|  |  |
| --- | --- |
| Error Correction Encoder & Decoder | **Digital Design and Logical Synthesis for Electric Computer Engineering** |
| **(36113611)**  **Course Project**  **Verification**  **Version 0.1** |

Version 0.1 4 June 2007

**Revision Log**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Rev** | **Change** | **Description** | **Reason for change** | **Done By** | **Date** |
| 0.1 | Initial document |  |  | Shy Hamami | 4,Jun,2007 |
| 0.2 | Digital Changes |  |  | Amir Kolaman | 14,Jul,2007 |
| 0.3 | Functional Verification |  |  | Amir Kolaman | 4,November 2007 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Table of Content**

Contents

[1. LISt of figures 4](#_Toc91160941)

[2. List of tables 5](#_Toc91160942)

[3. Verification Plan 6](#_Toc91160943)

[3.1 Verification Test Objectives 9](#_Toc91160944)

[3.2 Test Bench High Level Diagram and Architecture 9](#_Toc91160945)

[3.3 Test Bench Low Level Architecture and Functionality 13](#_Toc91160946)

[3.4 Stimulus 14](#_Toc91160947)

[3.5 Functional Coverage 15](#_Toc91160948)

[3.6 Test Bench Functional Checkers 19](#_Toc91160949)

[3.7 Golden Model 23](#_Toc91160950)

[3.8 Code Coverage 24](#_Toc91160951)

[**3.8.1 ECC\_ENC\_DEC 24**](#_Toc91160952)

[**3.8.2 Encoder Code Coverage 26**](#_Toc91160953)

[**3.8.1 Error\_Fix Code Coverage 27**](#_Toc91160954)

[**3.8.2 Num\_Of\_Errors Code Coverage 27**](#_Toc91160955)

[**3.8.3 Register\_Selector Code Coverage 27**](#_Toc91160956)

[3.9 Design Checker Approved Errors 28](#_Toc91160957)

[4. Appendix 28](#_Toc91160958)

[4.1 Terminology 28](#_Toc91160959)

[4.2 References 28](#_Toc91160960)

# LISt of figures

[Figure 1: ECC\_ENC\_DEC Interface – project goal. 6](#_Toc91160961)

[Figure 2: ECC\_ENC\_DEC Interface – High Level. 10](#_Toc91160962)

[Figure 3: Interface bus. 11](#_Toc91160963)

[Figure 4: ECC\_ENC\_DEC Interface – Low Level. 13](#_Toc91160964)

[Figure 5: Stimulus Interface. 14](#_Toc91160965)

[Figure 6: Coverage Interface. 15](#_Toc91160966)

[Figure 7: Signal Coverage. 16](#_Toc91160967)

[Figure 8: Num of errors coverage. 16](#_Toc91160968)

[Figure 9: Functional Checker Interface. 19](#_Toc91160969)

[Figure 10: Functional Checker - Assertions. 22](#_Toc91160970)

[Figure 11: GoldenModel Interface. 23](#_Toc91160971)

[Figure 12: DUT code Coverage. 24](#_Toc91160972)

[Figure 13: DUT coverage details. 24](#_Toc91160973)

[Figure 14: DUT coverage details. 25](#_Toc91160974)

[Figure 15: DUT coverage details. 25](#_Toc91160975)

[Figure 16: DUT coverage details. 25](#_Toc91160976)

[Figure 17: DUT coverage details. 26](#_Toc91160977)

[Figure 18: Error\_fix coverage details. 27](#_Toc91160978)

[Figure 19: Num\_of\_errors coverage details. 27](#_Toc91160979)

[Figure 20: RegSelector coverage details. 27](#_Toc91160980)

[Figure 21: DUT Design Ignored errors. 28](#_Toc91160981)

[Figure 22: DUT Design Ignored warnings. 28](#_Toc91160982)

# List of tables

[Table 1: Test name/number 8](#_Toc91160983)

[Table 2: Test Plan Functional Coverage 18](#_Toc91160984)

