2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Enhanced LDPC Codec Verification in UVM

Shriharsha Koila, Staff Member Technical, SmartIOPS, Bangalore, India (*shri@smartiops.com*)

Ganesh Shetti , Design and Verification Engineer, SmartIOPS, Bangalore, India (*ganesh@smartiops.com*)

Prateek Jain, Design and Verification Engineer, Yoctozant Technologies Pvt. Ltd, Bangalore, India (*prateek.ext@smartiops.com*)

Anand Shirahatti, Verification quality consultant, VerifSudha Technologies Pvt. Ltd., Bangalore, India (*anand.ext@smartiops.com*)

***Abstract*—Error control coding (ECC) methods are employed in communication and storage systems, to increase the reliability of the data. Low Density Parity Check (LDPC) is one of such methods, robust enough to boost the reliability near to theoretical limits. Unlike the classical cyclic ECC methods, LDPC has a probabilistic error correction behavior. Hence the functional verification of LDPC codec poses challenges in error pattern generation mechanism, checking the correctness of functionality and performance evaluation of the codec. In this paper, we describe the UVM test bench created to verify the LDPC codec. UVM test bench utilizes DPI based scoreboard, elegant error pattern generation mechanism and statistical coverage to further improve the quality of verification. End result of this verification process is the first time operational RTL on the FPGA.**

Keywords—ECC, LDPC, DPI, UVM

I. INTRODUCTION

Low Density Parity Check (LDPC) codes are well known as capacity approaching codes. LDPC codes outperform the other error correcting codes when length of the codeword is in order of thousands of bits [1]. The probabilistic error correction nature of LDPC codes differs from the definitive error correction of block codes, providing a better overall error performance. Hence making them very useful in high bandwidth systems where data reliability is most important. As the name suggests, these codes have very low number of ones in their parity check matrix. Parity check matrix is the representation of check equations. These check equations are used for correction of errors while decoding. Due to the sparseness in parity check matrix, the encoding and decoding process becomes simpler to implement. Further, the complexity can be reduced by way of construction of a parity check matrix. A regular type of parity check matrix has lower complexity in codec design compared to irregular check matrix with a small trade off in error correction. Various construction methods are available in literature for generating a regular parity check matrix [2],[3] which are good in error correction as well as simpler to implement. In this work, we have used a custom designed regular LDPC codec IP for verification.

A complete codec comprises of encoder and decoder pair. The encoder logic is derived from a well-known RU method described in [4]. Decoder is implemented based on message passing algorithm which is iterative in nature. Message passing algorithms works based on the mutual information transfer between check nodes and variable nodes. Variable nodes refer to the actual information bits and check nodes refer to the extra bits or parity added to the information during encoding. Iterative decoding algorithms are mostly used in LDPC codes to get higher error correction. In this algorithm, smaller number of errors gets decoded faster and as the number of errors increase, iterations required to decode also increases. Based on the specified performance designer identifies the maximum iterations required for the decoder.

The complete decoding of LDPC code is done by two methods, namely Hard decoding & Soft decoding [5]. Hard decoder works on the quantized bit values and soft decoder works on the probabilistic information extracted for every bit from the channel. There are three major operations namely Variable node updation (VNU), Check node updation (CNU) and Syndrome computation (SC) in decoding. VNU works on the information passed from all the check nodes connected to it and similarly CNU uses the information passed from corresponding variable nodes. SC is used to check whether the decoder has reached the solution.

In this paper, we will describe the challenges faced during the verification of the codec and techniques used in the complete verification. Detailed information on design and implementation of the LDPC codec are omitted as they are out of the scope of this paper. The next section briefly describes the algorithm level verification which is used to prove the correctness of encoding/decoding functionality. The actual Verilog based design is verified using the UVM test bench which is described in the later section. .

II. ALGORITHM LEVEL VERIFICATION

The design phase of LDPC involves generation of parity check matrix and selection of algorithms for encoding and decoding. The complete simulation model is developed in C++ as a system, including random data generation, encoding, channel error model and decoding functions. Moreover, the core LDPC model is developed as a generic encoding and decoding algorithm, wherein parity check matrix is taken as an input.

Simulations were performed to check the error correction capability of the LDPC code. The parity check matrix is then re-generated multiple times until the error correction requirement is fulfilled. Once the algorithm is proved and error correction requirements were met, the Verilog based LDPC codec is created which is the Design Under Test (DUT) for our UVM testbench.

C++ model is used for generation of reference data in the UVM test bench scoreboard. C++ model also supports debug capabilities, which will provide the necessary information after the end of every iteration. This information is utilized in the UVM test bench debug.

III. DESIGN UNDER TEST (DUT)

The LDPC IP or the DUT has four custom designed interfaces. Input data at encoder is qualified by start of packet, end of packet and valid signals. Output of encoder has a valid signal to indicate the availability of parity bits after the input data is completely fed to the encoder. Parity will be available after a fixed delay and at the same time new packet can be fed to the encoder. The internal logic is completely pipelined to maximize the throughput.

