FPGA accelerator of algebraic quasi cyclic LDPC codes for NAND flash memories

FPGA accelerator of Algebraic Quasi Cyclic LDPC Codes for NAND flash memories

Syed Azhar Ali Zaidi, Abuduwaili Tuoheti, Maurizio Martina, *Senior Member, IEEE* and Guido Masera, *Senior Member, IEEE*
Department of Electronics and Telecommunications,
Politecnico di Torino, Italy

***Abstract*—Error correction in high density multilevel cell NAND flash memories is of great concern and Low-Density-Parity-Check (LDPC) codes are attracting much interest due to their Shannon-capacity-approaching behavior. In this work, the error performance of very large block length quasi-cyclic (QC) LDPC codes is evaluated through a high speed FPGA based emulator. A novel algebraic QC-LDPC code of rate 0.96 is also proposed for the 8 KB page size of NAND flash memory and its performance is shown. At a frame error rate (FER) of 10^{-9}, the constructed code achieves a coding gain of 0.15 dB with respect to the previously proposed Euclidean geometry QC-LDPC code and does not suffer from any error floor.**

I. Introduction

The reliability of NAND flash memories has reduced due to continuous technology scaling and the use of multi-level per cell (MLC) approach. Typically, the uncorrectable bit error rate is specified as 10^{-13} to 10^{-16} by the storage device manufacturers [1]. Due to the reduced hardware complexity, hard decision error correcting codes (ECCs) have been widely used in NAND flash memory devices. However, as the raw bit error rate (RBER) is getting worse, more powerful ECCs are required. ECCs with soft decision decoding algorithm show better error correcting performance than hard decision codes [2].

Among the soft decision ECCs, Low-Density-Parity-Check (LDPC) codes provide very good error correcting performance. Recently, many researchers have used LDPC codes for addressing the error correction in NAND flash memory [2], [3]. However, in order to adapt LDPC codes for storage device applications, it is necessary to evaluate their error correcting performance at very low frame error rate (FER). This evaluation requires the acceleration of two key functions: i) the decoding algorithm and ii) a proper channel model. In this paper, we present an FPGA based accelerator dealing with both functions: the decoding part supports very high rate and large block length LDPC codes [4], while the additive white Gaussian noise (AWGN) model has been adopted as a good, low complexity approximation of the channel model.

Currently, NAND flash memories are using page sizes of 4 KB and 8 KB. These page sizes are expected to increase in the next few years, making difficult both the design and implementation of long LDPC codes with high code rate. Kou et al. [5] and Li et al. [6] have given the systematic algebraic construction of LDPC codes. These LDPC codes have high code rate and have good error correcting and error floor performance. However, long LDPC codes require a large amount of resource on FPGA for high throughput implementation. Moreover, the Euclidean Geometry (EG) LDPC codes presented in [5] require a complex switching network in the decoder. Despite of the large number of generalized FPGA based implementations of QC-LDPC codes in the open literature, none of them deals with such QC-LDPC codes. Moreover, to the best of our knowledge, error performance evaluation is not reported for such large page size storage devices. The authors in [7] have given the error performance of randomly constructed QC-LDPC codes up to the maximum block length of 2 KB. In this work, we have constructed and evaluated the performance of regular algebraic QC-LDPC codes for the page size of 8 KB of NAND flash memories.

The contributions of this work are as follows.

- Implementation of a generalized and high throughput FPGA emulator for very high rate and large block length regular algebraic QC-LDPC codes.
- Less hardware resources as compared to the decoder proposed in [4].
- Use of high-level-synthesis to implement a high quality AWGN channel.
- Construction of a novel algebraic QC-LDPC code for 8KB NAND flash memory. The proposed code has less hardware complexity and improved performance as compared with the previously proposed EG-LDPC code [3].

