Calc3 Verification Report

Anil Sai Kumar Bandi Venkata
40254601
venkataanilsaikumar3@gmail.com

Aryan Abrari Vajari
ABRA14289901
aryan.abrari-vajari.1@ens.etsmtl.ca

Shresth Niwahaal
40240087
shresthniwahaal@gmail.com

Anukriti Mukherjee
40237113
anukriti98@gmail.com

***Abstract*—This report explores the thorough verification of the Calc3 hardware design, a complex calculator driven by calc1 and calc2. We set out on a painstaking quest to construct a verification environment specifically designed to extensively verify the Calc. design against its specifications using SystemVerilog.**

Contents

I. Introduction 1
II. Calc3 Design Discription 1
A. PinOut 1
B. Functionality 2
III. Design Verification 2
A. Universal Verification Methodology (UVM) 2
B. Test Bench Description 3
C. Test Plan 3
D. Test Cases Explained in Detail with Outputs 4
E. List of bugs 5
F. Functional Coverage Report 5
IV. reference 5

I. Introduction

Calc3 is an RTL design implementation offering some basic mathematical tasks. Since any RTL design is prone to bugs, varification is a vital step in any design. In this project we put calc3 design under test to investigate the possibility of any kind of bugs using SystemVerilog on QuestaSim platform. A series of random testing is conducted on this design, while some test instances succeeded in passing the verification tests, others did not. In this report we will first describe the calc3 design, then we will obtain the test plan, and finally we will dig into the bugs found in this design.

II. Calc3 Design Discription

Calc3 is an RTL design implementation that offers arithmetic operations such as Add, Subtract, Shift Left, Shift Right, Fetch, and Store with two branch commands in which successful branch causes next command from port to be skipped.

Each port requestor sends an instruction stream. In the Calc 1 & 2 designs the data accompanied the command, while in this design, the arithmetic operands reference operand registers internal to the design. Therefore, instruction ordering (instruction stream) concepts must be obeyed by the design so that, within each port, commands may proceed out of order only when the operand registers are not conflict.

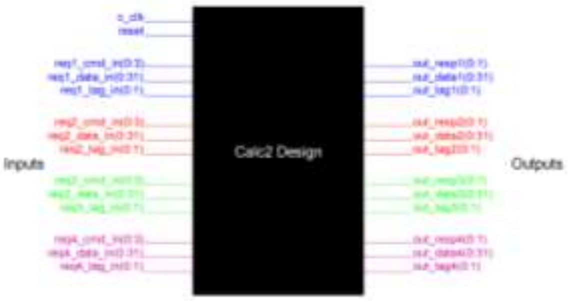


Fig 1 – Calc 2 block design [1]

The design specifications of Calc3 are comparable to those of Calc 2, with the addition of 16 internal registers. The requestor no longer sends the arithmetic operands. The operand data is read from internal registers. In order to access the registers, two new commands have been added: Fetch and Store, as well as two new branch conditions: Branch if equal and Branch if equal to zero, which cause the next command from port to be ignored on a successful branch. The provided command is accompanied by a 2-bit tag to distinguish the corresponding data at the output. The simultaneous use of the same tag is not supported.

A. PinOut

In this section, the pinout of calc3 design has been determined.

c_clk: Gives the clock to drive the input command, data/get the output data, response.
Inputs:
1. **reqX_cmd_in<0:3>:** It is the command bit that defines the operation to be performed
with the data provided. The commands used in this project are:
● **add:** 0001 adds contents of d1 to d2 and stores in r1
● **subtract:** 0010 subtracts contents of d2 from d1 and stores in r1
● **shift left:** 0101 shifts contents of d1 to the left d2(27:31) places and stores in r1
● **shift right:** 0110 shifts contents of d1 to the right d2(27:31) places and stores in r1
● **store:** 1001 stores reqX_data(0:31) into r1
● **fetch:**1010 fetches contents of d1 and outputs it on out_dataX(0:31)
● **branch if zero:** 1100 skip next valid command if contents of d1 are 0

