FPGA Micro-Architecture-Code Co-Design For Low-Density Parity-Check Codes For Flash Memories

by

Reza Nakhjavani

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy

Department of The Edward S. Rogers Sr. Department of Electrical & Computer Engineering
University of Toronto

FPGA Micro-Architecture-Code Co-Design For Low-Density Parity-Check Codes For Flash Memories

Reza Nakhjavani
Doctor of Philosophy
Department of The Edward S. Rogers Sr. Department of Electrical & Computer Engineering
University of Toronto
2021

Abstract

The exponential growth of digital data has led to the proliferation of cloud storage systems as well as high-capacity, low-latency storage devices such as flash memory based solid-state drives (SSDs). These advances, along with the *technology shrink*, have increased the error rate both at system and device level. Storage device manufacturers and service providers often promise data reliability through error control coding. This dissertation is centered around the implementation of error correcting code (ECC) in data storage. In particular, we target low-density parity-check (LDPC) codes used for device-level and erasure codes used for system-level data reliability. Due to the ever-changing ECC requirements in the storage industry, we focus on field programmable gate arrays (FPGAs), given their short design cycles. Many studies on FPGA implementation of ECC focus on improving the hardware efficiency for a certain code of interest. This thesis extends this theme by considering hardware and code performance simultaneously. With the focus on ECCs used in data storage, we demonstrate a study of hardware-code co-design through an efficient FPGA micro-architecture that strikes a trade off between hardware efficiency and code performance. To this end, we leverage the FPGA's inherent physical architecture to propose an efficient reconfigurable micro-architecture for LDPC decoders. Then, we address the limitations of ECC in flash memories and define a finite decoder design space. Finally, we propose an end-to-end solution in which we leverage machine learning techniques to design a finite alphabet iterative decoder which strikes a trade off between hardware efficiency and code performance. In a separate effort, we perform a quantitative study of erasure coding design on FPGAs. We demonstrate, through probabilistic analysis, that an efficient implementation ought to allocate more resources to the common-case, while reducing the performance target for less probable cases.

Acknowledgments

I would like to express my deep gratitude to my advisor, Prof. Jianwen Zhu, for his constant support, encouragement and guidance. Over almost seven years, his insightful directions and advice, which are based on his broad knowledge, have facilitated the progress of my research, including the completion of this thesis. I also thank other committee members of my dissertation, Prof. Paul Chow, Frank Kschischang, Hans-Arno Jacobsen and Warren Gross for their academic acumen and constructive feedback. This thesis has also benefited from the excellent work of group members Wai Sum Mong, Shu-yi Wong, and Rami Baidas. Lastly, but most importantly, I would like to thank my wife, Negin, for their support, patience and strongest belief in me throughout this journey.

Computations of this thesis were partly performed on the Niagara supercomputer at the SciNet HPC Consortium. SciNet is funded by: the Canada Foundation for Innovation; the Government of Ontario; Ontario Research Fund - Research Excellence; and the University of Toronto.

Contents

Acknowledgements iii

Contents iv

List of Tables vii

List of Figures viii

List of Abbreviations xi

1 Introduction 1

1.1 Motivation . . . 1
1.2 Challenge . . . 2
1.3 Contributions . . . 2
1.4 Dissertation Organization . . . 2

2 Background 4

2.1 Coding Process . . . 4
2.1.1 Encoding & Decoding . . . 5
2.1.2 Modulation . . . 6
2.1.3 Channel Model . . . 7
2.2 Error Correcting Codes In Data Storage . . . 8
2.3 FPGA Architecture: Soft-Logic Vs. Hard-Logic . . . 9
2.4 Definitions . . . 10
2.4.1 Raw Bit Error Rate (RBER) . . . 10
2.4.2 Frame Error Rate (FER) . . . 10
2.4.3 Uncorrectable Bit Error Rate (UBER) . . . 11
2.4.4 Signal-to-Noise Ration (SNR) . . . 11
2.4.5 Shannon Limit . . . 12
2.5 Performance Metrics . . . 13

2.5.1 Implementation Performance . . . 13
2.5.2 System Performance . . . 13
2.5.3 Code Performance . . . 14

3 Low-Density Parity-Check Codes 15
3.1 Characteristics . . . 16
3.2 Iterative Decoding Process . . . 16
3.2.1 Hard-Decision Decoding . . . 19
3.2.2 Soft-Decision Decoding . . . 20
3.2.3 Summary . . . 22
3.2.4 Finite Alphabet Iterative Decoding (FAID) . . . 22
3.2.5 Layered Decoding . . . 22
3.3 Implementation Challenge . . . 23
3.4 Quasi-Cyclic (QC) LDPC . . . 24
3.5 QC-LDPC Decoder Architectural Features . . . 25
3.6 Prior Works . . . 27
3.7 LDPC Requirements for Flash Memories . . . 29
3.8 Our Work . . . 30

4 Configurable Micro-Architecture For QC-LDPC Decoder 32
4.1 Computations in QC-LDPC Decoders . . . 32
4.1.1 Check Node Processing . . . 33
4.1.2 Variable Node Processing . . . 34
4.2 Rotation in Hardware . . . 35
4.2.1 Naïve Implementation . . . 35
4.2.2 Foldable Parallel Rotation in Memory . . . 36
4.2.3 Leveraging FPGA's Hard Logic: Distributed RAM . . . 36
4.3 Rotary Register File: A Hybrid Shuffle Network . . . 37
4.3.1 Problem Statement . . . 37
4.3.2 Micro-Architecture . . . 40
4.3.3 FPGA-Optimized Implementation . . . 42
4.4 Decoder Architecture . . . 43
4.4.1 Pipeline Timing . . . 44
4.4.2 Summary of Architectural Features . . . 46
4.5 Noisy Gradient Decent Bit-Flipping Algorithm . . . 46
4.5.1 CNU . . . 49
4.5.2 VNU . . . 49
4.6 Evaluation . . . 50

4.6.1 RRF Performance . . . 51
4.6.2 NGDBF Decoder Performance . . . 54
4.7 Summary . . . 55

5 Learning FAID: A Hardware-Code Co-design 56
5.1 Overcoming Flash Memory Error Correction Constraints . . . 56
5.1.1 Constraint 1: Limited Quantization . . . 56
5.1.2 Constraint 2: Limited Decoding Iterations . . . 57
5.1.3 Leveraging A Finite Design Space . . . 58
5.2 Finite Alphabet Iterative Decoding (FAID) . . . 58
5.2.1 Framework . . . 58
5.2.2 FAID History . . . 60
5.3 Learning Decoder . . . 61
5.3.1 Framework . . . 61
5.3.2 History . . . 62
5.4 Learning FAID: A Decoder for A Finite Design Space . . . 63
5.4.1 Conventional APP-based Decoder . . . 63
5.4.2 Design Space . . . 64
5.4.3 Learning Framework . . . 65
5.4.4 Learning FAID Design . . . 68
5.5 FPGA Micro-Architecture . . . 69
5.5.1 Check Node Unit (CNU) . . . 70
5.5.2 VNU: A Two-Stage Lookup Table . . . 71
5.6 Code Performance Evaluation Framework . . . 72
5.6.1 Software Simulation . . . 73
5.6.2 Hardware Emulation . . . 74
5.7 Experimental Results . . . 79
5.7.1 Experimental Setup . . . 79
5.7.2 Training Process . . . 80
5.7.3 Code Performance . . . 81
5.7.4 Hardware-Code Performance . . . 84
5.7.5 FPGA Hard-Mux Impact . . . 86
5.8 Summary . . . 86

6 A Reconfigurable Architecture for Erasure Coding 88
6.1 Erasure Code . . . 89
6.1.1 Encoding . . . 90
6.1.2 Decoding . . . 90

6.2 Reliability Metric . . . 91
6.3 Prior Works . . . 92
6.4 Design Decisions . . . 93
6.4.1 A Case For Common-Case . . . 94
6.4.2 A Case for Coefficient Pre-Computation . . . 96
6.4.3 A Case for Reduced Computations . . . 97
6.5 Architecture . . . 99
6.5.1 Design for General-Case Decoder . . . 100
6.5.2 Design for Common-Case Decoder and Encoder . . . 100
6.6 Evaluation . . . 102
6.6.1 Erasure Codec Results . . . 102
6.6.2 The Impact of Disk Failure Probability . . . 103
6.6.3 Design Space Exploration . . . 104
6.7 Summary . . . 105

7 Summary & Conclusion **106**
7.1 Summary of Contributions . . . 106
7.2 Conclusion . . . 107
7.3 Future Work . . . 108

Bibliography **109**

List of Tables

3.1 Summary of LDPC decoding algorithms 22
3.2 Specifications, features, and efficiency of prior QC-LDPC Decoders . 28

4.1 Strided circular access with $z = 11$, $s = 4$, $f = 6$, with column-major layout $(\mathcal{L}, \mathcal{T})$. 41
4.2 Soft-MUX Vs. Hard-MUX RRFs for $\mathbf{n} = 251$ 53

5.1 Major prior efforts on FAID 61
5.2 The candidate QC-PaG codes for our experiments 79
5.3 The learned Weights 81
5.4 Hardware-code performance for C_1 85
5.5 FAID Hardware-code performance improvement over APP 85

6.1 The main ideas and achievements of the related works in erasure coding 93
6.2 Example 6.5:Pre-computation overhead for RS(14,10) code with $w = 8$ 96
6.3 Resource usage and performance results for our erasure code design . 103
6.4 Experimental results for design space exploration 104

List of Figures

2.1 Coding process. . . . 4
2.2 A BSC channel with error probability p. . . . 7
2.3 A BEC channel with error probability p. . . . 7
2.4 Error correcting codes in data storage 8
2.5 A typical architecture of an FPGA 9
2.6 Symmetric DMC with $X \in \{0, 1\}$ and $Y \in \{00, 01, 11, 10\}$ 12
2.7 Target UBER, Noise Threshold, and Error-floor 14

3.1 Tanner graph 17
3.2 Iterative decoding process: (a) Check node process (b) Variable node process 17
3.3 Soft-decision vs. hard-decision decoding for a (9706,1266) LDPC code 19
3.4 Message passing order: (a) Non-layered decoding Vs. (b) Layered decoding 23
3.5 Generic architecture for QC-LDPC decoders 25

4.1 Naïve implementations for circular memory access of size 8, (a) Barrel shifter (b) Shift register 36
4.2 Rotary register file structure 41
4.3 Logic sharing (a) 2-bit 2:1 multiplexer (b) Single-LUT6 realization . . 43
4.4 QC-LDPC decoder architecture based on RRF shuffle network 44
4.5 Pipelined QC-LDPC decoder timing diagram 45
4.6 Data flow graph for **H** in Equation 4.12 processing the first row of circulants 48
4.7 Data path for the VNU 49
4.8 Our evaluation framework 50
4.9 (a) Peak throughput and (b) Efficiency, for RRF with z=251 51
4.10 The *surplus set* size impact 52
4.11 RRF's efficiency 52
4.12 The impact of logic sharing, z=251 53

4.13 Parallelism Vs. performance for the QC-LDPC (9650, 1351) code . . 54
4.14 Foldable parallelism and performance 55

5.1 SLC voltage threshold distribution 57
5.2 Variable node computation (a) MS Vs. (b) APP-based 64
5.3 Neural network for training the decoder with 5 decoding iterations . . 66
5.4 FAID architecture based on the decoder described in Chapter 4 . . . 69
5.5 FAID check node unit architecture 70
5.6 FAID VNU lookup table implementation (a) Base-line (b) Reduced size (c) Two-stage 71
5.7 FAID check node unit architecture 72
5.8 The major steps towards evaluating the correction capability of a decoder 72
5.9 Software simulation framework for code performance evaluation . . . 73
5.10 Hardware simulation framework for code performance evaluation . . . 74
5.11 Parallel random generation logic 76
5.12 Probability distribution of the bits with value 0 in a SLC flash memory 77
5.13 Quantized AWGN random generation from a uniform random variable 78
5.14 Noisy soft information generation logic 78
5.15 Evolution of loss over training epochs for (a)C_1 (b)C_2 (c)C_3 (d)C_4 . . 80
5.16 Floating-point APP Vs. Learned APP for C_1, C_2, C_3, and C_4 81
5.17 Learned 4-bit FAID Vs. APP Vs. MS in 7-level sensing SLC for C_1 . 82
5.18 4-bit FAID Vs. APP in 7-level sensing SLC for with 25 maximum iterations 83
5.19 Decoding iterations for 4-Bit FAID and 4-bit APP for 7-level sensing SLC for (a) C_1, (b) C_2, (c) C_3, and (d) C_4 83
5.20 3-level Vs. 7-level sensing for C_1 84
5.21 2-stage Vs. 1-stage table FAID 84
5.22 The impact of leveraging FPGAs hard-logic on efficiency 86

6.1 Example 6.2: (a) Erasure coding Vs. (b) Replication 89
6.2 Probability of a recoverable failure with single failed block for $m = 4$ codes 95
6.3 High-level diagram for the erasure code unit 99
6.4 Block diagram for the general-case decoder 100
6.5 Block diagram for the common-case erasure code decoder and encoder 102
6.6 The impact of disk failure probability and comparison with the previous work 103

List of Abbreviations

APP	A Posteriori Probability
ASIC	Application-specific Integrated Circuit
AWGN	Additive White Gaussian Noise
BEC	Binary Erasure Channel
BP	Belief Propagation
BPSK	Binary Phase Shift Keying
BRAM	Block RAM
BSC	Binary Symmetric Channel
CDF	Cumulative Distribution Function
CNU	Check Node Unit
CP	Circulant Processor
CPM	Circulant Permutation Matrix
DE	Density Evolution
DMC	Discrete Memory Channel
ECC	Error Correcting Code
FAID	Finite Alphabet Iterative Decoding
FEC	Forward Error Correction
FER	Frame Error Rate
FPGA	Field Programmable Gate Array
GDBF	Gradient Decent Bit Flipping
GF	Galois Field
HDD	Hard Disk Drive
HDFS	Hadoop Distributed File System
HLS	High-level Synthesis
Inter-CW	Inter-Codeword
IoT	Internet Of Things
JEDEC	Joint Electron Device Engineering Council
JPL	Jet Propulsion Laboratory

LDPC	Low-density Parity-check
LFSR	Linear Feedback Shift Register
LLR	Log-likelihood Ratio
LRC	Locally-repairable Code
LUT	Lookup Table
MI	Mutual Information
MMI	Maximum Mutual Information
MS	Min-sum
NGDBF	Noisy Gradient Descent Bit-flipping
P/E	Program/erase
PCG	Permuted Congruential Generator
PDF	Probability Distribution Function
PMF	Probability Mass Function
QC-LDPC	Quasi-cyclic LDPC
QC-PaG	Quasi-cyclic Partial Geometry
RBER	Raw Bit Error Rate
ROC	Row Of Circulants
RRF	Rotary Register File
RS	Reed-Solomon
RTL	Register Transfer Level
SATA	Serial Advanced Technology Attachment
SIMD	Single Instruction Multiple Data
SLC	Single-level Cell
SNR	Signal-to-Noise Ratio
SSD	Solid-state Drive
UBER	Uncorrectable Bit Error Rate
VNU	Variable Node Unit

Chapter 1

Introduction

1.1 Motivation

The demand for data storage has been rapidly growing over the past decade. This growth has accelerated further over the past few years as a result of emerging technologies such as Internet of things (IoT), autonomous vehicles, and 5G networks. According to *Science Daily* report [1], in 2013, 90% of all existing data in the world had been generated over the previous two years. Meanwhile, the aggregated amount of the digital data present on our planet was about 30 Zettabytes (30×10^{12} GB) in 2017 and it is expected to reach 160 Zettabytes in 2025 with an exponential rate [2].

This has led to the proliferation of cloud storage services such as Amazon's S3 [3] and Microsoft's oneDrive [4] as well as high-capacity, low-latency storage devices such as flash memory based solid-state drives (SSDs). These advances, together with technology shrink have made *data reliability* a major concern both at the system and device levels.

The major vehicle to address data reliability is error control coding, exemplified by erasure coding in storage systems, and low-density parity-check (LDPC) [5] codes in storage devices. This dissertation concerns the implementation of ECC in storage. In particular, we focus on LDPC and erasure codes as the common codes used for device and system level data reliability.

The rapid changes in ECC specifications and requirements in storage applications demand a short time-to-market for any practical implementation. Therefore, field programmable gate array (FPGA) is a suitable reconfigurable hardware platform in this context. Moreover, *computational storage* is believed to be the next big thing and SNIA organization which leads the research in the storage industry has assigned a group to establish a taxonomy for it. It includes a large class of applications where some key algorithms are implemented within the storage controller. However, not all

algorithms justify having an ASIC implementation and even if they do, it may not be economically justifiable to go directly through that path which makes FPGA a suitable platform in this context.

1.2 Challenge

While many studies have been reported on the implementation of LDPC and erasure codes on FPGAs, with application areas well beyond storage, a common theme of the efforts is to implement a code of interest *efficiently*, where efficiency is quantified by the hardware resources employed.

This dissertation extends the theme in two deeper questions:

- Is there an FPGA-optimized micro-architecture for LDPC? (The FPGA micro-architecture problem)
- Is there an interesting trade off between LDPC micro-architecture and code performance? (The micro-architecture-code co-design problem)

1.3 Contributions

Towards addressing the raised questions, we make the following contributions:

- We leverage the FPGA's inherent physical architecture to design and implement an efficient configurable micro-architecture for LDPC decoders. The results show that our architecture is 11% more efficient compared to a barrel-shifter implementation of the medium-sized, Wimax802.16e LDPC code.
- Considering the inherent constraints of flash memories for LDPC decoding, we *learn* some aspects of the code to improve the code performance while simultaneously respecting the hardware efficiency. We show that our approach improves the code correction capability by 10%-18% at the cost of 4%-18% less efficient hardware.
- In a separate effort, we propose a configurable architecture for Reed-Solomon (RS) erasure coding which allocates more FPGA resources to common-case failures while reducing the performance target for rare failure cases.

1.4 Dissertation Organization

This thesis is organized as follows: Chapter 2 provides the basic concepts and definitions in reliable data storage, error correcting codes, and the performance metrics.

Chapter 3 describes the basic concepts for LDPC encoding and decoding as well as establishing a taxonomy for the hardware implementations and prior works in this field. Chapter 4 discusses our proposed reconfigurable FPGA micro-architecture for LDPC decoders to address the FPGA micro-architecture problem. In Chapter 5, we explain our learning based approach which aims to address the micro-architecture-code co-design problem. Our configurable architecture for the erasure code design is presented in Chapter 6 while Chapter 7 concludes the thesis.

Chapter 2

Background

This chapter provides some background knowledge on the coding process, and reliable data storage as well as defining the performance metrics used in our evaluations.

2.1 Coding Process

A message being transmitted over a noisy communication channel, must undergo the following steps for reliable data transmission (See Figure 2.1):

1. The k-bit information $\boldsymbol{u} = \begin{bmatrix} u_1 & u_2 & \cdots & u_k \end{bmatrix}$ is generated by the data source.
2. Using a channel coding mechanism, $\boldsymbol{u}$ is encoded to an n-bit codeword $\boldsymbol{c} = \begin{bmatrix} c_1 & c_2 & \cdots & c_n \end{bmatrix}$.
3. The transmitter converts $\boldsymbol{c}$ to a vector of symbols, $\hat{\boldsymbol{c}} = \begin{bmatrix} \hat{c}_1 & \hat{c}_2 & \cdots & \hat{c}_n \end{bmatrix}$.
4. $\hat{\boldsymbol{c}}$ is transmitted over the noisy channel. The received message at destination, $\hat{\boldsymbol{y}}$, is affected by channel noise: $\hat{\boldsymbol{y}} = \hat{\boldsymbol{c}} + \boldsymbol{n} = \begin{bmatrix} \hat{y}_1 & \hat{y}_2 & \cdots & \hat{y}_n \end{bmatrix}$.
5. The receiver performs quantization and converts $\hat{\boldsymbol{y}}$ to $\boldsymbol{y}$.
6. $\boldsymbol{y}$ is decoded to retrieve the codeword $\boldsymbol{d}$. The decoding is successful when $\boldsymbol{d} = \boldsymbol{c}$.

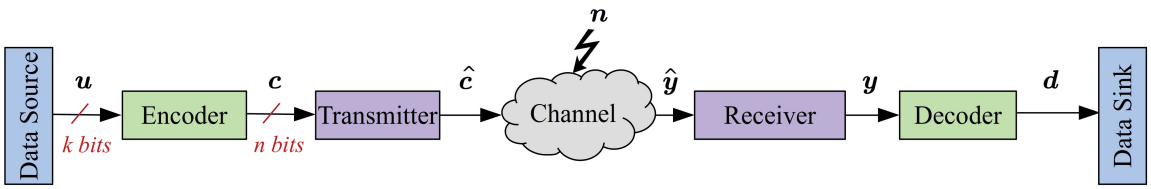


Figure 2.1: Coding process.

2.1.1 Encoding & Decoding

The presence of noise in a communication channel may change the value of symbols during transmission which necessitates an error correcting mechanism. Therefore, an error correcting code (ECC) is used to encode each message to a codeword with some added redundancy. This redundancy is then used at the destination for decoding to identify errors in the received message caused by the channel noise.

An $(n,\ k)$ linear block code encodes a message with k bits into a codeword with n bits adding $m = n - k$ parity bits. A linear code is defined by its encoding and decoding matrices. The former is known as the *generator* matrix , $\mathbf{G}_{k\times n}$, and the latter is known as the *parity check* matrix, $\mathbf{H}_{m\times n}$. During the encoding process the message is divided into k-bit blocks each being multiplied by $\mathbf{G}$ to generate an encoded codeword:

$$\boldsymbol{u}_{1\times k} \cdot \mathbf{G}_{k\times n} = \boldsymbol{c}_{1\times n} \tag{2.1}$$

During the decoding process, a hard-decision vector, $\bar{\boldsymbol{y}}$, is made from the values of the quantized vector, $\boldsymbol{y}$. Then $\mathbf{H}$ is multiplied to $\bar{\boldsymbol{y}}^T$ to generate the *syndrome*, $\boldsymbol{s}^T$:

$$\mathbf{H}_{m\times n} \cdot \bar{\boldsymbol{y}}^T_{n\times 1} = {\boldsymbol{s}^T}_{m\times 1} \tag{2.2}$$

If $\boldsymbol{s} = \mathbf{0}_{1\times m}$, then the received block is a codeword. Otherwise, further action is initiated to correct the errors.

Example 2.1 *Consider a (9,4) linear block code with the following generator and the parity check matrices:*

$$\mathbf{G}_{4\times 9} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \quad , \quad \mathbf{H}_{5\times 9} = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

and $\boldsymbol{u} = \begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix}$. *The codeword is generated by:*

$$\boldsymbol{c} = \begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{2.4}$$

The first 4 *bits are the original message bits while the last* 5 *bits are the parity bits.*

During the decoding process, assuming the decision vector, $\bar{\boldsymbol{y}}$, is same as the transmitted codeword, the syndrome matrix is computed as follows:

$$\boldsymbol{s}^T = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.5}$$

According to the above computations, we have:

$$\mathbf{H} \cdot \boldsymbol{c}^T = \mathbf{H} \cdot (\boldsymbol{u} \cdot \mathbf{G})^T = \mathbf{H} \cdot \mathbf{G}^T \cdot \boldsymbol{u}^T = \mathbf{0} \tag{2.6}$$

Since Equation 2.6 should be valid for any information vector, $\boldsymbol{u}$, we have:

$$\mathbf{H} \cdot \mathbf{G}^T = \mathbf{0} \tag{2.7}$$

Consequently, the parity check and generator matrices can be computed from each other.

A fundamental parameter to evaluate a (n, k) code is its rate defined as $R = \frac{k}{n}$. The extra $m = n - k$ transmitted bits are the parity bits added to enable error correction on the receiver side. In fact, code rate represents the fraction of a codeword that contains message bits. In general, higher-rate codes have less parity bit overhead and lower error correction capability.

2.1.2 Modulation

A signal has to be modulated in order to be transmitted over a channel. Common methods include frequency modulation, amplitude modulation, and phase modulation. This thesis assumes binary phase shift keying (BPSK) modulation in which each codeword bit is modulated according to the following equation:

$$\hat{c}_i = 1 - 2c_i \tag{2.8}$$

where c_i is the i^{th} bit of the codeword and $\hat{c}_i$ is the i^{th} modulated bit to be transmitted. Consequently, for a binary message, 0 and 1 codeword bits are modulated to $+1$ and -1 respectively.

2.1.3 Channel Model

A transmission channel is often defined by its noise characteristics. Three of the most well-known channel models are:

Binary Symmetric Channel (BSC)

A binary symmetric channel (BSC) describes a discrete memory channel (DMC) in which bits have equal probability to flip due to noise. Figure 2.2 illustrates a BSC channel in which every bit value has a probability, p, to flip while being transmitted over the channel.

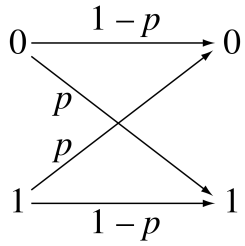


Figure 2.2: A BSC channel with error probability p.

Binary Erasure Channel (BEC)

A binary erasure channel (BEC) is a DMC with input symbols $\{0, 1\}$ and output symbols $\{0, 1, er\}$ where er is the erasure symbol and the input symbols have probability p to change to er at the receiver due to noise.

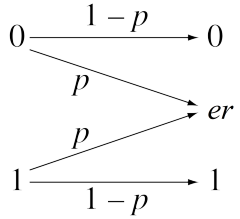


Figure 2.3: A BEC channel with error probability p.

Figure 2.3 illustrates a BEC channel. Unlike BSC, in a BEC channel when the receiver gets a 0 or 1, the bit is certainly correct. In other words, in an erasure channel, the error locations in the received message is known prior to the decoding process.