II. LDPC decoding and hardware acceleration

LDPC codes are characterized by a binary parity check matrix $\mathbf{H}$ with M rows and N columns. The $\mathbf{H}$ matrix is sparse and valid codewords x satisfy $\mathbf{H} \cdot x' = 0$, where x' is the transpose of x. In the Tanner graph terminology, columns of $\mathbf{H}$ (associated with bits of x) correspond to variable nodes, and rows (associated with parity equations) correspond to check nodes. Degrees of check and variable nodes are equal to the numbers of ones along rows and columns of $\mathbf{H}$, respectively. In structured LDPC codes, $\mathbf{H}$ is organized with partially regular submatrices, to simplify encoding and decoding procedures. In particular, QC-LDPC codes are a well known class of structured codes, where $\mathbf{H}$ can be represented as an $M_b \times N_b$ array of $z \times z$ circulant permutation submatrices, with $M = M_b \times z$ and $N = N_b \times z$ (z is called the circulant size of

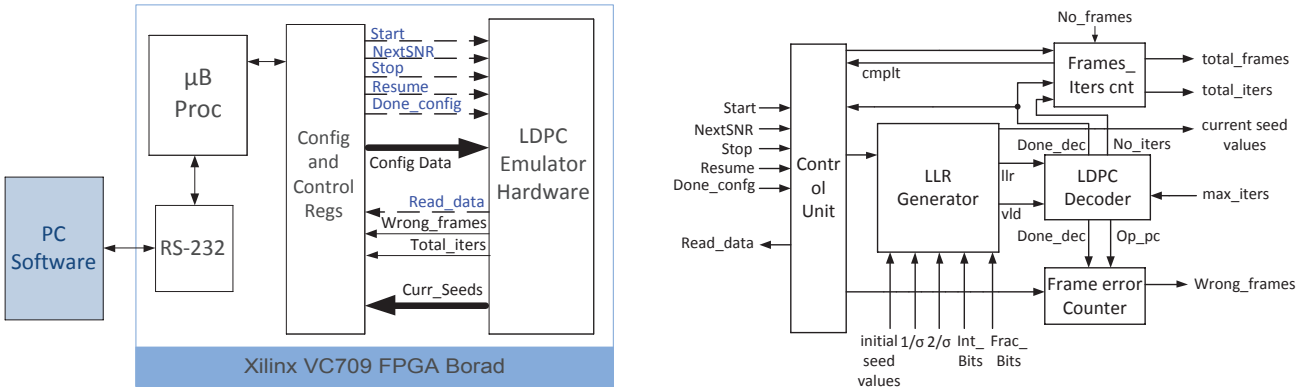

Figure 1: Block Diagram of (a) Complete emulation system and (b) Details of the LDPC emulator hardware.

H). Each submatrix is either a zero matrix or the superposition of w cyclic-shifted identity matrices ($w \geq 1$ is referred to as the circulant weight of the code).

LDPC decoding is usually handled via Belief Propagation algorithm or one of its approximations [8]. The decoding process iteratively updates bit error probabilities (usually represented as Logarithmic Likelihood Ratios or LLRs), which express both the value of codeword bits (sign) and their reliability (magnitude). The decoding algorithm can be seen as the repetitive exchange of messages between variable and check nodes, which leads to the progressive refinement of LLRs towards correct decoded bits.

FPGA based LDPC emulators can be classified based on i) the type of supported LDPC codes (structured, unstructured or both), ii) the architecture of the decoder (serial or parallel), iii) the decoding algorithm and iv) the target application. An FPGA emulator for structured LDPC codes is presented in [9]. The emulator is able to achieve an average throughput of 1.35 Gbps for the (2048,1723) Reed-Solomon LDPC code using a single partially parallel core based on the normalized min-sum algorithm [8]. However, the throughput of the decoder scales down increasing the circulant size.