● **branch if equal:** 1101 skip next valid command if contents of d1 and d2 are equal
2. **reqX_d1(0:3)**: It reads the data from internal register d1
3. **reqX_d2(0:3):** It reads the data from internal register d2
4. **reqX_r1(0:3)**: It is the input port to read the data from internal register r1
5. **reqX_tag(0:1)**: The tag bus is the two-bit identifier for each command from the port.
6. **reqX_data(0:31)**: It is the data that will be stored in the registers which is pointed by reqX_r1(0:3)
7. **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.
Outputs:
1. outX_resp(0:1): 2-bit output response bits validates the output;
● **Successful completion**: 01
● **overflow/underflow error:**10
● **Command skipped due to branch:** 11
2. outX_tag(0:1): The output tag bus shows the command tag sent by the requester.
3. outX_data(0:31): Gives corresponding valid data accompanied by response.

B. *Functionality*

As indicated in previous section, calc3 includes two input buses and two output buses for each of the four ports.

The command is transmitted over a 4-bit bus identified as reqX_cmd_in0:3> (where X represents port 1, 2, 3, or 4). The list of commands and their corresponding decode values are shown in the table1.

Command	Decode value
No operation	'0000'b
Add	'0001'b
Subtract	'0010'b
Shift Left	'0101'b
Shift Right	'0110'b
Fetch	'1001'b
Store	'1010'b
Branch if Zero	'1100'b
Branch if Equal	'1101'b
Invalid	All others

Table 1 – calc3 command description

On the output side there are two buses, the response bus out_respX<0:1> & the result data bus out_dataX<0:31> for each port. Table2 shows the responses for possible cases.

Response Decode	Response meaning
'00'b	No response on this cycle
'01'b	Successful response.
'10'b	Overflow, underflow or invalid command. (only valid for the add or subtract commands)
'11'b	Command skipped due to branch

Table 2 – calc3 output response description

III. Design Verification

Since calc3 is a more complex design than calc1 and calc2, a through verification over this design demands utilizing various classes combined with random testing in System Verilog.

Several classes are used in our verificatoin. We have imported these classes using UVM library.

A. *Universal Verification Methodology (UVM)*

The Universal Verification Methodology (UVM) is a well-known methodology for functional verification using SystemVerilog. UVM provides a robust and standardized framework for developing modular, and reusable verification testbench. UVM offer a transaction-based verification library, where stimuli are generated, applied, and monitored using sequences, drivers, and monitors. Therefore, testbenches developed with uvm are more flexible and can be easily adapted to various design scenarios and architectures[1].

The classes we have used from UVM in our testbench are:

1. Uvm_Driver: A class responsible for converting sequence items into bus transactions or pin-level transactions that drive the inputs of the DUT. The driver class is responsible for driving transactions onto the interface. We have used two functions:

• Build Phase Function retrieves the interface configuration and

• Run Phase Function continuously processes transactions, synchronizing each transaction to the positive edge of the clock.

2. Uvm_Env: A class representing a verification environment. It serves as a container for organizing and managing all the components of the testbench necessary for verifying a design. This class sets up a basic environment structure in a UVM-based verification environment, including the initialization of instances of classes (agent, subscriber, scoreboard) and the establishment of connections between them. Additionally, we have added a function in the Environment class to create a file which logs the results which makes it much easier to analyze the results after simulation.

3. Uvm_Monitor: The Monitor class is responsible for monitoring transaction activity on the interface.

• Build Phase Function is used to get the transaction information from the virtual interface

• Run Phase Function is used to wait for transactions to arrive on the interface by using a get method. When a transaction is executed it is written to the analysis port, synchronized to the positive clock edge.

4. Uvm_Scoreboard: This is the most important class to analyze the operations of ALU. The scoreboard compares the value from the expected behavior of the ALU (based on the

Golden Model Class) and the randomized requests generated (based on the Golden Requestor Class). It also has an function to write the comparative results on an analysis export port to show if the test case is a Pass or Fail. Scoreboard class is responsible for comparing received responses with expected responses and reporting any discrepancies.

5. uvm_sequence_item: A class to encapsulate the data and behavior associated with a specific transaction or stimulus. It is a fundamental class used to represent individual transactions or items of communication between the testbench and the design under test (DUT).