[Table 3: Test Plan FunctionalCheckers 21](#_Toc91160985)

# Verification Plan

Diagram

Description automatically generated

1. ECC\_ENC\_DEC Interface – project goal.

The top module for verification system design is “tb\_overall”, our strategy for defining out test bench start of understanding with what we can check and what we can compare with the use of the seven systems that we learned in Lab 2 to build the verification system.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test name/number | Functionality | Test data set | Expected result | Observations |
| Signals\_test | Checking if the amount is right | Rst, PENABLE , PSEL , PWRITE , Opration\_done | We expecting the right amount of high and low signalks | From the observartion the system giving the right amount og high and low |
| Amount of noise | Checking the amount of noise we had in the test including the Gm | Dut num\_of\_errros,  GM num\_of\_errors | Expecting to get the same amount in both of them and getting 2/3 of errors in total compare to operation\_done | From observartion we got what it need to give, also we getting amont almost the same amount for no\_errors and one and two errors |
| Error\_spot | make sure that the Noise got all the options of TWO ERRORS and ONE ERROR | NOISE | Expecting to see at One\_error\_spot and Two\_error\_spot full coverge | Because we used random noise generator, we observed 100% only after running sufficient amount of samples to the system |
| Rst\_active | Making sure the output of the system is zeros | Operation\_done, data\_out, num\_of\_errors | See that we didn’t got alert | Because we can’t read the rgister in the DUT we need to assumption that they reset as well |
| Registers Read Check | Checking that the data got read is the wanted data | PRDATA , GM PRDATA | Expecting to see no error what tell us they are equal and right | If we get rst at this check we don’t check it |
| Operation done active | Checking that the signal opretion\_done is “1” after 2-5 clk cycle | Operation\_done | With out any problem we need to see that after 2-5 cycle from when we write to the CTRL reg | We observed that operation done signal was active during the right period 100% of the times |
| Result Check | Making sure the output of the DUT and gm are equal | data\_out  gm\_data\_out | Of course, we expect to see equality at all samples | We saw that all the samples got the right output whenever operation done is active |
| Num Of Errors Check | Making sure the output of the DUT and gm are equal | Num\_of\_errors  gm Num\_of\_errors | Of course, we expect to see equality at all samples | We saw that all the samples got the right output whenever operation done is active |
| Num Of Errors Boundary Check | Making sure num\_of\_errors is never 3 when operation\_done is 1 | Num\_of\_errors | We expect to see Num\_of\_errors only equals to 0,1 or 2. | We saw that all the samples got the right output whenever operation done is active, meaning only 0,1 and 2 |

1. Test name/number

## Verification Test Objectives

The objective in this test, is making sure all the functionality of the DUT works as demanded, by the APB bus protocol, Testing extreme scenarios functionality, and illegal scenarios. The way we choose to do that, is by dividing the test to Pre test Matlab randomization process, and live feed of the DUT with the generated input with System verilog’s randomly generated noise vectors.

## Test Bench High Level Diagram and Architecture

We used the architecture we learned in the course, which divides the work in the following order:

1. Interface – Contain all the buses between all the new modules and the DUT.
2. Stimulus – In charge of creating random noise and inserting all the needed data form the file into the GoldenModule and the DUT using the AMBA protocol.
3. Coverage - Determines how much functionality of the design has been exercised.   
   Functional assertions are used to check whether each and every corner of the design is explored and functions properly.
4. Checker – Checking that we getting values that are within an acceptable range according to the proper design order , also we checking the input and output properties and compare them with the GoldenModule.
5. Golden Module – We use here the Matlab and the Verilog to send the right output to the checker.
6. Overall\_tb – This is the main component, creating all other components necessary in this verification process. This module is responsible for the clk and rst signals, this way we can truly test the functionality of the rst and clk signals.