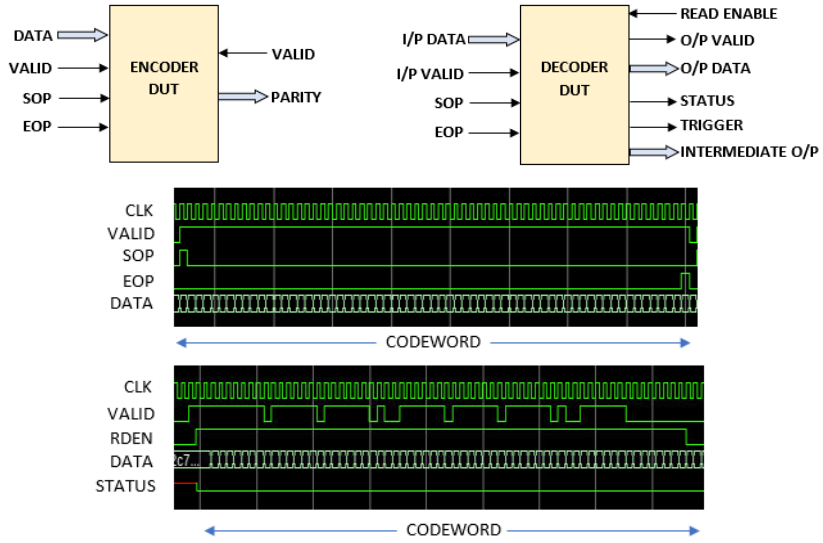

Figure 1: I/O Ports of design under test(DUT) and interface waveforms

Similar to encoder's input, the reception of codeword at decoder also follows the same convention. The start of codeword and end of codeword are indicated by two separate signals, along with valid signal. There are two data buffers inside the decoder, one to store input data and another to store output data. These storage spaces hold more than one codeword in the input side and decoded packets at the output side. When the decoding is done, it is indicated by a O/P valid signal and decoded data will be available at the output data buffer. Memory FIFO buffers are used in the design to manage the variable latency in the decoder because of iterative algorithm. The UVM test bench for unit level is described in the Figure.2.

IV. UVM TEST BENCH OVERVIEW

The architecture and development of complete testbench is driven by many aspects such as the errors injection mechanism, debuggability for iterative decoding algorithm and reusability requirement from unit to sub-system. The DUT targeted in our verification task is custom designed for very high error correction. Hence the error injection task becomes vital in proving the core logic of the codec.

The complete verification of DUT is done at both unit and sub-system level. Each of the above mentioned aspects are described in detail in the following sections.

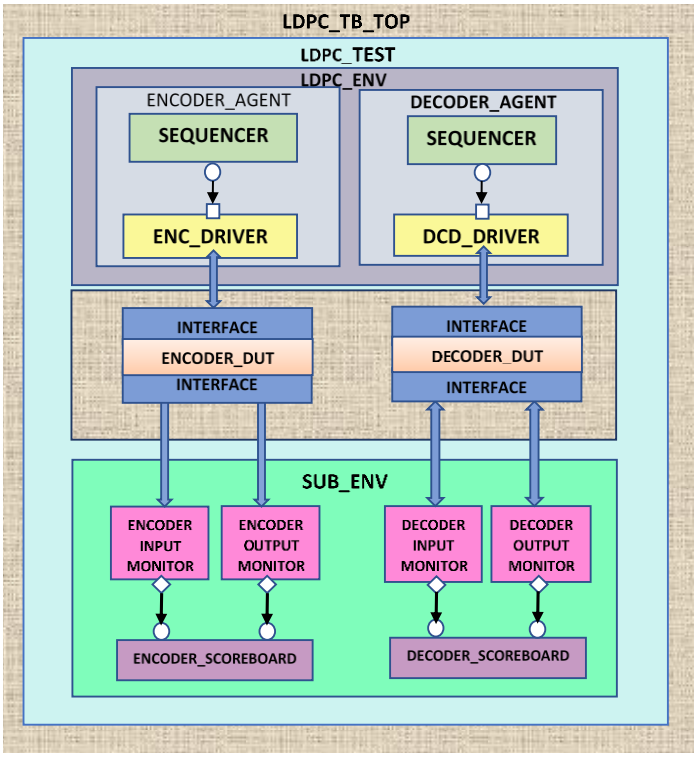

Figure 2: Unit level UVM test bench

A. Unit level test bench flow

Figure.2 shows the block diagram of unit level test bench. LDPC_TB_TOP includes LDPC_TEST and DESIGN UNDER TEST (DUT). LDPC_TEST contains LDPC_ENV and SUB_ENV classes both extended from UVM environment class. LDPC_ENV drives signals to DUT. SUB_ENV monitors and injects error to DUT. SUB_ENV is created to enable the reuse of monitors and scoreboards at the sub-system level. It will be described further in the portability section V.

