Winter 2023

COEN 6541 : FUNCTIONAL HARDWARE VERIFICATION

Dr. Otmane Ait Mohamed

CALC3 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

The objective of this project is to develop a verification test plan in accordance with the Calc3 design specifications. This technical report includes the verification of an RTL design implementation that performs arithmetic operations as in Calc1 and Calc2, as well as additional commands to access registers and verify branching conditions. The RTL design was realized with SystemVerilog and simulated with QuestaSim. The design was validated through a series of tests. Several test instances passed the verification tests, while others failed. This report presents the scenarios for which further development and testing of the calculator design may be necessary.

2. INTRODUCTION

Calc3 is an RTL design implementation that has four basic arithmetic operations,

- Add
- Subtract
- Shift Left
- Shift Right

two new commands to access the internal registers,

- Fetch
- Store

and two branch commands,

- BEQ(Branch Equal)
- BEQZ(Branch Equal to Zero)

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

3. PROJECT DESCRIPTION

A. CALC2 BRIEF DESCRIPTION

The figure below shows the design for Calc2 with a clock signal.

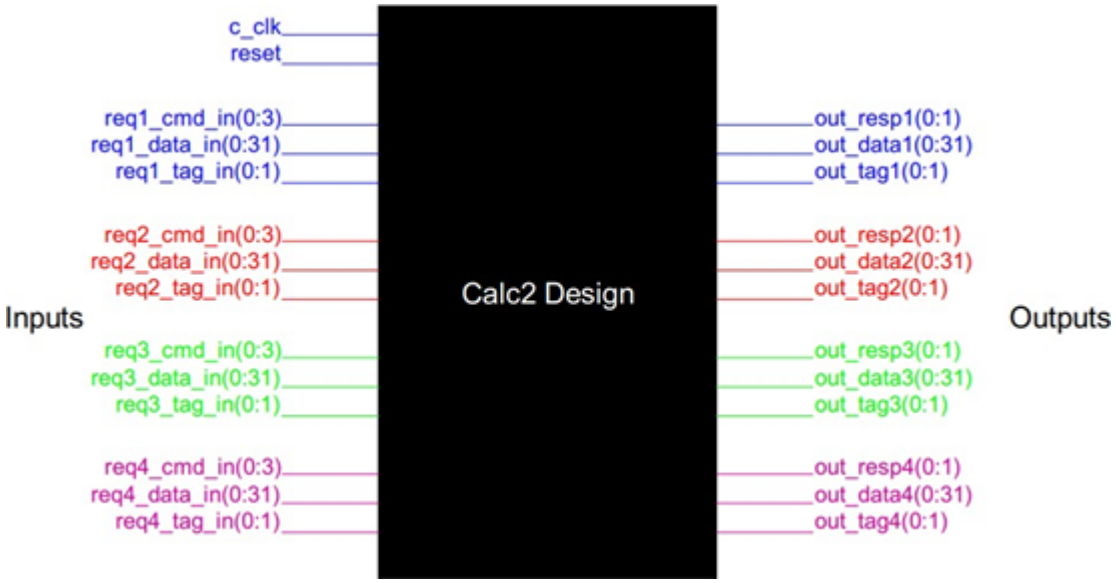


Fig 1. Calc1 design description.

Calc2 is similar to Calc1, with the exception that each port may now have up to four outstanding commands, whereas Calc1 can only manage a single command per port. Thus, 16 commands can be processed simultaneously. Each command from a port is sent individually, but the calculator log may respond "out of order" depending on the queues' internal state. A 2-bit tag is appended to the command in order to monitor the corresponding output data.

B. CALC3 DESIGN DESCRIPTION

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: BEQ(Branch Equal) and BEQZ(Branch 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.

Pin Description:

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 defines the operation performed at the output where

- **Successful completion:** 01
- **overflow/underflow error:**10
- **Command skipped due to branch:** 11

2. **outX_tag(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.
3. **outX_data(0:31)** Gives corresponding valid data accompanied by response.

C. EXPECTED FUNCTIONALITY

There are two input buses and two output buses for each of the four Calc3 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 can be found below.

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. Calculator command description

The operand data is read from the memory address pointed by the registers d1 and d2. The second input bus, reqX_data(0:31) passes the operand data to the calculator. It takes 2 cycles to send a complete command & operand data. The first cycle, the requestor ports pass the operand 1 data & then operand 2 data in the next cycle. The result of all the commands except store is written into register r1 and its data is read by reqX_r1(0:3).

