Error Correction Encoder & Decoder

Digital Design and Logical Synthesis for Electrical Computer Engineering

Digital High Level Verification

Version 0.1

By: Aharon lugassy

Saar Avraham

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

1. VERIFICATION PLAN……6

1.1 Verification Test Objectives……6

1.2 Test Bench High Level Diagram and Architecture……7

2. VERIFICATION IMPLEMENTATION……9

2.1 Stimulus……9

2.2 Functional Coverage……10

2.3 Functional Checker……11

2.4 Golden-Model……12

2.5 Functional Coverage table……15

2.6 Functional Checker table……16

2.7 High level Golden Model……17

3. VERIFICATION RESULTS……18

3.1 Functional Coverage report……18

3.2 Functional Checker report……19

3.3 Coverage Report……20

3.4 Golden Model Comparison……21

4. APPENDIX……22

4.1 Notes……22

4.2 Terminology……22

4.3 References……22

LIST OF FIGURES

Figure 1. high level block diagram 7
Figure 2. APB command example 9
Figure 3. Coverage coverpoint example 10
Figure 4. Coverage bins result example 10
Figure 5. Functional Checker property example 11
Figure 6. Functional Checker assertion result example 11
Figure 7. Functional Coverage report 18
Figure 8. Functional Checker report 19
Figure 9. Coverage report 20
Figure 10. Waveform results 21

LIST OF TABLES

Table 1: Golden model block 13
Table 2: Stimulus block 14
Table 3: Checker and Coverage block 14

1. VERIFICATION PLAN

In this lab we will perform verification of the ecc_enc_dec (the DUT) design we created in lab 1. Our plan for this verification will be to devise a test-bench which will thoroughly use and test every part of the design we built. This test will include a test for each operation our system is designed for, that being Encode, Decode, and Full Channel. We will test these 3 operation on each of the 3 input sizes, for a total of 9 tests. During the tests we will examine 3 main aspects while running the test_bench: Correctness of outputs, coverage of main signals, and functional behavior of signals. Since our output's length is 32 bits, it would be redundant to try and examine every single case, as we would get 2^{32} outputs. To solve this, we will randomize inputs in all lengths, save them in a file, and have a corresponding file with the correct output for each case. While the system runs, we will test the coverage and functional checker.

1.1 Verification Test Objectives

Our main verification test objective is to make sure the system is giving us a correct output to a valid input. As we cannot test every single input, we will use Coverage to make sure a wide enough range of inputs have been tested. We also have to make sure the outputs are valid and timed correctly, hence the use of the Functional checker.

What our test bench does in practice, is activating our ecc_enc_dec using the correct APB protocol, and check its output for correctness.