Additive White Gaussian Noise Channel

Additive white Gaussian noise (AWGN) is a continuous channel model commonly used in many applications such as satellite and deep-space communications. The noise in

this channel has a normal distribution with mean 0 and variance σ^2. Therefore, we have:

$$\hat{y}_i = \hat{c}_i + n_i, \qquad \hat{c}_i \sim N(0, \sigma^2) \tag{2.9}$$

where $\hat{y}_i$ is the value of bit i at the receiver, $\hat{c}_i$ is the value of bit i at the transmitter, and n_i is the channel noise for bit i.

2.2 Error Correcting Codes In Data Storage

Although error correcting codes were initially designed for communication channels, they are also used in data storage systems to prevent data loss and increase data durability. Occurrence of a failure in a storage device may result in change or loss of the stored data. As technology scales down, the failure rate increases and data storage devices rely more on ECC to provide data reliability. The failure cause depends on the physical properties of the underlying storage device. For instance, in flash memories, it includes:

- Read disturb: Reading a cell my affect the values stored in its neighbouring cells.
- Program disturb: Programming a cell requires applying a high voltage to the selected word-line which may affect the value stored in the neighbouring cells.
- Program/erase (P/E) cycles: The erase process in flash memories involves applying a relatively large electrical charge to flash cells. This causes the cells to degrade over time which results in a growth in bit-error rate as the memory ages.
- Retention issues: When data is stored in flash memory cells, the leakage may lead to a change of value in some cells.

The failures in a data storage system can be modelled as the noise in a communication channel (Figure 2.1) and ECC can be utilized to protect the data. Figure 2.4 illustrates the coding process utilized in a data storage device. The raw data needs to be encoded when it is written to the disk. A fraction of storage capacity is devoted to parity bits obtained through the encoding process. When retrieving the data from the disk, it has to be decoded to correct the errors (if any) through a decoding process.

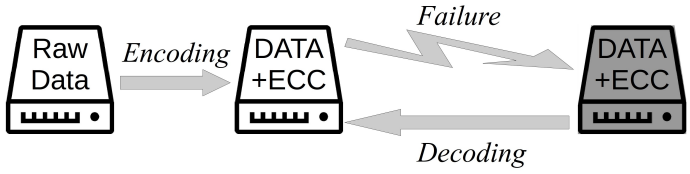


Figure 2.4: Error correcting codes in data storage

Utilizing ECC to improve data integrity imposes storage and access latency overhead. The former is caused by the required space to store the parity bits, while the latter is due to the encoding and decoding procedures. However, these overheads can be mitigated by using high-rate ECCs along with efficient encoding and decoding implementations. In this thesis, we focus on improving the efficiency of high-rate ECCs for data storage. Since decoding is usually the challenging step due to its iterative process, we aim to propose efficient decoder architectures to improve the read latency for high-rate codes.

2.3 FPGA Architecture: Soft-Logic Vs. Hard-Logic

Figure 2.5 illustrates the architecture of a typical FPGA. It consists of a set of configurable logic blocks (CLBs), memory blocks, and digital signal processors (DSPs). The CLBs consist of a set of lookup tables, implemented using memory elements and multiplexers. The memory blocks are the natural memory resources in FPGA that can be used to store the intermediate date during computation. The DSPs are essentially efficient multipliers that are beneficial in applications such as image processing and filters.

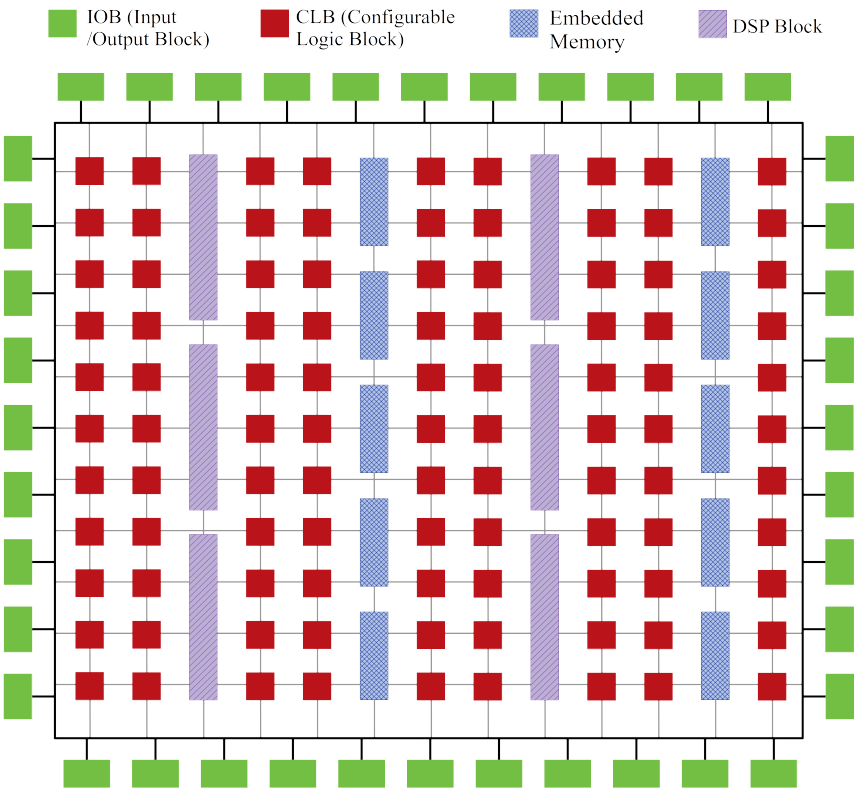


Figure 2.5: A typical architecture of an FPGA

Although CLB's lookup tables are designed to implement any logical function, considering the inherent physical architecture of the FPGA can lead to a significantly more efficient design. For instance, a multiplier that is implemented using the DSP blocks is remarkably more efficient than implementing the multiplication logic using the lookup tables. Similarly, a multiplexing logic that is realized through the existing multiplexers within th CLBs has a notably higher efficiency than a multiplexer that is implemented using the lookup tables. Utilizing the FPGA's physically implemented components is referred to as a *hard-logic* implementation. On the other hand, using FPGA's lookup tables to realize a logical function is referred to as a *soft-logic* implementation.

2.4 Definitions

2.4.1 Raw Bit Error Rate (RBER)

Let $\bar{\boldsymbol{y}}$ be the hard-decision vector of the $\hat{\boldsymbol{y}}$ received from the channel. We define the following Bernoulli decision function:

$$I(\bar{y}_i) = \begin{cases} 1 & \text{if } \bar{y}_i \neq c_i, \\ 0 & \text{otherwise.} \end{cases} \tag{2.10}$$

Therefore, raw bit error rate (RBER) can be expressed as:

$$p_e = P(\bar{y}_i \neq c_i) = P[I(\bar{y}_i) = 1] = E[I(\bar{y}_i)] \tag{2.11}$$

where $E[.]$ is the expectation operator [6]. In practice, using multiple realizations of the transmitter and the channel in the Monte-Carlo simulation, the RBER can be estimated using the ensemble average:

$$\hat{p}_e = \frac{1}{N}\sum_{i=1}^{N} I(\bar{y}_i) = \frac{BitErrors}{TotalBits} \tag{2.12}$$

2.4.2 Frame Error Rate (FER)

The frame error rate (FER) represents the number of codewords that are not decoded successfully. Considering an ECC with codeword size n which is able to correct up to t errors, FER can be computed as follows:

$$FER = 1 - \sum_{i=0}^{t} \binom{n}{i} p_e^i (1-p_e)^{n-i} \tag{2.13}$$

Where p_e is the RBER. It is impossible for some ECCs to theoretically compute the number of errors they can correct, t. In that case, FER can be estimated using the Monte-Carlo simulation:

$$F\hat{E}R = \frac{Unsucessful\ decodes}{Total\ decodes} \tag{2.14}$$

2.4.3 Uncorrectable Bit Error Rate (UBER)

The uncorrectable bit error rate (UBER) is defined as the bit error rate after the error correction process. The details of computing the UBER is discussed in the joint electron device engineering council (JEDEC) standard, JESD218 [7]. As a common practice, UBER at any instant is estimated as:

$$UBER = \frac{FER}{A} \tag{2.15}$$

where A is defined as the codeword size (n) [8], or the data bits per codeword (k) [9]. In this thesis, we compute UBER assuming A is the codeword size.

2.4.4 Signal-to-Noise Ration (SNR)

SNR is a metric that projects the level of a desired signal to the level of background noise. It is defined as the ratio of the signal power to the noise power, and it is often expressed as $\frac{E_b}{N_0}$, *i.e.*, energy per bit over noise. For a code with rate R being transmitted over an AWGN channel, SNR is computed as follows:

$$\frac{E_b}{N_0} = \frac{1}{2R\sigma^2} \tag{2.16}$$

where σ^2 is the variance of the noise. This ratio is often represented in decibels as follows:

$$\frac{E_b}{N_0}\ \text{[dB]} = 10\log_{10}(\frac{E_b}{N_0}) \tag{2.17}$$

In coding theory, SNR and RBER are related parameters. In fact, higher SNR leads to lower RBER. For an AWGN channel with BPSK modulation, the relationship between SNR and RBER is as follows:

$$RBER = \frac{1}{2}erfc\left(\sqrt{\frac{E_b}{N_0} \times R}\right) \tag{2.18}$$

where R is the code rate, and $erfc()$ is the complementary error function[10].

2.4.5 Shannon Limit

During the late 1940's, Claude Shannon established the *noisy channel coding theorem* [11]. Given the degree of noise in a channel, the theorem can determine the maximum rate discrete data can be transferred through that channel. The *Shannon limit* represents the maximum theoretical transfer rate over a noisy channel. For a continuous AWGN channel, *i.e.*, continuous input continuous output channel, the maximum capacity can be computed as follows:

$$C = B \log_2 (1 + SNR) \tag{2.19}$$

where B is the channel bandwidth and SNR is the signal-to-noise ratio.

The Shannon limit for a DMC can be computed through back calculation from the mutual information (MI) of input and output random variables. In fact, the MI is a measure of dependence between the two random variables. Let $X \in \{0, 1\}$ and Y be the input and output random variables for a DMC with binary input. Assuming $P(X = 0) = P(X = 1)$, the MI, $I(X;Y)$, between the input and output random variables can be calculated [12] as:

$$I(X;Y) = H(Y) - H(Y|X) \tag{2.20}$$

where H is the *entropy* function [13].

Figure 2.6 illustrates a symmetric DMC with $X \in \{0, 1\}$ and $Y \in \{00, 01, 11, 10\}$ where p_1, p_2, p_3, p_4 are the crossover probabilities. These probabilities can be computed based on the channel noise distribution and the output quantization thresholds.

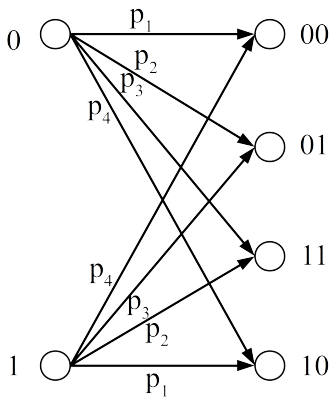


Figure 2.6: Symmetric DMC with $X \in \{0, 1\}$ and $Y \in \{00, 01, 11, 10\}$

In this case, the mutual information can be computed as follows:

$$I(X;Y) = H(Y) - H(Y|X) = H(\frac{p_1+p_4}{2}, \frac{p_2+p_3}{2}, \frac{p_3+p_2}{2}, \frac{p_4+p_1}{2}) - H(p_1, p_2, p_3, p_4) \tag{2.21}$$

The computation can simply be extended to DMCs with larger quantization in their output.

Given a channel noise distribution, it is possible to calculate the output quantization thresholds that results in p_1, p_2, p_3, p_4 values which, in turn, results in maximum mutual information (MMI) [12]. In fact, MMI represents the maximum code rate that can be transferred reliably through the given noisy channel. Therefore, given a code rate it would be possible, through interpolation, to back calculate the minimum SNR above which the data can be transferred reliably over the DMC, *i.e.*, Shannon limit [12].

2.5 Performance Metrics

Any system with error-correcting capability always faces two major questions; correction capability and implementation efficiency. A set of quantitative metrics are defined to address these questions.

2.5.1 Implementation Performance

The efficiency of any computational unit can be evaluated based on the ratio of its peak throughput to its utilized resources, where the resource is the efficiency subject. In this work, we focus on ECC decoding micro-architectures for FPGAs. Since these algorithms are compute-intensive, the efficiency of their architectures can be evaluated based on the number of lookup tables (LUTs) used to implement the algorithm. Therefore, we define *Throughput/1 Thousand LUTs* as the efficiency metric and it is measured as MBps/KLUT.

2.5.2 System Performance

Availability

In fault tolerant systems, *Availability* is defined as the fraction of time during which the system is up and running:

$$Availability(\mathcal{A}) = \frac{UpTime}{UpTime + DownTime} = \frac{MTTF}{MTTF + MTTR} \tag{2.22}$$

where $MTTF$ is mean time to failure, and $MTTR$ is mean time to repair.

Reliability

Reliability is defined as the probability of a system being up and running:

$$Reliability(\mathcal{R}) = 1 - P(fail) \tag{2.23}$$

where $P(fail)$ is the probability that the system fails.

2.5.3 Code Performance

Target UBER

Target UBER is the UBER below which a system is considered to be reliable. This metric is set based on application demands.

Noise Threshold

Noise threshold is defined as the power level of a signal above which the signal is discernible. It is the metric to evaluate an ECC system and can be defined as the lowest SNR above which the system can achieve a target UBER. In fact, noise threshold determines the maximum noise level that can be tolerated by an ECC. Therefore, higher noise threshold translates to higher error correcting capability. Noise threshold can also be expressed in terms of the bit error rate as well.

Error-Floor

Generally, every ECC can provide lower UBER as RBER declines. However, for some codes, there is a point beyond which UBER does not decrease with the same pace as the RBER is decreased. This phenomenon, known as *error-floor*, is illustrated in Figure 2.7.

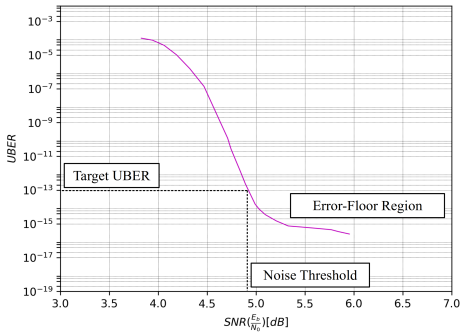


Figure 2.7: Target UBER, Noise Threshold, and Error-floor

Chapter 3

Low-Density Parity-Check Codes

Low-density parity-check codes are a class of linear block error correcting codes that was invented in the 60's by Robert G. Gallager [5]. Despite being a capacity approaching code, LDPC did not receive the deserved attention until its rediscovery by MacKay in 1996 [14] due to the limited computational resources in integrated circuits.

In 1998, MacKay presented a talk in NASA's jet propulsion laboratory (JPL) to discuss the feasibility of LDPC codes replacing turbo codes in NASA's deep-space applications. The talk was a renaissance for LDPC codes and led to more coding research around the world. Compared to turbo codes, LDPC codes had more degrees of design freedom that enabled designers to effectively strike a trade off between noise threshold for decoding throughput [15]. Since 2009, LDPC has been used in JPL's missions for deep-space networks and satellite communications. There is still ongoing research on improving LDPC performance as well as applying it to other applications.

LDPC has recently been considered as ECC in flash memories due to their high error-correcting capability. As the number of program/erase (P/E) cycles of a flash memory increases, the RBER is increased making the memory less reliable. In fact, this phenomenon is exacerbated as technology shrink increases the threshold voltage variation which results in a higher error rate specially when the number of P/E cycles is quite high [16]. Compared to BCH codes, the higher noise threshold of LDPC codes makes them a suitable ECC for flash memories to extend their lifetime.

In this chapter, we explain the basic concepts of LDPC codes as well as the architectural choices for an LDPC decoder and the *state-of-the-art* LDPC decoders. In particular, we focus on quasi-cyclic LDPC (QC-LDPC) codes, a class of LDPC codes that are not only suitable for hardware implementation, but also of good code performance.

3.1 Characteristics

As its name suggests, LDPC is a class of linear block codes that has a sparse parity check matrix, $\mathbf{H}$. Let w_i^r be the number of 1s in the i^{th} row of $\mathbf{H}$, *i.e.*, row weight, and w_j^c be the number of 1s in the j^{th} column of $\mathbf{H}$, *i.e.*, column weight. Then, the parity check matrix of a (n, k) LDPC code has the following properties:

$$\begin{aligned} \forall \quad 1 \leq i \leq m, \quad w_i^r \ll n \\ \forall \quad 1 \leq j \leq n, \quad w_j^c \ll m \end{aligned} \tag{3.1}$$

where $m = n - k$. Moreover, an LDPC code is *regular* if all rows and columns of its parity check matrix have the same weight:

$$\begin{aligned} \forall \quad 0 \leq i < m, \quad w_i^r = w^r \\ \forall \quad 0 \leq j < n, \quad w_j^c = w^c \end{aligned} \tag{3.2}$$

Upon arrival at the receiver, the decoder checks if the syndrome vector is zero ($\boldsymbol{s} \stackrel{?}{=} \mathbf{0}$). If so, The received block is a codeword. Otherwise, an iterative process is initiated to decode the received block and get the codeword.

3.2 Iterative Decoding Process

The most intuitive representation of the iterative decoding of an LDPC code is through considering $\mathbf{H}$ as the bi-adjacency matrix of a bipartite graph known as the *Tanner Graph*. The graph consists of two sets of nodes: *variable nodes* and *check nodes*. Each variable node represents a bit in the codeword and a column in $\mathbf{H}$. Conversely, each check node represents a parity constraint and a row in $\mathbf{H}$. An edge exists between the check node i and the variable node j if and only if the entry (i, j) of $\mathbf{H}$ is 1, *i.e.*, the j^{th} bit of the codeword participates in the i^{th} parity constraint.

The number of edges connected to a variable/check node is referred to as the variable/check node degree. In the Tanner graph representation of a regular LDPC code, all variable nodes have the same degree, d_v, and all check nodes have the same degree, d_c. Such a code is denoted as a (d_v, d_c)-regular LDPC code and according to Equation 3.2, $d_v = w^c$ and $d_c = w^r$.

Example 3.1 *Let* $\mathbf{H}$ *in Equation 3.3 be the parity check matrix for a (*9,3*) LDPC code:*

$$\mathbf{H}_{6\times 9} = \begin{array}{c} \\ C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \end{array} \begin{array}{c} \begin{array}{ccccccccc} V_0 & V_1 & V_2 & V_3 & V_4 & V_5 & V_6 & V_7 & V_8 \end{array} \\ \left[\begin{array}{ccccccccc} 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{array}\right] \end{array} \tag{3.3}$$

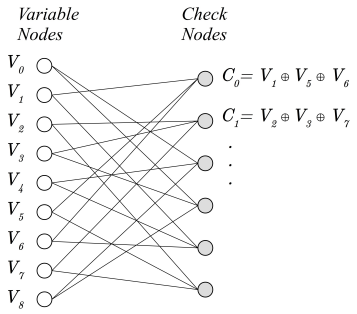


Figure 3.1: Tanner graph

Then, the corresponding Tanner graph of $\mathbf{H}$ *is illustrated in Figure 3.1. It consists of* 9 *variable nodes,* i.e., *columns in* $\mathbf{H}$*, and* 6 *check nodes,* i.e., *rows in* $\mathbf{H}$*. We have* $d_v = w^c = 2$ *and* $d_c = w^r = 3$.

The iterative decoding process is performed by message passing between connected variable and check nodes as depicted in Figure 3.2. Let C_i be a check node connected to V_j, the variable node corresponding to the j^{th} received codeword bit. Initially, each variable node sends a message regarding the value of its corresponding received codeword bit to the connected check nodes. Then, the following iterative message passing is performed:

1. **Check node process**: C_i checks if its corresponding parity constraint is satisfied based on the received bits. Then, it computes a message, $r_{i,j}$, and sends it to V_j. The message contains C_i's belief about the correct value of the j^{th} received bit.

2. **Variable node process**: V_j computes a message, $l_{i,j}$, and sends it to C_i. The message contains V_j's belief about the correct value of the j^{th} received bit.

The computations in the variable and check nodes depend on the decoding algorithm. The check and variable node processes are equivalent to processing rows and columns of $\mathbf{H}$ respectively.

Figure 3.2: Iterative decoding process: (a) Check node process (b) Variable node process

Example 3.2 *Let* $\mathbf{H}$ *in Equation 3.3 be the parity check matrix for a* $(9, 3)$ *LDPC code. According to the first row of* $\mathbf{H}$*,* C_0 *receives messages from* V_1*,* V_5*, and* V_6 *for computation. On the other hand,* V_0 *receives messages from* C_2 *and* C_3 *to perform its computations.*

In each decoding iteration, the decoder first processes the rows of $\mathbf{H}$. Once all check nodes are processed, the columns of $\mathbf{H}$ are processed for the variable node processing. This iterative process, continues until either all the check node constraints are satisfied or a predefined number of iterations is reached.

Algorithm 1 describes a generic iterative message passing decoder for a $(n,\ k)$ LDPC code with n variable and $m = n - k$ check nodes. Consider the following definitions:

- $\mathcal{V}_i$: The set of variable node indices connected to check node i.
- $\mathcal{C}_j$: The set of check nodes indices connected to variable node j.
- $r_{i,j}^{(t)}$: The message sent from the check node C_i to the variable node V_j at the t^{th} iteration.
- $l_{i,j}^{(t)}$: The message sent from the variable node V_j to the check node C_i at the t^{th} iteration.

Algorithm 1 Generic Message Passing Algorithm

for $0 \leq j < n$ **do**
  for $i \in \mathcal{C}_j$ **do**
    $l_{i,j}^{(0)} = \mathbf{I}(y_j)$;
  end for
end for
for $1 \leq t \leq maxIter$ **do**
  // Check Node Process
  for $0 \leq i < m$ **do**
    for $j \in \mathcal{V}_i$ **do**
      $r_{i,j}^{(t)} = \mathbf{\Psi}(\mathcal{V}_i, j, t-1)$;
    end for
  end for
  // Variable Node Process
  for $0 \leq j < n$ **do**
    for $i \in \mathcal{C}_j$ **do**
      $l_{i,j}^{(t)} = \mathbf{\Phi}(y_j, \mathcal{C}_j, i, t)$;
    end for
    // Hard Decision for each bit
    $\bar{y}_j = \mathbf{\Omega}(y_j, \mathcal{C}_j, t)$;
  end for
  // Early Exit Condition
  if $\mathbf{H} \cdot \bar{\boldsymbol{y}}^T = \mathbf{0}$ **then**
    break;
  end if
end for

▷ n: Number of variable nodes
▷ m: Number of check nodes
▷ $maxIter$: Maximum number of decoding iterations

Lines $1-5$ of the Algorithm 1 are for the initial messages sent from the variable nodes to the check nodes. The check node processing is performed at lines $8-12$ and the variable node processing is at lines $14-20$. During each iteration the variable nodes also make a hard decision for their corresponding codeword bit value based on the received messages and the check nodes compute the syndrome matrix to check if the received codeword has been successfully decoded.

The main difference between LDPC decoding algorithms comes from their implementations of the $\mathbf{I}(.)$, $\mathbf{\Psi}(.)$, $\mathbf{\Phi}(.)$, and $\mathbf{\Omega}(.)$ functions in Algorithm 1. These algorithms are mainly divided into two major categories, *hard-decision* and *soft-decision*. The former receives a single-bit (hard) information from the channel, while the latter receives multi-bit (soft) information from the channel. A single-bit information can only determine whether a received bit is 1 or 0. On the other hand, a multi-bit information, known as *soft information*, is more accurate since it contains a probability for a bit being 0 or 1. In general, soft-decision decoders achieve higher noise threshold at the cost of increased implementation complexity.

Example 3.3 *Figure 3.3 compares the performance for soft-decision and hard-decision decoding algorithms for a (9706,1266) LDPC code. With respect to the UBER* 10^{-6}*, the soft-decision algorithm's noise threshold is* 1.8 *dB higher than the hard-decision algorithm.*

3.2.1 Hard-Decision Decoding

In hard-decision decoding, no soft information is received from the channel. The parity equations are computed in the check nodes based on the hard information received from the variable nodes. Then, the check nodes send a single bit to their connected variable nodes. If a check node C_i sends 1 to a variable node V_j, then it means C_i is not satisfied and it is suggesting that V_j should be flipped. The node

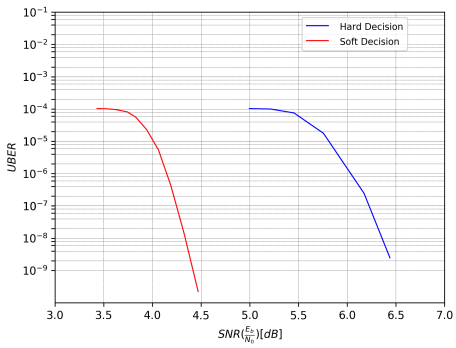


Figure 3.3: Soft-decision vs. hard-decision decoding for a (9706,1266) LDPC code

V_j will then make a hard-decision on whet to flip its corresponding bit based on the received messages from all the check nodes. Different hard-decision algorithms differ on their variable node hard-decision policy.

Gallager-A

In the hard-decision Gallager-A decoding algorithm, the variable node V_i with degree d_i flips its corresponding bit if $d_i - 1$ received messages from the check node are 1.

3.2.2 Soft-Decision Decoding

In soft-decision algorithms, multi-bit information is received from the channel. In this section, we discuss the most well-known soft-decision algorithm, *min-sum (MS)*, as well as the *a posteriori probability (APP)* based algorithm which simplifies the variable node process to enable a hardware-friendly implementation. Moreover, we discuss the *noisy gradient decent bit flipping* (NGDBF) algorithm which simplifies the messages to single bit for further simplification complexity. We explain the algorithms based on the terminology defined for Algorithm 1.

