Grey Code Counter Verification

Pranav T. Varkey, Shrikrishna Pandit
PES2UG22EC098, PES2UG22EC135

AGENDA

01 DUT

- Code snippet
- Explanation
- RTL

02 Directed Test-Bench

- Code snippet
- Implementation
- Outputs
- Waveforms

03 Layered Test-Bench

- Top
- Scoreboard
- Monitor
- Generator
- Driver
- Interface
- Environment
- Test
- Transaction

04 Outputs

- Simulation Output
- Waveform

01

DUT

Code Snippet


```
1  `timescale 1ns/1ps
2  module gray_counter (input clk, input rst, output reg [2:0]
   o_o, output [2:0] gray);
3      reg [2:0] bun; // Internal binary counter
4      always @(posedge clk) begin
5          if (rst)
6              bun <= 3'b000; // Reset counter
7          else
8              bun <= bun + 1; // Increment counter
9      end
10     // Assign the binary counter to `o_o`
11     always @(posedge clk) begin
12         o_o <= bun;
13     end
14     // Gray code conversion
15     assign gray = {bun[2], bun[2] ^ bun[1], bun[1] ^
   bun[0]};
16 endmodule
17
```

Explanation of DUT

The Gray Counter module implements a synchronized 3-bit counter with binary-to-Gray code conversion. The system comprises:

1. Binary Counter Component (`bun`):

   - 3-bit synchronous counter

   - Clock-driven incrementing

   - Active reset functionality (`rst`)

