmp;
   bit cst = $cast(tmp, rhs);
   bit[7:0] uni_q[$] = tmp.payload.unique;
   compare = (
     cst          &&
     uni_q.size() == tmp.payload.size() &&
     src_id  == tmp.src_id   &&
     dest_id == tmp.dest_id  &&
     payload == tmp.payload  &&
     pkt_id  == tmp.pkt_id
   );
 endfunction
```

```
 function void copy(Packet_base rhs);
   Packet tmp;
   $cast(tmp, rhs);
   src_id    = tmp.src_id ;
   dest_id   = tmp.dest_id;
   payload   = tmp.payload;
   pkt_id    = tmp.pkt_id;
  endfunction

 // function to init Packet properties
  function void init_txn(
                 bit[7:0] s_id,
                 bit[7:0] d_id,
                 bit[7:0] p_load[],
                 int p_id );
    src_id = s_id;
    dest_id = d_id;
    payload = p_load;
    pkt_id = p_id;
  endfunction
```

# Sample Solution Randomization: txn/Packet.svh (Pt 2)

```
function string convert2string();
  string str1;
  str1 = {    "\n-------------------- Start Packet txn --------------------\n",
              "Packet txn:\n",
    $sformatf("  src_id          : 'h%h\n", src_id),
    $sformatf("  dest_id         : 'h%h\n", dest_id),
    $sformatf("  pkt_id          :  %0d\n", pkt_id),
              "  payload:\n"};
    if(payload.size() < 11 )
      for(int i=0; i<payload.size(); i++)
        str1 = {str1,$sformatf("    payload[%0d] = 'h%h\n", i, payload[i])};
    else  begin
      for(int i=0; i<6; i++)
        str1 = {str1,$sformatf("    payload[%0d] = 'h%h\n", i, payload[i])};
      str1 = {str1, ". . .\n"};
      for(int i=payload.size()-5; i<= payload.size(); i++)
        str1 = {str1,$sformatf("    payload[%0d] = 'h%h\n", i, payload[i])};
    end

    str1 = {str1,"-------------------- End Packet txn ---------------------\n"};
  return (str1);
 endfunction

 function void print();
  $display(convert2string());
 endfunction

endclass
```

# Sample Solution Randomization: stim/stim_gen.svh

```
//-----------------------------------------
// stimulus generator
// generates Packets for the router
//
import analysis_pkg::verbosity;

class stim_gen;

 mailbox #(Packet_base) gen2drv[ROUTER_SIZE];  // mailbox handles

 int num_to_send = 1000;

 task run();
  Packet txn;

  if(verbosity >= analysis_pkg::MEDIUM)
    $display("stim_gen:  Creating %0d Packets\n", num_to_send);
  for(int i = 1; i <= num_to_send; i++) begin
   txn = new();
   if(!txn.randomize())
      if(verbosity >= analysis_pkg::MEDIUM)
            $display("stim_gen:  Randomization error");
   txn.pkt_id = i; // set Packet id
   gen2drv[txn.src_id].put(txn);
  end
 endtask

endclass
```

# Sample Solution Coverage: analysis/router_coverage.svh

```
//-----------------------------------------
// coverage object
// tracks functional coverage

class router_coverage;

  Packet temp_txn;

  covergroup cov1;
   s: coverpoint temp_txn.src_id {
       bins src[ROUTER_SIZE] = {[0:ROUTER_SIZE-1]};
      }
   d: coverpoint temp_txn.dest_id {
       bins dest[ROUTER_SIZE] = {[0:ROUTER_SIZE-1]};
      }
   cross s, d;
  endgroup

  function new();
    cov1 = new();
  endfunction


  virtual function void sample(Packet p);
    temp_txn = p;
    cov1.sample();
  endfunction


  virtual function real get_coverage();
    return cov1.get_coverage();
  endfunction

endclass
```

# Sample Solution Coverage: analysis/coverage_sb.svh

```
//-----------------------------------------
// coverage scoreboard
// tracks functional coverage

class coverage_sb;