1.2 Test Bench High Level Diagram and Architecture

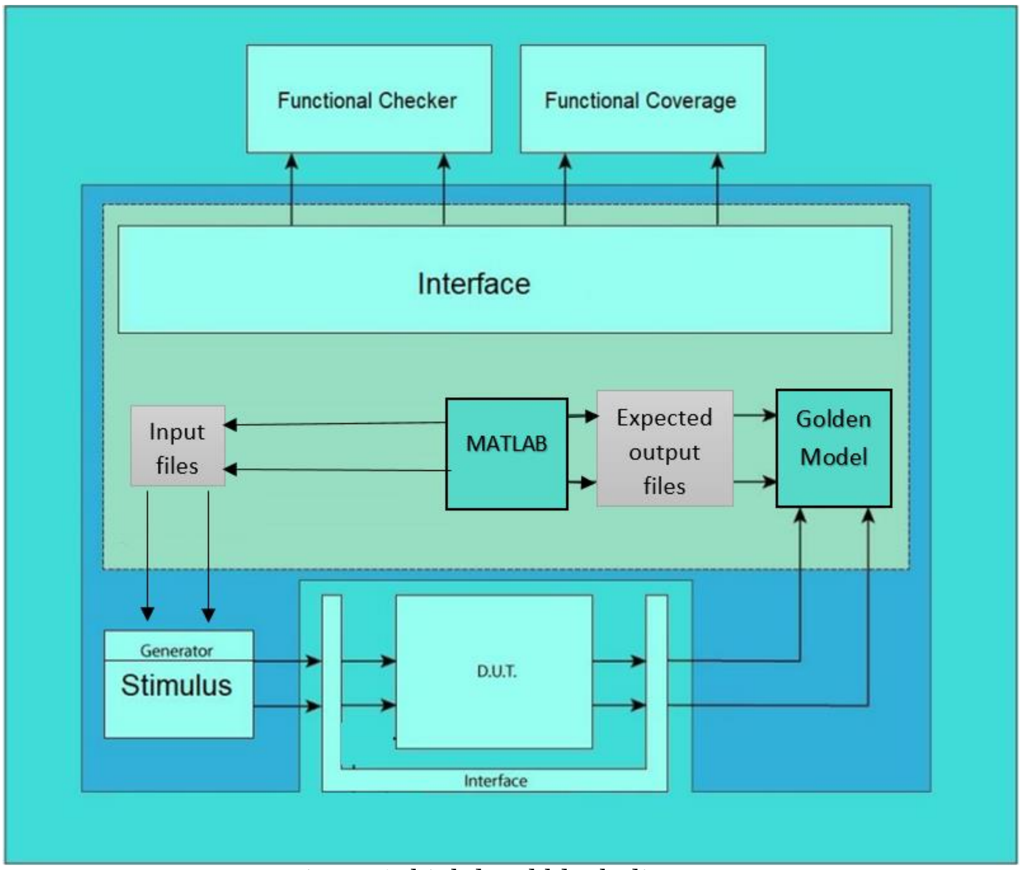


Figure 1. high level block diagram

The Stimulus is our main module for this test bench and is the driver for all actions. We connect the DUT to the Stimulus using the Interface. Inside the Stimulus we generate the inputs of the DUT, and so in order for the DUT to react correctly, we must simulate an APB master unit. We do that by creating two-cycle pattern, in which the control signals are generated appropriately, resulting in a correct instruction to write/read one of the APB registers. The content of the input commands to the registers is coming out of an external text file. We also generate the Clock and Reset signals from the stimulus and connect it to every module, synchronizing the entire test bench.

We chose to have every module in our test bench be connected via the Interface. This allows us to easily share needed signals, and overall simplify the design.

The Golden Model module is where we check the correctness of the DUT output. It works by extracting a binary line, which corresponds to the input line we sent the DUT, and compare it with the DUT’s output. We have considered to have the compare action take place in another, specified module, but decided against it as it would be more simple and efficient to do it in the same module which receives the text line. Upon comparing, we keep record of hit/miss registers, updating in each iteration. We also compare the number of errors in case the last operation that took place was Full Channel.

The Coverage unit purpose is to track system events and signals, which will help us decide if we have covered enough scenarios in the test bench.

The Functional Checker purpose is to make sure that upon specific events, important signals holds a correct and valid value.

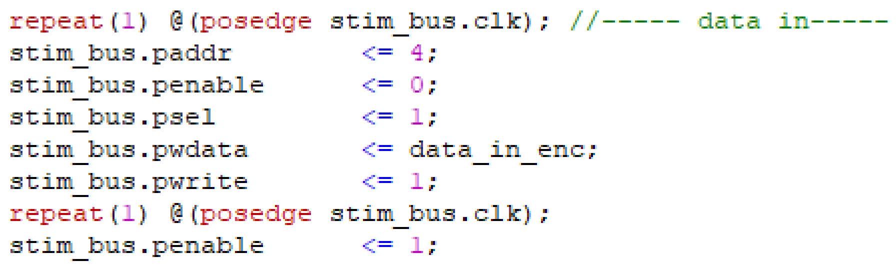
Both the Coverage and the Functional Checker are connected separately to the interface, making sure both work independently and do not rely on each other, making the system more fail-safe.

2. VERIFECATION IMPLEMENTATION

2.1 Stimulus

This module is the generator for all operations in the test bench, and act as a traditional tb unit, generating the signals and sending them as inputs to the DUT and other test units. As established, we need to test the system for every operation, and every codeword length. We chose to implement this by using a fixed codeword length, then have the system do encode, decode, and full channel operations one after the other, all inputs determined by a line from pre-made text files (more on this in Golden Model explanation). We then load another line from each file, and go again with the three operation. Once we reach the end of the files, we switch files, moving to files corresponding to a different codeword length, and go again. To implement this we use a for loop of 3 iteration, where in each beginning of the loop we load the correct files, then we have another for loop where in each iteration we load different lines, and send the DUT the inputs.

To operate the DUT, we have to use the APB protocol. Below is how we implemented this in the Stimulus.



```
repeat(1) @(posedge stim_bus.clk); //----- data in-----
stim_bus.paddr        <= 4;
stim_bus.penable      <= 0;
stim_bus.psel         <= 1;
stim_bus.pwdata       <= data_in_enc;
stim_bus.pwrite       <= 1;
repeat(1) @(posedge stim_bus.clk);
stim_bus.penable      <= 1;
```

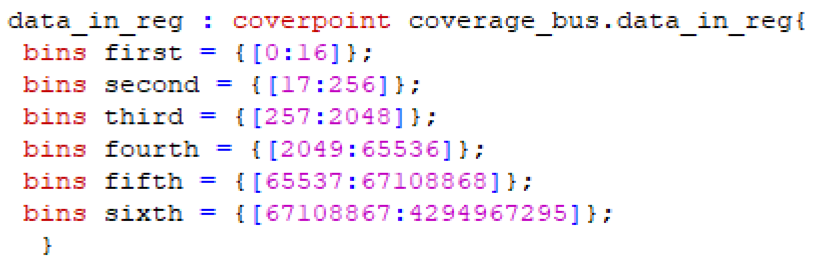
Figure 2. APB command example

This is an example for DATA_IN loading command for an encode operation. You can see we raise the correct APB signals, and load the data_in_enc to pwdata, as in “setup” phase. After a single cycle delay, we raise the penable signal, corresponding to the “access” phase. (in each iteration, data_in_enc holds the value of 1 line from the ‘encoder input’ file)

2.2 Functional Coverage

In this module we want to see that we covered enough options for the inputs, and if important signals got a wide enough range of values.

