BY: Tom Kessous 206018749

Dan Ben Ami 316333079

Error Correction Encoder & Decoder

Digital Design and Logical Synthesis for Electric Computer Engineering (36113611)
Course Project

Verification

Version 1.0

Revision Log

Rev	Change	Description	Reason for change	Done By	Date
0.1	Initial document			Tom Kessous	12,Dec,2021
0.2					
0.3					

Table of Content

LIST OF FIGURES

LIST OF TABLES

1. VERIFICATION PLAN

As we learned in class, we designed a verification plan that including the following components: Coverage, Checker, Stimulus, Interface, Python Gold model, Compare and tb_overall. In addition, we used Python as high-level language in order to generate text files, each file contain the desired inputs or outputs of our DUT. Since python is high-level language, we can perform complex algorithms and calculations with relative convenient manner and confident. We made generalization across all text files, the input text files named "golden_model_inputs_XY" and the output text files named "golden_model_outputs_XY". where X indicate on the operation to perform and Y indicate on the amount of bits.

value	X	Y
0	Encode	8-bits
1	Decode	16-bits
2	Full Channel	32-bits

Table 1: Generalization of the files

Since there are reasonable amount of inputs in 8 and 16 bits, we generated all possible inputs. For the 8 bits there are 16 (because there are 4 bits of data - 2^4) possible legal inputs. Similarly, For 16 bits there are 2048 (2^11) possible legal inputs. In 32 bits option there are 26 bits of data, so there are 2^26 possible inputs. That amount of inputs is too large to cover, so we generated inputs according the following pattern: we generate inputs in interval of 256 and in each interval we randomized an additional input. For instance, the first input is zeros inputs (26 bits of 0), the second input is a random number between 1-255 (in binary), the third input is 256 (in binary) the next input is random between 256-511 (in binary) and so on. In that pattern we cover 0.78% (total of 524,288 inputs) of all possible inputs. In addition, we tested the read operation from the all registers.

Our verification environment create a text report that inform the user about the verification tests results. In case of difference between the golden model and the DUT, the DUT result and the golden model result will write to the report. At the end of each test the hit and miss rate will write to the report.

The DUT passed all the tests (described in table 2) successfully without any errors, The results also attached in reporst.txt file.

In the table below we describe the test flow of our verification:

test number	functionality being tested	Test data set	expected result
00	Encode 8-bits	golden_model_inputs_00	golden_model_outputs_00
01	Encode 16-bits	golden_model_inputs_01	golden_model_outputs_01
02	Encode 32-bits	golden_model_inputs_02	golden_model_outputs_02
10	Decode 8-bits	golden_model_inputs_10	golden_model_outputs_10
11	Decode 16-bits	golden_model_inputs_11	golden_model_outputs_11
12	Decode 32-bits	golden_model_inputs_12	golden_model_outputs_12
20	Full Channel 8-bits	golden_model_inputs_20 & noise_0	golden_model_outputs_20
21	Full Channel 16-bits	golden_model_inputs_21 & noise_1	golden_model_outputs_21
22	Full Channel 32-bits	golden_model_inputs_22 & noise_2	golden_model_outputs_22
registers	Reading from registers	Stimulus generate the inputs	Taken from Stimulus

Table 2: Test flow

All the files in the table above are in hdl->verification files.

1.1 Verification Test Objectives

The main objective of our verification test is to make sure that our model perform as designed to. We try to feed our DUT with variety of inputs and then compare the DUT output with the correct output (the correct output computed with python).