The authors in [7] and [10] investigated the error floor performance of LDPC codes for the magnetic recording channel. In [7] the authors implemented an FPGA based simulator for hardware-aware and performance oriented QC-LDPC codes. They constructed randomly high rate QC-LDPC codes with maximum circulant and column weight of 2 and 4, respectively. The block lengths of the codes used are from 4608 to 16384 with rates varying from 8/9 to 15/16. The maximum throughput is 360 Mb/s with iterative detection and decoding. In [10], a high throughput emulator is designed resorting to multi core processing. The system occupies three BEE2 boards, each one containing five Xilinx Virtex-II Pro FPGAs, and implements 27 parallel LDPC decoding cores, with code of length 4923 and code rate 8/9. The throughput of each core is 175 Mb/s, achieving a total throughput of 4.725 Gb/s. However, the implementation of such large parallelism is difficult and costly for large block length LDPC codes.

The emulators discussed above have mostly targeted structured LDPC codes as they feature simple encoder and decoder architectures. Indeed, the circulant weight of structured LDPC codes is usually 1. For EG-LDPC codes [5], the circulant weight is rather high and it is difficult to find a conflict free memory mapping for these high-circulant-weight codes. The authors in [11] have proposed a partially parallel decoder architecture for regular QC-LDPC codes. To manage high-circulant-weight matrices they proposed to use one separate memory bank for each cyclically shifted identity matrix with a switching network between the memories and the variable-node and check-node processing units. However, as the circulant weight becomes high, a large number of memory banks are required and the complexity of the switching network increases as well.

III. LDPC Emulator

This section presents the proposed FPGA emulator for regular QC-LDPC codes. The emulator is implemented on a Xilinx VC709 FPGA board, containing a Virtex-7 XC7VX690T FPGA device. The emulator does not require the use of hardware aware QC-LDPC codes and achieves a throughput of more than 1 Gb/s. Moreover, the emulator can also be used to evaluate the performance of very high circulant weight LDPC codes, such as EG-LDPC codes.

A block diagram of the complete emulator system is shown in Fig. 1 (a). The hardware is controlled by a GUI based software running on a PC. The communication between the hardware and the PC relies on the RS-232 port. The Microblaze (μB) soft processor receives the configuration data and the control signals from the PC and sends them to the *Configuration and control registers block*. Similarly, it also receives the data from the *LDPC Emulator Hardware unit*, through the *Configuration and control registers block*, and sends them to the PC. The control signals, shown as dotted lines in Fig. 1