Diagram

Description automatically generated

1. ECC\_ENC\_DEC Interface – High Level.

A picture containing application

Description automatically generated

For the verification plan we started by first planning how the structure of our test needs to be, and what are the common signals in our interface.  
The final signals for our Interface are:

Graphical user interface, text

Description automatically generated with medium confidence

1. Interface bus.

The second step is creating the Stimulus – the simulator for this verification. This module is responsible on the design signals, that feeds the whole system – except clk and rst that will be an input from the Overall\_tb. The output of the DUT is going to the coverage and the checker modules. At first, we use the clk and rst as a signal from the stimulus itself, but then we realized that the rst and the clk should come from the verification system we change it to get form the overall\_tb, That way we gain the use of asynchronous reset tests, also this way demonstrate a closer act as a CPU. The stimulus model is responsible for following the APB Bus protocol, while feeding the DUT with its inputs. The fact that the stimulus is not allowed to read operation\_done signal, because he acts as a CPU, we needed to make sure we wait the right number of cycles before continuing to the next sample. A better way to run the test was to wait for operation\_done to be active, before continuing to the next test, but this was not the case.

Next, we have the Golden Model, representing a machine that give us perfect outputs for each stimulus input. We did it by making a Matlab code that creates random inputs, and their parities. On the other hand, we used a System Verilog module to assert the right answer simultaneously with the DUT. At first, we decide to use only Matlab to do all the calculation, but after understanding that we can’t use the Matlab in run time, we changed our plan, and we came with the idea to use System Verilog and Matlab together when Matlab is first creating the input for the stimulus, already with the right parity, and also creating with this data the input for the test and with the System Verilog, by sending it in run time to the golden model, and to the DUT. This way we let the checker and coverage modules to validate the DUT and GM output.

To check whether golden model’s and the DUT’s output match, we used the Functional checker. This module receives the output of both entities and continuously make sure both are on the same page. Also this module acts as an APB interface monitor, making sure we followed the right protocol.

Finally, the Coverage module is designed to monitor the input of our system, checking if the verity of the inputs was satisfying enough, for this verification plan. For example, checking that all the noise options were covered.

## Test Bench Low Level Architecture and Functionality

Diagram, schematic

Description automatically generated

1. ECC\_ENC\_DEC Interface – Low Level.

This is the low level of the all the overall\_tb we can see here that the clk and rst is connected from the outside and not from the stimulus. The next following pages will go into details about the functionality of each entity, and the buses and signals connecting them.

## Stimulus

Chart

Description automatically generated with low confidence

1. Stimulus Interface.

Here we have the stimulus get feed of clk and rst and sending the data of the signal (**input** clk, rst, **output** PADDR, PWDATA, PENABLE, PSEL, PWRITE, RegistersW, RegistersR, FullWord, NOISE).  
when PADDR is the wanted register to write in to , PWDATA is what we want to write to the register , PENABLE, PSEL and PWRITE relate to the APB bus protocol , RegistersW and RegistersR relate to the GM when W is when we write and R is when we read, FullWord is the generated word with the right parity into the GM for the checker to compare. NOISE is a random class we made, to generate random noises, those random vectors are assigned to the system, and connected to the coverage module to make sure all the options of noise has been covered.

The samples this module feeds the system with, were pre processed with Matlab, each one in size of 32 bits, containing a sample and its parity padded with zeroes. There are 3 files for each Code\_word\_width option. The test is parted for each of those files. At the first part, we test 8 bits data, while also checking extreme scenario after that we checking the 16 bits and 32 bits in this order with the extreme scenario of input vector of that all the bits are one.