1.2 Test Bench High Level Diagram and Architecture

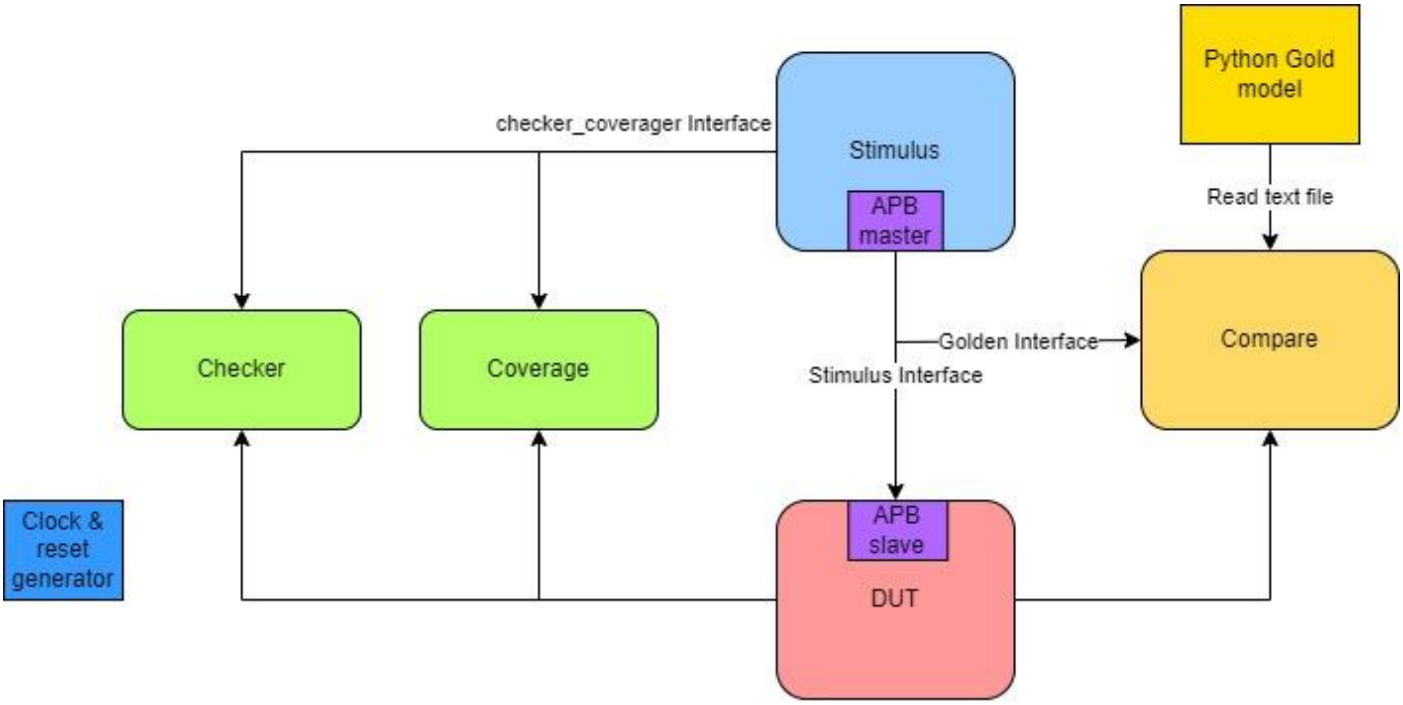

Figure 1: Test Bench High Level Diagram

The purpose of the stimulus is to generate inputs signals to the DUT. It contains and APB master which is responsible to toggle APB bus properly. It also contains a files reader which read an text file which contain legal inputs made by Python. The Stimulus read file after file as described in table 2, and in each file read line after line. Each line contain input to the DUT according to the current operation. In addition, the Stimulus generate input to check the read operation from the registers. It also contains a APB protocol function which receive an address, data and read/write flag. The APB protocol function perform a legal transaction according to the APB protocol.

The decided to connect clock and reset signals from a generator, to all blocks in the testbench, and not from stimulus because if in the future our DUT will be part of a bigger system, we would like that our design will work with the system clock and not an internal clock.

The role of the Compare is to understand what are the inputs to the DUT, what are the outputs from the DUT, and to check if the expected results match the actual results. The Compare read the correct result from text file made by Python, and then compare the DUT output to the correct result.

The role of the Checker is to perform functional checkers on the code, using properties. The checker validate that the code perform as designed to. For example, we check if the registers address in the right range.

We created the APB interface with three modports: dut/slave, stimulus/master, and checker_coverager because each module need different inputs and output. With these modports we can work in a generic and scalability manner.

The purpose of the Coverage module is to track verification events and signals, and tell us if we covered all the scenarios that needed to be covered.