Min-Sum (MS) Decoding

The most well-known soft-decision LDPC decoding algorithm is the MS algorithm. Initially, the log-likelihood ratio (LLR) for each received bit is computed by:

$$l_{i,j}^{(0)} = \mathbf{I}(j) \triangleq LLR(y_j) = \ln\left[\frac{P(c_j = 0 \mid y_j)}{P(c_j = 1 \mid y_j)}\right] \tag{3.4}$$

where y_j is the soft information for the j^{th} codeword bit received from the noisy channel, and e_j is its correct value. In fact, $P(c_j = 0|y_j)$ is the conditional probability that the correct value for the j^{th} bit is 0, given the received soft information y_j. The message sent from C_i to v_j, $r_{ij}^{(t)}$, is:

$$r_{i,j}^{(t)} = \mathbf{\Psi}(\mathcal{V}_i, j, t-1) = \prod_{k \in \mathcal{V}_i/j} sign(l_{i,k}^{(t-1)}) \times \min_{k \in \mathcal{V}_i/j} |l_{i,k}^{(t-1)}| \tag{3.5}$$

The message sent from V_j to C_i, $l_{i,j}^{(t)}$, is:

$$l_{i,j}^{(t)} = \mathbf{\Phi}(y_j, \mathcal{C}_j, i, t) = LLR(y_j) + \sum_{k \in \mathcal{C}_j/i} r_{k,j}^{(t)} \tag{3.6}$$

During each decoding iteration, the variable node also has to compute a hard decision for the bit value:

$$\bar{y}_j = \mathbf{\Omega}(y_j, \mathcal{C}_j, t) = \begin{cases} 0, & LLR(y_j) + \sum_{k \in \mathcal{C}_j} r_{k,j}^{(t)} \geq 0, \\ 1, & \text{otherwise.} \end{cases} \tag{3.7}$$

A Posteriori Probability (APP) Based Decoding

According to Equation 3.6, each variable node in the MS decoder sends different messages to each one of their connected check nodes due to the exclusion of i in the summation. In fact, a hardware implementation of the MS decoder for a variable node with degree d_v requires d_v adders. The APP based decoding algorithm simplifies the variable node computation by sending identical messages to all connected check nodes [17].

$$l_{i,j}^{(t)} = \mathbf{\Phi}(y_j, \mathcal{C}_j, i, t) = LLR(y_j) + \sum_{k \in \mathcal{C}_j} r_{k,j}^{(t)} \tag{3.8}$$

Therefore, the variable node computation in the APP decoding can be accomplished using a single adder. This results in a significant computation reduction for LDPC codes with a large number of variable nodes. However, it imposes a correlation between the outgoing messages of a variable node which degrades code performance. In order to diminish the impact of this correlation, a normalization process is performed for the check node operation:

$$r_{i,j}^{(t)} = \mathbf{\Psi}(\mathcal{V}_i, j, t-1) = \prod_{k \in \mathcal{V}_i / j} sign(l_{i,k}^{(t-1)}) \times \min_{k \in \mathcal{V}_i / j} |l_{i,k}^{(t-1)}| \times \alpha \tag{3.9}$$

where $0 < \alpha < 1$ is the normalization factor.

Simulation results confirm that the APP decoding has a slight performance degradation compared to the MS algorithm [18]. However, the hardware implementation of the variable node can be significantly simplified.

Noisy Gradient Decent Bit Flipping (NGDBF)

NGDBF [19] is a soft-decision decoding algorithm that uses single-bit messages during the decoding procedure. In NGDBF, the check nodes perform a simple XoR operation to check if their corresponding parity constraint is satisfied. The variable nodes use the channel's soft information in order to decide whether to flip the hard decision for their corresponding bit. Despite using soft information, NGDBF does not update the soft information in every decoding iteration which, in turn, results in a simpler

Table 3.1: Summary of LDPC decoding algorithms

FUNCTION()	Soft Decision			Hard Decision
	Min-Sum	**APP**	**NGDBF**	**Gallager-A**
$\mathbf{\Psi}()$	$\prod_{k\in\mathcal{V}_i/j} sign(l_{i,k}^{(t-1)})\times \min_{k\in\mathcal{V}_i/j} \lvert l_{ki}^{(t-1)}\rvert$	$\prod_{k\in\mathcal{V}_i/j} sign(l_{i,k}^{(t-1)})\times \min_{k\in\mathcal{V}_i/j} \lvert l_{ki}^{(t-1)}\rvert \times \alpha$	$1-2\times \bigoplus_{k\in\mathcal{V}_i} l_{i,k}^{(t-1)}$	$\bigoplus_{k\in\mathcal{V}_i} l_{i,k}^{(t-1)}$
$\mathbf{\Phi}()$	$LLR(y_j)+\sum_{k\in\mathcal{C}_j/i} r_{k,j}^{(t)}$	$LLR(y_j)+\sum_{k\in\mathcal{C}_j} r_{k,j}^{(t)}$	$\sum_{k\in\mathcal{C}_j} r_{k,j}^{(t)}$	$\sum_{k\in\mathcal{C}_j} r_{k,j}^{(t)}$
$\mathbf{\Omega}()$	$\begin{cases} 0, & LLR(y_j)+\sum_{k\in\mathcal{C}_j} r_{k,j}^{(t)} >= 0 \\ 1, & otherwise \end{cases}$		$y_j+\sum_{k\in\mathcal{C}_j} r_{k,j}^{(t)} < \theta$	$\sum_{k\in\mathcal{C}_j} r_{k,j}^{(t)} > \lvert\mathcal{C}_j\rvert - 1$

hardware implementation compared to other soft-decision decoders. In fact, NGDBF strikes a trade-off between hard-decision and other soft-decision decoding algorithms in terms of error correction capability and hardware efficiency. The details of this algorithm are discussed in Chapter 4.

3.2.3 Summary

Table 3.1 summarizes the main operations in soft-decision and hard-decision LDPC decoding algorithms. The $\mathbf{\Omega}()$ functions for the NGDBF and Gallager-A determine whether the j^{th} bit has to be flipped.

3.2.4 Finite Alphabet Iterative Decoding (FAID)

Finite alphabet iterative decoders are a class of non-conventional decoders which have received more attention in the recent years due to their higher code performance in certain applications. Unlike conventional decoding algorithms, the messages in FAID are not the likelihood values for each received bit. Instead, they are chosen from a finite set of alphabet that are generated through a table lookup operation. We believe the properties of these decoders make them suitable for flash memory error correction and will discuss them in details in Chapter 5.

3.2.5 Layered Decoding

Layered decoding is a technique used in LDPC decoders to increase the decoding convergence rate so that fewer decoding iterations are required. Figure 3.4 compares the message passing order in the layered and non-layered decoders. The numbers on the arrows show the communication order. In a non-layered decoder (Figure 3.4a), a variable node sends its messages to all check nodes and then waits for their responses. On the other hand, in a layered decoder (Figure 3.4b), the variable node utilizes each check node's response to prepare its message for the next check node.

(a) (b)

Figure 3.4: Message passing order: (a) Non-layered decoding Vs. (b) Layered decoding

From a parity check matrix perspective, the non-layered decoder waits for all rows of **H** before updating the variable node values. In contrast, the layered decoder does not wait for all rows of **H** to be processed before updating the variable nodes. In general, these partial variable node updates within a decoding iteration are expected to results in faster convergence.

3.3 Implementation Challenge

Although LDPC codes with large codewords (1 KB - 4 KB) have higher noise threshold, implementing a decoder for them is challenging. In an ideal fully parallel implementation, each variable and check node has a separate processing unit so that all rows and columns of **H** are processed simultaneously. However, such a realization is not feasible due to its significant amount of required computational and routing resources. Consequently, a partially parallel implementation is desired to process a subset of the rows/columns of **H** simultaneously. Partial parallelism not only enables implementing decoders for LDPC codes with large codeword, but it can also benefit from layered decoding due to its sequential processing of the rows of **H**.

On the other hand, communicating messages between the nodes in a LDPC decoder with large codeword through physical links results in a highly congested network which makes the routing challenging. One solution is to utilize a shared-memory message passing mechanism in which the nodes can communicate through memory *Read/Write* operations. This research is aimed at overcoming these challenges and proposing efficient LDPC decoder architectures.

3.4 Quasi-Cyclic (QC) LDPC

Designing a partially parallel LDPC decoder requires identifying a pattern in the parity check matrix that enables folding the processing order for rows/columns. QC-LDPC codes are a special class of LDPC codes which their parity check matrices have a quasi-cyclic structure. In fact, an LDPC code is quasi-cyclic if its parity check matrix has the following structure:

$$\mathbf{H} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} & \dots & \mathbf{C}_{1,c} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} & \dots & \mathbf{C}_{2,c} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{C}_{r,1} & \mathbf{C}_{r,2} & \dots & \mathbf{C}_{r,c} \end{bmatrix} \tag{3.10}$$

where $\mathbf{C}_{i,j}$ is a cyclic matrix known as *circulant*, *i.e.*, each row of $\mathbf{C}_{i,j}$ can be obtained by rotating its previous row. Assuming each circulant is a $z \times z$ matrix, $\mathbf{H}$ is a $(r \cdot z \times c \cdot z)$ matrix. If the row and column weight for $\mathbf{C}_{i,j}$ is 1, then the circulant is called a circulant permutation matrix (CPM) and can be achieved by rotating the rows of the *Identity* matrix by a certain amount. In that case, since the row and column weights for each CPM is 1, we have $d_c = r$ and $d_v = c$.

Example 3.4 *Equation 3.11 shows the parity check matrix of a QC-LDPC code with* $r = 2$, $c = 3$, *and* $z = 5$.

$$\mathbf{H}_{10\times 15} = \left[\begin{array}{ccccc|ccccc|ccccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right] \tag{3.11}$$

In addition to high error correction capability, QC-LDPC codes are suitable for hardware implementation since the presence of the sub-matrices in $\mathbf{H}$ enables folding the variable/check node processing in order to design a partial parallel decoder. For instance, each circulant can be assigned to a separate computational unit to process its rows and columns. In that case, although all circulants are being processed in parallel, the rows and columns of each circulant are processed sequentially.

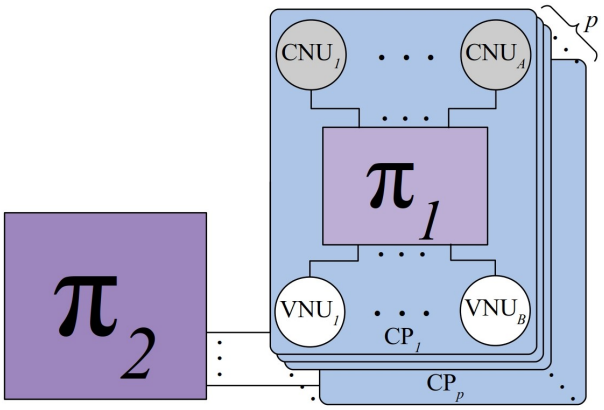


Figure 3.5: Generic architecture for QC-LDPC decoders

Due to the cyclic nature of the sub-matrices, if the row i of a circulant has a 1 in the column j, the row $i+1$ has a 1 in the column $j+1$. This means if V_j is connected to C_i, then C_{i+1} connected to V_{j+1}. This property facilitates transferring messages between check and variable nodes. This research focus on QC-LDPC codes due to their high noise threshold and potential for efficient hardware implementation.

3.5 QC-LDPC Decoder Architectural Features

Figure 3.5 illustrates a generic architecture for a QC-LDPC decoder. The i^{th} row of circulants (ROC) of $\mathbf{H}$ in Equation 3.10 is defined as $\{\mathbf{C}_{i,j} \mid 0 < j \leq c\}$.

Each circulant processor (CP) consists of A check node units (CNUs) and B variable node units (VNUs) and processes a row of circulants in $\mathbf{H}$, where A and B are the parallelization factors for the rows and columns within a row of circulants respectively.

The $\boldsymbol{\pi_1}$ and $\boldsymbol{\pi_2}$ are the shuffle networks in charge of data movement to communicate messages between the nodes. The $\boldsymbol{\pi_1}$ is for intra-circulant message passing, while $\boldsymbol{\pi_2}$ is for inter-circulant message passing.

A majority of QC-LDPC decoders can be mapped to the architecture in Figure 3.5. However, they differ on their parallelism level, implementation of the $\boldsymbol{\pi}$ networks, and their adaptability to new codes. In this section, we define a set of architectural features that facilitate evaluating QC-LDPC decoders.

Parallelism

Three parallelism levels can be defined for a QC-LDPC decoder:

- **Inter-Codeword (Inter-CW)**: Processing multiple codewords in parallel.

- **Inter-ROC**: Processing multiple rows of circulants simultaneously.

- **Intra-ROC**: Processing multiple rows within a row of circulant in parallel.

While *fully parallel* and *fully sequential* decoders are the two extremes of parallelism, a *foldable parallel* decoder is a more versatile architecture that enables adjusting the parallelism level according to performance requirements. In fact, by allowing to strike a trade-off between computational resources and throughput, foldable architectures facilitate the design space exploration.

The inter-CW parallelism may result in under utilizing computational resources since multiple codewords may require different numbers of decoding iterations. This research is focused on foldable parallel architectures at either inter-ROC or intra-ROC levels. Nevertheless, the inter-CW parallelism can always be realized through multiple instances of the decoder.

Shuffle Network

The most important feature of the QC-LDPC codes is the cyclic nature of their submatrices which facilitates the efficient implementation of their shuffle networks. The fact that subsequent check nodes communicate with subsequent variable nodes leaves the following options to implement the $\boldsymbol{\pi}$ networks:

- **Hardwired**: The communicating units are hardwired according to the connection pattern in $\mathbf{H}$. This approach is not scalable for large codewords and circulants.

- **Shift-Register**: The messages are loaded in a shift register so that a shift operation is required to process subsequent rows of a circulant. Although this approach is area-efficient, it may stall the processors for multiple cycles in order to align the data based on the circulants offset.

- **Barrel-Shifter**: The barrel-shifter has the same logic as the shift-register except that it can perform the required shifts in a single cycle. However, it imposes significant area and power consumption overhead.

- **In-Place Shifter**: In this method, the messages are stored in a memory and the memory addressing controls the message passing sequence. However, implementing a parallel in-place shifter that can process multiple rows of circulant per cycle is challenging.

Adaptability

Any coding system, may need to change the code depending on the operating condition. In the data storage context, a storage device's RBER may vary depending on its operating age, temperature, etc. An adaptive decoder is able to adapt to new codes *on-the-fly*. Adaptability can be implemented in two levels:

- **Code-Adaptive**: A code-adaptive decoder can adapt to new codes with same rates.
- **Rate-Adaptive**: A rate-adaptive decoder can adapt to any code rate.

In general, lower rate codes have higher-noise threshold. Consequently, rate-adaptive decoders are useful specially for ECC in data storage applications where noise (RBER) increases during disk's lifetime and lower-rate codes can be beneficial to increase their lifespan.

3.6 Prior Works

In 2009, the introduction of the first use of LDPC in a hard disk drive (HDD) by LSI corporation shed light on the possibility of utilizing LDPC in solid-state drive (SSD) [20]. Since 2009, the research community has been investigating efficient architectures for LDPC codes to replace conventional BCH codes as ECC for data storage systems.

Authors in [21] present a QC-LDPC decoder by leveraging the configurable data-width of block RAM (BRAM) in FPGA. Multiple messages are stored in a single memory word to enable intra-ROC parallelism. The parallelism is foldable since the number of messages stored in a memory word can be adjusted. Moreover, since the code is stored in a memory, it is possible to change the code at runtime. However, since the decoder is fully parallel at inter-ROC level, it does not support rate-adaptability. In addition, using FPGA's BRAMs as the memory for shuffle network may impose some routing constraints for more parallel decoders.

Zaidi *et al.* proposed a layered decoder with foldable intra-ROC parallelism [22]. However, using shift registers and barrel shifters for the $\boldsymbol{\pi}$ networks degrades the architecture's area efficiency. Moreover, the decoder's fully parallel structure at inter-ROC level does not allow for rate-adaptability.

Amaricai and Boncalo [23] proposed a hard-decision decoder that performs the gradient decent bit flipping (GDBF) algorithm, *i.e.*, a simplified version of NGDBF that operates with single bit LLRs. Although the decoder's folded structure enables rate-adaptability, it lacks area efficiency since it uses barrel shifters for its $\boldsymbol{\pi_2}$ network.

An inter-codeword parallel QC-LDPC decoder is presented in [24]. Similar to the technique presented in [21], each BRAM contains multiple messages. However, the drawback with inter-CW parallelism is that the decoder is underutilized if multiple codewords require different numbers of decoding iterations. In addition, since all rows of circulants are processed in parallel, the design is less flexible in adjusting the parallelism according to the performance requirements.

Yanhuan *et al.* [25], present a FPGA-based rate-adaptive QC-LDPC decoder for SSDs. The decoder uses an in-place shuffle network for its $\boldsymbol{\pi_1}$. However, the use of shift-registers for the $\boldsymbol{\pi_2}$ network makes the decoder less efficient due to the inevitable stalls.

Authors in [26], presented an unrolled LDPC decoder. In this pipelined design, each row of circulants, is processed in one macro-pipeline stage resulting in a high throughput. The data movement between pipeline stages *i.e.*, the $\boldsymbol{\pi_2}$ shuffle network, is hardwired which makes the approach impractical for codes with large codewords due to their congested network.

Milicevic and Gulak[27], proposed an application-specific integrated circuit (ASIC) implementation of a rate-adaptive, frame-interleaved LDPC decoder which offered

Table 3.2: Specifications, features, and efficiency of prior QC-LDPC Decoders

	Ref.	[21]	[22]	[23]	[24]	[25]	**This work**		
	Year	2011	2013	2016	2016	2017	2021		
	Codeword Size	8176	8176	2304	1536	35840	9457 - 9766		
	Alg.	NMS*	NMS	GDBF	Min-Sum	NMS	NGDBF	APP	FAID
	Adaptability	Code	Code	Rate	Code	Rate	Rate		
Parallelism	**Inter-ROC**	Full	Full	Sequential	Full	Foldable	Sequential		
Parallelism	**Intra-ROC**	Foldable	Foldable	Full	Sequential	Sequential	Foldable		
Shuffle Net.	$\boldsymbol{\pi_1}$	In-Place	Shift Register	In-Place	In-Place	In-Place	In-Place		
Shuffle Net.	$\boldsymbol{\pi_2}$	—	Barrel Shifter	Barrel Shifter	—	Barrel Shifter	In-Place		
	Device	Virtex-4	Virtex-7	Virtex-7	Virtex-7	Virtex-7	Virtex-7		
	Efficiency ($\frac{MB/s/KLUT}{Iteration}$)	38.7	3.97	107.8	74.02	12.79	79.34 - 130.62	20.04 - 27.77	16.26 - 20.96
	Throughput ($\frac{GB/s}{Iteration}$)	0.64	0.4	0.42	1.52	1.32	2.91 - 3.08	1.8 - 2.42	1.7 - 2.34

*NMS: Normalized Min-Sum algorithm, Improved version of the Min-Sum algorithm.

inter-CW parallelism. By combining the variable and check node operations, and defining a novel processing order, their decoder simplifies the shuffle network which facilitate its hardwired implementation for their target codes. However, the parallelism offered by Milicevic's decoder is determined by the number of columns of circulants in the code thus not foldable based on application requirements.

Table 3.2 summarizes major relevant prior QC-LDPC decoders, implemented in FPGAs, along with their architectural features and efficiency. For a fair comparison, each 4-LUT in Xilinx Virtex-4 is counted as 0.625 6-LUT in Xilinx Virtex-7 FPGAs in computing the efficiency [28]. In general, LDPC decoders for codes with smaller block sizes tend to have higher efficiency due to their less congested Tanner graph. Although the decoders reported in Table 3.2 belong to a wide range of block sizes, we believe a fair evaluation should compare decoders with similar block sizes.

3.7 LDPC Requirements for Flash Memories

LDPC codes are becoming the mainstream ECC in flash memory-based SSDs due to their high error correction capability. The requirements when applying LDPC as an ECC for SSD storage devices is as follows:

- **Throughput**: The SSD bandwidth can be scaled by increasing the number of flash channels and chips [29][30]. However, the ultimate read speed is mainly determined by the interface. While SSDs with serial advanced technology attachment (SATA) interface can achieve up to 530 MB/s [31], modern PCI-e SSDs have read speeds of 1-6 GB/s depending on the application. Therefore, it is desirable to have LDPC decoders with similar throughput to prevent ECC from becoming the performance bottleneck.
- **Latency** The read latency in SSD memories has three major contributors [29]:
  - **Memory Sensing**: Refers to the phase where the cell values are read through sensing. NAND flash memory sensing latency is linearly proportional to the number of sensing quantization levels.
  - **Data Transfer**: Refers to the phase where the sensing data is transferred to the flash controller.
  - **Decoding**: Once the data is transferred to controller, it has to be decoded to correct any possible errors.

  The read latency is a critical metric for flash memories. For modern flash devices, the aggregated memory sensing and data transfer latency for a 16 KB page is about $100 - 120\mu s$. It is desirable to keep the decoder latency below $10\mu s$.

- **Reliability**: The SSD reliability is measured by its uncorrectable bit error rate (UBER). As outlined in the JESD218 standard [7], consumer-level SSDs should reach the UBER below 10^{-15} and enterprise-level SSDs should reach the UBER below 10^{-16}. A typical UBER for enterprise-level SSD is 10^{-17}.
- **Code Rate**: Manufacturers of flash memory devices often devote about $10-12\%$ of the storage capacity for error correction which means the codes with rate 0.88-0.90 should be used for these devices.
- **Block Size**: The code rate requirement in the SSD devices $(0.88 - 0.90)$, necessitates utilizing LDPC codes with 1-4 KB block size to meet the reliability requirements (UBER $\leq 10^{-15}$).

3.8 Our Work

According to table 3.2, prior QC-LDPC decoder architectures suffer from either lack of adaptability or foldable parallelism. Moreover, they are generally centered around improving the hardware efficiency for a certain code of interest and often with application areas well beyond storage. This thesis extends the theme by addressing the hardware and code performance while respecting limitations in storage application. Throughout this research, we pursue interesting trade offs between parallelism, hardware efficiency, and code performance. To this end, we make the following contributions:

- An FPGA-efficient micro-architecture for QC-LDPC decoders:
  - We leverage an FPGA's inherent physical architecture to implement rotary register file (RRF), an efficient shuffle network for QC-LDPC decoders, that enables foldable parallelism.
  - Using RRF as the basic block, we propose a pipelined, rate-adaptive decoder micro-architecture.
  - We use the proposed micro-architecture to implement the noisy gradient descent bit-flipping (NGDBF), a simple soft-decision decoding algorithm that strikes a trade off between hardware efficiency and code performance.
  - Our micro-architecture offers a simple form of foldable parallelism which enables a spectrum of design instances from the most parallel to the most serial decoders, a feature that facilitates the design space exploration.
- A micro-architecture-code co-design for QC-LDPC decoding in flash memories:

- We address the inherent limitations in ECC for flash memories and define a finite decoder design space.
- While respecting the hardware efficiency, we use machine learning techniques to design a Finite alphabet iterative decoding (FAID) that outperforms conventional decoders, given the flash memory constraints.
- We use our FPGA-efficient micro-architecture to implement the proposed FAID.

We will discuss the results and achievements of each contributions in the upcoming chapters. Meanwhile, the last columns in Table 3.2 reports a brief summary of our achievements in designing LDPC decoders. We report a range for the performance metrics as we evaluated our decoder on a set of codes during our hardware-code co-design.

Chapter 4

Configurable Micro-Architecture For QC-LDPC Decoder

This chapter presents our configurable micro-architecture for the QC-LDPC decoder. We first review the operations and computations involved in QC-LDPC decoding and decompose the computations to enable efficient hardware implementation. Then, we define the *strided circular access* problem to propose the *rotary register file* (RRF), an FPGA-efficient foldable parallel structure for strided circular access, seen in QC-LDPC decoding. Next, the RRF is used as the basic-block to implement our QC-LDPC decoder architecture. Finally, we use the proposed decoder architecture to implement the NGDBF soft-decision algorithm. Nevertheless, our micro-architecture can easily be used to implement other decoding algorithms as well. In fact, Chapter 5 discusses implementing our soft-decision decoder using the proposed micro-architecture.

4.1 Computations in QC-LDPC Decoders

A CPM is a square matrix that is achieved by rotating the rows of the *Identity* matrix by a certain amount. Any CPM can be defined by two parameters:

- f: The rotation offset for each row.
- z: The CPM's number of rows.

Therefore, a CPM can be denoted as $\mathbf{C}^{f}_{(z)}$. The parity check matrix, $\mathbf{H}$, for a QC-LDPC code consists of a set of same size CPMs as its sub-matrices. We explain the decoding computation flow using an example.

Example 4.1 *Consider* $\mathbf{H}$ *in Equation 4.1 as the parity check matrix of a QC-LDPC code formed by a* 2×3 *formation of CPM sub-matrices.*

$$\mathbf{H}_{10\times 15} = \begin{array}{c} \\ C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \\ C_7 \\ C_8 \\ C_9 \end{array} \begin{array}{c} \begin{array}{ccccc} V_0 & V_1 & V_2 & V_3 & V_4 \end{array} \\ \left[\begin{array}{ccccc} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array}\right. \end{array} \begin{array}{c} \begin{array}{ccccc} V_5 & V_6 & V_7 & V_8 & V_9 \end{array} \\ \left[\begin{array}{ccccc} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right. \end{array} \begin{array}{c} \begin{array}{ccccc} V_{10} & V_{11} & V_{12} & V_{13} & V_{14} \end{array} \\ \left[\begin{array}{ccccc} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right. \end{array} \tag{4.1}$$

$$= \begin{bmatrix} \mathbf{C}^1_{(5)} & \mathbf{C}^2_{(5)} & \mathbf{C}^3_{(5)} \\ \mathbf{C}^3_{(5)} & \mathbf{C}^4_{(5)} & \mathbf{C}^2_{(5)} \end{bmatrix}$$

4.1.1 Check Node Processing

Each check node computes and sends a set of messages to its connected variable nodes. Let $\mathbf{\Psi}(\mathcal{V}_i, j, t-1)$ be the processing function in a check node i, which computes response to the variable node j, at iteration t, based on the incoming messages from the set of variable nodes with indices in the set $\mathcal{V}_i$.