6. uvm_sequence: A class representing a sequence of transactions or events that need to be applied to the DUT (Design Under Test). It is a class that represents a sequence of transactions or a sequence of actions that are performed by the verification environment. Sequence Class is of the type transaction. We have specified the constraint of ports in this class. Constraint is specified to have port numbers randomly generated between 0 to 4.

- Task Operation Body is specified in this class which randomizes the sequence of operation on ports and connects commands to ports.

- Task Operation Ports is specified in this class which randomizes the array values of commands on all ports.

7. uvm_subscriber: A class for observing transactions or events generated by the testbench. employed in verification environments to monitor the output of the design under test (DUT) and perform analysis. Subscriber class is used to define the cover groups to monitor the functional coverage in our test bench. The coverpoints are defined to show success, overflow/ underflow and skipped operations.

8. Uvm_agent: It encapsulates several components, including drivers, monitors, sequencers, and potentially other components necessary for verifying a specific interface. Agent Class is used to perform the instantiations of drivers, monitors, sequencers, and potentially other components required for verification. In our testbench we have used the agent to instantiate sequencer, driver, monitor and the analysis port. We have used

- Function Connection to connect the sequencer to get the items from driver and monitor to the analysis port.

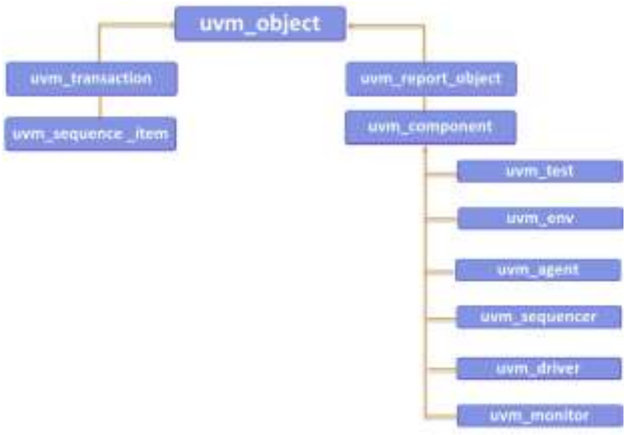


Figure 1: UVM Hierarchical view

B. Test Bench Description

- **Transaction**

Used to define all the inputs, outputs, commands and tags in a user defined data type structure. All the functions used in the testbench are defined in this class.

1. Function to copy the inputs and commands to the ports
2. Function to make all values to zero for a specific port
3. Function to make all output values zero
4. Function to Get values of the inputs, commands and tags from the virtual interface
5. Function to get output values from the virtual interface
6. Function to put the values from the virtual interface to the ports
7. Reset function
8. Function to print the values of each operation

- **Operation Classes**

All operation classes have the function new which is used to define the command associated with that operation and another function to associate to task Operations Body and Operations Ports. We have defined unique classes for each operation : Addition, Subtraction, Shift left, Shift Right, Branching, Store and Fetch

- **Scoreboard Class**

Uses arrays of Transaction requests and responses. Responses are taken from the Golden Model. Function is written to check if out response after operation and data out based on golden model match. If they match the "Passed" message is displayed by Scoreboard. If they don't match then "Failed" message is displayed.

- **Test Classes**

The top test class is the base class which interfaces with uvm testing macro to connect the test cases to the environment. Test 1, 2 and 3 are used to execute all the commands and repeat the sequence x number of times (specified by the verification engineer) at every positive clock cycle edge.

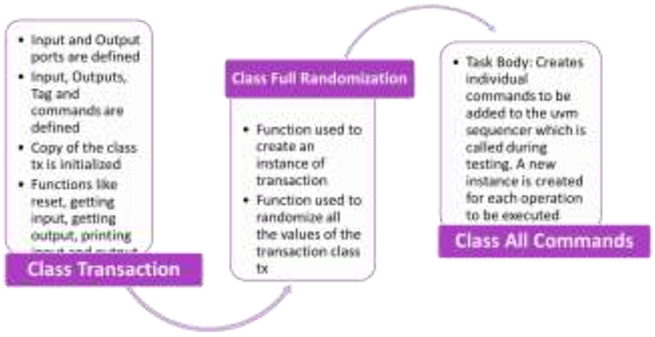

Input and Output ports are defined
Input, Outputs, Tag and commands are defined
Copy of the class tx is initialized
Functions like reset, getting input, getting output, printing
Class Transaction
Class Full Randomization
Function used to create an instance of transaction
Function used to randomize all the values of the transaction class tx
Task Body: Creates individual commands to be added to the uvm sequencer which is called during testing. A new instance is created for each operation to be executed
Class All Commands

