

LOW-DENSITY PARITY-CHECK CODE DECODER
DESIGN AND ERROR CHARACTERIZATION ON AN
FPGA BASED FRAMEWORK

by
Burak Unal

Copyright © Burak Unal 2019

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

2019

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Burak Unal, entitled Low-Density Parity-Check Code Decoder Design and Error Characterization on an FPGA Based Framework and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.



Ali Akoglu Date: April 8, 2019



Bane Vasic Date: April 8, 2019



Ali Bilgin Date: April 8, 2019



Tosiron Adegbija Date: April 8, 2019

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.



Dissertation Director: Ali Akoglu Date: April 8, 2019

ACKNOWLEDGMENTS

I would like to thank all the people who contributed to my PhD journey with their support. Those who inspired me; those who encouraged me; those who were with me on difficult times.

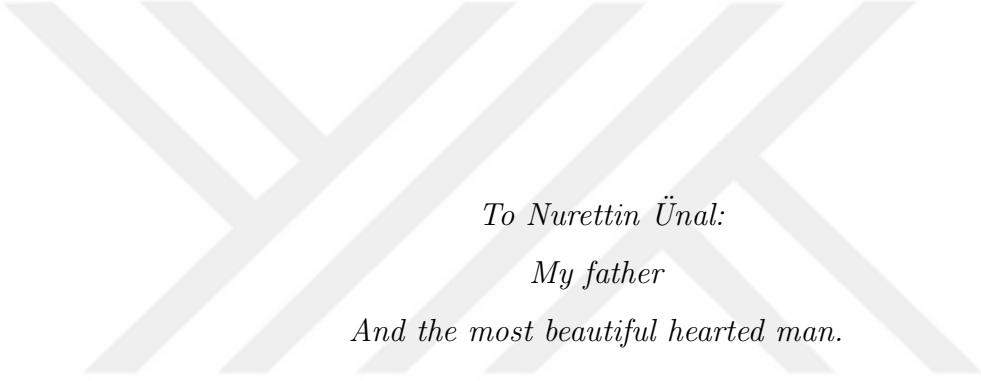
First and foremost, I would like to express my deepest acknowledgments to the most valuable persons in my life, my beloved family, and most importantly my parents, Nurettin Ünal and Güldane Ünal. I would not be reaching for and achieving my dreams without their encouragement and endless love. I am sincerely grateful to my grandfather, Bayram Ünal and my wonderful sisters, Cansu, Özge, and İpek Naz Ünal.

I would like to thank you Dr. Ali Akoglu for serving as my advisor during past several years. This dissertation is the outcome of several years of research and collaborations. I would like to thank the following people who have made this work possible either through a direct collaboration or through their guidance: Dr. Ali Bilgin and Dr. Bane Vasić. In addition, I would like to thank Dr. Fakhreddine Ghaffari for his collaboration.

I would like to thank all my friends and colleagues in the Reconfigurable Computing Laboratory for sharing their enjoyable moments.

I am very grateful to my undergrad advisor, Prof. Kenan Danisman for his friendship, support, and understanding throughout both my professional career and personal life.

DEDICATION



*To Nurettin Ünal:
My father
And the most beautiful hearted man.*

TABLE OF CONTENTS

LIST OF FIGURES	7
LIST OF TABLES	9
ABSTRACT	10
CHAPTER 1. INTRODUCTION	12
1.1. Overview	12
1.2. Problem Statement and Aims	13
1.3. Summary of Contributions	14
1.3.1. Algorithmic Contribution	14
1.3.2. Architecture Specific Contribution	15
1.3.3. FPGA Based Framework	16
CHAPTER 2. PRELIMINARIES	20
2.1. Overview of Decoding Algorithms	20
2.1.1. Gallager B (GaB)	21
2.1.2. GDBF and PGDBF Analysis	22
CHAPTER 3. PROBABILISTIC GAB ALGORITHM	26
3.1. Probabilistic GaB Algorithm Methodology	27
3.1.1. Determining How to Disturb the VNU	28
3.1.2. Determining the p_v Value	31
3.1.3. Determining the s_i Value	32
CHAPTER 4. HARDWARE DESIGN OF LDPC DECODERS	36
4.1. Hardware Design	36
4.1.1. GaB and PGaB Hardware Design	36
4.1.2. GDBF and PGDBF Hardware Design	39
4.2. Simulation Environment	43
4.3. Performance Analysis	45
4.4. Robustness Analysis	48
4.4.1. Effect of Code Rate	49
4.4.2. Effect of Codeword Length	51
CHAPTER 5. ROUTABILITY PROBLEM OF THE LDPC CODE	54
5.1. Background	54
5.2. Congestion Analysis	55
5.3. Trend Analysis	57

TABLE OF CONTENTS—*Continued*

5.3.1. Codeword Length and Code Rate vs. Resource Usage	58
5.3.2. Codeword Length and Code Rate vs. Critical Path Delay	61
5.3.3. Partitioning Amount vs. LUT Usage-Delay Product Analysis	63
CHAPTER 6. FPGA BASED FRAMEWORK	67
6.1. Related Work	67
6.2. Framework and Hardware Implementation	70
6.3. Quantifying the Benefits of the FPGA Based Framework	73
6.3.1. Case Study: Time to Generate FER Analysis	74
6.3.2. Case Study: Error Pattern Analysis	74
CHAPTER 7. CONCLUSIONS AND PERSPECTIVES	79
7.1. Summary and Contributions	79
REFERENCES	81

LIST OF FIGURES

FIGURE 2.1. Tanner graph (left) and its parity check matrix (right).	20
FIGURE 2.2. General architecture of VNUs for (a) GaB, (b) PGaB, (c) GDBF, and (d) PGDBF for $d_v = 4$ and $N = 1296$. (e) Maximum finder unit for GDBF and PGDBF decoders.	24
FIGURE 2.3. The evolution of decoding algorithms in term of error correction performance. GaB, GDBF, PGDBF, MS, and OMS FER comparison: FER vs. probability of error introduced to each bit of the 1296-bit codeword with $d_v = 4, d_c = 8, M = 648$, and Code Rate = 0.5.	25
FIGURE 3.1. Frame Error Rate versus p_v ($\alpha = 0.02, 0.025, 0.03$, and 0.035). LDPC code ($d_v = 4, d_c = 8, Z = 54$), ($N = 1296, M = 648$) when switching iteration s_i is set to 15.	32
FIGURE 3.2. Frame Error Rate and Average iteration versus iteration number to switch from GaB to PGaB (QC-LDPC codes with (N, d_v, d_c, R) configurations of (155, 3, 5, 0.5), (1296, 3, 6, 0.5) and (1296, 4, 8, 0.5)).	33
FIGURE 3.3. Comparison of average number of iterations for PGaB, GaB, PGaB Hybrid (GaB for the first 15 iterations, and PGaB onwards) ($d_v = 4, d_c = 8, N = 1296, M = 648$).	34
FIGURE 4.1. Overall decoder architecture (a) for PGaB, VNU architecture for $d_v = 4$ (c) and LFSR-32bits based Random Number Generator (b).	37
FIGURE 4.2. Architecture of GDBF and PGDBF for $d_v = 4$, and $N = 1296$, where d_c is determined by the code rate.	39
FIGURE 4.3. Architecture of VNU for GDBF and PGDBF for $d_v = 4$, and $N = 1296$	39
FIGURE 4.4. GaB, PGaB, GDBF, PGDBF, MS, OMS FER comparison: FER vs. probability of error introduced to each bit of the 1296-bit codeword with $d_v = 4, d_c = 8, M = 648$, and Code Rate = 0.5.	44
FIGURE 4.5. GaB, PGaB, GDBF, PGDBF, MS, OMS FER comparison: FER vs. probability of error introduced to each bit of the 1296-bit codeword with $d_v = 4, d_c = 16, M = 324$, and Code Rate = 0.75.	45
FIGURE 4.6. GaB, PGaB, GDBF, PGDBF, MS, and OMS: FER vs. crossover probability for the QC-LDPC code with ($d_v = 4, d_c = 28, N = 2212, M = 312$, and Code Rate = 0.857).	47
FIGURE 4.7. Post-routing layout for the PGaB designs with two code rates. a) PGaB decoder on the QC-LDPC code with ($d_v = 4, d_c = 8, N = 1296, M = 648$, and Rate = 0.5). b) PGaB decoder on the QC-LDPC code with ($d_v = 4, d_c = 16, N = 1296, M = 324$, and Rate = 0.75).	50

LIST OF FIGURES—*Continued*

FIGURE 5.1. Post-routing layout of GaB decoder for the QC-LDPC code ($d_v = 4, d_c = 28, n = 1106$, and code rate = 0.857) on the Xilinx Zynq XC7Z020 FPGA for Regular (left) and Depopulation based implementation (right).	56
FIGURE 5.2. Two-way partitioning methodology for CNU with $d_c = 28$.	57
FIGURE 5.3. Resource usage vs Codeword length comparison for the QC-LDPC code with ($d_v = 4, d_c = 8$, code rate=0.5) based on Xilinx Zynq XC7Z020 FPGA.	59
FIGURE 5.4. Resource usage vs. Codeword Length comparison for the QC-LDPC code with ($d_v = 4, d_c = 28$, code rate=0.857) based on Xilinx Zynq XC7Z020 FPGA.	60
FIGURE 5.5. Critical path delay (ns) vs Codeword length for the QC-LDPC code with ($d_v = 4, d_c = 8$, code rate=0.5) based on Xilinx Zynq XC7Z020 FPGA.	61
FIGURE 5.6. Critical path delay (ns) vs Codeword length for the QC-LDPC code with ($d_v = 4, d_c = 28$, code rate=0.857) based on Xilinx Zynq XC7Z020 FPGA.	62
FIGURE 5.7. Resource usage trend with respect to the x-way partitioning for the QC-LDPC code with codeword length of 1106 ($d_v = 4, d_c = 28$, code rate=0.857) based on Xilinx Zynq XC7Z020 FPGA.	64
FIGURE 5.8. Critical path delay trend with respect to the x-way partitioning for the QC-LDPC code with codeword length of 1106 ($d_v = 4, d_c = 28$, code rate=0.857) based on Xilinx Zynq XC7Z020 FPGA.	64
FIGURE 5.9. LUT usage-delay product trend with respect to the x-way partitioning for the QC-LDPC code with codeword length of 1106 ($d_v = 4, d_c = 28$, code rate=0.857) based on Xilinx Zynq XC7Z020 FPGA.	66
FIGURE 6.1. Overall architecture of FPGA based testbed. Red colored wires indicate control signals, others indicate internal connections.	72
FIGURE 6.2. GaB and PGaB FER comparison: FER vs. probability of error introduced to each bit of the 1296-bit codeword with ($d_v = 4, d_c = 8$, code rate = 0.5).	75
FIGURE 6.3. 4-error pattern for GaB algorithm. Black/white denotes erroneous/correct VNU, and unsatisfied/satisfied CNU.	76

LIST OF TABLES

TABLE 3.1. Truth Table for PGaB Algorithm	29
TABLE 4.1. Hardware Resource Utilization, Throughput and Clock Rate of Decoding Algorithms Implemented for Tanner Code on Virtex6 FPGA .	40
TABLE 4.2. Resource usage of GaB, PGaB, GDBF, and PGDBF based on the FPGA implementations for QC-LDPC $(N, d_v, R)=(1296, 4, 0.5)$, QC-LDPC $(N, d_v, R)=(1296, 4, 0.75)$, and QC-LDPC $(N, d_v, R)=(2212, 4, 0.857)$. (% indicates the difference with respect to GaB)	46
TABLE 4.3. Maximum clock rate and throughput of GaB, PGaB, GDBF, and PGDBF based on the FPGA implementations for QC-LDPC $(N, d_v, R)=(1296, 4, 0.5)$, QC-LDPC $(N, d_v, R)=(1296, 4, 0.75)$, and QC-LDPC $(N, d_v, R)=(2212, 4, 0.857)$. (% indicates the difference with respect to GaB)	46
TABLE 4.4. Throughput-to-area ratio (TAR) and Normalized Throughput (T_p) for GaB, PGaB, GDBF, and PGDBF when the crossover probability is fixed. Based on the FPGA implementations for QC-LDPC with $(N, d_v, R)=(1296, 4, 0.5)$, QC-LDPC $(N, d_v, R)=(1296, 4, 0.75)$, and QC-LDPC $(N, d_v, R)=(2212, 4, 0.857)$	51
TABLE 5.1. The number of LUTs used as routing resource for different code-word length on the LDPC code with $(d_v = 4, d_c = 28, \text{code rate}=0.857)$ based on Xilinx Zynq XC7Z020 FPGA.	60
TABLE 5.2. Total number of paths vs. codeword length (n) on the LDPC code with $(d_v = 4, d_c = 28, \text{rate}=0.857)$	61
TABLE 6.1. FPGA based LDPC testbeds.	69
TABLE 6.2. Hardware resource utilization of LDPC testbed implementation on the FPGA. Available resource is based on the Xilinx Zynq XC7Z020 and Virtex-7 XC7VX485T FPGAs.	73
TABLE 6.3. Execution time for GaB and PGaB on the FPGA based testbed (Xilinx Zynq XC7Z020 FPGA) for the crossover probabilities of 0.005 and 0.01 where $(d_v = 4, d_c = 8, n = 1296, \text{code rate}=0.5)$ (* indicates estimated time).	75

ABSTRACT

Low-Density Parity-Check (LDPC) codes have gained popularity in communication systems and standards due to their capacity approaching error correction performance. Among all the hard-decision based LDPC decoders, Gallager B (GaB), due to simplicity of its operations, poses as the most hardware friendly algorithm and an attractive solution for meeting the high-throughput demand in communication systems. However, GaB suffers from poor error correction performance. In this work, we first propose a resource efficient GaB hardware architecture that delivers the best throughput while using fewest Field Programmable Gate Array (FPGA) resources with respect to the state of the art comparable LDPC decoding algorithms. We then introduce a Probabilistic GaB (PGaB) algorithm that disturbs the decisions made during the decoding iterations randomly with a probability value determined based on experimental studies. We achieve up to four orders of magnitude better error correction performance than the GaB with a 3.4% improvement in normalized throughput performance. PGaB requires around 40% less energy than GaB as the probabilistic execution results with reducing the average iteration count by up to 62% compared to the GaB. We also show that our PGaB consistently results with an improvement in maximum operational clock rate compared to the state of the art implementations.

In this dissertation, we also present a high throughput FPGA based framework to accelerate error characterization of the LDPC codes. Our flexible framework allows the end user adjust the simulation parameters and rapidly study various LDPC codes and decoders. We first show that the connection intensive bipartite graph based LDPC decoder hardware architecture creates routing stress for longer codewords that are utilized in today's communications systems and standards. We address this problem by partitioning each processing element (PE) in the bipartite graph in such a

way that the inputs of a PE are evenly distributed over its partitions. This allows depopulating the Look Up Table (LUT) resources utilized for the decoder architecture by spreading the logic across the FPGA. We show that even though LUT usage increases, critical path delay reduces with the depopulation. More importantly, with the depopulation technique an unroutable design becomes routable, which allows longer codewords to be mapped on the FPGA. We then conduct two experiments on error correction performance analysis for the GaB and PGaB algorithms, demonstrate our framework's ability to reach a resolution level that is not attainable with general purpose processor (GPP) based simulations, which reduces the time scale of simulations to 24 hours from an estimated 199 years. We finally conduct the first study on identifying all possible codewords that are not correctable by the GaB for the case where a codeword has four errors. We reduce the time scale of this simulation that requires processing 117 billion codewords to 4 hours and 38 minutes with our framework from an estimated 7800 days on a single GPP.

Chapter 1

INTRODUCTION

1.1 Overview

Error correction codes have been utilized in several communication systems to ensure reliable transmission of information. Claude Shannon established theoretical limit at which information can be transmitted reliably over a noisy channel in 1948 [1]. Transmitting information reliably with a rate close to this theoretical limit is known as the channel capacity. Research efforts in decoding Low-Density Parity-Check (LDPC) codes have led to design and implementation of a myriad iterative decoding algorithms approaching channel capacity [2], [3], [4], [5]. LDPC codes offer performance improvement and implementation cost saving for long codeword lengths compared to Reed-Solomon (RS) [6] and Bose-Chaudhuri-Hocquenghem (BCH) [7], [8] codes as they are theoretically proven to be asymptotically good family of codes [9]. Therefore, for a sufficiently high codeword length, LDPC will outperform a BCH or RS code of a comparable rate. Binary LDPC codes have been widely adopted in several standards and applications [10], such as mobile communications [11], 10 Gigabit Ethernet (10GBase-T) [12], [13], digital video broadcasting (DVB-S2) [14], wireless local area network (WiFi IEEE 802.11n) [15], WiMAX (IEEE 802.16e) wireless communications [16], deep-space communications [17], as well as data storage systems [18]. LDPC codes have also been selected as the data channel coding scheme for the 3GPP new radio access technology of the fifth generation (5G) mobile communication standard [19], [20]. In addition, LDPC codes handle soft channel outputs which is essential in numerous applications even in optical communications and data storage channels, especially in flash memories [21], [22].

LDPC decoding algorithms mainly differ based on the nature of iterative opera-

tions applied over the received messages. Complexity level of these operations determine the trade-off between hardware performance and decoding performance. Here we note that, throughout this dissertation, with the hardware performance, we refer to the operational clock rate and throughput as well as the resource requirements of the decoder algorithm, and with the decoder performance, we refer to the error correction capability of the decoder algorithm measured based on the frame error rate metric. In the literature, we have seen soft-decision and hard-decision decoders as two main classes of LDPC decoding algorithms. Soft-decision decoders such as Belief Propagation (BP) [23],[24], Sum Product (SP) [25], Min-Sum (MS) [26], and Offset Min-Sum [27] offer high error correction performance with the cost of high computation complexity. On the other hand, hard-decision decoders such as Gallager B (GaB) [28],[29], Bit-Flipping (BF) [30], Gradient Descent Bit-Flipping (GDDBF) [31], and Probabilistic Gradient Descent Bit-Flipping (PGDBF) [32] have much less hardware requirements than soft-decision decoders, and achieve higher throughput with a trade-off in the error correction performance.

1.2 Problem Statement and Aims

Research efforts to this date for improving the error correction performance of LDPC decoding algorithms have inevitably faced the trade off on increased computational complexity. From hardware implementation and practical use perspectives, the increase in computational complexity results with increased demand for hardware resources. Therefore, these implementations, even though algorithmically efficient and highly parallelizable, become less scalable and harder to deploy as a component in systems designed for emerging standards that require longer codewords [33], [34], [35], [36]. We believe that there is a need for algorithms that target resource efficiency, scalability and error correction performance metrics concurrently. Among the hard-decision class of LDPC algorithms, hardware realization of the GaB has not been favorable due

to its poor decoding performance. On the other hand, GaB is an ideal candidate for designing a high-throughput decoder due to its simplicity of computations requiring combinational circuits at the scale of only 2-bit multiplication operations [28], [37]. Rapid evaluation of the LDPC algorithms and decoders while maintaining the trade-off between the hardware implementation efficiency and error correction performance becomes critical during this research. Therefore, this dissertation is concerned with a) improving the error correction performance of the GaB algorithm through algorithmic contributions without sacrificing its hardware efficiency, b) improving the scalability of the GaB hardware architecture to make it feasible to implement for longer code-words, and c) implementation of a general purpose Field Programmable Gate Array (FPGA) based framework to accelerate the simulations of hard-decision decoders for error correction performance analysis.

1.3 Summary of Contributions

1.3.1 Algorithmic Contribution

In this dissertation, our aim is to answer the question of whether it is feasible or not to bridge the gap between GaB and better performing hard-decision (bit flipping) based algorithms in terms of decoding performance without sacrificing its suitability for hardware implementation. We introduce a new algorithm called Probabilistic Gallager B (PGaB) by applying a probabilistic stimulation function over the iterative decoding process, conduct detailed experimental evaluations with respect to other decoders and show that our algorithm not only improves the decoding performance with respect to GaB by four orders of magnitude, but also requires fewest amount of hardware resources with respect to other comparable decoding algorithms GDBF and PGDBF while achieving equivalent or better decoding performance. We present the details of our incremental approach to designing and implementing the GaB and PGaB hardware architecture.

1.3.2 Architecture Specific Contribution

The connection intensive bipartite graph based LDPC decoder hardware architecture creates routing stress when implemented on the FPGA for longer codewords that are utilized in today's communications systems and standards. From FPGA point of view, even though there is sufficient amount of computing resources that would match the degree of parallelism desired by the design, implementation is less likely to pass the routing stage of the synthesis as the number of connections in the implementation increase with the code length, which in turn increases the stress on FPGA routing resources. Another contributor to the routing stress is the number of parity bits used by the communication medium, which has direct impact on the number of connections between each iteration of the decoding process since increasing the ratio of parity bit to data from 0.5 to 0.75 would mean increasing number of connections by a factor of 4 for a given codeword. Therefore for implementations of longer codewords and/or higher code rates, designers resort to reducing the degree of parallelism in their implementations. We address the routability problem by partitioning each processing element (PE) in the bipartite graph based LDPC decoder hardware architecture in such a way that we distribute inputs of a PE evenly over its partitions. This allows depopulating the Look Up Table (LUT) resources available on the FPGA fabric utilized for the decoder architecture by spreading the logic across the FPGA. Spreading the logic across the FPGA allows reducing the stress on routing. We use the GaB decoder as a case study and show that even though LUT usage increases, critical path delay reduces with the depopulation. More importantly, with the depopulation technique, an unroutable design becomes routable, which allows longer codewords to be mapped on the FPGA.