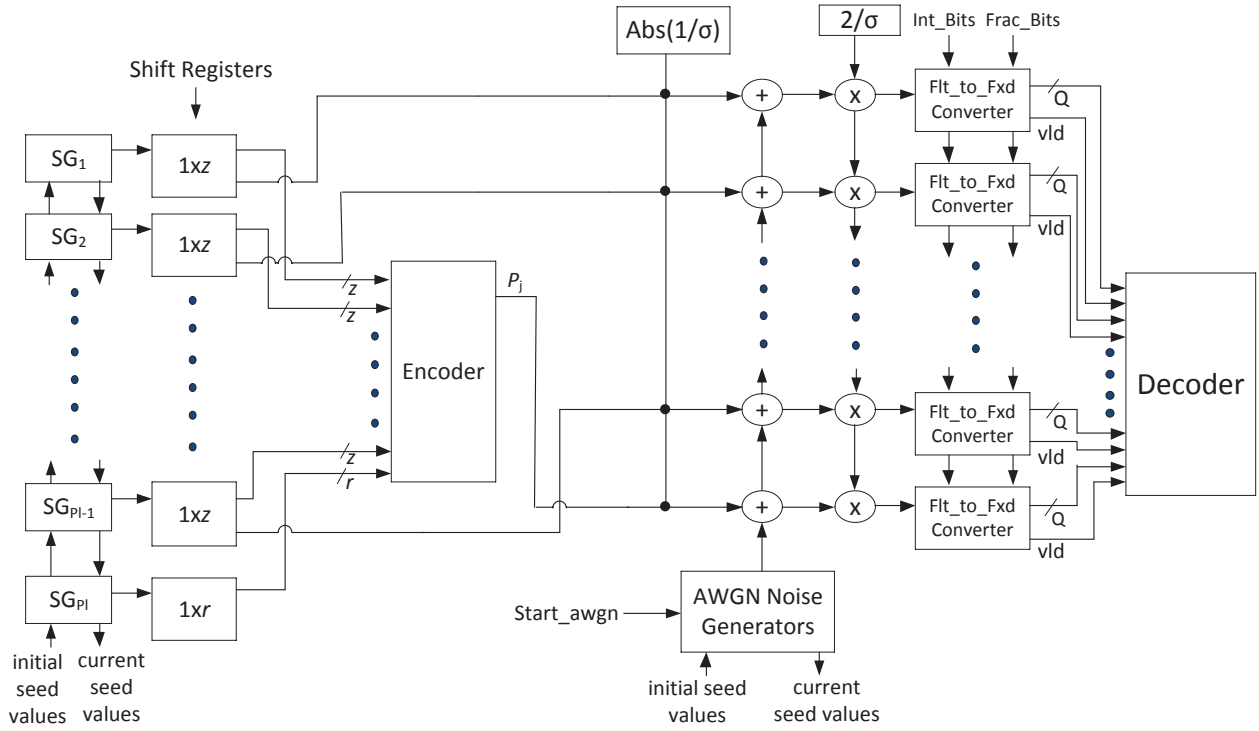

Figure 2: Architecture of the LLR Generator.

(a), include the *Start* and *Stop* signals, for starting and stopping the simulation, *NextSNR* signal for moving to the next signal-to-noise ratio (SNR) point, *Resume* signal for resuming the simulation for a given SNR point and *Done_config* signal, which is asserted when the configuration of the hardware is completed for the current SNR point. The *Read_data* signal is driven by the *LDPC Emulator Hardware* module for reading the number of wrong frames, total iterations of the decoder and the current seed values of the noise and the source bit generators. The *Read_data* signal is activated after a specified number of frames is completed or the maximum number of wrong frames is reached.

The main blocks of the *LDPC Emulator Hardware* are shown in Fig. 1 (b). It consists of the *LLR Generator*, the *LDPC Decoder*, the *Frame error Counter*, the total iterations and total frames counters (*Frames_Iters cnt*) and a control unit. The configuration data for the *LLR Generator* includes i) the initial values of the seed for the noise samples and the source bit generators, ii) the number of integer and fractional bits for the LLRs of the channel symbols, iii) $1/\sigma$ and $2/\sigma$ values, where σ is the standard deviation of the noise at a given SNR point. Similarly, the maximum number of iterations and the number of frames, after which the value of total iterations is read, are applied to the *LDPC Decoder* and *Frames_Iters cnt* modules, respectively. The *total_frames* output is read when the maximum number of wrong frames is reached. The decoder asserts the *Done_dec* signal after decoding completion of each frame and provides i) the number of iterations (*No_iters*) used to complete the decoding and ii) the output of the parity check circuit, which is high when the frame is not decoded correctly. These data are added to the current values of the total iterations and the wrong frames, respectively, by means of two counters.

A. LLR Generator

The *LLR Generator* hardware, shown in Fig. 2, produces the source bits, encodes the information frame, adds the Gaussian noise to model the AWGN channel and generates the LLRs of the received bits. These LLRs are transfered to the decoder. As detailed in Fig. 2, source bit generation is implemented with P_l source generators (SGs), SG_1 to SG_{P_l}, where P_l is the number of LLRs generated in parallel by the LLR generator. For simplicity, the architecture of the LLR generator is shown for the case $P_l = N_b - M_b + 1$ and $M_b = 1$. The first $P_l - 1$ SGs generate z bits, whereas the last SG generates *ldr* bits, where *ldr* is the number of redundant rows (linearly dependent rows) in the parity check matrix. These information bits are stored in the corresponding shift registers attached to the source generators.