## Functional Coverage

Text

Description automatically generated with medium confidence

1. Coverage Interface.

This Coverage with the signals (**input** clk, rst, PADDR, PWDATA, PENABLE, PSEL, PWRITE, PRDATA, data\_out, operation\_done, num\_of\_errors, NOISE,CTRL\_REG, gm\_number\_of\_errors)

The module is using all the signal as input when rst and clk are from the overall\_tb

We use them in the following covergroup “signals\_test” when this is the covergroup for all the signals simulated at the stimulus module to make sure all the data has been covered and standing by the protocol of APB bus with this cover group we checking the following :

1. reset ranged from 1:0
2. checking if the result PENABLE went to all the ranges
3. checking if the result PSEL went to all the ranges
4. checking if the result PWRITE went to all the ranges
5. checking if the result operation\_done went to all the ranges
6. PENABLE\_X\_PSEL: cross PENABLE,PSEL

Graphical user interface, application

Description automatically generated

1. Signal Coverage.

With the operation\_done signal we use as event in two cover groups the first one is amount\_of\_noise\_test we checking the following:

1. checking if the amount of noise is good or passing the oreder for only 2 error at max for the DUT
2. checking if the amount of noise is good or passing the oreder for only 2 error at max for the GM

The second cover group is Error\_spot we checking the following:

1. Checking if we got all the options of one error
2. Checking if we got all the options of two errors (Use the cross of two cover points)

Graphical user interface, application

Description automatically generated with medium confidenceGraphical user interface, application

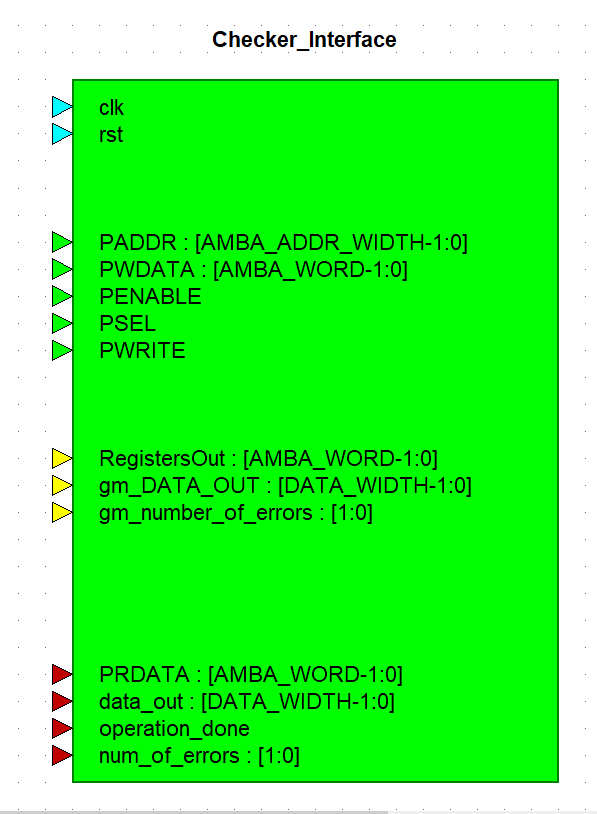
Description automatically generated

1. Num of errors coverage.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **FUNCTION** | **EVENT** | **COVERAGE POINT** | **BINS** | **scenario** |
| **reset** | **Poseedge clk , negedge rst** | **rst** | **Low = 0 ,high = 1** | **standard** |
| **PENABLE** | **Poseedge clk , negedge rst** | **PENABLE** | **Low = 0 ,high = 1** | **Standard** |
| **PSEL** | **Poseedge clk , negedge rst** | **PSEL** | **Low = 0 ,high = 1** | **Standard** |
| **PWRITE** | **Poseedge clk , negedge rst** | **PWRITE** | **Low = 0 ,high = 1** | **Standard** |
| **Operation\_done** | **Poseedge clk , negedge rst** | **Operation\_done** | **Low = 0 ,high = 1** | **Standard** |
| **APB\_bus\_rule\_test** | **Poseedge clk , negedge rst** | **APB\_bus\_Test = { PENABLE , PSEL }** | **Good = {2'b00,2'b01,2'b11}**  **illegal\_bins bad = {2'b10}** | **Standard** |
| **amount\_of\_noise\_test** | **Negedge operation\_done** | **coverage\_bus.num\_of\_errors iff(coverage\_bus.CTRL\_REG[1:0] != 2'b00)** | **no\_error = {0}**  **one\_error = {1}**  **two\_error = {2}**  **system\_error = default** | **Standard** |
| **amount\_of\_noise\_test\_gm** | **Negedge operation\_done** | **coverage\_bus.gm\_number\_of\_errors iff(coverage\_bus.CTRL\_REG[1:0] != 2'b00)** | **no\_error = {0}**  **one\_error = {1}**  **two\_error = {2}**  **system\_error = default** | **Standard** |
| **One\_error\_spot** | **negedge operation\_done** | **One\_error\_spot\_test** | **Noise\_index[] = {[0:DATA\_WIDTH-1]}**  **Noise\_two\_zero = {-1 };** | **Standard** |
| **Two\_error\_spot** | **negedge operation\_done** | **Two\_error\_spot\_test** | **Show all the cross option** | **Standard** |

