Task 2 Verification

overview

The task is to build a complete class based verification environment to verify an encrypted counter design

Specs:

- Positive edge clock
- Asynchronous active low reset
- Begin working at the positive edge of *start*
- If *flag* becomes high, or if *count_value* reaches its maximum, the FSM stops and returns to the idle state
- While running, it asserts *busy*
- increments *count_value* each time a *wait_timer* interval finishes without the *flag* being asserted.

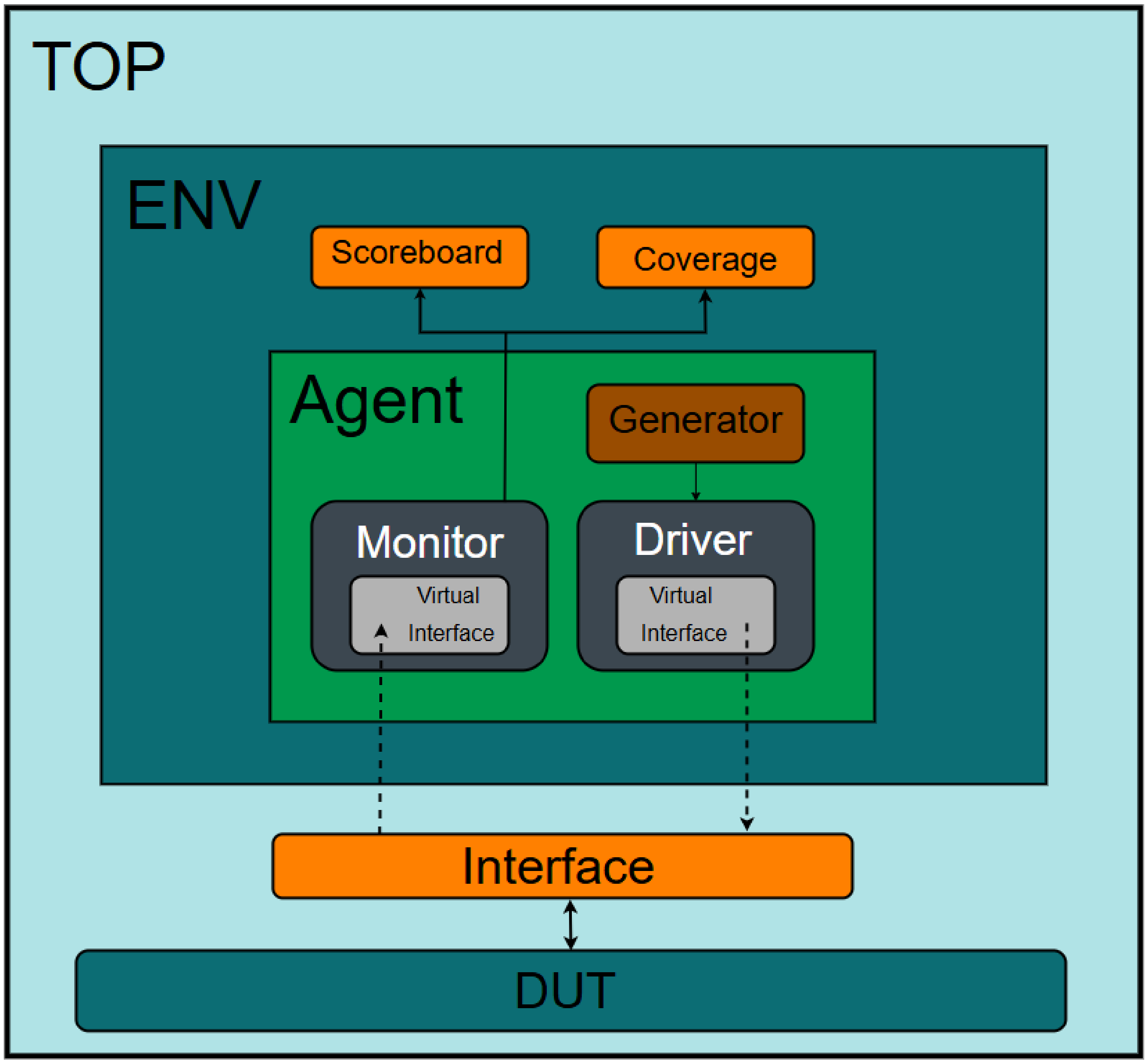
FSM:

- for states we have two states: IDEL, COUNTING
- IDLE:
  - Next state logic: when *start* pulse detected we go to COUNTING
  - Output logic: *busy* is low, *count* is zero, *internal counter* resettled
- COUNTING:
  - Next state logic: when flag or count value reaches max (‘d31) we go back to IDLE
  - Output logic: *busy* is high, *count* signal increments every time *internal counter* hits multiples of the *wait_timer* value (without the flag being asserted, at any multiple of wait_timer we found *flag* is high we stop and don’t increment the *count* value), *internal counter* is incremented with every positive clock edge

Verification Environment

- In this part we will try to verify our design using modular, reusable and scalable class based environment
- We will introduce new and better modifications to our last class based environment
- We will go for each class (top down) and compare it with its pair in our new environment declaring the new modifications and what are the benefits of them

Environment:



ENV:

```
// function to connect seq_item to scoreboard and coverage
    function void connect();
        // connect virtual interface to agent virtual interface
        agt.counter_vif = counter_vif;
        // connect mailbox to scoreboard and coverage
        sb.mon2sb = agt.mon2sb;
        cov.mon2cov = agt.mon2cov;
    endfunction

    task run();
```

```
        agt.connect(); // to connect mailboxes and vifs
        fork
            agt.run();
            sb.run();
            cov.run();
        join_any
        sb.report(); // call report task to print the final count of errors and correct transactions
    endtask
```

- I used join_any rather than join_none, as not to estimate the time to stop in top
- used a reporting task in scoreboard (mimicking report phase in UVM)
- connected virtual interfaces and mailboxes using equal operator rather than functions arguments

Driver and Generator:

```
import counter_seq_item_pkg::*;
class counter_driver;
    string name;

    virtual counter_if counter_vif;
    mailbox #(counter_seq_item) gen2drv;

    event drv_rqt; // driver request event to acknowledge generator
```

```
import counter_seq_item_pkg::*;
class counter_generator;
    string name;
    mailbox #(counter_seq_item) gen2drv;

    event gen_ack; // acknowledgment event to send transactions to driver
```

- I used events between generator and driver to synch up the transaction between them (like port-export TLM in UVM)

Monitor and Interface:

```
interface counter_if(clk);

    // clocking block for monitor to sample data slightly after posedge of clock
    clocking mon_cb @(posedge clk);
        default input #1step output #1step;
        output rst_n, start, wait_timer, flag;
        input busy, count_value;
    endclocking
```

```
          @(counter_vif.mon_cb); // wait for monitor clocking block to sample
data
```

- I used clocking block in the interface to eliminate the dependence on time delay in the sampling moment, as the clocking block gives scalability to our test

Scoreboard:

- I used the FSM modeling using classes and pointers to make it more clear and reusable

Verification plan:

I started with making a simple FSM design (sequence detector) to test the state machine modeling method on it as to be sure that it will work on our target design

You may find the dummy test here:
Class_Based_Verification_Environments/Sequence_Detector_Environment at main · MohamedHussein27/Class_Based_Verification_Environments

in my test I used directed and randomized stimulus

For directed tests:

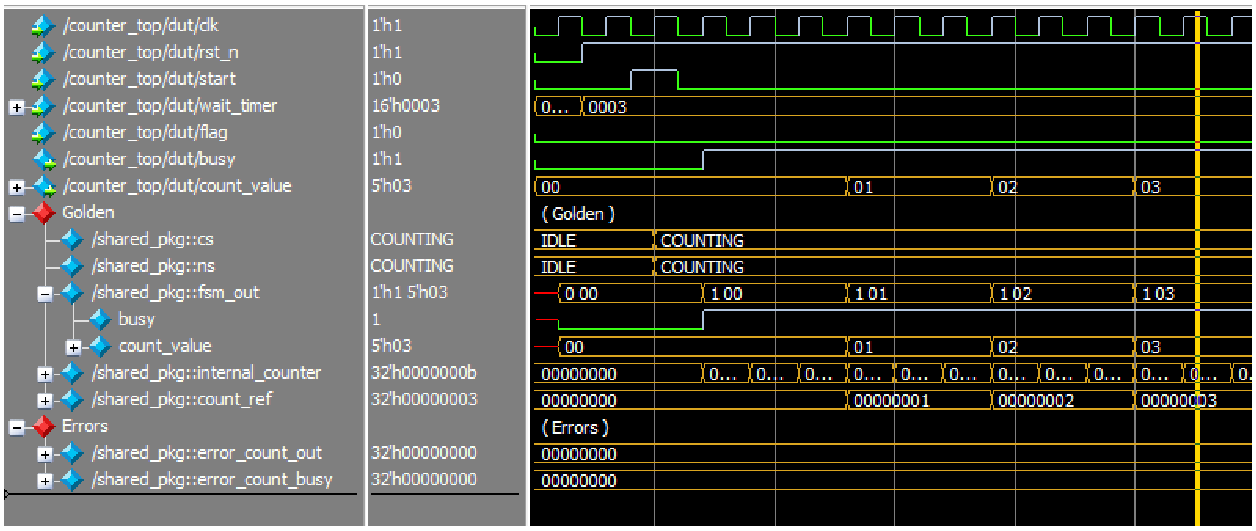
- Test 1: to test the start and that the *count_value* increments every *wait_timer* interval
- Test 2: to put a flag signal and remove it before the time interval of wait_timer ends so design should ignore it
- Test 3: to put a flag signal at when internal_counter equals a multiple of wait_timer so design should go to IDLE state
- Test 4: testing start signal without asserting rst_n before it
- Test 5: testing asserting start signal while normal counting operation, design should ignore it
- Test 6: testing reaching max value of count, design should go to IDLE
- Test 7: testing assert start signal after hitting max value of count, design should work as usual

For Randomization tests:

- Test 8: Randomization, with fixed wait_timer (programmable)

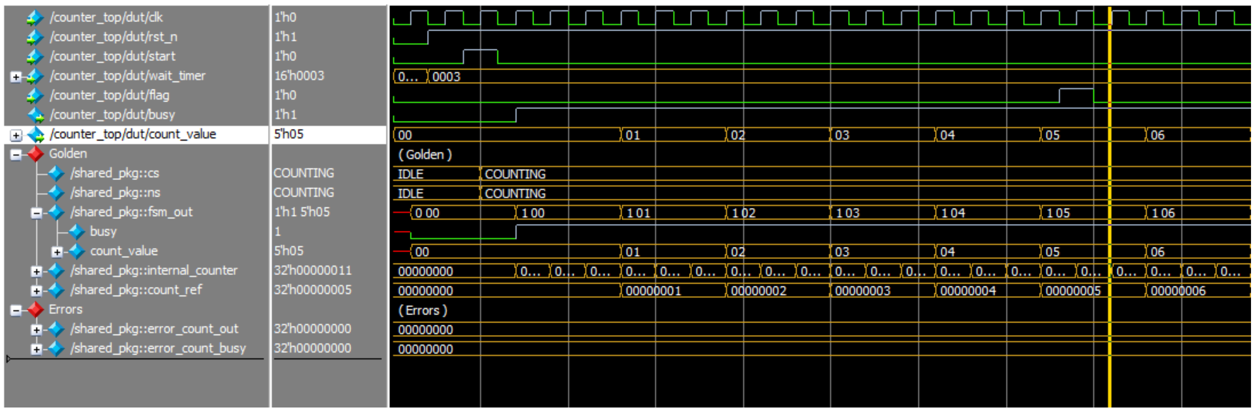
Results:

Test 1: passed, however there was a one cycle delay in the first count unlike after it.



