Winter 2023

COEN 6541: FUNCTIONAL HARDWARE VERIFICATION

Dr. Otmane Ait Mohamed

CALC_2 TEST PLAN PROJECT REPORT

Date: 16-04-2023

Submitted by:

Prithvik Adithya Ravindran - 40195464
Avaneesh Reddy Gaddam - 40217009
Irfanul Islam - 40207134
Sonam Zam Guddati - 40237335

TABLE OF CONTENTS

1. ABSTRACT

This technical report presents the verification of an RTL design implementation of a four-operation calculator. The RTL design was implemented using Verilog HDL and simulated using QuestaSim. The design was verified by performing various tests on the calculator's arithmetic operations. The verification tests were successful for a couple of test cases and failed for a few. This report presents the scenarios for which the calculator design may require further development and testing.

2. INTRODUCTION

In the previous project, we implemented Calc1. Calc1 is an RTL design implementation that has four basic operations. The four operations include:

- Add
- Subtract
- Shift Left
- Shift Right

The calculator operates when a "requestor" sends an operator through the command input bus followed by the data of the operand. It has four input ports & each "requestor" may send the command & data through one of the four ports. They all have equal priority & each port must wait for the previous response before sending the next comman4. Moreover, all the ports can handle a single command parallelly. Each command will return a response with the result, unless it is an error condition. For this project, Calc2, is much like Calc1, except that each port may now have up to 4 outstanding commands at a time unlike Calc1 which can handle only one single command per port. So, a total of 16 commands can be handled at a tim5. Each command from a port is sent one at a time, but the calculator log may respond "out of order" depending upon the internal state of the queues. A 2-bit tag is added with the command, provided to track the corresponding data obtained at the output.

3. PROJECT DESCRIPTION

A. CALC2 DESIGN DESCRIPTION

The figure below shows the design for Calc1 with a clock signal.

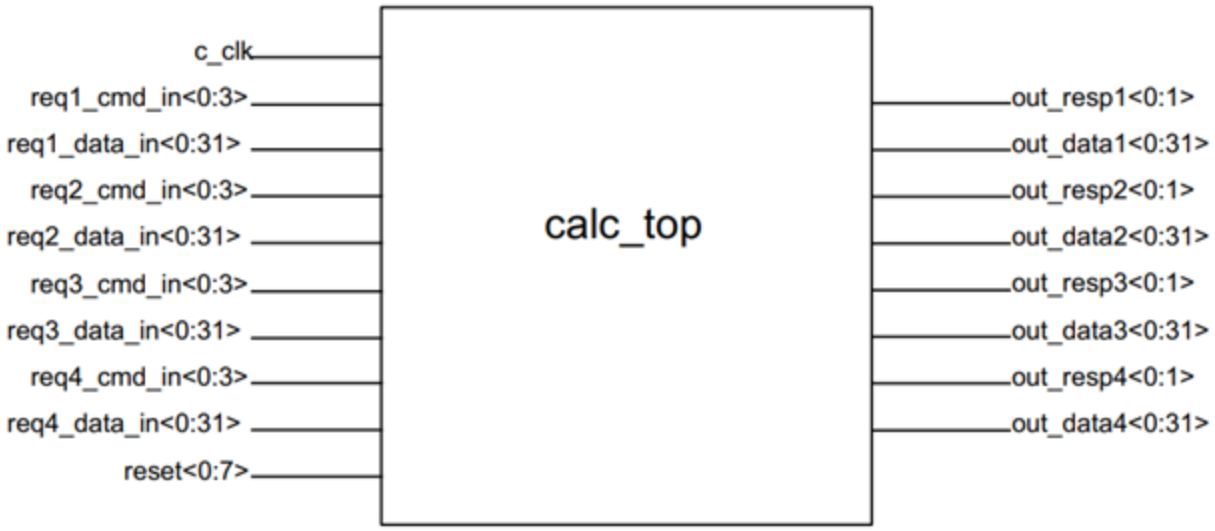


Fig 1. Calc1 design

The calculator has 4 different input ports consisting of a command bus & a data bus, with a clock & a reset bus. On the output side, the calculator has 4 output ports with their own response bus. The figure below shows the design for Calc2

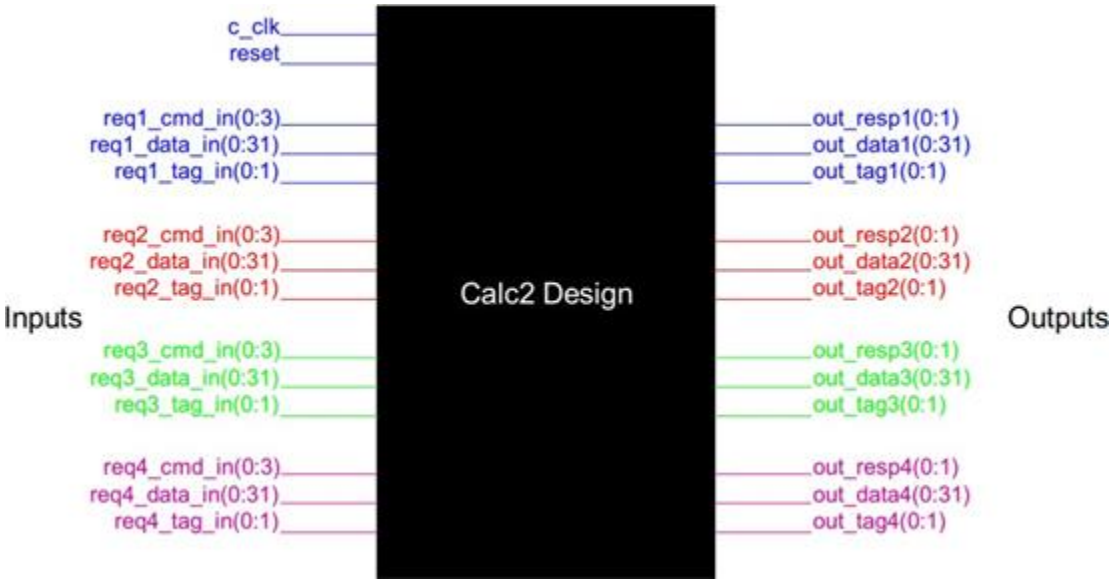


Fig 2. Calc2 design

The difference between Calc1 & Cacl2 are the added ports of input tag & output tag which help identify the command sent.

B. EXPECTED FUNCTIONALITY

The function of each of the Calc1 ports is given below.