Example 4.2 *According to Equation 4.1:*

- *Process in* C_0*:* $r^{(t)}_{0,j} = \mathbf{\Psi}(\{1, 7, 13\}, j, t-1), \quad j \in \{1, 7, 13\}$
- *Process in* C_1*:* $r^{(t)}_{1,j} = \mathbf{\Psi}(\{2, 8, 14\}, j, t-1), \quad j \in \{2, 8, 14\}$
- *Process in* C_2*:* $r^{(t)}_{2,j} = \mathbf{\Psi}(\{3, 9, 10\}, j, t-1), \quad j \in \{3, 9, 10\}$
- $\cdots$

where $r^{(t)}_{i,j}$ *is the messages from the check node* i *to the variable node* j *at iteration* t*.*

Due to the cyclic nature of the sub-matrices in $\mathbf{H}$, the access pattern to the variable node messages corresponding to each sub-matrix has a rotary form. Therefore, if the messages corresponding to each sub-matrix are rotated based the offset of the CPMs, the output data would be in order for check node processing.

Example 4.3 *For the* $\mathbf{H}$ *in Equation 4.1, it is possible to compute the check node messages for the first row of circulants,* $(C_0, C_1, C_2, C_3, C_4)$*, by applying the appropri-*

ate left rotation and performing the $\mathbf{\Psi}$ *function to the columns as follows:*

$$\begin{array}{ccccc} & & & & & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ l_{4,0} & l_{0,1} & l_{1,2} & l_{2,3} & l_{3,4} & \xrightarrow{\textit{Rotate Left (1)}} & l_{0,1} & l_{1,2} & l_{2,3} & l_{3,4} & l_{4,0} \\ l_{3,5} & l_{4,6} & l_{0,7} & l_{1,8} & l_{2,9} & \xrightarrow{\textit{Rotate Left (2)}} & l_{0,7} & l_{1,8} & l_{2,9} & l_{3,5} & l_{4,6} \\ l_{2,10} & l_{3,11} & l_{4,12} & l_{0,13} & l_{1,14} & \xrightarrow{\textit{Rotate Left (3)}} & l_{0,13} & l_{1,14} & l_{2,10} & l_{3,11} & l_{4,12} \\ \hline & & & & & & r_{0,j} & r_{1,j} & r_{2,j} & r_{3,j} & r_{4,j} \end{array} \quad \downarrow \mathbf{\Psi}() \tag{4.2}$$

where $l_{i,j}$ *is the variable node* j *message to the check node* i*, and* $r_{i,j}$ *is the check node* i *message to the variable node* j*. The second set of check node messages.* $(\{r_{5,j}, r_{6,j}, r_{7,j}, r_{8,j}, r_{9,j}\})$ *can be computed by rotating the variable node messages based on the offsets of the second row of circulants in* $\mathbf{H}$ $(\mathbf{C}^3_{(5)}, \mathbf{C}^4_{(5)}, \mathbf{C}^2_{(5)})$*.*

4.1.2 Variable Node Processing

Let $\mathbf{\Phi}(y_j, \mathcal{C}_j, i, t)$ be the processing function in the variable node j, to compute the response for the check node i, at iteration t, based on the incoming messages from the set of check nodes with indices in the set $\mathcal{C}_j$ and the channel information y_j.

Example 4.4 *For the* $\mathbf{H}$ *in Equation 4.1, we have:*

- *Process in* V_0*:* $l^{(t)}_{i,0} = \mathbf{\Phi}(y_0, \{4, 7\}, i, t), \quad i \in \{4, 7\}$
- *Process in* V_1*:* $l^{(t)}_{i,1} = \mathbf{\Phi}(y_1, \{0, 8\}, i, t), \quad i \in \{0, 8\}$
- *Process in* V_2*:* $l^{(t)}_{i,2} = \mathbf{\Phi}(y_2, \{1, 9\}, i, t), \quad i \in \{1, 9\}$
- $\cdots$

where $l^{(t)}_{i,j}$ *is the messages from the variable node* j *to the check node* i *at iteration* t*, and* y_j *is the channel information for the* j^{th} *bit of the received codeword.*

The cyclic nature of the sub-matrices can be utilized in the variable node processing as well. The only difference is that the check node messages need to be rotated in the right direction. However, the i^{th} right rotation of a sequence of s elements, is equivalent to its $(z-i)^{th}$ left rotation. We refer to $z - i$ as the *complement of offset* i. Since the check node processing involves left rotations, we define variable node processing using left rotations as well.

Example 4.5 *For the* $\mathbf{H}$ *in Equation 4.1, the variable node processing can be done by applying the appropriate left rotation and performing the* $\mathbf{\Phi}$ *function to the columns*

as follows:

$$\begin{array}{ccccccc} & & & & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\ r_{0,1} & r_{1,2} & r_{2,3} & r_{3,4} & r_{4,0} & \xrightarrow{\textit{Rotate Left (5-1=4)}} & r_{4,0} & r_{0,1} & r_{1,2} & r_{2,3} & r_{3,4} \\ r_{5,3} & r_{6,4} & r_{7,0} & r_{8,1} & r_{9,2} & \xrightarrow{\textit{Rotate Left (5-3=2)}} & r_{7,0} & r_{8,1} & r_{9,2} & r_{5,3} & r_{6,4} \\ \hline & & & & & & l_{i,0} & l_{i,1} & l_{i,2} & l_{i,3} & l_{i,4} \end{array} \quad \downarrow \Phi() \tag{4.3}$$

where $r_{i,j}$ is the check node i message to the variable node j, and $l_{i,j}$ is the variable node j message to the check node i. The second set of check node messages. Similarly, other variable node messages ($\{l_{i,5}, l_{i,6}, l_{i,7}, l_{i,8}, l_{i,9}\}$, and $\{l_{i,10}, l_{i,11}, l_{i,12}, l_{i,13}, l_{i,14}\}$) are computed based on the offsets of the second and third columns of circulants.

4.2 Rotation in Hardware

The computations in QC-LDPC decoding show that the messages can be transferred between the nodes through rotation operations. Therefore, efficient implementation of the rotation logic as the shuffle network is essential to designing an efficient QC-LDPC decoder architecture. Moreover, the type of parallelism offered by the rotation logic determines the parallelism in the final decoder architecture.

4.2.1 Naïve Implementation

A rotation logic can naïvely be implemented with extremely parallel or serial structures as depicted in Figure 4.1.

Barrel Shifter A logic circuit that utilizes several layers of multiplexers to shift/rotate a set of inputs in parallel. Although a barrel shifter can be pipelined to increase throughput, it imposes a significant amount of hardware resources. This issue is specifically worsened for FPGAs in which multiplexers are expensive. Moreover, since the resource usage scale factor is $n \log n$, where n is the number of inputs, this approach is not scalable.

Shift Register A logic circuit that uses a set of cascaded registers to shift/rotate the data sequentially. Despite using relatively less hardware resources, a shift register imposes high latency as it takes multiple cycles to store and then rotate the data. In addition, the latency is not constant and it depends on the offset of the circular access.

The main drawback to both approaches is their lack of flexibility. A barrel shifter is a fully parallel, high-throughput, but high-cost implementation while a shift-register is a fully sequential, low-cost, but low-throughput implementation.

4.2.2 Foldable Parallel Rotation in Memory

The extreme characteristics of the naïve implementations allow no trade-off between the resource usage and throughput. Moreover, their implementation costs abundant routing resources which may lead to inefficient implementation on FPGA.

To overcome the inefficiencies and rigidness of the naïve implementations, memory modules can be used to turn the rotation in the spacial domain into the time domain. In this approach, the set of messages to be rotated are stored in a memory and the rotation is done through appropriate memory addressing order. Moreover, the foldable parallelism could be provided by generating multiple messages of the rotation simultaneously. This can be done through accessing multiple memory banks in parallel or packing multiple messages in a single memory word.

4.2.3 Leveraging FPGA's Hard Logic: Distributed RAM

The natural memory resources on FPGAs are the block RAMs (BRAMs). However, these memories are very coarse-grained modules and using them to implement the rotation could reduce and confine the parallelism which may not lead to an efficient design.

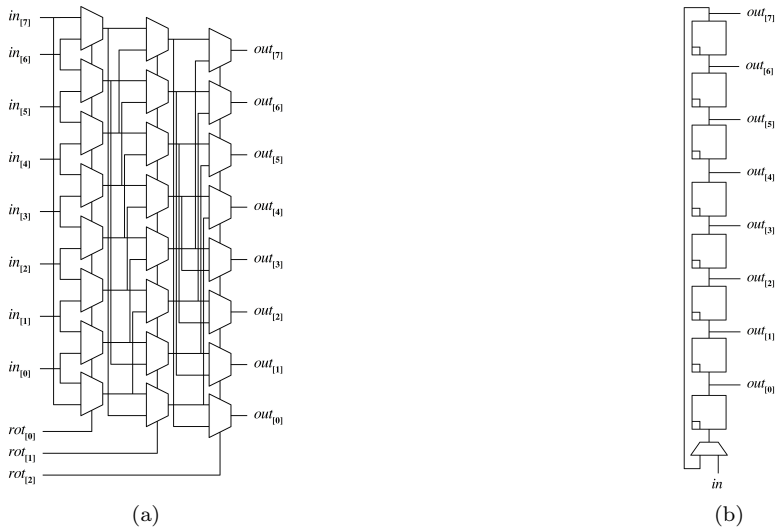


Figure 4.1: Naïve implementations for circular memory access of size 8, (a) Barrel shifter (b) Shift register

Aside from being used as a *soft logic* to implement the truth table of a given logic function, FPGA's lookup tables (LUTs) can also be configured as a memory bank, referred to as the *distributed RAM*. In that case, the LUT's physically implemented multiplexers are used for memory addressing. These multiplexers are referred to as *hard multiplexers* (*hard logic*) and utilizing them is significantly more efficient than implementing the multiplexing function as a soft logic *i.e.*, as a truth table in LUTs.

In this work, we propose leveraging FPGA's distributed RAMs to implement the rotation logic required in QC-LDPC decoding. In addition to leveraging FPGA's hard logic by using the LUTs as distributed RAMs, FPGA's abundance of its LUTs allows more freedom in adjusting the parallelism.

4.3 Rotary Register File: A Hybrid Shuffle Network

We propose the *rotary register file* (RRF), an FPGA-optimized micro-architectural primitive that not only provides rotation in the time domain, but also offers foldable parallelism which enables a spectrum of design instances between the most serial and parallel designs. We refer to the memory accesses during a rotation operation as the *strided circular access* and propose a memory structure that does not require redundant memory modules to provide parallel memory access. Instead, we orchestrate the accesses in such a way that guarantees conflict-free access.

4.3.1 Problem Statement

In this section, we formally establish the concept of the circular access pattern, and the condition of conflict-free access that is critical for a foldable parallel implementation.

Definition 4.1 ***(Circular Access):*** *Consider a set $\mathcal{A} = [0, z) = \{0, 1, \cdots, z-1\}$ of z consecutive addresses, and a set $\mathcal{I} = \{0, 1, \cdots, z-1\}$ of z consecutive indices. A circular access of offset $f \in [0, z)$ is a function $\mathcal{CA}_f : \mathcal{I} \to \mathcal{A}$ where*

$$\mathcal{CA}_f(i) = (f + i) \mod z \tag{4.4}$$

Definition 4.2 ***(Strided Partition):*** *A strided partition, $\{\mathcal{SP}_j \subset \mathcal{I}\}$, of stride s, is a partition of index set $\mathcal{I}$, where*

$$0 \le j < s, \quad 0 \le j < s, \quad \mathcal{SP}_j = \{k \times z + j \mid k \in [0, b = \lfloor \frac{z}{s} \rfloor)\}. \tag{4.5}$$

Apparently, we have:

$$\forall 0 \le i < s, \quad 0 \le j < s, \quad i \ne j, \quad \mathcal{SP}_i \cap \mathcal{SP}_j = \varnothing. \tag{4.6}$$

Moreover, assuming $z = b \times s + q$, we have:

$$\bigcup_{0 \le j < s} \mathcal{SP}_j = \begin{cases} \mathcal{I} & q = 0, \\ \mathcal{I} - \{z - q, \cdots, z - 1\} & 0 < q < s. \end{cases} \tag{4.7}$$

Definition 4.3 ***(Surplus Set)****: For $0 < q < s$ in a strided partition, the* surplus set*, $\{\mathcal{SU} \subset \mathcal{I}\}$ is defined as follows:*

$$\mathcal{SU} = \{z - i \mid i \in (0, q]\}$$

Definitions 4.2 and 4.3 imply that for $0 < q < s$:

$$(\bigcup_{0 \le j < s} \mathcal{SP}_j) \cup \mathcal{SU} = \mathcal{I} \tag{4.8}$$

Example 4.6 ***Case 1:*** *For $z = 12$ and stride $s = 4$, we have*

$$\mathcal{I} = \{0, 1, 2, \cdots, 11\}, \quad 12 = 3 \times 4 + 0 \Rightarrow b = 3, q = 0$$

and

$$\mathcal{SP}_0 = \{0, 4, 8\} \quad \mathcal{SP}_1 = \{1, 5, 9\}$$
$$\mathcal{SP}_2 = \{2, 6, 10\} \quad \mathcal{SP}_3 = \{3, 7, 11\}$$

Therefore,

$$\bigcup_{0 \le j < 4} \mathcal{SP}_j = \{0, 1, 2, \cdots, 11\} = \mathcal{I}.$$

Case 2: *For $z = 14$ and stride $s = 4$, we have*

$$\mathcal{I} = \{0, 1, 2, \cdots, 13\}, \quad 14 = 3 \times 4 + 2 \Rightarrow b = 3, q = 2$$

and

$$\mathcal{SP}_0 = \{0, 4, 8\} \quad \mathcal{SP}_1 = \{1, 5, 9\}$$
$$\mathcal{SP}_2 = \{2, 6, 10\} \quad \mathcal{SP}_3 = \{3, 7, 11\}$$
$$\mathcal{SU} = \{12, 13\}$$

Therefore,

$$(\bigcup_{0 \le j < 4} \mathcal{SP}_j) \cup \mathcal{SU} = \{0, 1, 2, \cdots, 11\} \cup \{12, 13\} = \mathcal{I}.$$

Definition 4.4 ***(Strided Circular Access):*** *A strided circular access of size z,*

stride s, and offset f, is a partition of address set $\mathcal{A}$, $\{\mathcal{SCA}_j^f \subset \mathcal{A}\}$, where

$$\begin{aligned}\mathcal{SCA}_j^f &= \{\mathcal{CA}_f(i) \mid i \in \mathcal{SP}_j\} \\ &= \{(f + j + k \times s) \mod z \mid k \in [0, b)\}\end{aligned}$$

For the surplus access, we define the following special access:

$$\mathcal{SCA}_{\mathcal{SU}}^f = \{\mathcal{CA}_f(i) \mid i \in \mathcal{SU}\}$$

Example 4.7 *In the Example 4.6, Case 2, the strided circular accesses with offset 6 are as follows:*

$$\begin{aligned}\mathcal{SCA}_0^6 &= \{\mathcal{CA}_6(0), \mathcal{CA}_6(4), \mathcal{CA}_6(8)\} = \{6, 10, 0\} \\ \mathcal{SCA}_1^6 &= \{\mathcal{CA}_6(1), \mathcal{CA}_6(5), \mathcal{CA}_6(9)\} = \{7, 11, 1\} \\ \mathcal{SCA}_2^6 &= \{\mathcal{CA}_6(2), \mathcal{CA}_6(6), \mathcal{CA}_6(10)\} = \{8, 12, 2\} \\ \mathcal{SCA}_3^6 &= \{\mathcal{CA}_6(3), \mathcal{CA}_6(7), \mathcal{CA}_6(11)\} = \{9, 13, 3\} \\ \mathcal{SCA}_{\mathcal{SU}}^6 &= \{\mathcal{CA}_6(12), \mathcal{CA}_6(13)\} = \{4, 5\}\end{aligned}$$

Definition 4.5 ***(Column-Major Layout):*** *Given the sets $\mathcal{B} = [0, \mathbf{b}] = \{0, 1, ..., b\}$ of $b+1$ memory banks of size s and a $\mathcal{S} = [0, s) = \{0, 1, ..., s\}$, of s intra-bank memory addresses, a column-major layout, $(\mathcal{L}, \mathcal{T})$, is defined by two functions $\mathcal{L} : \mathcal{A} \to \mathcal{B}$ and $\mathcal{T} : \mathcal{A} \to \mathcal{S}$ where:*

$$\begin{aligned}\mathcal{L}(a) &= \lfloor \frac{a}{s} \rfloor \\ \mathcal{T}(a) &= a \mod s\end{aligned}$$

In fact, the function $\mathcal{L}$ computes the bank index for each address in the set $\mathcal{A}$, while the function $\mathcal{T}$ computes the address within that bank.

Definition 4.6 ***(Conflict-Free Access Group):*** *An access group $\mathcal{G} \subset \mathcal{A}$ of the address set $\mathcal{A}$ under the memory layout $(\mathcal{L}, \mathcal{T})$ is defined as conflict-free iff*

$$\forall i, j \in \mathcal{G}, \quad \mathcal{L}(i) \neq \mathcal{L}(j) \tag{4.9}$$

According to Definition 4.6, any two conflict-free addresses of set $\mathcal{A}$ can be accessed simultaneously. Therefore, the goal is to orchestrate the circular access pattern so that simultaneous accesses are conflict-free.

Proposition 1 *An access group $\mathcal{G}$ in a strided circular access $\{\mathcal{SCA}_j^f\}$ with offset f,*

stride s, and the column-major layout $(\mathcal{L}, \mathcal{T})$ is conflict-free if:

$$\forall j, \quad 0 \leq |\mathcal{SCA}_j^f| \leq b$$

Proposition 1 implies that in a *strided circular access* with any offset f in a *column-major layout*, it is always possible to access at most b elements in parallel.

Example 4.8 *In Example 4.7, the column-major layout memory bank for the stride circular access are as follows:*

$$\begin{aligned}
\mathcal{L}(\mathcal{SCA}_0^6) &= \{\mathcal{L}(6), \mathcal{L}(10), \mathcal{L}(0)\} = \{1, 2, 0\} \\
\mathcal{L}(\mathcal{SCA}_1^6) &= \{\mathcal{L}(7), \mathcal{L}(11), \mathcal{L}(1)\} = \{1, 2, 0\} \\
\mathcal{L}(\mathcal{SCA}_2^6) &= \{\mathcal{L}(8), \mathcal{L}(12), \mathcal{L}(2)\} = \{2, 3, 0\} \\
\mathcal{L}(\mathcal{SCA}_3^6) &= \{\mathcal{L}(9), \mathcal{L}(13), \mathcal{L}(3)\} = \{2, 3, 0\}
\end{aligned}$$

Since all $\mathcal{L}(\mathcal{SCA}_i^f)$s satisfy Equation 4.9, they are conflict-free access groups. For the surplus access, we have:

$$\mathcal{L}(\mathcal{SCA}_{\mathcal{SU}}^6) = \{\mathcal{L}(4), \mathcal{L}(5)\} = \{1, 1\}$$

which is not a conflict-free access group and thus not accessible in parallel.

4.3.2 Micro-Architecture

We use the formulation in Section 4.3.1 to design a micro-architectural structure that allows, and desirably exploits, circular accesses. Presumably, such structure needs to maximize the amount of conflict-free accesses established above.

For any strided circular access with size z, and stride s, Proposition 1 can be used to orchestrate a conflict-free strided circular access. According to Definition 4.2, there are two cases for any strided partition: $q = 0$ and $0 < q < s$. Although designing the RRF for $q = 0$ is relatively straightforward, it is not necessarily the case in many cases. Many QC-LDPC codes need z to be a prime number which makes it indivisible to any s. Consequently, it is crucial for RRF to support $0 < q < s$. Assuming:

$$z = b \times s + q \tag{4.10}$$

the RRF stores the data in $b + 1$ memory banks in a column-major layout. Then, for any offset f, the first $b \times s$ circular accesses can be performed in s cycles without any conflict *i.e.*, b conflict-free accesses per cycle. However, the last q circular accesses, *i.e.*, access with respect to the *surplus set*, have no guarantee of being conflict-free.

Table 4.1: Strided circular access with $z = 11$, $s = 4$, $f = 6$, with column-major layout $(\mathcal{L}, \mathcal{T})$.

$\mathcal{A}$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\mathcal{L}$	0				1				2				3	
$\mathcal{T}$	0	1	2	3	0	1	2	3	0	1	2	3	0	1
$\mathcal{CA}_6$	8	9	10	11	12	13	0	1	2	3	4	5	6	7
$\mathcal{SCA}_0^6$	↑						↑				↑			
$\mathcal{SCA}_1^6$		↑						↑				↑		
$\mathcal{SCA}_2^6$			↑						↑				↑	
$\mathcal{SCA}_3^6$				↑						↑				↑
$\mathcal{SCA}_{\mathcal{SU}}^6$					↑	↑								

Consequently, they have to be performed sequentially. As a result, the entire circular access is performed in $s + q$ cycles. Apparently, for $q = 0$, the entire circular access requires only s cycles.

Example 4.9 *Table 4.1 illustrates the address set, $\mathcal{A}$, the column-major layout, $(\mathcal{L}, \mathcal{T})$, the strided circular accesses, $\mathcal{SCA}$, and the surplus circular access, $\mathcal{SCA}_{\mathcal{SU}}$ for Example 4.8. The marked cells in each row represent the members of the corresponding strided circular access. Based on Definition 4.6, and Proposition 1 each $\mathcal{SCA}$ is a conflict-free access group thus accessible simultaneously. However, $\mathcal{SCA}_{\mathcal{SU}}$ is not a conflict-free access group and it has to be accessed sequentially.*

Figure 4.2 illustrates the structure of the RRF for a strided circular access with size z, and stride size s, where $z = b \times s + q$. In order to reduce the complexity of the design, RRF utilizes a two-level rotation approach. The first level consists of $b + 1$ memory banks of size s storing the data in a column-major layout, each generating an output through a multiplexer. The numbers within each cell represents the *Address set* $\mathcal{A}$, while the number on top of the cell shows its intra-bank memory address, $\mathcal{T}$. The *select* signal for each multiplexer is indeed the intra-bank memory address. It

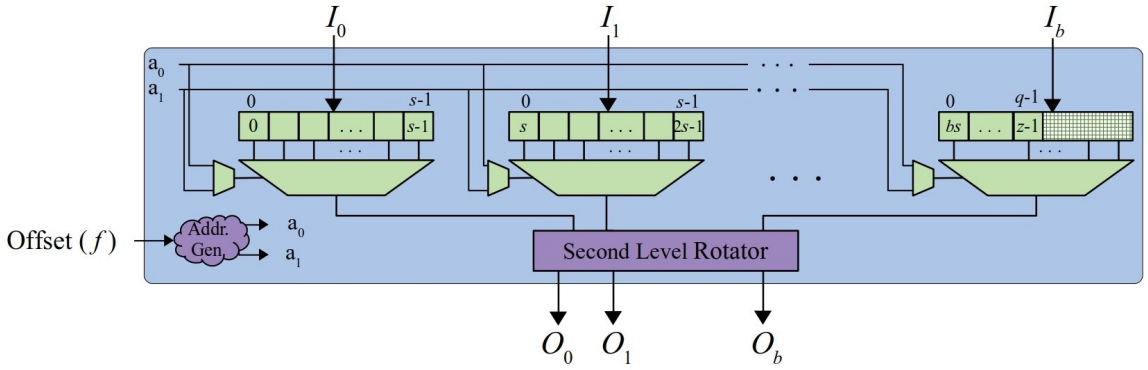


Figure 4.2: Rotary register file structure

can be proved that each *Strided Circular Access*, $\mathcal{SCA}_j^f$ can occur in at most two intra-bank memory addresses, denoted by a_0 and a_1, in all memory banks. These addresses are computed by address generator using the offset, f, the index j, and the stride size s.

Once the appropriate intra-bank addresses with respect to a strided circular access are read, a second-level rotation is required to sort the output order based on the offset, f. For that, RRF uses a simple barrel rotator to perform a single-cycle rotation. Unlike the extreme barrel rotator structure, the complexity of RRF's barrel rotator depends on the number of memory banks, b, rather than the circular access size, z.

Given a constant circular access size, z, the stride size, s, determines the number of memory banks, b which, in turn, determines the number of simultaneous memory accesses. Thus it can be used as a knob to adjust the parallelism. In fact, $s = 1$ and $s = z$ are the fully parallel and serial RRFs that correspond to the barrel shifter and shift register respectively.

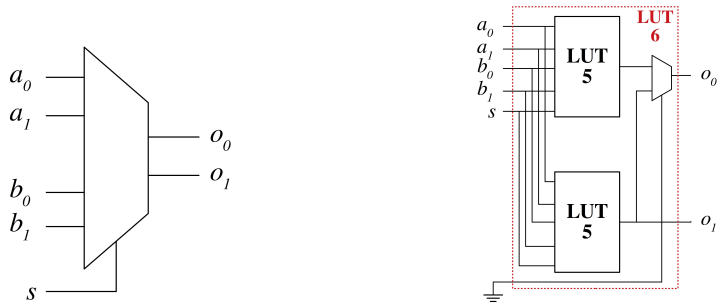
4.3.3 FPGA-Optimized Implementation

As illustrated in Figure 4.2, RRF's first-level rotation logic involves a set of storage components and multiplexers. The multiplexers can be realized using FPGA's soft-logic, *i.e.*, implementing the multiplexing logic through LUTs. However, this approach lacks scalability since large multiplexers require several layers of lookup tables (LUTs) to be cascaded which, in turn, degrades the performance significantly.