1.3.3 FPGA Based Framework

Evaluating the decoding performance of an LDPC code on a general purpose processor based single node requires extremely long simulation times, scaling to months and even years [38]. A typical simulation involves generating codewords (frames), injecting random errors to each, and measuring the ratio of codewords that are not recovered (frame error) to the total number of codewords tested. This ratio is referred to as the Frame Error Rate (FER). Many LDPC codes today reach to FER of 10^{-12} , which indicates that 10^{14} codewords have been tested and a benchmark of 100 codes were not corrected within a predefined number of iterations per codeword (typically 100 iterations). Therefore such simulations involve well beyond hundreds of millions of iterative error correction processes, and in most cases, the iteration count exceeds the billion mark for a conclusive evaluation at a resolution of 10^{-12} . Furthermore, in parallel to the technological advances in communication systems, the length of codewords have been steadily increasing. The need for extremely high resolution simulations combined with growing codeword length trends lead to excessively long simulation times, which makes software based simulations unpractical for the information theory researchers. From this regard, we propose to design and implement a flexible FPGA based framework to rapidly evaluate a given decoder algorithm with user defined simulation parameters. Our aim is to reduce the time scale of simulations and further allow researcher to conduct analysis such as error pattern, trapping set, and absorbing set. We present our approach for implementing the entire simulation flow on the FPGA as a self contained testbed.

The technical contributions of this dissertation are as follows:

- We propose a resource efficient GaB architecture for widely used quasi-cyclic (QC)-LDPC codes, implement it on the FPGA, and evaluate its hardware performance.

- We analytically study the cases for which a message bit received from the channel becomes the determining factor in GaB for a decision made during the iterative decoding process. We introduce an algorithm that disturbs those decisions with a predefined probability, and experimentally identify the probability value that results with optimal decoder performance.
- We experimentally show that a simple hardware-friendly random number generator based on linear feedback shift register (LFSR) is sufficient to disturb the decoder and improve the decoding performance.
- We propose a heuristic that allows switching to PGaB only after when GaB is not able to correct the errors in predetermined number of iterations.
- We investigate the impact of switching from GaB to PGaB at a specific iteration, and experimentally identify the iteration number that results with optimal decoder performance.
- We design and implement GaB and PGaB along with two hard decision based algorithms (GDBF and PGDBF) on the Xilinx Virtex- 6 FPGA (vc6vlx240t-2ff1156).
- We conduct a detailed robustness analysis that involves evaluating the impact of a change in code rate and codeword length over the FPGA based implementations of GaB, PGaB, GDBF and PGDBF covering 12 hardware implementations.
- We show that GaB architecture delivers the best throughput while using fewest FPGA resources, however performs the worst in terms of decoding performance. The PGaB results with up to four orders of magnitude decoding performance improvement over the GaB, exceeding the performance of GDBF over the codes

studied in this dissertation, with a negligible loss (less than 1%) in throughput performance compared to the GaB.

- We analyze the critical path delay and resource usage trends for hardware implementations of the GaB algorithm with respect to the increase in codeword length.
- We show that routability becomes a bottleneck as the codeword length increases and adapt design partitioning technique to depopulate the logic across the FPGA and reduce congestion.
- We present an experimental analysis through resource usage, delay, and resource usage-delay product trends with respect to the amount of partitioning. We also correlate these trends with fracturable LUT utilization based on the level of partitioning and number of occupied paths.
- We propose a depopulation based hardware implementation technique and show that designs for the codewords that are not routable with the regular implementation become routable with the depopulation approach, while reducing the critical path delay by up 32% and increasing the LUT usage by 9%.
- We propose an FPGA based framework to accelerate the study of error correction performance analysis for LDPC codes. We present our approach to implementing the entire simulation flow on the FPGA as a self contained framework in order to reduce the timescale of our simulations.
- Finally we present two case studies on investigating the error correction performance and the types of error patterns that are not recoverable by a given LDPC algorithm. We show that our testbed reduces the timescale of error correction performance simulations from an estimated time scale of 199 years on a CPU

to less than a day, and four error pattern analysis from an estimated time of 7800 days to less than five hours.

We believe that our self-contained FPGA based framework [39] is a valuable tool for information theorists to expose the weaknesses of a decoder algorithm under investigation through rapid error analysis and study ways to improve that decoding algorithm [40], [41]. The rest of this dissertation is organized as follows. In Chapter 2, we first provide the background necessary for the discussions LDPC code, and then give an overview of the baseline GaB LDPC algorithm, along with the GDBF and PGDBF decoding algorithms. In Chapter 3, we present our methodology for introducing the probabilistic behavior to the GaB and determining the critical parameters for the PGaB implementation. In Chapter 4 we first discuss the hardware implementations for GaB, PGaB, GDBF, and PGDBF, and then we evaluate the decoding performance and hardware performance of PGaB after giving an overview of our simulation environment. In Chapter 5, we investigate the congestion problem experimentally, introduce the partitioning approach for depopulating the logic, and conduct resource usage and path delay trend analysis to quantify the benefits of the depopulation strategy from FPGA implementation point of view. After discussing the details of our FPGA based framework in Chapter 6, we present our error correction performance and error pattern analysis case studies based on the proposed framework. Finally, in Chapter 7, we present our conclusions and future work.

Chapter 2

PRELIMINARIES

In this chapter, we provide the background information necessary for discussion of the LDPC code.

2.1 Overview of Decoding Algorithms

An LDPC code is defined by a sparse parity-check matrix H [2], with size (M, N) , where $N > M$. A codeword is a vector $\mathbf{x} = (x_1, x_2, \dots, x_N) \in \{0, 1\}^N$, which satisfies $H\mathbf{x}^T = 0$. We denote by $\mathbf{r} = \{r_1, r_2, \dots, r_N\} \in \{0, 1\}^N$ the output of a Binary Symmetric Channel (BSC), in which the bits of the transmitted codeword \mathbf{x} have been flipped with crossover probability α . The graphical representation of an LDPC code is a bipartite graph called Tanner graph [42], [43] composed of two types of nodes including N number of Variable Node Units (VNUs, $v_n, n = 1, \dots, N$) and M number of Check Node Units (CNUs, $c_m, m = 1, \dots, M$). In the Tanner graph, a VNU v_n is connected to a CNU c_m when $H(m, n) = 1$. An example Tanner graph and its H

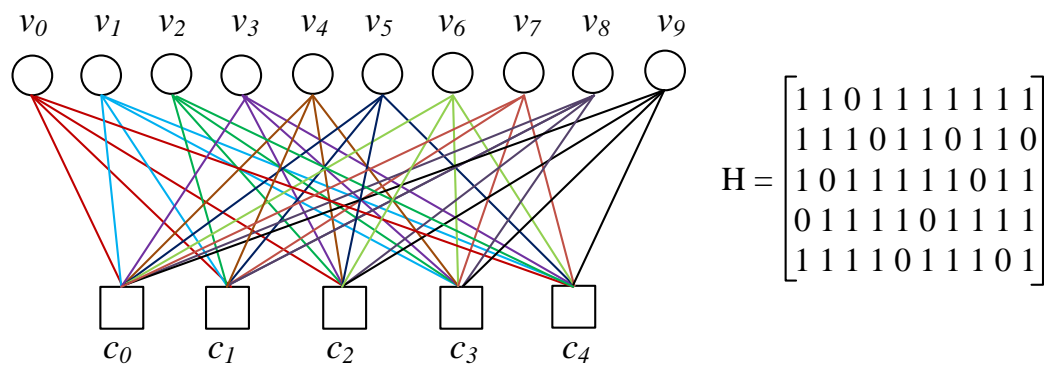


FIGURE 2.1. Tanner graph (left) and its parity check matrix (right).

matrix are shown in Figure 2.1. Let us also denote $\mathcal{N}(v_n)$ the set of CNUs connected to the VNU v_n , with a connection degree $d_v = |\mathcal{N}(v_n)|$, and denote $\mathcal{N}(c_m)$ the set of VNUs connected to the CNU c_m , with a connection degree $d_c = |\mathcal{N}(c_m)|$. Based on a decision function applied over the received messages from each adjacent vertex, each CNU and VNU sends a message back to its adjacent vertices. This iterative message processing between nodes recover the original data, which may have been exposed to channel noise.

2.1.1 Gallager B (GaB)

Binary messages are exchanged between CNUs and VNUs during each iteration of the decoding process and new messages are computed in an extrinsic manner. A VNU excludes the message received from a CNU, when the VNU is calculating the message to be sent back to that specific CNU. This is valid for the message calculation for the CNU as well. Each message represents an estimation on the correctness of the received word from the channel. Eventually, VNUs and CNUs accumulate gradually more information with each new iteration, which increasingly improves the codeword correction capacity. The estimation of the codeword is called posteriori decision information and is represented by $d_{n,m}^{(i)}$. Let $E(x)$ represent a set of edges connected to a node x in the Tanner graph. The $v_{n,m}^{(i)}(e)$ denotes the extrinsic messages sent on edge e from a VNU v_n to a CNU c_m at iteration i and the $c_{m,n}^{(i)}(e)$ represents the extrinsic messages sent on edge e from a CNU c_m to a VNU v_n at iteration i . The received word from the channel at a VNU v_n is denoted as r_n . We express the operation of VNU and CNU using Equations 2.1.1 and 2.1.2 respectively.

$$v_{n,m}^{(i)}(e) = \begin{cases} 1, & \text{if } r_n + (\sum_{e' \in \mathcal{N}(v_n) \setminus e} c_{m,n}^{(i)}(e')) > b_n \\ 0, & \text{if } r_n + (\sum_{e' \in \mathcal{N}(v_n) \setminus e} c_{m,n}^{(i)}(e')) < b_n \\ r_n, & \text{otherwise} \end{cases} \quad (2.1.1)$$

where i is the iteration count, e' is the set of extrinsic edges, and b_n is the threshold calculated as $b_n = d_v/2$.

$$c_{m,n}^{(i)}(e) = \left(\sum_{e' \in \mathcal{N}(c_m) \setminus e} v_{n,m}^{(i)}(e') \right) \text{mod} 2 \quad (2.1.2)$$

At each iteration, a new value of posteriori decision $d_{n,m}^{(i)}$ is computed as follows

$$d_{n,m}^{(i)} = \begin{cases} 1, & \text{if } r_n + \left(\sum_{e \in \mathcal{N}(v_n)} c_{m,n}^{(i)}(e) \right) > b_n \\ 0, & \text{if } r_n + \left(\sum_{e \in \mathcal{N}(v_n)} c_{m,n}^{(i)}(e) \right) < b_n \\ r_n, & \text{otherwise} \end{cases} \quad (2.1.3)$$

The GaB decoding process is shown in Algorithm 1. This iterative decoding process begins with sending the received message bit from each VNU to its CNUs defined by the H matrix. In a series of iterations CNUs and VNUs exchange information till a satisfaction criteria is met, which indicates successful recovery of the original data transmitted over a channel, which may have been exposed to errors due to noise. The CNU and VNU functions, satisfaction criteria, and connection topology among CNUs and VNUs determine nature of the LDPC algorithm. The VNU for GaB can be implemented using *Majority* gates (based on *and* and *or* logic functions only), and does not require complex operations such as the maximum finder required by the GDBF and PGDBF, along with the additional random number generator required by the PGDBF, which will be described in the following subsection.

2.1.2 GDBF and PGDBF Analysis

The Gradient Descent formulation of Bit Flipping (BF) algorithm for the Binary Symmetric Channel (BSC) [31] sets a threshold for each VNU unit to determine whether the output of the VNU should be flipped or not based on an energy objective function. Energy objective is an integer value that varies between 0 and $d_v + 1$ and results with fewer number of flips in the successive iterations of the decoding

Algorithm 1 Gallager B

Initialization $i = 0, v_{n,m}^{(0)}(e)_{e \in \mathcal{N}(v_n)} \leftarrow r_n, n = 1, \dots, N.$
 $d_{n,m}^{(0)} \leftarrow r_n, n = 1, \dots, N.$
 $s = H\mathbf{v}^{(0)T} \bmod 2$
while $s \neq 0$ *and* $i \leq i_{max}$ **do**
 for $n = 1, \dots, N$ **do**
 Compute
 $c_{m,n}^{(i+1)}(e)_{e \in \mathcal{N}(c_m)}$ using Equation 2.1.3
 $v_{n,m}^{(i+1)}(e)_{e \in \mathcal{N}(v_n)}$ using Equation 2.1.1
 $d_{n,m}^{(i+1)}$ using Equation 2.1.2
 end for
 $s = H\mathbf{v}^{(i+1)T} \bmod 2$
 $i = i + 1$
 end while
Output: $\mathbf{v}^{(i)}$

process. Due to the integer representation of energy function, several VNUs may share the same maximum of energy value resulting with several bits to be flipped in one iteration. This may induce a negative impact on the convergence of the algorithm [32]. The Probabilistic GDBF (PGDBF) has been proposed to flip the outputs of only a random number of those VNUs with the maximum energy value. Energy calculations for the GDBF and PGDBF are governed by expressions similar to Equations 2.1.1 and 2.1.2 [31], but they involve finding the maximum value across all VNUs in each iteration of the decoding process as illustrated in Figure 2.2. This gradient descent algorithm used in the PGDBF increases hardware complexity of PGDBF. On the other hand, the VNU for GaB can be implemented using majority logic and *xor* gates and does not require complex operations. Later in section 4.1.2, we will show that the maximum energy computation is the main bottleneck on the throughput performance of GDBF and PGDBF implementations.

In Figure 2.3, the evolution of hard decision LDPC decoding algorithms is presented in term of error correction performance. Figure 2.3 shows the FER performance of three decoding algorithms, GaB, GDBF, and PGDBF, based on simulations

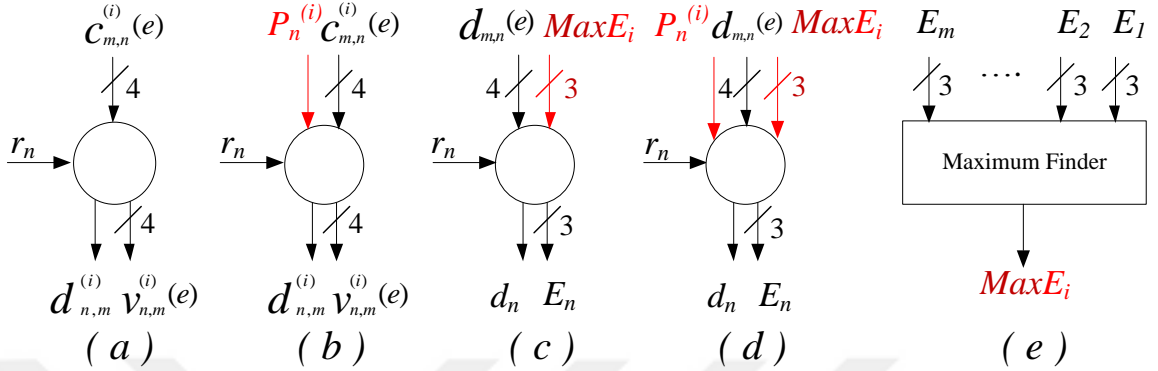


FIGURE 2.2. General architecture of VNUs for (a) GaB, (b) PGaB, (c) GDBF, and (d) PGDBF for $d_v = 4$ and $N = 1296$. (e) Maximum finder unit for GDBF and PGDBF decoders.

conducted for codeword length N of 1296 bits. In this figure, we show FER curve of the MinSum (MS) and Offset MinSum (OMS) [27] based decoder even though they are different class of decoder algorithm, where CNUs and VNUs exchange messages of multi-bit granularity, as opposed to the bit flip class of algorithms with single-bit granularity that are considered in this dissertation. We include the MS and OMS in the figure to set the stage on where the hard-decision (bit flip) based algorithms stand with respect to this best performing soft-decision decoder. As shown from the Figure 2.3, GaB is the worst performing among the four algorithms. Based on the scale and nature of the arithmetic operations involved during each iteration of the decoding process, GaB method is the most hardware friendly among the three algorithms, requiring combinational circuits at the scale of only 2-bit multiplication operations. Given the codeword length is N , the GDBF design requires N number of 3-bit maximum finder components, which returns the maximum of all. Additionally, the PGDBF design incorporates a 32-bit LFSR-based (Linear Feedback Shift Register) random number generator. We observe that as the complexity of the computation units increases, the performance of the decoding algorithm improves significantly compared to GaB. Unlike other methods, hardware realization of GaB has not been

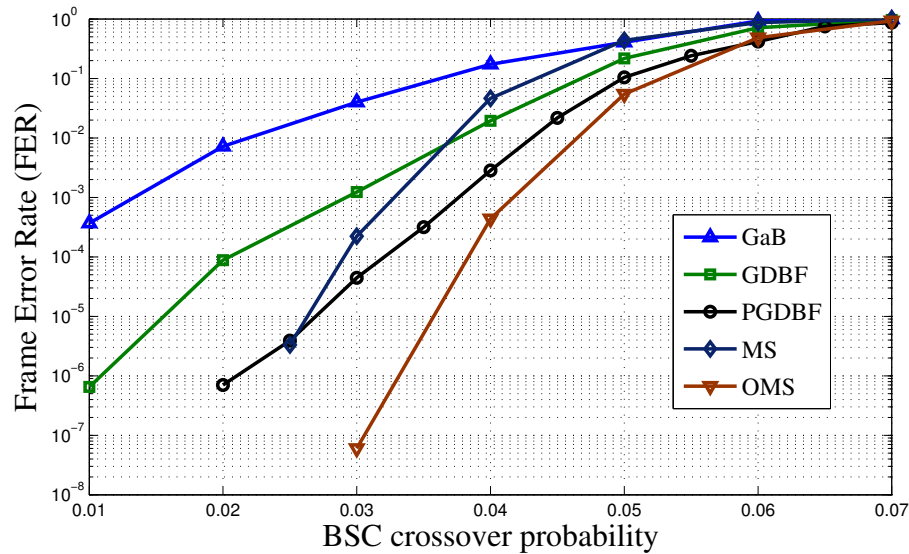


FIGURE 2.3. The evolution of decoding algorithms in term of error correction performance. GaB, GDBF, PGDBF, MS, and OMS FER comparison: FER vs. probability of error introduced to each bit of the 1296-bit codeword with $d_v = 4$, $d_c = 8$, $M = 648$, and Code Rate = 0.5.

desirable due to its poor decoding performance, and its application is limited to environments that require fast execution. Given that GaB offers speed advantage over the other methods, in this dissertation our aim is to answer the question of whether it is feasible or not to bridge the gap between GaB and better performing algorithms without sacrificing its throughput performance.

Chapter 3

PROBABILISTIC GAB ALGORITHM

In this chapter, we introduce a Probabilistic GaB (PGaB) algorithm by applying a probabilistic stimulation function over the iterative decoding process. We present the details of our incremental approach to designing and implementing the PGaB hardware architecture.

In order to improve the GaB decoding performance, we first analytically study the cases for which a message bit received from the channel becomes the determining factor in GaB for a decision made during the iterative decoding process. We then introduce an algorithm that disturbs those decisions with a predefined probability, which we refer to as p_v . We experimentally identify the p_v that results with preferable decoder performance.

In order to reduce the hardware cost and improve the throughput of the implementation, we first show that, rather than using a complex and hardware demanding random number generator, using a less sophisticated random number generator based on the linear feedback shift register (LFSR), which requires fewer hardware resources, is sufficient to improve the decoding performance. We then propose a heuristic that allows switching to PGaB only when GaB is not able to correct the errors in predetermined number of iterations. We investigate the impact of switching from GaB to PGaB at a specific iteration, which we refer to as s_i . We experimentally identify the s_i that results with preferable decoder performance, and show that when s_i is set to fifteen, we also drastically reduce the average iteration count by up to 62% compared to GaB.