LDPC_ENV contains two UVM agents, one for encoder and decoder respectively. These agents include a sequencer and a driver. Sequencer passes the sequence_item created by sequence to the driver. Sequence produces programmed number of sequence_item containing codeword and length of the codeword. Codeword is the data on which the DUT operate. The length is typically fixed for given DUT. Driver receives sequence_item from sequencer and drives it out on the signals to DUT following the rules of the interface. DCD_DRIVER also supports the transaction based error injection described in detail in section V.(D).

SUB_ENV contains input monitor, output monitor and a scoreboard each for encoder and decoder. Input monitor monitors data on DUT's input interface and passes the packet issued on DUT's interface to ENCODER_SCOREBOARD through analysis port. ENCODER OUTPUT MONITOR monitors the parity from

encoder DUT's output interface and passes it to ENCODER_SCOREBOARD through analysis port. DECODER INPUT MONITOR receives data from input interface of the decoder. Here the live errors are injected to the codeword and is forced back to the input interface. It also passes data to the DECODER_SCOREBOARD through analysis port. DECODER OUTPUT MONITOR extracts the decoded data whenever its ready from the decoder and sends it to DECODER_SCOREBOARD via analysis port. Input to the decoder is the encoded data concatenated with its appropriate parity. Since the unit level testbench has no access to the actual channel model (flash memory model in our application), a channeling process is created wherein data and parity received from ENCODER OUTPUT MONITOR is stored in a queue. Data and parity together form a complete codeword. This codeword represents sequence_item for decoder and is processed further as mentioned above.

Separate scoreboards for encoder and decoder are shown in Figure.3, which receives the output from encoder and decoder respectively and compares them with the expected outputs. Expected outputs are taken from C++ models which are plugged into the Testbench through Direct Programming Interface (DPI).

B. C++ MODEL INTEGRATION THROUGH DPI

SystemVerilog (SV) DPI is basically an interface between SV and the foreign programming language which, in this project is C++ language. SV simulator typically uses GNU compiler collection (GCC) internally to compile .cpp files. C++ model used in this project utilizes some features from c++11 standards. This requires passing additional GCC compiler flags to the simulator. For example, Questasim provides a flag "-ccflags -std=c++11" to pass this additional argument to GCC.

SV uses import "DPI-C" declaration to implement C++ functions. C++ function takes in the "char" pointers for both the input and output parameters. When SV interacts with C/C++ model, data exchange happens by exchanging "char" pointer with byte data type from SV [7]. C++ model represents every bit of codeword as one character data type. SV represents the codeword as a bit vector. Therefore a mechanism is introduced where bits are converted into bytes while passing to C++ model and vice-versa while getting from C++ model.

C. SCOREBOARD

Figure.3(a) describes encoder scoreboard in detail. Through analysis port from ENCODER INPUT MONITOR, codeword is received and passed to the encoder C++ model after a bit to byte conversion. Encoder C++ model processes the codeword and provides appropriate parity output to the scoreboard which is in byte format. A reference parity is generated after byte to bit conversion. ENCODER OUTPUT MONITOR then passes DUT's parity through analysis port. Reference and the actual parity are compared and error message is flagged when they mismatch.

Figure.3(b) describes decoder scoreboard in detail. Decoder scoreboard operates in two modes, normal and debug mode. In both the modes, it performs three common checks. These three checks are number of iterations, corrected codeword and correction status. In debug mode, it additionally compares the intermediate outputs at the end of each iteration.

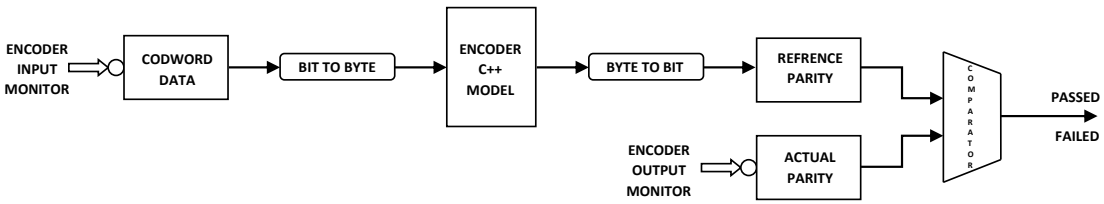

Figure 3(a): LDPC encoder scoreboard

A corrupted codeword received from DECODER INPUT MONITOR is sent to decoder C++ model similar to the encoder scoreboard. In normal mode, decoder C++ model iterates up to max iteration or until status=0. After end of iteration, C++ model provides reference data for the common checks.