Figure 2: Flow of Classes

C. Test Plan

No.	Test Plan

1	Apply basic commands (Add, subtract, Shift left, Shift right, …) on each of the four ports.
2	Check the operation of store and fetch commands
3	Check execution of reset function
4	Apply add and subtract commands to check overflow
5	Apply add on two numbers that overflow by 1
6	Apply subtract commands to check underflow
7	Check that different commands on each port, does not effect the result of other ports
8	Check that the design does not prioritize any of the commands on each port to commands on others by completely randomizing the commands and ports
9	Check that the command is skipped when branch is true
9	Apply shift left and shift right with 0 place
11	Apply shift left and shift right with 31 places
12	Apply illegal commands on each port individually and simultaneously
13	Apply commands on each port with variable timing
14	Apply different tags for each command on each port and check the response for tags that does not match the command
15	Apply different commands with deffirent order in each port

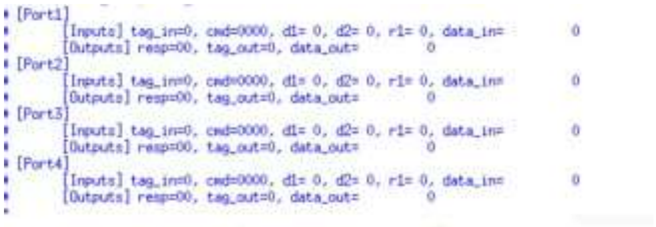
No.	Test Plan for Complex Operations
16	For each port, check that each command can have any command follow it without leaving the state of the design corrupted
17	Check if the branch evaluates true, the following command should be skipped– Add/Sub/SL/SR/Store any of the operation will not write to array

D. Test Cases Explained in Detail with Outputs

In this senction we will make a list of test cases executed on calc3 design. Note that we have used random testing instead of direct testing, Therefore the order of commands and input values are randomly generated in order to better cover the possible behaviour of the design.

Here we dig into a few test cases in detail;

1. Assign command 0000 and expect 00 value in response output.

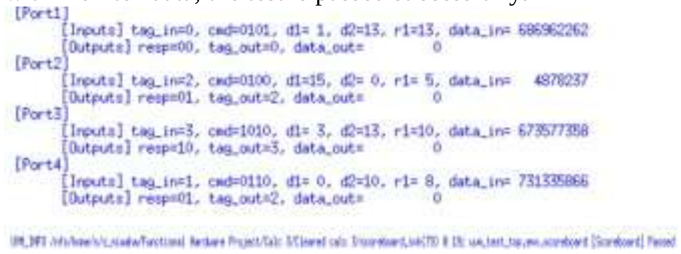


```
# [Port1]
#       [Inputs] tag_in=0, cmd=0000, d1= 0, d2= 0, r1= 0, data_in=          0
#       [Outputs] resp=00, tag_out=0, data_out=          0
# [Port2]
#       [Inputs] tag_in=0, cmd=0000, d1= 0, d2= 0, r1= 0, data_in=          0
#       [Outputs] resp=00, tag_out=0, data_out=          0
# [Port3]
#       [Inputs] tag_in=0, cmd=0000, d1= 0, d2= 0, r1= 0, data_in=          0
#       [Outputs] resp=00, tag_out=0, data_out=          0
# [Port4]
#       [Inputs] tag_in=0, cmd=0000, d1= 0, d2= 0, r1= 0, data_in=          0
#       [Outputs] resp=00, tag_out=0, data_out=          0
#
```

All 0000 commands return the value of 0 and perform no operations when operand values are also 0.

2. Check basic commands on different ports(fetch,add,shift left):

As scoreboard result below indicates, basic commands such as add, fetch, and store passes the test.
The input will be assigned with driver. After comparison with monitor data, the test is passed successfully.