3.1 Probabilistic GaB Algorithm Methodology

During the decoding process, the interactions between CNUs and VNUs may result in an oscillation phenomena due to the n^{th} order dependencies between CNUs and VNUs. In such cases, the decoding process may get trapped in a cyclic behavior. For example, in the Tanner graph [43] given in Figure 2.1, c_0 transmits message to v_1 and v_3 . After receiving their inputs from all CNUs, v_1 and v_3 send their messages back to their designated CNUs. In this example, there is a third order dependency between c_0 and v_0 based on the message passing in the order of $(v_0 - c_0 - v_5 - c_4 - v_3 - c_3 - v_2)$. If we count each CNU-VNU interaction as one iteration, then it would take three iterations for the message of c_0 to propagate to v_0 . Similarly, there is also a second order dependency in the order of $(c_0 - v_1 - c_1 - v_0)$. During each iteration, CNUs and VNUs update their states. The sequence of states observed for a given VNU may show repeating pattern, which is called a trapping set [44]. Trapping means that the decoder cannot correct the error, and then it remains in the cyclic sequences of states. One way to break this cyclic behavior is to disturb the VNU when such a pattern is detected. One may introduce large memory to keep track of the states, but that would not be hardware friendly, since the trapping set size is unknown and there can be many thousands of different trapping sets. Therefore, we randomly disturb the state of each VNU to be able to escape from the trapping set. Of course, one may question that such disturbance could adversely affect the normal behavior of the VNU, but theoretical results indicate that this side effect does not significantly increase the number of iterations [45], [46], [47]. If the GaB decoder does not converge within user-defined number of (k) iterations, then we apply this probabilistic strategy (Probabilistic GaB) to escape from trapping set. We modify Equation 2.1.1 by introducing a probability function $p_n^{(i)}$ as shown in Equation 3.1.1. The PGaB flow is shown in Algorithm 2.

$$v_{n,m}^{(i)}(e) = \begin{cases} 1, & \text{if } p_n^{(i)} \oplus r_n + (\sum_{e' \in \mathcal{N}(v_n) \setminus e} c_{m,n}^{(i)}(e')) > b_n \\ 0, & \text{if } p_n^{(i)} \oplus r_n + (\sum_{e' \in \mathcal{N}(v_n) \setminus e} c_{m,n}^{(i)}(e')) < b_n \\ r_n, & \text{otherwise} \end{cases} \quad (3.1.1)$$

Algorithm 2 Probabilistic Gallager B

Initialization $i = 0$, $v_{n,m}^{(0)}(e)_{e \in \mathcal{N}(v_n)} \leftarrow r_n$, $n = 1, \dots, N$.

$d_{n,m}^{(0)} \leftarrow r_n$, $n = 1, \dots, N$.

$s = H\mathbf{v}^{(0)T} \bmod 2$

while $s \neq 0$ *and* $i \leq i_{max}$ **do**

Generate $p_n^{(i)}$, $n = 1, \dots, N$, from $\mathcal{B}(p_v)$.

for $n = 1, \dots, N$ **do**

Compute

$c_{m,n}^{(i+1)}(e)_{e \in \mathcal{N}(c_m)}$ using Equation 2.1.3

$v_{n,m}^{(i+1)}(e)_{e \in \mathcal{N}(v_n)}$ using Equation 3.1.1

$d_{n,m}^{(i+1)}$ using Equation 2.1.2

end for

$s = H\mathbf{v}^{(i+1)T} \bmod 2$

$i = i + 1$

end while

Output: $\mathbf{v}^{(i)}$

3.1.1 Determining How to Disturb the VNU

The truth table shown in Table 3.1 captures how we propose to modify the VNU function with an example on calculating only one of the output messages ($v_{n,m}^{(i)}(4)$). In this example we assume that the d_v is four where each VNU has five inputs including the received word (r_n) and four CNU messages ($c_{m,n}^{(i)}(1, 2, 3, 4)$). Since in this example we are calculating the message for the fourth output of the VNU, message ($c_{m,n}^{(i)}(4)$) received from CNU is not used in the calculation.

TABLE 3.1. Truth Table for PGaB Algorithm

Inputs for VNU				GaB	PGaB
r_n	$c_{m,n}^{(i)}(1)$	$c_{m,n}^{(i)}(2)$	$c_{m,n}^{(i)}(3)$	$v_{n,m}^{(i)}(4)$	$v_{n,m}^{(i)}(4)$
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	1	0
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

In GaB algorithm, $v_{n,m}^{(i)}(e)$ is calculated by Equation 2.1.1 and is illustrated in Table 3.1. CNU messages ($c_{m,n}^{(i)}(1, 2, 3)$) represent whether the previous decision of VNU is correct or not. We take a close look at the GaB VNU logic for the cases where there is a tie over the three inputs ($c_{m,n}^{(i)}(1)$, $c_{m,n}^{(i)}(2)$, $c_{m,n}^{(i)}(3)$) and the received message (r_n). In such cases, shown with rows in bold in Table 3.1, the VNU output is determined by the r_n input. We argue that when the decoder is stuck in the trapping set, we should not use the r_n as a determining factor. Looking closely, when we express $v_{n,m}^{(i)}(4)$ for the PGaB (column 5) of Table 1, we see that function is equivalent to $c_{m,n}^{(i)}(2) \cdot c_{m,n}^{(i)}(3) + c_{m,n}^{(i)}(1) \cdot c_{m,n}^{(i)}(3) + c_{m,n}^{(i)}(1) \cdot c_{m,n}^{(i)}(2)$, and shows that we

ignore the received message for all input scenarios. If we ignore the received messages completely, decoder will fail. If we force all VNUs to rely on the received messages from the channel for the tie cases, then for the trapping set cases the decoder may not converge. The decoder cannot ignore the received messages, however during the decoding we do not know which VNU is in the trapping set. For this we introduce a mechanism that selects a predefined percentage of VNUs to ignore the received message and operate as the PGaB column of Table 3.1. We refer to predefined percentage of VNUs as the p_v term in our implementation. The subset of VNUs that ignore the received message is randomly chosen based on the p_v value. In the following section we present our experimental approach for determining the value of p_v . The probability function can be applied to the decoder in various positions. For example, in the PGDBF decoder [48], [49] the probabilistic function is applied randomly during the final output decision of a VNU to decide whether to flip the channel value or not. In the Noisy GaB [50], the randomness effect acts arbitrarily on both messages exchanged mutually between VNU and CNU. The main objective of these these studies is to distract the decoder by adding noise. Our approach to utilization of randomness is different from these studies as we attempt to use randomness in a more deterministic way. Rather than disturbing the outcome of decisions made during each iteration, we incorporate randomness directly into the message computation only for the cases when a tie occurs among the received messages of a VNU. Our Monte-Carlo simulations show better decoding performance, for the studied LDPC codes, when we introduce randomness as a tie-breaker for the VNU function when computing only messages sent from VNU to CNU. In the following subsection we present our approach for determining the p_v value.

3.1.2 Determining the p_v Value

Before proceeding to the hardware implementation, we need to determine Bernoulli distribution p_v , which represents the probability of $p_n^{(i)}$ taking the value of 1 ($P(p_n^{(i)} = 1)$). This will indicate the proportion of VNUs that will be disturbed in the hardware architecture. In this case, setting p_v to 0 would mean no disturbance for all VNUs. We conduct experiments, as shown in Figure 3.1, for four values of channel crossover probability α (0.025, 0.02, 0.03, and 0.035) by sweeping the p_v between 0 and 1. We choose four α values in order to check the consistency on the FER performance. In the figure, x-axis shows the range of p_v being 1 and y-axis shows the frame error rate for the LDPC codeword length of 1296 with degrees of VNU and CNU set to 4 and 8 respectively. As shown in Figure 3.1, for the case of alpha 0.02, the simulation point labeled with A indicates no stochastic behavior (GaB) where p_v is 0 and the point labeled with B shows the case where all VNUs operate as PGaB where p_v is 1.

Based on the plots in Figure 3.1, we conclude that disturbing all VNUs results with an improvement over the GaB (point B). We observe two trends in the figure that reveal important insights for determining the p_v . As the p_v value reduces to 0.4, the FER is almost insensitive to this change for both α values. We also observe a flood region between 0.1 and 0.2 where FER performance is the best for both α values. Based on this observation we set the p_v value to 0.2 for the hardware implementation. This leads us to selecting a random number generator (RNG) for generating the p_v with Bernoulli distribution. Random number generators have been studied in terms of their quality and complexity in the literature extensively [51]. For example, the Park-Miller [52] algorithm is one of the high quality random number generators that relies on linear congruential method, which would require complex hardware components. Therefore this type of RNG even though generates strong random numbers is not hardware friendly. Our design choice favors simplicity with the objective of a light-weight decoder architecture in terms of its hardware resource

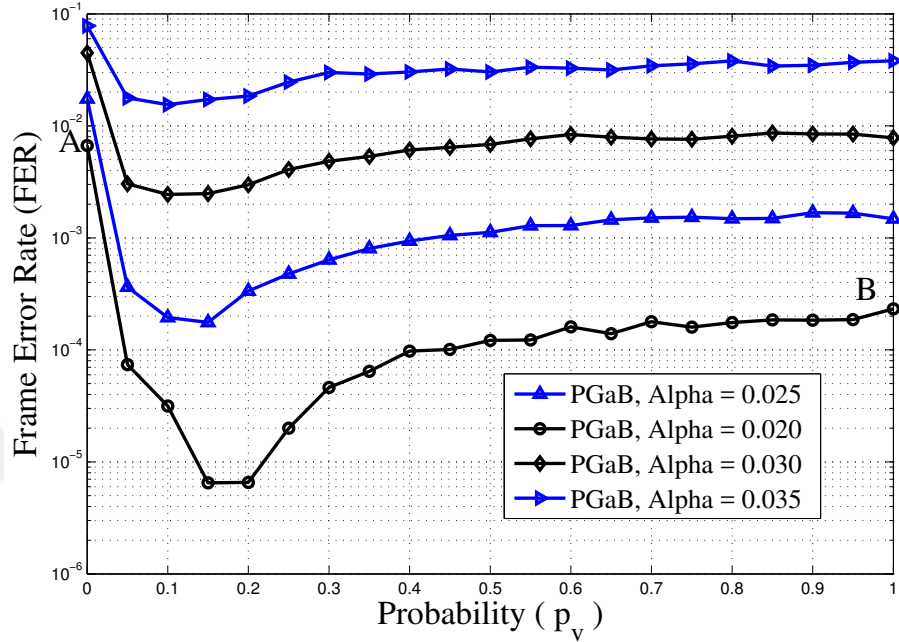


FIGURE 3.1. Frame Error Rate versus p_v ($\alpha = 0.02, 0.025, 0.03, \text{ and } 0.035$). LDPC code ($d_v = 4, d_c = 8, Z = 54$), ($N = 1296, M = 648$) when switching iteration s_i is set to 15.

requirement. We argue that, in our case there is no need for a sophisticated RNG in the hardware implementation that gives precise distribution for a given p_v . This is because, for all cases where p_v is set to a non-zero value, we observe improvement over the GaB and the preferable performance occurs in a window ranging between 0.1 and 0.2. We use this conclusion as basis for choosing a simpler and hardware friendly linear feedback shift register (LFSR) based RNG.

3.1.3 Determining the s_i Value

In section 3.1.1 we discussed the way we introduce probabilistic behavior to GaB to overcome trapping sets. Based on our simulations, we observe that GaB when successfully decodes a code, typically resolves the errors in less than ten iterations. Therefore we believe that a hybrid implementation that switches to PGaB only when GaB is not able to correct the errors in predetermined number of iterations would

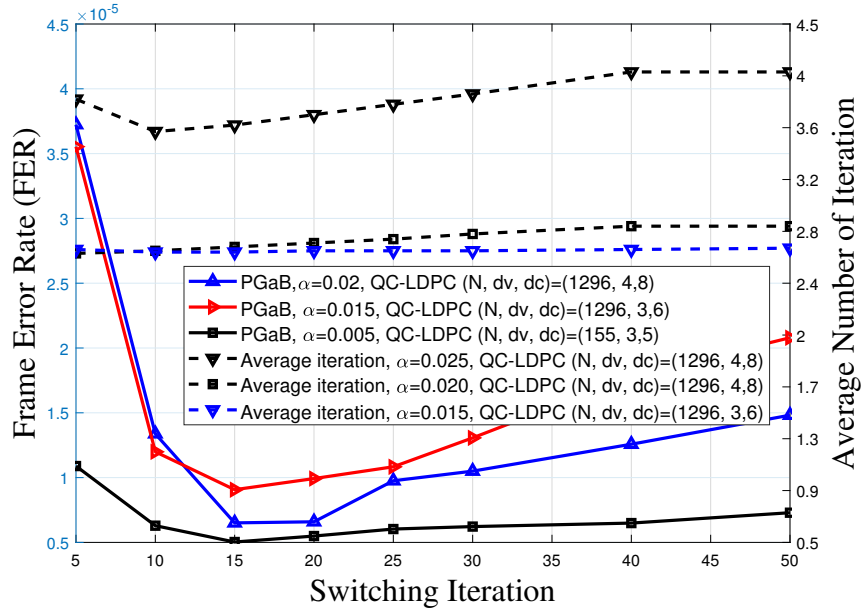


FIGURE 3.2. Frame Error Rate and Average iteration versus iteration number to switch from GaB to PGaB (QC-LDPC codes with (N, d_v, d_c, R) configurations of $(155, 3, 5, 0.5)$, $(1296, 3, 6, 0.5)$ and $(1296, 4, 8, 0.5)$).

be a better approach than executing only PGaB in terms of FER performance. We conduct two experiments to validate our claim.

In the first experiment, we evaluate the impact of switching from GaB to PGaB after a specific number of iterations (s_i) for three regular LDPC codes with (N, d_v, d_c, R) configurations of $(155, 3, 5, 0.5)$, $(1296, 3, 6, 0.5)$ and $(1296, 4, 8, 0.5)$. We vary the switching point from 5 to 50 and show the FER performance for different α values for each code shown in Figure 3.2. In the same plot we also plot the average number of iterations for three α values. For all experiments, we set the maximum number of iterations to 300. We evaluate the impact of change in codeword length on switching iteration using codeword length of 155 and 1296; and the impact of change in VNU and CNU degree using codes with connection degree d_v set to 3 and 4, and d_c set to 5, 6, and 8. All of these experiments indicate that setting the switching point between 15 and 20 iterations would be preferable for achieving better FER performance. Since

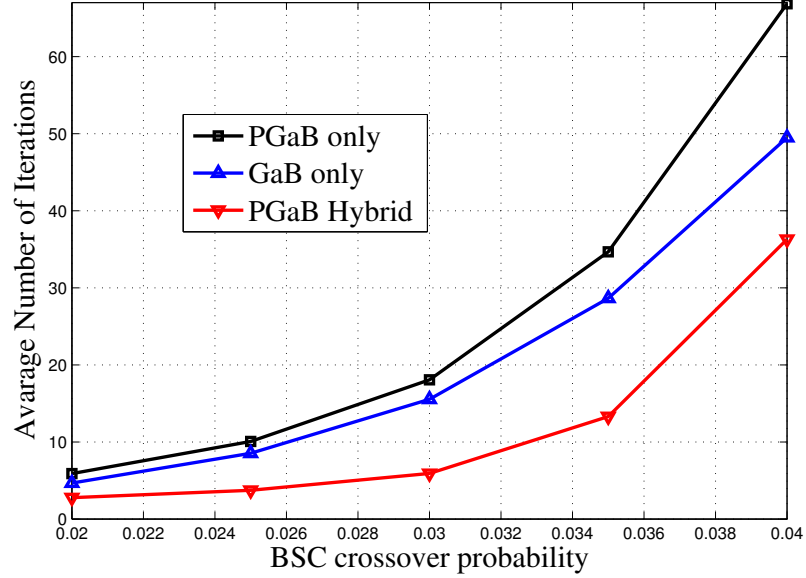


FIGURE 3.3. Comparison of average number of iterations for PGaB, GaB, PGaB Hybrid (GaB for the first 15 iterations, and PGaB onwards) ($d_v = 4, d_c = 8, N = 1296, M = 648$).

the average number of iterations for three α values show an increasing trend as the switching point moves from 5 to 50, we conclude that 15 is the ideal point to make the switching from GaB to PGaB. A side benefit of switching after 15 iterations is the reduced power consumption since we don't use PGaB for all iterations and in hardware implementation we turn on the the RNG unit only after iteration count 15 has been reached.

In the second experiment, we set the switching point to 15 and p_v value to 0.2, and evaluate the impact of disturbing VNUs on average number of iterations. In Figure 3.3, we compare average number of iterations for the baseline GaB, the PGaB, and the hybrid implementation that relies on the execution of GaB for the first 15 iterations of the decoding process and PGaB afterwards. In the figure, x-axis shows the α range and y-axis shows the average number of iterations for three simulations. Figure shows that when we start using the PGaB after 15 iterations, the average number

of iterations is always better than the deterministic GaB. We reduce the average iteration count by 40%, 56%, 62%, 54%, and 26% compared to the GaB for the α values studied in this experiment respectively. The PGaB only approach consistently results with larger number of iterations compared to the GaB only method. Disturbing the decoder starting with the first iteration results with adding more noise and therefore leads to increase in the average number of iterations. The hybrid PGaB on the other hand reduces the average number of iterations consistently with respect to the GaB only method. We believe that disturbing the GaB decoder after 15 iterations helps resolve some of the trapping set cases as shown theoretically by Ivanis and Vasic [50], which contribute to the increase in average number of iterations for the GaB only method. Reducing the maximum iteration count has also direct impact on the power consumed by the decoder. By reducing average iteration number, we also increase the throughput of the decoder. PGaB spends fewer iterations in average compared to the GaB to correct the errors.

In the following chapter we will present our hardware results and decoding performance based on the hybrid PGaB implementation where we set p_v value to 0.2 and the s_i to 15. For the remainder of this dissertation we refer to the hybrid PGaB as the PGaB. We will compare the decoding performance of the PGaB with GaB, GDBF, PGDBF, and MinSum based on the FER performance for each code studied in this dissertation. We will show that the PGaB results with up to four orders of magnitude decoding performance improvement over the GaB.

Chapter 4

HARDWARE DESIGN OF LDPC DECODERS

In this chapter, we present the details of hardware implementations for GaB, PGaB, GDBF, and PGDBF LDPC decoders. We show that GaB architecture delivers the best throughput while using fewest FPGA resources, however performs the worst in terms of decoding performance. We compare the decoding performance of the PGaB with GaB, GDBF, and PGDBF based on the FER performance. We show that the PGaB results with up to four orders of magnitude decoding performance improvement over the GaB, exceeding the performance of GDBF over the codes studied in this dissertation, with a negligible loss (less than 1%) in throughput performance compared to the GaB. We conclude that the PGaB is able to bridge the gap between GaB and complex decoding algorithms such as GDBF and PGDBF without sacrificing the throughput advantage of the GaB by consistently exceeding FER performance of the GDBF.

4.1 Hardware Design

4.1.1 GaB and PGaB Hardware Design

We first show the generic architecture for GaB and PGaB with VNUs, CNUs and the H matrix based on a regular QC-LDPC code for codeword length (N) in Figure 4.1(a). The *Compute Syndrome* unit in the figure checks whether all of the CNUs are satisfied or not.

We show the details of the VNU architecture for d_v equals to 4 in Figure 4.1(c). The colored arrows along with the *and* and *xor* gates in the VNU architecture are used by the PGaB implementation. When d_v is set to 4, besides the control input

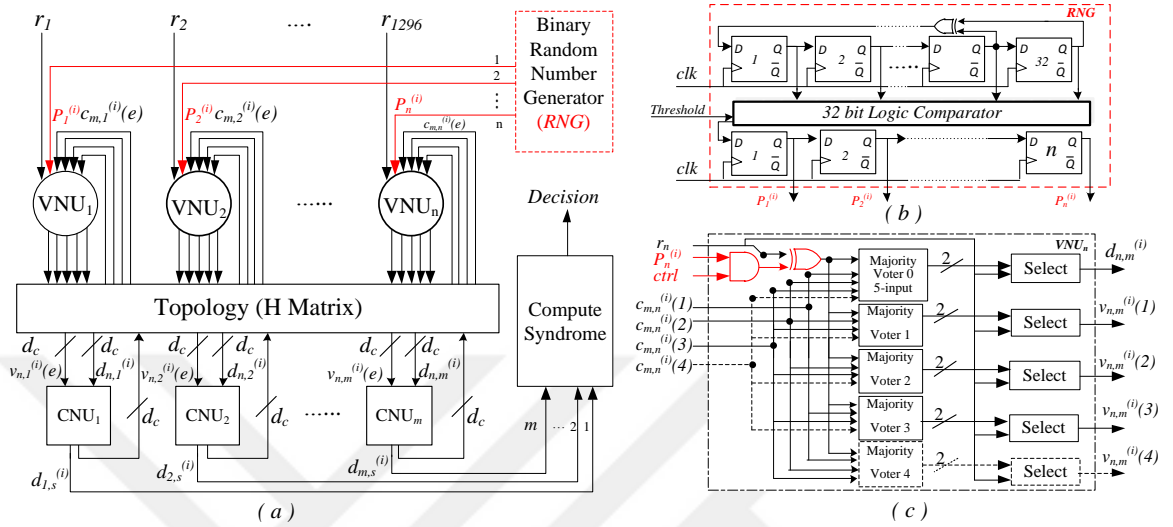


FIGURE 4.1. Overall decoder architecture (a) for PGaB, VNU architecture for $d_v=4$ (c) and LFSR-32bits based Random Number Generator (b).