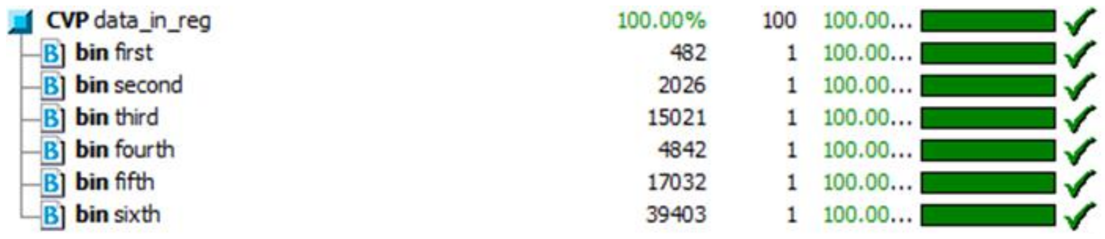
To implement this, we do a coverage for each signal independently, making sure those signals are accepted as inputs to this module. For instance, we want to make sure the DATA_IN register get a wide variety of values, as we know the input value to the ecc_enc_dec can be up to 2^{32}, for that we have the following coverage:



```
data_in_reg : coverpoint coverage_bus.data_in_reg{
 bins first = {[0:16]};
 bins second = {[17:256]};
 bins third = {[257:2048]};
 bins fourth = {[2049:65536]};
 bins fifth = {[65537:67108868]};
 bins sixth = {[67108867:4294967295]};
  }
```

Figure 3. Coverage coverpoint example

We devide the values the data_in into 6 bins, corresponding to 6 possibilities for input of the ecc_enc_dec (3 for encode, 3 others for decode). You can see in the following picture the distribution of those bins:

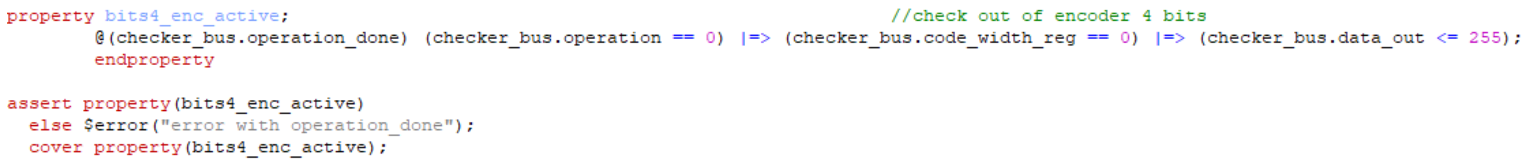


CVP data_in_reg	100.00%	100	100.00...	🟩	✓
bin first	482	1	100.00...	🟩	✓
bin second	2026	1	100.00...	🟩	✓
bin third	15021	1	100.00...	🟩	✓
bin fourth	4842	1	100.00...	🟩	✓
bin fifth	17032	1	100.00...	🟩	✓
bin sixth	39403	1	100.00...	🟩	✓

Figure 4. Coverage bins result example

2.3 Functional Checker

In this module, we are validating the correct outcome of specific signals, in specific events. To do this we are using the assert function, making sure that in specific conditions, which we determine, a certain signal holds a certain value, for example:



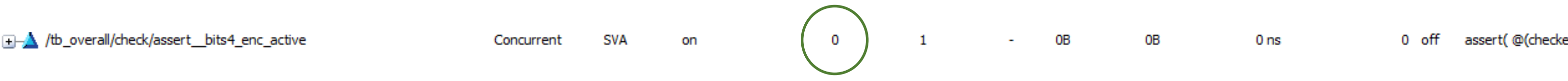
```
property bits4_enc_active;                                              //check out of encoder 4 bits
        @(checker_bus.operation_done) (checker_bus.operation == 0) |=> (checker_bus.code_width_reg == 0) |=> (checker_bus.data_out <= 255);
        endproperty

assert property(bits4_enc_active)
  else $error("error with operation_done");
  cover property(bits4_enc_active);
```

Figure 5. Functional Checker property example

Here we can see the checker for the data out, in the event of encoding a 4 bits word. You can see the condition is having operation = 0, meaning operation is encoding, and code_width_reg = 0, meaning we are in 4 bits format. Based on these conditions, we know the output should be no longer than 8 bits word, meaning its value by numerical can be no bigger than 255, thus we have the property ‘data out’ <= 255. If at any point in the test bench, having the conditions we determined, the output will be bigger than 255, we will know a mistake has occurred and will record it.

We can see the result for this check, assuring us that in no point of the test bench, this condition has been violated.



```
/tb_overall/check/assert__bits4_enc_active    Concurrent   SVA   on   0   1   -   0B   0B   0 ns   0  off   assert( @(checke
```

Figure 6. Functional Checker assertion result example

2.4 Golden-Model

In this module, we check the correctness of our output. We receive the output signals from the DUT, and the needed signals from the stimulus to know when to move iteration. Similar to the stimulus, we need to read the pre-made text files holding the correct values for each of the input files we sent to the DUT. We chose to implement this using 3 sensitivity lists.