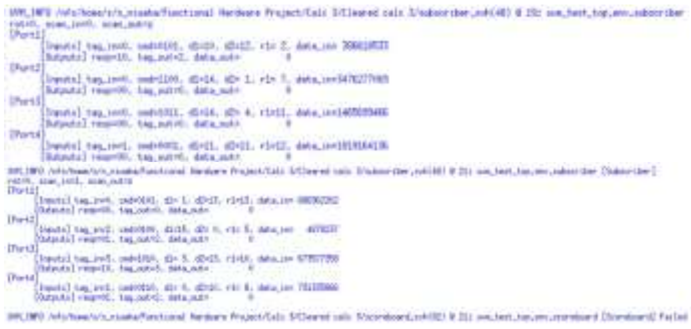


```
[Port1]
      [Inputs] tag_in=0, cmd=0101, d1= 1, d2=13, r1=13, data_in= 686962262
      [Outputs] resp=00, tag_out=0, data_out=          0
[Port2]
      [Inputs] tag_in=2, cmd=0100, d1=15, d2= 0, r1= 5, data_in=   4878237
      [Outputs] resp=01, tag_out=2, data_out=          0
[Port3]
      [Inputs] tag_in=3, cmd=1010, d1= 3, d2=13, r1=10, data_in= 673577358
      [Outputs] resp=10, tag_out=3, data_out=          0
[Port4]
      [Inputs] tag_in=1, cmd=0110, d1= 0, d2=10, r1= 8, data_in= 731335866
      [Outputs] resp=01, tag_out=2, data_out=          0

UVM_INFO /nfs/home/s/s_[illegible]/Functional Hardware Project/Calc 3/Cleared calc 3/scoreboard.sv(70) @ 15: uvm_test_top.env.scoreboard [Scoreboard] Passed
```

We have checked the command for shift left, fetch, shift right and we can see the expected response as follow.

3. Check the Add and subtract in case of overflow and underflow



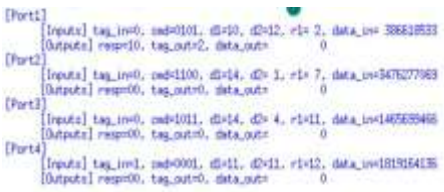
```
UVM_INFO /nfs/home/s/s_[illegible]/Functional Hardware Project/Calc 3/Cleared calc 3/subscriber.sv(48) @ 15: uvm_test_top.env.subscriber
[illegible]
[Port1]
      [Inputs] tag_in=0, cmd=0101, d1=10, d2=12, r1= 2, data_in= 386618533
      [Outputs] resp=10, tag_out=2, data_out=          0
[Port2]
      [Inputs] tag_in=0, cmd=1100, d1=14, d2= 1, r1= 7, data_in=3476277069
      [Outputs] resp=00, tag_out=0, data_out=          0
[Port3]
      [Inputs] tag_in=0, cmd=1011, d1=14, d2= 4, r1=11, data_in=1465699466
      [Outputs] resp=00, tag_out=0, data_out=          0
[Port4]
      [Inputs] tag_in=1, cmd=0001, d1=11, d2=11, r1=12, data_in=1819164136
      [Outputs] resp=00, tag_out=0, data_out=          0
UVM_INFO /nfs/home/s/s_[illegible]/Functional Hardware Project/Calc 3/Cleared calc 3/subscriber.sv(48) @ 21: uvm_test_top.env.subscriber [Subscriber]
[illegible]
[Port1]
      [Inputs] tag_in=0, cmd=0101, d1= 1, d2=13, r1=13, data_in= 686962262
      [Outputs] resp=00, tag_out=0, data_out=          0
[Port2]
      [Inputs] tag_in=2, cmd=0100, d1=15, d2= 0, r1= 5, data_in=   4878237
      [Outputs] resp=01, tag_out=2, data_out=          0
[Port3]
      [Inputs] tag_in=3, cmd=1010, d1= 3, d2=13, r1=10, data_in= 673577358
      [Outputs] resp=10, tag_out=3, data_out=          0
[Port4]
      [Inputs] tag_in=1, cmd=0110, d1= 0, d2=10, r1= 8, data_in= 731335866
      [Outputs] resp=01, tag_out=2, data_out=          0
UVM_INFO /nfs/home/s/s_[illegible]/Functional Hardware Project/Calc 3/Cleared calc 3/scoreboard.sv(82) @ 21: uvm_test_top.env.scoreboard [Scoreboard] Failed
```