A more efficient approach is to leverage the FPGA's physically implemented multiplexers, referred to as hard MUXes. For that, we use the FPGA's LUTs as distributed RAMs to implement RRF's first-level rotation logic. In fact, each RRF's memory bank and its output multiplexer is mapped to a distributed RAM and the RAM's address signal serves as the *select* signal for RRF's multiplexers.

Utilizing the FPGA's hard multiplexers through distributed RAMs results in faster and smaller RRFs as it leverages the FPGA's physical architecture. Moreover, spatial distribution and high availability of the distributed RAMs increase the placement and routing flexibility which, in turn, results in a more efficient design.

Aside from the memory banks, a significant part of the RRF's area is consumed by the $2:1$ multiplexers that select between the a_0 and a_1 addresses for each memory bank. In order to further improve the RRF's performance on FPGA, we manually imposed logic sharing to utilize both outputs of the 6-input LUTs in order to implement 2-bit $2:1$ multiplexers. This technique is applicable to both Intel's and Xilinx's modern FPGA devices [32][33]. Figure 4.3 illustrates both logical representation and efficient realization of this circuit.



(a) (b)

Figure 4.3: Logic sharing (a) 2-bit 2:1 multiplexer (b) Single-LUT6 realization

4.4 Decoder Architecture

The proposed FPGA-optimized RRF enables a regularity for communicating messages between the nodes which, in turn, results in a regular structure for the QC-LDPC decoder. Therefore, rather than having yet-another-design, we use our RRF to propose a decoder architecture based on a regular structure that can be configured for many variants. Moreover, we orchestrate the check and variable node processes to enable pipelining for maximized throughput. Figure 4.4 illustrates the architecture for our QC-LDPC decoder for a $\mathbf{H}_{r \cdot s \times c \cdot s}$ parity check matrix where c is the number of columns of circulant, r is the number of rows of circulants, and s is the CPM size.

The architecture involves an initial LLR memory to store the received codeword from the channel, two sets of c RRFs, a set of check node unit (CNU), and a set of variable node unit (VNU). Each RRF in the level 1, $RRF^{(1)}$s, stores the variable node messages l_js corresponding to one column of CPMs in $\mathbf{H}$. The outputs are sent to the CNUs to perform the check node processing. The result is written into c RRFs in the level 2, $RRF^{(2)}$s. Once the check node processing for one row of circulant is finished, the level 2 RRFs rotate the data for variable node processing. The outputs are then sent to the VNUs for variable node processing. Once all rows of circulants are processed, the VNUs update the values in $RRF^{(1)}$s to perform the next decoding iteration.

The decoding iterations are processed sequentially in our architecture. While increasing the maximum number of decoding iterations could lead to a higher noise threshold, a majority of codewords may need a few iterations to be decoded. The *early termination* logic is a simple hardware comprising of a set of XOR gates and a 1-bit accumulator that checks if all parity constraints are satisfied at the end of each iteration. In that case, the decoding process is terminated and the decoded codeword is written in the output memory. In fact, the early termination logic is a low-cost

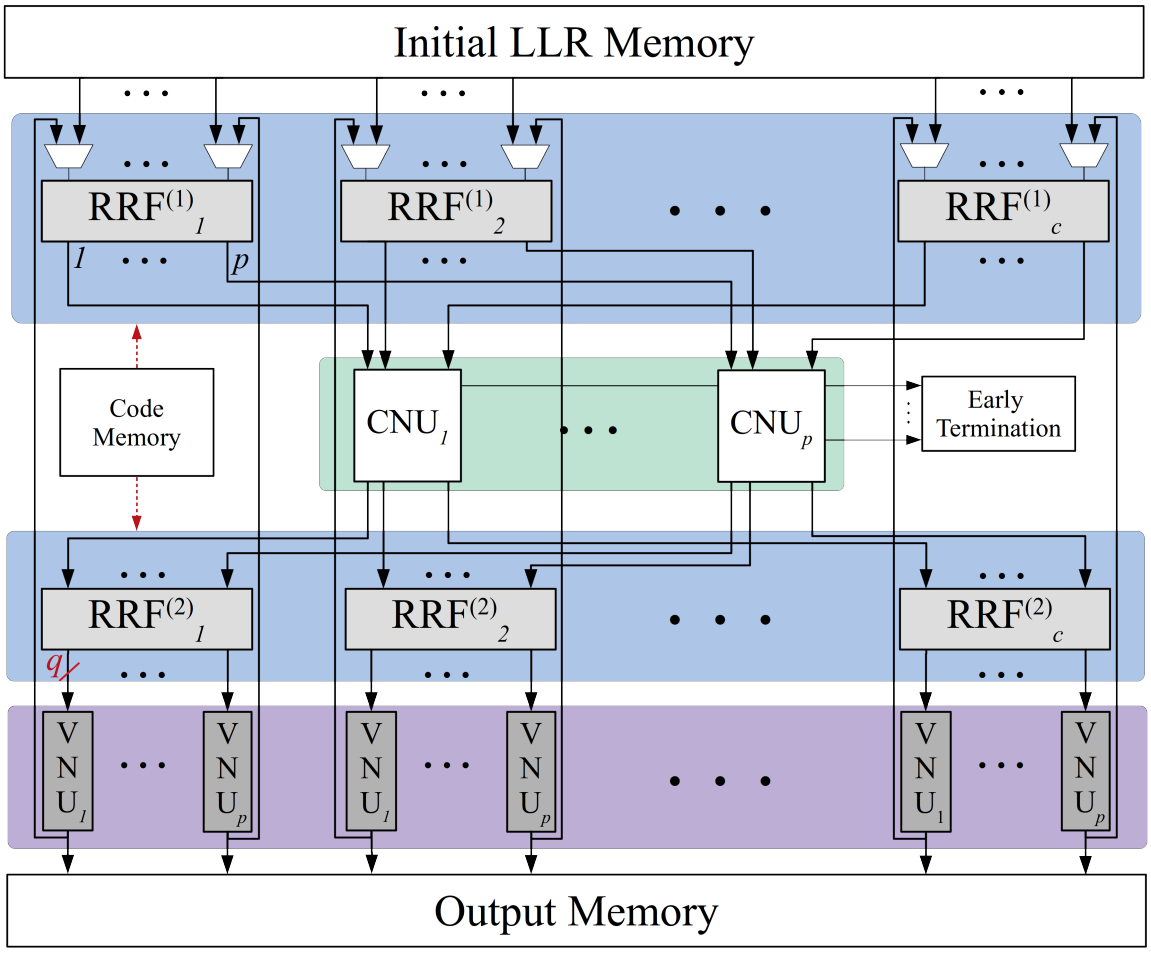


Figure 4.4: QC-LDPC decoder architecture based on RRF shuffle network

component to increase the throughput while increasing maximum number of decoding iterations to achieve a higher noise threshold.

The *code memory* contains the information regarding the code. It essentially contains the required rotation amount for the RRFs. Since the rows of circulants are processed sequentially in our architecture, the number of rows of circulants can change at run time. Therefore, by storing the codes with different rates (number of rows of circulants) in the code memory, it would be possible for our decoder to adapt to codes with different rates at run time.

4.4.1 Pipeline Timing

As depicted in Figure 4.4, our decoder allocates a unit containing a pair of RRFs and a VNU unit to each column of circulants. These units process the CPMs in their corresponding column sequentially. Figure 4.5 depicts the timing for pipeline operations for one decoding iteration for a QC-LDPC code with the parity check

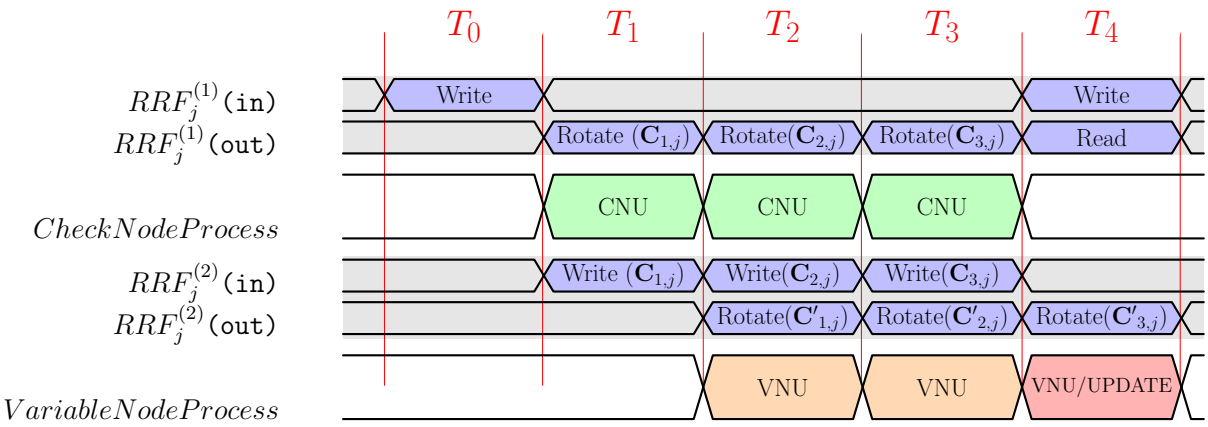


Figure 4.5: Pipelined QC-LDPC decoder timing diagram

matrix:

$$\mathbf{H} = \begin{bmatrix} \mathbf{C}_{1,1} & \cdots & \mathbf{C}_{1,c} \\ \mathbf{C}_{2,1} & \cdots & \mathbf{C}_{2,c} \\ \mathbf{C}_{3,1} & \cdots & \mathbf{C}_{3,c} \end{bmatrix} \tag{4.11}$$

The diagram shows the operation for the j^{th} column in Figure 4.4. Each RRF may perform two operations simultaneously; input and output operations. The operation for each stage is as follows:

- $\mathbf{T_0}$: The input data (LLRs) are written into the $RRF_j^{(1)}$.
- $\mathbf{T_1}$:
  - $RRF_j^{(1)}$ generates a rotation of its data based on the offset of the $\mathbf{C}_{1,j}$ circulant.
  - The CNUs perform the check node process.
  - The CNU outputs are written into $RRF_j^{(2)}$.
- $\mathbf{T_2}$:
  - $RRF_j^{(1)}$ provides the rotation based on the offset of the $\mathbf{C}_{1,j}$ circulant.
  - The outputs of the CNUs are stored in $RRF_j^{(2)}$.
  - $RRF_j^{(2)}$ rotates its stored data from previous cycle (T_1) based on the complement offset of the $\mathbf{C}_{1,j}$ circulant ($\mathbf{C}'_{1,j}$).
  - The VNU performs variable node process on the outputs of $RRF_j^{(2)}$.
- $\mathbf{T_3}$: The operations in T_3 is similar to T_2 except that all units process their next row of circulants.
- $\mathbf{T_4}$:

- $RRF_j^{(2)}$ generates the rotation of its data based on the complement offset of the $\mathbf{C}_{2,j}$ circulant.
- The VNUs perform variable node process and update the $RRF_j^{(1)}$s.
- $RRF_j^{(1)}$ writes the results of the VNU for each bit in its memories.

This sequence is performed until the code is successfully decoded or a predefined maximum number of iterations is reached. The level 1 RRFs need to support two operations on their memory during T_4 (Read and Write). Consequently, they have to be implemented using dual-port distributed RAMs. On the other hand, the level 2 RRFs should store the data for next stage while rotating the currently stored data during T_2 and $T3$. We used the double-buffering technique for level 2 RRFs to support this operation. Since our decoder processes the rows of circulants sequentially, it can easily adapt to any code-rate by simply repeating the T_2 stage in the pipeline.

4.4.2 Summary of Architectural Features

Our QC-LDPC decoder has the following architectural properties:

- **Parallelism:** The decoder provides foldable intra-RoC parallelism. The parallelism is adjustable through determining the number of rows processed in parallel.
- **Shuffle Network:** The decoder's shuffle network implements a hybrid rotation logic that combines in-place shifting with low-cost barrel-shifters.
- **Adaptability:** Since the decoder processes the rows of circulants sequentially, the architecture does not depend on the code's number of rows of circulants. Therefore, it can adapt to a new code with a different number of rows by changing the code registers in the controller.

4.5 Noisy Gradient Decent Bit-Flipping Algorithm

We used our QC-LDPC decoder architecture to implement the simple soft-decision NGDBF algorithm [19]. Algorithm 2 shows the pseudo-code for the NGDBF algorithm. Since all nodes send identical messages to their neighbours, each node needs to compute only one message during each decoding iteration. Therefore, in Algorithm 2, all r_{ij}s and l_{ji}s are replaced with r_is and l_js respectively. The algorithm performs the following computations:

- **Check node processing:** Each check node computes the parity for its connected variable nodes (line 8). A parity result of $+1$ (-1) means the check node is (not) satisfied.

Algorithm 2 Noisy Gradient Descent Bit-Flipping

1: **for** $0 \leq j < n$ **do**
2: $\theta_j = \theta_0$
3: $l_j^{(0)} = (y_j \geq 0)\ ?\ 0\ :\ 1;$
4: **end for**
5: **for** $0 < t < maxIter$ **do**
6: // Check Node Process
7: **for** $0 \leq i < m$ **do**
8: $r_i^{(t)} = (\underset{k \in \mathcal{V}_i}{\oplus} l_k^{(t-1)})\ ?\ -1\ :\ +1;$
9: **end for**
10: // Variable Node Process
11: **for** $0 \leq j < n$ **do**
12: $a_j^{(t)} = \sum_{k \in \mathcal{C}_j} r_k^{(t)}$
13: $flip = \left([l_j^{(t-1)} \times y_j] + a_j^{(t)} < \theta_j\right)\ ?\ 1\ :\ 0;$
14: $l_j^{(t)} = l_j^{(t-1)} \oplus flip;$
15: $u = (flip)\ ?\ -1\ :\ +1;$
16: $\theta_j = \lambda^u \cdot \theta_j$;// Update θ based on the flip decision
17: $\bar{y}_j = l_j^{(t)}$;// Hard Decision for each bit
18: **end for**
19: // Early Exit Condition
20: **if** $\mathbf{H} \cdot \bar{\boldsymbol{y}}^T = \mathbf{0}$; **then**
21: break;
22: **end if**
23: **end for**

▷ n: Number of variable nodes
▷ m: Number of check nodes
▷ $maxIter$: Maximum number of iterations
▷ y_j: Soft information received for bit j
▷ $\theta_j < 0$: Bit flip threshold for bit j
▷ $\theta_0 < 0$: Initial flip threshold
▷ $\lambda < 1$: Threshold update step

- **Variable node processing:** Each variable node computes a value based on:
  - The number of unsatisfied check nodes connected to the variable node.
  - The soft information for the bit received from the channel.
  - The corresponding bit's hard-decision value at the current iteration.

  Then, it flips its corresponding bit if the value is lower than a threshold, θ_j. Each bit has its own threshold and it is updated at each iteration based on the bit's flip decision. If the bit is flipped in the current iteration, its threshold is multiplied by a $\lambda < 1$ to make the bit less likely to flip at future iterations. Otherwise, it is multiplied by λ^{-1} to make the bit more likely to flip in future iterations (line $16 - 17$).

The computations in NGDBF can be simply mapped to our decoder architecture. At first, the check nodes for the first row of circulants are computed. Then, all the rotations of the result required for the variable node processing are calculated. These rotations are partial results of the a_js (line 12 in Algorithm 2). We denote these partial results as ρ_js. Similar operation can be performed based on the offsets of the second row of circulants and the results should be accumulated to the partial results obtained from the previous row. We explain the process through an example.

Consider the following parity check matrix:

$$\mathbf{H}_{10\times 15} = \begin{bmatrix} \mathbf{C}^1_{(5)} & \mathbf{C}^2_{(5)} & \mathbf{C}^3_{(5)} \\ \mathbf{C}^3_{(5)} & \mathbf{C}^4_{(5)} & \mathbf{C}^2_{(5)} \end{bmatrix} \tag{4.12}$$

We have:

$$\begin{array}{lll} l_0 \cdots l_4 & \xrightarrow{\text{Rotate Left (1)}} & l_1 \cdots l_0 \quad \oplus \\ l_5 \cdots l_9 & \xrightarrow{\text{Rotate Left (2)}} & l_7 \cdots l_1 \quad \oplus \\ l_{10} \cdots l_{14} & \xrightarrow{\text{Rotate Left (3)}} & l_{13} \cdots l_{12} \\ \hline & & r_0 \cdots r_4 \end{array} \quad \begin{array}{ll} \xrightarrow{\text{Rotate Left (5-1=4)}} & r_4 \cdots r_3 \quad (\rho_0 \ldots \rho_4) \\ \xrightarrow{\text{Rotate Left (5-2=3)}} & r_3 \cdots r_2 \quad (\rho_5 \ldots \rho_9) \\ \xrightarrow{\text{Rotate Left (5-3=2)}} & r_2 \cdots r_1 \quad (\rho_{10} \ldots \rho_{14}) \end{array}$$

Figure 4.6 illustrates the data flow for processing the first row of circulants with check and variable node process interleaving. It consists of 4 stages:

1. Rotating the variable to check node messages based on a row of circulants.
2. Compute the check to variable node messages for that row of circulants through a set of XOR operations (Check node process).
3. Rotating the check to message node messages based on the complement offsets of the row of circulants.

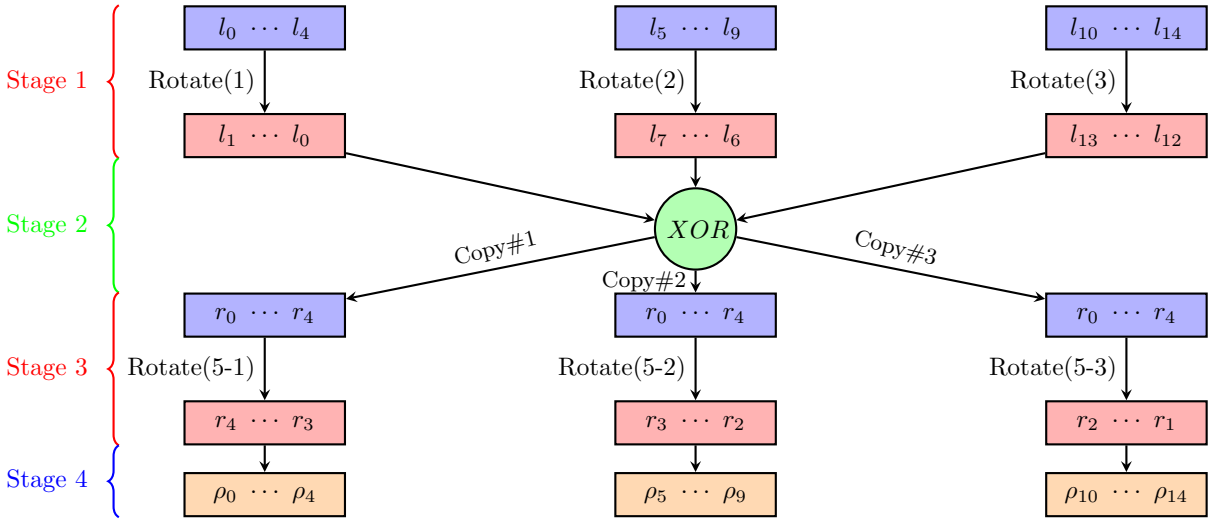


Figure 4.6: Data flow graph for $\mathbf{H}$ in Equation 4.12 processing the first row of circulants

4. Aggregating the results with previous rows of circulants (Variable node process).

This sequence is repeated for each row of circulants and the partial results, ρ_js, are accumulated. Once all rows of circulants are processed, a_js are computed for all bits and the variable nodes can decide whether to flip each bit.

4.5.1 CNU

The CNUs in NGDBF are just a set of XOR gates that perform reduction XOR operations to compute the parity constraints for the check nodes.

4.5.2 VNU

The VNU in NGDBF accumulates the partial results (ρ_js) for each bit and decides whether to flip each bit. Figure 4.7 depicts the data path for this unit. The 3 dashed lines show the activated paths during different operations.

- **Path 1**: This path is active during the accumulation phase where the ρ_js are accumulated in the *Accumulator Memory*.
- **Path 2**: This path is active during the decision-making phase, where the unit decides whether to flip each bit. Instead of explicitly multiplying the threshold by λ or λ^{-1} in each iteration, a counter is used for each bit which is incremented whenever the bit is flipped and decremented otherwise. The values for $\{\theta_0 \cdot \lambda^{-maxIter}, \theta_0 \cdot \lambda^{-maxIter+1}, \cdots, \theta_0 \cdot \lambda^{maxIter}\}$ are pre-computed and stored in the *Theta Memory*. Therefore, the appropriate threshold is read from this memory based on the value of the counter in the *Counter Memory* for each bit.

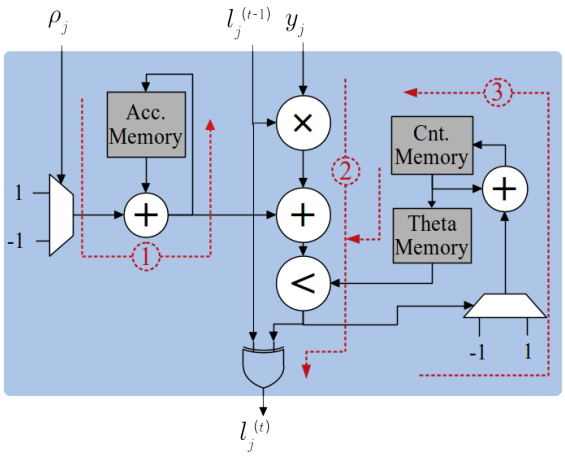


Figure 4.7: Data path for the VNU

- **Path 3:** This path is activated to update the counter for each bit depending on the flip decision.

4.6 Evaluation

In order to perform an efficient evaluation process, we used Verilator [34], a free open-source tool which converts Verilog to a cycle-accurate C++ model. Compared to the Verilog simulation, the generated C++ model can be simulated at a much higher speed. The model is encapsulated as a library where Verilog wires, registers, and functions are accessible.

Figure 4.8 illustrates our framework to perform the evaluation process. It consists of three parts:

- *Simulation*: Verilator 4.014 is used in this part to convert the Verilog design into a C++ library.
- *Validation*: A testbench is written in C++ which invokes the design's golden model to validate the design functionality. Our golden model is a bit-true model of the hardware implementation that takes quantization into account for its computations.
- *Implementation*: The design is connected to a light-weight *Command Processor* and implemented on an FPGA board using VIVADO 2017.4.

We developed a hardware library, *HW Lib*, in C++ which acts as an interface between the three segments of our framework. Through the *HW Lib*, the testbench can transfer the data with both the C++ model and the implemented hardware. For simulation, the *HW Lib* makes function calls to the library generated by Verilator. For emulation, it communicates with the *Command Processor* through universal asynchronous receiver-transmitter (UART) port. The *HW Lib* and the *Command*

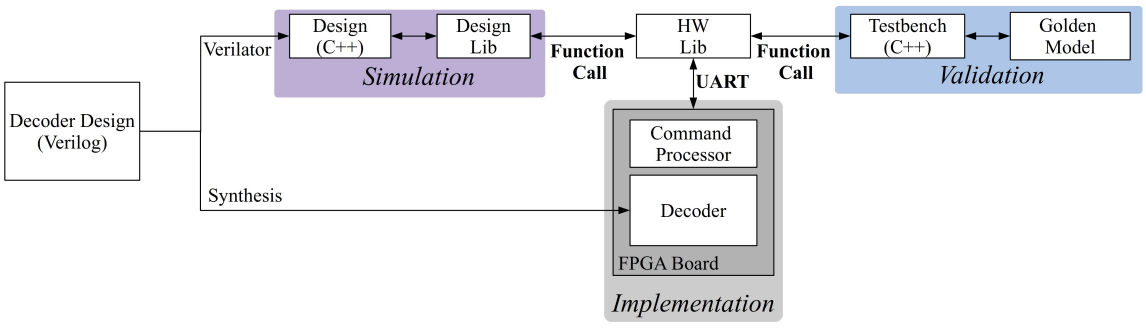


Figure 4.8: Our evaluation framework

Processor modules are generic components and can be utilized for any design with two FIFOs and two BRAMs interfaces for command and data transfer respectively. This makes the framework reusable for evaluating any design.We used the NetFPGA-SUME development board that features a Virtex-7 FPGA (XC7V690T FFG1761-3).

4.6.1 RRF Performance

For an RRF with size, z, and stride size s, where $z = b \times s + q$, and data is stored in $b+1$ memory banks and it takes $s+q$ clock cycles to generate a complete rotation in the output (s cycles for strided circular access and q cycles for the sequential surplus accesses). Therefore, throughput is computed as follows:

$$Throughput = \frac{f_{max} \times z}{s+q} \tag{4.13}$$

where f_{max} is the maximum clock frequency. The efficiency of an RRF, is computed by normalizing its peak throughput based on its resource usage. Consequently, we define the following efficiency metric:

$$Efficiency = \frac{Throughput \times 1000}{LUT\ count} \tag{4.14}$$

In fact, *Efficiency* represents the achievable throughput per one thousand LUTs.