1. **c_clk:** Gives the clock signal to drive the input command, data/get the output data, response.
2. **reqX_cmd_in<0:3>:** It is the command bit that defines the operation to be performed with the data provided where

   '0' specifies **No Operation**

   '1' specifies **addition** of Operand 1 and Operand 2

   '2' specifies **subtraction** of Operand 1 and Operand 2 **'5'** specifies **Shift left** Operand 1 by Operand 2 places **'6'** specifies **Shift right** Operand 1 by Operand 2 places

3. **reqX_data_in<0:31>:** Operand 1 data comes in with a command bit and Operand 2 arrives after the following cycl5.
4. **reset<0:7>:** Resets the data to '1111111'b at the start of the test case for seven cycles. Outputs are ignored during this period.
5. **out_respX<0:1>:** 2-bit output response bits defines the operation performed at the output where

   '0' specifies **No response**

   '1' specifies **successful operation completion**

   '2' specifies **invalid command or overflow/underflow error '1'** specifies **internal error**

6. **out_dataX<0:31>:** Gives corresponding valid data accompanied by respons5.

For Calc2 all the ports work in the same way as Calc1 except for the two new added ports.

1. **reqX_tag_in(0:1):** The tag bus is the two-bit identifier for each command from the port. The port requestor can reuse the tag as soon as the Calc2 HDL responds to the comman4.

2. **out_tagX(0:1):** The output tag bus corresponds to the command tag sent by the requester. It is used to identify which command the response if for.

Each command from a port is sent in serially (one at a time) but the calculator log may respond "out of order," depending upon the internal state of the queues. As a result, each command is now accompanied by a 2-bit tag, which identifies the command when the response is received.

The following timing diagram shows when an input command signal is sent through input ports followed by a 2 bit-tag which identifies this command.

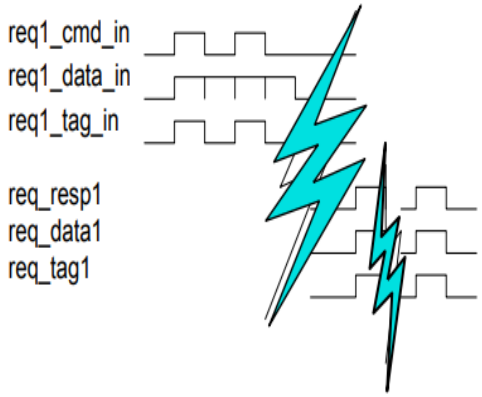


Fig 3. Timing diagram 1

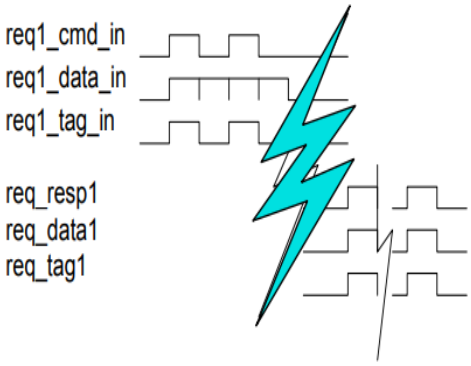


Fig 4. Timing diagram 2

C. VERIFICATION TESTING REQUIREMENTS

Calc2 has a complex design & requires a verification test bench design. Random testing method is used to verify the different operations & ports functioning properly. The verification test bench method used is shown below.

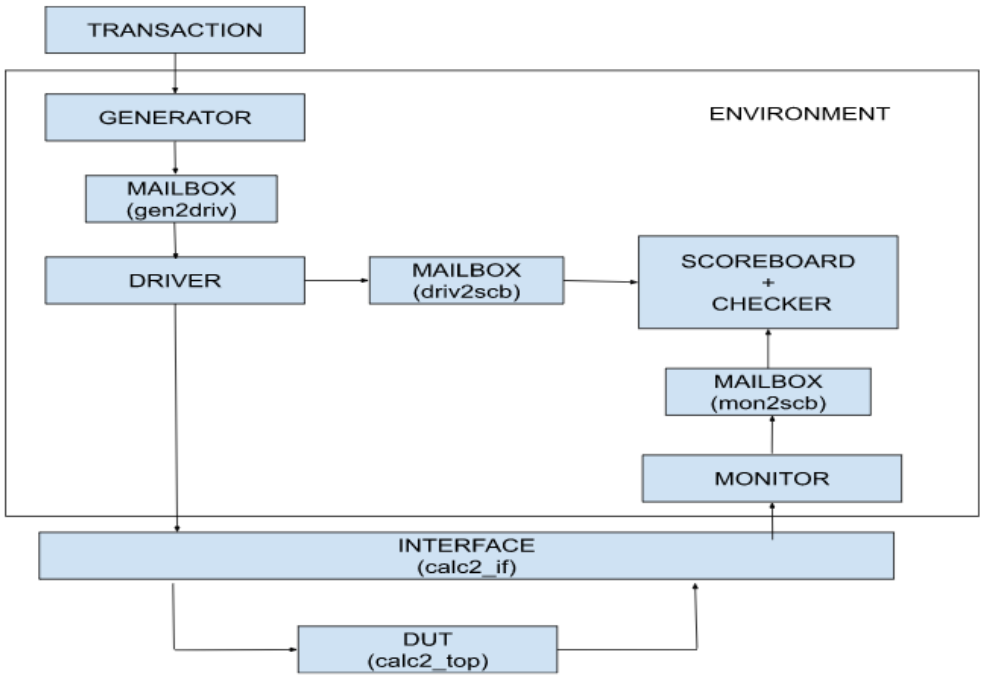


Fig 5. Test Bench

Transaction

The transaction class was created to generate the randomized stimulus. The stimulus can be controlled to get the required values/range neede4. The command and tag bits are also generated in this class for all the ports. The expected tag bits, captured tag bits, output data bits, expected data bits, output response bits, expected response bits, are also declared in this class. Copying current random values for constraints not to repeat the values are also done in this class.

Generator

Generator class that we created randomizes the transaction class that is coded to create multiple input stimuli to be driven to DUT via driver class. This class adds a control to the loop, the number of iterations done by the loop is controlled by declaring a variable. Also displays whether the randomization succeeded or not. Mailbox, gen2drive is generated in this class to send the randomized stimulus from the generator class to the driver. An event is also declared to note the completion of transactions. The

Interface

Multiple signals in a device are placed in a single band called interface for easy handling, reusability of the input-output signals. In the interface class, all the input-output signals are declare4. Clocking blocks for the monitor and the test class are also declared at the posedge of the clock in this class. Also, the direction of the signals for monitor, DUT and test are declared in the modport in this class.

Driver

The driver class creates a mailbox driv2scb in addition to the mailbox gen2driv. A custom constructor is created in order to pass the values of the mailbox. The instance of the virtual interface is declared in this block. Timing information is defined within the task provided in the class for all the 4 ports. Reset is declared in this class. Expected data, expected response and expected tag bits are calculated in this class. The randomized stimulus is driven to the DUT.