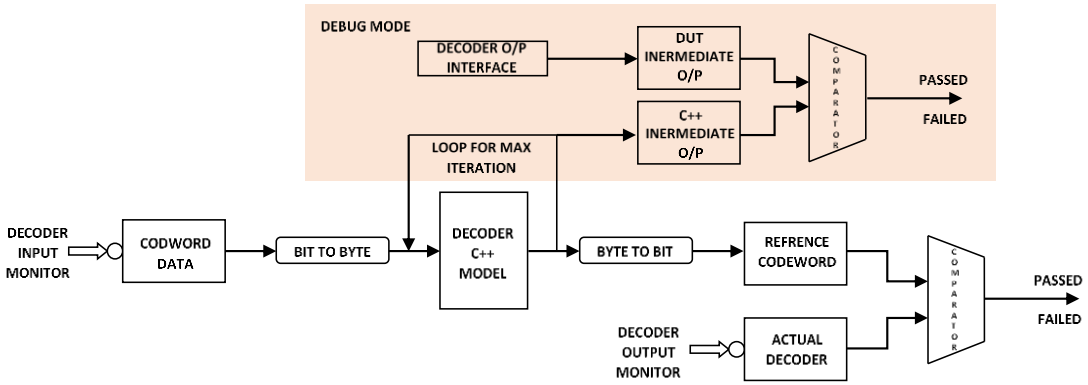

Figure 3(b): LDPC decoder scoreboard

In DEBUG MODE, a trigger from decoder DUT initiate the iteration in C++ Model. After each iteration, an intermediate output from C++ model is compared with that of DUT. Trigger and intermediate signals are received through decoder output interface. At the end of all iteration, the reference codeword will be compared with the DUT's codeword only if the status is set to zero. However, status and iteration count comparisons are done in both the modes to validate its correctness.

V. ERROR INJECTION

Error injection mechanism is one of the prominent tasks in ECC verification. The IP targeted for verification in our work is a binary LDPC codec. In this context, error injection implies simply a flip in the bit. The codelength supported in the current IP is bytes . Though it is possible to generate all error patterns for smaller ECC, it is nearly impossible to get all error patterns when the codeword-lengths are in order of thousands. The total possible error patterns up to 'x' errors in a codeword of length 'n' can be written as,

$$\sum_{t=1}^{x} {}^{n}_{t}C \qquad (1)$$

From (1), we can calculate the total error patterns up-to 5 bits of error in a codeword of 1KB as ~$3x10^{17}$. If the codec can correct greater than 50 bits of error, then the total error patterns reach beyond 10^{131}. It is impossible to generate all possible test vectors for such high sizes. Hence, instead of generating all error patterns, error injection mechanism should be designed to cover various types of error patterns to verify the decoding logic and get enough confidence on the functionality of the core logic.

A. Error injection types & configurations:

Major error injection types supported in the current verification environment are listed below.

1. **RANDOM**:
   The bitflip locations are generated randomly.

2. **Channel Model:**
   For generating error based on Additive White Gaussian Noise (AWGN) channel, we have used the standard library function. DPI can be used to plug-in other channel models as required by the application.

3. **BURST**:
   Two types of Burst errors are defined,
   a. **Single burst: C**orrupts bits continuously once in the packet. Minimum burst length considered here is 2 and maximum burst length is10.
   b. **Multiple bursts:** Generates multiple burst patterns per packet.

4. **BIASED:**
   Two types of biased error are defined,
   a. **ONE to ZERO**: Corrupts only bits which are 1 i.e. it can flip any bit which is 1 to 0.
   b. **ZERO to ONE**: Corrupts only bits which are 0 i.e. it can flip any bit which is 0 to 1.

In the current test bench, error injection is cleanly separated out as the error configuration object. It is divided into multiple objects and post randomization is used to create all the random errors. Two more additional configurations are provided such as 'error percentage' and 'error window selection' to facilitate the creation of various error cases.

Error percentage is required to tune the error injection. For example, when error percentage is given as 80%, it will corrupt 8 out of 10 codewords. Error window selection enables the error injection in a specific range of bit positions within the codeword. Specific window selection for error injection is needed to verify the error correction when size of trapping set and absorbing sets in LDPC code are known [6]. Command line controls are also provided to decide the error type, number of errors, window size and error percentage. Hence, creating the dedicated test-cases become easy especially during the initial flow flush. Above error types, along with the error percentage & window selection, enables the test bench to create error scenarios capable of providing the required coverage.

B. Error Configuration Class

Error configuration contains properties and methods to control the error injection. Some of the key properties and methods are described below.

- ERROR_TYPE is randomized to one of the RANDOM, BIASED, BURST and CHANNEL MODEL. The functionalities of each these type are already described above.
- ERROR_NUMBER is randomized to one of the LESS_THAN_30, BETWEEN_30_TO_60, GREATER_THAN_60. This enables control on the range of number of errors injected.
- NUMBER_OF_ERRORS is randomized to one of the value in the ERROR_NUMBER range selected.
- WINDOW_SIZE is randomized to width less than or equal to the codeword length where the errors are to be injected.
- WINDOW_START and WINDOW_END are randomized to start and end bit positions within the codeword length satisfying the WINDOW_SIZE where the errors are going to be injected.
- <ERROR_TYPE>_POSSIBLE_ERROR_POISITION_Q is System Verilog integer queue which is randomized to give all the possible bit positions which can be corrupted in a codeword satisfying WINDOW_START and WINDOW_END for each ERROR_TYPE. These queues are populated in the post randomization.
- SELECTED_ERROR_POISITION_Q is a single System Verilog integer queue containing the bit positions of the codeword from the POSSIBLE_ERROR_POISITION_Q selected for corruption. This queue is updated in the post randomization.