We studied the RRF performance for stride sizes of 4, 8, 16, 32, and 64 and the word sizes, w, of 1, 4, and 8 bits. We also investigated the performance under a wide range of circular access sizes. Figure 4.9a illustrates the peak throughput for circular access size z=251. As mentioned before, smaller stride sizes result in higher parallelism which leads to higher throughput. However, despite the general increasing

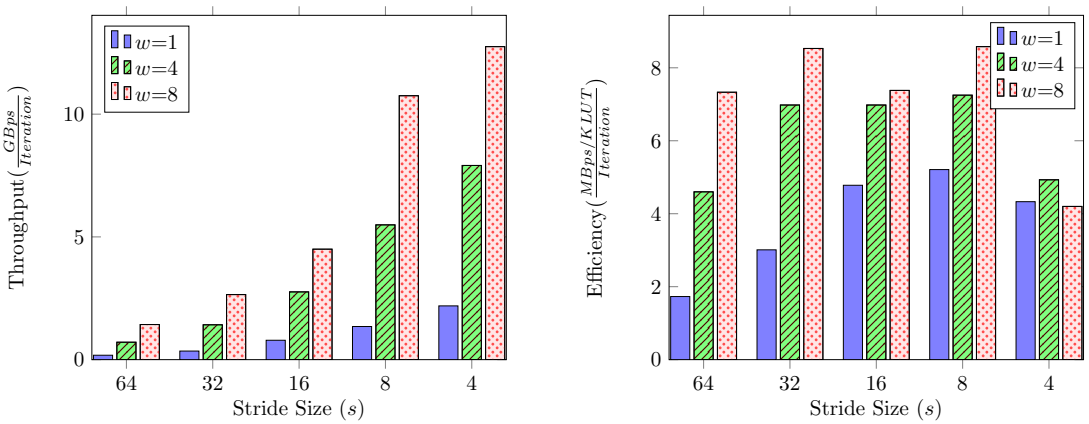


Figure 4.9: (a) Peak throughput and (b) Efficiency, for RRF with z=251

trend, there are cases that the throughput does not increase with the same rate as the parallelism *e.g.*, from s=8 to s=4. This phenomenon is observed when a large fraction of the RRF's latency is due to the sequential surplus accesses *i.e.*, $0 \ll \frac{q}{q+s}$.

The graph in Figure 4.9b shows that the RRF's efficiency for z=251. In general, RRFs with larger word sizes are more efficient since the address generation logic is shared among the word bits. However, this property is not observed in designs where $0 \ll \frac{q}{q+s}$ *e.g.*, s=4 in Figure 4.9b. In such cases, the address logic sharing among multiple word bits does not compensate the imposed sequential surplus accesses. Figure 4.9b shows that depending on the circular access and the word size, our RRF provides an opportunity to find the most efficient design through design space exploration.

Since RRF's structure necessitates serial processing of the surplus accesses, the *surplus set* size determined by q in Equation 4.10 affects the overall efficiency of the RRF. Figure 4.10 plots the efficiency with respect to the *surplus set* size for $s = 16$ and the number of banks, $b \in \{12, 24, 38, 48\}$. Intuitively, increasing the *surplus set* size decrease the efficiency as it imposes more sequentiality to the RRF. However, the results show a non-linear impact of the *surplus set* size on the efficiency.

Circular Access Size

Figure 4.11 illustrates the RRF's efficiency for circular access size, $z \in \{75, 251, 451\}$ the stride size, $s \in \{4, 8, 16, 32, 64\}$. The results show that for small circular access sizes ($z = 75$) the efficiency increases with higher parallelism. In fact, the more sequential RRFs have larger memory banks with wider address signals which result in more complex address generation logic. However, this effect is diminished for the

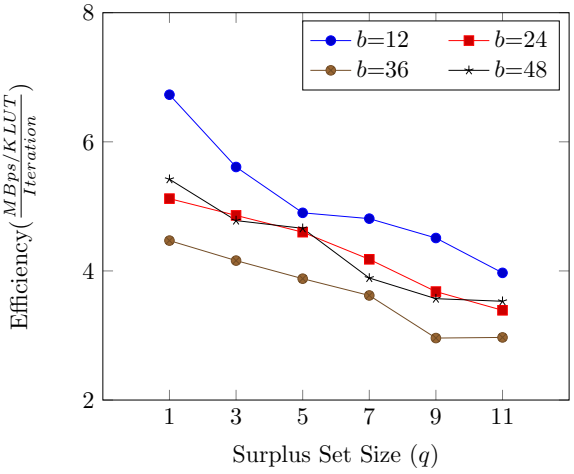


Figure 4.10: The *surplus set* size impact

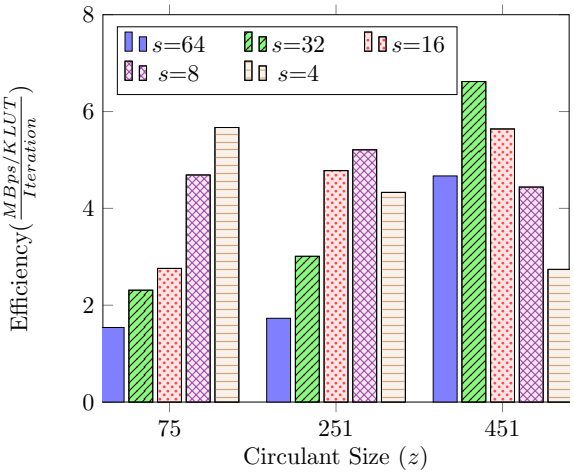


Figure 4.11: RRF's efficiency

RRFs with larger circular access sizes ($z = 251$ and $z = 451$) due to their relatively higher throughput. Consequently, a design space exploration is required to find the most efficient design for these cases.

Logic Sharing Impact

Figure 4.12 depicts the impact of logic sharing discussed in Section 4.3.3 for stride sizes $s \in \{4, 8, 16, 32, 64\}$. The results confirm that logic sharing can improve the resource usage by $5\% - 10\%$ depending on the parallelism level. Moreover, a majority of logic sharing occurs in the address generation logic. Therefore, more sequential designs in which wider addresses need to be generated, have relatively higher gain from the logic sharing.

Soft-MUX Vs. Hard-MUX RRF

In order to evaluate the impact of leveraging FPGA's inherent micro-architecture,we compared the performance for soft-MUX and hard-MUX RRFs. The former utilized FPGA's soft logic while the latter uses physically implemented multiplexers in the FPGA. Table 4.2 reports the LUT count and the efficiency for both RRFs. The results show that the hard-MUX RRF consumes $36\% - 64\%$ less lookup tables. In addition, hard-MUX RRFs are more flexible for routing and placement. As a result, they are 1.5x to 4x more efficient than the soft-MUX RRFs.

There is a non-linear relationship between the parallelism and the resource usage. The more parallel the RRF, the more logic resources are required to implement its first and second level rotation logic. On the other hand, the more serial RRFs have more complicated address calculation logic since more address bits need to be generated and

Stride	LUT Count		Efficiency (GB/s/KLUT)	
	Soft MUX	Hard MUX	Soft MUX	Hard MUX
4	784	505	2.81	4.33
8	548	260	2.45	5.21
16	466	166	1.22	4.78
32	302	116	0.97	3.01
64	255	106	0.58	1.73

Table 4.2: Soft-MUX Vs. Hard-MUX RRFs for $\mathbf{n} = 251$

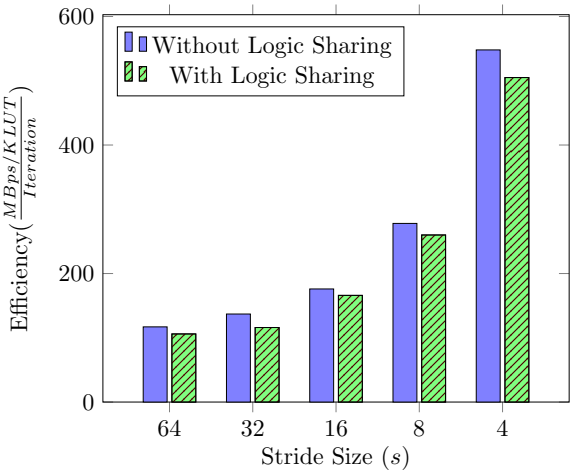


Figure 4.12: The impact of logic sharing, z=251

routed to the distributed RAMs. This results in a non-linear relationship between the stride size and the fraction of the resources consumed by an RRF's first-level rotation logic. Since the hard-MUXes are only leveraged in the first-level rotation logic, the results in Table 4.2 demonstrate a non-linear improvement achieved by leveraging the hard-MUXes with respect to the stride size.

4.6.2 NGDBF Decoder Performance

According to the decoder's pipeline timing in Figure 4.5, the throughput of our NGDBF decoder can be computed as follows:

$$Throughput_{/Iteration} = \frac{n \times f_{max}}{(s+q) \times (r+2) + 5} \tag{4.15}$$

where n is the codeword length, f_{max} is the maximum frequency, s is the RRF memory size, x is the circulant size, and r is the number of rows of circulants in the $\mathbf{H}$. The added 5 cycles is the pipeline latency.

Parallelism Vs. Efficiency

We generated four variants of our NGDBF decoder architecture for a (9650, 1351) QC-LDPC code with circulant size 193. The variants differ in terms of their parallelization. Figure 4.13 illustrates the efficiency and throughput for all variants. The parallelization values on the x axis represent the number of rows of $\mathbf{H}$ processed in parallel. The results show the increasing trend of the throughput as the parallelization is increased. However, the efficiency, shows a different trend. This is due to the fact that although the RRF's memory banks scale with the parallelism, its second-level rotation (the barrel-shifter) scales at a relatively faster rate. Moreover, the

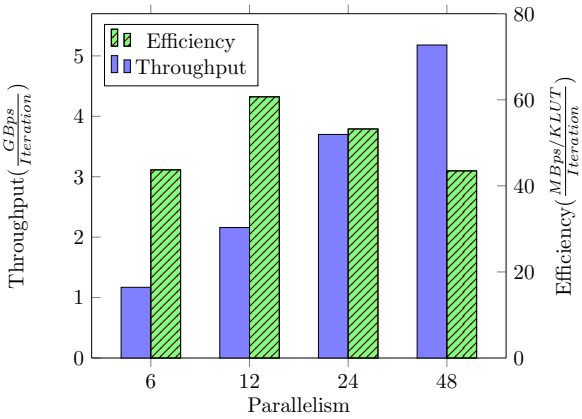


Figure 4.13: Parallelism Vs. performance for the QC-LDPC (9650, 1351) code

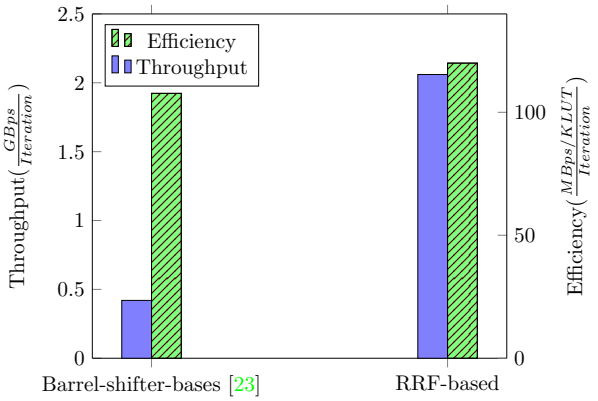


Figure 4.14: Foldable parallelism and performance

address generation logic's LUT utilization increases as we decrease the parallelism. These properties results in a sweet spot for the most efficient design in terms of parallelism. The foldable parallelism provided by our decoder architecture provides an opportunity to find the most efficient design by sweeping the parallelism.

WiMax 802.16e LDPC Code

For comparison, we compiled our NGDBF decoder architecture for the medium-size (2304, 1152) QC-LDPC code with circulant size, z=96, used in WiMax 802.16e standard. Figure 4.14 compares our decoder's performance with the barrel-shifter implementation proposed by Amaricai and Boncalo [23]. The results show that our decoder is about 11% more efficient while achieving 400% higher throughput.

4.7 Summary

In this chapter, we proposed a configurable micro-architecture for QC-LDPC decoders. We leveraged the FPGA's inherent physical architecture to design RRF, a foldable parallel structure for strided circular access. Then, we used the RRF as the basic block to propose an elegant structure for QC-LDPC decoders. Rather than being *yet-another-design*, our decoder architecture is based on a regular structure that can be configured for many variants. The architecture's support for foldable parallelism enables striking a trade-off between performance and resource usage. Moreover, the decoder can adapt to any code and code rate based on the application requirements. The architecture's support for foldable parallelism and rate-adaptability enables design space exploration for pursuing interesting trade offs between hardware efficiency and code performance.

Chapter 5

Learning FAID: A Hardware-Code Co-design

The micro-architecture proposed in Chapters 4 is mainly focused on hardware efficiency and foldable parallelism that facilitate the design space exploration. This chapter aims to explore interesting trade offs between hardware efficiency and code performance in soft-decision QC-LDPC decoders for flash memories. To that end, we explore the limitations of error correction in flash memories and propose an end-to-end solution for improving the correction capability of the conventional soft-decision decoders for these devices. Then, starting from the micro-architecture discussed in Chapter 4, we propose an efficient foldable parallel micro-architecture for our soft-decision decoder. Considering hardware and code efficiency simultaneously has made this effort a hardware-code co-design.

5.1 Overcoming Flash Memory Error Correction Constraints

A practical error correction mechanism for a flash memory should carefully consider its inherent constraints. For instance, the limited bit budget for ECC necessitates the use of high-rate codes in these devices. Aside from resource scarcity, there are some other constraints that directly impact the correction capability of ECC in flash memories. However, these constraints shrink the decoder design space which can be leveraged to design decoders with improved code performance.

5.1.1 Constraint 1: Limited Quantization

Soft-decision LDPC decoders rely on accurate soft information for their computations. In a discrete domain, more quantization levels result in better correction capability.

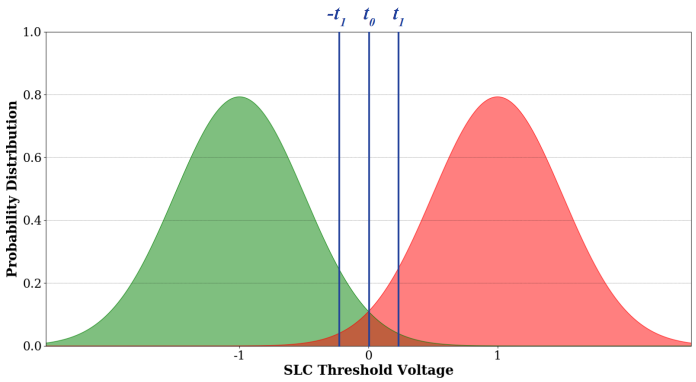


Figure 5.1: SLC voltage threshold distribution

However, a fine-grained quantization is not feasible for flash memories. A flash memory cell value is read through sensing which is done by sweeping a threshold voltage on a word-line. Figure 5.1 illustrates a simplistic model of the threshold voltage distribution for a single-level cell (SLC) memory in which each cell stores one information bit. The distribution is a combination of two Gaussian random variables. In particular, cells with information bit 0, are modelled as a Gaussian random variable with mean +1 and variance $\sqrt{N_0/2}$. Similarly, the cells with information bit 1, are modelled as a Gaussian random variable with mean -1 and variance $\sqrt{N_0/2}$. The vertical lines represent the word line voltages for memory sensing. they divide the distribution into four regions which can be represented by two bits.

Flash memory read latency is linearly proportional to the number of sensing quantization levels. Each extra soft information quantization bit not only doubles the required number of sensing levels, but it also increases the flash-to-controller latency. This is due to the fact that the read result must be transferred to the memory controller through standard chip-to-chip links. Since flash memories are classified as fast storage devices, increasing their read latency for error correction contradicts their existential philosophy. Consequently, the quantized soft information has to be limited to at most 2-3 bits *i.e.*, 4-8 quantization levels and 3-7 sensing levels.

5.1.2 Constraint 2: Limited Decoding Iterations

As discussed in previous chapters, LDPC decoding is an iterative process in which the computations iterate until either the codeword is successfully decoded or a predefined number of iterations has reached, in which case the decoder has been unsuccessful at decoding the codeword. A hardware implementation of the decoder should perform the decoding iterations either sequentially or in a pipeline unrolled fashion. Although more decoding iterations could typically improve the code correction capability, it in-

creases the decoding latency which, in turn, diminishes hardware efficiency regardless of the implementation approach. Consequently, a practical ECC decoder for a flash memory has to limit the decoding iterations. Our code studies showed that setting the maximum decoding iteration of 4-5 would be reasonable. In addition, improving the computations to perform decoding in as few iterations as possible could directly improve the hardware efficiency without affecting the code correction capability.

5.1.3 Leveraging A Finite Design Space

The inherent constraints in flash memory error correction have paralyzed the conventional decoders to meet the device's error correction requirements. These decoders typically perform algorithm design and quantization as separate steps. However, since decoding is performed using a finite set of values and it is repeated for a limited number of iterations, the entire decoder design space is finite for flash memories. In this thesis, we introduce a learned finite alphabet decoder, a decoder for a finite design space. In particular, we offer an end-to-end solution that utilizes a brute-force machine learning approach along with finite alphabet iterative decoding (FAID) to learn the best decoder for a given code and quantization.

5.2 Finite Alphabet Iterative Decoding (FAID)

Conventional LDPC decoders are mainly based on message-passing belief propagation (BP) in which the decoder operates on a graphical model of a code, the Tanner graph, to compute the most-likely value for each bit during each iteration. However, these algorithms may not be as effective when decoders are realized in hardware and the effect of finite precision is in place.

There has been several efforts to compensate the negative impact of limited quantization on LDPC decoders. The majority of these efforts suggest modifying the variable node function based on some knowledge of the code or the transferred messages. In this section, we first use the existing literature to define the framework for quantized decoders. Then, we focus on a special class of these decoders, FAID, which appear to be promising for limited-quantization decoding.

5.2.1 Framework

For a (n, k) binary LDPC code with n variable nodes and $m = n - k$ check nodes in its Tanner graph. A quantized decoder, $\mathcal{Q}$, can be defined as $\mathcal{Q} = \{\mathbf{\Phi}, \mathbf{\Psi}, \mathcal{M}, \mathcal{Y}\}$

where:

$$\mathcal{M} = \{0, \pm L_i | L_i \in \mathbb{R}^+, \forall 1 \leq i < j \leq S, L_i < L_j\} \tag{5.1}$$

is the finite set of transferred messages with $2S+1$ alphabet, $\mathcal{Y}$ is the set of possible channel values, and $\mathbf{\Phi}$ and $\mathbf{\Psi}$ are variable to check node, and check to variable node functions respectively. For the check node, C_i, with the set of indices of connected variable nodes $\mathcal{V}_i$, the degree d_c, and the received messages $\{l_{i,j}^{(t-1)} | j \in \mathcal{V}_i, l_{i,j}^{(t-1)} \in \mathcal{M}\}$, at decoding iteration t, the response to the variable node V_j can be computed as follows:

$$r_{i,j}^{(t)} = \mathbf{\Psi}(\mathcal{V}_i, j, t-1), \qquad \mathbf{\Psi} : \mathcal{M}^{d_c} \times d_c \times T \longrightarrow \mathcal{M}$$

where T is the maximum number of decoding iterations. Likewise, for the variable node, V_j, with the set of indices of connected check nodes $\mathcal{C}_j$, the degree d_v, the received messages $\{r_{i,j}^{(t)} | i \in \mathcal{C}_j, r_{i,j}^{(t)} \in \mathcal{M}\}$, and the channel value $y_j \in \mathcal{Y}$, at decoding iteration t, the output message to check node C_i, can be computed as follows:

$$l_{i,j}^{(t)} = \mathbf{\Phi}(y_j, \mathcal{V}_j, i, t-1), \qquad \mathbf{\Phi} : \mathcal{Y} \times \mathcal{M}^{d_v} \times d_v \times T \longrightarrow \mathcal{M}$$

where T is the maximum number of decoding iterations. A FAID, $\mathcal{F} = \{\mathbf{\Phi}_\mathcal{F}, \mathbf{\Psi}_\mathcal{F}, \mathcal{M}, \mathcal{Y}\}$ is distinguished from conventional quantized decoders by its update functions, $\mathbf{\Phi}_\mathcal{F}$ and $\mathbf{\Psi}_\mathcal{F}$. Rather than the messages being the approximation of the log-likelihoods or probabilities, the functions are based on simple well-defined maps. To the best of our knowledge, the majority of the prior FAIDs are focused on modifying the variable node function, $\mathbf{\Phi}_\mathcal{F}$, while keeping the check node function, $\mathbf{\Psi}_\mathcal{F}$ same as conventional decoders. In particular, a FAID's variable node function is defined as a quantization function over the conventional update functions:

$$\mathbf{\Phi}_\mathcal{F}(y_j, \mathcal{V}_j, i, t-1) = \mathcal{Q}\big(\mathbf{\Phi}(\alpha \times y_i, \mathcal{V}_j, i, t-1)\big)$$

where α is a weight parameter that can be computed based on a linear or non-linear function. Considering a threshold set:

$$\mathcal{T} = \{\theta_i | \quad \forall 1 \leq i < j \leq S+1, \theta_i < \theta_j, \theta_{S+1} = \infty\}, \tag{5.2}$$

the $\mathcal{Q}$ is a quantization function defined as:

$$\mathcal{Q}(x) = \begin{cases} sgn(x) L_i, & \theta_i \leq |x| < \theta_{i+1} \\ 0, & |x| \leq \theta_1 \end{cases} \tag{5.3}$$

5.2.2 FAID History

The first major work in the area of quantized decoders was introduced by Richardson and Urbanke [35], in which they used density evolution (DE), to approach the performance of the floating-point BP decoder. DE is a technique that recursively computes the probability mass function (PMF) of the transferred messages assuming infinite code length with a cycle-free Tanner graph. Since then, normalized min-sum and offset min-sum decoders by Chen et al. [36] are among the main decoders that used DE to reach floating-point BP correction capability. However, the DE technique does not guarantee a good performance on a finite length code. In addition, it may increase the implementation complexity.

The term, FAID, was first introduced by Planjery et al. [37] in which they proposed a variable node function based on simple well-defined maps. The maps were designed based on the knowledge of the code's trapping sets with the goal of increasing the correction capability in the error-floor region. Being a simple map function, allows the variable node update rule, $\mathbf{\Phi}(.)$, to be implemented using a lookup table. This approach is only feasible for limited-quantization and small d_v values. Their experimental results showed that for a $d_v = 3$ code, their FAID can surpass the floating-point BP in the error-floor region.

In order to achieve a simpler implementation, Truong et al. [38] suggested a *non-surjective* FAID in which the messages are stored with a lower precision than the precision they are being transmitted. Using density evolution, the non-surjective FAID can be optimized for any given code. Their experimental results showed a 25%/35% reduction in memory/interconnect compared to the MS decoder while improving the decoding gain up to 0.36 dB for a $d_v = 3$ code.

In order to improve the correction capability, probabilistic FAID (P-FAID) was introduced by Le et al. [39]. Rather than using a single lookup table, as its name suggests, P-FAID uses multiple lookup tables probabilistically. The probabilities for using different lookup tables are computed based on the density evolution of the transferred messages. Using a smart computation flow, P-FAID realizes the probabilistic behavior without a real random generator. The experimental results showed 0.2 dB decoding gain at no hardware overhead compared to the MS decoder for a $d_v = 3$ code.

Vasic et al. [40] took a different approach for designing the FAID's variable node function. Rather than leveraging the knowledge of the code or analyzing the transferred messages, they took a machine learning approach to learn the best variable node update function. Then, they used the learning process outcome to design a LUT-based FAID. Their experimental results showed that a 3-bit FAID improves the

Table 5.1: Major prior efforts on FAID

Work	Year	Idea	Achievement	d_v of interest
FAID [37]	2012	Using the knowledge of the trapping sets	Improved noise threshold in the error-floor region over floating-point BP	3
NS-FAID [38]	2016	DE for lower precision messages	25% Memory Reduction, 35% Interconnect reduction over MS	3
P-FAID [39]	2018	DE for using multiple $\mathbf{\Phi}$s probabilistically	0.2 dB noise threshold improvement over MS	3
Learning FAID [40]	2018	Learning the update rules	0.4 dB noise threshold improvement over MS	3

MS's noise threshold by 0.4 dB for a $d_v = 3$ code.

Table 5.1 summarizes the major prior efforts on finite alphabet decoders mentioned above. To the best of our knowledge, the majority of academic works on FAID are mostly suggesting some knowledge of the code or messages to improve the correction capability of conventional BP and MS decoders in a limited-quantization setting. Moreover, they focus on codes with $d_v = 3$ in which case the variable update function can efficiently be implemented with using a lookup table. However, the LUT size grows exponentially with larger d_v values. The idea of learning the best FAID was first explored by Vasic and the results showed a promising threshold improvement [40].

5.3 Learning Decoder

Any LDPC decoder has three computation steps in each iteration; Variable to check node function ($\mathbf{\Phi}$), check to variable node function ($\mathbf{\Psi}$) , and variable node update rule ($\mathbf{\Omega}$), in which a decision value is computed for each bit. In a conventional MS decoder, these computations are based on log-likelihood of the transmitted message bits. A learning decoder, is a decoder that learns some aspects of these computations through assigning a set of weight parameters and takes a machine learning approach to find the best values for these parameters.