Monitor

The monitor class creates a new virtual interface intf2 and mailbox mon2sc2. A custom constructor is used to pass the values of the interface and mailbox to the scoreboard. Output data, Output response and output tag bits are captured in this class for all the 4 ports from the DUT. All the tasks to capture the data, response and tags are parallel.

Scoreboard Checker

Scoreboard gets the expected data, response and tag bits from the driv2scb mailbox using the get() function and stores them in it. It also gets the captured data from the DUT through the mon2scb mailbox and compares them. Here as we have out of order execution of input data depending on the commands. So we need to store the previous out tag, out response and out data values of the previous iteration. For this purpose we are pushing the data into the queue whenever the tags are not matching. This is shown in the code snippet below.

```
if (exp_tag1 == mon_tr.out_tag1)
  begin
  if ((exp_data1[1:32] == mon_tr.out_data1) && (exp_res1 == mon_tr.out_resp1))
     $display("@%0d: Data match---port 1- expected_data1=%H monitor_Data1=%H
  else begin
  $display("@%0d: *** ERROR *** expected_data1=%H monitor_Data1=%H ; expected
  error_count++;
  end
  end
else
   begin
     $display("@%0d: Tag didn't match for port 1---sending data into queue ",
     qa1.push_front(mon_tr.out_tag1);
     qb1.push_front(exp_tag1);
     qc1.push_front(mon_tr.out_data1);
     qd1.push_front(exp_data1[1:32]);
     qe1.push_front(mon_tr.out_resp1);
     qf1.push_front(exp_res1);
   end
```

Code Snippet.1

If the tag of a port is not matching the data it is sent into the queue. So in the next iteration, if the tag gets matched, the output tag value in the previous iteration is wrong and the data in the queue is flushed. If the tag is not matched, it is sent into the queue, where it is compared by the previous tag and the data validity is checked.It can be understood from the code snippet below.

```
if ((qb1.size() > 1) && (qb1.size() < 3) )
begin
  for (int i = 0; i < qb1.size(); i = i + 1)
  begin
    if (qb1[i] == mon_tr.out_tag1)
    begin
        if ((qc1[0] == qd1[i]) && (qe1[0] == qf1[i]))begin
          $display("@%0d: Data match--port 1-- expected_data1=%H monitor_
          qb1.pop_front();
          qd1.pop_front();
          qf1.pop_front();
          qa1.pop_front();
          qc1.pop_front();
          qe1.pop_front();
          break;
        end
        else  begin
        $display("@%0d: *** ERROR *** expected_data1=%H monitor_Data1=%H
        error_count++;
      end
  end
  qb1.shuffle(); // Move the first element to the back of the queue
  qd1.shuffle();
  qf1.shuffle();
```

Code Snippet.2

Here qb1 is the tag queue for port 1, it is fixed and will be compared with values and expected tag till a comparison output is displayed. In a queue there can never be more than two

elements as there are only four tag options possible. If a queue has more than two elements, it is declared to be corrupted and all values are dumped.

Environment

Instances of all the classes are created and components are connected in this class. Generator, Scoreboard and driver iterations are done in this and displaye4. All the components are run in parallel in this class using a fork-join.

4. TOOLS USED

The software QuestaSim is used for the simulation & verification of Calc2.

5. TEST PLAN

1. TEST PLAN FOR BASIC FUNCTION TESTS:

S.No.	Test Case Description
1.1	Check the basic command-response protocol on each of the four ports.
1.2	Check overflow and underflow cases for add and subtract commands.

Table 1. Basic function test plan

2. TEST PLAN FOR ADVANCED FUNCTION TESTS:

S.No.	Test Case Description
2.1	For each port, check that each command can have any command follow it without leaving the state of the design dirty such that the following command is corrupted.
2.2	Across all ports (eg. four concurrent ADDs doesn't interfere with each other) check that each command can have any command follow it without leaving the design dirty, such that the following command is corrupted.

Table 2. Advanced test plan

3. TEST PLAN FOR ADVANCED FUNCTION TESTS - DATA DEPENDENT CASES:

S.No.	Test Case Description
3.1	Add two numbers that overflow by 1 ("FFFFFFFF"X + 1).
3.2	Add two numbers whose sum is "FFFFFFFF"X.
3.3	Subtract two equal numbers.
3.4	Check that the design ignores data inputs unless the data are supposed to be valid (concurrent with the command and the following cycle). Remember that "00000000"X is a data value just as any other 32- bit combination. Here, the check must include verifying that the design latches the data only when appropriate, and does not key off nonzero dat1.

Table 3. Data dependent case test plan

4. TEST PLAN FOR GENERIC FUNCTION TESTS AND CHECKS:

S.No.	Test Case Description
4.1	Check that the reset function correctly resets the design.

Table 4. Generic function test plan

5. ADVANCED TEST CASES

S.No.	Test Case Description
5.1	Send multiple invalid commands.
5.2	Send only a mix of add and subtract commands to fill the add queue.
5.3	Send only a mix of shift commands to fill the shift queue.
5.4	Verify mixes of overflow, underflow and good response cases across back to back port commands.
5.5	Verify the DUV does not allow output collisions from both pipelines sending results to the same port simultaneously.

5.6	Check for unmatched tags on the command port.
5.7	Check for correctness of out-of-order responses across command pipeline types but never across the same command type.
5.8	Verify in-order execution of all adds/subtracts or shifts, no matter which port sent the command.
5.9.1	Check that the response tag matches the data for the command that was sent
5.9.2	Check that there are no unexpected or stray values on the output.

Table 5. Advanced Test Cases

6. VERIFICATION RESULTS

The verification test results are given below and also attached in the Excel sheet along with the report.

Test Scenarios :-

1.Basic function test cases

1.1 Check the basic command-response protocol on each of the four ports.

```
# No of Iterations: 1
# [Driver] : req1_data_in1 : c43dccbd    req1_data_in2 : a2a012bc        req1_cmd_in :2        req1_tag_in:0    @950
# [Driver] : req2_data_in1 : a3b7dd3f    req2_data_in2 : b5b05c05        req2_cmd_in :2        req2_tag_in:3    @950
# [Driver] : req2_data_in1 : 57269fd3    req3_data_in2 : 564ee03c        req3_cmd_in :6        req3_tag_in:1    @950
# [Driver] : req4_data_in1 : 85065d9b    req4_data_in2 : 9a672e06        req4_cmd_in :5        req4_tag_in:0    @950
#
```

Fig 6. Driver driving the inputs to the DUT.