First one is reacting to the change of the codeword length. Doing so, we can ‘know’ when the files have been switched in the stimulus, meaning we have to switch files in the Golden Model too. The switching to the next file is done similarly to the Stimulus.

Second one is reacting to a specified signal called ‘next line’, which is coming out of the stimulus specifically to the Golden Model. As its name suggests, This signal tells us a line has been switched in the Stimulus, and so we need to switch line in the Golden Model too.

Third one is reacting to the operation_done signal from the DUT. This tells us that an operation has took place, and so we need to make a comparison, and see if the DUT output matches the correct result from the files. We use ‘operation’ signal from the Stimulus to know which operation last took place, and then compare the DUT output with the correct value. If its matches, we count 1 up to the ‘hit’ register, otherwise we count 1 up for the ‘miss’ register.

signals	length	I/O
data_out	[DATA_WIDTH-1 :0]	output
operation_done		output
num_of_errors	[1:0]	output
operation	[1:0]	output
next_line		output
code_width	[1:0]	output

Table 1: Golden model block

signals	length	I/O
clk		output
reset		output
paddr	[AMBA_ADDR_WIDTH-1:0]	output
penable		output
psel		output
pwdata		output
pwrite	[AMBA_WORD-1:0]	output
operation	[1:0]	output
control_reg	[AMBA_WORD-1:0]	output
data_in_reg	[AMBA_WORD-1:0]	output
code_width_reg	[AMBA_WORD-1:0]	output
noise_reg	[AMBA_WORD-1:0]	output
prdata	[AMBA_WORD-1 :0]	output
data_out	[DATA_WIDTH-1 :0]	output
operation_done		output
num_of_errors	[1:0]	output

Table 2: Stimulus block

signals	length	I/O
clk		input
reset		input
paddr	[AMBA_ADDR_WIDTH-1:0]	input
penable		input
psel		input
pwdata		input
pwrite	[AMBA_WORD-1:0]	input
operation	[1:0]	input
next_line		input
code_width	[1:0]	input
control_reg	[AMBA_WORD-1:0]	input
data_in_reg	[AMBA_WORD-1:0]	input
code_width_reg	[AMBA_WORD-1:0]	input
noise_reg	[AMBA_WORD-1:0]	input

Table 3: Checker and Coverage block

2.5 Functional Coverage table

FUNCTION	EVENT	COVERAGE POINT	BINS	scenario
Reset	**Posedge clock**	**Reset**	**0,1**	**standard**
Codeword width register	**Posedge clock**	**code_width_reg**	**0,1,2**	**Standard**
noise register	**Posedge clock**	**noise_reg**	**0,1,2**	**Standard**
Data in register	**operation**	**data_in_reg**	$\mathbf{0 \to 2^4 - 1}$ $\mathbf{2^4 \to 2^8 - 1}$ $\mathbf{2^8 \to 2^{11} - 1}$ $\mathbf{2^{11} \to 2^{16} - 1}$ $\mathbf{2^{16} \to 2^{26} - 1}$ $\mathbf{2^{26} \to 2^{32} - 1}$	**Standard**
Paddr signal	**Posedge clock**	**paddr**	**0,4,8,12**	**Standard**
Penable signal	**Posedge clock**	**penable**	**0,1**	**Standard**
Psel signal	**Posedge clock**	**psel**	**0,1**	**Standard**
pwrite signal	**Posedge clock**	**pwrite**	**0,1**	**Standard**
control register	**control_reg**	**control_reg**	**0,1,2**	**Standard**

2.6 Functional Checker table

Condition	**Expected Result**	**Scenario**
Changing control register	**control_reg < 3**	**Standard**
Changing codeword width register	**control_reg < 3**	**Standard**
Changing noise register	**Num_of_error < 3**	**Standard**
Encoding 4 bits	$\boldsymbol{Data\ out < 2^{8} - 1}$	**Standard**
decoding 4 bits	$\boldsymbol{Data\ out < 2^{4} - 1}$	**Standard**
Full channel 4 bits	$\boldsymbol{Data\ out < 2^{8} - 1}$	**Standard**
Encoding 11 bits	$\boldsymbol{Data\ out < 2^{16} - 1}$	**Standard**
decoding 11 bits	$\boldsymbol{Data\ out < 2^{11} - 1}$	**Standard**
Full channel 11 bits	$\boldsymbol{Data\ out < 2^{16} - 1}$	**Standard**
Encoding 26 bits	$\boldsymbol{Data\ out < 2^{32} - 1}$	**Standard**
decoding 26 bits	$\boldsymbol{Data\ out < 2^{26} - 1}$	**Standard**
Full channel 26 bits	$\boldsymbol{Data\ out < 2^{32} - 1}$	**Standard**

2.7 High level Golden Model

As we explained previously, our strategy for this test bench was to use text files made beforehand using Matlab. The algorithm is very simple. We start with a codeword in the desired length. For the 4 bit words we use every combination possible, giving us 16 lines saved in ‘encode input’ file. For the other 2 codeword length, we randomize 1000 input words in the correct length and save them in two other ‘encode input’ files. After that, we have to create a correct encoded word for each of the words in the input files. We do this exactly as explained in LAB 1, deriving the linear equations for each parity bit from the matrix multiplication, then concatenate them to the original word. We then save those in three ‘decoder input’. In addition, we create three noise files. First one having 16 lines, other two 1000, their length the same as the ‘decoder input’ files. We randomize each line to have between 0 and 2 bits, and randomize their location, giving us correct error vectors.