Post randomization process is described below for generating SELECTED_ERROR_POISITION_Q and <ERROR_TYPE>_POSSIBLE_ERROR_POISITION_Q. The basic idea is for any given type of error, all possible error positions within the window are created in the <ERROR_TYPE>_POSSIBLE_ERROR_POISITION_Q. Based on the number of errors and error type, the sub set of the bit positions are selected and pushed in to SELECTED_ERROR_POISITION_Q. These bit positions are flipped in the codeword before sending it to the LDPC decoder.

The details of the process of selection from bit positions for corruptions for each error type is described below:

1. **RANDOM:** All the bit positions from WINDOW_START to WINDOW_END are pushed in to the RANDOM_POSSIBLE_ERROR_POISITION_Q. After that it's shuffled. After shuffling the first "NUMBER_OF_ERRORS" bit position entries from RANDOM_POSSIBLE_ERROR_POISITION_Q are selected for corruption and copied over to the SELECTED_ERROR_POISITION_Q.
2. **CHANNEL MODEL:** Uses the channel model DPI calls to get the total number of errors and error positions from the C++ code. The total number of errors are copied over to NUMBER_OF_ERRORS and error positions are copied over to the SELECTED_ERROR_POISITION_Q.
3. **BURST:** Single burst related randomizations are managed within the error configuration. Whereas for the multiple bursts related randomizations are managed through an additional helper class.
    a. **Single burst:** Two more controls are provided to select the BURST_START and BURST_SIZE. BURST_SIZE is randomized to value between 2 to 10. BURST_START is randomized between the WINDOW_START and WINDOW_END satisfying the BURST_SIZE and ensuring that it does not cross the selected window. The bit positions from starting from BURST_START to BURST_START + BURST_SIZE are copied over to the SELECTED_ERROR_POISITION_Q.

b. **Multiple bursts:** Total number bursts is randomized between the maximum number of bursts and minimum number of bursts possible. Maximum and minimum number of bursts possible is dependent on NUMBER_OF_ERRORS **,** minimum burst size(2) and maximum burst size(10)**.** Based on total number of bursts and NUMBER_OF_ERRORS size of each of burst is determined. For each burst the start and end are allocated such that they get maximum possible spread over the window size. Bit positions between each burst start and end are copied over to the SELECTED_ERROR_POISITION_Q.
As this involves many randomizations and allocations, its managed in separate class called multiple_burst_helper. Its instanced within the main error configuration class and gets invoked inside the post randomization of the error configuration class.

4. **BIASED:** All the bit positions in the codeword which are either 1 (ONE to ZERO) or 0 (ZERO to ONE) within the WINDOW_START to WINDOW_END are pushed in to the BIASED_POSSIBLE_ERROR_POISITION_Q. After that it's shuffled. After shuffling the first "NUMBER_OF_ERRORS" bit position entries from BIASED _POSSIBLE_ERROR_POISITION_Q are selected for corruption and copied over to the SELECTED_ERROR_POISITION_Q.

C. Observations:

1. While randomizing the error configuration class, complex inter dependencies between the variables were resolved using the SOLVE BEFORE construct as shown in the Figure.4a.

```
constraint solve_before{
    solve WINDOW_START before WINDOW_END;
    solve WINDOW_SIZE before WINDOW_END;
    solve WINDOW_SIZE before NUMBER_OF_ERRORS;
    solve ERROR_NUMBER before NUMBER_OF_ERRORS;
    solve ERROR_TYPE before NUMBER_OF_ERRORS;
    solve ERROR_TYPE before error;
    solve WINDOW_SIZE before WINDOW_START;
}
```

Figure 4(a): LDPC decoder scoreboard

2. In order to provide the right distribution of errors, "dist" operator has been used as shown in the Figure.4(b).

```
constraint WINDOW_START_c{
    if (WINDOW_SIZE == 360)
        WINDOW_START dist {[0:999]:/10, [1000:1999]:/10, ..... ,[4000:4999]:/10,
                           ..... , [`CODEWORD_WIDTH-1000:`CODEWORD_WIDTH]:/10};
}
```

Figure 4(b): LDPC decoder scoreboard

3. It has been observed that constraints fall short for some problems where there is dynamic random selections. Such cases can be dealt with overriding constraints behavior with the behavioral code in the post randomization. Some problems need constraints support in post randomizations as well. One such case in the error configuration class was generating the multiple burst errors. Multiple burst error requires multiple randomizations interleaved with behavioral code. Such complex randomizations can be handled better with the post randomizations coupled with class compositions.