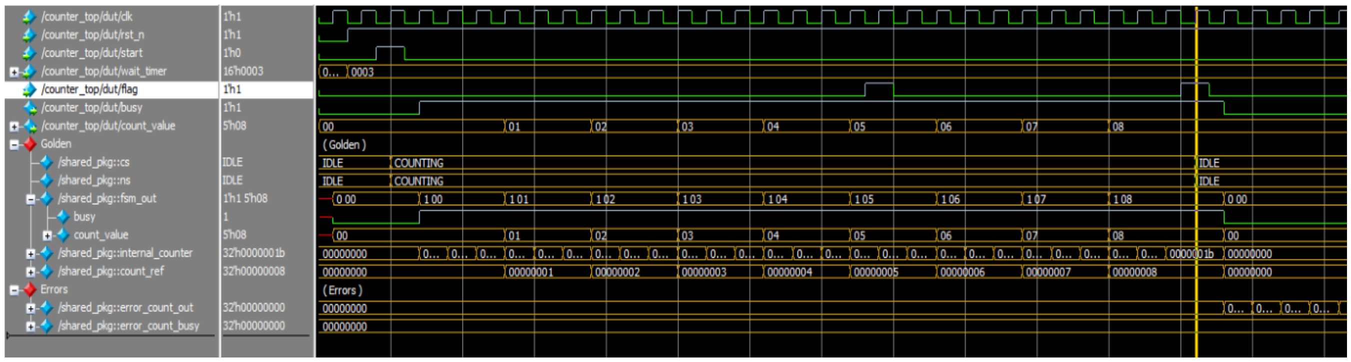
```
/counter_top/dut/clk                  1'h1
/counter_top/dut/rst_n                1'h1
/counter_top/dut/start                1'h0
/counter_top/dut/wait_timer           16'h0003
/counter_top/dut/flag                 1'h0
/counter_top/dut/busy                 1'h1
/counter_top/dut/count_value          5'h03
Golden
  /shared_pkg::cs                     COUNTING
  /shared_pkg::ns                     COUNTING
  /shared_pkg::fsm_out                1'h1 5'h03
    busy                              1
    count_value                       5'h03
  /shared_pkg::internal_counter       32'h0000000b
  /shared_pkg::count_ref              32'h00000003
Errors
  /shared_pkg::error_count_out        32'h00000000
  /shared_pkg::error_count_busy       32'h00000000
```