2. Output Architecture:

   - Binary output (`o_o`): Direct counter value

   - Gray code output (`gray`): Converted binary value

3. Binary-to-Gray Conversion Logic:

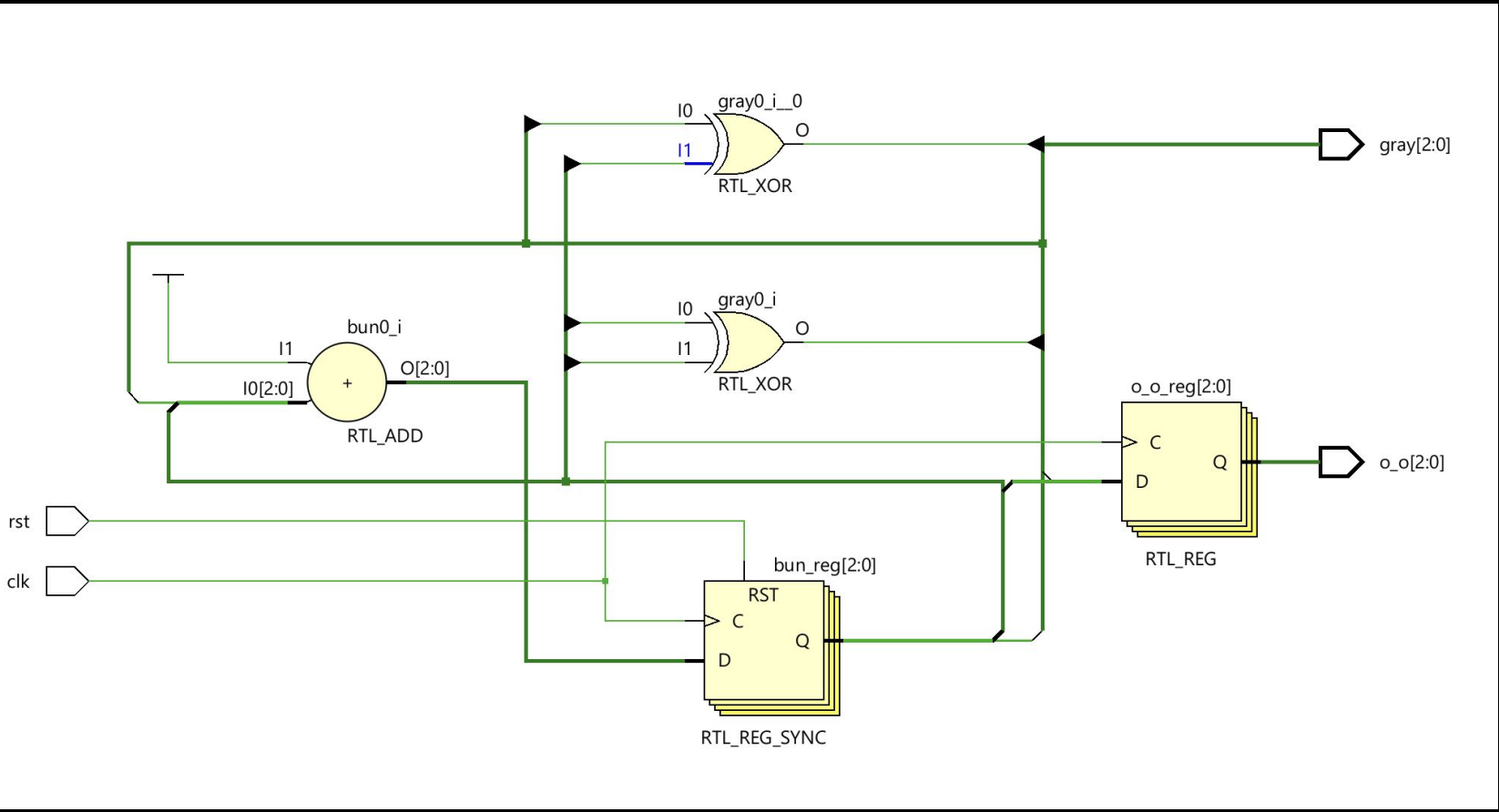
   - MSB: Retained from binary value

   - Other bits: XOR operation between adjacent binary bits

   - Mathematical representation: `gray[i] = binary[i] XOR binary[i+1]`

This implementation provides an efficient synchronous counting system with simultaneous binary and Gray code outputs, suitable for applications requiring minimal bit transitions between sequential states.

RTL


I0
gray0_i__0
O
I1
RTL_XOR
gray[2:0]
I0
gray0_i
O
I1
RTL_XOR
bun0_i
I1
O[2:0]
I0[2:0]
+
RTL_ADD
o_o_reg[2:0]
C
Q
D
RTL_REG
o_o[2:0]
rst
clk
bun_reg[2:0]
RST
C
Q
D
RTL_REG_SYNC

02

Test-Bench

Code Snippet


```
module gray_counter_tb;
    reg clk, rst; wire [2:0] o_o, gray;
    // Instantiate the gray counter
    gray_counter dut (
      .clk(clk), .rst(rst), .o_o(o_o), .gray(gray));
    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end
    // Test stimulus
    initial begin
        // Initialize waveform dump
        $dumpfile("gray_counter_tb.vcd");
        $dumpvars(0, gray_counter_tb);
        // Reset test
        rst = 1;
        #20;
        rst = 0;
        // Run for 8 clock cycles to observe full counting sequence
        #80;
        // Apply reset again
        rst = 1;
        #10;
        rst = 0;
        #30;
        $finish;
    end
    // Monitor changes
    initial begin
      $monitor("Time=%0t rst=%b binary=%b gray=%b", $time, rst, o_o, gray);
    end
endmodule
```

Explanation of Test-Bench

A directed testbench was developed to verify the gray_counter module's functionality through predetermined test scenarios. The testbench implements:

1. Test Environment Setup:

   - Clock generation with 10ns period

   - Module instantiation with port mapping

   - Signal monitoring and waveform dumping

2. Test Scenarios:

   - Initial reset verification

   - Complete counting sequence (0-7)

   - Secondary reset validation

   - Concurrent binary and Gray code output monitoring

3. Verification Methods:

   - Automated signal monitoring via $monitor

   - Waveform generation for visual analysis

   - Controlled test duration with $finish

This structured approach ensures comprehensive verification of both reset functionality and counting sequence, validating the module's binary-to-Gray code conversion accuracy.