1. Test Plan Functional Coverage

## Test Bench Functional Checkers



1. Functional Checker Interface.

In this module we validate the functionality of our DUT. We do this by passing the following lines from our interface:

**Inputs:** clk, rst, PADDR, PWDATA, PENABLE, PSEL, PWRITE, PRDATA, data\_out, operation\_done, num\_of\_errors, gm\_DATA\_OUT,gm\_number\_of\_errors

With those lines already mentioned and explained before, we can now cover the next properties:

**rst\_active:**

make sure all the registers and outputs are set to 0's when rst=0:

Samples when checker\_bus.rst changes.

**RegistersReadCheck:**

This property checks the functionality of the registor selector in the DUT, by comparing it to the golden model's register selector with the DUT’s. Samples when checker\_bus.clk changes, disabled iff checker\_bus.rst is low.

**operation\_done\_active:**

Make sure that after 2 to 5 cycles from the change of the CTRL register operation\_done is set to 1. Samples when checker\_bus.clk changes, disabled iff checker\_bus.rst is low.

**ResultCheck:**

Check the functionality of the result, when operation\_done is set to 1,the data that comes from the DUT are the same as the golden models, same for NumOfErrorsCheck. Samples when checker\_bus.operation\_done is active.

**NumOfErrorsCheck:**

Make sure the output num\_of\_errors from the DUT is the same as the GM’s num\_of\_errors output. Samples when checker\_bus.operation\_done is active, disabled iff checker\_bus.CTRL\_REG[1:0]== 2'b00 (Encode).

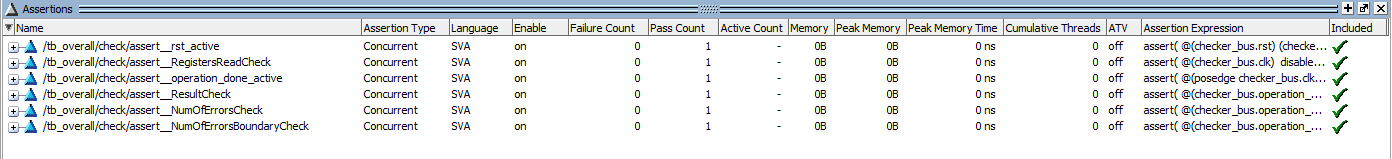
**NumOfErrorsBounderyCheck:**

Make sure num\_of\_errors is never 3 when operation\_done is 1. Samples when checker\_bus.operation\_done is active.

<<Counter Example external checker:

|  |  |  |
| --- | --- | --- |
| **Condition** | **Expected Result** | **Scenario** |
| **Reset Active in low** | **Data\_out,operation\_done, num\_of\_errors🡪0** | **Standard** |
| **Register Read** | **PRDATA == GM\_RGISTER** | **Standard** |
| **PADDR[3:2] = 00 && PWRITE&&PENABLE && PSEL are on high** | **(Only if we didn’t have Reset)[2-5] Clock poseedge 🡪  operation\_ done = 1** | **Standard** |
| **Operation\_done->1 && num\_of\_errors <= 2** | **Data\_out == GM\_data\_out** | **Standard** |
| **Operation\_done->1** | **Num\_of\_errors == GM\_num\_of\_erros** | **Standard** |
| **Operation\_done->1** | **Num\_of\_errors = [0 : 2]** | **Standard** |

1. Test Plan FunctionalCheckers



1. Functional Checker - Assertions.

Here you can see all the functional test have been validated.