Test 2: passed, the design did ignore the flag signal



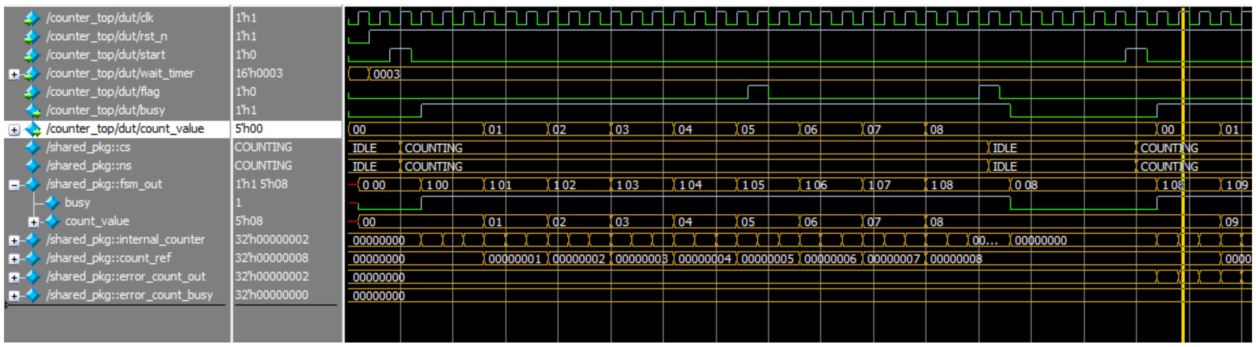
```
/counter_top/dut/clk                  1'h0
/counter_top/dut/rst_n                1'h1
/counter_top/dut/start                1'h0
/counter_top/dut/wait_timer           16'h0003
/counter_top/dut/flag                 1'h0
/counter_top/dut/busy                 1'h1
/counter_top/dut/count_value          5'h05
Golden
  /shared_pkg::cs                     COUNTING
  /shared_pkg::ns                     COUNTING
  /shared_pkg::fsm_out                1'h1 5'h05
    busy                              1
    count_value                       5'h05
  /shared_pkg::internal_counter       32'h00000011
  /shared_pkg::count_ref              32'h00000005
Errors
  /shared_pkg::error_count_out        32'h00000000
  /shared_pkg::error_count_busy       32'h00000000
```

Test 3: final decision: passed, the design stops counting, however I thought when returning back to IDLE the count_value should be zero (specs say stops and return back to IDLE) so I modified my model to stop only and not resetting the count_value as I was not sure



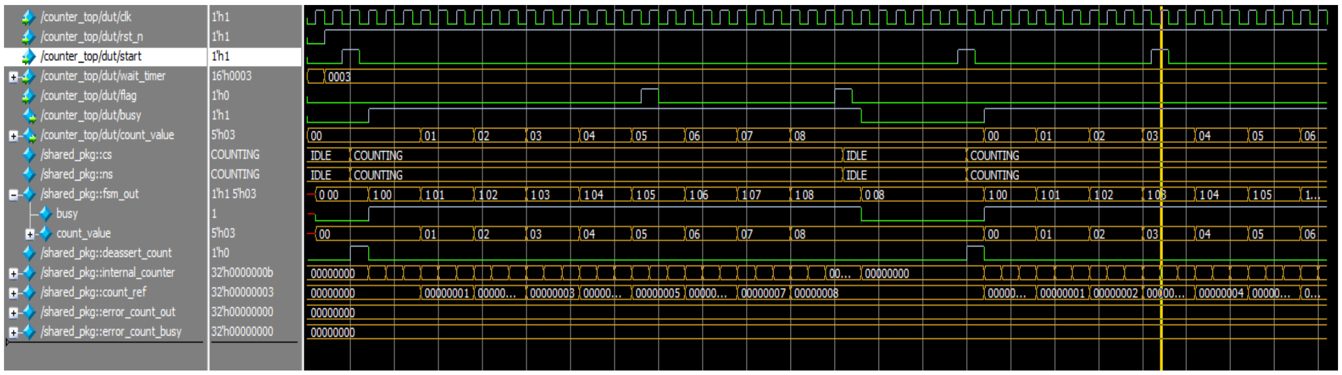
```
/counter_top/dut/clk                  1'h1
/counter_top/dut/rst_n                1'h1
/counter_top/dut/start                1'h0
/counter_top/dut/wait_timer           16'h0003
/counter_top/dut/flag                 1'h1
/counter_top/dut/busy                 1'h1
/counter_top/dut/count_value          5'h08
Golden
  /shared_pkg::cs                     IDLE
  /shared_pkg::ns                     IDLE
  /shared_pkg::fsm_out                1'h1 5'h08
    busy                              1
    count_value                       5'h08
  /shared_pkg::internal_counter       32'h0000001b
  /shared_pkg::count_ref              32'h00000008
Errors
  /shared_pkg::error_count_out        32'h00000000
  /shared_pkg::error_count_busy       32'h00000000
```