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. The table below shows the responses for possible cases.

Response Decode	Response meaning
‘00’b	No response on this cycle.
‘01’b	Successful response. Response data is on the output data bus.
‘10’b	Overflow, underflow or invalid command. Overflow/underflow only valid for the add or subtract commands.
‘11’b	Command skipped due to branch

Table 2. Calculator output response code

The following timing diagrams illustrate when different input command signals are transmitted through input ports, followed by a 2 bit-tag.

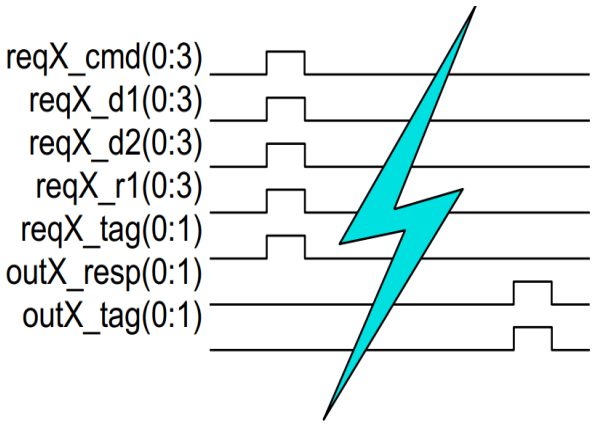


Fig 2. Arithmetic command I/O timing diagram

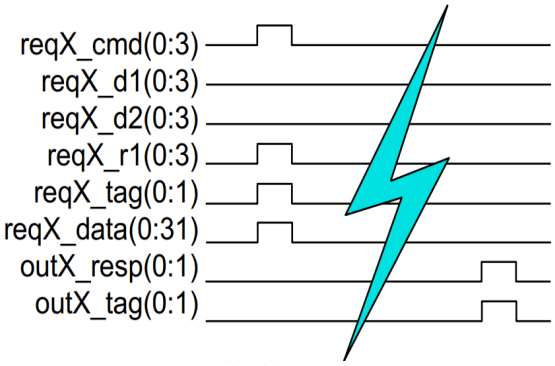
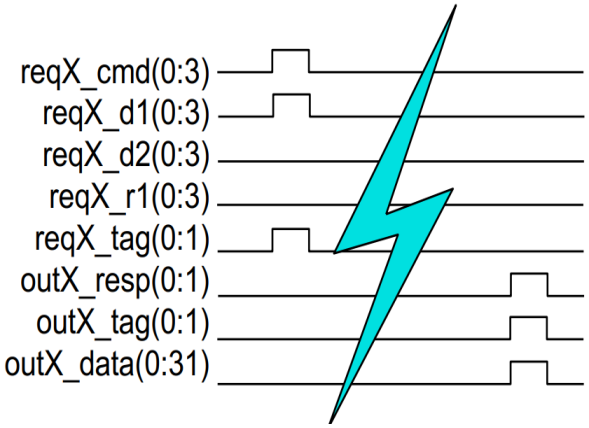


Fig 3. Store command I/O timing diagram



1.