The driver drives the input to the DUT and it can be seen in the transcript by calling the display function from the driver class.

```
@1200: Data match---port 1- expected_data1=219dba01 monitor_Data1=219dba01 ; expected_res1 =  1 monitor_res1 = 1; expected tag =0 and monitor tag = 0;
@1200: Data match---- expected_data2=ee07813a monitor_Data2=ee07813a ; expected_res2 =  2 monitor_res2 = 2; expected tag =3 and monitor tag = 3;
@1200: Data match---- expected_data3=00000005 monitor_Data3=00000005 ; expected_res3 =  1 monitor_res3 = 1; expected tag =1 and monitor tag = 1;
@1200: *** ERROR *** expected_data4=419766c0 monitor_Data4=419766c0 ; expected_res4 =  2 monitor_res4 = 1; expected tag =0 and monitor tag = 0;
```

Fig 7. Scoreboard results after comparison with monitor data.

The above results showcase the fact that port1, port 3, port 4 are performing basic commands and receiving apt response from the DUT and scoreboard is verifying it in return.

```
# No of Iterations: 12
# [Driver] : req1_data_in1 : c78c6bf4    req1_data_in2 : 2ebba57         req1_cmd_in :2        req1_tag_in:2    @5750
# [Driver] : req2_data_in1 : a34db331    req2_data_in2 : 44826d2b        req2_cmd_in :5        req2_tag_in:1    @5750
# [Driver] : req2_data_in1 : 1043a835    req3_data_in2 : e8ce5d1a        req3_cmd_in :1        req3_tag_in:0    @5750
# [Driver] : req4_data_in1 : 5fce48c8    req4_data_in2 : ffc2c949        req4_cmd_in :2        req4_tag_in:2    @5750
#
```

Fig 8. Driver driving the inputs to the DUT.

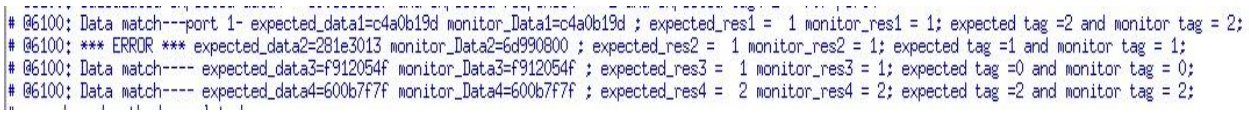


Fig 9. Scoreboard results after comparison with monitor data.

The above results show that port 2 performing ADD operation and giving appropriate response according to the scoreboard.

1.2 Check overflow and underflow operation for ADD and SUB operations.

Fig 10. Driver driving the inputs to the DUT.

The driver drives the input to the DUT and it can be seen in the transcript by calling the display function from the driver class.

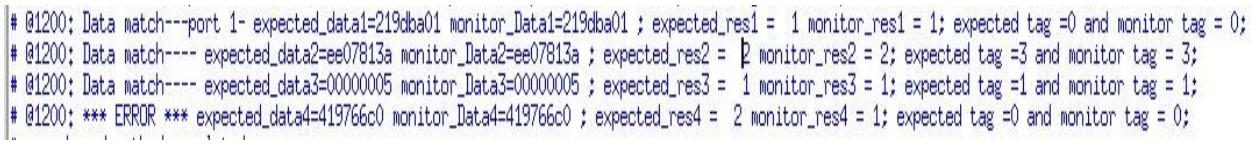


Fig 11. Scoreboard results after comparison with monitor data.

The comparison between the scoreboard and monitor results establish the underflow that's taking place in port2 for the subtraction operation. The underflow operation seems to work for all the ports.

Fig 12. Scoreboard results after comparison with monitor data.

Similarly, when the overflow operation is studied in a different iteration we can see that for the command 1 which is addition its overflow for the given operands and the response should be 2 but the monitor response is 1 which is wrong. Hence, there is an error in port 4 where the overflow data is depicted in a wrong way.

2.Generic function test cases

2.1 For each port, check that each command can have any command follow it without leaving the state of the design dirty such that the following command is corrupted.

```
# No of Iterations: 13
# [Driver] : req1_data_in1 : 52d3e272    req1_data_in2 : 18e9773d        req1_cmd_in :5          req1_tag_in:2    @6350
# [Driver] : req2_data_in1 : 6cec3951    req2_data_in2 : 75f71e77        req2_cmd_in :1          req2_tag_in:3    @6350
# [Driver] : req3_data_in1 : 25b1ee8a    req3_data_in2 : 166332af        req3_cmd_in :1          req3_tag_in:1    @6350
# [Driver] : req4_data_in1 : 4382a5a9    req4_data_in2 : 7ab73418        req4_cmd_in :2          req4_tag_in:1    @6350
#
```

Fig 13. Driver driving the inputs to the DUT.

```
# No of Iterations: 14
# [Driver] : req1_data_in1 : 94bb044b    req1_data_in2 : a09305a6        req1_cmd_in :1          req1_tag_in:0    @6750
# [Driver] : req2_data_in1 : 36a23aa4    req2_data_in2 : 2ab82ae8        req2_cmd_in :5          req2_tag_in:2    @6750
# [Driver] : req3_data_in1 : f3966f76    req3_data_in2 : bdaca06a        req3_cmd_in :5          req3_tag_in:0    @6750
# [Driver] : req4_data_in1 : c463a365    req4_data_in2 : 6213702a        req4_cmd_in :6          req4_tag_in:1    @6750
#
```

Fig 14. Driver driving the inputs to the DUT.

Figure 13 and Figure 14 shows how the data is being driven by the master to the DUT by command by command without leaving the state of the design dirty such that the following command is corrupted.

```
# @6500: *** ERROR *** expected_data1=40000000 monitor_Data1=00000000 ; expected_res1 =  1 monitor_res1 = 2; expected tag =2 and monitor tag = 2;
# @6500: *** ERROR *** expected_data2=e2e357c8 monitor_Data2=00000000 ; expected_res2 =  1 monitor_res2 = 2; expected tag =3 and monitor tag = 3;
# @6500: Data match---- expected_data3=3c152139 monitor_Data3=3c152139 ; expected_res3 =  1 monitor_res3 = 1; expected tag =1 and monitor tag = 1;
# @6500: *** ERROR *** expected_data4=c8cb7191 monitor_Data4=00000000 ; expected_res4 =  2 monitor_res4 = 2; expected tag =1 and monitor tag = 1;
```

Fig 15. Scoreboard results after comparison with monitor data.

Figure 15, shows the scoreboard calculation and compared data which clearly shows that the command after command can be fed to the design without leaving the design corrupted. Two iterations are taken for the comparison such as iteration 13 and 14.

2.2Across all ports (eg. four concurrent ADDs doesn't interfere with each other) check that each command can have any command follow it without leaving the design dirty, such that the following command is corrupted.

```
run
# No of Iterations: 1
# [Driver] : req1_data_in1 : c43dccbd    req1_data_in2 : a2a012bc        req1_cmd_in :1          req1_tag_in:0    @950
# [Driver] : req2_data_in1 : a3b7dd3f    req2_data_in2 : b5b05c05        req2_cmd_in :1          req2_tag_in:3    @950
# [Driver] : req3_data_in1 : 57269fd3    req3_data_in2 : 564ee03c        req3_cmd_in :1          req3_tag_in:1    @950
# [Driver] : req4_data_in1 : 85065d9b    req4_data_in2 : 9a672e06        req4_cmd_in :1          req4_tag_in:0    @950
#
#
```

Fig 16. Driver driving the inputs to the DUT.