(*ctrl*), there are six 1-bit inputs for each VNU. The first two inputs from top to bottom are 1-bit data received from the channel (r_n) and 1-bit random value ($P_n^{(i)}$) generated by the LFSR based RNG. Remaining four inputs ($c_{m,n}^{(i)}(1), c_{m,n}^{(i)}(2), c_{m,n}^{(i)}(3), c_{m,n}^{(i)}(4)$) are the 1-bit messages received from CNUs (as $d_v = 4$). There are four 4-input majority voter units (labeled as 1-4) and one 5-input majority voter unit (labeled as 0). The majority voter generates a 2-bit output representing majority of 1s, majority of 0s, or tie cases. The *select* unit acts as a selection of the majority output, which generates 1 for the majority of 1s case, and 0 for the majority of 0s case. In the case of a tie, the *select* unit passes the received word to its output. In this generic architecture, if the d_v changes than the number of *Majority Voters* and the number of inputs need to be adjusted properly. For example, when d_v is set to three, the input ($c_{m,n}^{(i)}(4)$), output ($v_{n,m}^{(i)}(4)$), and components (*Majority Voter 4* and *Select* for $v_{n,m}^{(i)}(4)$ output) marked with dotted lines are excluded from the VNU. We implement a regular majority voter based on Table I. We do not show the details of the *Majority Voter* architecture, since it is straightforward to implement. The VNU

operation is modified with the red marked lines and glue logic to adopt its function to PGaB. The control bit (*ctrl*) sets the first input of the *xor* gate to 0 if the algorithm is GaB, in which the *xor* gate passes the r_n input to its output, otherwise the output becomes a function of r_n and the $P_n^{(i)}$ for implementing the PGaB. A state machine controls switching between GaB and PGaB. After iteration number 15, if the decoder does not converge, the control bit (*ctrl*) is set to 1 by the state machine to switch to PGaB. The controller allows us to use VNU architecture of GaB to implement PGaB. Based on our conclusion about the RNG type to utilize in Section III.B, we implement a regular LFSR based RNG, shown in Figure 4.1(b) to feed a 1-bit random value to each VNU. We implement 32-bit LFSR to generate a 32-bit random number. The *32 bits Logic Comparator* compares 32-bit random number with the user defined *Threshold* value determined in Section 3.1.2. Finally, the output of the comparator, a one bit random number, is stored in the shift register. If the codeword length is N , then the RNG will take N number of cycles to generate the bits needed by all the VNUs. This N cycle overhead is applied only once during the first iteration of the decoding. During the subsequent iterations between the CNUs and VNUs, we simply generate one bit and use a shift register of size N to distribute the values to each VNU.

We do not show the details of the CNU architecture, since it is straightforward to implement. When d_c is set to 8, inputs are eight bit messages $v_{n,m}^{(i)}(e)$, and eight bit decision information ($d_{n,m}^{(i)}$) received from the VNUs. The outputs of a CNU are an eight bit message ($c_{m,n}^{(i)}(e)$) and one bit decision information ($d_{m,s}^{(i)}$). The $d_{m,s}^{(i)}$ is the output of the *xor* operation on the 8-bit input $d_{n,m}^{(i)}$. Decision information is sent to the *ComputeSyndrome* unit to decide whether the decoder has converged or not. Message calculation is different than decision information calculation as it is executed in an extrinsic manner. The $c_{m,n}^{(i)}(e)$ is calculated by Equation 2.1.2. For instance, $c_{m,n}^{(i)}(1)$ is determined by calculating the *xor* of messages $v_{n,m}^{(i)}(2), \dots, v_{n,m}^{(i)}(8)$ and excludes the $v_{n,m}^{(i)}(1)$. In summary, the CNU implementation requires one 8-bit

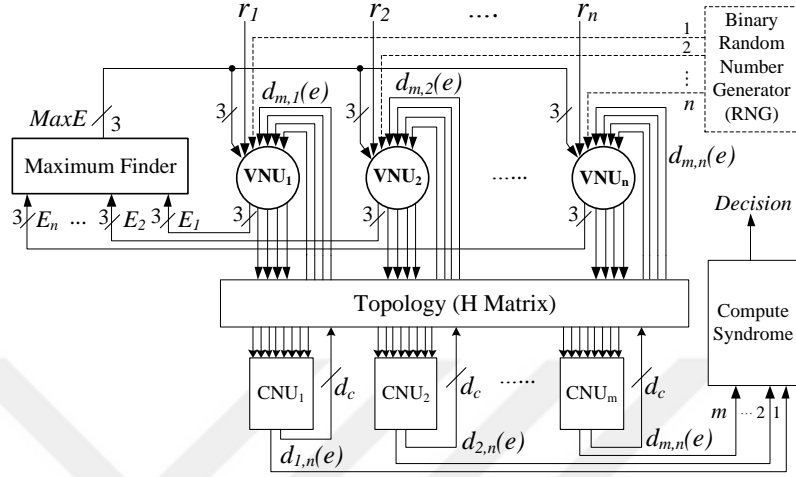


FIGURE 4.2. Architecture of GDBF and PGDBF for $d_v = 4$, and $N = 1296$, where d_c is determined by the code rate.

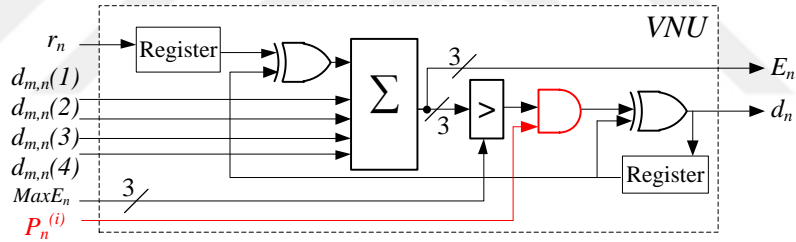


FIGURE 4.3. Architecture of VNU for GDBF and PGDBF for $d_v = 4$, and $N = 1296$.

xor gate and eight 7-bit xor gates.

4.1.2 GDBF and PGDBF Hardware Design

In order to present a comprehensive analysis on the decoding and hardware performance of PGaB, we implement two hard-decision based algorithms (GDBF and PGDBF). In this section, we present hardware implementations for these two algorithms. High level architectures for the GDBF and PGDBF decoders are shown in Figure 4.2 for a QC-LDPC code with codeword length of 1296 bits ($d_v = 4$, $d_c = 8$). The only difference between GDBF and PGDBF architectures is the binary RNG indicated with the dotted lines in Figure 4.2. The *RNG* generates 1296 binary 1-bit

TABLE 4.1. Hardware Resource Utilization, Throughput and Clock Rate of Decoding Algorithms Implemented for Tanner Code on Virtex6 FPGA

Algorithm	1-bit register	Slice LUTs	Fmax (MHz)	Throughput (Mbps)
GDBF* [48]	946	2151	132.7	4114.3
PGDBF* [48]	9161	3545	135.6	4202.5
MinSum [48]	13694	15350	237.2	197.5
GDBF	502	1630	137.5	4263.4
PGDBF	687	1802	138.2	4285.8

random numbers ($P_n^{(i)}$) to distribute to each VNU. Detailed architecture for the VNU is shown in Figure 4.3. The 1-bit received message (r_n) from the channel, the 1-bit decision estimations from the four CNUs ($d_{m,n}(e)$), and the 3-bit maximum energy value for the current iteration ($MaxE_n$) are common inputs for the VNU in GDBF and PGDBF. The summation operation in the VNU calculates the output energy value (E_n), which can be between zero and five. Therefore bit-width for the E_n and $MaxE_n$ are set to three bits. The *Maximum Finder* unit shown in Figure 4.2 computes the maximum of the E_n values received from each VNU in the current iteration i labeled as $MaxE_n$ in the figure. Each VNU uses the $MaxE_n$ and E_n to generate a 1-bit decision value (d_n). In the same iteration, if the E_n of a VNU is equal to the $MaxE_n$, then the output message d_n is flipped. If the E_n is less than the $MaxE_n$, then the d_n is not flipped. Additionally, for the PGDBF, a VNU receives a 1-bit random value generated by the LFSR based RNG ($P_n^{(i)}$). The d_n is a new message for all CNUs connected to the VNU. This iterative process continues till all CNUs are satisfied. A 1-bit message is sent by CNU to *Compute Syndrome* unit indicating whether a CNU it is satisfied or not. A state machine controls the *Compute Syndrome* unit to make a decision on whether the decoder has converged or not.

The hardware implementations for the GDBF and PGDBF have been studied

Algorithm 3 Simulation flow for generating FER plots

Input : *Decoding Algorithm* (GaB, PGaB, GDBF, PGDBF), *Codeword length* of 1296 (code rate 0.5 and 0.75) and *Codeword length* of 2212 (code rate 0.857) and *Crossover Probability* (α)

Output : FER plot of each algorithm over α

```

1 foreach Decoding Algorithm do
2   foreach Code do
3     FrameCounter = 0;
4     foreach  $\alpha$  do
5       #  $\alpha \in [0.001, 0.07]$  for  $N = 1296$ , rate 0.5
6       #  $\alpha \in [0.02, 0.03]$  for  $N = 1296$ , rate 0.75
7       #  $\alpha \in [0.005, 0.6]$  for  $N = 2212$ , rate 0.857
8       ErrorCount = 0;
9       while (ErrorCount < 100) do
10        Generate a random codeword (Frame);
11        FrameCounter = FrameCounter + 1;
12        Add noise to Frame using  $\alpha$ ;
13        Value = Decoding Algorithm();
14        # Value from Compute Syndrome
15        if (Value == 0) then
16          | ErrorCount = ErrorCount + 1;
17        end
18        if ErrorCount == 100 then
19          | FER = 100/FrameCounter;
20        end
21      end
22      Mark FER for  $\alpha$  on FER plot;
23    end
24  end
25 end

```

based on the Tanner code ($N = 155, M = 93, d_v = 3, d_c = 5$) in [48]. We first implement these two algorithms based on the same code and compare their hardware resource usage and throughput performance with the published results (indicated as * in the table) on the Xilinx Virtex6 FPGA using Table 4.1. With this comparison, our aim is to show that our implementations form a credible baseline for our extensive performance evaluations in the following section across GaB, PGaB, GDBF and PGDBF over various code lengths and code rates. We include MinSum in the table just to highlight the hardware efficiency of the hard-decision based algorithms with respect to this best performing soft-decision decoder. As shown in the table, we reduce the 1-bit register usage significantly by 92% compared to the PGDBF*. We also reduce the Slice LUT usage by 24% and 49% with our implementations of the GDBF and PGDBF respectively. The study by Le et. al. [48] reveals limited amount of information about the hardware implementation approach for the GDBF* and PGDBF*. We believe that there are two factors contributing to significant reduction on resource usage for our implementations. First, our CNU implementation does not require any register, as we implement it as a combinatorial logic and each CNU sends its output message $d_{m,n}(e)$ back to the VNU without having to store it. Secondly, we take advantage of a resource efficient implementation of maximum finder logic as shown by [53] based on parallel tree structure for calculating the $MaxE_n$. Earlier we claimed that maximum finder unit was a critical factor on throughput performance of the GDBF. When we replaced the "Maximum Finder" logic with a hard coded maximum value in our version of the GDBF implementation, we observed a reduction in logic block resource usage by 14.7% and an increase in the maximum clock rate by a factor of 2.99x for this hypothetical implementation. Nevertheless, with our implementation by reducing the resource usage for GDBF and PGDBF significantly with a slight improvement in the maximum clock rate, we are setting a tighter constraint on measuring the hardware performance in terms of resource usage and throughput for the PGaB implementation.

4.2 Simulation Environment

Our simulation environment includes the GaB, PGaB, GDBF, and PGDBF implementations in C programming language. The simulation flow is shown in Algorithm 3. We evaluate the impact of change in code rate on performance using codeword length of 1296 [54] with rates of 0.5 and 0.75; and the impact of change in codeword length using codeword length of 2212 [55]. For all algorithms, the d_v is equal to 4. For the FER analysis we include the FER performance of the flooding scheduling MinSum (MS) [26] and Offset MinSum (OMS) based decoders even though they belong to a different class of decoder algorithm, where CNUs and VNUs exchange messages of multi-bit granularity, as opposed to the bit flip class of algorithms with single-bit granularity that are considered in this dissertation. We include MS and OMS just to set the stage on where the hard-decision (bit flipping) based algorithms stand with respect to these best performing soft decision decoders. The MS and OMS used in this work are the quantized decoders with 4 bits for passed messages and 6 bits for A posteriori Log Likelihood Ratios (AP-LLR). We set the number of iterations to 20, the channel gain factor to 2, and the offset factor to 1 for OMS. We design and implement each decoder for each code (total of 12 architectures) on the Xilinx Virtex-6 FPGA (vc6vlx240t-2ff1156) and conduct post placement and routing analysis over hardware cost in terms of logic and register usage, and hardware performance in terms of maximum clock rate, and throughput. For each algorithm, FER curves are plotted as a function of the cross-over probability (α) over the BSC channel based on the simulation flow shown in Algorithm 3. Similar to other studies ([26], [48]), we calculate the system throughput using Equation 4.2.1. All designs have been implemented in VHDL. Functional verification is conducted by validating iteration by iteration post-routing CNU and VNU values against the C equivalent bit accurate implementation. We implemented the PGaB hardware architecture after completing a preliminary analysis and confirming the decoding performance of the

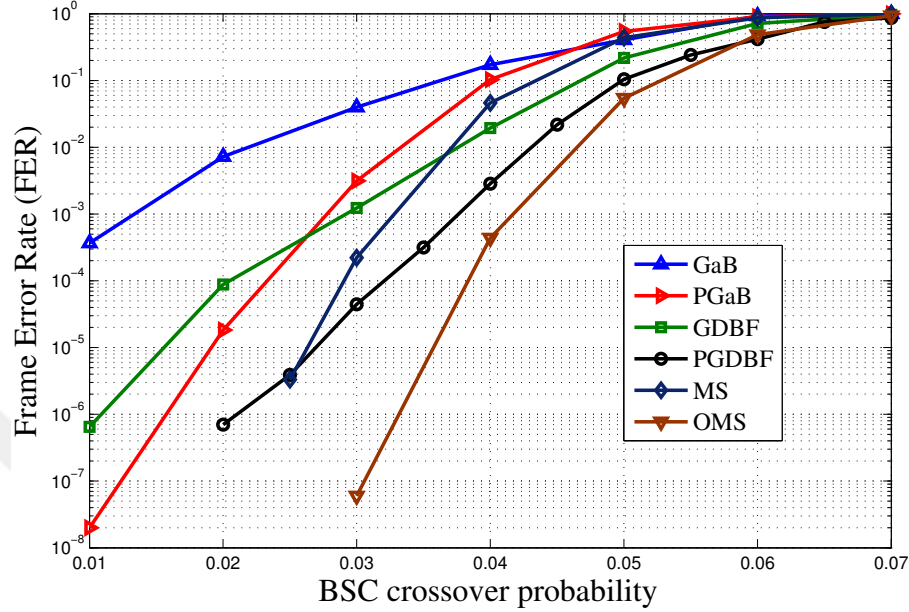


FIGURE 4.4. GaB, PGaB, GDBF, PGDBF, MS, OMS FER comparison: FER vs. probability of error introduced to each bit of the 1296-bit codeword with $d_v = 4$, $d_c = 8$, $M = 648$, and Code Rate = 0.5.

PGaB based on the FER plots generated using the C simulation. We measured the total simulation time for PGaB on codeword length of 1296 (code rate 0.5) as 116 days on the Intel Xeon (2.33GHz, 8GB RAM) processor. The same simulation takes slightly over 5 minutes on our FPGA based testbed. Therefore, for certain cross-over probability values, since the simulation times for the C code are extremely long, we used the FPGA based simulations to generate the points on the FER plots. For example, in the case of cross-over probability value of 0.01, we reached up to processing 10^{10} codewords with PGaB to generate the point that represents the 10^{-8} frame error rate.

$$\text{SystemThroughput} = \frac{\text{CodeLength} \times \text{MaxClockRate}}{\text{AvgIteration} \times \text{CyclesPerIteration}} \quad (4.2.1)$$

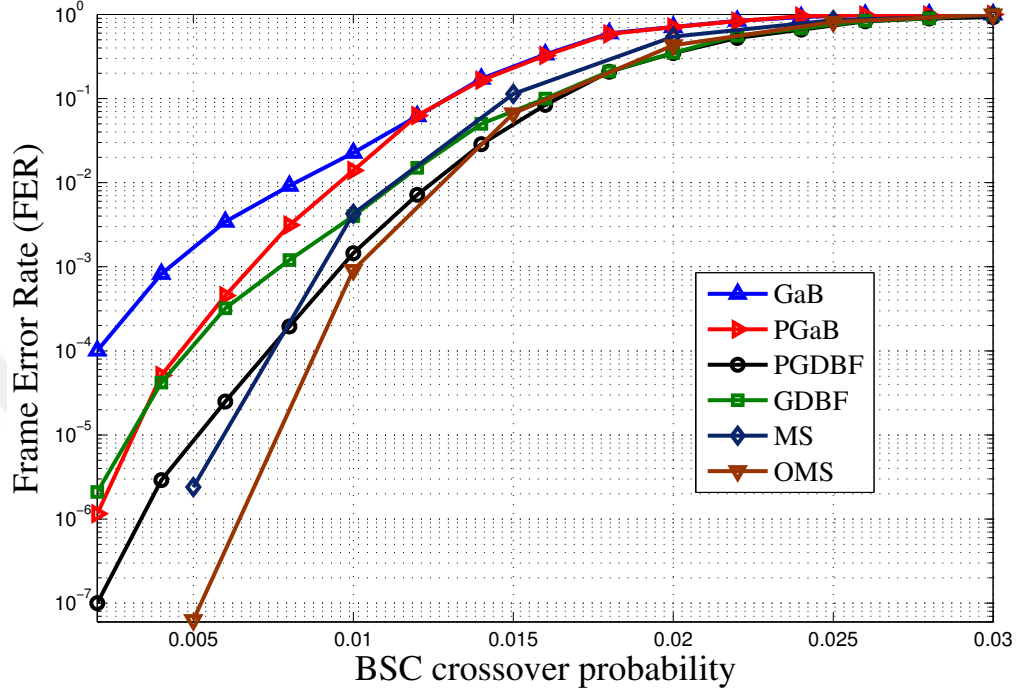


FIGURE 4.5. GaB, PGaB, GDBF, PGDBF, MS, OMS FER comparison: FER vs. probability of error introduced to each bit of the 1296-bit codeword with $d_v = 4$, $d_c = 16$, $M = 324$, and Code Rate = 0.75.

4.3 Performance Analysis

Figure 4.4 shows the FER performance comparison between the GaB, PGaB, GDBF and PGDBF algorithms as a function of the cross-over probability over the binary symmetric channel (BSC) on LDPC code with the rate 0.5. This chart shows that, with the probabilistic execution, we are able to bridge the gap between the GaB and the better performing decoding algorithms through PGaB. Another remarkable conclusion is our ability to perform better than the GDBF with the PGaB. The gap between PGaB and GaB in the error floor region where α is 0.01 quantifies the dramatic improvement (up to four orders of magnitude) achieved by disturbing randomly the state of the decoder.