Fig 4. Fetch command I/O timing diagram

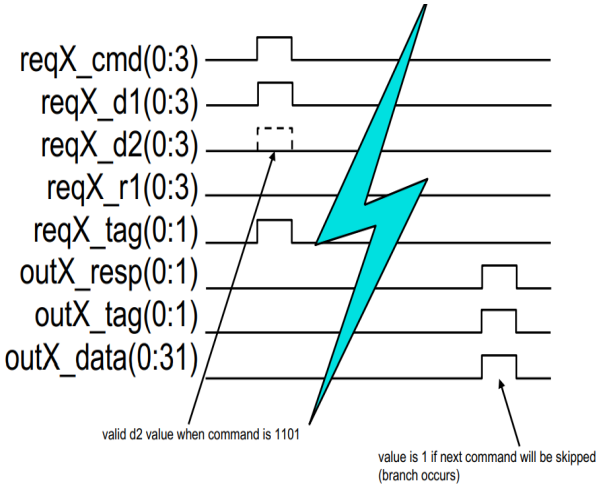


Fig 5. Branch command I/O timing diagram

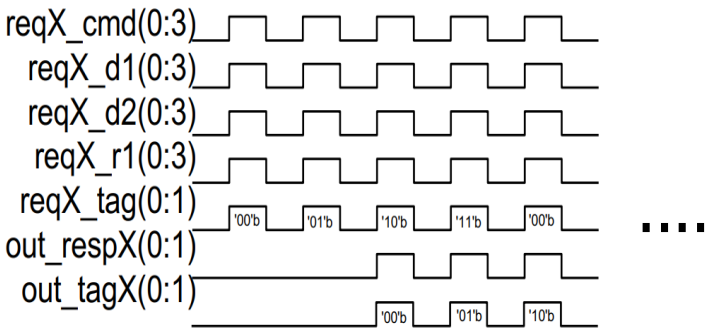


Fig 6. Multiple command I/O timing diagram

D. VERIFICATION TESTING REQUIREMENTS

The verification test bench design for Calc3 is complex. The various functions and ports are tested using Random testing method to ensure they are all operational. The below image depicts the test bench verification method that was used.

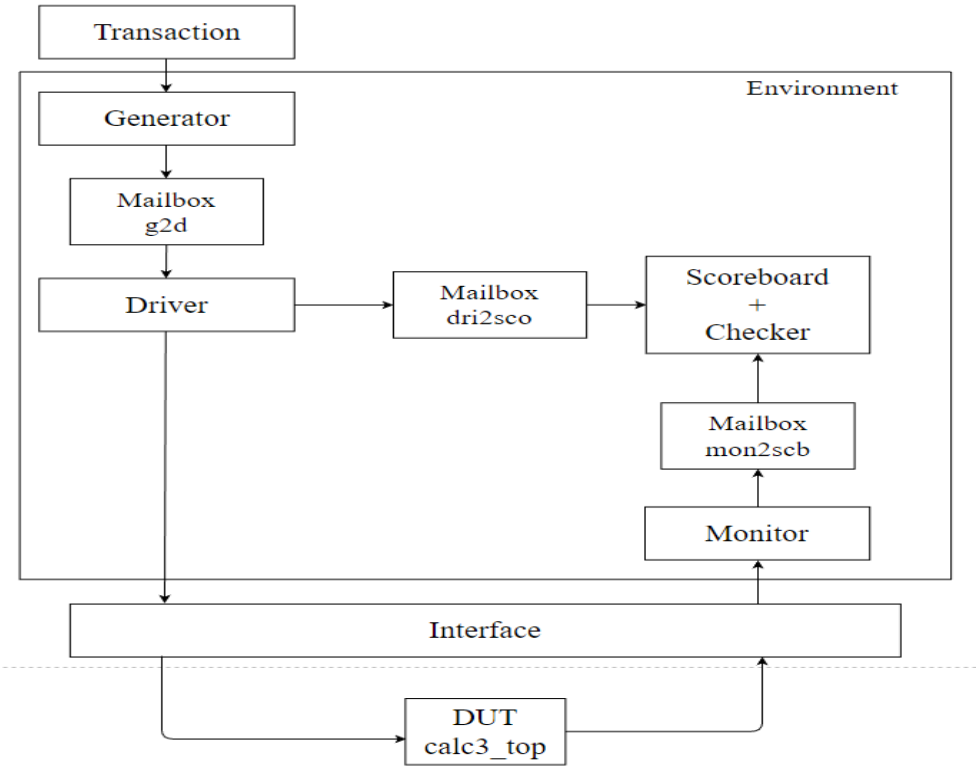


Fig 7. Test Bench used

Transaction

The transaction class is necessary to produce the random stimulus. The stimulus can be controlled to achieve the desired values/range. The command and tag bits are also generated for all ports by this class. This class also declares the expected tag bits, captured tag bits, output data bits, expected data bits, output response bits, and expected response bits. This class also handles the copying of current random values for constraints that prohibit the repetition of values.

Generator

The generator class that we developed randomizes the transaction class that is programmed to generate multiple input stimuli that will be transmitted to the DUT via the driver class. Declaring a variable controls the number of iterations performed by the loop, which is added by this class. This class also displays whether or not the randomization was successful. This class generates the Mailbox, g2d for sending the random stimulus from the generator class to the driver. In addition, an event is declared to indicate the conclusion of transactions.

Interface

For ease of handling and signal reuse, many signals are placed in a single band within a device called the interface. All input-output signals are defined in the interface class. At the posedge of the clock in this class, clocking blocks are also stated for the test class and the monitor. Additionally, the modport declares the signal directions for the monitor, DUT, and test.