In the Golden Model module itself, we use the same files we used in the stimulus, but in a different order. Once we read 3 files in the same codeword length, we know the stimulus is ordering the DUT to operate encode, decode, and full operation one after the other. So, we compare the first output of the DUT with the first line of the ‘decoder input’ file, then compare the second output with the ‘encode input’ file. For the full channel, we know the output ideally should be the same as the input, so we compare the output with the first line of the ‘encoder input’ file. We also check beforehand how many errors were in the noise vector. If we had 2 errors, we don’t check the output and only check if the num_of_errors output from the DUT equals 2. Otherwise, we compare both the output and num_of_errors, only adding to the ‘hit’ register if both match the values from the files.

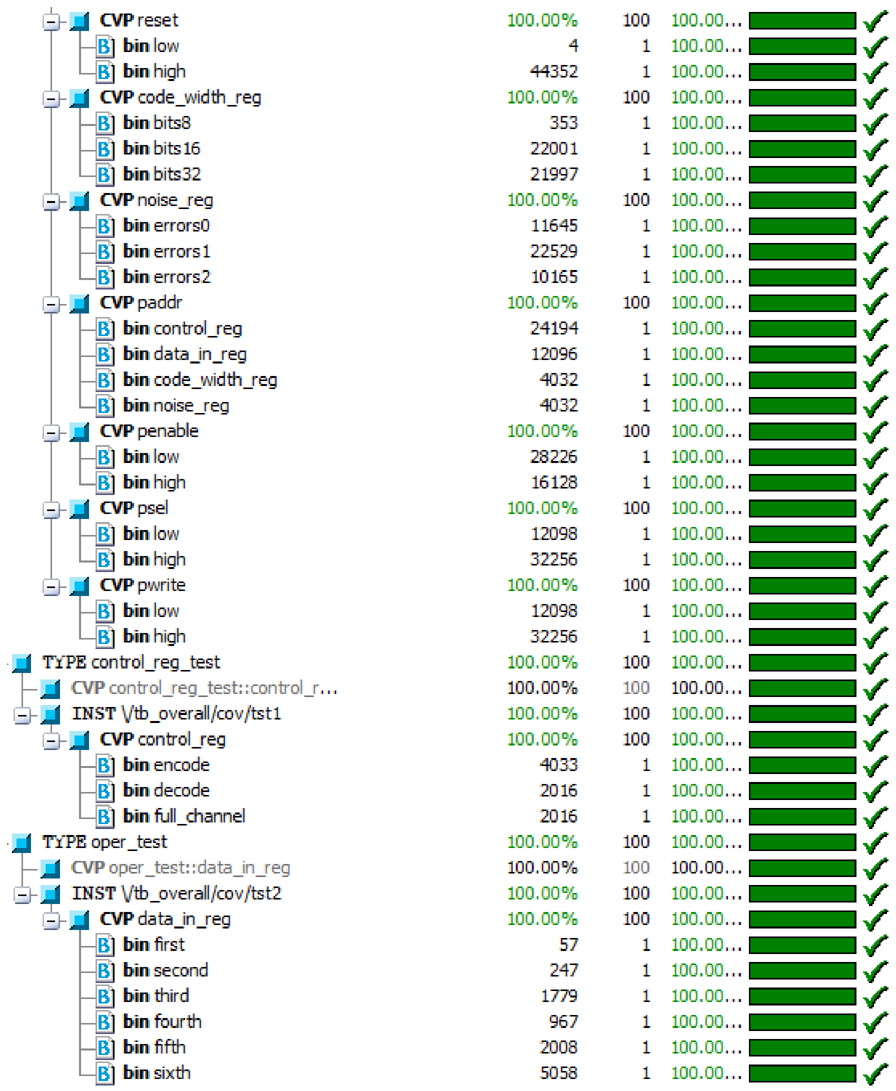
3. VERIFICATION RESULTS

3.1 Functional Coverage report

Here you can see the official Functional Coverage report. Overall, the results line up with our expectation. We can see for example the cover group for the code_word_reg, showing us an almost equal distribution between the two longer lengths, and a much lower number of the short length codeword, as we expected from out test bench plan.

In addition, we can see that for the noise register we have about equal probability between having zero errors and having 2 errors, both are about half of the probability to get a 1 error vector. This matches our expectation as we random 2 bits to get a value from {0,1}.

