

# Data Wrapper Module Test Plan and Results

Hongkuan Yu

Jan 2025

## Contents

<b>1</b>	<b>DUT</b>	<b>2</b>
<b>2</b>	<b>Simulation Testbench (Dynamic)</b>	<b>4</b>
<b>3</b>	<b>Formal Property Verification (Static)</b>	<b>5</b>
<b>4</b>	<b>UVM (1.1d)</b>	<b>8</b>

# 1 DUT

This module is simple, so I include the SystemVerilog code of `dut.sv` below.

```
module dut(clk,
           rst_n,
           rxd,
           rx_dv,
           txd,
           tx_en);
input clk;
input rst_n;
input[7:0] rxd;
input rx_dv;
output [7:0] txd;
output tx_en;

reg[7:0] txd;
reg tx_en;

always @(posedge clk) begin
    if(!rst_n) begin
        txd <= 8'b0;
        tx_en <= 1'b0;
    end
    else begin
        txd <= rxd;
        tx_en <= rx_dv;
    end
end
endmodule
```

This `dut.sv` serves as a **simple data wrapper**. The I/O ports of `dut.sv` and their functions are shown in Table 1 below.

input (wire)	clk	1 bit	Global clock
input (wire)	rst_n	1 bit	Active-low reset
input (wire)	rx_dv	1 bit	Valid for data received
input (wire)	rx_d	8 bits	Data received
output (reg)	tx_d	8 bits	Data transmitted
output (reg)	tx_en	1 bit	Valid for data transmitted

Table 1: I/O ports of `dut.sv` and their functions

The module's I/O graphic overview is shown in Figure 1 below.

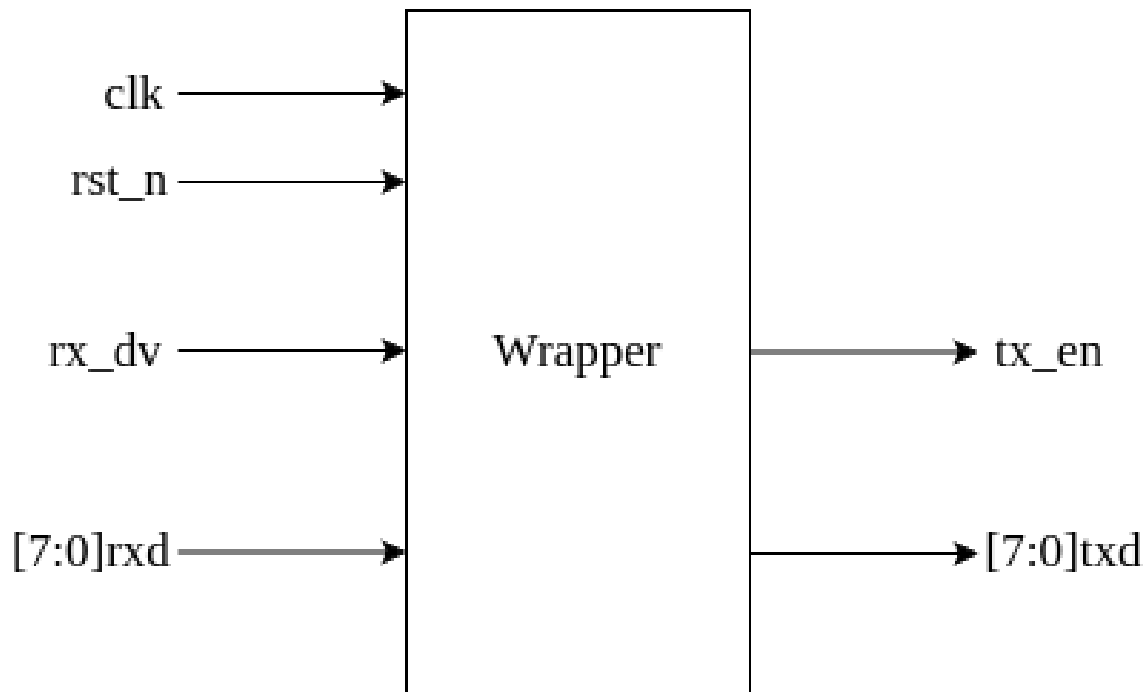


Figure 1: Wrapper Module's I/O Overview

## 2 Simulation Testbench (Dynamic)

Create `sim_tb` folder holding simulation testbench. Write a testbench `dut_tb.sv` to verify the `dut.sv` dynamically by Synopsys VCS.

Create `sim_tb_script` folder for necessary scripts. Create `filelist.f` and `Makefile` containing normal simulation commands.

Create `sim_tb_results` folder for simulation testbench results. The results of the simulation testbench meet expectations: the output signals `tx_en` and `txd` successfully get the correct value of `rx_dv` and `rx_d` after one clock period, separately. The result waveform is shown below in Figure 2 with decimal representation.