Driver

The driver class creates a mailbox dri2scb in addition to the mailbox g2d. 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 new virtual interface intf2 and mailbox mon2scb are created by the monitor class. The interface and mailbox values are sent to the scoreboard using a custom constructor. For all 4 ports from the DUT, output data, output response, and output tag bits are recorded in this class. The tasks to collect data, response, and tag are all being done in parallel.

Scoreboard Checker

This block works both as a reference as well as a checker. The expected results are calculated using the operands that are fetched from the internal registers transmitted through the dri2sco mailbox using get() function. These operands are stored locally in the scoreboard in order to perform this operation. Once a command has been successfully executed, the DUT's output, sent to the scoreboard through mon2scb mailbox and the calculated expected results are compared.

If the results do not match, an error message indicating data mismatch is printed. In order to avoid write after write error and as write commands on r1 follow order execution we are using flags to flag

the registers when they are in writing operation, which can be showed by following code snippet.

```
16'b1001 : begin
            while (exp_tagr[driv_tr.req2_r1] != 0) begin
              #1ns;
           end
           exp_R[driv_tr.req2_r1] = driv_tr.req2_data;
            exp_res2 = 2'b10;
            exp_tagr[driv_tr.req2_r1]=1;
         end
```

code snippet 1; avoiding write after write error

Environment

In this class, instances of every class are produced, and the components are connected. This performs and displays iterations of the generator, scoreboard, and driver. The fork-join method is used in this class to run each component in parallel.

4. TOOLS USED

The software QuestaSim is used for the simulation & verification of Calc3.

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 the basic operation of each command on each port.
1.3	Check overflow and underflow cases for add and subtract commands.

Table 3. Basic functionality 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.
2.3	Check that there is fairness across all four ports such that no port has higher priority than others

2.4	Check that high order 27 bits are ignored in the second operand of shift commands
2.5	Check for correct execution of store and fetch commands
2.6	Check if the branch evaluates true, the following command should be “skipped”: – Add/Sub/SL/SR/Store will not write to array
2.7	Check if the branch evaluates true, the following command should be “skipped” – Store will not write to array
2.8	Check if the branch evaluates true, the following command should be “skipped”: – Fetch will not return data
2.9	Check if the branch evaluates true, the following command should be “skipped”– Branch will evaluate to false (case of branch followed by branch)
2.10	Check if the branch evaluates true, the following command should be “skipped”– Output response tag value should be “11”b

Table 4. 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. Expected output response: 10’b (OVERFLOW)
3.2	Add two numbers whose sum is “FFFFFFFF”X.Expected output response:01’b Expected output result: FFFFFFFF
3.3	Subtract two equal numbers. Expected output response: 01’b Expected output result: 0
3.4	Subtract a number that underflows by 1. Expected output response: 10’b (UNDERFLOW)
3.5	Shift left 0 places. Expected output response: 01’b Expected output result: Operand 1
3.6	Shift right 0 places. Expected output response: 01’b Expected output result: Operand 1
3.7	Shift left 31 places (max allowable shift places). Expected output response: 01’b
3.8	Shift right 31 places (max allowable shift places). Expected output response: 01’b

Table 5. Data dependent case test plan

4. TEST PLAN FOR GENERIC FUNCTION TESTS AND CHECKS:

S.No.	Test Case Description
4.1	Check that the design correctly handles illegal commands
4.2	Check all outputs all the time. Calc1 should not generate superfluous output values.
4.3	Check that the reset function correctly resets the design

Table 6. Generic function test plan

5. OTHER FUNCTIONS TESTS:

S.No.	Test Case Description
5.1	Send Multiple commands with variable timing between commands from the same port.
5.2	Send commands using variable tags for each command.
5.3	Send multiple invalid commands.
5.4	Check that the response value matches expected response based on command and data.
5.5	Check that every command gets a response.
5.6	Check for unmatched tags on the command port.
5.7	Check that the result data matches the expected result based on command and data.
5.8	Check for correctness of out-of-order responses across command pipeline types but never across the same command type.
5.9	Verify in-order execution of all adds/subtracts or shifts, no matter which port sent the command.
5.10	Check that the response tag matches the data for the command that was sent
5.11	Check that there are no unexpected or stray values on the output.

Table 7. Other functions test plan

6. VERIFICATION RESULTS

The verification test results are given in the attached Excel sheet with the report. Our code is only able to generate upto 7 iterations and is getting crashed after that. THese are the results upto 7 iterations.