## Golden Model

A picture containing diagram

Description automatically generated

1. GoldenModel Interface.

This GoldenMoudle with the signals (**input** rst, PWDATA, PADDR, RegistersW, RegistersR, FullWord, operation\_done, **output** RegistersOut, gm\_DATA\_OUT, gm\_number\_of\_errors, CTRL\_REG) The module get rst signal as well, so it can perform a reset for all its internal signals, same as the DUT.

When RegisterR/W signals are active, With the PADDR and PWDATA, we update the internal registers of this module, to keep the same values the DUT have, This way we can test and validate the data in the DUT, by reading the data and sending both GM’s and DUT’s outputs to the functional checker.

FullWord is the generated word with the right parity that matlab prepared. When operation\_done is active we write the FullWord to gm\_data\_out, so the functional checker will compare it with the DUT’s data\_out.

Opertation\_done is an input coming from the DUT, when it’s active, Full word is written as described.

RegistersOut is an output created for the validation of the functionality of the register selector module, when the APB bus protocol shows a signal for reading from a register, RegistersOut will use the PADDR to select between the golden models registers.

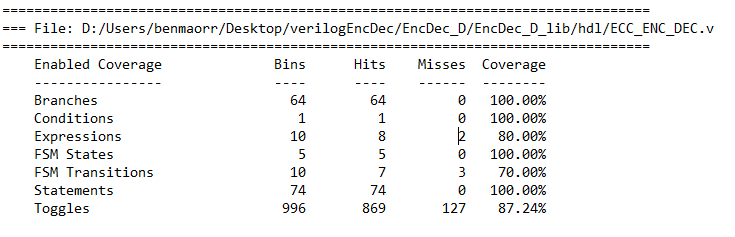
gm\_DATA\_OUT is actually FullWord, that was sampled from stimulus, it is an output for the checker to compare it to the DUT’s data\_out when operation done is active.

gm\_number\_of\_errors when operation done is active, golden model checks the amount of noise in the Noise\_Reg and set it as an output for the checker.

CTRL\_REG with this register, we know the operation that being performed, and when an Encode performance is being executed, number\_of\_errors should be 0, the same as the DUT’s.

## Code Coverage

### ECC\_ENC\_DEC



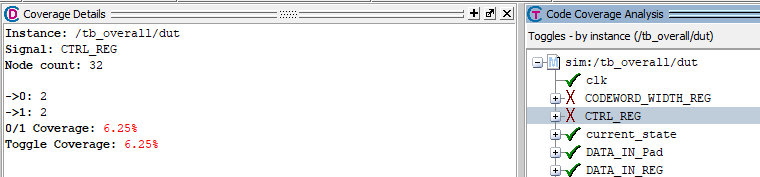
1. DUT code Coverage.

Toggles:



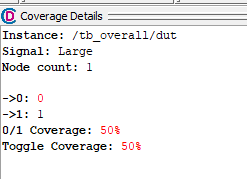
1. DUT coverage details.

Here only one bit change to zero because we start with WIDTH[1:0] = 0 and then change to 1 after that to 2  
what make the second bit to change only to 1 when WIDTH[1:0] = 2 and the rest are don’t care because we don’t using them



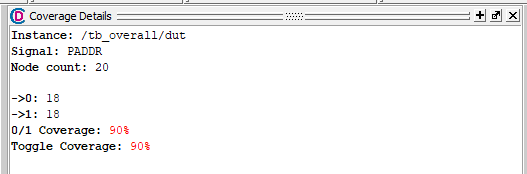
1. DUT coverage details.