We debated to which modules we want to connect the Coverage module. First option was to connect it to the two interfaces, so we cover all scenarios directly from signals that are connected to DUT. Second options was to connect Coverage to the Stimulus and Compare so we can easily cover both the transactions that we decide to send to DUT, and the outputs that we get and the scenarios that the Compare found that are happening. The second option has a big disadvantage that we cannot take this testbench to a higher hierarchy (when ECC_ENC_DEC is tested as part of a bigger design) by simply disconnecting the stimulus and connecting to another design. The third option was to take the signals that are connected to DUT in addition to one more signal that goes from the Stimulus directly to the Coverage ("operation"). We decided to go with the third option because this way we can take coverage on the "num_of_errors" signal only when it is relevant i.e. in the decode mode and in full channel mode. With this option there is a slightly lack of scalability because the reason that was mentioned, but in this trade-off we decided that the "num_of_errors"`s relevancy is more important.

1.3 Test Bench Low Level Architecture and Functionality

Our verification architecture build from 5 main components: Stimulus, Gold Model, Compare, Checker, Coverage.

1.3.1 Coverage

The purpose of the stimulus is to generate inputs signals to the DUT. It contains and APB master which is responsible to toggle APB bus properly. It also contains a files reader which read an text file which contain legal inputs made by Python. The Stimulus read file after file as described in table 2, and in each file read line after line. Each line contain input to the DUT according to the current operation. In addition, the Stimulus generate input to check the read operation from the registers. It also contains a APB protocol function which receive an address, data and read/write flag. The APB protocol function perform a legal transaction according to the APB protocol.

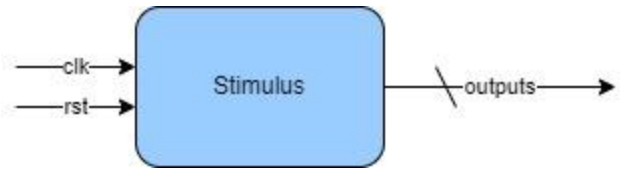

Figure 2: Stimulus

Outputs list: PADDR, PENABLE, PSEL, PWDATA, PWRITE, operation, val.

Operation indicate on the current operation (Encode, Decode or Full channel).

Val is the value that written to a register during the read from register check.

1.3.2 Gold Model

We wrote the Gold Model in Python. Since python is high-level language, we can perform complex algorithms and calculations with relative convenient manner and confident. We simulate our DUT operation in python and generated for each operation and codeword length a text files that contain the inputs and outputs. The list of files and test flow described in table 2.

1.3.3 Compare

The Compare is responsible to compare the DUT outputs against the Gold Model outputs. The Compare create a text report that inform the user about the verification tests results. In case of difference between the golden model and the DUT, the DUT result and the golden model result will write to the report. At the end of each test the hit and miss rate will write to the report.

Figure 3: Compare

Inputs list: operation_done,PRDATA, num_of_errors, data_out, arstn,PENABLE,PWRITE,val.

1.3.4 Coverage

The Coverage collect statistics information about the inputs and output to the DUT and create text report that include all the results. The Coverage is not a must in our verification architecture because we generate all the inputs and outputs to the DUT with python, So we know exactly what is the coverage of each signal. However, for the completely of the assignment We collect statistics information on the following signals: system reset, PENABLE, operation_done, data_out (the DUT results), num_of_errors, data_in (the input to the DUT),PADDR,CTRL,CodeWord_Width.

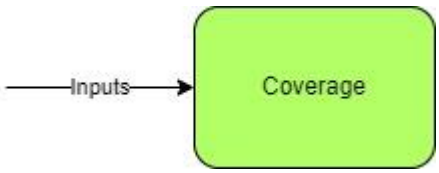

Figure 4: Coverage

Inputs list: PADDR, PENABLE, PSEL, PWDATA, PWRITE, clk, arstn, PRDATA, data_out, operation_done, num_of_errors, operation.

1.3.5 Checker

The Checker asserts some properties in order to make sure that the design work properly. The Checker monitor the asserted properties and in case of valuation of a property rise an error.

For example, We check the if the operation done rise when it should.

Figure 5: Checker

Inputs list: PADDR, PENABLE, PSEL, PWDATA, PWRITE, clk, arstn, PRDATA, data_out, operation_done, num_of_errors, operation.