In Table 4.2, we present the resource usage, maximum clock rate, and throughput

TABLE 4.2. Resource usage of GaB, PGaB, GDBF, and PGDBF based on the FPGA implementations for QC-LDPC $(N, d_v, R)=(1296, 4, 0.5)$, QC-LDPC $(N, d_v, R)=(1296, 4, 0.75)$, and QC-LDPC $(N, d_v, R)=(2212, 4, 0.857)$. (% indicates the difference with respect to GaB)

Codeword	1-bit Register			Slice LUTs		
	R=0.50	R=0.75	R=0.857	R=0.50	R=0.75	R=0.857
GaB	7812	4596	13308	11784	6097	23292
PGaB	9141	5601	15552	14605 (24%)	7133 (17%)	28895 (24%)
GDBF	3923	3923	6871	14822 (26%)	12024 (97%)	25037 (7%)
PGDBF	5251	5251	8915	16091 (37%)	15224 (150%)	29613 (27%)

TABLE 4.3. Maximum clock rate and throughput of GaB, PGaB, GDBF, and PGDBF based on the FPGA implementations for QC-LDPC $(N, d_v, R)=(1296, 4, 0.5)$, QC-LDPC $(N, d_v, R)=(1296, 4, 0.75)$, and QC-LDPC $(N, d_v, R)=(2212, 4, 0.857)$. (% indicates the difference with respect to GaB)

Codeword	FMax (MHz)			Throughput (Gbps)		
	R=0.50	R=0.75	R=0.857	R=0.50	R=0.75	R=0.857
GaB	147	114	59	38224	29575	26410
PGaB	146 (-0.8%)	113 (0%)	59 (0%)	37900 (-1%)	29471 (0%)	26281 (0%)
GDBF	44 (-70%)	43 (-62%)	32 (-46%)	11423 (-70%)	11270 (-62%)	14322 (-46%)
PGDBF	45 (-70%)	41 (-62%)	32 (-46%)	11583 (-70%)	10765 (-64%)	14348 (-46%)

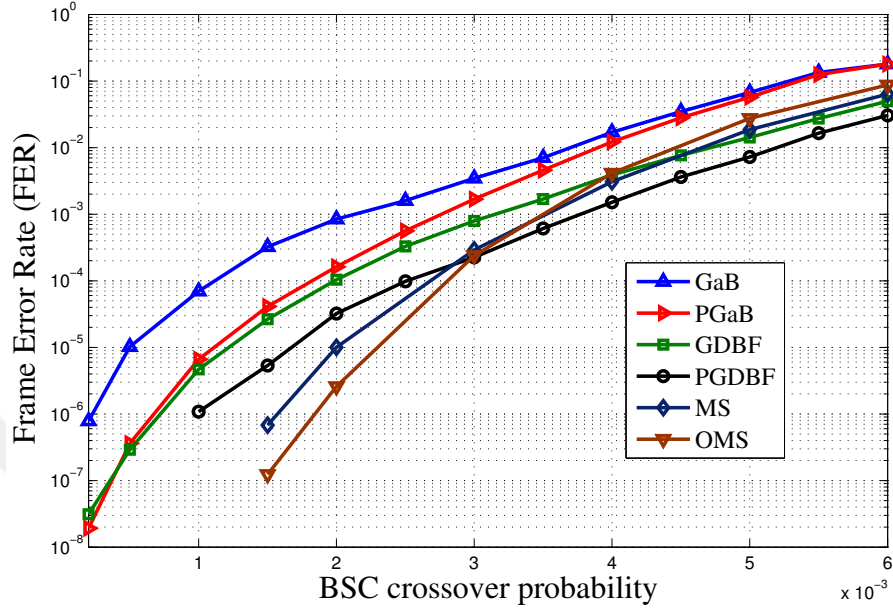


FIGURE 4.6. GaB, PGaB, GDBF, PGDBF, MS, and OMS: FER vs. crossover probability for the QC-LDPC code with ($d_v = 4, d_c = 28, N = 2212, M = 312$, and Code Rate = 0.857).

for the GaB and PGaB based on their FPGA implementations with 0.5 code rate. The percentage sign (%) in Table 4.2 and Table 4.3 indicates the change in resource usage, FMax, and Throughput for PGaB, GDBF, and PGDBF with respect to the GaB implementation. It takes 2 clock cycles to complete one iteration of the decoder, one cycle for VNU and one cycle for CNU. Average number of iterations and throughput of the decoder vary at different FERs. Throughput values in Table 4.3 are calculated based on the average number of iterations set to 2.5. With the probabilistic execution, the improvement in the error floor over the GaB comes with a negligible amount (0.85%) of loss in throughput performance. Even though modification to GaB involved including a RNG and an additional input to each VNU, the clock rate difference between the two designs is negligible. However, the decoding performance improvement is achieved with an increase on register and slice LUT usage by 17% and 24% respectively. Register overhead of the PGaB implementation includes the

1296 bits to store the 1-bit random number for each VNU and the 32-bit shift register to implement the random number generator. The increase in resource usage is a reasonable tradeoff for improving the decoding performance. The maximum clock rate for the PGaB is 3.3 times better than GDBF and PGDBF for this code rate. Since our optimized versions of the GDBF and PGDBF implementations do not use registers for the CNUs, PGaB has larger register foot print. However, the slice LUT (logic block) usage is comparable with PGDBF.

4.4 Robustness Analysis

In the following series of experiments we evaluate the impact of changes in code rate and codeword length on decoding performance and hardware cost over the four algorithms, and demonstrate that PGaB consistently outperforms the GDBF and PGDBF in terms of throughput. Code rate indicates the ratio of data to the length of the codeword that includes the data and parity bits. The higher the code rate, the higher the probability of noise over the communication medium effecting the data portion of the codeword. This increases the stress on the decoding algorithm on correcting errors. As the bandwidth for communications systems increase, the packet lengths (codewords) become longer. Therefore ability to process longer codewords encoded with higher code rates are important criteria for evaluating the efficiency of a decoding algorithm. Furthermore, a change in code rate or codeword length involves modification to the decoder hardware architecture and imposes routability and critical path delay constraints from FPGA implementation point of view. In this section, we summarize the hardware modifications, present performance analysis, and correlate resource requirement and throughput trends with respect to changes in code rate and codeword length.

4.4.1 Effect of Code Rate

In this experiment, we change the code rate from 0.5 to 0.75 when the codeword length remains as 1296 bits and the degree of a VNU is four. From hardware implementation perspective, the number of VNUs depend on the codeword length, whereas the number of CNUs and the number of connections per CNU (degree of CNU) depend on the code rate. With fewer number of CNUs, the number of connections per CNU increases. As the code rate increases from 0.5 to 0.75, based on Table 4.2, we observe that the register and Slice LUT resources are reduced by 39% and 51% respectively for the PGaB. Similar hardware resource usage trend can be noticed for the GaB. Cost of a single CNU implementation increases with 8 additional inputs to the summation operation (*xor* in Equation 2.1.2) since the degree of a CNU increases from 8 to 16. However, increasing the code rate to 0.75 reduces the number of CNUs from 648 to 324. Since the VNU operations are at *and* and *or* logic gate levels, the increase in CNU complexity is compensated by the reduction in the CNU count, which is the primary reason for reduction in the total logic block usage. Interestingly, the maximum clock rate for the new PGaB design is 113 MHz, which is 22% slower. We believe that the degree of the CNU is the primary reason for this performance loss. Placement and routing attempt to reduce the total wire length for a design by positioning the logic blocks closer and utilizing the flexibility of connection boxes and switch boxes on the FPGA to establish short paths for each net. Code rate of 0.75 results with a design that doubles the number of connections for each CNU. This in turn creates additional stress on routability, and the shorter wire segments available for the code rate of 0.5 are no longer available for the code rate of 0.75 due to congestion in regions that are densely populated with logic blocks. Therefore, nets take longer paths for routability and the critical path delay increases due to congestion.

Figure 4.7(a) and Figure 4.7(b) show the layouts obtained for PGaB over the two code rates. The area expansion of the design for code rate of 0.75 is due to the router

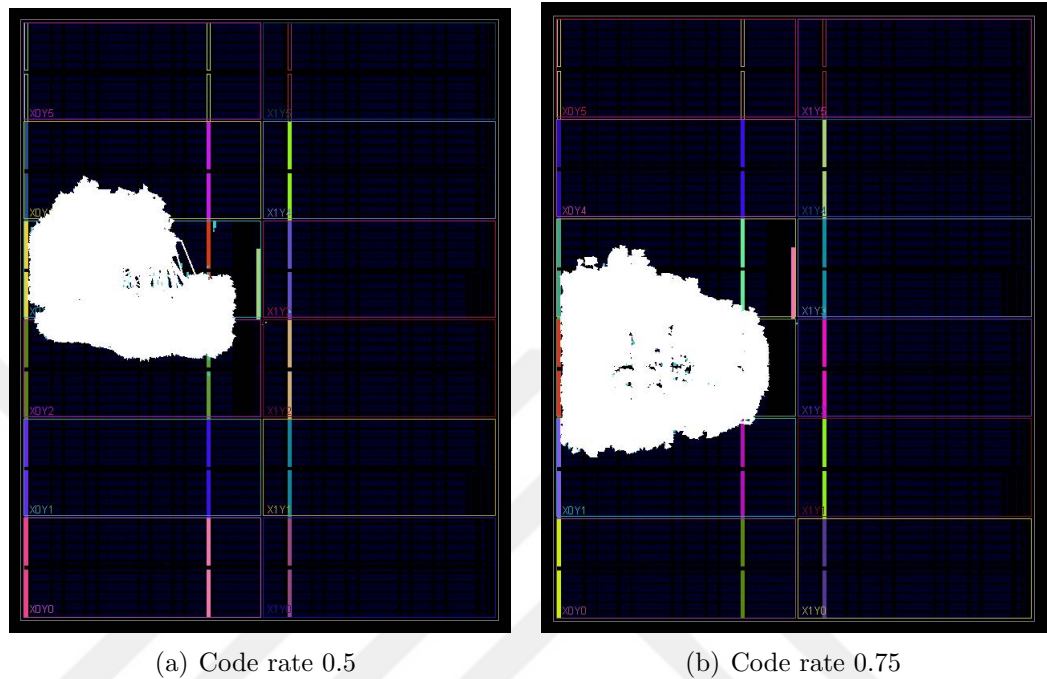


FIGURE 4.7. Post-routing layout for the PGaB designs with two code rates. a) PGaB decoder on the QC-LDPC code with ($d_v = 4, d_c = 8, N = 1296, M = 648$, and Rate = 0.5). b) PGaB decoder on the QC-LDPC code with ($d_v = 4, d_c = 16, N = 1296, M = 324$, and Rate = 0.75).

ripping up and rerouting nets through longer paths to avoid congestion. Even though the maximum clock rate is slower with the higher code rate, the user throughput shows 16% improvement over 0.5 code rate.

Based on Table 4.2, PGaB results with an increase in 1-bit register and Slice LUT usage by 17% and 24% respectively over GaB. Similarly, from Table 4.2, PGaB at 0.75 code rate shows an increase in 1-bit register and Slice LUT usage by 22% and 17% respectively over GaB.

Each signal that is generated by the CNU for GaB and PGaB implementations are stored in a register. On the other hand, in our GDBF and PGDBF implementations, the CNU does not include any register. Given that the number of connections per VNU remains the same, even though the code rate changes, there is no additional

TABLE 4.4. Throughput-to-area ratio (TAR) and Normalized Throughput (T_p) for GaB, PGaB, GDBF, and PGDBF when the crossover probability is fixed. Based on the FPGA implementations for QC-LDPC with $(N, d_v, R)=(1296, 4, 0.5)$, QC-LDPC $(N, d_v, R)=(1296, 4, 0.75)$, and QC-LDPC $(N, d_v, R)=(2212, 4, 0.857)$.

	$N, d_v, R=(1296,4,.5)$			$N, d_v, R=(1296,4,.75)$			$N, d_v, R=(2212,4,.857)$		
	$I_{ave} (\alpha = 0.02)$	$T_p (Mbps)$	TAR	$I_{ave} (\alpha = 0.002)$	$T_p (Mbps)$	TAR	$I_{ave} (\alpha = 0.001)$	$T_p (Mbps)$	TAR
GaB	3.37	28362	2.41	1.14	64857	10.64	1.12	58954	2.53
PGaB	2.78	34078	2.33	1.09	67594	9.48	1.09	60272	2.07
GDBF	3.70	7723	0.52	1.18	23888	1.99	1.16	30892	1.23
PGDBF	5.78	5011	0.31	1.62	16600	1.09	1.44	24885	0.84

register demand. Therefore the number of registers used by these implementations remain the same. We observe less than one percent reduction in Slice LUTs for the GDBF and PGDBF implementations with code rate of 0.75 over code rate of 0.5. This results with slight change in maximum clock rate and throughput performance. In overall, the PGaB implementation results with a maximum clock rate that is around 2.6 times better than GDBF and PGDBF.

In Figure 4.5, we compare the FER performance of the GaB and PGaB for the code rate 0.75. The PGaB consistently outperforms the GaB decoder especially in the error-floor region (more than two orders of magnitude at crossover probability of 2×10^{-3}). The PGaB catches the GDBF at α value of 0.04 and performs better than GDBF beyond this point. Considering the FER performance in the error floor region shown in Figure 4.5, and the negligible loss (0.004%) in throughput performance, we conclude that PGaB decoding and throughput performance is consistent across two code rates.

4.4.2 Effect of Codeword Length

Next, we implement the GaB, PGaB, GDBF, and PGDBF for the QC-LDPC code constructed by [55], which has a length of 2212 bits and $(d_v = 4, d_c = 28)$ with a higher code rate of 0.857. Each CNU has a degree of 28 for all four hardware

implementations. Note that the total number of inputs per CNU for the GaB and PGaB implementations are 56 as illustrated in Figure 4.1.

As shown in Table 4.2, resource usage for all design increases significantly compared to the implementations based on the codeword length of 1296 primarily due to the increase in the VNU count. The size of LDPC code affects the complexity of interconnection network [56]. This is reflected in the maximum clock rate for the PGaB implementation, which drops from 113.7MHz to 59.4MHz. However, the rate of the throughput loss is much smaller (by 10.8%), because, throughput is linearly proportional to the length of the codeword as shown in Equation 4.2.1 independent from the design. The longer codeword compensates for the reduction in clock rate. In overall for the longer codeword we observe that PGaB achieves higher clock rate and throughout performance with respect to the GDBF and PGDBF.

The slice count for an implementation is widely used as the area metric by FPGA researchers. In Table 4.4, we show the throughput-to-area ratio (TAR) based on slice count for each algorithm for each code rate. The PGaB consistently results with better TAR performance than both GDBF and PGDBF. Even though GaB and PGaB throughput performances are similar for each code rate, due to higher resource usage of the PGaB, the TAR performance is worse than the GaB. In Table 4.4, we also compare the normalized throughput (T_p) that is calculated based on fixing the α value for each code rate studied in this dissertation and using the average number of iterations for that α value. As seen in Table 4.4, average number of iterations for the PGaB implementation is consistently lower than GaB, GDBF and PGDBF implementations for each code rate. In Table 4.3, we notice that GaB resulted with better throughput than the PGaB implementation. However when we take the actual iteration number into account we observe that the PGaB achieves better throughput than the GaB and the throughput gap between PGaB implementation with respect to GDBF and PGDBF further improves.

Figure 4.6 shows the FER performance comparison between the GaB, PGaB,

GDBF, and PGDBF decoders as a function of the cross-over probability on QC-LDPC $(N, d_v, d_c, R)=(2212, 4, 28, 0.857)$ code with rate of 0.857. We conclude that for longer codewords the PGaB can surpass the GaB by more than one order of magnitude at crossover probability of 2×10^{-4} without sacrificing the throughput advantage of the GaB. We observe that, as the code rate increases and as the codeword length increases, the gap between GaB and PGaB on the FER plots shrinks. This trend is expected, since each case stresses the decoder. The important observation here is the superiority of the PGaB for all scenarios considered in this study in terms of decoding performance without sacrificing the throughput performance of the GaB.

Chapter 5

ROUTABILITY PROBLEM OF THE LDPC CODE

The connection intensive bipartite graph based LDPC decoder hardware architecture creates routing stress for longer codewords that are utilized in today's communications systems and standards. In this chapter, we study the routability problem of mapping LDPC codes on to the FPGA.

5.1 Background

LDPC decoder architectures utilize a large number of processing nodes and excessive amount of connections between these nodes. Routability can be a major challenge in the implementation of a decoder depending on the code rate ([33], [28], [57], [58], [59]). To the best of our knowledge, the routability challenge of error correction algorithm implementation has been studied in the context of FPGA domain only in [28]. In the VLSI domain there are studies describing various techniques to improve the routability. Even though these are not applicable to FPGA context, for the completeness of our related work here we give an overview of those techniques.

The study by [28] shows the implementation of decoder architectures for GaB and PGaB based on codeword length of 1296 over two different code rates. This study exposes the routability problem with respect to change in code rate through a visualization on the post-routing resource usage and discusses the impact of congestion on critical path delay.

In the VLSI domain, the study by [60] investigates the routing complexity problem for the Min-Sum algorithm, and introduces bi-directional routing network to reduce the number of connections and reorders the data stored in the memory so that a single connection is used for sending and receiving data. The solution is tailored for

the highly memory dependent Min-Sum algorithm and is not applicable to other class of algorithms.

[61] and [62] propose to change structure of the LDPC code for reducing the routing complexity. In these studies, the positions of the processing nodes are rearranged iteratively to limit the interconnect length and find a design with improved routability. This design space exploration type of approach requires extensive amount of experiments to identify a better configuration. More importantly, while routability problem is being addressed, the nature of the algorithm is inevitably modified, which effects error correction performance of the code.

5.2 Congestion Analysis

Routability can be a major challenge in the implementation of a decoder depending on the code rate ([33], [28], [57]). For the case of QC-LDPC with code rate of 0.5 ($d_v = 4$, $d_c = 8$) each CNU has eight connections. For a codeword length of 1106 bits that has a higher code rate of 0.857, there are 8848 input and output connections ($2 * 4 * 1106$ since $d_v = 4$). In this case, design requires 158 CNUs. Given that there are 8848 connections, each CNU has 56 connections. For the code rate 0.5, the number of input and output connections is 16. The significant amount of increase in the number of connections per CNU from 16 to 56 as the code rate increases is the primary source for the congestion problem. Furthermore, the positions of CNUs in the post-placement layout does not show regularity due to competition between pulling forces on a CNU by its VNUs and the pulling forces on those VNUs by their respective CNUs. The congested region is shown in Figure 5.1 “left”). The congestion problem due to increase in number of connections around a CLB is further amplified with the increase in codeword length ([63], [64], [65], [66]).

During packing stage of the FPGA CAD flow where LUT based netlist is transformed into Configurable Logic Block (CLB) based netlist, the synthesis tools attempt

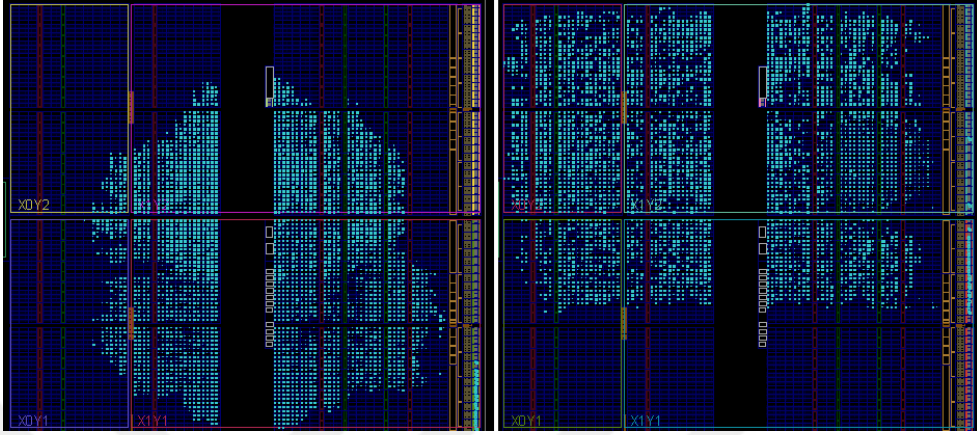


FIGURE 5.1. Post-routing layout of GaB decoder for the QC-LDPC code ($d_v = 4$, $d_c = 28$, $n = 1106$, and code rate = 0.857) on the Xilinx Zynq XC7Z020 FPGA for Regular (left) and Depopulation based implementation (right).

to pack as much logic as possible into a single CLB to minimize the number of CLBs. Filling the CLB to its capacity tends to increase the number of inputs per CLB. The increase in number of connections around a CLB in turn increases the routing demand in its peripheral region. Depopulation by not filling each CLB to its capacity results with spreading the logic across the FPGA and improves the routability of a design [67]. Based on this principle, we apply depopulation by partitioning the function of each CNU into two CNUs as illustrated in Figure 5.2, where first half of the inputs to the CNU are processed by the CNU_1 and the second half of the inputs are processed by the CNU_2 . In order to realize the CNU function, we merge the outputs of the two CNUs with an *xor* operation. In the case of QC-LDPC with code rate of 0.857, instead of implementing a 28-input *xor* based CNU function, we implement two 14-input *xor* based CNU functions along with a 2-input *xor* function. Figure 5.1 (“right”) shows the layout for the partitioning based implementation. Compared to Figure 5.1 (“left”), we observe that the congested regions disappear as we spread the logic across the FPGA.

In order to demonstrate the benefits of the partitioning method, we first study the impact of increase in codeword length and code rate on resource usage and crit-

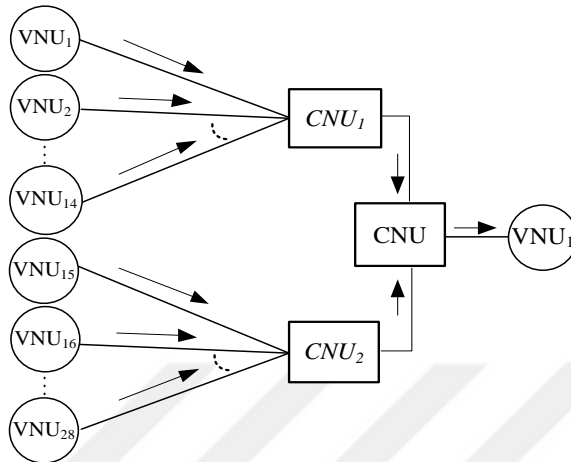


FIGURE 5.2. Two-way partitioning methodology for CNU with $d_c = 28$.

ical path delay in Sections 5.3.1 and 5.3.2 respectively. We show that two-way partitioning allows reducing the critical path delay with a negligible increase in LUT usage. More importantly, the unroutable designs for codeword lengths beyond 1106 becomes routable for the GaB algorithm with the two-way partitioning method. In section 5.3.3 we quantify the impact of scaling the partitioning level from two to four, six, and eight on *LUT usage - delay* product as a case study based on two different codewords.

5.3 Trend Analysis

In section 5.2, we exposed the congestion issue based on a single codeword length with a layout comparison between the regular and depopulation based GaB implementations. In this section, we expand our analysis over GaB to the effect of increasing the codeword length on critical path delay and resource usage. We show that routing congestion becomes a bottleneck for longer codewords making the designs not feasible to implement even on resource rich FPGAs, and depopulation resolves this problem.