03

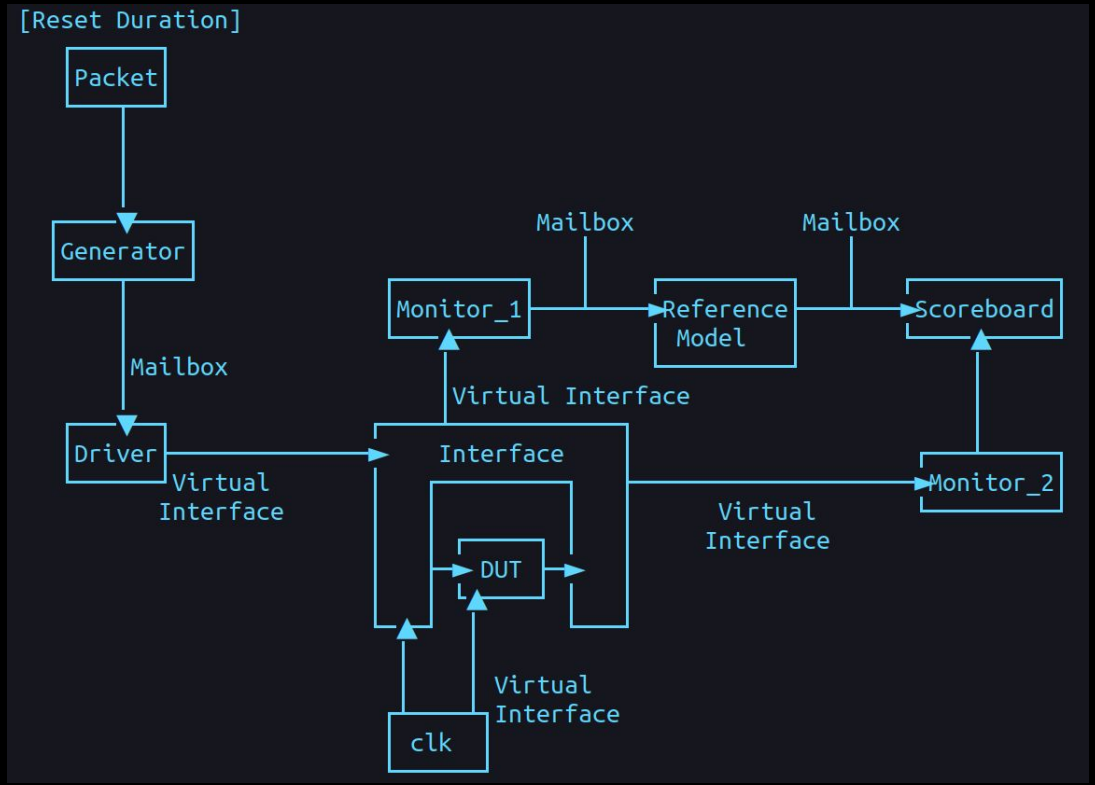
Layered Test-Bench

All the different Layers

1. Top
2. Environment
3. Test
4. Interface
5. Transaction
6. Driver
7. Generator
8. Scoreboard
9. Monitor

A **layered testbench** is a modular approach to verifying digital designs, dividing functionality into distinct layers for better structure and reusability. It consists of a **driver** that generates low-level signals or transactions to stimulate the Design Under Test (DUT), a **monitor** that observes DUT outputs and translates them into high-level data, and a **scoreboard** or **checker** that compares expected and actual results to validate functionality. A **test control layer** coordinates these components to execute specific test scenarios. This separation of concerns simplifies debugging, enhances reusability, and improves the efficiency and effectiveness of the verification process.

Layered Test-bench



3.a

Top

Code Snippet

```
`include "interface.sv"
//`include "test.sv"
module tbench_top_gray;

        intf i_intf();

        clock cl(.clk(i_intf.clk), .Tc(i_intf.Tc));

        test t1(i_intf);

        gray_counter c1 (i_intf.out, i_intf.count, i_intf.clk, i_intf.rst);

        initial begin
                $dumpfile("dump.vcd"); $dumpvars;
        end

endmodule

module clock (
        output bit clk,
        input int Tc
        );
        always #(Tc) clk = ~clk;
        initial clk = 0;
endmodule
```

This SystemVerilog testbench validates a `gray_counter` module using an interface (`intf`) for connectivity. A `clock` module generates a clock signal based on the period `Tc`. The `test` module (`t1`) provides stimulus, and the `gray_counter` (`c1`) is connected via the interface. Waveform generation is enabled with `$dumpfile` and `$dumpvars`.