Test 4: final decision: passed, first I thought as we stop the count when flag, then we should continue counting when start and not restart the count but spec says: The FSM starts counting when it detects a rising edge on start, so I modified my test again to match it



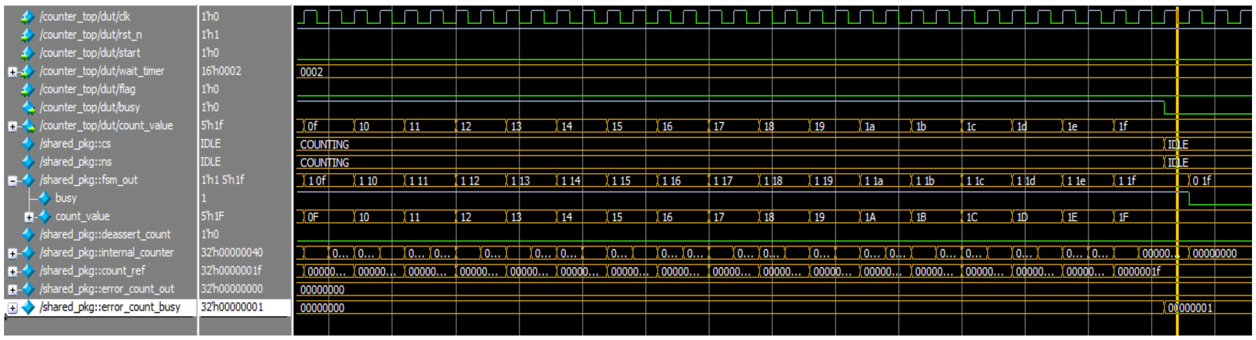
```
/counter_top/dut/clk                 1'h1
/counter_top/dut/rst_n               1'h1
/counter_top/dut/start               1'h0
/counter_top/dut/wait_timer          16'h0003
/counter_top/dut/flag                1'h0
/counter_top/dut/busy                1'h1
/counter_top/dut/count_value         5'h00
/shared_pkg::cs                      COUNTING
/shared_pkg::ns                      COUNTING
/shared_pkg::fsm_out                 1'h1 5'h08
  busy                               1
  count_value                        5'h08
/shared_pkg::internal_counter        32'h00000002
/shared_pkg::count_ref               32'h00000008
/shared_pkg::error_count_out         32'h00000002
/shared_pkg::error_count_busy        32'h00000000
```

Test 5: passed; design did ignore the start signal while operating



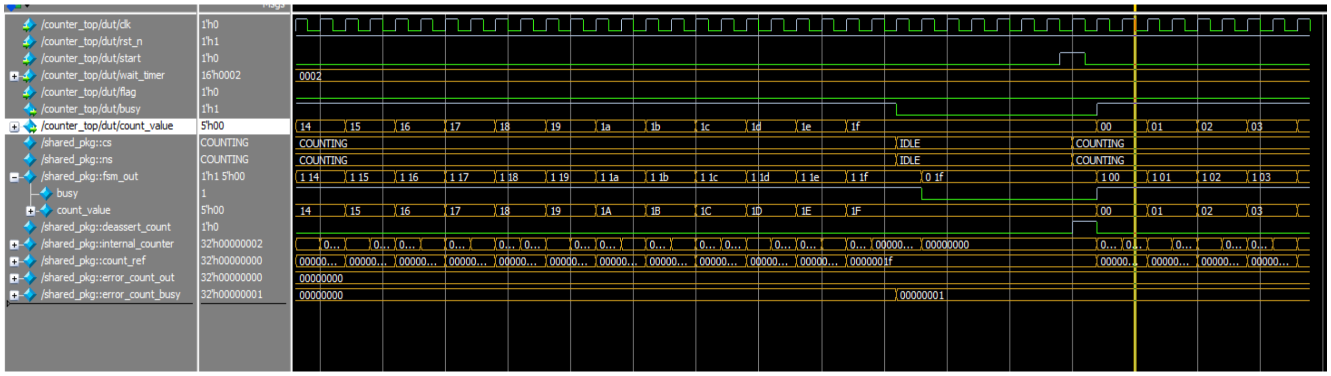
```
/counter_top/dut/clk                 1'h1
/counter_top/dut/rst_n               1'h1
/counter_top/dut/start               1'h1
/counter_top/dut/wait_timer          16'h0003
/counter_top/dut/flag                1'h0
/counter_top/dut/busy                1'h1
/counter_top/dut/count_value         5'h03
/shared_pkg::cs                      COUNTING
/shared_pkg::ns                      COUNTING
/shared_pkg::fsm_out                 1'h1 5'h03
  busy                               1
  count_value                        5'h03
/shared_pkg::deassert_count          1'h0
/shared_pkg::internal_counter        32'h0000000b
/shared_pkg::count_ref               32'h00000003
/shared_pkg::error_count_out         32'h00000000
/shared_pkg::error_count_busy        32'h00000000
```