1.4 Functional Coverage

Function	Event	Coverage Point	Bins
Reset	negedge arstn	arstn	0,1
enable	posedge clock	PENABLE	0,1
operation_done	posedge clock	operation_done	0,1
data_out	posedge operation_done	data_out	0:15, 16:(2**8-1), (2**8):(2**11-1), (2**11):(2**16-1), (2**16):(2**26-1), (2**26):(2**32-1)
num_of_errors	posedge operation_done	num_of_errors	0,1,2,default
data_in	negedge PSEL && PADDR==4	PWDATA	0:15, 16:(2**8-1), (2**8):(2**11-1), (2**11):(2**16-1), (2**16):(2**26-1), (2**26):(2**32-1)
adress	PADDR	PADDR	0,4,8,12,default
ctrl	negedge PSEL && PADDR==0	PWDATA	0,1,2,3
codeword_width	negedge PSEL && PADDR==8	PWDATA	0,1,2,3

Table 3: Test Plan Functional Coverage

1.5 Test Bench Functional Checkers

Condition	Event	Expected Result
op_done_timing	posedge clk	((checker_bus.PENABLE==1) && (checker_bus.PWRITE==1) && (checker_bus.PADDR==0)) |=> (checker_bus.operation_done==1); // בודק אם לאחר מחזור שעון אחד לאחר שהתבצעה כתיבה לCTRL עלה הקו op_done
right_adress	Posedge PENABLE	(checker_bus.PADDR==0) || (checker_bus.PADDR==4)||(checker_bus.PADDR==8) || (checker_bus.PADDR==12) // בודק האם כאשר מתבצעת כתיבה לרגיסטר כלשהו תחום הכתובות הינו 0,4,8 או 12.
rst_active	clk	arstn==0 |=> ((checker_bus.data_out == 0) && (checker_bus.num_of_errors == 0)) //בודק האם מחזור שעון לאחר שיש ריסט המוצאים מתאפסים
proper_noise	PENABLE	((checker_bus.PENABLE==1) && (checker_bus.PADDR==12)) |-> ($countones(checker_bus.PWRITE)<3); //בודק האם הרגיסטר רעש מכיל לכל היותר 2 אחדים

Table 4: Test Plan Functional Coverage

1.6 Golden Model

As describe above we used Python to generate text files which contain inputs and outputs. In case of input file, each line contain a legal input. In case of output file, each line contain the correct output, if the operation is Decode or Full channel then then in each line there is a output space and num of errors.

For example:

Line from input file: 00000000000000100011000000000001101

Line from output file of Encode: 00000000000000000011110110

Line from output file of Decode or full channel: 00000000000000000011110110 2

In the last example there is 2 errors.

Since there are reasonable amount of inputs in 8 and 16 bits, we generated all possible inputs. For the 8 bits there are 16 (because there are 4 bits of data - 2^4) possible legal inputs. Similarly, For 16 bits there are 2048 (2^11) possible legal inputs. In 32 bits option there are 26 bits of data, so there are 2^26 possible inputs. That amount of inputs is too large to cover, so we generated inputs according the following pattern: we generate inputs in interval of 256 and in each interval we randomized an additional input. For instance, the first input is zeros inputs (26 bits of 0), the second input is a random number between 1-255 (in binary), the third input is 256 (in binary) the next input is random between 256-511 (in binary) and so on. In that pattern we cover 0.78% (total of 524,288 inputs) of all possible inputs. In addition, we tested the read operation from the all registers.

2. APPENDIX

2.1 Terminology

LSB - Least Significant Bit

TBR - To Be Reviewed

TBD - To Be Defined

IF - Inteface

2.2 Verification results

Functional Coverage report:

The Coverage is not a must in our verification architecture because we generate all the inputs and outputs to the DUT with python, So we know exactly what is the coverage of each signal.

Since there are reasonable amount of inputs in 8 and 16 bits, we generated all possible inputs. For the 8 bits there are 16 (because there are 4 bits of data - 2^4) possible legal inputs. Similarly, For 16 bits there are 2048 (2^11) possible legal inputs. In 32 bits option there are 26 bits of data, so there are 2^26 possible inputs. That amount of inputs is too large to cover, so we generated inputs according the following pattern: we generate inputs in interval of 256 and in each interval we randomized an additional input. For instance, the first input is zeros inputs (26 bits of 0), the second input is a random number between 1-255 (in binary), the third input is 256 (in binary) the next input is random between 256-511 (in binary) and so on. In that pattern we cover 0.78% (total of 524,288 inputs) of all possible inputs. In addition, we tested the read operation from the all registers.