D. Error configuration usage

Error configuration class is reused in both the unit and sub-system level. Unit level relies on the transaction based error injection whereas the sub-system level uses the live error injection.

Transaction based error injection

Transaction based error injection waits for the complete codeword to be available. Based on the data in the codeword the error injection takes place completely in the zero time. Then the corrupted transaction is driven to the design.
In the test bench before driving the new codeword to the decoder the error configuration is randomized passing the codeword as input. The randomized error configuration object is stored inside the codeword transaction after corrupting the data. The corrupted codeword is driven out by the driver. The decision for corrupting the codeword is based on either the error injection percentage or the directed error injection handle stored in the driver through test.

Live error injection

Live error injection does not wait for the complete codeword to be available. A codeword is driven to DUT over multiple clock cycles depending on the width of the input data bus. Corruption is expected to take place on the same clock while its being driven to decoder inputs.

The start of frame signal is asserted with valid signal going from low to high. At the rising edge of the valid signal, decision is made whether to corrupt the codeword based error percentage given during the run-time or the directed error configuration set by the test. If the codeword is supposed to be corrupted then the randomization of the error configuration class takes place.

Subsequently on every rising edge of the clock the data is being driven out. In the same clock cycle's falling edge, correct data is sampled and the check is made if the position of the data matches with the selected error data bit positions. If there is a match, the corruption will take place in the respective positions which will be captured by the decoder in the next rising edge. The correct and corrupted data are stored in separate queues. This process will continue till end of the codeword comes in. At the end of codeword, both the correct and corrupted data will be exported to the scoreboard for the data integrity checks.

Why live error injection over transaction based error injection in sub-system level?

Transaction based error injection utilizes the complete data present in the codeword. For example, in biased error injection the error configurations looks for the posititions of the bit value ZERO and ONE in the codeword.
In sub-system collecting the complete codeword, corrupting and replaying it back would cause a delay of one codeword. This behavior was not acceptable for the susb-system verification. Hence the live error injection mechanism was built and used. Sub-system does not support the biased error injection capabilities.

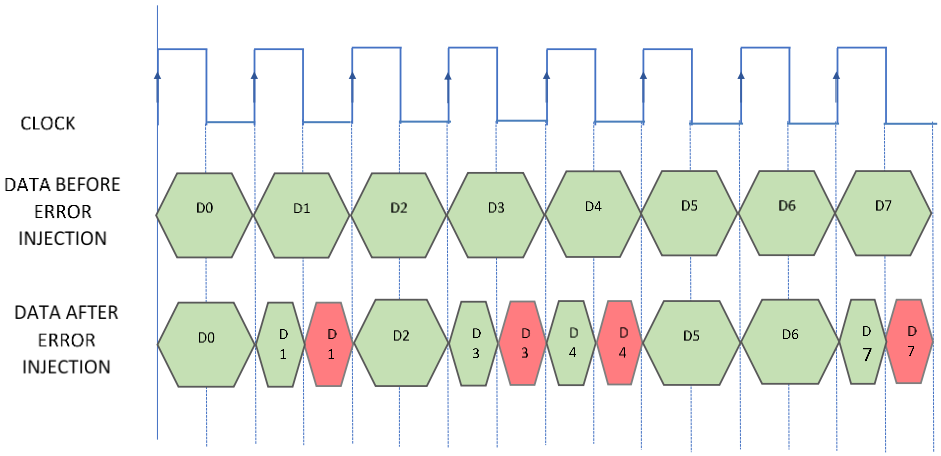

Figure 5: Live error injection waveforms showing sampling and corruption in same clock cycle

E. Channel model

Another challenge was to integrate the channel model for error injection. For implementing AWGN channel-like error injection, we have used dist_normal function and statistics are provided based on the channel defined for the application of IP. Using DPI, we have integrated the C-model and extracted the error locations for other channel models. Figure.6 shows the histogram of the "number of errors" injected as per uniform and normal distribution functions. The plots are generated for errors ranging from 1 to 60 with the help of standard UVM library functions.