Test 6: failed; design did go back to IDLE but de-asserted busy signal one cycle early; I've changed the wait_timer value to 2, **this is the bug.**



```
/counter_top/dut/clk                 1'h0
/counter_top/dut/rst_n               1'h1
/counter_top/dut/start               1'h0
/counter_top/dut/wait_timer          16'h0002
/counter_top/dut/flag                1'h0
/counter_top/dut/busy                1'h0
/counter_top/dut/count_value         5'h1f
/shared_pkg::cs                      IDLE
/shared_pkg::ns                      IDLE
/shared_pkg::fsm_out                 1'h1 5'h1f
  busy                               1
  count_value                        5'h1F
/shared_pkg::deassert_count          1'h0
/shared_pkg::internal_counter        32'h00000040
/shared_pkg::count_ref               32'h0000001f
/shared_pkg::error_count_out         32'h00000000
/shared_pkg::error_count_busy        32'h00000001
```

Test 7: **passed**, design went back to COUNTING state

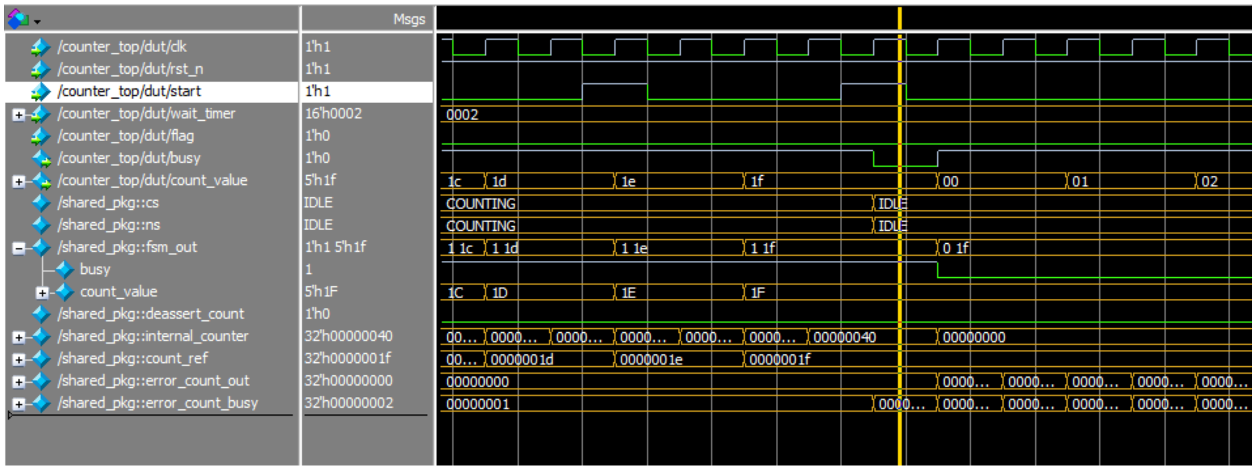


/counter_top/dut/clk 1'h0
/counter_top/dut/rst_n 1'h1
/counter_top/dut/start 1'h0
/counter_top/dut/wait_timer 16'h0002
/counter_top/dut/flag 1'h0
/counter_top/dut/busy 1'h1
/counter_top/dut/count_value 5'h00
/shared_pkg::cs COUNTING
/shared_pkg::ns COUNTING
/shared_pkg::fsm_out 1'h1 5'h00
busy 1
count_value 5'h00
/shared_pkg::deassert_count 1'h0
/shared_pkg::internal_counter 32'h00000002
/shared_pkg::count_ref 32'h00000000
/shared_pkg::error_count_out 32'h00000000
/shared_pkg::error_count_busy 32'h00000001

Transcript:

```
# error in fsm output, ref_busy is: 1    while dut busy is: 0
# error count out = 0, correct count out = 142
# error count busy = 1, correct count busy = 141
```

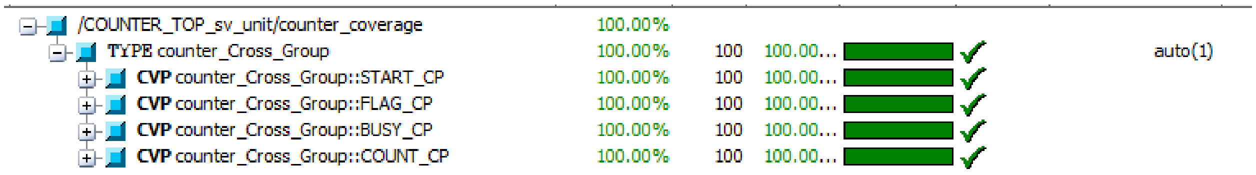
Test 8: From randomization I highlighted a case stimulus:



Msgs