After generating the information frame, the whole frame is encoded. The encoder takes z clock cycles to encode one frame and produces M_b parity bits per clock cycle. These M_b parity bits correspond to the last M_b sub-matrices of the H matrix. After encoding the frame, the codeword bits c_n are modulated: as an example, for the case of a single bit memory cell, simple two level amplitude modulation is used and $mb_n = (-1)^{c_n}$. The

LLR for each codeword bit c_n is calculated as follows:

$$l_n = (\sigma N(0,1) + mb_n) \times 2/\sigma^2. \quad (1)$$

For hardware implementation purpose, (1) is modified as follows

$$l_n = (N(0,1) + mb_n/\sigma) \times 2/\sigma. \quad (2)$$

Thus, if $1/\sigma$ is pre-computed, then (2) needs a single multiplication. The single precision floating point AWGN generator proposed in [12] is used to produce high quality noise samples. The AWGN generator is based on the Box-Muller algorithm and is implemented using the Xilinx high-level-synthesis tool Vivado HLS 2014.2. The HLS tool provides the flexibility of specifying the throughput of the design and therefore, high speed implementations can be obtained. The high speed implementation and the high quality of the noise samples are very important for accurately measuring the BER or FER, especially at high SNR.

The *Flt_to_Fxd Converter* module is devoted to represent the LLRs as Q bit fixed point values, where *Int_Bits* and *Frac_Bits* are the numbers of integer (excluding the sign) and fractional bits, respectively. The *vld* signal (as shown in Fig. 2) is asserted whenever there are valid LLR values at the output of *Flt_to_Fxd Converter* module. There are 31 pipeline stages in the AWGN generator, 7 pipeline stages in the floating point adder and multiplier and 17 pipeline stages in the *Flt_to_Fxd Converter*. Therefore, after asserting the *Start_awgn* signal, a valid LLR value appears at the input of the decoder after 55 clock cycles. The architecture works in pipeline, namely the SGs generate the source bits of the next frame while the LLRs are transfered to the decoder.

B. Decoder

The partially parallel decoder architecture for regular QC-LDPC codes presented in [4] has been reused to implement the decoder core. The initial architecture was conceived to support generic EG-LDPC codes with high code rate and circulant weight. Key architectural features include compile time flexibility, with respect to the size and rate of the selected code, high level of parallelism, both at the decoder level and inside the check node unit, support for layered decoding scheduling. Two important changes have been introduced in the decoder architecture. i) To save hardware resources, right shifting in LLR memory due to pipeline stages in the decoder is avoided by connecting each processing element to the LLR memory through multiplexers. This results in a reduction of 50% resources of FPGA as compared to the decoder in [4]. ii) Decoding performance has been enhanced, by adopting the conditional variable to check node updating rule proposed in [8], which avoids the performance degradation introduced by a-posteriori LLR saturation. The update rule is given as follows:

$$VTC_{ij}^{(k)} = \begin{cases} APP_j^{(k-1)} & \text{if } APP_j^{(k-1)} = APP_j^{(max)} \\ APP_j^{(k-1)} - CTV_{ij}^{(k-1)} & \text{else} \end{cases}, \quad (3)$$

where $VTC_{ij}^{(k)}$ is the variable-to-check (VTC) message in kth iteration from jth variable node to ith check node, where $1 \leq i \leq M$ and $1 \leq j \leq N$, respectively.

IV. Results

In this section, FPGA implementation and simulation results of the two algebraic QC-LDPC codes developed for the 8 KB page size of NAND flash memory with 3450 (5%) spare bits are discussed.

The first code used is the rate 0.961, (69615,66897) EG-LDPC code [3]. The 4095×69615 parity check matrix of this code consists of a 1×17 array of 4095×4095 sub-matrices. The row and column weights are 272 and 16, respectively. There are 1377 linearly dependent rows in the matrix. For adapting this code to the 8 KB page size of NAND flash memory, 1361 zero bits are inserted at the beginning of the information bits and therefore, the code becomes a (68254,65536) shortened EG-LDPC code [3]. For the shortening process, the first 1361 locations of the LLR memory on the decoder side are initialized with the maximum LLR value.