3.b

Test

Code Snippet

```
`include "environment.sv"

program test(intf i_intf);
        environment env;

        initial
        begin
                env = new(i_intf);
                repeat (2) env.run();
                $finish;
        end

endprogram
```

This SystemVerilog program `test` sets up a verification environment using the `intf` interface. An environment object env is created and initialized with i_intf. The env.run() task is executed twice to drive and monitor the simulation, followed by ending the simulation with $finish.

3.c

Environment

Code Snippet


```
`include "transaction.sv"
`include "generator.sv"
`include "driver.sv"
`include "monitor.sv"
`include "scoreboard.sv"

class environment;
        generator       gen;
        driver          driv;
        monitor         mon;
        scoreboard      scb;
        mailbox                 m1;
        mailbox                 m2;

        virtual intf vif;
        function new(virtual intf vif);
                this.vif = vif;
                m1   = new();
                m2   = new();
                gen  = new(m1);
                driv = new(vif,m1);
                mon  = new(vif,m2);
                scb  = new(m2);
        endfunction

        task gener();
                        $display("============= Generating @ %0d =============", $time);
                        fork
                                gen.main();
                                driv.main();
                        join
        endtask

        task test();
                        $display("=============== Testing @ %0d ===============", $time);
                        fork
                                mon.main();
                                scb.main();
                        join
        endtask


        task run;
                driv.reset;
                gener();
                fork
                        repeat(7) begin
                                test();
                                #(2* vif.Tc);
                        end
                        driv.reset;
                join
        endtask

endclass
```

Environment

This SystemVerilog code defines an `environment` class for a modular verification environment. It integrates key components:

Components:

- `**generator**`: Generates transactions.
- `**driver**`: Drives signals based on transactions.
- `**monitor**`: Observes and collects signals.
- `**scoreboard**`: Compares actual and expected results.
- **Mailboxes**: `m1` and `m2` are used for inter-component communication.

Initialization:

- Constructor initializes the virtual interface (`vif`) and components, linking mailboxes for communication.

Tasks:

- `gener()`: Runs the generator and driver concurrently.
- `test()`: Executes monitor and scoreboard processes concurrently.
- `run()`: Resets the driver, runs generation, and performs repeated testing over 7 iterations, syncing with `vif.Tc`.

This class organizes a comprehensive simulation environment with parallel execution of verification processes.

3.d

Interface

Code Snippet


```
interface intf();

        logic rst;
        logic clk;
        logic [2:0] count;
        logic [2:0] out;
        int reset_duration, Tc = 1; // Clock Duration

endinterface
```

This SystemVerilog code defines an interface `intf` to centralize signal and parameter declarations for modular communication. Key elements include:
Signals:
- `rst`: Reset signal.
- `clk`: Clock signal.
- `count` and `out`: 3-bit data signals for counter operations.
Parameters:
- `reset_duration`: Duration for the reset signal.
- `Tc`: Clock period, initialized to 1.

This interface simplifies connectivity and improves modularity in testbench design.

3.e

Transaction

Code Snippet


```
class transaction;
        rand int reset_duration;        // Duration to release the reset signal

        logic [2:0] out, count;
        logic rst, clk;

        // Constraint to limit reset duration to a reasonable range
        constraint reset_duration_c { reset_duration inside {[2:7]}; }

        // Function to randomize the transaction values
        function void randomize_transaction();
                if (!this.randomize())
                        $display("Randomization failed");
        endfunction

        function void display(string name);
                $display("\n %s ------------------",name);
                $display("reset_duration = %0d", reset_duration);
                $display("count = %0d, out = %b", count, out);
                $display("------------------------");
        endfunction
endclass
```

This SystemVerilog code defines a `transaction` class for encapsulating stimulus data in a verification environment.
Attributes:
- `reset_duration`: Randomized reset release duration with constraints (2–7).
- `out` and `count`: 3-bit data signals.
- `rst` and `clk`: Control signals.
- Constraints: Enforces a range for `reset_duration`.

Methods:
- `randomize_transaction()`: Randomizes transaction fields, with error handling for failures.
- `display(string name)`: Prints transaction details with a customizable label.