4. Apply branch if zero command and expect the next command to be ignored
# [Inputs] tag_in=0, cmd=0101, d1=10, d2=12, r1= 2, data_in= 386618533
# [Port2]
# [Inputs] tag_in=0, cmd=1100, d1=14, d2= 1, r1= 7, data_in=3476277069
# [Port3]
# [Inputs] tag_in=0, cmd=1011, d1=14, d2= 4, r1=11, data_in=1465699466
# [Port4]
# [Inputs] tag_in=1, cmd=0001, d1=11, d2=11, r1=12, data_in=1819164136

Branch command to check if the next value is skipped

5. Check the tag_out port matches the command

Identify applicable funding agency here. If none, delete this text box.

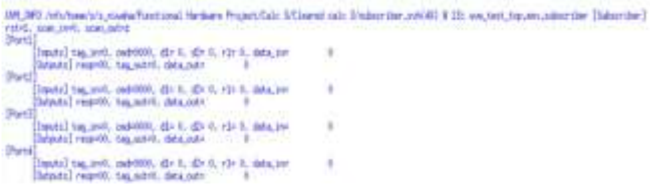


```
[Port1]
      [Inputs] tag_in=0, cmd=0101, d1=10, d2=12, r1= 2, data_in= 386618533
      [Outputs] resp=10, tag_out=2, data_out=          0
[Port2]
      [Inputs] tag_in=0, cmd=1100, d1=14, d2= 1, r1= 7, data_in=3476277069
      [Outputs] resp=00, tag_out=0, data_out=          0
[Port3]
      [Inputs] tag_in=0, cmd=1011, d1=14, d2= 4, r1=11, data_in=1465699466
      [Outputs] resp=00, tag_out=0, data_out=          0
[Port4]
      [Inputs] tag_in=1, cmd=0001, d1=11, d2=11, r1=12, data_in=1819164136
      [Outputs] resp=00, tag_out=0, data_out=          0
```

Check the values of tag_in and tag_out at each port shows that for port 4 the tag_in was 1 but tag_out was 0.

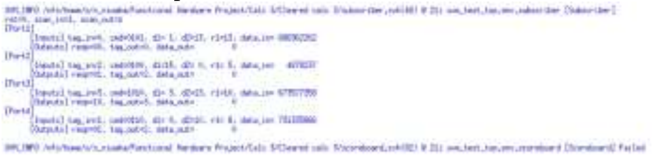
6. Check the tag_out port matches the command

7. Check functionality of reset:



```
UVM_INFO [illegible] Hardware Project/Calc [illegible] @ [illegible]: uvm_test_top.env.subscriber [Subscriber]
[illegible]
[Port1]
      [Inputs] tag_in=0, cmd=0000, d1= 0, d2= 0, r1= 0, data_in=          0
      [Outputs] resp=00, tag_out=0, data_out=          0
[Port2]
      [Inputs] tag_in=0, cmd=0000, d1= 0, d2= 0, r1= 0, data_in=          0
      [Outputs] resp=00, tag_out=0, data_out=          0
[Port3]
      [Inputs] tag_in=0, cmd=0000, d1= 0, d2= 0, r1= 0, data_in=          0
      [Outputs] resp=00, tag_out=0, data_out=          0
[Port4]
      [Inputs] tag_in=0, cmd=0000, d1= 0, d2= 0, r1= 0, data_in=          0
      [Outputs] resp=00, tag_out=0, data_out=          0
```