The second developed code is the rate 0.96, (68544,65861) algebraic QC-LDPC code, based on the construction method proposed in [6]. We choose the Galois field (GF) 449 and took two subsets of elements from this field, i.e. $S1 = \{\alpha^0, \alpha^1, \alpha^2, \alpha^3, \alpha^4, \alpha^5\}$ and $S2 = \{\alpha^{50}, \alpha^{51}, \alpha^{52}, \ldots, \alpha^{202}\}$, where α is the primitive element of GF. The multiplication factor β [6], is taken as 1. Based on these two subsets and β, we constructed a 2688×68544 parity check matrix that consists of 6×153 array of 448×448 sub-matrices. Each sub-matrix is a cyclically shifted identity matrix. There are 5 linearly dependent rows in the matrix. The row and column weights are 153 and 6, respectively. For this matrix we obtain a (68219,65536) shortened code by inserting 325 zero bits to the information bits. The shortening process is the same as for the (69615, 66897) code.

We used the Xilinx Vivado 14.2 tool for all the design flow, including simulation, synthesis, mapping and place and route of the whole system for the two codes. We targeted the Xilinx

Table I: FPGA Implementation Results of (69615,66897) EG and (68544,65861) Algebraic QC-LDPC Codes.

(69615,66897) EG-LDPC Code				
	AWGN Generator	Encoder	Decoder	Overall System
6-input LUTs	4360 (1.00%)	37365 (8.6%)	233368 (53.87%)	339269 (78.32%)
Slice Registers	4296 (0.5%)	69615 (8%)	160897 (18.57%)	351470 (40.57%)
Block RAMs	1 (0.0%)	0	66 (4.5%)	97 (6.6%)
DSP Slices	56(1.55%)	0	0	640 (17.78%)
clock frequency	-	-	-	100 MHz
Throughput per iteration	-	-	1.47 Gb/s	1.47 Gb/s
(68544,65861) Algebraic LDPC Code				
	AWGN Generator	Encoder	Decoder	Overall System
6-input LUTs	4360 (1.00%)	72701 (16.78%)	154364 (35.63%)	308736 (71.26%)
Slice Registers	4296 (0.5%)	65861 (7.6%)	99802 (11.5%)	301321 (34.77%)
Block RAMs	1 (0.0%)	0	6 (0.4%)	48 (3.26%)
DSP Slices	56(1.55%)	0	0	640 (17.78%)
clock frequency	-	-	-	100 MHz
Throughput per iteration	-	-	1.51 Gb/s	1.51 Gb/s

XC7VX690T FPGA device for the implementation of the system. For the (69615,66897) EG-LDPC code, we selected the parallelism factor for the decoder as 7 and therefore, the number of rows processed by a single check node unit of the decoder is $N_r = 585$. We used 7 bits for the representation of LLRs, where 2 bits are used for the fractional part. The resources consumed by the different modules and the overall resources of the hardware are shown in Table I. The floating point AWGN channel unit consumes only 1% of the slice LUTs of FPGA, while consuming more dedicated DSP slices. The maximum clock frequency is 100 MHz (9.9 ns). The parallelism factor at the *LLR Generator* module, as mentioned in section III, is taken as 17 i.e. $P_l = N_b$ and, therefore, 17 LLRs are generated in parallel and transfered to the LLR memory. As a consequence, it takes $s = 4095+55$ clock cycles to transfer all the LLRs of the channel to the decoder, where 4095 is the circulant weight and 55 clock cycles are required due to the pipeline stages in the *LLR Generator* module. The throughput per iteration of the system for this code is also given in Table I and can be calculated as