// mailbox handle
 mailbox #(Packet_base) mon2cov;

 router_coverage r_cov;
 int txn_cnt;
 real current_coverage;
 int percentage_100_cnt;
 bit percentage_100_met;

 function new();
   r_cov = new();
 endfunction

 task run();
    Packet_base temp;
    Packet txn;
    forever  begin
      mon2cov.get(temp);
      $cast(txn, temp); // cast so can access all the properties
      txn_cnt++;
      r_cov.sample(txn);
      current_coverage = r_cov.get_coverage();
      if(current_coverage == 100 && !percentage_100_met) begin
        percentage_100_cnt = txn_cnt;
        percentage_100_met = 1;
      end
      if(verbosity >= DEBUG)
        $display("%0d Packets sampled,  Coverage = %f%% ",
                 txn_cnt,current_coverage);
    end
  endtask

 function void report();
  if(verbosity >= MEDIUM) begin
    $display("\n*********************************");
    $display(" Final Coverage = %f%% ",current_coverage);
    if(percentage_100_met)
      $display(" 100%% Coverage met with %0d transactions",
       percentage_100_cnt);
    $display("*********************************\n");
  end
 endfunction

endclass
```

# Sample Solution Simple Assertions: Properties

```
property p_nop;
  @(posedge clk) state==IDLE && opcode == NOP |=> state == IDLE;
endproperty

property p_wt_wd;
  @(posedge clk) state==IDLE && opcode == WT_WD |=>
  state == WT_WD_1 ##1 state == WT_WD_2 ##1 state == IDLE;
endproperty

property p_wt_blk;
  @(posedge clk) state==IDLE && opcode == WT_BLK |=>
  state == WT_BLK_1 ##1 state == WT_BLK_2 ##1 state == WT_BLK_3 ##1
  state == WT_BLK_4 ##1 state == WT_BLK_5 ##1 state == IDLE;
endproperty

property p_rd_wd;
  @(posedge clk) state==IDLE && opcode == RD_WD |=>
  state == RD_WD_1 ##1 state == RD_WD_2 ##1  state == IDLE;
endproperty
```

# Sample Solution Simple Assertions: assert

```
assert_p_nop1: assert property(p_nop)
                    else $display ("p_nop error");

assert_p_wt_wd1: assert property(p_wt_wd)
                    else $display ("p_wt_wd error");

assert_p_wt_blk1: assert property(p_wt_blk)
                    else $display ("p_wt_blk error");

assert_p_rd_wd1: assert property(p_rd_wd)
                    else $display ("p_rd_wd error");
```

# Sample Solution: UART sequences

```
sequence s_uart_sys16;
  uart_clk[*8] ##1 !uart_clk[*8] ##1 uart_clk;
   // Alternative:  Non duty-cycle dependant but less efficient
   //  ##1(!$rose(uart_clk)) [*15] ##1 $rose(uart_clk);
endsequence
sequence s_xmit_hi16;
  @(posedge sys_clk) xmit[*16] ##1 !xmit;
endsequence
sequence s_xmit_done;
// ##1 (!done) && !uart_out;
  ##1 $fell(done) && $fell(uart_out);
endsequence
sequence s_xmit_nc_data;
   $stable(data) [*1:$] ##1 !xmit;
endsequence
```

*What is the difference between these 2 alternate solutions?*

# Sample Solution: bind

```
module my_assertions( input sys_clk, uart_clk, sys_rst_l,
                                 uart_out, xmit, done,
                           input [7:0] data );

. . .
. . .


sequence seq1(cmp1);
   (uart_out == cmp1[0], cmp1 = cmp1>>1);
endsequence

property p_val_bit_stream;
  logic [7:0] cmp;
  @(posedge uart_clk) ($rose(xmit), cmp = data) |->
    !uart_out ##1 (seq1(cmp))[*8]
    ##1 uart_out;
endproperty
. . .
. . .

// assertions

assert_val_bit_stream:
     assert property (p_val_bit_stream)
         else $display("%m : uart_out bitstream incorrect");

. . .
endmodule
```

```
module test_u_xmit;


. . .
. . .
. . .

u_xmit U1(  .*  );


bind U1 my_assertions A1 ( .* );


endmodule
```