/counter_top/dut/clk 1'h1
/counter_top/dut/rst_n 1'h1
/counter_top/dut/start 1'h1
/counter_top/dut/wait_timer 16'h0002
/counter_top/dut/flag 1'h0
/counter_top/dut/busy 1'h0
/counter_top/dut/count_value 5'h1f
/shared_pkg::cs IDLE
/shared_pkg::ns IDLE
/shared_pkg::fsm_out 1'h1 5'h1f
busy 1
count_value 5'h1F
/shared_pkg::deassert_count 1'h0
/shared_pkg::internal_counter 32'h00000040
/shared_pkg::count_ref 32'h0000001f
/shared_pkg::error_count_out 32'h00000000
/shared_pkg::error_count_busy 32'h00000002

- This is the bug we talked about in test 6, and as a consequence here we find that start is asserted while busy is deasseted (bug), then design does not ignore it and this causes a significant dynamic mismatch!
- This assures us that the design has a consecutive behavior

Functional Coverage:



/COUNTER_TOP_sv_unit/counter_coverage	100.00%				
TYPE counter_Cross_Group	100.00%	100	100.00...	✓	auto(1)
CVP counter_Cross_Group::START_CP	100.00%	100	100.00...	✓	
CVP counter_Cross_Group::FLAG_CP	100.00%	100	100.00...	✓	
CVP counter_Cross_Group::BUSY_CP	100.00%	100	100.00...	✓	
CVP counter_Cross_Group::COUNT_CP	100.00%	100	100.00...	✓	

Idea regarding FSM modeling:

I had an idea for state machine modeling regarding next state and current state, we monitor the outputs at positive edge of clock, but this is not the case when modeling next state logic, so we could split the monitor transaction using fork and send even the inputs only at the negative edge after driver sent its transaction to the dut, and by using those inputs to the scoreboard state machine classes which by their turn will determine the next state logic, then at positive edge (regular monitoring event) we now assign the next state handle to the current state handle and by this approach we can model the state machine logic exactly.