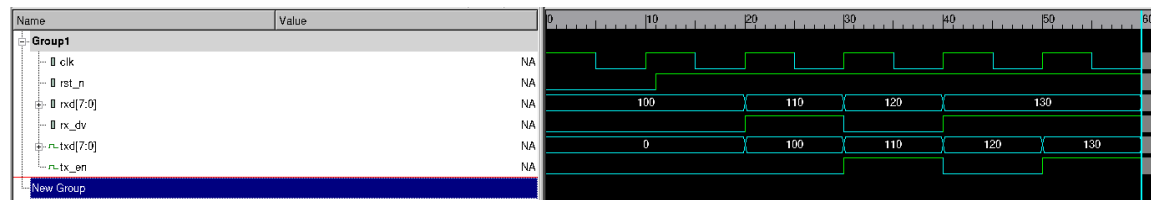


Figure 2: Simulation Testbench Result

### 3 Formal Property Verification (Static)

Create `sva` folder for formal property verification. Write a formal property verification `dut_sva.sv` to verify `dut.sv` statically.

Create `sva_script` folder for necessary scripts. Create `filelist.f` and `Makefile` containing operation commands of formal verification tool.

Create `sva_results` folder for formal property verification results.

The truth table of implication operator  $\rightarrow$  and  $\Rightarrow$  (`(|-> ##1)`) is shown below in Figure 3.

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Figure 3: Truth Table of Implication Operator

According to the truth table, the implication will always be true if the first condition goes false. Hence, for the same first condition, both itself and its negation should be checked (or use cover). The SystemVerilog Assertions written for `dut.sv` are shown below.

```
property check_signal_and_data_1;
    @(posedge clk) disable iff(!rst_n)
        !rx_dv | => !tx_en && txd == $past(rxd);
endproperty
```

```
property check_signal_and_data_2;
    @(posedge clk) disable iff(!rst_n)
        rx_dv | => tx_en && txd == $past(rxd);
endproperty
```

Also, check the functionality of the active-low reset.

```
property active_low_reset_functionality;
    @(posedge clk)
        $rose(rst_n) | -> (txd == 8'h0 && tx_en == 0);
endproperty
```

Notice that VC Formal will give half-correct with red vacuity when checking the negation of the active-low reset, i.e. `$fell(rst_n)`. So does checking to hold for the active-low reset, i.e. `!rst_n`. Still trying to figure out the reason for both of them.

All showed property passed. The result is shown below in Figure 4.

The screenshot displays the VC Formal FPV Result window. The top section shows the 'Task List' with a single task 'FPV' in a green state, indicating success. The 'Verification Targets: ALL' table lists three targets, all of which passed. The 'Constraints: ALL' table shows one constraint, 'constant\_3', which is also satisfied. The 'Properties' section indicates that 3 properties passed, 0 failed, and 0 were disabled. The 'VC Formal Console' at the bottom shows the execution log, including the creation of the simulation model and the start of formal verification.

status	depth	name	vacuity	witness	engine	type	clock	elapsed_time
✓		dut.dut_sva_check.Active_low_reset_functionality	1		t1	assert	..._sva_check.clk	00:00:00
✓		dut.dut_sva_check.Check_signal_and_data_1	1		e1	assert	..._sva_check.clk	00:00:00
✓		dut.dut_sva_check.Check_signal_and_data_2	1		e1	assert	..._sva_check.clk	00:00:00

name	vacuity	witness	usage	type	class	language
constant_3			assume	constconstraint	script	

Properties: 3 - passed[3] - failed[0] - disabled[0] ; Constraints Enabled: 1 ; Run Time: 0:00:04

```

45 1
46 create_reset rst_n -sense low
47 1
48 # Running a reset simulation
49 sim_run -stable
50 [Info] SIM_I_CREATE: Create Simulation Model.
51 1
52 sim_save_reset<_ma_ />
53 #check_fv
54 [Info] FORMAL_I_CREATE: Create Formal Model:dut.
55 [Info] FORMAL_I_RUN: Starting formal verification for check_fv
56   Id: 0 Goals: 6 Constraints: 0 Block Mode: false
57 [Info] APP_LIC_CHKOUT: Checkout 1 app license(s).

```

Figure 4: VC Formal FPV Result

## 4 UVM (1.1d)

Create `uvm` folder holding all UVM components. The UVM components include (from top to bottom): a base test, environment, model, scoreboard, agent, sequencer, transaction, driver, and monitor. Two different test cases serve as two different sequences. An interface is included for portable design. `top_tb.sv` connects them all.

Create `uvm_script` folder for necessary scripts. Create `filelist.f` and `Makefile` containing UVM simulation commands.

Create `uvm_results` folder for UVM results.

The structure of this UVM platform is shown in Figure 5 below. This platform does not contain the UVM register model.