This class models input stimuli, ensuring controlled randomness and easy debugging.

3.f

Driver

Code Snippet

```
class driver;

        virtual intf vif;

        mailbox gen2driv;

        function new(virtual intf vif,mailbox gen2driv);
                this.vif = vif;
                this.gen2driv = gen2driv;
        endfunction

        task reset;
                transaction trans;
                vif.rst     <= 0;
                #(vif.reset_duration * 2 * vif.Tc);
                vif.rst     <= 1;
                #(2 * vif.Tc);
                vif.rst     <= 0;
                $display("============= Driver for rst @ %0d ==============", $time);
        endtask

        task main;
                repeat(1) begin
                        transaction trans;
                        gen2driv.get(trans);

                        trans.count     = vif.count;
                        trans.out       = vif.out;
                        vif.reset_duration = trans.reset_duration;
                        trans.display("Driver");
                end
        endtask
endclass
```

This SystemVerilog `driver` class handles stimulus application and signal driving in a verification environment.
Attributes:
 - `vif`: Virtual interface for signal access.
 - `gen2driv`: Mailbox for receiving transactions from the generator.
Methods:
 - `new`: Constructor initializes the virtual interface and mailbox.
 - `reset`: Drives the reset signal (`rst`) with timing based on `reset_duration` and clock period (`Tc`).
 - `main`: Fetches transactions from `gen2driv`, updates the virtual interface signals, and logs transaction details for debugging.
This class ensures accurate signal driving and interaction with the generator, maintaining synchronization in the testbench.

3.g

Generator

Code Snippet


```
class generator;

        transaction trans; //Handle of Transaction class

        mailbox gen2driv;  //Mailbox declaration

        function new(mailbox gen2driv);  //creation of mailbox and constructor
                this.gen2driv = gen2driv;
        endfunction


        task main();
                repeat(1)
                begin
                        trans = new();
                        trans.randomize();
                        trans.display("Generator");
                        gen2driv.put(trans);
                end
        endtask

endclass
```

The `generator` class is a component in the verification environment responsible for creating and sending randomized transactions to the driver.
Attributes:
- `trans`: A handle for the `transaction` class.
- `gen2driv`: A mailbox for sending transactions to the driver.
Methods:
- `new`: Constructor initializes the mailbox.
- `main`: Creates a new transaction, randomizes its fields, displays the transaction details, and sends it to the `gen2driv` mailbox.
This class ensures the generation of dynamic, randomized stimulus for effective testing.

3.h

Scoreboard

Code Snippet


```
class scoreboard;
        mailbox mon_scb;

        function new(mailbox mon_scb);
                this.mon_scb = mon_scb;
        endfunction

        task main;
                transaction trans;
                repeat(1)
                begin
                        mon_scb.get(trans);

                        if(trans.out == {trans.count[2], trans.count[2] ^ trans.count[1], trans.count[1] ^ trans.count[0]})
                                $display("Result is correct");
                        else if (trans.rst == 1 && trans.out == 0)
                                $display("Result is correct");
                        else
                                $error("Wrong result");

                        trans.display("Scoreboard");
                end
        endtask
endclass
```

Scoreboard

This SystemVerilog scoreboard class validates output signals against expected results in a verification environment.

Attributes:

-mon_scb: Mailbox for receiving transactions from the monitor.

Methods:

- new: Constructor initializes the mailbox.

- main: Retrieves transactions, checks output correctness based on XOR logic, and validates reset behavior (rst).
  Logs results using $display and $error.

- trans.display: Outputs transaction details for debugging.

This class ensures output verification and functional correctness while providing debugging support within the testbench.

3.i

Monitor

Code Snippet

```
class monitor;

        virtual intf vif;
        mailbox mon_scb;

        function new(virtual intf vif,mailbox mon_scb);
                this.vif     = vif;
                this.mon_scb = mon_scb;
        endfunction

        task main;
                repeat(1)
                        begin
                        transaction trans;
                        trans                   = new();
                        trans.rst               = vif.rst;
                        trans.clk               = vif.clk;
                        trans.count             = vif.count;
                        trans.out               = vif.out;
                        mon_scb.put(trans);
                        trans.display("Monitor");
                end
        endtask


endclass
```

This SystemVerilog `monitor` class observes DUT signals and forwards captured transactions to the scoreboard via a mailbox.
Attributes:
- `vif`: Virtual interface for accessing DUT signals.
- `mon_scb`: Mailbox for sending transactions to the scoreboard.