$$T = \frac{z \times f_{clk}}{s + no\_iters \times (N_r + 5)}, \quad (4)$$

where f_{clk} is the clock frequency, s is the LLR transfer latency and no_iters the number of iterations. For the second (68544,65861) QC-LDPC code, we took the parallelism factor for the decoder as 6 with $N_r = 448$. We used 7 bits for the LLR, where 0 bits are used for the fractional part. The hardware resources for this code are summarized in Table I. The number of clock cycles required to transfer the LLRs for this case is equal to $s = 448\times\text{ceil}(153/17)+55$, where the same parallelism factor of 17 is taken for the LLR generator. The throughput per iteration for this code is also given in Table I. As it can be seen from the table, this code is able to achieve a higher throughput than the EG-LDPC code due to the number of check nodes which are almost 65% less than EG code. Moreover, this code features the degree of the check node unit which is almost half the EG code, therefore, the resources consumed by the decoder are less than the ones required by the decoder of the EG code. The encoder of this code consumes twice the resources as compared to the encoder of the EG code. This is due to the fact that all the M_b parity bits ($M_b = 6$ for second code) are generated in parallel and the encoding process is completed in 448 clock cycles. The resources can be reduced by generating one parity bit per clock cycle. In this case, the encoding will be done in 2688 clock cycles, which is still faster than the first code i.e. 4095 clock cycles to generate all parity bits.

Fig. 3 shows the comparison of the FER performance of both codes. The normalization factor γ in the normalized min-sum decoding algorithm used for the EG-LDPC code is 0.25, whereas the γ used for the second code is 0.375. The maximum number of iterations of the decoder is set to 8 for both codes. The normalization factor is set based on the simulation results obtained from a software model written in C. 100 wrong frames are observed at each SNR point except at a FER of 10^{-9} and 10^{-10}, where at least 10 and 4 wrong frames are observed, respectively. The average number of iterations for the first and second code at higher SNR are observed to be

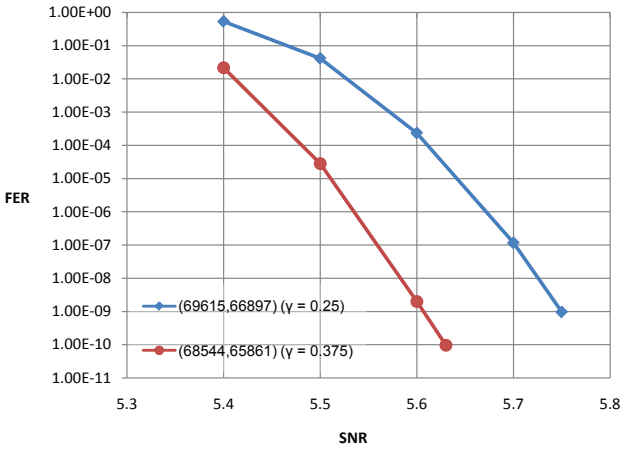

Figure 3: Frame error rate performance of rate 0.96 QC-LDPC codes.

Table II: Throughput and Block length Comparison with Previous Implementations (single core versions)

	[7]	[10]	[9]	This work
Max. Block length	16384	4923	2048	69615
Max. Rate	0.937	0.89	0.84	0.96
Throughput Single core	360 Mb/s	175 Mb/s	1350 Mb/s	1150 Mb/s

3.2 and 4.23, respectively. Therefore, an average throughput of 1.15 and 1.13 Gb/s is achieved for these codes. The FER of 10^{-9}, which requires simulation of at least 10^{10} frames, is achieved in 7 days. Both codes do not show any error floor at a FER of at least 10^{-9}. However, Fig. 3 shows that the proposed code outperforms the EG code by 0.15 dB at a FER of 10^{-9}. The software simulations show that the number of bit errors/block error at different SNR points are also less for the proposed code as compared to the EG code. It also features less memory requirements and reduced decoder complexity, which are important features for the application to NAND flash memories.

Table II shows the comparison of the block lengths, code rate and throughput of our work with the state-of-art FPGA based implementations. As it can be seen from the table, the block length of the code is very high as compared to the previously reported work. Moreover, to the best of our knowledge, the high speed FPGA implementation of very high-circulant-weight LDPC codes is not addressed in the literature. The speed of our implementation is comparable to the speed of the fastest reported emulator (single core version). This speed can be increased by increasing P_l. As an example, by taking $P_l = 22$ for the second code, which consumes only 71.26% LUTs of FPGA, the average throughput of 1.35 Gb/s can be achieved. This increase in P_l requires two more AWGN generators, 5 more SGs, adders, multipliers and *Flt_to_Fxd Converter* modules, respectively. As these modules use more DSP slices and less logic resources of FPGA, this increase in

parallelism factor will result in slight increase in the percentage of the LUTs usage of FPGA. Moreover, the achievable throughput of all implementations reported in Table II scales linearly with the number of allocated cores.