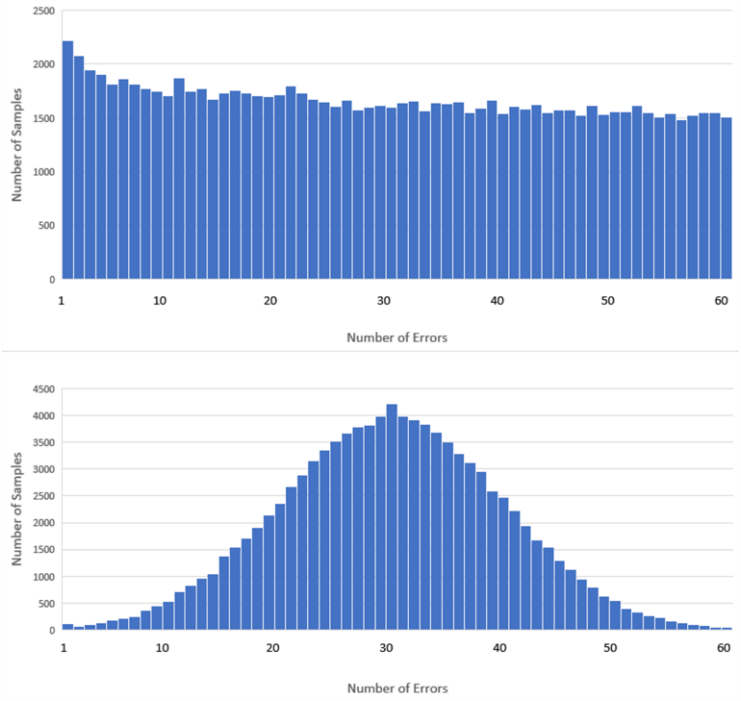

Figure 6: Histogram of number of errors a) Uniform b) Normal

F. Directed vs Random Error Injection

Random error injection can generate all possible combinations of available error types and window selection. During the regression, random seeds are used for each test thereby increasing the quality of testing.

Directed error injection case have been written to cover the corner scenarios like corrupting the first and the last bit of the codeword, corrupting alternate codewords, introducing uncorrectable error patterns in all codewords etc. The specific cases are constructed based on two aspects. One is for verification of core logic and another for checking behavior of pipelined architecture of complete codec. Therefore, all the dedicated testcases help to visualize the critical interaction of algorithmic level logic and control logic for the codec.

VI. PORTABILITY

Sub-system DUT is as shown in the Figure.7 below. SUB DUT 1,2,3 & 4 depicts the adjacent modules connected to the LDPC IP in full system level. SUB DUT-1 is connected to the input of the encoder and responsible for the data transfer into the encoder. SUB DUT-2 extracts the data and parity from encoder and pushes the codeword into the channel. SUB DUT-3 gets the codeword from the channel and pushes into the decoder and SUB DUT-4 extracts the decoded data.

Unit to system portability is done by porting SUB_ENV from unit level, to that of sub-system level. SUB_ENV is registered under sub-system level test bench's test. Interfaces of SUB_ENV are connected to appropriate ports

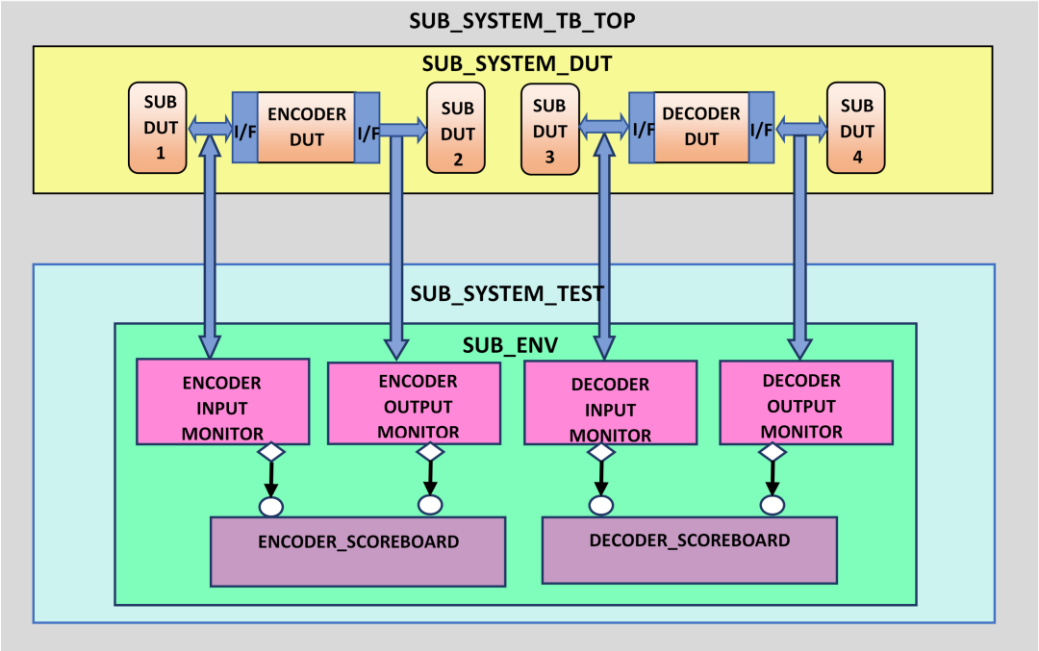

Figure 7: Full system Test bench block diagram

to receive and drive the signals. For error injection, SUB_ENV receives the codeword, corrupts it and force it back to DUT ports as described in the live error injection.

VII. STATISTICAL AND FUNCTIONAL COVERAGE

Code coverage quickly crossed greater than 90% with the seeded regressions. Along with the statistical coverage both the Whitebox and black box functional coverage was also added.