Four concurrent Add's are being driven into the DUT by simplifying modifying the constraints and this helps in achieving and testing the above test scenario.

```
# @1400; Data match---port 1- expected_data1=66dddf79 monitor_Data1=66dddf79 ; expected_res1 =  2 monitor_res1 = 2; expected tag =0 and monitor tag = 0;
# @1400; Data match---- expected_data2=59683944 monitor_Data2=59683944 ; expected_res2 =  2 monitor_res2 = 2; expected tag =3 and monitor tag = 3;
# @1400; Data match---- expected_data3=ad75800f monitor_Data3=ad75800f ; expected_res3 =  1 monitor_res3 = 1; expected tag =1 and monitor tag = 1;
# @1400; Data match---- expected_data4=1f6d8ba1 monitor_Data4=1f6d8ba1 ; expected_res4 =  2 monitor_res4 = 2; expected tag =0 and monitor tag = 0;
# scoreboard method completed
```

Fig 17. Scoreboard results after comparison with monitor data.

The Data here is enough to prove the above test scenario where by giving four concurrent commands the design can perform each without interfering with each other.

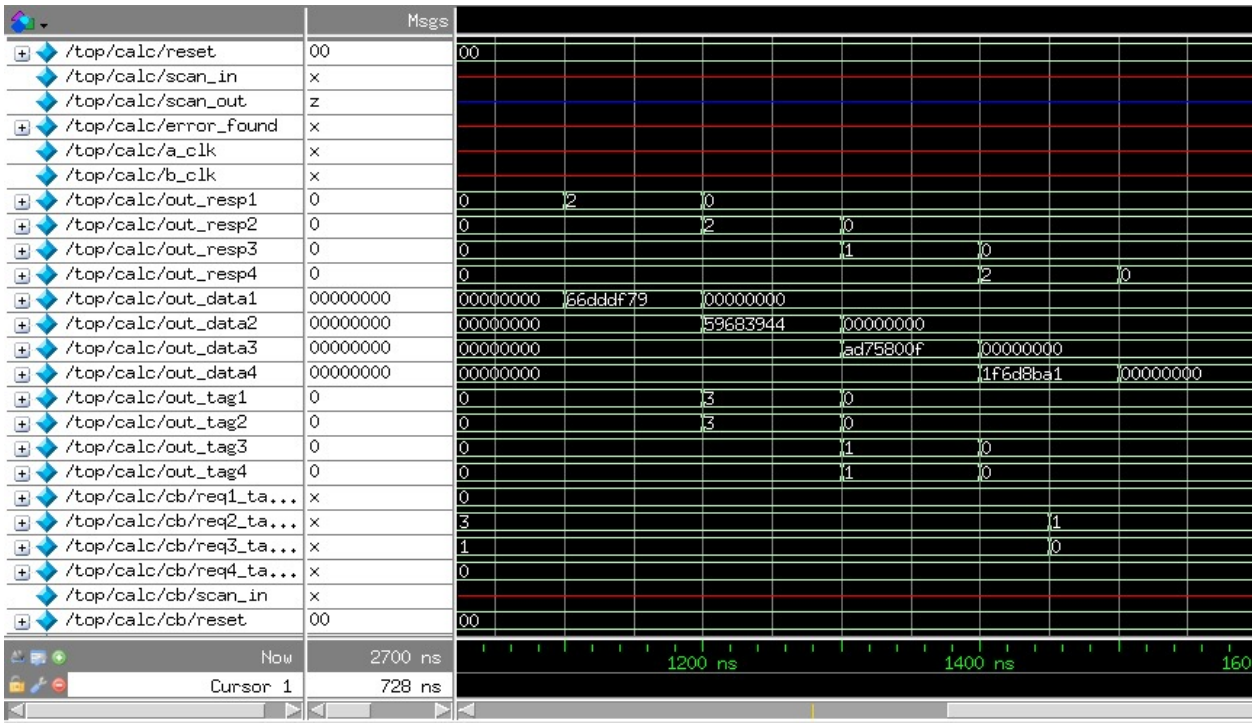


Fig 18.Waveform obtained for the above test scenario.

3. Data dependent test cases

3.1 - Add two numbers that overflow by 1 ("FFFFFFFF"+ "00000001")
3.2- Add two numbers whose sum is "FFFFFFFF" X.
3.3- Subtract two equal numbers

3.4-Check that the design ignores data input unless the data are supposed to be valid. (concurrent with the command and the following cycle). Remember that "00000000"X is a data value just as any other 32-bit combination. Here, the check must include verifying that the design latches the data only when appropriate, and does not key off non zero data.

```
# No of Iterations: 1
# [Driver] : req1_data_in1 : ffffffff   req1_data_in2 : 5b91bac2      req1_cmd_in :1       req1_tag_in:0   @950
# [Driver] : req2_data_in1 : ffffffff   req2_data_in2 : ffffffff      req2_cmd_in :2       req2_tag_in:3   @950
# [Driver] : req3_data_in1 : ffffffff   req3_data_in2 : 79eef2eb      req3_cmd_in :1       req3_tag_in:1   @950
# [Driver] : req4_data_in1 : 0   req4_data_in2 : 0      req4_cmd_in :1       req4_tag_in:0   @950
#
```

Fig 19.Driver driving the inputs to the DUT.

3.1 Test scenario is being implemented in the port where overflow is studied and in the subsequent iteration other ports were also able to showcase this behavior.
3.2 Test scenario two numbers whose sum is "FFFFFFFF"X is added and can be seen in the port 3 and in the subsequent iteration other ports were also able to showcase this behavior.
3.3 Test scenario is implemented in the port 2 where two equal numbers are subtracted and the desired output is obtained.
3.4 Test scenario where the design simply ignores the data input if they are not valid is being implemented in the port 4.

```
# @1400: Data match---port 1- expected_data1=5b91bac1 monitor_Data1=5b91bac1 ; expected_res1 =  2 monitor_res1 = 2; expected tag =0 and monitor tag = 0;
# @1400: Data match---- expected_data2=00000000 monitor_Data2=00000000 ; expected_res2 =  1 monitor_res2 = 1; expected tag =3 and monitor tag = 3;
# @1400: Data match---- expected_data3=79eef2ea monitor_Data3=79eef2ea ; expected_res3 =  2 monitor_res3 = 2; expected tag =1 and monitor tag = 1;
# @1400: Data match---- expected_data4=00000000 monitor_Data4=00000000 ; expected_res4 =  1 monitor_res4 = 1; expected tag =0 and monitor tag = 0;
# scoreboard method completed
```

Fig 20. Scoreboard results after comparison with monitor data.
From the above data, it's quite evident how the above test scenarios were studied and tested.