5.3.1 Framework

We define the learning framework based on the conventional MS and APP decoding algorithms. Given the Equations 3.5 and 3.6 for an MS decoder, the $\mathbf{\Phi}$ and $\mathbf{\Psi}$

functions can be defined as:

$$r_{i,j}^{(t)} = \mathbf{\Psi}(\mathcal{V}_i, j, t-1) = \prod_{k \in \mathcal{V}_i/j} \lambda_{i,j,k}^{(t)} \times sign(l_{i,k}^{(t-1)}) \times \min_{k \in \mathcal{V}_i/j} |l_{i,k}^{(t-1)}| \quad \forall i,j : r_{i,j}^{(0)} = 0 \tag{5.4}$$

$$l_{i,j}^{(t)} = \mathbf{\Phi}(y_j, \mathcal{C}_j, i, t) = \alpha_{i,j}^{(t)} \times LLR(y_j) + \sum_{k \in \mathcal{C}_j} (\beta_{i,j,k}^{(t)} \times r_{k,j}^{(t)}) \tag{5.5}$$

where the superscript t denotes the decoding iteration and α, β, and λ are the learning parameters. As a general learning framework that supports the entire design space, the subscripts, i, j, and k, and the superscript, t, on the learning parameters are to allow a separate set of parameter to be used for each individual message per iteration. Similarly, the variable node update rule can be defined as:

$$\bar{y}_j^{(t)} = \mathbf{\Omega}(y_j, \mathcal{C}_j, t) = \begin{cases} 0, & (\gamma_j^{(t)} \times LLR(y_j)) + \sum_{k \in \mathcal{C}_j} (\theta_{k,j}^{(t)} \times r_{k,j}^{(t)}) \geq 0, \\ 1, & \text{otherwise.} \end{cases} \tag{5.6}$$

5.3.2 History

The general learning framework defined in Section 5.3.1 defines a vast design space for learning parameters. However, exploring this entire design space is impractical for large block codes. Moreover, a practical implementation of such hypothetical learned decoder is not feasible for large block codes. Therefore, the existing learning decoders have made efforts to explore a segment of this design space.

Nachmani et al. [41] proposed a learning method to improve the Φ and Ω functions on the BP algorithm. They used deep learning techniques to train $\alpha_i^{(t)}$, $\beta_{i,j,k}^{(t)}$, $\gamma_j^{(t)}$, and $\theta_{k,j}^{(t)}$ parameters for a BCH(63, 45) code. Compared to Equations 5.5 and 5.6, the dropped subscripts are due to design space reduction. Their experimental results showed up to 0.9 dB threshold improvement for a learned BP decoder. Moreover, given a noise threshold, their learned BP converges 10× faster than the conventional decoder.

A learning MS decoder was proposed by Lugosch and Warren [42] in which they focused on learning the check to variable node function, $\mathbf{\Psi}$. In fact, they used a neural network to train $\lambda_{i,j}^{(t)}$ and their experimental results showed up to 1 dB noise threshold improvement compared to a BP decoder for a BCH(63, 45) code.

Wu et al. [43] proposed a learning LDPC decoder that took a machine learning approach to optimize the $\mathbf{\Psi}$ and $\mathbf{\Omega}$ for the MS decoder. They trained the network to find the best values for $\lambda_{i,j}^{(t)}$ and $\gamma_j^{(t)}$ parameters. Their experimental results showed

that their learned decoder improves MS decoder's noise threshold by 0.2 dB.

Vasic et al. [40] were the first that took a learning approach to design a FAID for limited quantization. After training the $\alpha_{i,j}^{(t)}$, and $\beta^{(t)}$ parameters on variable to check node function, $\mathbf{\Phi}$, they designed a finite alphabet based on these learned parameters. Their experimental results show that the a 3-bit learned FAID has 0.4 dB noise threshold gain over the MS decoder. Moreover, given a noise threshold, the 3-bit learned FAID converges 2× faster than the MS decoder which, in turn, results in increased hardware efficiency.

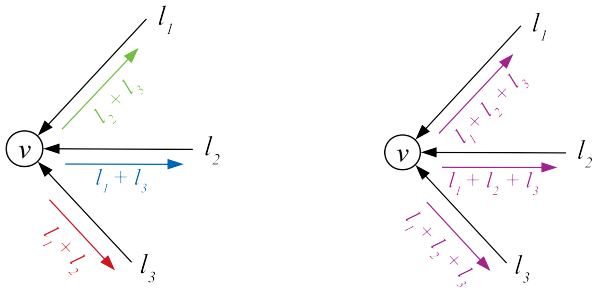
5.4 Learning FAID: A Decoder for A Finite Design Space

The flash memory error correction constraints, limited quantization and number of decoding iterations, results in a finite design space for a decoder for these devices. Meanwhile, prior FAID and learning decoders have shown to be promising in improving the threshold and the number of decoding iterations in a limited quantization setting. To the best of our knowledge, Vasic et al. [40] were the first that proposed a machine learning approach to designing a FAID. However, their suggested design space for learning necessitates using a specific FAID table for each variable node. Despite being reasonable in their case study with small block codes, this approach is not practical for a large codeword decoder with a partially parallel architecture in which each computational unit is shared among multiple variable nodes. In this thesis, we propose an end-to-end solution that takes a brute-force machine learning approach to find the best FAID for a given code. Considering this research's mandate to respect hardware and code performance simultaneously, our approach is distinguished from Vasic's work in the following aspects:

- Rather than using the min-sum as the baseline decoder, we start from the more hardware-friendly APP-based decoder discussed in Chapter 2.
- Compared to Vasic's FAID [40], we reduce the FAID design space to make it practical for large block codes.
- We propose a scalable 2-stage lookup-table approach that enables efficient FAID implementation for larger d_v values.

5.4.1 Conventional APP-based Decoder

As discussed in Section 3.2.2, compared to the MS decoder, the variable nodes in the APP-based decoder send identical messages to all their connected check nodes.



(a) (b)

Figure 5.2: Variable node computation (a) MS Vs. (b) APP-based

Figure 5.2 compares the variable node computations in MS and APP-based decoders for a variable node with $d_v = 3$. The identical messages in the APP-based decoder not only results in a simpler hardware implementation for variable node computation, but it also reduces the required memory to communication the messages to the check nodes. This results in a significantly lower-complexity decoders especially in case of large block codes with a large number of variable nodes. However, compared to the MS decoder, the APP-based decoder suffers from a code performance degradation due to its correlation of the variable node messages. Given the limited decoder design space for flash memories, we aim to learn some aspects of decoding and design a FAID to improve APP-based decoder's performance in flash memories while preserving its hardware-friendliness.

5.4.2 Design Space

According to Vasic's learning design space [40]:

$$l_{i,j}^{(t)} = \alpha_{i,j}^{(t)} \times LLR(y_i) + \beta^{(t)} \sum_{k \in \mathcal{C}_j / i} r_{k,j}^{(t)} \tag{5.7}$$

a separate weight α is trained for each message sent from variable node i to check node j. This results in a specific update function per variable node per its outgoing message. A FAID designed based on such a learned decoder has to have separate lookup tables per variable node per outgoing message. However, this is not practical for large block codes where only a partially parallel implementation is feasible and multiple variable nodes have to share a single processing unit. Moreover, assigning different weights to each variable node message is essentially equivalent to discriminating the channel values which disrupts the channel information symmetry.

Consequently, the design space can be reduced to:

$$l_{i,j}^{(t)} = \alpha^{(t)} LLR(y_i) + \beta^{(t)} \sum_{k \in \mathcal{C}_j / i} r_{k,j}^{(t)} \tag{5.8}$$

where the subscripts i, j are dropped from the α parameter so that all outgoing variable node messages have the same weight. This reduction results in learning 2 parameters per iteration. The design space can further be reduced through normalization:

$$l_{i,j}^{(t)} = LLR(y_i) + \beta^{(t)} \sum_{k \in \mathcal{C}_j / i} r_{k,j}^{(t)} \tag{5.9}$$

Finally, since we are focused the more hardware-friendly APP based decoding algorithm, the variables node send same message to all their connected check nodes. Therefore, we drop the exclusion of the check node C_i message ($r_{i,j}^{(t)}$), from the summation in Equation 5.9:

$$l_{i,j}^{(t)} = LLR(y_i) + \beta^{(t)} \sum_{k \in \mathcal{C}_j} r_{k,j}^{(t)} \tag{5.10}$$

In fact, our learning design space suggests learning only one parameter per decoding iteration to perform the weighted sum of the channel value and the incoming messages. In other words, the incoming messages to variable nodes have a different weight in computing the node message in each iteration.

5.4.3 Learning Framework

Given a Tanner graph of a (n, k) LDPC code with n variable nodes and $m = n - k$ check nodes and the maximum number of decoding iterations, L, we construct our $2L + 1$ layer neural network for training.

Neural Network

We explain the network structure through an example. Figure 5.3 illustrates the neural network for a $(6, 3)$ LDPC code with 4 decoding iterations and check node degree $d_c = 4$. The even layers $(a_0, a_2, \ldots, a_{2L})$ perform the variable node function, $\mathbf{\Phi}$, while the check node function, $\mathbf{\Psi}$, is performed by the odd layers $(a_1, a_3, \ldots, a_{2L-1})$. In fact, every 2 consecutive layers represent one decoding iteration while the last layer computes the final value for each variable node at the end of last iteration. The variable node (even) layers have n neurons each, one per variable node. For odd layers, since each check node sends a separate message to each of its connected variable node,

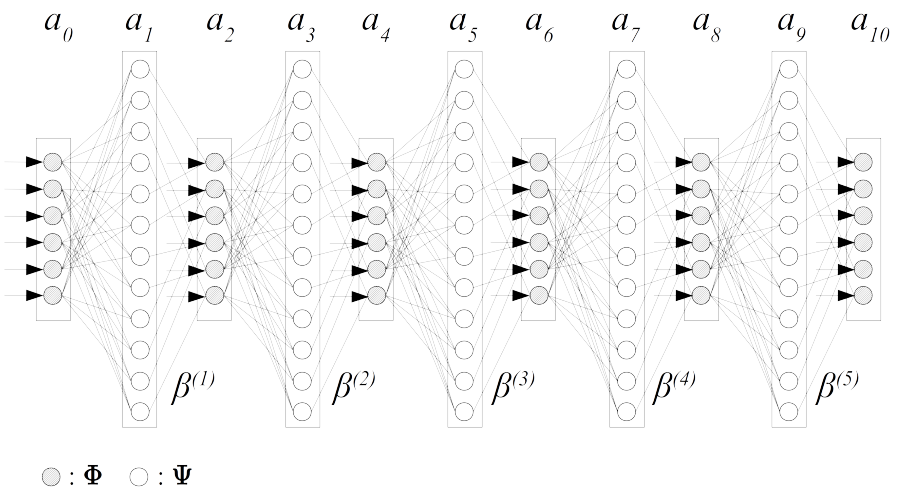


Figure 5.3: Neural network for training the decoder with 5 decoding iterations

we replace each check node of the Tanner graph with multiple neurons in the odd layers, each representing an outgoing message of the check node. Therefore, a neural network for a regular LDPC code with check node degree d_c, has $m \times d_c$ in its odd layers. The bias to intermediate variable node layers represent the channel value used in each decoding iteration.

The connection between the layers is defined as follows:

- **Even to odd layers**: The connection matrix is a $m.d_c \times n$ matrix. Let the index set $\mathcal{I}_i = \{i_0, i_1, \cdots, i_{w^r-1}\}$ be the indices of 1s of the i^{th} row of the parity check matrix, $\mathbf{H}$, with row weight w^r. Then, for each row in the parity check matrix, $\mathbf{H}$, we have a submatrix with w^r rows in the connection matrix where:

$$\forall 0 \leq j < w^r, \quad \mathcal{I}_j = \mathcal{I}_i - \{i_j\} \tag{5.11}$$

  where $\mathcal{I}_j$ is the index set of 1s in the j^{th} column of the submatrix.

- **Odd to even layers**: The connection matrix is a $n \times m.d_c$ matrix. Let the index set $\mathcal{I}_i = \{i_0, i_1, \cdots, i_{w^r-1}\}$ be the indices of 1s of the i^{th} row of the parity check matrix, $\mathbf{H}$, with row weight w^r. Then, for each row in $\mathbf{H}$, we have a submatrix with w^r columns in the connection matrix where:

$$\forall 0 \leq j < w^r, \quad \mathcal{I}_j = \{i_j\} \tag{5.12}$$

  where $\mathcal{I}_j$ is the index set of 1s in the j^{th} column of the submatrix.

Data Set

Given a codeword $\boldsymbol{c} = \begin{bmatrix} c_1 & c_2 & \cdots & c_n \end{bmatrix}$ and its transmitted symbols $\hat{\boldsymbol{c}} = \begin{bmatrix} \hat{c}_1 & \hat{c}_2 & \cdots & \hat{c}_n \end{bmatrix}$, the receiver receives $\hat{\boldsymbol{y}} = \hat{\boldsymbol{c}} + \boldsymbol{n}$ where $\boldsymbol{n}$ is the noise vector, and $\hat{\boldsymbol{y}}$ is quantized to $\boldsymbol{y}$. The LLRs of the received message is $\boldsymbol{\Lambda} = [\lambda_1, \lambda_2, \ldots, \lambda_n]$, where:

$$\forall 1 \leq i \leq n, \qquad \lambda_i = log\frac{Pr(c_i = 0|y_i)}{Pr(c_i = 1|y_i)} \tag{5.13}$$

One advantage of training a neural network for a decoder is that given the channel model, it would be possible to generate infinite size data set due to the random nature of the noise vector. Moreover, since the expected output of the network is known in advance, the training process can proceed without any supervision.

Functions

Let $\mathbf{I}_i$ be the input vector to layer i of the neural network. Then, for the input layer, we have:

$$\mathbf{I}_0 = \boldsymbol{\Lambda} \tag{5.14}$$

where $\boldsymbol{\Lambda}$ is the LLR computed by Equation 5.13. According to Equation 5.10 the learning parameter, β is a weight multiplied by all the incoming messages to the variable nodes. Therefore, we apply the weight to the odd-to-even layer edges and force same weight among all edges for each stage representing a decoding iteration. Therefore, for hidden layers, we have:

$$\forall 0 \leq i \leq 2L, \qquad \mathbf{I}_{i+1} = \begin{cases} LLR(y_i) + \beta^{(\frac{i}{2}+1)}\mathbf{I}_i, & \text{if } i \text{ is even,} \\ sgn(\mathbf{I}_i)min(|\mathbf{I}_i|), & \text{if } i \text{ is odd.} \end{cases} \tag{5.15}$$

where L is the number of decoding iterations, $\beta^{(t)}$ is the parameter for the t^{th} decoding iteration subject to optimization, and $\mathbf{I}_{2L+1} = \mathbf{x}$ is the decoder's output.

Loss Function

In order to compute the loss to perform the back propagation, we first apply the non-linear *Sigmoid* function to the network's output to convert the likelihood messages into probability:

$$\sigma(x_i) = (1 + e^{-x_i})^{-1} = Pr(c_i = 0|y_i) \tag{5.16}$$

A decoder can be mapped to a binary classification problem in machine learning in which the outputs have to be classified in two categories; 0 and 1. Since LDPC decoding algorithm operates based on the probability of the bits being 0 or 1, a binary

cross-entropy loss can be used:

$$BCELoss(\mathbf{x}, \mathbf{c}) = -\frac{1}{n}\sum_{i=1}^{n}(1-c_i)log(\sigma(x_i)) + c_i log(1-\sigma(x_i)) \tag{5.17}$$

However, the LDPC decoding is an iterative process, and although the ultimate goal is to eventually decode the codeword at the end of the last iteration, it would be preferable for the codeword to be decoded in fewer iterations. Therefore, the intermediate decoding iterations should impact the calculation of the loss. Unlike Vasic's learning framework [40], we use a *multi-loss* function as follows:

$$\Gamma(\mathbf{I}_3, \mathbf{I}_5, \ldots, \mathbf{I}_{2L+1}, \mathbf{c}) = \frac{1}{L}\sum_{i=1}^{L} BCELoss(\mathbf{I}_{2i+1}, \mathbf{c}) \tag{5.18}$$

Training

We use the Adam [44] optimizer with mini-batches for training. The training process is performed in the floating-point domain. Then, the optimized parameters, $\beta = (\beta^{(1)}, \beta^{(2)}, \ldots, \beta^{(L)})$, are applied to Equation 5.10 to determine the decoder's variable node function.

5.4.4 Learning FAID Design

Once the variable node function is determined through training, a FAID has to be designed based the outcome. Depending on the quantization, we define our finite alphabet in Equation 5.1 as follows:

$$\mathcal{M} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\} \tag{5.19}$$

According to Equations 5.2 and 5.3, a threshold set, $\mathcal{T}$, has to be defined for any FAID. Similar to Vasic's approach [40], we define a set of scalars, $\mathcal{A} = \{\alpha_1, \alpha_2, \ldots, \alpha_s\}$, so that:

$$\theta_1 = \alpha_1 L_1, \qquad \forall 2 \leq i \leq S, \quad \theta_i = \alpha_i L_{i-1} + (1-\alpha_i)L_i \tag{5.20}$$

where L_i is from the alphabet set defined in Equation 5.19. In fact, the set $\mathcal{A}$ controls the relative distance between two consecutive levels.

Given the quantization levels and the trained parameters, $\beta^{(t)}$s, we sweep the values for $LLR(y_i)$ and $\sum_{k\in\mathcal{C}_j} r_{k,j}^{(t)}$ in Equation 5.10. Then, it would be possible to generate

a lookup table to compute $l_{i,j}^{(t)}$ based on the values of $LLR(y_i)$, and $\sum_{k \in \mathcal{C}_j} r_{k,j}^{(t)}$ for each decoding iteration.

5.5 FPGA Micro-Architecture

We utilize our QC-LDPC decoder architecture described in Chapter 4 to design an efficient hardware for our finite alphabet decoder. Figure 5.4 illustrates the FAID architecture for a QC-LDPC code with c columns of circulants and any number of rows of circulants. The CNUs and VNUs perform the check and variable node computations respectively. Similar to the NGDBF decoder, our soft-decision FAID processes all columns of circulants in parallel while processing the rows of circulants sequentially. The RRFs are the rotation units that enable processing of p variable/check nodes at the same time. Moreover, $\text{RRF}^{(1)}$s store variable to check node while $\text{RRF}^{(2)}$s store check to variable node messages. However, unlike the NGDBF decoder, RRFs in which each output is a single-bit value, each soft-decision FAID's RRF output is of q bits, where q is the precision of internal computations. However, the internal

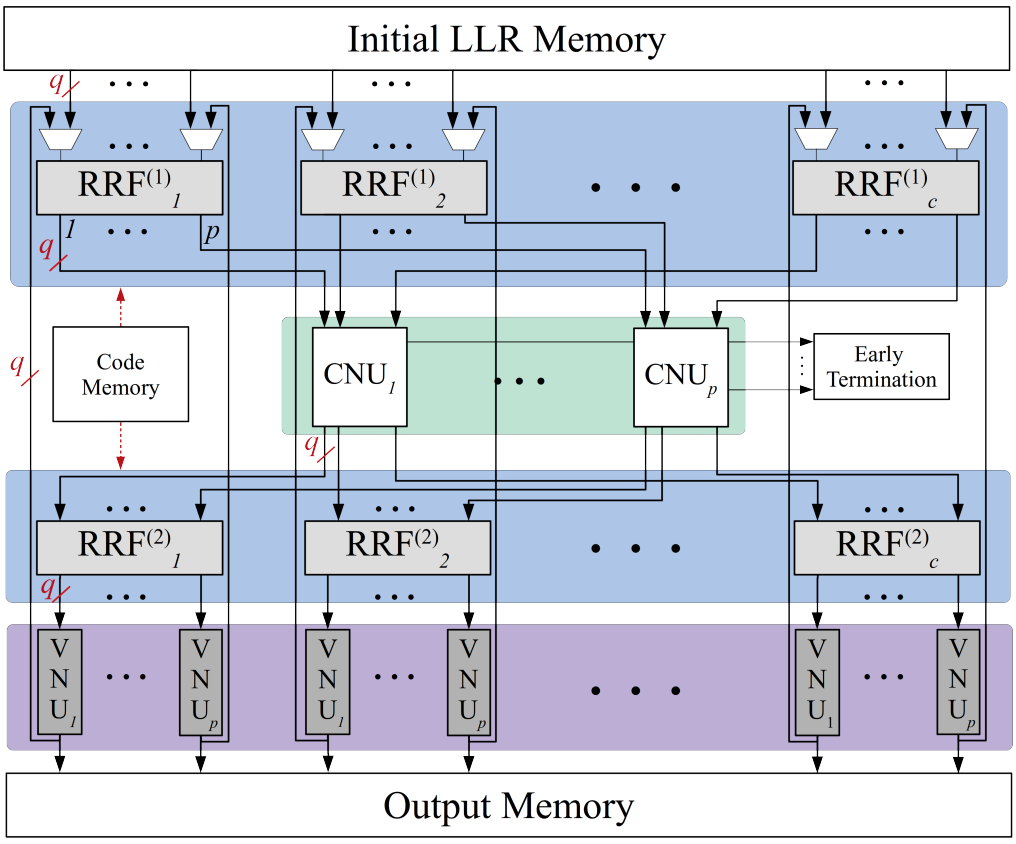


Figure 5.4: FAID architecture based on the decoder described in Chapter 4

computation precision can be higher than channel value precision (Read from flash memory through sensing). In fact, a higher than channel value precision on internal computations allows a wider range of values during the decoding process which results in a better code performance. The early termination block keeps monitoring the parity of the entire block so that the process can be terminated as soon as the codewords is decoded.

5.5.1 Check Node Unit (CNU)

A check node unit receives c variable node messages and has to generate c responses each of which calculated based on Equation 5.4.

Output i is comprised of a sign, s_i, and a magnitude, mag_i. mag_i can be computed through finding the first and second minimum magnitude of the entire inputs, min_1 and min_2 and using Equation 5.21 to compute the output.

$$mag_i = \begin{cases} min_1, & |in_i| \neq min_1, \\ min_2, & otherwise. \end{cases} \tag{5.21}$$

Similarly, s_i, can be computed as:

$$s_i = (sgn(in_1) \oplus sgn(in_2) \oplus \cdots \oplus sgn(in_c)) \oplus sgn(in_i) \tag{5.22}$$

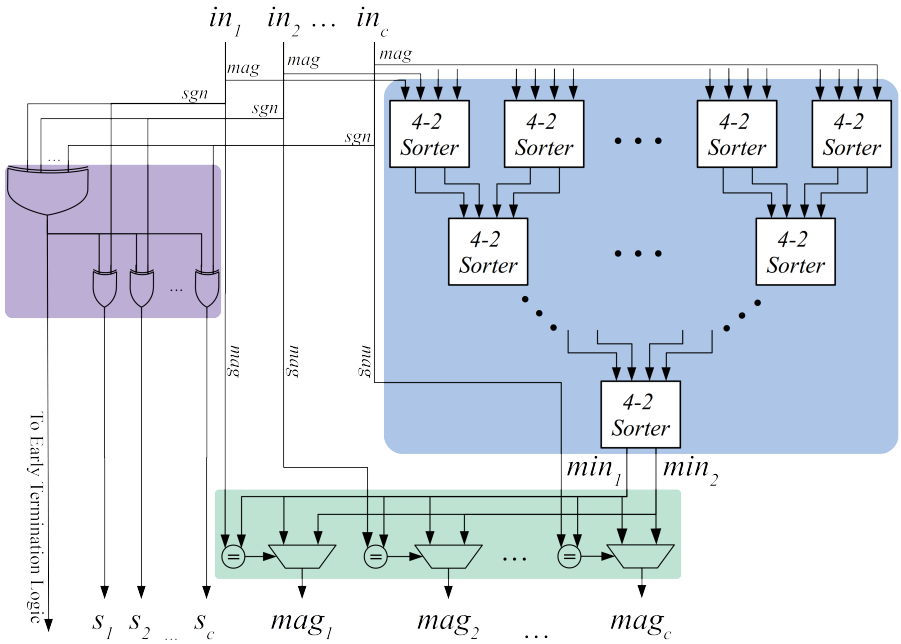


Figure 5.5: FAID check node unit architecture

where *sgn* is the standard *signum* function.

Figure 5.5 depicts the CNU architecture. a The first and second minimums are first found through a sort reduction tree. Then, the outputs are multiplexed depending on input values based on Equation 5.21. The *4-2 Sorter* components find the first and second minimum among 4 inputs and they are used as the basic block to build the sort tree for larger number of inputs. The XOR logic performs Equation 5.22 to calculate output signs.

5.5.2 VNU: A Two-Stage Lookup Table

The variable node units have to aggregate all the incoming check node messages *i.e.*, $m_{i,j}$s from CNU, and after processing the last row of circulants, they should perform the finite alphabet table lookup operation to generate to output message. In order to implement a FAID's variable node unit, most prior works including Vasic's FAID [40] suggest using a single lookup table. This approach is depicted in Figure 5.6a where the variable node uses channel value, y_i, the incoming messages $r_{i,1}, r_{i,2}, \ldots, r_{i,d_v}$, and current iteration, l, as the address to lookup the output value in a table. However, this approach is not scalable for higher d_v values. Moreover, the table size increases drastically as the decoder's internal computation precision, q, increases.

Depending on the function used to fill the lookup table, its size can be reduced by performing some parts of the function through logic rather than memory elements. For instance, as depicted in Figure 5.6b, the output is usually a function of the addition of all incoming messages rather than a function of the individual messages. Pre-computing the sum of incoming messages results in a smaller address space for the lookup table which, in turn, results in a smaller lookup table.