5.3.1 Codeword Length and Code Rate vs. Resource Usage

In the first experiment, we sweep the codeword length from 324 to 1296 for the QC-LDPC code with degrees of VNU and CNU set to 4 and 8 respectively, where code rate is 0.5. Figure 5.3 shows the resource usage trend in terms of LUT usage (*y-axis*) with respect to the increase in codeword length (*x-axis*) for regular and depopulation based implementations. From hardware implementation perspective, as explained in Section 6.2, the number of VNUs depends on the codeword length and the number of CNUs depends on both the codeword length and code rate. Therefore in the plot, we observe that the resource usage increases linearly as the codeword length increases.

For the codeword length of 1296, the decoder consumes 22% of LUT resources on the Xilinx Zynq XC7Z020 FPGA. As shown in Figure 5.3, resource usage for the depopulation based implementation is slightly higher than the regular implementation, and the gap is less than 9% for any codeword length studied in this experiment. This gap remains the same when we map the decoders onto Virtex-7 XC7VX485T FPGA. Even though we double the number of CNUs with the depopulation based implementation, LUT usage by the design increases slightly as the number of LUTs required to implement the CNU function remains the same. A single CNU implementation requires 10 LUTs. However, for the two way partitioning, each CNU requires 6 LUTs, resulting with a total of 12 LUTs for the equivalent CNU functionality. This is the source of increase in LUT usage by 9% with the depopulation technique.

We run a second experiment to evaluate the impact of code rate increase on resource usage. In this experiment, we choose a QC-LDPC code with degrees of VNU and CNU set to 4 and 28 respectively, with a code rate increase from 0.5 to 0.857. We evaluate the LUT usage with respect to codeword length up to 1936 bits based on Figure 5.4, where *x-axis* shows the range of codeword length and *y-axis* shows the resource usage. Similar to the Figure 5.3, we observe a linear increase in resource usage for both implementations (regular and depopulation) as the length of

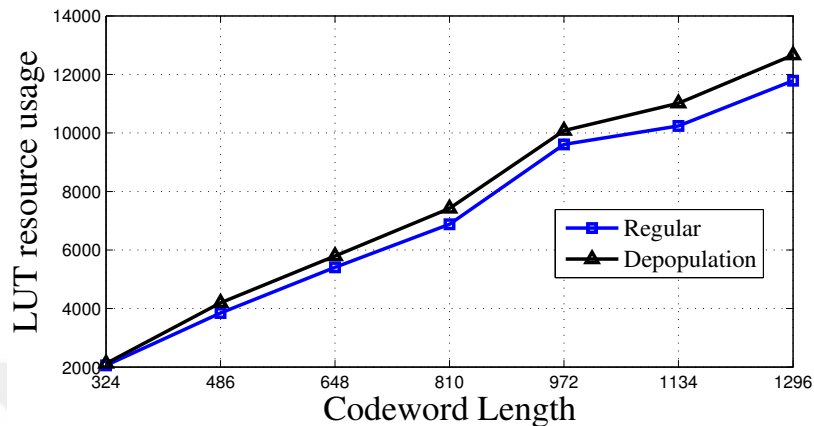


FIGURE 5.3. Resource usage vs Codeword length comparison for the QC-LDPC code with ($d_v = 4, d_c = 8$, code rate=0.5) based on Xilinx Zynq XC7Z020 FPGA.

the codeword increases for the first three points (553, 830, and 1106). The resource usage for the depopulation based implementation is 1% and 9% higher than the regular implementation for the codeword lengths of 830 and 1106 respectively. In parallel to our analysis on Figure 5.3, we attribute the increase in the gap between the two designs on this code rate from 1% to 9% to the increase in the number of CNUs coupled with the increase in the number of input connections per CNU.

Compared to the code rate of 0.5, the number of LUTs increase from 10 to 39 with the code rate of 0.857 due to increase in number of input ports per CNU from 8 to 28. The increase in LUT count combined with the increased number of connections with higher code rate, the regular based designs over the codeword lengths of 1383 and higher are not possible to implement even on the resource rich Virtex7 XC7VX485T FPGA as the routing stage fails.

In Table 5.1, we show the number of LUTs used as a routing resource for regular and depopulation based implementations for a QC-LDPC code ($d_v = 4, d_c = 28$, and code rate= 0.857). We observe that fewer LUTs are used as route-through with the depopulation technique for all codeword lengths. We believe this is a strong indicator of synthesis tool resorting to LUT resources in order to realize route through

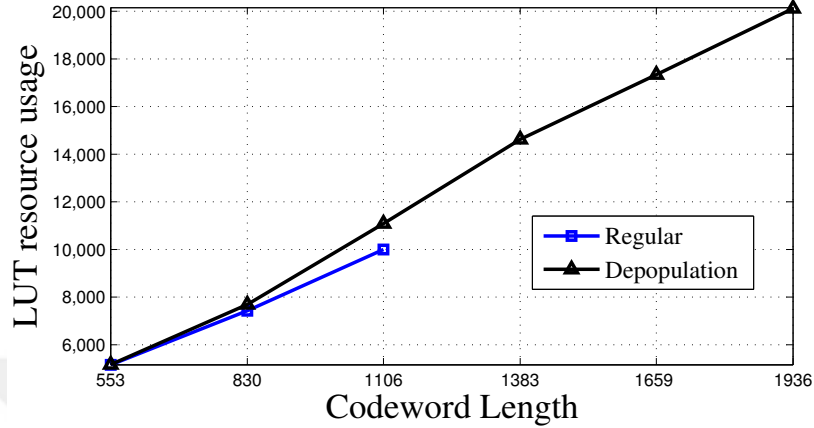


FIGURE 5.4. Resource usage vs. Codeword Length comparison for the QC-LDPC code with ($d_v = 4, d_c = 28, \text{code rate}=0.857$) based on Xilinx Zynq XC7Z020 FPGA.

TABLE 5.1. The number of LUTs used as routing resource for different codeword length on the LDPC code with ($d_v = 4, d_c = 28, \text{code rate}=0.857$) based on Xilinx Zynq XC7Z020 FPGA.

Codeword Length	553	830	1106	1383	1659	1936
Regular	53	110	301	308	287	497
Depopulation	25	29	29	70	33	41

functionality when there is stress on the routing resources. This supports our claim that the depopulation strategy helps reduce the channel width demand of the design, hence the stress on routing.

In Table 5.2, we show the total number of connections (extracted from post synthesis report) required by the regular and depopulation based implementations with respect to codeword length. Even though consistently the depopulation based implementation requires more number of connections, the designs that are not routable with the regular implementation (indicated as "fails" in the table) become routable with the depopulation based implementation. This is also a strong indication of the depopulation based implementation's ability to reduce the channel width demand.

TABLE 5.2. Total number of paths vs. codeword length (n) on the LDPC code with ($d_v = 4$, $d_c = 28$, rate=0.857).

n	553	830	1106	1383	1659	1936
Regular	124612	233530	372376	fails	fails	fails
Depopulation	124917	260236	407210	579069	792067	1041652

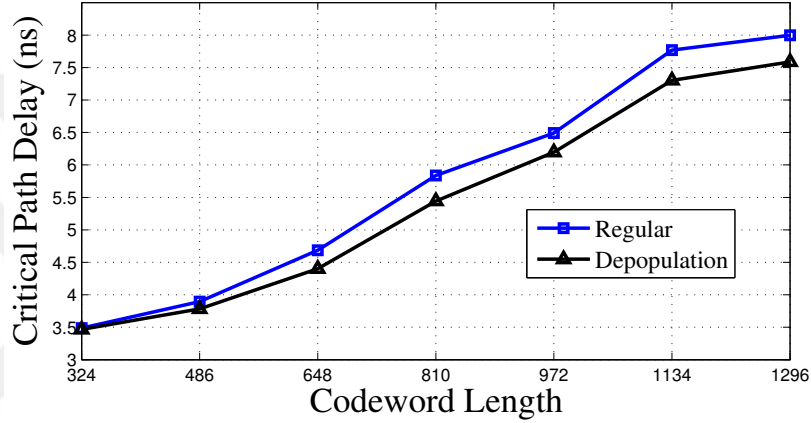


FIGURE 5.5. Critical path delay (ns) vs Codeword length for the QC-LDPC code with ($d_v = 4$, $d_c = 8$, code rate=0.5) based on Xilinx Zynq XC7Z020 FPGA.

5.3.2 Codeword Length and Code Rate vs. Critical Path Delay

Based on the resource usage trend analysis, we observe that the LUT usage is not the main constraint since up to 22% of the LUT resources are needed for the largest codeword length in our experiments. In this section we study the effect of increase in codeword length and code rate on critical path delay. We also relate the impact of increase in LUT usage due to the depopulation technique over the regular implementation with the critical path delay.

As shown in Figure 5.5, critical path delay increases linearly with the codeword length for both the regular and depopulation based implementations. However, depopulation allows reducing the critical path delay consistently compared to the regular implementation by up to 7.3%. When we increase the code rate to 0.857, we observe that the critical path delay does not increase linearly with the increase in codeword

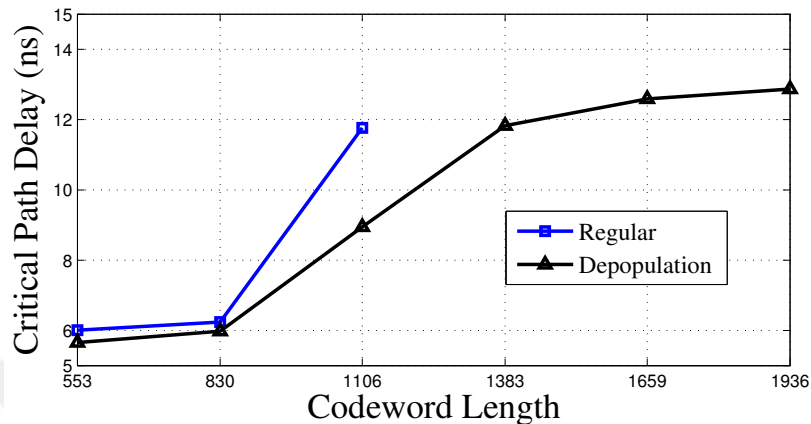


FIGURE 5.6. Critical path delay (ns) vs Codeword length for the QC-LDPC code with ($d_v = 4$, $d_c = 28$, code rate=0.857) based on Xilinx Zynq XC7Z020 FPGA.

length as shown in Figure 5.6. There is a rapid increase in delay when the codeword length is higher than 830 bits. This is an indication of the stress on routability as anticipated. Placement and routing stages attempt to reduce the total wire length for a design by positioning the logic blocks closer and utilizing the flexibility of connection boxes and switch boxes on the FPGA to establish short paths for each net. Code rate of 0.857 results with a design that increases the number of connections for each CNU. This in turn creates additional stress on routability, and the shorter wire segments available for the code rate of 0.5 are no longer available for the code rate of 0.857 due to congestion in regions that are densely populated with logic blocks. Therefore, nets take longer paths for routability and the critical path delay increases due to congestion. Beyond codeword length of 1383, the design is not routable for the regular implementation. Depopulation based implementation results with shorter critical path and unroutable designs for the last three codeword lengths become routable. We observe a saturating trend for critical path with the depopulation based implementation. Since even for the largest codeword length the design occupies 37% of the LUTs, the reduction in channel width demand allows router to find feasible paths by increasing the LUT usage. We believe this leads to the saturating trend.

5.3.3 Partitioning Amount vs. LUT Usage-Delay Product Analysis

We conducted another experiment to observe the impact of the amount of partitioning on critical path delay, resource usage and resource-delay product by increasing the number of partitions from two to four, six, and eight. We ran experiments based on the GaB implementation on the QC-LDPC code ($d_v = 4$, $d_c = 28$, and code rate= 0.857) with the codeword length of 1106. We choose 1106, since this is the longest codeword possible to route on the FPGA for the regular implementation as discussed earlier over Figure 5.6. With this experiment, our aim is to also identify the most suitable partitioning strategy based on resource usage-delay product. Figure 5.7 shows the resource usage trend with respect to the x -way partitioning, in which x -axis shows the range of partitioning and y -axis shows the Look Up Table (LUT) usage. The zero-way partitioning represents the regular implementation without partitioning. Compared to our default two-way partitioning based implementation, interestingly we observe a reduction in LUT usage by 4.5% with the four-way partitioning. However when we increase the number of partitions further to six and eight, the LUT usage increases by 4% and 20% respectively. For the eight-way partitioning, when we distribute the 28 inputs over eight CNUs, half of the CNUs have four inputs and the other half has three inputs. Since the opportunity of input sharing diminishes compared to the two or four way partitioning scenarios, the implementation results with larger number of under utilized LUTs. The four-way partitioning based design results with a fracturable LUT utilization of 20.7% in which out of 10,784 LUTs, 2,227 are used as fracturable LUT (O5 and O6 - two outputs generated). On the other hand, the eight-way partitioning based design results with reduced fracturable LUT utilization of 11.8%, in which out of 13,066 LUTs, 1,544 of them are utilized as fracturable LUT.

Figure 5.8 shows the critical path delay with respect to the x -way partitioning, where x -axis shows the range of partitioning and y -axis shows the critical path delay in nanoseconds. We observe that four-way partitioning results with better critical path

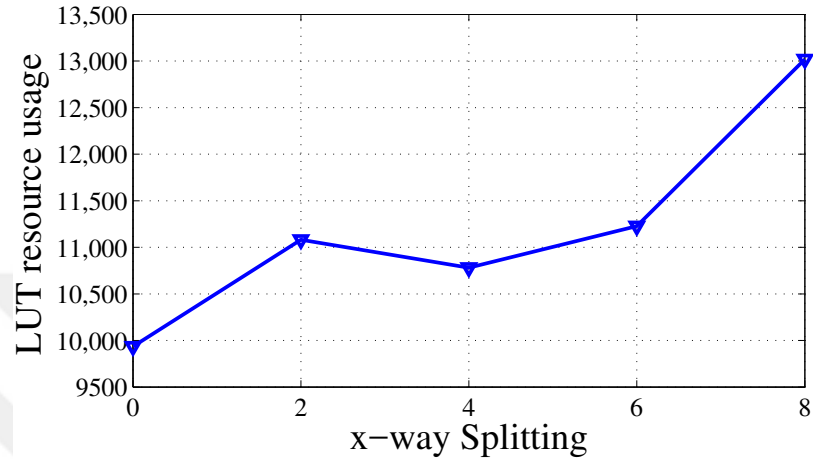


FIGURE 5.7. Resource usage trend with respect to the x-way partitioning for the QC-LDPC code with codeword length of 1106 ($d_v = 4$, $d_c = 28$, code rate=0.857) based on Xilinx Zynq XC7Z020 FPGA.

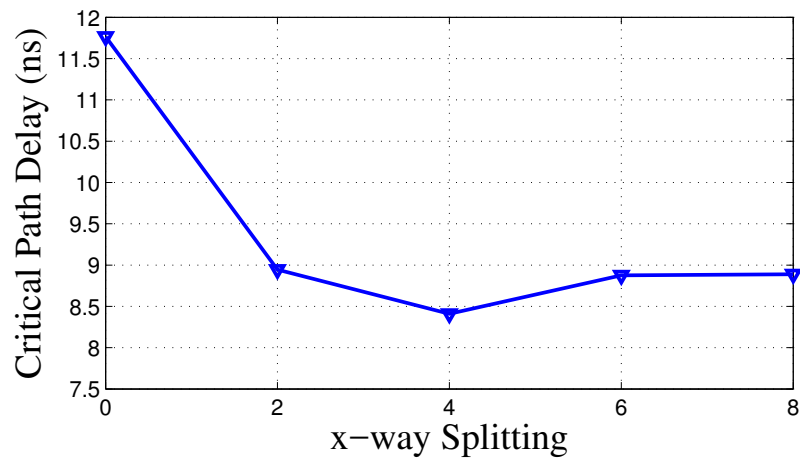


FIGURE 5.8. Critical path delay trend with respect to the x-way partitioning for the QC-LDPC code with codeword length of 1106 ($d_v = 4$, $d_c = 28$, code rate=0.857) based on Xilinx Zynq XC7Z020 FPGA.

delay among all the partitioning approaches. Critical path delay has a saturating trend as we increase the partitioning level from four to six and eight. This trend is in agreement with our earlier analysis on delay versus codeword length discussed over Figure 5.6, such that the increase in number of LUTs is not stressing the available LUT resources. Therefore depopulation is able to spread the logic evenly without increasing the critical path delay. Since four-way partitioning allows the most compact form due to higher fracturable LUT utilization, we see a reduction in critical path delay by 6.4% compared to the two-way partitioning scenario.

The throughput to LUT usage ratio is a good indicator of hardware efficiency, which is captured by the *LUT usage – delay product* as illustrated in Figure 5.3.3. We observe that the four-way partitioning results with better performance over the other partitioning choices. In overall, the four-way partitioning based design reduces the critical path delay by 32% and increases the LUT usage by 8.5% with respect to the regular implementation for this code. Here we note that the optimal partitioning level depends on the codeword length and code rate parameters that play a key role on resource usage and delay trend lines. This experiment shows that partitioning method allows resolving the congestion related critical path delay but the amount of partitioning level has a limit from LUT usage overhead perspective.

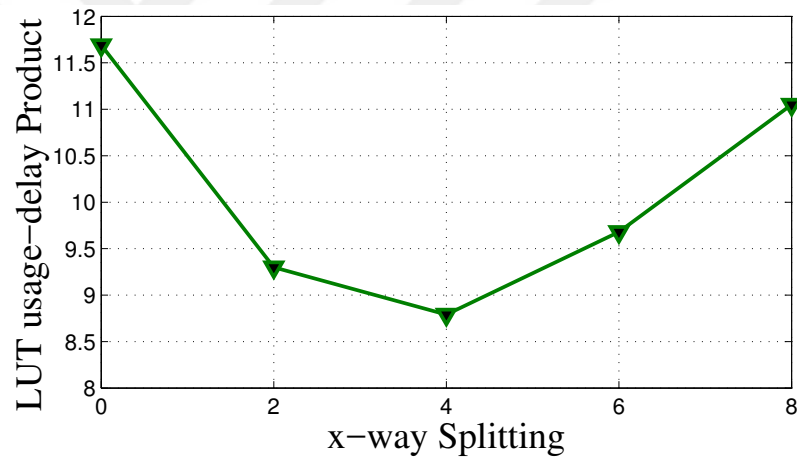


FIGURE 5.9. LUT usage-delay product trend with respect to the x-way partitioning for the QC-LDPC code with codeword length of 1106 ($d_v = 4$, $d_c = 28$, code rate=0.857) based on Xilinx Zynq XC7Z020 FPGA.

Chapter 6

FPGA BASED FRAMEWORK

In this chapter we present our FPGA based framework for studying error correction performance of the target LDPC decoder. After giving an overview of the related work from FPGA based LDPC testbed implementation point of view, we discuss the details of the hardware implementation. We then demonstrate our framework's ability to reduce the time scale of simulations for two case studies on error correction performance analysis using Gallager B and Probabilistic Gallager B algorithms and identifying all possible codewords that are not correctable by Gallager B for codewords that have four errors.

6.1 Related Work

Evaluating the decoding performance of an LDPC code on a general purpose processor based single node requires extremely long simulation times based on the targeted degree of quality in terms of error correction capability. Cluster based computing systems and more recently fine grained parallel computing platforms such as GPGPUs have been utilized to reduce the time scale of LDPC simulations with some degree of success [68]. Because the decoding process is iterative and the data access patterns are not regular between the iterations, parallelism efforts are limited to iteration level only, which also incurs significant amount of cycles spent for memory updates between the iterations. The use of FPGAs allows for substantial acceleration in the simulation of LDPC codes [69], [10]. Bit-wise intensive operations involved in the decoding process and the iterative flow that is amendable to loop-unrolling style execution make the program architecture of the LDPC decoder algorithms overlap with the fine-grained massively parallel structure of the FPGA architecture. Modern

FPGAs have particularly become a practical option for reducing the timescale LDPC simulations as they house more than a million logic cells coupled with rich pool of memory and DSP resources.

FPGA based implementation of error correction algorithms is a standard way of studying their hardware characteristics and conducting hardware performance comparison in terms of resource usage, critical path delay, energy efficiency, and throughput metrics [70], [71], [72], [73], [74], [63], [75]. The paper on FPGA based LDPC decoders by [10] presents a comprehensive analysis. Dominantly, these studies are limited to extracting hardware characteristics from the FPGA based implementation, rather than utilizing the FPGA as a testbed for simulating a decoder and studying its error correction performance.

The literature on FPGA based testbeds, as summarized in Table 6.1, focuses on a few best performing algorithms in their class to accelerate their simulation based investigations with speedup values reaching three orders of magnitude. To the best of our knowledge, the type of algorithms that we investigate in our framework have not been studied on a completely FPGA based simulation framework. Even though it is not fair to compare different classes of algorithms in terms of their hardware performance and resource utilization, for the completeness of our analysis we give an overview of each by highlighting their key features.

The study by [69] proposes an FPGA based testbed and investigates the impact of employing various levels of parallelism on resource usage and throughput when implementing LDPC decoder architecture for the sum-product [64] algorithm with codeword length of 2048. The testbed allows for the exploration of the low FER region and provides statistics of the error traces. The testbed is implemented on the Virtex-II Pro XC2VP70 series Xilinx FPGA achieving the FER resolution of 10^{-8} within one hour. The decoder implementation reaches a throughput of 240 Mb/s on the FPGA, while the C based implementation achieves a throughput of 260 kb/s on an Intel Xeon 2.4 GHz processor.