4.Generic function test case

4.1 Check that the reset function correctly resets the design.

```
VSIM 133> run
# reset is turned on
run
run
run
run
run
run
# reset is turned off
```

Fig 21. Reset getting turned on during start of the simulation

The Reset gets turned off after Seven cycles as per the instructions given in the description. So, the reset seems to work in the DUT.

5.Advanced Test cases

5.1 Send multiple invalid commands.

```
No of Iterations: 1
[Driver] : req1_data_in1 : c43dccbd    req1_data_in2 : a2a012bc    req1_cmd_in :b    req1_tag_in:0    @950
[Driver] : req2_data_in1 : a3b7dd3f    req2_data_in2 : b5b05c05    req2_cmd_in :c    req2_tag_in:3    @950
[Driver] : req3_data_in1 : 57269fd3    req3_data_in2 : 564ee03c    req3_cmd_in :a    req3_tag_in:1    @950
[Driver] : req4_data_in1 : 85065d9b    req4_data_in2 : 9a672e06    req4_cmd_in :8    req4_tag_in:0    @950
```

Fig 22. Inputs that are being fed to the DUT using the Driver.

The commands here ‘b’, ‘c’,’a’ and ‘8’ are invalid which is being fed to the ports.

```
# @1100: *** ERROR *** expected_data1=xxxxxxxx monitor_Data1=00000000 ; expected_res1 =  x monitor_res1 = 2; expected tag =0 and monitor tag = 0;
# @1100: *** ERROR *** expected_data2=xxxxxxxx monitor_Data2=00000000 ; expected_res2 =  x monitor_res2 = 2; expected tag =3 and monitor tag = 3;
# @1100: *** ERROR *** expected_data3=xxxxxxxx monitor_Data3=00000000 ; expected_res3 =  x monitor_res3 = 2; expected tag =1 and monitor tag = 1;
# @1100: *** ERROR *** expected_data4=xxxxxxxx monitor_Data4=00000000 ; expected_res4 =  x monitor_res4 = 2; expected tag =0 and monitor tag = 0;
# scoreboard method completed
```

Fig 23. Scoreboard results after comparison with monitor data.

So, for the above input which is fed to the DUT we are getting outputs in this fashion as shown in Fig 23. Error message ‘X’ which is undefined is being shown since multiple invalid commands are fed to the DUT.

5.2 Send only a mix of add and subtract commands to fill the add queue.

```
# No of Iterations: 6
# [Driver] : req1_data_in1 : 1bcf2b7b    req1_data_in2 : d3c3984a    req1_cmd_in :1    req1_tag_in:0    @3450
# [Driver] : req2_data_in1 : 31f067be    req2_data_in2 : 56f8ae93    req2_cmd_in :1    req2_tag_in:0    @3450
# [Driver] : req3_data_in1 : b78b908     req3_data_in2 : 744cfc15    req3_cmd_in :1    req3_tag_in:2    @3450
# [Driver] : req4_data_in1 : e3dad4c     req4_data_in2 : 22383cf9    req4_cmd_in :2    req4_tag_in:1    @3450
#
```

Fig 24. Inputs that are being fed to the DUT using the Driver.

The inputs which are being driven are by modifying the constraints by only activating the preference to add and sub commands alone and then driven to the ports.

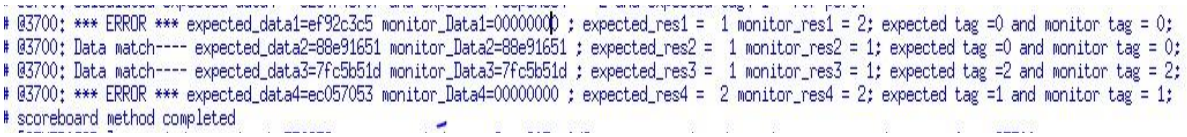


Fig 25. Scoreboard results after comparison with monitor data.

This is the scoreboard outcome which is obtained for the above test scenario. There is an error in the port 1 response that is wrong where there is overflow. Constraints here are given using weighted distribution format.

5.3 Send only a mix of shift commands to fill the shift queue.

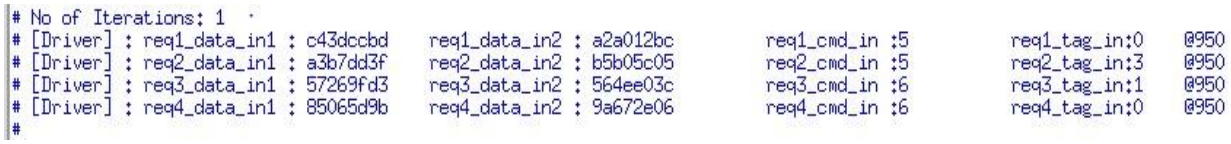


Fig 26. Inputs that are being fed to the DUT using the Driver.

The inputs which are being driven by modifying the constraints by only activating the preference to SLL and SLR commands alone and then driven to the ports.

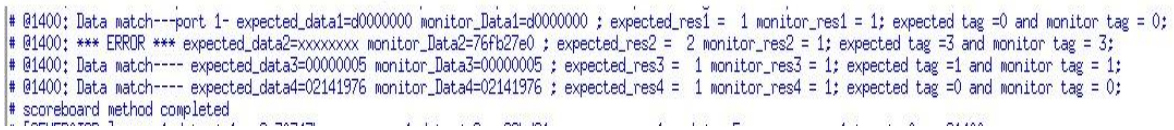


Fig 27. Scoreboard results after comparison with monitor data.

This is the scoreboard outcome which is obtained for the above test scenario. There is an error in the port 2 response that is wrong where there is overflow but it's showing as a successful response . Constraints here are given using weighted distribution format.

5.4 Verify mixes of overflow, underflow and good response cases across back to back port commands.

Fig 28. Inputs that are being fed to the DUT using the Driver.

The driven data to the DUT to PORT1,2,3 and 4 are addition and subtraction equally.
It is driven using the weighted distribution constraints.

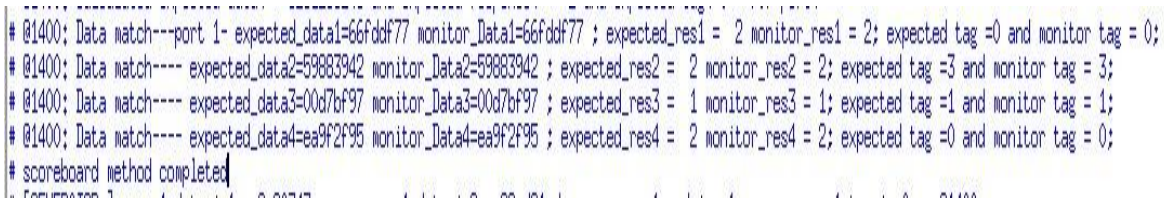


Fig 29. Scoreboard results after comparison with monitor data.