V. Conclusion

For addressing the error correction in 8 KB page size of NAND flash memories, we evaluated the performance of very high code rate (0.96) and large block length algebraic QC-LDPC codes through a generalized and high throughput FPGA based emulator system. We used two codes, (69615,66897) EG-LDPC code and (68544,65861) algebraic QC-LDPC code. Simulation results on AWGN channel show that these codes do not suffer from error floor at a FER of 10^{-9}. Moreover, the proposed (68544,65861) algebraic QC-LDPC code shows good error performance and reduced hardware complexity as compared to the EG-LDPC code.

References

[1] N. Mielke, T. Marquart, Ning Wu, J. Kessenich, H. Belgal, Eric Schares, F. Trivedi, E. Goodness, and L.R. Nevill. Bit error rate in NAND flash memories. In *Reliability Physics Symposium, 2008. IRPS 2008. IEEE International*, pages 9–19, April 2008.

[2] G. Dong, N. Xie, and T. Zhang. On the Use of Soft-Decision Error-Correction Codes in NAND Flash Memory. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 58(2):429–439, 2011.

[3] Jonghong Kim and Wonyong Sung. Rate-0.96 LDPC decoding VLSI for soft-decision error correction of NAND flash memory. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 22(5):1004–1015, May 2014.

[4] S.A.A. Zaidi, M. Awais, C. Condo, M. Martina, and G. Masera. FPGA accelerator of quasi cyclic EG-LDPC codes decoder for NAND flash memories. In *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pages 190–195, Oct 2013.

[5] Yu., S. Lin, and M.P.C. Fossorier. Low-density parity-check codes based on finite geometries: a rediscovery and new results. *Information Theory, IEEE Transactions on*, 47(7):2711–2736, 2001.

[6] Juane Li, Keke Liu, Shu Lin, and K. Abdel-Ghaffar. Algebraic quasi-cyclic LDPC codes: Construction, low error-floor, large girth and a reduced-complexity decoding scheme. *Communications, IEEE Transactions on*, 62(8):2626–2637, Aug 2014.

[7] H. Zhong, T. Zhong, and E.F. Haratsch. Quasi-Cyclic LDPC Codes for the Magnetic Recording Channel: Code Design and VLSI Implementation. *Magnetics, IEEE Transactions on*, 43(3):1118–1123, 2007.

[8] C. Marchand, L. Conde-Canencia, and E. Boutillon. Architecture and finite precision optimization for layered LDPC decoders. In *Signal Processing Systems (SIPS), 2010 IEEE Workshop on*, pages 350–355, Oct 2010.

[9] F. Angarita, V. Torres, A. Perez-Pascual, and J. Valls. High-throughput FPGA-based emulator for structured LDPC codes. In *Electronics, Circuits and Systems (ICECS), 2012 19th IEEE International Conference on*, pages 404–407, 2012.

[10] Yu Cai, S. Jeon, Ken Mai, and B.V.K.V. Kumar. Highly parallel FPGA emulation for LDPC error floor characterization in perpendicular magnetic recording channel. *Magnetics, IEEE Transactions on*, 45(10):3761–3764, Oct 2009.

[11] Z. Wang and Z. Cui. Low-Complexity High-Speed Decoder Design for Quasi-Cyclic LDPC Codes. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(1):104–114, 2007.

[12] S.A.A. Zaidi, M. Martina, and G. Masera. Rapid prototyping of floating point AWGN channel using high-level synthesis. In *Forum on specification and Design Languages (FDL),2014 Conference on*, pages 165–168, Oct 2014.