For the last example, we can look at the control_reg. We can see that both decode and full channel have been operated 2016 times, exactly as they should based on our test plan. The encode operation should be 2016 too, but is counted twice. This is a mistake we assume happens at the end of each iteration,cause by the way we are generating signals in the Stimulus.

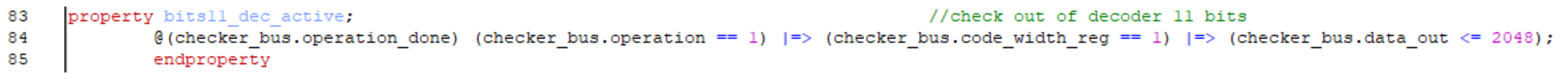


```
CVP reset                            100.00%   100  100.00...
    bin low                                4     1  100.00...
    bin high                           44352     1  100.00...
CVP code_width_reg                   100.00%   100  100.00...
    bin bits8                            353     1  100.00...
    bin bits16                         22001     1  100.00...
    bin bits32                         21997     1  100.00...
CVP noise_reg                        100.00%   100  100.00...
    bin errors0                        11645     1  100.00...
    bin errors1                        22529     1  100.00...
    bin errors2                        10165     1  100.00...
CVP paddr                            100.00%   100  100.00...
    bin control_reg                    24194     1  100.00...
    bin data_in_reg                    12096     1  100.00...
    bin code_width_reg                  4032     1  100.00...
    bin noise_reg                       4032     1  100.00...
CVP penable                          100.00%   100  100.00...
    bin low                            28226     1  100.00...
    bin high                           16128     1  100.00...
CVP psel                             100.00%   100  100.00...
    bin low                            12098     1  100.00...
    bin high                           32256     1  100.00...
CVP pwrite                           100.00%   100  100.00...
    bin low                            12098     1  100.00...
    bin high                           32256     1  100.00...
TYPE control_reg_test                100.00%   100  100.00...
  CVP control_reg_test::control_r... 100.00%   100  100.00...
  INST \/tb_overall/cov/tst1         100.00%   100  100.00...
    CVP control_reg                  100.00%   100  100.00...
      bin encode                        4033     1  100.00...
      bin decode                        2016     1  100.00...
      bin full_channel                  2016     1  100.00...
TYPE oper_test                       100.00%   100  100.00...
  CVP oper_test::data_in_reg         100.00%   100  100.00...
  INST \/tb_overall/cov/tst2         100.00%   100  100.00...
    CVP data_in_reg                  100.00%   100  100.00...
      bin first                           57     1  100.00...
      bin second                         247     1  100.00...
      bin third                         1779     1  100.00...
      bin fourth                         967     1  100.00...
      bin fifth                         2008     1  100.00...
      bin sixth                         5058     1  100.00...
```

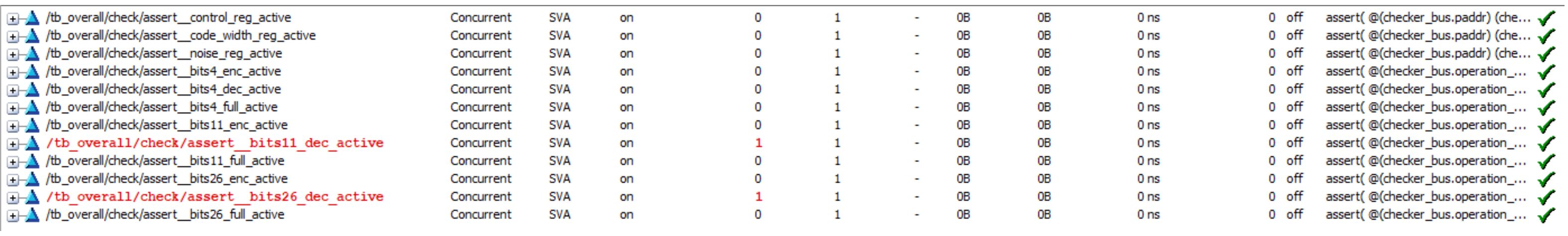
Figure 7. Functional Coverage report

3.2 Functional Checker report

Here you can see the official report of the Functional Checker. We can see that almost every signal we checked is working as intended throughout the entire test bench. We have two errors occurring for the property regarding the data_out validity. The assertion is for making sure that for decode operation, considering we have code_word equaling 16 bits input, we should get an output no longer than 11 bits. Meaning the output result should be no bigger by numeric value than 2^{11}. We can see that this assertion holds for the entire test bench but for one occasion. We have not managed to understand when this error takes place, but are assuming this happens for the final iteration before switching files to a bigger code word size, and is caused by delay of the assertion property signals and the actual values that the data output holds.