TABLE 6.1. FPGA based LDPC testbeds.

	Platforms	Algorithms	Throughput	Codeword length
[69]	Virtex-2 Pro XC2VP70	sum-product	240 Mb/s	2048
[76]	Virtex-5 XC5VLX155T (4 nodes)	belief propagation	380 Mb/s	576
[77]	Virtex-2 Pro (6 nodes)	Min-Sum	NA	4923
[78]	Virtex-5 XC5VSX240T-2	Min-Sum	332 Mb/s	3369
[79]	Virtex-5 FPGA	sum-product	344 Mb/s	2304

The study by [76] proposes an automated design flow for FPGA based LDPC code simulation. The testbed offers the end user the flexibility of specifying the LDPC decoder parameters to implement the hardware architecture automatically. Several LDPC codes for the belief propagation algorithm [24] are implemented with the codeword lengths ranging from 576 to 2304. The simulation platform employs four Xilinx Virtex-5 XC5VLX155T devices and achieves throughput from 380 to 950 Mb/s. The testbed allows researchers to reach a FER resolution of 10^{-11} in less than one hour.

The study by [77] presents a highly parallel FPGA-based testbed to evaluate the performance of the Min-Sum [65] algorithm on LDPC code with the codeword length of 4923. Eight Xilinx Virtex-2 Pro FPGAs are utilized to achieve 100x speed up over the same class of application specific simulation platforms and achieve the first exploration for the error floor of LDPC codes at FER resolution of 10^{-12} in magnetic recording channel [80], [81], [82]. The testbed reaches a FER of 10^{-10} in less than six hours and FER of 10^{-12} in 24 days. The authors estimate that reaching the scale of 10^{-12} would take 10 years in software simulation.

The study by [78] proposes a scalable FPGA-based vector decoder implementation for the Min-Sum algorithm on the LDPC codes with the codeword length of up to 3369. This implementation packs the data used in the subsequent iteration of the decoding process into the embedded memory blocks to minimize LUT-RAM usage on the target Virtex 5 XC5VSX240T-2 FPGA, and achieves a maximum throughput of

332 Mb/s.

[79] propose an FPGA based testbed implementation of sum-product algorithm for the LDPC code with the codeword lengths of 1056, 1944, and 2304. Design and implementation of a backtracking scheme to detect error patterns at FER as low as 10^{-10} is presented. The proposed testbed achieves a throughput of 344 Mb/s for the codeword length of 2304.

As summarized in Table 6.1, the literature on FPGA-based testbeds target effective error correction algorithms such as sum-product, Min-Sum, and belief propagation. These algorithms are strong in terms of error correction performance but require much more hardware resources than the class of algorithms we target.

The motivation behind our PGaB theoretical work [83] was to improve the error correction performance of the GaB through algorithmic modifications while inheriting its simplicity from hardware implementation point of view. The GaB and PGaB class of algorithms we investigate in this dissertation are different from the class of algorithms listed in Table 6.1. Our implementations outperform all the testbed based implementations in terms of throughput reaching up to 4780 Mb/s as we will present in Section 6.3.1. From testbed point of view, these studies are also limited to a single algorithm. In our work, we introduce a framework that allows studying hard-decision class of algorithms involving GaB, PGaB, GDBF, and PGDBF.

6.2 Framework and Hardware Implementation

The overall block diagram, shown in Figure 6.1, consists of seven units: controller, codeword generator, noise generator, LDPC decoder, Random Number Generator (RNG), statistic and error analysis, and interface units. Our testbed offers flexibility of changing the decoder type and simulation settings based on the desired resolution in terms of FER performance. For this, the end-user sets the channel crossover probability (*Alpha*), maximum number of iterations (*max_iter*), and maximum number of

codewords in fail (max_cwf), which form the inputs to the *Controller* in Figure 6.1 as simulation parameters. $Alpha$ defines channel crossover probability, which is the probability of error introduced to each bit of the codeword. This parameter allows the end-user to test the decoder for various channel noise scenarios. The higher the $Alpha$ value is, the harder it is for the decoder to recover the codeword. The max_iter determines the limit of iteration count per codeword, after which the simulation terminates. If the decoder can not correct the codeword before reaching this limit, the codeword is designated as not recoverable. If the codeword is corrected earlier than max_iter , decoder continues to execute the next codeword. Depending on the LDPC algorithm, the maximum number of attempts to correct codeword may change. The simulation stops when the total number of codewords that are not recoverable reaches the max_cwf . Setting max_cwf to 100 is a typical choice for a reliable analysis, but it can be set to larger or smaller values depending on the desired resolution. We keep track of the total number of codewords tested, number of iterations spent for each codeword, and the total number of iterations spent over all the codewords tested. Based on this information, we calculate the average number of iterations and the FER for a given $Alpha$.

The controller module in the testbed manages the processes of each component. The codeword generator unit produces codewords specified by parity check matrix H . Each time a component completes its task, an enable signal indicated with " ena " in the figure triggers controller to activate the next component. In order to emulate the effect of noise in the communication channel on the transmitted n -bit codeword, the noise generator unit flips each bit of the codeword with the probability of $Alpha$. We implement a linear feedback shift register (LFSR) based random number generator to generate a 1-bit random value with Bernoulli distribution of $Alpha$, which represents the probability of each bit in the codeword being flipped or not. After the decoder receives its enable signal along with the noisy codeword, iterative process continues until either the error has been corrected or the decoder has failed within the

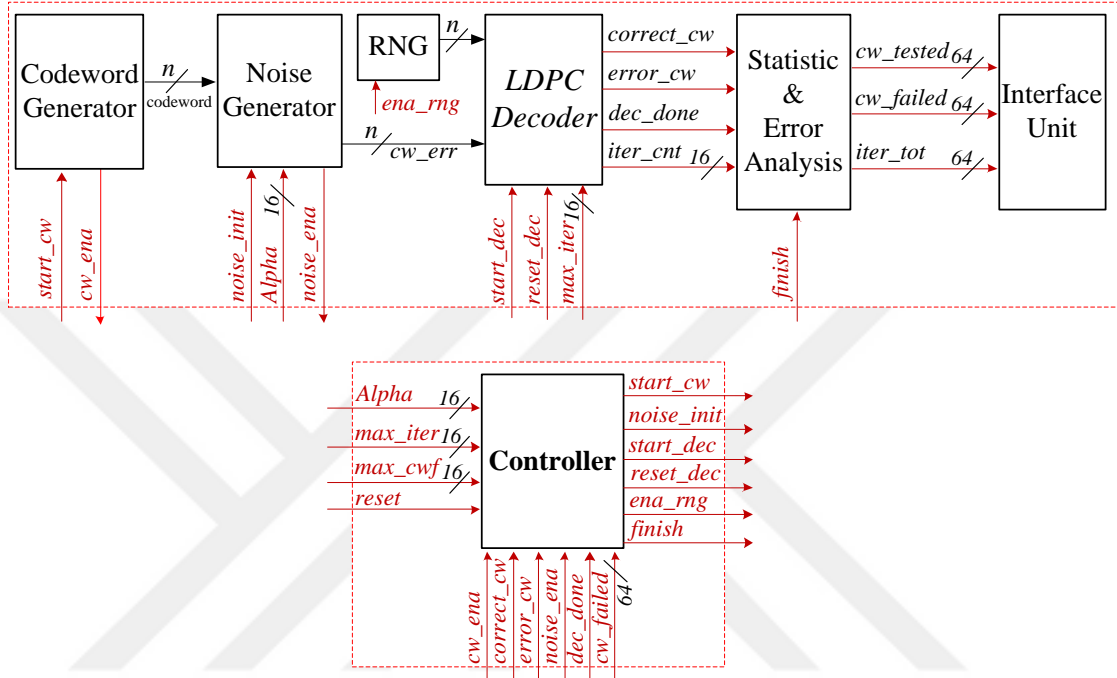


FIGURE 6.1. Overall architecture of FPGA based testbed. Red colored wires indicate control signals, others indicate internal connections.

predefined maximum iteration count (*max_iter* input). Termination of the decoding process starts the statistical analysis process, which receives 16 bits of iteration count (*iter_cnt*) along with 1-bit successful correction (*correct_cw*) and failed correction (*error_cw*) signals from the decoder. We keep track of total number of codewords tested (*cw_tested*), number of codewords in fail (*cw_failed*), and total number of iterations (*iter_tot*) spent with 64 bit registers. Random number generator (RNG) unit is a special purpose module needed to simulate probabilistic LDPC decoders, which generates an *n*-bit random sequence to pair with the *n*-bit codeword length. We implement 32-bits LFSR to generate a random number. The controller enables or disables the RNG with the the signal (*ena_rng*) based on the decoder algorithm type.

TABLE 6.2. Hardware resource utilization of LDPC testbed implementation on the FPGA. Available resource is based on the Xilinx Zynq XC7Z020 and Virtex-7 XC7VX485T FPGAs.

	Resource Usage		Utilization (%)	
	Zynq	Virtex-7	Zynq	Virtex-7
Slice Luts	11884	11688	22%	4%
1-bit Register	19731	19731	18%	3%
Number of Slices	3086	2775	23%	4%

Table 6.2 shows the resource utilization of the testbed excluding the decoder block on Xilinx Zynq XC7Z020 and Virtex-7 XC7VX485T FPGAs. We use two types of FPGAs to show that even a resource rich FPGA based implementation faces routability problem for longer codewords. Table 6.2 shows that the testbed with its low resource utilization allows designers to map various decoding algorithms with different degrees of parallelism to exploit for their decoder.

6.3 Quantifying the Benefits of the FPGA Based Framework

In the following subsections we first quantify the benefit of our testbed based on the execution time of generating the FER curves for the GaB and PGaB. We show that the reduction of simulation time from hundred year scale to hours scale makes our testbed a practical solution for evaluating the error correction quality of the GaB and PGaB decoders. Our testbed allows the end user to reach resolution levels that has not been reported before. We then evaluate the utility of our testbed for conducting error pattern analysis needed by information theorists towards improving the error correction performance of the decoder. We show that sweeping the combinatorial search space at a scale of 10^{12} patterns is completed within hours, which is not attainable with software based simulations.

6.3.1 Case Study: Time to Generate FER Analysis

Figure 6.2 shows the FER curves for the GaB and PGaB algorithms on the QC-LDPC code with codeword length of 1296 ($d_v = 4$, $d_c = 8$, code rate=0.5). The *y-axis* indicates the frame error rate (FER) calculated based on number of codewords processed till a total of 100 code words fail. The *x-axis* indicate the probability of error (crossover probability) introduced to each bit of the input codeword. The slope of the FER curve decreases suddenly beyond certain crossover probability and this region of the curve is called as error floor [84]. The later this error floor region occurs, the better the decoding algorithm is. Therefore it is crucial to be able to conduct simulations with a resolution that reaches the error floor region to be able to make conclusive comparison on the error correction quality with respect to other decoder algorithms. To the best of our knowledge, the lowest resolution reported for the PGaB is for the 0.01 crossover probability. The C based simulation takes 116 days to complete on the Intel Xeon (2.33GHz, 8GB RAM) processor, while the FPGA based testbed completed the simulation in four minutes for the PGaB algorithm. With our testbed we are able to reach to 0.005 resolution at FER of 0.917×10^{-11} . This simulation takes 24 hours and involves processing 1,090,097,582,683 codewords (4780 Mb/s). On an Intel Xeon processor, the C based simulation of the PGaB algorithm for the same configuration ($d_v = 4$, $d_c = 8$, codeword length = 1296, and code rate = 0.5) takes 1 minute to process 10,396 codewords. Based on this throughput, we estimate that reaching to 1,090,097,582,683 codewords at crossover probability of 0.005 would take 199 years. Table 6.3 shows the execution time comparison for the crossover probabilities of 0.005 and 0.01 over GaB and PGaB.

6.3.2 Case Study: Error Pattern Analysis

For information theorists one of the interesting problems to investigate is on understanding the error floor region particularly the type of errors that cause a decoder

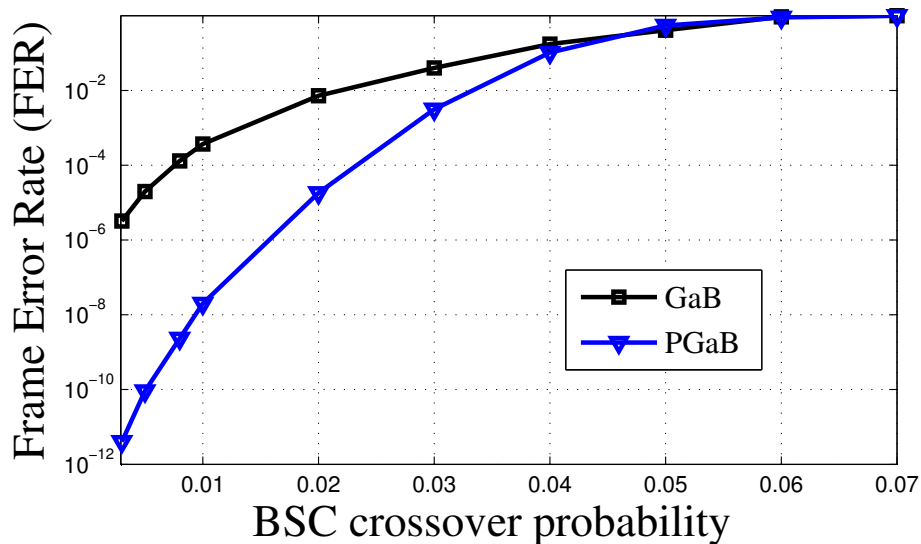


FIGURE 6.2. GaB and PGaB FER comparison: FER vs. probability of error introduced to each bit of the 1296-bit codeword with ($d_v = 4, d_c = 8$, code rate = 0.5).

TABLE 6.3. Execution time for GaB and PGaB on the FPGA based testbed (Xilinx Zynq XC7Z020 FPGA) for the crossover probabilities of 0.005 and 0.01 where ($d_v = 4, d_c = 8, n = 1296$, code rate=0.5) (* indicates estimated time).

<i>Alpha</i>	0.01		0.005	
Environment	FPGA	PC	FPGA	PC
GaB	1 min<	2h' 27 min	1 min<	18h' 47 min
PGaB	4 min	116 days	24 hours	199 years*

fail [85]. Dominant error patterns, most common and harmful ones, should be detected and removed when a new decoder is designed.

Error Pattern Definition: Before presenting our results, for the completeness of the study here we give a definition of the error pattern. During the decoding process, the interactions between CNUs and VNUs may result in an oscillation phenomena due to the n^{th} order dependencies between CNUs and VNUs. In such cases, the decoding process may get trapped in a cyclic behavior. For example, in the Tanner graph given

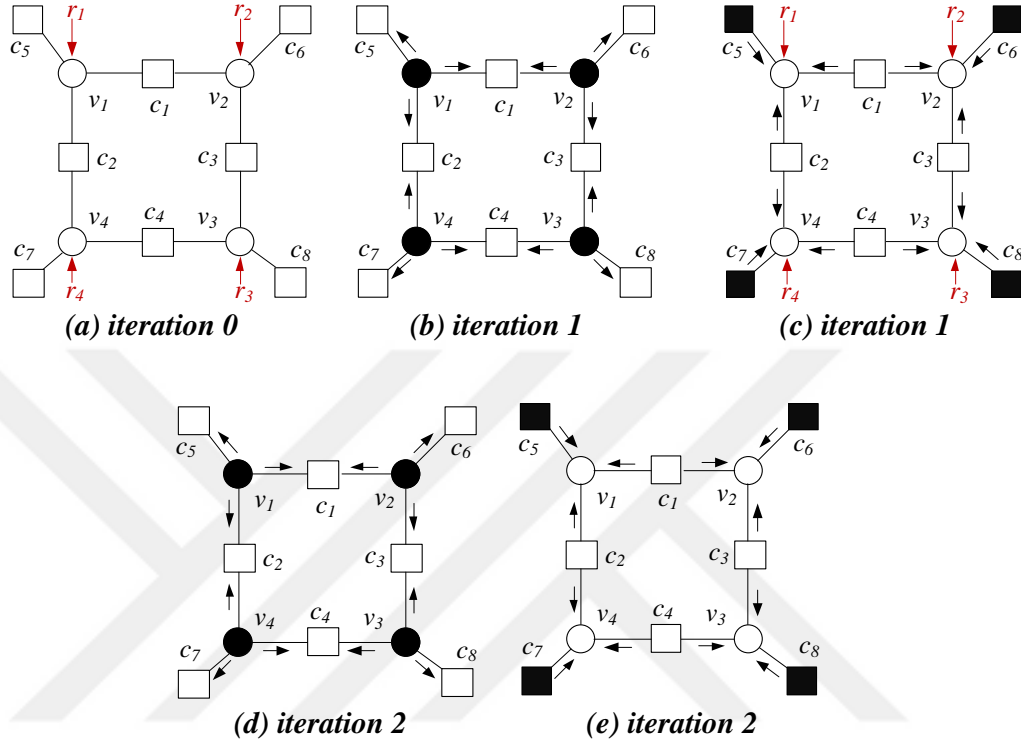


FIGURE 6.3. 4-error pattern for GaB algorithm. Black/white denotes erroneous/correct VNU, and unsatisfied/satisfied CNU.

in Figure 2.1, v_0 transmits message to c_0 , c_1 , and c_2 . After receiving their inputs from all VNUs, c_0 , c_1 , and c_2 send their messages back to their designated VNUs. In this example, there is a third order dependency between v_0 and v_2 based on the message passing in the order of $(v_0 - c_0 - v_5 - c_4 - v_3 - c_3 - v_2)$. If we count each VNU-CNU interaction as one iteration, then it would take three iterations for the message of v_0 to propagate to v_2 . During each iteration, CNUs and VNUs update their states. The sequence of states observed for a given VNU may show repeating pattern, which is called a trapping set [44]. Trapping means that the decoder cannot correct the error, and then it remains in the cyclic sequences of states.

Rather than conducting exhaustive sweeping based experiments, researchers avoid the computation time barrier through analytical studies. For example, the study

by [50] analytically shows that the error pattern indicated in Figure 6.3 makes the GaB decoding algorithm fail. The figure represents a subgraph of LDPC code where circles denote VNUs and squares denote CNU. For the sake of simplicity, we do not include all connections for VNUs and CNU. An error is denoted by black filling for VNUs and CNU. Assume that the codeword initially has four errors represented with received message $r_1, r_2, r_3,$ and r_4 (Figure 6.3a - Iteration 0). When each VNU receives its message bit from the channel, the $v_1, v_2, v_3,$ and v_4 are in error as illustrated in Figure 6.3(b). All VNUs send message to their designated CNU. After receiving their inputs from all VNUs, $c_5, c_6, c_7,$ and c_8 generate a message as unsatisfied since each of them has one error message from their designated VNUs (Figure 6.3(c)). In the second iteration, $v_1, v_2, v_3,$ and v_4 receive one error message from their designated CNU and receive one error message from channel. Since the function of VNU is finding majority of the inputs, $v_1, v_2, v_3,$ and v_4 continue to generate error message. In such cases, the decoder cannot correct this 4-error pattern. Therefore it remains in the cyclic sequences of error states.

Analysis: The decoding algorithm under investigation may be failing to correct the received codeword when " k " number of bits in a codeword are in error (bit flipped) and these " k " error bits appear in certain patterns. In such cases, the designer goes through a massive exploration space to understand which error patterns cause the decoding algorithm fail. After discovering these patterns, then the designer establishes a path for modifying the LDPC code to address this problem.

A typical design space exploration starts with 2-error patterns. This involves studying all possible error bit positions in a codeword length of " n " with only two errors, and identifying the patterns that the decoder fails to correct. The exploration incrementally may go up to six error patterns based on the application [85]. Let " n " be the length of the codeword, and " k " be the number of error bits in a codeword. In this case we can formulate the number of all possible error patterns as shown in

Equation 6.3.1.

$$C(n, k) = \binom{n}{k} = \frac{n!}{(n-k)!k!} \quad (6.3.1)$$

where $C(n, k)$ refers combination of k bits in error out of codeword length of n .

In order to evaluate all 4-bit error patterns for a codeword length of 1296, based on Equation 6.3.1, we need to test 117,002,820,060 codewords. When "k" is 5, this value is in the order of 30 (30,233,528,703,504) trillion.

We choose the GaB algorithm for this case study as it has been extensively studied in terms of error patterns [45]. For the GaB algorithm the throughput on the Intel Xeon (2.33GHz, 8GB RAM) processor is 10,396 codewords per minute on the QC-LDPC code with codeword length of 1296 ($d_v = 4$, $d_c = 8$, code rate=0.5). Based on this, we estimate the 4-bit error pattern experiment to take 7,803 days. We conducted the same experiment to simulate all possible 4-error patterns for the GaB algorithm on our FPGA based testbed, and discovered all 4-error patterns, including the ones published in [50] that make the codeword uncorrectable. Total execution time for this FPGA based simulation is 4.5 hours. For verification purpose, we ran the GaB in C based simulation for those 4-error patterns and verified that the decoder could not correct these errors. The entire source code for the FPGA based testbed is available at [39].