Statistical coverage or analytics has given another important dimension to the verification metrics. It helps in two different ways. First functional coverage indicates if some value or scenario has happened and numbers of times it has happened. But whether its duration has been sufficient or its relative distribution compared to other features, whether such distributions are in alignment to project priorities cannot be inferred from functional coverage alone. Second is designer has certain intent while creating the design. Due to limited traffic used in the simulation there is possibility that design may appear operational even when it's not completely in-line with the intent of designer. Some cases this deviation from expectation may take large number of cycles to show up as functional failures visible to test bench and hence may get masked. Both are addressed with the statistical coverage.

For decoder outstanding codeword count functional coverage would tell us if all the values are covered and how many times they have been hit. But it will not tell us about how long design stayed in each of these utilization level and their relative distributions. For LDPC decoder the outstanding buffer utilization percentage duration for various depths was measured. Figure.8 shows the data about one such measurement early during the verification cycle.

WBV_NAME	WBV_TYPE	Valu	Value duration(	Total sim time(f	Percenta
l_dec_os_codeword_utilization	WBV_DURATION_MONITOR	0	85770017214	3.71E+11	23.13
l_dec_os_codeword_utilization	WBV_DURATION_MONITOR	1	1.98E+11	3.71E+11	53.53
l_dec_os_codeword_utilization	WBV_DURATION_MONITOR	2	61643345662	3.71E+11	16.62
l_dec_os_codeword_utilization	WBV_DURATION_MONITOR	3	19506670568	3.71E+11	5.26
l_dec_os_codeword_utilization	WBV_DURATION_MONITOR	4	4806667628	3.71E+11	1.3
l_dec_os_codeword_utilization	WBV_DURATION_MONITOR	5	286666724	3.71E+11	0.08

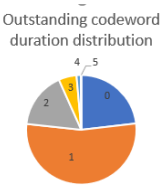

Figure 8: Statistical coverage of duration outstanding codeword depth

Their relative distribution helped us in creating additional scenarios to keep design in higher side of the outstanding buffer utilization longer to stress it sufficiently.

For LDPC decoder especially various statistical measurements were done on the micro-architecture to make sure its operation is in-line with the intent of designer. Some of the examples are, the arbiter receiving the read and write requests in specific ratio desired. Multiple Block RAM instances were used for the parallel access of the data. The percentage duration of read, write and read/write together utilization for all the memory banks was expected to be closely matching in the same range. Minimum and maximum delays between output back pressure duration was expected to hit certain limits. All these statistical measurements were done for less than X errors, between to X to Y errors and greater than Y errors.

Black box functional coverage included, all the errors types, number of errors injected, whether each and every bit position is corrupted, burst lengths, number of bursts, various window lengths and positions where the corruption was localized.

White box functional coverage included inter-packet delay between the packets at the input of encoder and decoder, concurrency between input and output, decoder input and output back pressure durations, decoder enable and disable with traffic. One interesting cross added was the error combinations for the pipeline of the decoder to make sure all combination of the error types are experienced by the design. This cross was showing lot of holes where the mix of no error injection and different error types. A directed test case was added to control the error injected for the pipeline depth number of packets to achieve the desired mix.

Throughput measurements were done both for the encoder and decoder. Decoder throughput measurements were done for less than X errors, between to X to Y errors and greater than Y errors

VIII. RESULTS AND CONCLUSIONS

One of the key result is, the basic functionality of the LDPC encoder and decoder became operational on the FPGA board without any issues and meeting the desired performance. With 20 seeds per tests and total of 300+ command lines both the functional and code coverage targets were met. Regression failures have reached stability.

UVM has proven yet another time to deliver on its promise of reusability and flexibility. Cleanly separating out the error pattern generation functionality allowed good random error injection as well control to do directed error injection, coupled with the code coverage, white box functional coverage, black box functional coverage and statistical coverage metrics have enabled in achieving high degree of functional verification quality[8].

REFERENCES

[1] R. G. Gallager, “Low density parity check codes,” IRE Trans. Inform. Theory, vol. IT-8, no.1, pp. 21–28, Jan. 1962

[2] Marc P. C. Fossorier, Senior Member, IEEE, “Quasi-cyclic low-density parity-check codes from circulant permutation matrices,” IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. 50, NO. 8, AUGUST 2004.

[3] K. Liu et al., "Quasi-cyclic ldpc codes: construction and rank analysis of their parity-check matrices", Information Theory and Applications Workshop (ITA), pp. 227-233, Feb. 2012

[4] Thomas J. Richardson and Rüdiger L. Urbanke, “Efficient encoding of low-density parity-check codes,” IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. 47, NO. 2, FEBRUARY 2001.

[5] Sarah J. Johnson, “Introducing Low-Density Parity-Check Codes,” University of Newcastle, Australia, 2006

[6] Price, Aiden & Hall, Joanne, “A survey on trapping sets and stopping sets,” 2017.

[7] DPI Data mapping: https://www.amiq.com/consulting/2019/01/30/how-to-call-c-functions-from-systemverilog-using-dpi-c/

[8] Functional verification quality : https://www.verifsudha.com/2016/06/22/functional-verification-quality-improvement/