8. Check the response of invalid command;



```
UVM_INFO [illegible] Hardware Project/Calc [illegible] @ [illegible]: uvm_test_top.env.subscriber [Subscriber]
[illegible]
[Port1]
      [Inputs] tag_in=0, cmd=0101, d1= 1, d2=13, r1=13, data_in= [illegible]
      [Outputs] resp=00, tag_out=0, data_out=          0
[Port2]
      [Inputs] tag_in=2, cmd=0100, d1=15, d2= 0, r1= 5, data_in=   [illegible]
      [Outputs] resp=01, tag_out=2, data_out=          0
[Port3]
      [Inputs] tag_in=3, cmd=1010, d1= 3, d2=13, r1=10, data_in= [illegible]
      [Outputs] resp=10, tag_out=3, data_out=          0
[Port4]
      [Inputs] tag_in=1, cmd=0110, d1= 0, d2=10, r1= 8, data_in= [illegible]
      [Outputs] resp=01, tag_out=2, data_out=          0

UVM_INFO [illegible] Hardware Project/Calc [illegible] @ [illegible]: uvm_test_top.env.scoreboard [Scoreboard] Failed
```

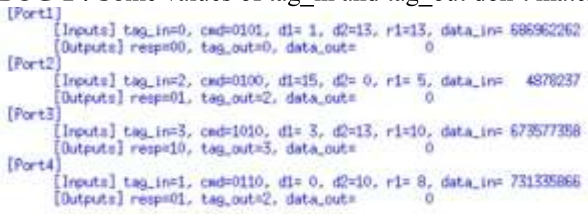
Using illegal command 0100 should give invalid output of 00 but it returns the value of 01 and shows that it is a successful response

E. *List of bugs*

1. For a few test cases the tag commands value entered in input, resulted in wrong "tag_out" value.
2. For an invalid command, the resp output showed successful result.
3. Addition and subtraction operations on port 4 results in incorrect output
4. Some operations like shift return incorrect values

Bugs Explained:

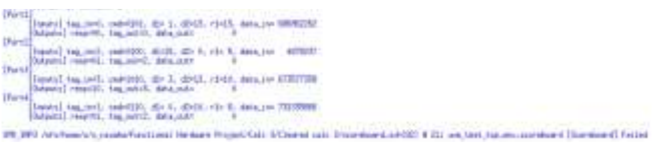
BUG 1 : Some values of tag_in and tag_out don't match



```
[Port1]
      [Inputs] tag_in=0, cmd=0101, d1= 1, d2=13, r1=13, data_in= 686962262
      [Outputs] resp=00, tag_out=0, data_out=          0
[Port2]
      [Inputs] tag_in=2, cmd=0100, d1=15, d2= 0, r1= 5, data_in=   4878237
      [Outputs] resp=01, tag_out=2, data_out=          0
[Port3]
      [Inputs] tag_in=3, cmd=1010, d1= 3, d2=13, r1=10, data_in= 673577358
      [Outputs] resp=10, tag_out=3, data_out=          0
[Port4]
      [Inputs] tag_in=1, cmd=0110, d1= 0, d2=10, r1= 8, data_in= 731335866
      [Outputs] resp=01, tag_out=2, data_out=          0
```

For port 4 we can see that tag_in is 1 but tag_out is 2.

BUG 2 :



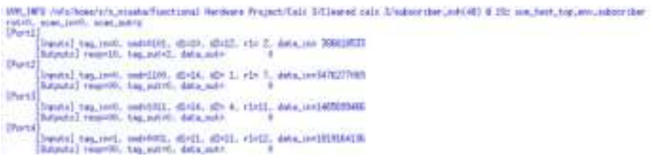
```
[Port1]
      [Inputs] tag_in=0, cmd=0101, d1= 1, d2=13, r1=13, data_in= [illegible]
      [Outputs] resp=00, tag_out=0, data_out=          0
[Port2]
      [Inputs] tag_in=2, cmd=0100, d1=15, d2= 0, r1= 5, data_in=   [illegible]
      [Outputs] resp=01, tag_out=2, data_out=          0
[Port3]
      [Inputs] tag_in=3, cmd=1010, d1= 3, d2=13, r1=10, data_in= [illegible]
      [Outputs] resp=10, tag_out=3, data_out=          0
[Port4]
      [Inputs] tag_in=1, cmd=0110, d1= 0, d2=10, r1= 8, data_in= [illegible]
      [Outputs] resp=01, tag_out=2, data_out=          0
UVM_INFO [illegible] Hardware Project/Calc [illegible] @ [illegible]: uvm_test_top.env.scoreboard [Scoreboard] Failed
```

- Expected result for port 2 should be : resp=00, tag_out=2, data_out= 0

- Observed as: resp = 01, tag_out=2, data_out= 0, which is for a successful response.