Here we can see that 2 bit are 0 and 1 because we care about only 2 bits CTRL [1:0] and the rest are don’t care



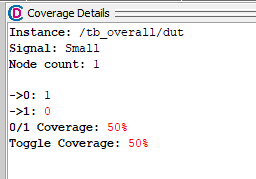
1. DUT coverage details.

Only hitting 1 because we start with 0 and the coverage don’t see change back to 0 after he got 1



1. DUT coverage details.

Here we care only for 2 bits that are PADDR [3:2] and the rest are don’t care but because we entering random number to this bus that cover more but the only last two bits we entering only 0 and don’t changing it



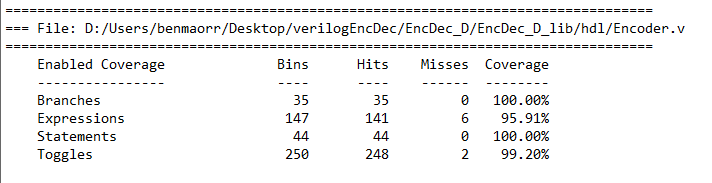
1. DUT coverage details.

Same problem as the signal Large we started as 1 and change it only to 0

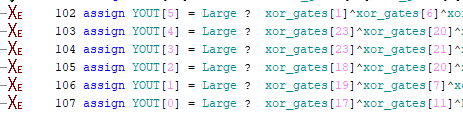
FSM Transitions:

We have 3 miss because only 3 states can’t go to one another for example from IDLE we can’t go to the state NOISE

### Encoder Code Coverage

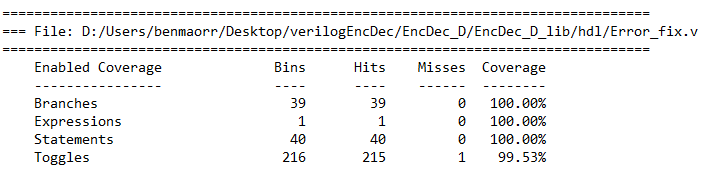


**Expressions**:



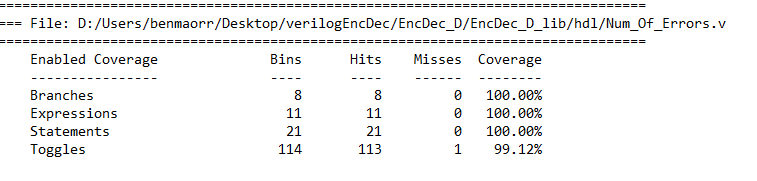
This lack of coverage is due to zero padding. When large is not active, YOUT in those places will always get a 0, this happens because the same module needs to support all the 3 types of code word width

### Error\_Fix Code Coverage



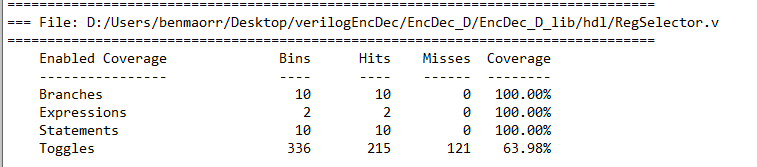
1. Error\_fix coverage details.

### Num\_Of\_Errors Code Coverage



1. Num\_of\_errors coverage details.

### Register\_Selector Code Coverage

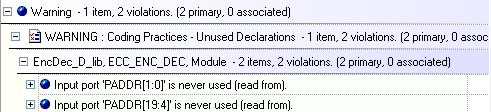


1. RegSelector coverage details.

Same explanation as before in the ecc\_enc\_dec

## Design Checker Approved Errors

1. DUT Design Ignored errors.

We can ignore this error, the rst signal is coming from the overall\_tb

1. DUT Design Ignored warnings.

This bits are kept for reading, but we used only the offset at curtain bits.

# Appendix

## Terminology

**LSB** - Least Significant Bit

**TBR** - To Be Reviewed

**TBD** - To Be Defined

**IF** - Inteface

**DUT** - Device Under Test

**GM** - Golden Model

**tb** - Test Bench

## References