To noise register we generated file which input all the different combinations: $\binom{32}{2} + 32 + 1 = 529$ options. So the coverage collecting in the noise signals is not necessary.

<u>The full functional coverage report Attached in hdl->verification files->fcover_report</u>

Name	Class Type	Coverage	Goal	% of Goal	Status	Included	Merge_instances	Get_inst_coverage	Comment
/tb_overall/cov		92.85%							
TYPE reset		100.00%	100	100.00...		✓	auto(0)		
TYPE regular_...		100.00%	100	100.00...		✓	auto(0)		
TYPE results		100.00%	100	100.00...		✓	auto(0)		
TYPE data_in		100.00%	100	100.00...		✓	auto(0)		
TYPE adress		100.00%	100	100.00...		✓	auto(0)		
TYPE ctrl		75.00%	100	75.00%		✓	auto(0)		
CVP ctrl::ct...		75.00%	100	75.00%		✓			
INST \/tb_...		75.00%	100	75.00%		✓			0
CVP ctr...		75.00%	100	75.00%		✓			
bin ...		526354	1	100.00...		✓			
bin ...		526354	1	100.00...		✓			
bin t...		526354	1	100.00...		✓			
bin t...		0	1	0.00%		✓			
TYPE codewor...		75.00%	100	75.00%		✓	auto(0)		
CVP codew...		75.00%	100	75.00%		✓			
INST \/tb_...		75.00%	100	75.00%		✓			0
CVP co...		75.00%	100	75.00%		✓			
bin ...		18	1	100.00...		✓			
bin ...		18	1	100.00...		✓			
bin t...		17	1	100.00...		✓			
bin t...		0	1	0.00%		✓			

Functional checker report:

Name	Language	Enabled	Log	Count	AtLeast	Limit	Weight	Cmplt %	Cmplt graph	Includ
/tb_overall/chec/cover__proper_noise	SVA	✓	Off	528352	1	Unli...	1	100%		✓
/tb_overall/chec/cover__rst_active	SVA	✓	Off	18	1	Unli...	1	100%		✓
/tb_overall/chec/cover__right_adress	SVA	✓	Off	3688485	1	Unli...	1	100%		✓
/tb_overall/chec/cover__op_done_timing	SVA	✓	Off	1579059	1	Unli...	1	100%		✓

Code coverage report:

```
TOTAL COVERGROUP COVERAGE: 92.85%  COVERGROUP TYPES: 7

TOTAL DIRECTIVE COVERAGE: 100.00%  COVERS: 4

TOTAL ASSERTION COVERAGE: 100.00%  ASSERTIONS: 4
```

Golden model comparison Report:

```
Encode 8-bits
hits:          16 out of          16
miss:           0 out of          16
Encode 16-bits
hits:        2048 out of        2048
miss:           0 out of        2048
Encode 32_bits
hits:      524288 out of      524288
miss:           0 out of      524288
Decode 8-bits
hits:          16 out of          16
miss:           0 out of          16
Decode 16-bits
hits:        2048 out of        2048
miss:           0 out of        2048
Decode 32_bits
hits:      524288 out of      524288
miss:           0 out of      524288
Full Channel 8-bits
hits:          16 out of          16
miss:           0 out of          16
Full Channel 16-bits
hits:        2048 out of        2048
miss:           0 out of        2048
Full Channel 32_bits
hits:      524288 out of      524288
miss:           0 out of      524288
Read from CTRL register:
hits:           3 out of           3
miss:           0 out of           3
Read from DATA register:
hits:        1000 out of        1000
miss:           0 out of        1000
Read from CodeWord_Width register:
hits:           3 out of           3
miss:           0 out of           3
Read from Noise register:
hits:        1000 out of        1000
miss:           0 out of        1000
```

As we can see there is no errors and we get 100% hit rate.