For the above the driven data to the DUT, Port 1 and Port 2 are overflow while Port 3 and Port 4 are underflow and a Good response respectively. Like this in subsequent iterations the DUT was tested for the above test scenario.

5.5 Verify the DUV does not allow output collisions from both pipelines sending results to the same port simultaneously.

Fig 30. Inputs that are being fed to the DUT using the Driver (iteration-6).

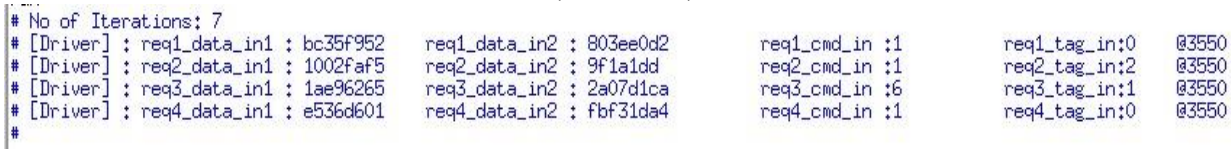


Fig 31. Inputs that are being fed to the DUT using the Driver. (iteration-7).

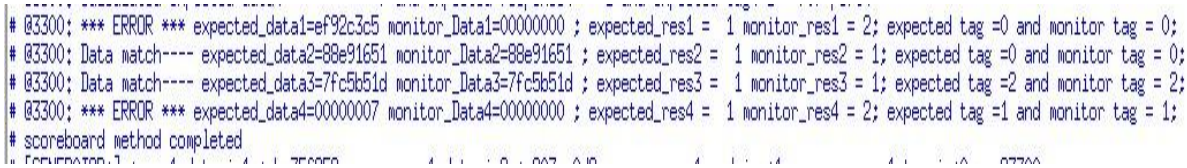


Fig 32. Scoreboard results after comparison with monitor data.(iteration-6).

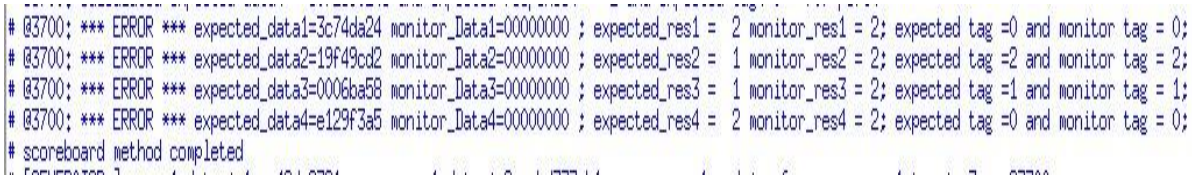


Fig 33. Scoreboard results after comparison with monitor data.(iteration-7).

When we try to send results from both the pipelines simultaneously, we get error messages and DUV fails to deliver the results as expected. The above code is a sample of a couple of iterations which was analyzed for this particular test scenario.

5.6 Check for unmatched tags on the command port.

Fig 34. Inputs that are being fed to the DUT using the Driver.(iteration-4)

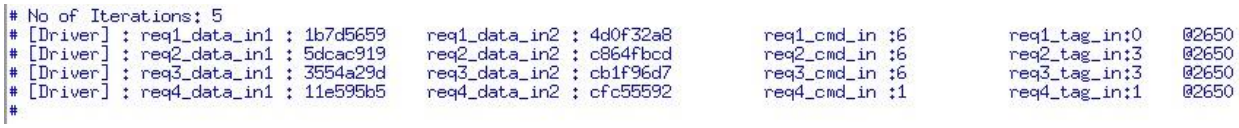


Fig 35. Inputs that are being fed to the DUT using the Driver.(iteration-5)

From Fig 34 and 35, the unmatched tags can be studied and analyzed. Fig 34 is sample iteration 4 and fig 35 is iteration respectively..

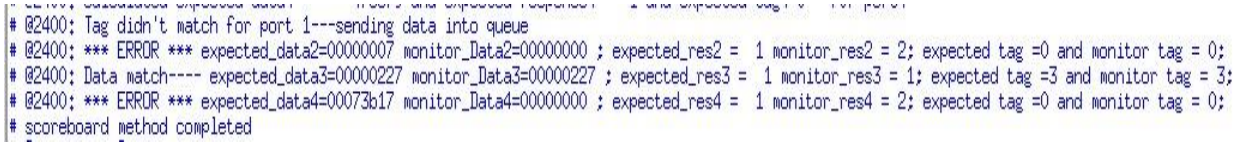


Fig 36. Scoreboard results after comparison with monitor data.(iteration-4)

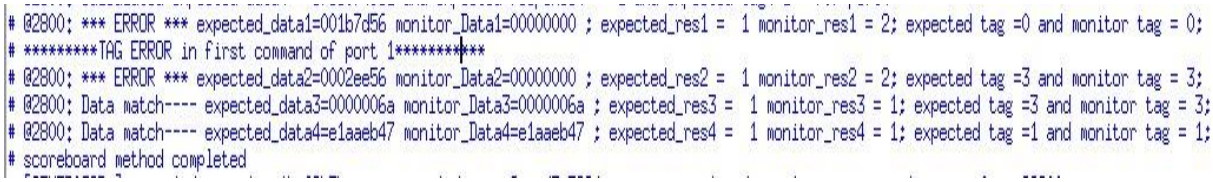


Fig 37. Scoreboard results after comparison with monitor data.(iteration-5)

In iteration 4 there is a tag error at port-1, so the scoreboard sends it to the queue for the tag checking which is done by the checker. In the next iteration itself, tag gets compared and gets credited to be a tag error.

5.7 Check for correctness of out-of-order responses across command pipeline types but never across the same command type.

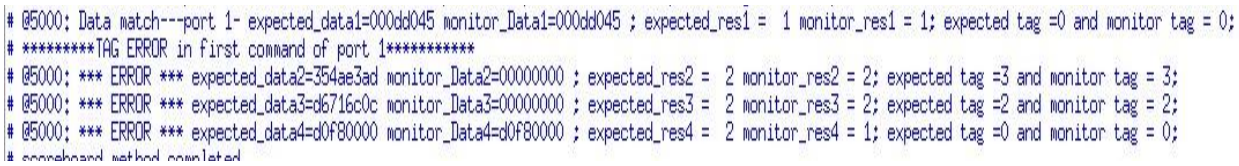


Fig 38. Scoreboard results after comparison with monitor data.