Methods:
- `new`: Constructor initializes the virtual interface and mailbox.
- `main`:
  - Captures DUT signals (`rst`, `clk`, `count`, `out`) and stores them in a `transaction` object.
  - Sends the transaction to the `mon_scb` for further processing by the scoreboard.
  - Logs transaction details using `trans.display` for debugging.

This class ensures accurate signal observation and transaction forwarding, acting as a bridge between the DUT and the scoreboard in the layered testbench.

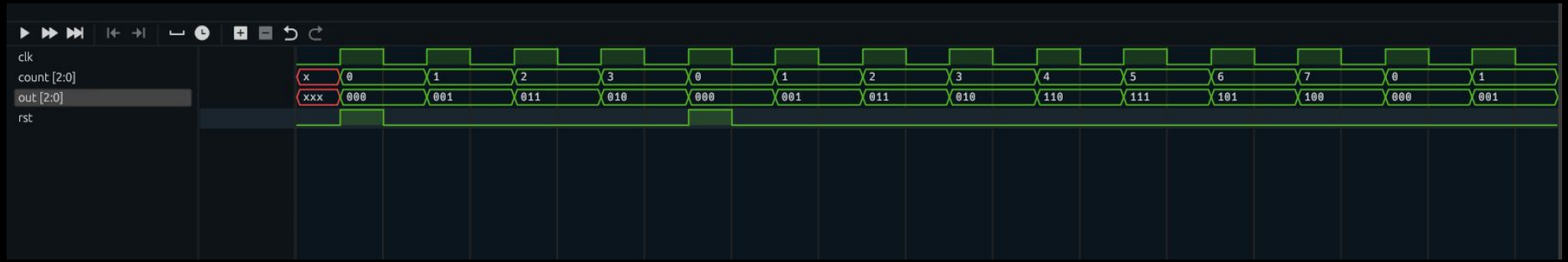
04

Outputs

Simulation Outputs


```
============= Driver for rst @ 2 ===============
============= Generating @ 2 =============

 Generator -------------------
reset_duration = 4
count = x, out = xxx
------------------------

 Driver -------------------
reset_duration = 4
count = 0, out = 000
------------------------
=============== Testing @ 2 ===============

 Monitor -------------------
reset_duration = 0
count = 0, out = 000
------------------------
Result is correct

 Scoreboard -------------------
reset_duration = 0
count = 0, out = 000
------------------------
=============== Testing @ 4 ===============

 Monitor -------------------
reset_duration = 0
count = 1, out = 001
------------------------
Result is correct

 Scoreboard -------------------
reset_duration = 0
count = 1, out = 001
------------------------
=============== Testing @ 6 ===============

 Monitor -------------------
reset_duration = 0
count = 2, out = 011
------------------------
Result is correct

 Scoreboard -------------------
reset_duration = 0
count = 4, out = 110
------------------------
============= Driver for rst @ 12 ===============
=============== Testing @ 12 ===============

 Monitor -------------------
reset_duration = 0
count = 0, out = 000
------------------------
Result is correct

 Scoreboard -------------------
reset_duration = 0
count = 0, out = 000
------------------------
=============== Testing @ 14 ===============

 Monitor -------------------
reset_duration = 0
count = 1, out = 001
------------------------
Result is correct

 Scoreboard -------------------
reset_duration = 0
count = 1, out = 001
------------------------
============= Driver for rst @ 26 ===============
============= Generating @ 26 =============

 Generator -------------------
reset_duration = 6
count = x, out = xxx
------------------------

 Driver -------------------
reset_duration = 6
count = 0, out = 000
```

Waveform


clk
count [2:0]
out [2:0]
rst
x 0 1 2 3 0 1 2 3 4 5 6 7 0 1
xxx 000 001 011 010 000 001 011 010 110 111 101 100 000 001

THANK YOU