Test Scenarios :-

1.Basic function test cases

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

```
#                 700: Starting scoreboard for 0 transaction
# No of Iterations: 1
# [Driver] :req1_cmd : 5          req1_tag:2     req1_d1 : 10    req1_d2 : 14    req1_data: 6f5657bc     req1_r1 : 11
# [Driver] :req2_cmd : 10         req2_tag:1     req2_d1 : 6     req2_d2 : 0     req2_data: 69cfbd68     req2_r1 : 11
# [Driver] :req3_cmd : 1          req3_tag:1     req3_d1 : 10    req3_d2 : 2     req3_data: 5dfa0fe5     req3_r1 : 8
# [Driver] :req4_cmd : 5          req4_tag:1     req4_d1 : 1     req4_d2 : 2     req4_data: 27979d91     req4_r1 : 4
# @ 1200 [MONITOR] write to port1
#
```

Fig 8. 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.

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

Fig 9. 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 after reset and scoreboard is verifying it in return.

```
# No of Iterations: 2
# [Driver] :req1_cmd : 12         req1_tag:2     req1_d1 : 0     req1_d2 : 6     req1_data: ae3396f5     req1_r1 : 8
# [Driver] :req2_cmd : 13         req2_tag:1     req2_d1 : 9     req2_d2 : 9     req2_data: 03f42fc6     req2_r1 : 6
# [Driver] :req3_cmd : 9          req3_tag:0     req3_d1 : 1     req3_d2 : 15    req3_data: f19bb22b     req3_r1 : 2
# [Driver] :req4_cmd : 1          req4_tag:3     req4_d1 : 11    req4_d2 : 2     req4_data: 2690d133     req4_r1 : 12
# @ 1600 [MONITOR] write to port2
#
```

Fig 10. Driver driving the inputs to the DUT.

```
# @1900: Tag didn't match---
# @1900: *** ERROR *** expected_data2=00000000 monitor_Data2=00000000 ; expected_res2 = 1 monitor_res2 = 1; expected tag =1 and monitor tag = 1;
# @1900: *** ERROR *** expected_data3=00000000 monitor_Data3=00000000 ; expected_res3 = 2 monitor_res3 = 0; expected tag =0 and monitor tag = 0;
# @1900: Tag didn't match---
# scoreboard method completed
```

Fig 11. Scoreboard results after comparison with monitor data.

The above results show that port 1,2,3,4 performing different operation and giving appropriate response according to the scoreboard.

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

This can be inferred from the above results

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.

```
#                  700: Starting scoreboard for 0 transaction
# No of Iterations: 1
# [Driver] :req1_cmd : 5          req1_tag:2     req1_d1 : 10    req1_d2 : 14    req1_data: 6f5657bc     req1_r1 : 11
# [Driver] :req2_cmd : 10         req2_tag:1     req2_d1 : 6     req2_d2 : 0     req2_data: 69cfbd68     req2_r1 : 11
# [Driver] :req3_cmd : 1          req3_tag:1     req3_d1 : 10    req3_d2 : 2     req3_data: 5dfa0fe5     req3_r1 : 8
# [Driver] :req4_cmd : 5          req4_tag:1     req4_d1 : 1     req4_d2 : 2     req4_data: 27979d91     req4_r1 : 4
# @ 1200 [MONITOR] write to port1
#
```

Fig 12. Driver driving the inputs to the DUT.

```
# No of Iterations: 2
# [Driver] :req1_cmd : 12         req1_tag:2     req1_d1 : 0     req1_d2 : 6     req1_data: ae3396f5     req1_r1 : 8
# [Driver] :req2_cmd : 13         req2_tag:1     req2_d1 : 9     req2_d2 : 9     req2_data: 03f42fc6     req2_r1 : 6
# [Driver] :req3_cmd : 9          req3_tag:0     req3_d1 : 1     req3_d2 : 15    req3_data: f19bb22b     req3_r1 : 2
# [Driver] :req4_cmd : 1          req4_tag:3     req4_d1 : 11    req4_d2 : 2     req4_data: 2690d133     req4_r1 : 12
# @ 1600 [MONITOR] write to port2
#
```

Fig 13. Driver driving the inputs to the DUT.