When the Input is driven to the DUV using the driver in out of order execution for around 20-25 iterations. We are experiencing a Tag error in Port1 no matter what the order of the command being fed to the DUV. This is one of the bugs which we found during the verification process.

5.8 Verify in-order execution of all adds/subtracts or shifts, no matter which port sent the command.

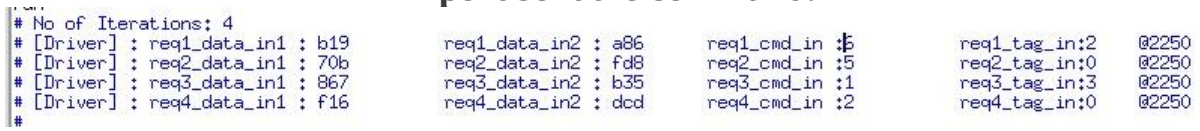


Fig 39. Inputs that are being fed to the DUT using the Driver.

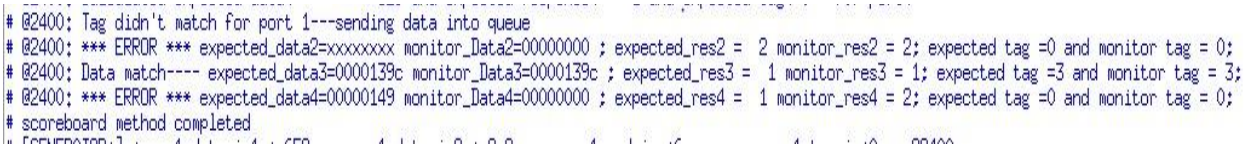


Fig 40. Scoreboard results after comparison with monitor data.

From Fig. 39 and Fig 40, we can see the data being driven to the DUV, in an order. During In order execution there is no problem in execution but the bug present in port-1 in the form of Tag error is persistent.

5.9.1 Check that the response tag matches the data for the command that was sent.
&&
5.9.2 Check that the response tag matches the data for the command that was sent.

The above test scenarios are covered using the previous basic and intermediate test scenarios which were performed earlier.

7. BUG REPORT

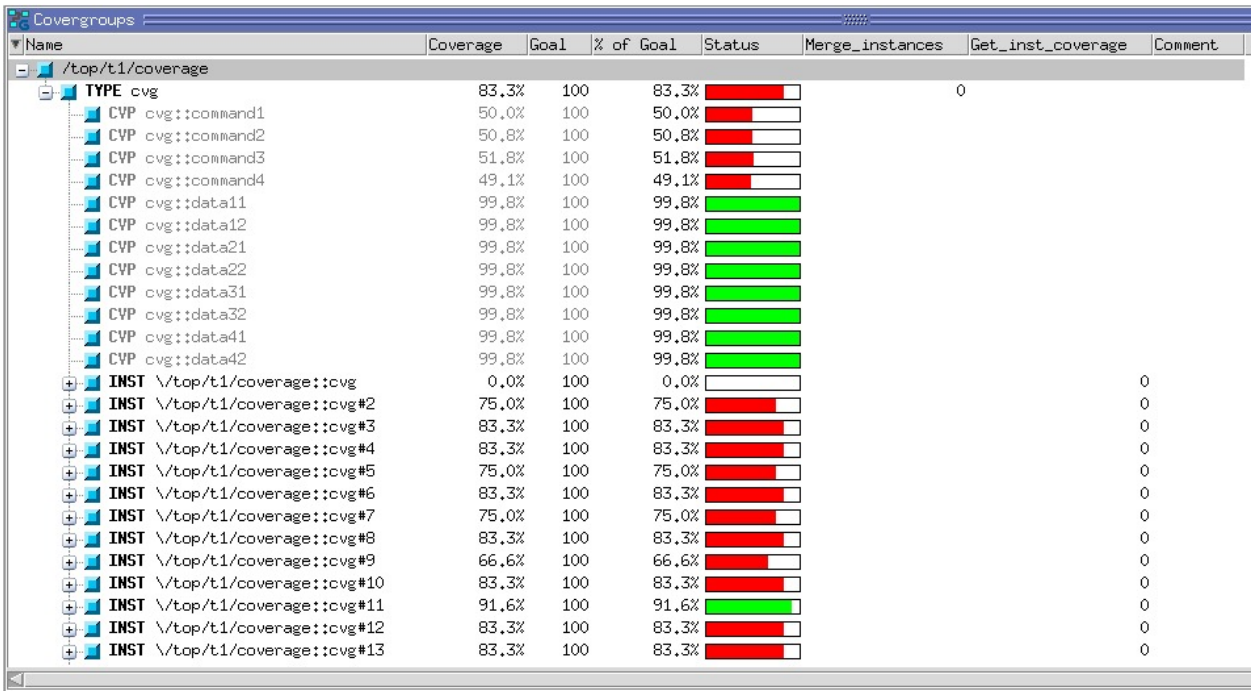
Total Error count for 20 Iterations is 24.

S.No	CASE	BUG
1	Port 1 Tag Error	In out of order as well as in order execution, there is persistent tag Error that occurs.
2	Higher order 27 bits are not Ignored (at times)	The 2nd operand is 20 (Decimal – 32,Binary -100000), which has '00000' as the first 5 bits, meaning the output is supposed to shift 0 places. But, the code is considering 6th bit as well (100000) for decimal value 32. It is working fine until decimal 31.
3	Underflow operation	The underflow operation produces the error in the response. The response is 1 instead of 2.
4	Port 4:- Overflow operation	The overflow operation produces the error in the response . The response is 1, whereas the expected is 2.
5	Port 1: addition	Instead of overflow while performing addition operations it shows successful response.
6	Shift by 0 places for 32 (at times)	The shift by 0 places command is producing an error because DUT is considering some random operands.
7	Overflow Undefined	At times when the value goes out of

		bound during the golden reference data calculation, the output data seems to be undefined.
8	Multiple Illegal Commands	When feeding the multiple illegal commands to the DUV, it gives undefined values as outputs.'X'.

Table 6. Bug Report Table

8. COVERAGE REPORT



The Coverage report that got generated for our code. The coverage achieved is **83.3 %.**
The class coverage has a total 12 bins and 12 features are getting coverage tested.

The trend that can be seen is that the more the iterations are, the more the coverage we get and at a fixed number of iterations we are getting fixed coverage. We are planning to further increase our coverage by including cross coverage which will gives 12 * 12 bins = 144 bin coverage.

9. CONCLUSION

From the verification test results, it can be concluded that the design has a few bugs that are hampering the system from working perfectly under certain scenarios. It was a good thing that a random testing method was used for the verification of the design. From the coverage report, it can be seen that the test bench covers 83% verification of the design. This is a very good percentage & shows a thorough verification was done on the design. Overall, the design works as intended and just needs some minor tweaks for it to function without any errors.