Chapter 7

CONCLUSIONS AND PERSPECTIVES

7.1 Summary and Contributions

In this dissertation we propose probabilistic GaB algorithm and quantified the hardware cost and performance of the GaB and PGaB decoder. We showed that without a performance loss in throughput, we improved the decoding performance of the GaB significantly and bridged the gap between GaB and other hard decision bit flipping decoding algorithms. Our simulation results showed that the PGaB now even has a better decoding performance than GDBF. In our current designs, we are generating a 1296-bit random number register. We will investigate ways to reduce this register footprint by sharing 1-bit register among a cluster of VNUs. This would reduce the size of the shift register on the datapath and have considerable impact on resource usage. Our evaluations rely on implementation of a new architecture for each codeword and core rate combination. Ability to switch the context from one implementation to another at runtime would allow us to evaluate multi-rate LDPC codes. For future work we plan to expand the capabilities of our FPGA based simulation testbed with partial reconfiguration and run time configuration to conduct a run-time flexible analysis.

We showed that the depopulation strategy spreads the logic across the target FPGA resources. We analyzed the resource usage, critical path delay, and resource usage delay product trends with respect to the degree of partitioning applied to the design. We concluded that even though resource usage increases with respect to regular implementation, depopulation reduces the path delay of the implementations for all codewords studied in this dissertation. We showed that the depopulation strategy reduces congestion and allows unroutable designs become routable for longer codeword

sizes, that is needed by today's applications and standards in mobile communications, video broadcasting, and wireless communications.

In this dissertation we also presented an FPGA based framework for accelerating the simulations of LDPC codes. With its tunable parameters, the testbed allows the end user to conduct experiments based on the desired resolution for the target decoder algorithm. We conducted a series of experiments based on GaB and PGaB algorithms and show that our testbed reduces the time scale of simulations from years scale to hours scale, which allows analysis of error correction performance at a resolution that is not attainable with CPU based simulations. We conducted an error pattern analysis study to detect all possible 4-bit error patterns in a codeword, identified all 4-bit error patterns that are not correctable by the GaB algorithm in less than five hours that is estimated to be completed in over 7800 days on a CPU based testbed. An error pattern discovered on specific CNUs and VNUs (subgroup in the architecture) may repeat itself with other CNU and VNU subgroups due to quasi cyclic nature of the LDPC codes. Therefore next step in our investigation will be to categorize these error patterns into groups. We will be in a position to rank and prioritize these patterns and incrementally modify the GaB algorithm to address each pattern group. We also plan to expand supporting other classes of decoder algorithms such as the sum-product algorithm. The entire source code for the FPGA based simulation framework along with the generated data is available as open source for the community [39].

REFERENCES

- [1] C. E. Shannon, "A mathematical theory of communication," *Bell System Tech. Journal*, vol. 27, pp. 379–423, 623–656, 1948.
- [2] R. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1962.
- [3] H. D. Pfister, I. Sason, and R. Urbanke, "Capacity achieving ensembles for the binary erasure channel with bounded complexity," *IEEE Transactions on Information Theory*, vol. 51, no. 7, pp. 2352–2379, July 2005.
- [4] D. J. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electronics letters*, vol. 32, no. 18, pp. 1645–1646, 1996.
- [5] D. J. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE transactions on Information Theory*, vol. 45, no. 2, pp. 399–431, 1999.
- [6] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.
- [7] R. C. Bose and D. K. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Information and control*, vol. 3, no. 1, pp. 68–79, 1960.
- [8] A. Hocquenghem, "Codes correcteurs derreurs," *Chiffres*, vol. 2, no. 2, pp. 147–56, 1959.
- [9] T. Richardson and R. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 599–618, Feb. 2001.
- [10] P. Hales, L. Xu, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, "A survey of FPGA-based LDPC decoders," *IEEE Communications Surveys Tutorials*, vol. 18, no. 2, pp. 1098–1122, 2016.
- [11] LDPC Codes for the Enhanced Mobile Broadband (eMBB) Data Channel the 3GPP 5G New Radio. [Online]. Available: <http://www.3gpp.org/ftp/tsgan/WG1RL1/TSGR188/Report/>
- [12] *IEEE Standard for Information Technology-Telecommunications and Information Exchange Between Systems-Local and Metropolitan Area Networks-Specific Requirements Part 3: Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, Std. IEEE 802.3an, Sep. 2006.
- [13] Z. Zhang, V. Anantharam, M. J. Wainwright, and B. Nikolic, "An efficient 10gbase-t ethernet ldpc decoder design with low error floors," *IEEE Journal of Solid-State Circuits*, vol. 45, no. 4, pp. 843–855, April 2010.
- [14] ETSI. (2013). ETSI EN 302 307 v1.3.1, "Digital video broadcasting (dvb); second generation [online]. available: <https://www.dvb.org/standards/dvb-s2>."
- [15] *Standard for Information Technology Local and Metropolitan Area Networks Specific Requirements, Part 11:; Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification*, Std. IEEE 802.11n-2009, 2009.
- [16] *Standard for Local and Metropolitan Area Networks Part 16:; Air Interface for Fixed Broadband Wireless Access Systems*, Std. IEEE 802.16-2004, 2004.
- [17] K. S. Andrews, D. Divsalar, S. Dolinar, J. Hamkins, C. R. Jones, and F. Pollara, "The development of turbo and LDPC codes for deep-space applications," *Proceedings of the IEEE*, vol. 95, no. 11, pp. 2142–2156, Nov 2007.

- [18] J. Kim and J. Lee, "Two-dimensional SOVA and LDPC codes for holographic data storage system," *IEEE Transactions on Magnetics*, vol. 45, no. 5, pp. 2260–2263, May 2009.
- [19] 3GPP TSG RAN WG1 Meeting RAN1-86 R1-167703, "Channel coding scheme for urllc, mmhc and control channels," Intel Corporation, Reno, USA, August 2016.
- [20] 3rd Generation Partnership Project, Technical Specification Group Radio Access Network, "Nr; multiplexing and channel coding," 3GPP TS 38.212 V15.0.0, December 2017.
- [21] F. Ghaffari, A. Akoglu, B. Vasic, and D. Declercq, "Multi-mode low-latency software-defined error correction for data centers," pp. 1–8, July 2017.
- [22] A. Kavcic and A. Patapoutian, "The read channel," *Proceedings of the IEEE*, vol. 96, no. 11, pp. 1761–1774, 2008.
- [23] J. Chen and M. P. Fossorier, "Near optimum universal belief propagation based decoding of low-density parity check codes," *IEEE Transactions on communications*, vol. 50, no. 3, pp. 406–414, 2002.
- [24] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Transactions on Information Theory*, vol. 45, no. 2, pp. 399–431, March 1999.
- [25] D. Declercq, M. Fossorier, and E. E. Biglieri, *Channel Coding: Theory, Algorithms, And Applications*. Academic Press Library in Mobile and Wireless Communications, Elsevier, 2014.
- [26] M. Fossorier, M. Mihaljevic, and H. Imai, "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation," *IEEE Transactions on Communications*, vol. 47, no. 5, pp. 673–680, May 1999.
- [27] X. Wu, Y. Song, M. Jiang, and C. Zhao, "Adaptive-normalized/offset min-sum algorithm," *IEEE Communications Letters*, vol. 14, no. 7, pp. 667–669, July 2010.
- [28] B. Unal, F. Ghaffari, A. Akoglu, D. Declercq, and B. Vasic, "Analysis and implementation of resource efficient probabilistic Gallager B LDPC decoder," *2017 15th IEEE International New Circuits and Systems Conference (NEWCAS)*, pp. 333–336, June 2017.
- [29] R. G. Gallager, *Low Density Parity Check Codes*. Cambridge, MA: MIT Press, 1963.
- [30] V. Zyablov and M. Pinsker, "Estimation of the error-correction complexity for gallager low-density codes," *Problems of Information Transmission*, vol. 11, no. 6, pp. 18–26, July 1976.
- [31] T. Wadayama, K. Nakamura, M. Yagita, Y. Funahashi, S. Usami, and I. Takumi, "Gradient descent bit flipping algorithms for decoding LDPC codes," *IEEE Transactions on Communu-cations*, vol. 58, no. 6, pp. 1610–1614, June 2010.
- [32] O. Rasheed, P. Ivanis, and B. Vasic, "Fault-tolerant probabilistic gradient-descent bit flipping decoder," *IEEE Transactions on Communications*, vol. 18, no. 9, pp. 1487–1490, September 2014.
- [33] A. J. Blanksby and C. J. Howland, "A 690-mw 1-gb/s 1024-b, rate-1/2 low-density parity-check code decoder," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 3, pp. 404–412, March 2002.
- [34] X.-Y. Shih, C.-Z. Zhan, C.-H. Lin, and A.-Y. Wu, "An 8.29 mm² 52 mW multi-mode LDPC decoder design for mobile WiMAX system in 0.13 um CMOS process," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 3, pp. 672–683, 2008.
- [35] P. Urard, E. Yeo, L. Paumier, P. Georgelin, T. Michel, V. Lebars, E. Lantreibecq, and B. Gupta, "A 135Mb/s DVB-S2 compliant codec based on 64800b LDPC and BCH codes," in *ISSCC. 2005 IEEE International Digest of Technical Papers. Solid-State Circuits Conference, 2005*. IEEE, 2005, pp. 446–609.

- [36] H.-Y. Liu, C.-C. Lin, Y.-W. Lin, C.-C. Chung, K.-L. Lin, W.-C. Chang, L.-H. Chen, H.-C. Chang, and C.-Y. Lee, "A 480Mb/s LDPC-COFDM-based UWB baseband transceiver," in *ISSCC. 2005 IEEE International Digest of Technical Papers. Solid-State Circuits Conference, 2005*. IEEE, 2005, pp. 444–609.
- [37] F. Ghaffari, B. Unal, A. Akoglu, K. Le, D. Declercq, and B. Vasi, "Efficient FPGA implementation of probabilistic Gallager B LDPC decoder," in *2017 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, Dec 2017, pp. 178–181.
- [38] G. Falcao, M. Owaida, D. Novo, M. Purnaprajna, N. Bellas, C. D. Antonopoulos, G. Karakostas, A. Burg, and P. Ienne, "Shortening design time through multiplatform simulations with a portable OpenCL golden-model: The LDPC decoder case," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, April 2012, pp. 224–231.
- [39] "Complete source code for the FPGA based testbed along with the data is available at," 2019. [Online]. Available: http://www2.engr.arizona.edu/~rcl/publications/source_codes/ldpc/framework.html
- [40] B. Unal and A. Akoglu, "Design of high throughput FPGA based framework for accelerating error characterization of LDPC codes," *ACM Transactions on Reconfigurable Technology and Systems (TRETTS)*, pp. 1–20, 2019, (in review).
- [41] M. S. Hassan, B. Unal, and A. Akoglu, "FPGA based testbed for large scale simulations of LDPC codes," *2019 IEEE 62nd International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 1–4, 2019, (in review).
- [42] R. M. Tanner, D. Sridhara, and T. Fuja, "A class of group-structured LDPC codes," *Proceedings of ICSTA: International Conference on Space-Time Absoluteness*, pp. 365–370, July 2001.
- [43] R. Tanner, "A recursive approach to low complexity codes," *IEEE Transactions on Information Theory*, vol. 27, no. 5, pp. 533–547, September 1981.
- [44] B. Vasić, S. K. Chilappagari, and D. V. Nguyen, "Failures and error floors of iterative decoders," *Channel Coding: Theory, Algorithms, and Applications: Academic Press Library in Mobile and Wireless Communications*, pp. 299–341, June 26 2014.
- [45] B. Vasic, P. Ivanis, and D. Declercq, "Approaching maximum likelihood performance of LDPC codes by stochastic resonance in noisy iterative decoders," *Information Theory and Applications Workshop (ITA)*, pp. 4291–4296, February 2016.
- [46] F. Leduc-Primeau, S. Hemati, S. Mannor, and W. J. Gross, "Dithered belief propagation decoding," *IEEE Transactions on Communications*, vol. 60, no. 8, pp. 2042 – 2047, August 2012.
- [47] C. Leroux, S. Hemati, S. Mannor, and W. J. Gross, "Stochastic chase decoding of reed-solomon codes," *IEEE Communications Letters*, vol. 14, no. 9, pp. 863 – 865, September 2010.
- [48] K. Le, D. Declercq, F. Ghaffari, C. Spagnol, P. Ivanis, and B. Vasic, "Efficient realization of probabilistic gradient descent bit flipping decoders," *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1494–1497, May 2015.
- [49] K. Le, F. Ghaffari, D. Declercq, and B. Vasic, "Efficient hardware implementation of probabilistic gradient descent bit-flipping," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 4, pp. 906–917, April 2016.
- [50] P. Ivanis and B. Vasic, "Error error eicitur: A stochastic resonance paradigm for reliable storage of information on unreliable media," *IEEE Trans on Comm*, vol. 64, no. 9, pp. 3596 – 3608, September 2016.

- [51] N. Deak, T. Gyrfi, K. Mrton, L. Vacariu, and O. Cret, "Highly efficient true random number generator in FPGA devices using phase-locked loops," *20th International Conference on Control Systems and Computer Science*, pp. 453–458, May 2015.
- [52] J. Lan, W. L. Goh, Z. H. Kong, and K. S. Yeo, "A random number generator for low power cryptographic application," *2010 International SoC Design Conference*, pp. 328–331, November 2010.
- [53] B. Yuce, H. F. Ugurdag, S. Gren, and G. Dndar, "Fast and efficient circuit topologies for finding the maximum of n k-bit numbers," *IEEE Transactions on Computers*, vol. 63, no. 8, pp. 1868–1881, Aug 2014.
- [54] T. Nguyen-Ly, K. Le, F. Ghaffari, A. Amaricai, O. Boncalo, V. Savin, and D. Declercq, "FPGA design of high throughput LDPC decoder based on imprecise offset min-sum decoding," *IEEE 13th International New Circuits and Systems Conference (NEWCAS)*, pp. 1–4, June 2015.
- [55] J. Zhang, J. Wang, S. G. Srinivasa, and L. Dolecek, "Achieving flexibility in LDPC code design by absorbing set elimination," *Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pp. 669–673, November 2011.
- [56] A. Darabiha, A. C. Carusone, and F. R. Kschischang, "Block-interlaced LDPC decoders with reduced interconnect complexity," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 55, no. 1, pp. 74–78, January 2008.
- [57] C. Howland and A. Blanksby, "Parallel decoding architectures for low density parity check codes," *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No. 01CH37196)*, vol. 4, pp. 742–745, May 2001.
- [58] E. Yeo and B. Nikolic, "A 1.1-gb/s 4092-bit low-density parity-check decoder," in *2005 IEEE Asian Solid-State Circuits Conference*. IEEE, 2005, pp. 237–240.
- [59] P. Urard, L. Paumier, V. Heinrich, N. Raina, and N. Chawla, "A 360mW 105Mb/s DVB-S2 compliant codec based on 64800b LDPC and BCH codes enabling satellite-transmission portable devices," in *2008 IEEE International Solid-State Circuits Conference-Digest of Technical Papers*. IEEE, 2008, pp. 310–311.
- [60] G. A. Malema, M. J. Liebelt, and C. C. Lim, "Reduced wire-length and routing complexity for LDPC decoders," vol. *Microelectronics: Design, Technology, and Packaging II*, 2006, p. 6035.
- [61] D. Bao, X. Chen, Y. Huang, C. Wu, Y. Chen, and X. Y. Zeng, "A single-routing layered LDPC decoder for 10Gbase-T ethernet in 130nm CMOS," pp. 565–566, Jan 2012.
- [62] E. Kim and G. S. Choi, "LDPC code for reduced routing decoder," in *2005 Asia-Pacific Conference on Communications*, Oct 2005, pp. 991–994.
- [63] S. S. Tehrani, C. Jego, B. Zhu, and W. J. Gross, "Stochastic decoding of linear block codes with high-density parity-check matrices," *IEEE Transactions on Signal Processing*, vol. 56, no. 11, pp. 5733–5739, Nov 2008.
- [64] F. R. Kschischang, B. J. Frey, and H. A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Trans. Inf. Theor.*, vol. 47, no. 2, pp. 498–519, Sep. 2006.
- [65] F. Angarita, J. Valls, V. Almenar, and V. Torres, "Reduced-complexity min-sum algorithm for decoding LDPC codes with low error-floor," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 61, no. 7, pp. 2150–2158, July 2014.
- [66] V. A. Chandrasetty and S. M. Aziz, "An area efficient LDPC decoder using a reduced complexity min-sum algorithm," *Integr. VLSI J.*, vol. 45, no. 2, pp. 141–148, Mar. 2012.

- [67] H. Liu and A. Akoglu, "Timing-driven nonuniform depopulation-based clustering," *International Journal Reconfigurable Computing*, no. 3, pp. 1–11, Jan 2010.
- [68] G. Falcao, L. Sousa, and V. Silva, "Massively LDPC decoding on multicore architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 2, pp. 309–322, Feb 2011.
- [69] Z. Zhang, L. Dolecek, B. Nikolic, V. Anantharam, and M. J. Wainwright, "Design of LDPC decoders for improved low error rate performance: quantization and algorithm choices," *IEEE Transactions on Communications*, vol. 57, no. 11, pp. 3258–3268, Nov 2009.
- [70] X. Chen and V. Akella, "Exploiting data-level parallelism for energy-efficient implementation of LDPC decoders and DCT on an FPGA," *ACM Transactions on Reconfigurable Technology and Systems (TRETTS)*, vol. 4, no. 4, pp. 17–37, Dec. 2011.
- [71] M. Owaida, G. Falcao, J. Andrade, C. Antonopoulos, N. Bellas, M. Purnaprajna, D. Novo, G. Karakostas, A. Burg, and P. Ienne, "Enhancing design space exploration by extending CPU/GPU specifications onto FPGAs," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 2, pp. 33:1–33:23, Feb. 2015. [Online]. Available: <http://doi.acm.org/10.1145/2656207>
- [72] L. Yang, H. Liu, and C. R. Shi, "Code construction and FPGA implementation of a low-error-floor multi-rate low-density parity-check code decoder," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 53, no. 4, pp. 892–904, April 2006.
- [73] C. Beuschel and H. Pfeiderer, "FPGA implementation of a flexible decoder for long LDPC codes," in *2008 International Conference on Field Programmable Logic and Applications*, Sep. 2008, pp. 185–190.
- [74] X. Chen, J. Kang, S. Lin, and V. Akella, "Memory system optimization for FPGA-based implementation of quasi-cyclic LDPC codes decoders," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 1, pp. 98–111, Jan 2011.
- [75] P. Hailes, L. Xu, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, "Hardware-efficient node processing unit architectures for flexible LDPC decoder implementations," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 12, pp. 1919–1923, Dec 2018.
- [76] H. Li, Y. S. Park, and Z. Zhang, "Reconfigurable architecture and automated design flow for rapid FPGA-based LDPC code emulation," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. ACM, 2012, pp. 167–170.
- [77] Y. Cai, S. Jeon, K. Mai, and B. V. K. V. Kumar, "Highly parallel FPGA emulation for LDPC error floor characterization in perpendicular magnetic recording channel," *IEEE Transactions on Magnetics*, vol. 45, no. 10, pp. 3761–3764, Oct 2009.
- [78] X. Chen, J. Kang, S. Lin, and V. Akella, "Accelerating FPGA-based emulation of quasi-cyclic LDPC codes with vector processing," in *2009 Design, Automation Test in Europe Conference Exhibition*, April 2009, pp. 1530–1535.
- [79] X. Chen, J. Kang, S. Lin, and V. V. Akella, "Hardware implementation of a backtracking-based reconfigurable decoder for lowering the error floor of quasi-cyclic LDPC codes," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 12, pp. 2931–2943, Dec 2011.
- [80] L. Sun, H. Song, B. V. Kumar, and Z. Keirn, "Field-programmable gate-array-based investigation of the error floor of low-density parity check codes for magnetic recording channels," *IEEE transactions on magnetics*, vol. 41, no. 10, pp. 2983–2985, 2005.

- [81] H. Zhong, T. Zhang, and E. F. Haratsch, "High-rate quasi-cyclic LDPC codes for magnetic recording channel with low error floor," in *2006 IEEE International Symposium on Circuits and Systems*. IEEE, 2006, pp. 4–pp.
- [82] X. Hu, B. V. Kumar, L. Sun, and J. Xie, "Decoding behavior study of LDPC codes under a realistic magnetic recording channel model," *IEEE transactions on magnetics*, vol. 42, no. 10, pp. 2606–2608, 2006.
- [83] B. Unal, A. Akoglu, F. Ghaffari, and B. Vasić, "Hardware implementation and performance analysis of resource efficient probabilistic hard decision LDPC decoders," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 9, pp. 3074–3084, Sept 2018.
- [84] T. Richardson, "Error floors of LDPC codes," in *Proceedings of the annual Allerton conference on communication control and computing*, vol. 41, no. 3. The University; 1998, 2003, pp. 1426–1435.
- [85] B. Vasić, S. K. Chilappagari, D. V. Nguyen, and S. K. Planjery, "Trapping set ontology," *2009 47th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 1–7, Sept 2009.