Hence we can see that for invalid command the response is successful 01 instead of 00.

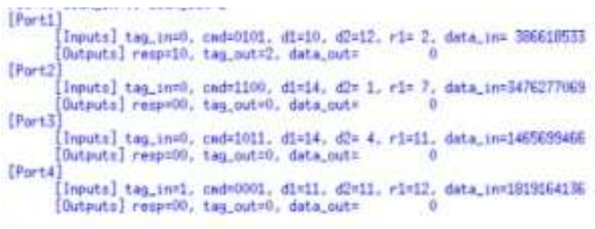
BUG 3:



```
UVM_INFO [illegible] Hardware Project/Calc [illegible] @ [illegible]: uvm_test_top.env.subscriber
[illegible]
[Port1]
      [Inputs] tag_in=0, cmd=0101, d1=10, d2=12, r1= 2, data_in= [illegible]
      [Outputs] resp=10, tag_out=2, data_out=          0
[Port2]
      [Inputs] tag_in=0, cmd=1100, d1=14, d2= 1, r1= 7, data_in=[illegible]
      [Outputs] resp=00, tag_out=0, data_out=          0
[Port3]
      [Inputs] tag_in=0, cmd=1011, d1=14, d2= 4, r1=11, data_in=[illegible]
      [Outputs] resp=00, tag_out=0, data_out=          0
[Port4]
      [Inputs] tag_in=1, cmd=0001, d1=11, d2=11, r1=12, data_in=[illegible]
      [Outputs] resp=00, tag_out=0, data_out=          0
```

On Port 4 the addition values should give the response of overflow and the expected response should be 10 but the observed response is 00. This shows that addition operation on port 4 is not correct.

BUG 4:

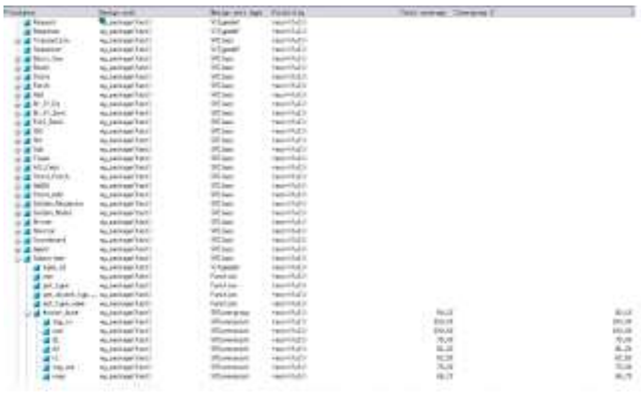


```
[Port1]
      [Inputs] tag_in=0, cmd=0101, d1=10, d2=12, r1= 2, data_in= 386618533
      [Outputs] resp=10, tag_out=2, data_out=          0
[Port2]
      [Inputs] tag_in=0, cmd=1100, d1=14, d2= 1, r1= 7, data_in=3476277069
      [Outputs] resp=00, tag_out=0, data_out=          0
[Port3]
      [Inputs] tag_in=0, cmd=1011, d1=14, d2= 4, r1=11, data_in=1465699466
      [Outputs] resp=00, tag_out=0, data_out=          0
[Port4]
      [Inputs] tag_in=1, cmd=0001, d1=11, d2=11, r1=12, data_in=1819164136
      [Outputs] resp=00, tag_out=0, data_out=          0
```

On port 1 the operation of shift left gives output response of 10 which is the response for overflow/underflow. This is an incorrect response to shift operation.

F. *Functional Coverage Report*

Our test methodology could reach 80% coverage based on the coverage report in Questa.



IV. REFERENCE

[1] https://verificationacademy.com/topics/uvm-universal-verification-methodology/

[2] https://www.doulos.com/knowhow/systemverilog/uvm/uvm-verification-primer/

[3] UVM Rapid Adoption: A Practical Subset of UVM – Sutherland and Fitzpatrick – DVCon, March 2015