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 14. Reset getting turned on during start of the simulation

5.5 Check for unmatched tags on the command port.

```
# No of Iterations: 2
# [Driver] :req1_cmd : 12         req1_tag:2     req1_d1 : 0     req1_d2 : 6     req1_data: ae3396f5     req1_r1 : 8
# [Driver] :req2_cmd : 13         req2_tag:1     req2_d1 : 9     req2_d2 : 9     req2_data: 03f42fc6     req2_r1 : 6
# [Driver] :req3_cmd : 9          req3_tag:0     req3_d1 : 1     req3_d2 : 15    req3_data: f19bb22b     req3_r1 : 2
# [Driver] :req4_cmd : 1          req4_tag:3     req4_d1 : 11    req4_d2 : 2     req4_data: 2690d133     req4_r1 : 12
# @ 1600 [MONITOR] write to port2
#
```

Fig 15. Driver driving the inputs to the DUT.

```
# @1900: Tag didn't match---
# @1900: *** ERROR *** expected_data2=00000000 monitor_Data2=00000000 ; expected_res2 = 1 monitor_res2 = 1; expected tag =1 and monitor tag = 1;
# @1900: *** ERROR *** expected_data3=00000000 monitor_Data3=00000000 ; expected_res3 = 2 monitor_res3 = 0; expected tag =0 and monitor tag = 0;
# @1900: Tag didn't match---
# scoreboard method completed
```

Fig 16. Scoreboard results after comparison with monitor data.

7. BUG REPORT

No	CASE	BUG
1	Invalid Command	For an invalid command, the response shown is successful instead of invalid.
2	Operations in Port 4	The port does not support ADD & SUB operations.
3	ADD/sub/shift l/sr operators error	unmatched data and responses
4	commands after branch error	Gives the successful response ‘01’b instead of the expected response ‘11’b.
5	branch after branch	Gives the successful response ‘11’b instead of the expected response ‘10’b.
6	tag error	Most time tag values are getting mismatched

Table 8. Bug Report Table

8. COVERAGE REPORT

Covergroups

Name	Coverage	Goal	% of Goal	Status
/top/t1/coverage				
TYPE cvg	73.6%	100	73.6%	
CVP cvg::command1	44.4%	100	44.4%	
CVP cvg::command2	44.4%	100	44.4%	
CVP cvg::command3	55.5%	100	55.5%	
CVP cvg::command4	66.6%	100	66.6%	
CVP cvg::data11	77.7%	100	77.7%	
CVP cvg::data12	77.7%	100	77.7%	
CVP cvg::data21	77.7%	100	77.7%	
CVP cvg::data22	77.7%	100	77.7%	
CVP cvg::data31	77.7%	100	77.7%	
CVP cvg::data32	77.7%	100	77.7%	
CVP cvg::data41	77.7%	100	77.7%	
CVP cvg::data42	77.7%	100	77.7%	
CVP cvg::data51	77.7%	100	77.7%	
CVP cvg::data52	77.7%	100	77.7%	
CVP cvg::data61	77.7%	100	77.7%	
CVP cvg::data62	77.7%	100	77.7%	
CVP cvg::data71	77.7%	100	77.7%	
CVP cvg::data72	77.7%	100	77.7%	
CVP cvg::data81	77.7%	100	77.7%	
CVP cvg::data82	77.7%	100	77.7%	
CVP cvg::data91	77.7%	100	77.7%	
CVP cvg::data92	77.7%	100	77.7%	
CVP cvg::data93	77.7%	100	77.7%	
CVP cvg::data94	77.7%	100	77.7%	
INST \/top/t1/coverage::cvg	0.0%	100	0.0%	
INST \/top/t1/coverage::cvg#2	0.0%	100	0.0%	
INST \/top/t1/coverage::cvg#3	95.8%	100	95.8%	
INST \/top/t1/coverage::cvg#4	95.8%	100	95.8%	
INST \/top/t1/coverage::cvg#5	91.6%	100	91.6%	
INST \/top/t1/coverage::cvg#6	91.6%	100	91.6%	
INST \/top/t1/coverage::cvg#7	95.8%	100	95.8%	
INST \/top/t1/coverage::cvg#8	95.8%	100	95.8%	
INST \/top/t1/coverage::cvg#9	95.8%	100	95.8%	

scoreboard.sv | Covergroups

Fig 16. Cover report

9. CONCLUSION

From the verification test results, it can be concluded that the design has quite a few bugs that are hampering the design from working perfectly under certain scenarios. It was a good thing that a random testing method was used for the verification of the design. Random testing is preferred compared to direct testing in larger designs. Here due to error in code we were not able to produce accurate verification results but this method is feasible.