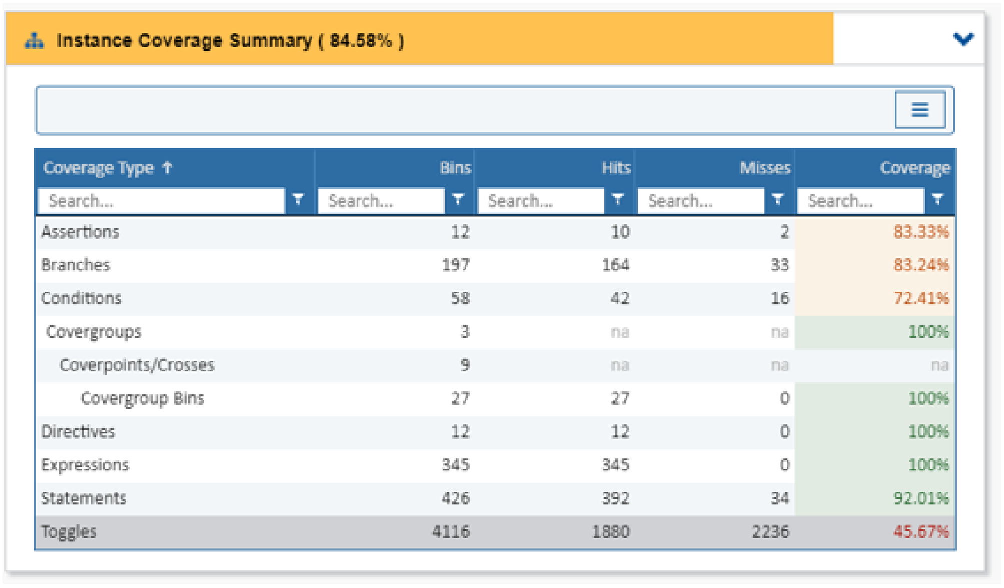
```
property bits11_dec_active;                                   //check out of decoder 11 bits
      @(checker_bus.operation_done) (checker_bus.operation == 1) |=> (checker_bus.code_width_reg == 1) |=> (checker_bus.data_out <= 2048);
      endproperty
```



/tb_overall/check/assert__control_reg_active	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(checker_bus.paddr) (che...	✓
/tb_overall/check/assert__code_width_reg_active	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(checker_bus.paddr) (che...	✓
/tb_overall/check/assert__noise_reg_active	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(checker_bus.paddr) (che...	✓
/tb_overall/check/assert__bits4_enc_active	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(checker_bus.operation_...	✓
/tb_overall/check/assert__bits4_dec_active	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(checker_bus.operation_...	✓
/tb_overall/check/assert__bits4_full_active	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(checker_bus.operation_...	✓
/tb_overall/check/assert__bits11_enc_active	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(checker_bus.operation_...	✓
/tb_overall/check/assert__bits11_dec_active	Concurrent	SVA	on	1	1	-	0B	0B	0 ns	0	off	assert(@(checker_bus.operation_...	✓
/tb_overall/check/assert__bits11_full_active	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(checker_bus.operation_...	✓
/tb_overall/check/assert__bits26_enc_active	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(checker_bus.operation_...	✓
/tb_overall/check/assert__bits26_dec_active	Concurrent	SVA	on	1	1	-	0B	0B	0 ns	0	off	assert(@(checker_bus.operation_...	✓
/tb_overall/check/assert__bits26_full_active	Concurrent	SVA	on	0	1	-	0B	0B	0 ns	0	off	assert(@(checker_bus.operation_...	✓

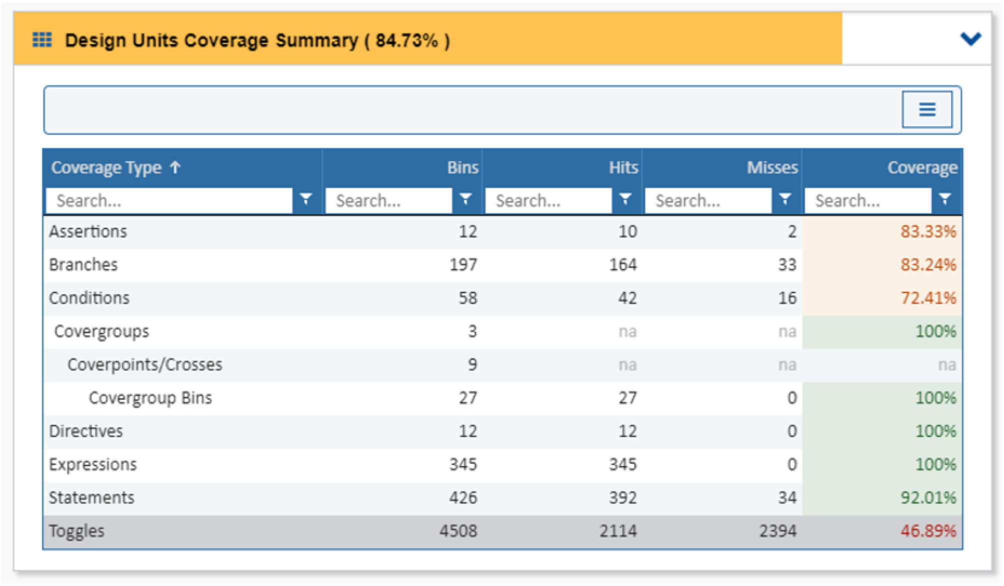
Figure 8. Functional Checker report

3.3 Coverage Report



Instance Coverage Summary (84.58%)