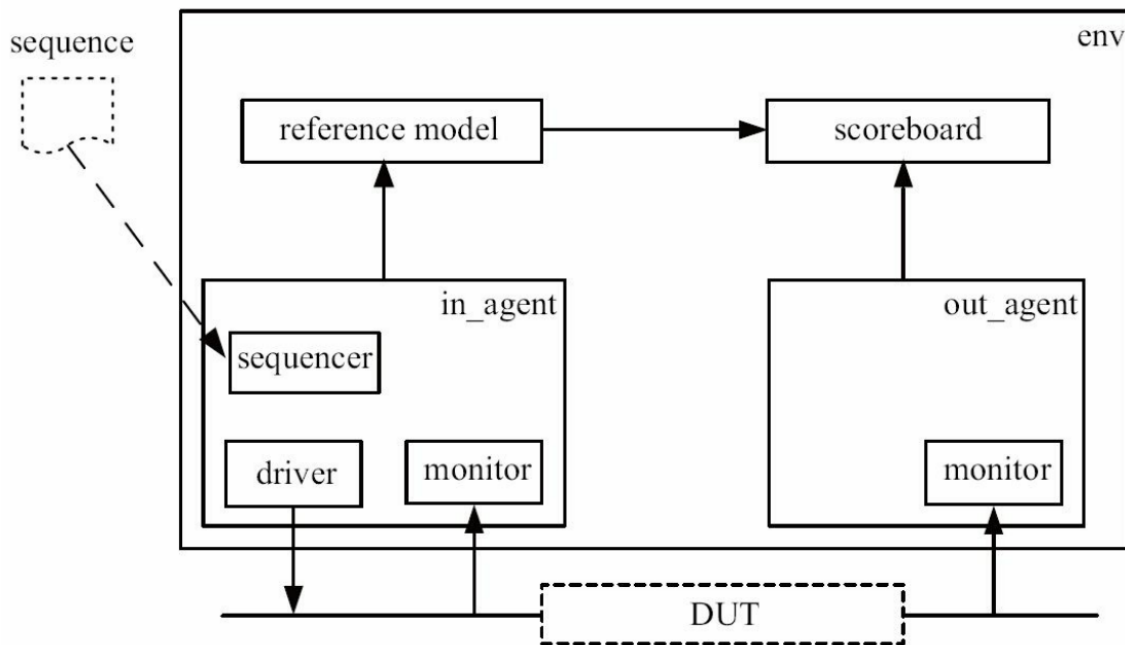


Figure 5: Structure of UVM Platform



This platform's UVM tree is shown in Figure 6 below.

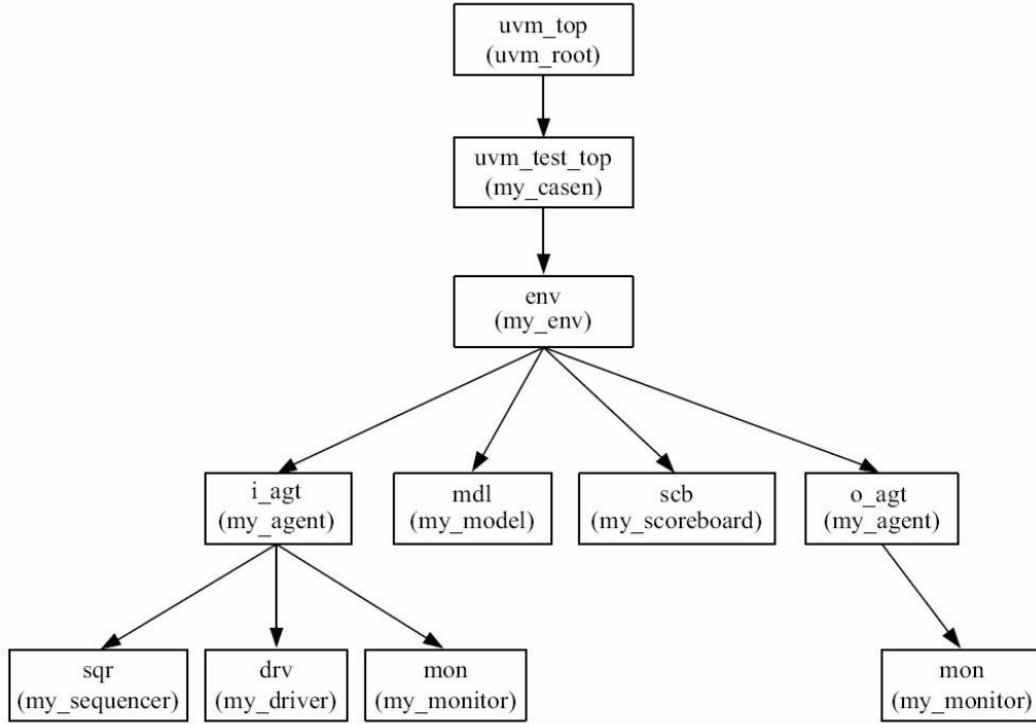


Figure 6: UVM Tree

The transaction mimics the bitstream of an ethernet connection by generating random MAC addresses and other related information. Test case 0 does not restrict the bitstream length. Test case 1 restricts the bitstream length to 60 bytes. Both test cases passed. Please refer to the .log file respectively in the `uvm_results` folder.

## References