Coverage Type	Bins	Hits	Misses	Coverage
Assertions	12	10	2	83.33%
Branches	197	164	33	83.24%
Conditions	58	42	16	72.41%
Covergroups	3	na	na	100%
Coverpoints/Crosses	9	na	na	na
Covergroup Bins	27	27	0	100%
Directives	12	12	0	100%
Expressions	345	345	0	100%
Statements	426	392	34	92.01%
Toggles	4116	1880	2236	45.67%



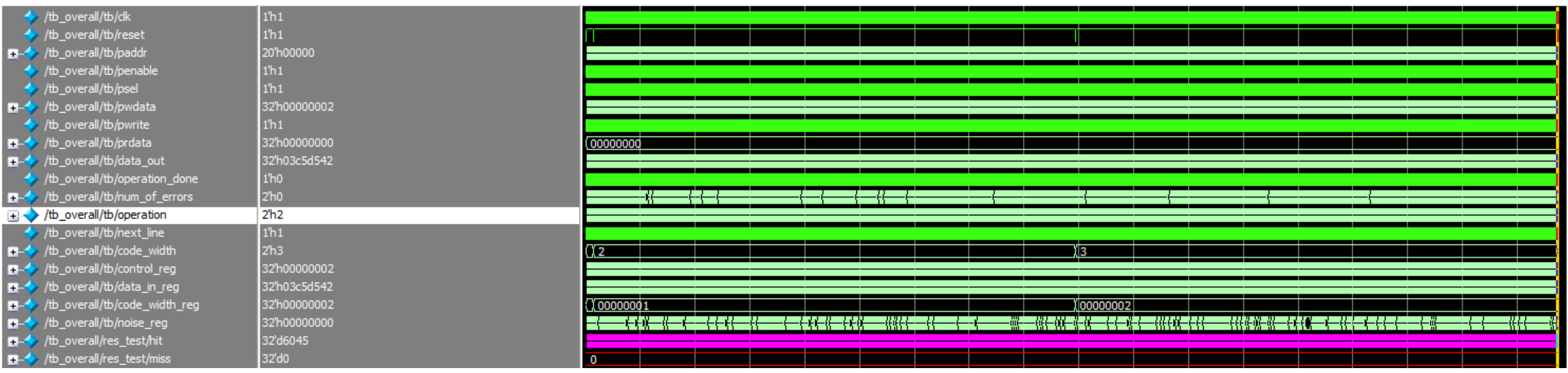
Design Units Coverage Summary (84.73%)

Coverage Type	Bins	Hits	Misses	Coverage
Assertions	12	10	2	83.33%
Branches	197	164	33	83.24%
Conditions	58	42	16	72.41%
Covergroups	3	na	na	100%
Coverpoints/Crosses	9	na	na	na
Covergroup Bins	27	27	0	100%
Directives	12	12	0	100%
Expressions	345	345	0	100%
Statements	426	392	34	92.01%
Toggles	4508	2114	2394	46.89%

Figure 9. Coverage report

3.4 Golden Model Comparison

Here we can see the resulting waveform for our test bench, where we can see the ‘hit’ and ‘miss’ registers. As it shows, every test of the 6048 made in this test bench is correct, meaning our ecc_end_dec is working as intended.



```
/tb_overall/tb/clk              1'h1
/tb_overall/tb/reset            1'h1
/tb_overall/tb/paddr            20'h00000
/tb_overall/tb/penable          1'h1
/tb_overall/tb/psel             1'h1
/tb_overall/tb/pwdata           32'h00000002
/tb_overall/tb/pwrite           1'h1
/tb_overall/tb/prdata           32'h00000000
/tb_overall/tb/data_out         32'h03c5d542
/tb_overall/tb/operation_done   1'h0
/tb_overall/tb/num_of_errors    2'h0
/tb_overall/tb/operation        2'h2
/tb_overall/tb/next_line        1'h1
/tb_overall/tb/code_width       2'h3
/tb_overall/tb/control_reg      32'h00000002
/tb_overall/tb/data_in_reg      32'h03c5d542
/tb_overall/tb/code_width_reg   32'h00000002
/tb_overall/tb/noise_reg        32'h00000000
/tb_overall/res_test/hit        32'd6045
/tb_overall/res_test/miss       32'd0
```

Figure 10. Waveform results

4. APENDIX

4.1 Notes

Dealing with the early stages of the verification, we have discovered an error for the largest codeword width encode operation. Upon inspection, we could tell the fourth bit was coming out wrong, and by extension causing a mistake for the sixth bit too. We have found out the error to be a simple typo for the calculation of the fourth parity bit, and have easily fixed this. We have uploaded the updated version of this encoder.

4.2 Terminology

LSB	-	Least Significant Bit
TBR	-	To Be Reviewed
TBD	-	To Be Defined
IF	-	Inteface

4.3 References