Based on the computation in Equation 5.3 and 5.10, our FAID lookup table function would be:

$$l_{i,j}^{(t)} = \mathcal{Q}\big(LLR(y_i) + \beta^{(t)} \sum_{k \in \mathcal{C}_j} r_{k,j}^{(t)}\big) \tag{5.23}$$

where $\mathcal{Q}$ is the quantization function based on the thresholds computed in Equa-

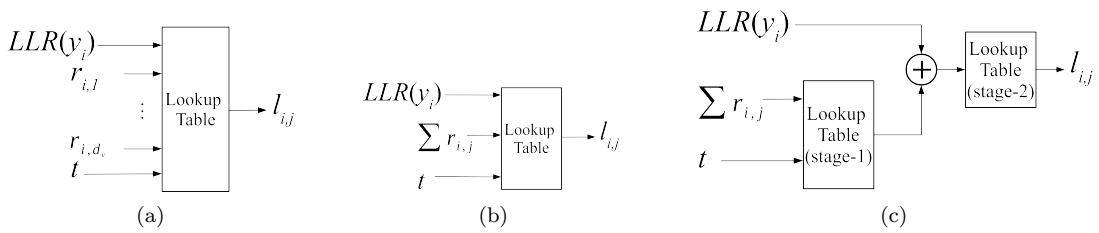


Figure 5.6: FAID VNU lookup table implementation (a) Base-line (b) Reduced size (c) Two-stage

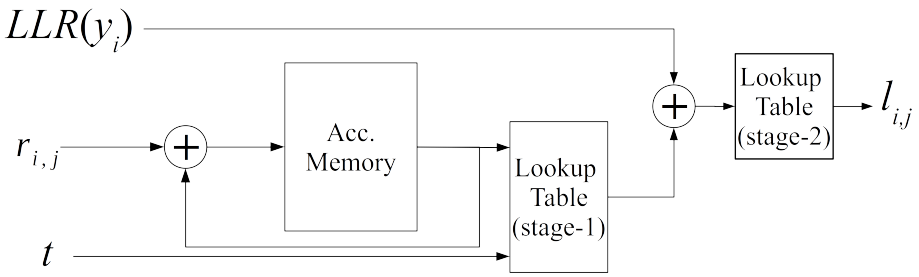


Figure 5.7: FAID check node unit architecture

tion 5.20. The computations can be decomposed into a two-stage quantization process as follows:

$$l_{i,j}^{(t)} = \mathcal{Q}\big(LLR(y_i) + \mathcal{P}\big(\beta^{(t)} \sum_{k \in \mathcal{C}_j} r_{k,j}^{(t)}\big)\big) \tag{5.24}$$

where $\mathcal{P}$ is the standard linear quantization function. The hardware implementation of the two stage FAID lookup table is illustrated in Figure 5.6c in which the $\mathcal{P}$ function is used to fill the stage-1 table while the stage-2 table is filled using the $\mathcal{Q}$ function. The full architecture of the VNU is depicted in Figure 5.7. The *Acc. Memory* is the storage used to compute the $\beta^{(t)} \sum_{k \in \mathcal{C}_j} r_{k,j}^{(t)}$ term.

5.6 Code Performance Evaluation Framework

Figure 5.8 illustrates the major steps in evaluating the correction capability of a decoder. At first, information bits are randomly generated. Then, the encoder is used to encode the information bits and generate a codeword. Depending on the channel model, a noise has to be added to the codeword then. Finally, the noisy codeword is sent to the decoder and its output is examined to compute the FER. The FER is then used to calculate the UBER. These steps are repeated until a pre-defined number of frame errors are found or a pre-defined upper-bound for UBER has reached. We set the pre-defined number of frame errors to 10 in our experiments. In other words, the evaluation is repeated until 10 frame errors are found. The main challenge towards evaluating an LDPC decoder is the runtime for experiments that achieve extremely low UBERs since it requires experimenting with a large number of codewords.

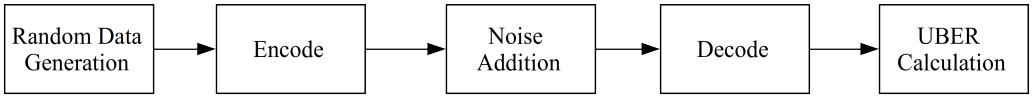


Figure 5.8: The major steps towards evaluating the correction capability of a decoder

However, these codewords are independent and the experiments can run in parallel. Therefore, a massively parallel platform can be leveraged to enable extremely-low-UBER experiments to finish in a reasonable time. In this thesis, we developed a software simulation as well as a hardware emulation framework to evaluate the code performance of our FAID.

5.6.1 Software Simulation

We performed all our software simulations on Scinet's Niagara supercomputer [45][46] which allocates up to 800 Intel Skylake processors at 2.4 GHz. We leveraged this computing power to perform extremely-low-UBER experiments in a reasonable time by designing a massively parallel simulation framework. In particular, we used Intel's AVX-512 SIMD vector instruction set to implement all steps, from random data generation to UBER calculation. Moreover, we used MPI to utilize all available processors for computing. In order to increase the likelihood of getting the required resources, we break long experiments into a set of short jobs each performing the experiment on a subset of codewords.

Figure 5.9 depicts our software simulation framework. Each job contains a set of processes each performing the evaluation iteratively. We used the Linux lrand48 library [47] as a baseline for our random generation steps, *i.e.*, data and noise generation. It is necessary for each process of each job to use an appropriate seed to ensure the independence of randomness in all processes. Given the number of jobs,

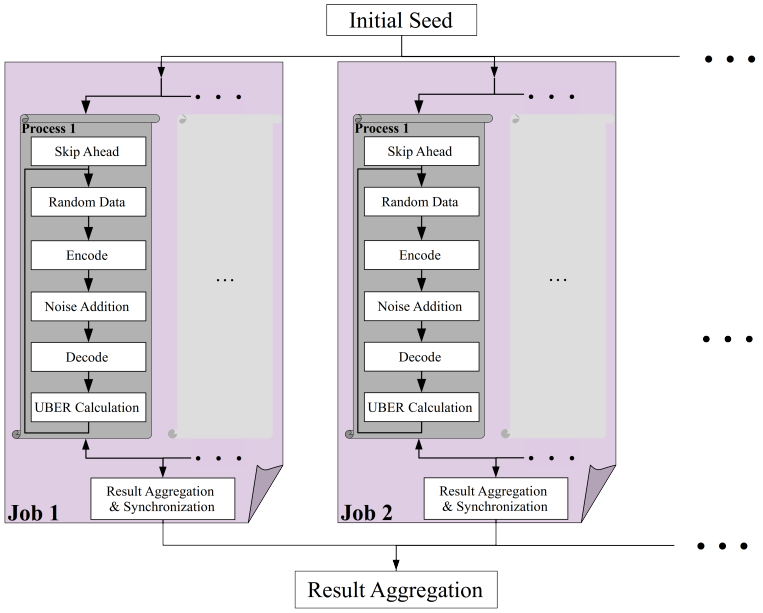


Figure 5.9: Software simulation framework for code performance evaluation

processes, and the total number of required random numbers of an experiment, we compute the total number of random generations required by each process. Then, starting from the initial seed, we skip a certain number of random numbers for each process to ensure that it is working on a unique sequence of random numbers. For that, we use Ramses's implementation of the lrand48 which enables *skip ahead* for the original lrand48 library in $\mathcal{O}(\log n)$ [48]. All processes of a job participate in computing synchronizing the overall UBER among themselves periodically. Once all jobs are finished their results are used to compute the overall UBER for the entire experiment.

5.6.2 Hardware Emulation

Figure 5.10 illustrates our hardware emulation framework for code performance evaluation. The host communicates to the driver on the board through the UART port. The driver has to initialize the random generators in the *Random Data Generator* and the *Noise Addition* units. Then, the experiment can be started by sending the random information bits to the encoder. Once a codeword is generated by the encoder, a random noise is added to the data. Finally, the decoder processes the codeword and the frame errors are computed in the *Experiment Status* module.

In order to keep track of the experiment, the random generators use the *State Queue*s to send a snapshot of their random generation state to the *Experiment Status* module. Then, this module creates a checkpoint of the experiment and the *Host* saves the checkpoint periodically. In the event of a failure during an emulation experiment, the host can use the *Driver* to load the latest saved checkpoint to the random gener-

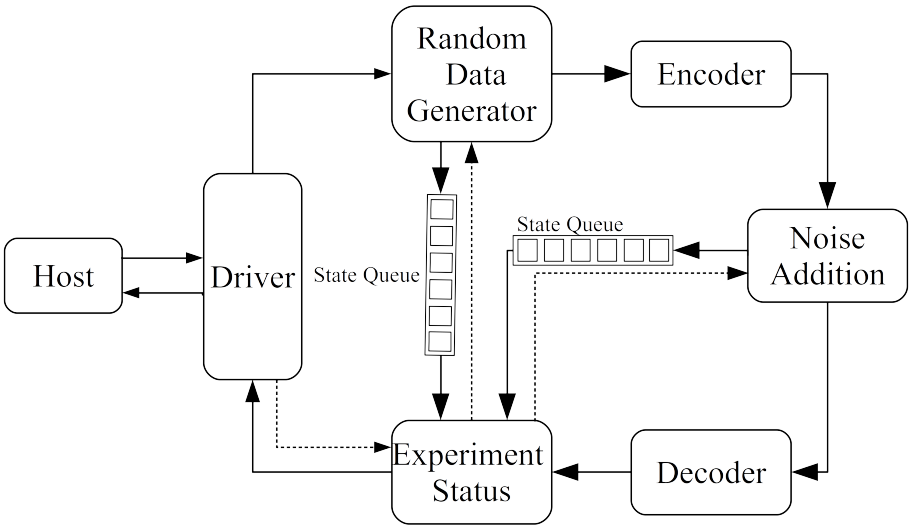


Figure 5.10: Hardware simulation framework for code performance evaluation

ators so that the failed experiment can be resumed without losing any experimental data.

Uniform Random Generation

As depicted in Figure 5.10, the hardware emulation framework involves random generation both for information bits, and noise addition. Therefore, it is necessary to design a high-quality uniform random generator. L'Ecuyer proposed a maximally-equidistributed combined linear feedback shift register (LFSR) for random number generation [49]. In this thesis, we implemented the hardware for L'Ecuyer's 32-bit random generation algorithm detailed in Algorithm 5.1. The period length for such random generator would be $2^{113} \approx 10^{34}$ which is well beyond the ultimate required random numbers in our experiments *i.e.*, 10^{18} random numbers to reach UBER=10^{-17}. The random generation algorithm can simply be implemented in hardware by a set of four 32-bit LFSRs with output produced by XORing the four LFSR values.

Listing 5.1: L'Ecuyer's 32-bit random generation algorithm [49]

```
1  unsigned int z1,z2,z3,z4;
2  unsigned int rnd_32(){
3      unsigned int b;
4      b = (((z1 << 6) ^ z1) >> 13);
5      z1 = (((z1 & 0xFFFFFFFE) << 18) ^ b);
6      b = (((z2 << 2) ^ z2) >> 27);
7      z2 = (((z2 & 0xFFFFFFF8) << 2) ^ b);
8      b = (((z3 << 13) ^ z3) >> 21);
9      z3 = (((z3 & 0xFFFFFFF0) << 7) ^ b);
10     b = (((z4 << 3) ^ z4) >> 12);
11     z4 = (((z4  & 0xFFFFFF80) << 13) ^ b);
12     return z1 ^ z2 ^ z3 ^ z4;
13 }
```

Parallel Random Generation

The main challenge with any LFSR-based random generator is the sequential nature of the algorithm which may become the main bottleneck in an emulation environment. Several parallel random generators have been proposed in the literature [50, 51, 52]. In this work, we propose a parallel implementation of Tausworthe random generation algorithm. In order for our parallel random generator to have the same distribution as the serial version, it is necessary for its outputs to be the same as the serial version.

An LFSR-based random number generator uses currently generated output as the seed to generate the next output. Let $\mathbf{S} = (s_0, s_1, s_2, \dots)$ be the output sequence

of a random generator with s_0 as the initial seed and $\mathcal{F}$ as the random generation function. Then,

$$\forall 0 \leq i, s_i \in \mathbf{S}, \qquad s_{i+1} = \mathcal{F}(s_i). \tag{5.25}$$

Consider a random generation function as $\mathcal{F}^P$:

$$\forall 0 \leq i, \quad s_i \in \mathbf{S}, \qquad s_{i+P} = \mathcal{F}^P(s_i). \tag{5.26}$$

Then, we have:

$$\mathcal{F}^P(s_0) = s_P, \quad \mathcal{F}^P(s_P) = s_{2P}, \quad \mathcal{F}^P(s_{2P}) = s_{3P}, \quad \ldots$$

and

$$\mathcal{F}^P(s_1) = s_{P+1}, \quad \mathcal{F}^P(s_{P+1}) = s_{2P+1}, \quad \ldots.$$

Therefore, a parallel implementation of the random generator function, $\mathcal{F}$, should have P instances of the $\mathcal{F}^P$ function. The initial seed for the i^{th} instance of $\mathcal{F}^P$ should be s_i. Figure 5.11 illustrates this implementation where the top values are the initial seeds while the bottom values show the sequence of output values for each random generator.

L'Ecuyer's random generator uses four state variables z_1, z_2, z_3, and z_4 to generate the next random value. Given the computations in Algorithm 5.1 as $\mathcal{F}$ and the parallelization factor, P, we compute the $\mathcal{F}^P$ function for each state variable by unrolling the P iterations of the $\mathcal{F}$ function and simplifying the equations. Then we use the approach depicted in Figure 5.11 to implement our parallel uniform random generator.

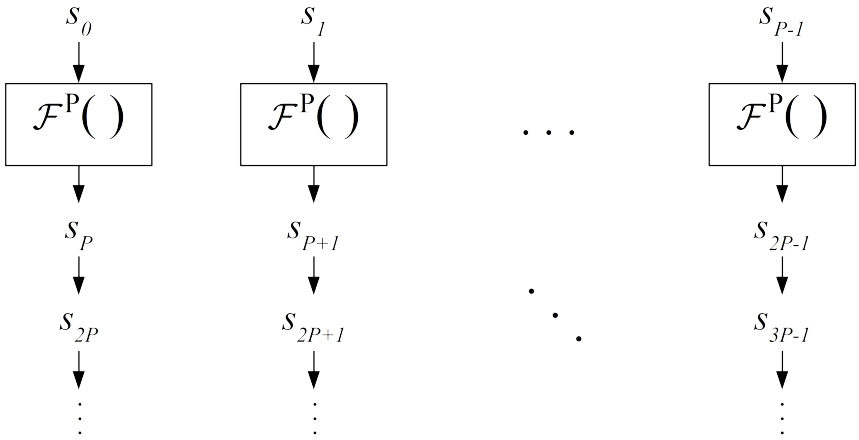


Figure 5.11: Parallel random generation logic

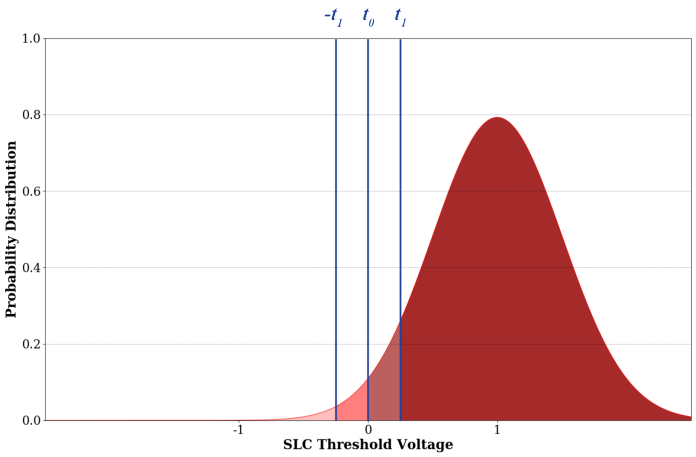


Figure 5.12: Probability distribution of the bits with value 0 in a SLC flash memory

Noise Soft Information

The *Noise Addition* module has to add a noise to the encoded codeword based on the assumed channel model. As depicted in Figure 5.1, we assume AWGN channel throughout our experiments. In this section, we explain our method for noisy soft information generation for codeword bits with value 0. However, the approach can also be used to generate noisy soft information for codeword bits with value 1. Figure 5.12 depicts the probability distribution of the bits with value 0 in a SLC flash memory with a 3-level sensing with threshold voltages, $-t_1$, t_0, and t_1. Let $\mathcal{Q}$ be the function that generates the input soft information for decoder based on the cell value, V. Then:

$$\mathcal{Q}(V) = \begin{cases} L_0, & V < -t_1, \\ L_1, & -t_1 \leq V < t_0, \\ L_2, & t_0 \leq V < t_1, \\ L_3, & t_1 \leq V. \end{cases} \tag{5.27}$$

Consequently, $\mathcal{Q}$'s output has 4 different values that can be represented by 2 bits. Given the variance of the voltage distribution, the probability of visiting each value as the soft information can be computed. Let $F(L_i)$ be the probability of visiting value L_i and $P(x)$ be the probability distribution function (PDF) of voltage. Then,

$$\begin{aligned} F(L_0) &= \int_{-\infty}^{-t_1} P(x)dx, & F(L_1) &= \int_{-t_1}^{t_0} P(x)dx, \\ F(L_2) &= \int_{t_0}^{t_1} P(x)dx, & F(L_3) &= \int_{t_1}^{\infty} P(x)dx. \end{aligned} \tag{5.28}$$

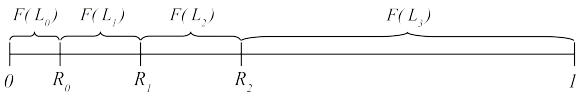


Figure 5.13: Quantized AWGN random generation from a uniform random variable

The value of $F(L_i)$s can be visualized as the area below the PDF curve as it is illustrated in Figure 5.12. In fact, they represent the probability of visiting each value as soft information for decoder input. Thus,

$$F(L_0) + F(L_1) + F(L_2) + F(L_3) = 1. \tag{5.29}$$

In order to generate a random value, n, from the Gaussian distribution with a given variance, we compute the $F(L_i)$ values. Then, as depicted in Figure 5.13, we divide the range $(0, 1)$ into 4 regions according to the computed $F(L_i)$s. Therefore,

$$\begin{aligned} R_0 &= F(L_0), & R_1 &= R_0 + F(L_1), \\ R_2 &= R_1 + F(L_2), & R_3 &= R_2 + F(L_3). \end{aligned} \tag{5.30}$$

Finally, we generate a uniform random number, r, in the range $(0, 1)$ and depending on the region it falls into, we generate the corresponding L_i value:

$$n = \begin{cases} L_0, & 0 < r < R_0, \\ L_1, & R_0 \leq r < R_1, \\ L_2, & R_1 \leq r < R_2, \\ L_3, & R_2 \leq r < 1. \end{cases} \tag{5.31}$$

These computations can be implemented in hardware using the uniform random

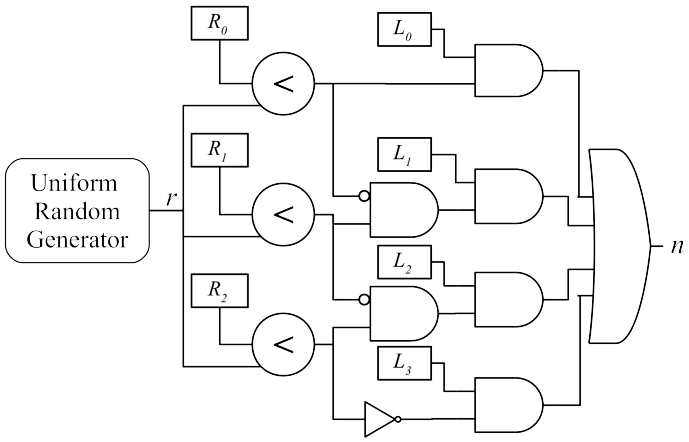


Figure 5.14: Noisy soft information generation logic

Table 5.2: The candidate QC-PaG codes for our experiments

Code	Rate	Circulant Size	# Rows of Circulants	# Columns of Circulants
C_1	0.88	179	6	53
C_2	0.89	193	5	49
C_3	0.87	211	6	46
C_4	0.86	257	5	38

generator discussed in Section 5.6.2, a set of comparators, and a selection logic as illustrated in Figure 5.14. The values of R_0, R_1, R_2 should be computed based on the voltage distribution. The design can simply be expanded for a 7-threshold voltage sensing as well. Moreover, our parallelization method discussed in Section 5.6.2 can be utilized to parallelize the noisy soft information generator as well.

5.7 Experimental Results

In order to study our proposed FAID, we used the quasi-cyclic partial geometry (QC-PaG) method based on prime fields introduced by Diao et al. [53] to construct a set of QC-LDPC codes with appropriate rate for flash memories. Then, we used our software simulation framework to study the relative performance of these codes and selected a subset of 4 codes for our hardware emulation experiments. Table 5.2 shows the specifications of the candidate QC-LDPC codes.

5.7.1 Experimental Setup

We implemented our learning framework in PyTorch [54] for the training phase and used both hardware emulation and software simulation frameworks to evaluate the code performance. For the software simulation experiments, we used Scinet's Niagara cluster that provides 20 nodes each with 40 Intel Skylake cores. For hardware emulation experiments, we used Verilator 4.014 to verify the functionality of our framework. Then, Vivado Design Suit 2020.2 is used to implement it on Xilinx Virtex-7 FPGA (xc7vx690tffg1761-3).

As discussed in Section 5.6, we estimate the UBER by repeatedly performing the decoding process until either 10 frame errors are observed, or a target upper-bound for the estimated UBER has reached. Therefore, assuming the target UBER 10^{-15} for consumer-level flash memories, we need to perform decoding on at least 10^{16} bits to ensure that our estimated UBER is below 10^{-15}. Using the 20 nodes provided by the Niagara cluster, our simulation framework is able to simulate up to the decoding experiment for up to 10^{13} bits in 1-2 hours and 10^{16} in about $1-2$ months. The

exact duration depends on the cluster's workload as the nodes are not dedicated to our experiments. On the other hand, our emulation framework can perform the experiment for up to 10^{13} bits in 20 minutes, 10^{15} bits in 30 hours, and 10^{16} bits in about 15 days.

5.7.2 Training Process

We constructed and trained our neural network decoder with 4 decoding iterations in Pytorch. In order to generate input data for our model, the message bits were produced using a uniform random bit generator and encoded to codewords. Using our simulation framework, we found the SNR range [4.5 dB - 5 dB] as the waterfall region of floating-point APP decoding for the codes of interest. Since our training is performed in the floating-point domain, our data set includes a set of 500 noisy codewords with SNR range [4.5 dB - 5 dB].

We used the Adam optimizer with mini-batch size 10, and learning rate 0.09, with 20 learning epochs. In order to speedup the training process, we used the PyTorch multiprocessing library to run a distributed data-parallel training. Figure 5.15 plots the loss evolution through out learning epochs for all 10 processes for our four candidate codes. Despite some fluctuations in early stages of the the training, our neural network is able to reduce the loss over the 20 learning epochs for all the codes.

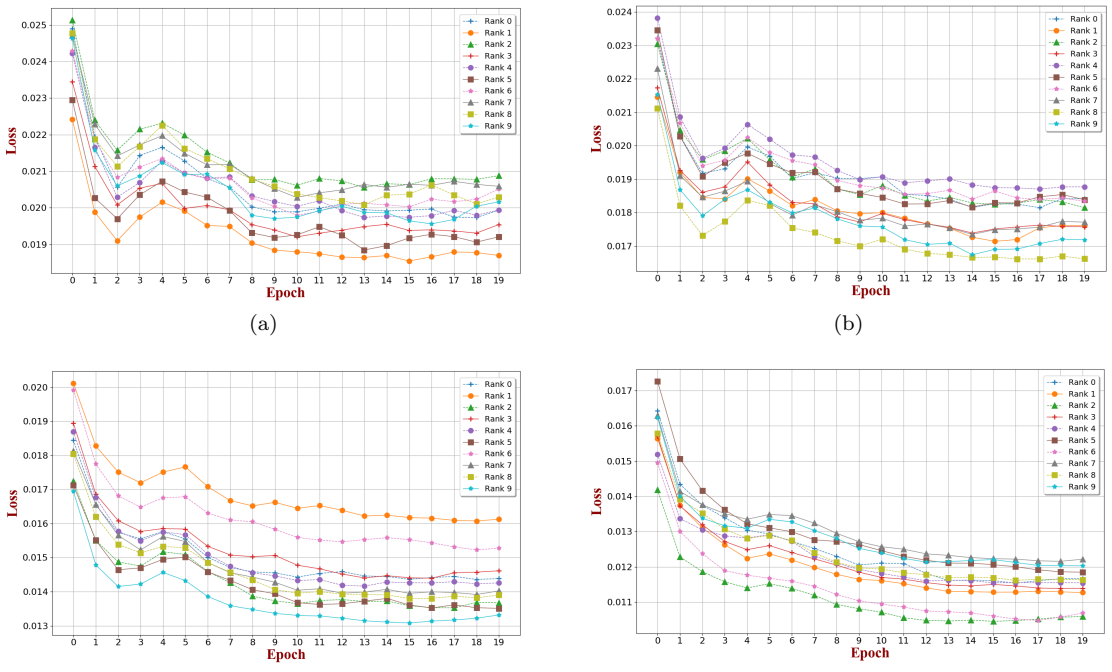


Figure 5.15: Evolution of loss over training epochs for (a)C_1 (b)C_2 (c)C_3 (d)C_4

Code	$\beta^{(1)}$	$\beta^{(2)}$	$\beta^{(3)}$	$\beta^{(4)}$
C_1	0.7048	0.6461	0.5633	0.5502
C_2	0.7555	0.6940	0.6033	0.5406
C_3	0.7465	0.6997	0.5919	0.5615
C_4	0.7928	0.7522	0.5969	0.5736

Table 5.3: The learned Weights

The outcome of the training process are the 4 weight parameters, $\beta^{(1)}$, $\beta^{(2)}$, $\beta^{(3)}$, $\beta^{(4)}$, for the 4 decoding iterations for each code. Table 5.3 reports the learned weight parameters for the four candidate codes. The learned β parameter shows a general declining trend over decoding iterations, *i.e.*, from $\beta^{(1)}$ to $\beta^{(4)}$. This essentially means the early iterations generally update the bit values more aggressively than the later iterations.

5.7.3 Code Performance

We used the outcome of our learning process *i.e.*, the learned weights, to evaluate correction capability of the learned decoder compared to the conventional APP algorithm. The full-precision experiments are performed on our software simulation platform while the quantized experiments were performed on our hardware emulation platform. We examined our software and hardware platforms to ensure the integrity of the results.

Floating-Point Performance

Figure 5.16 plots the UBER against SNR for original and learned APP in the floating-point domain. The dashed lines show the extrapolations of the curves for extremely-

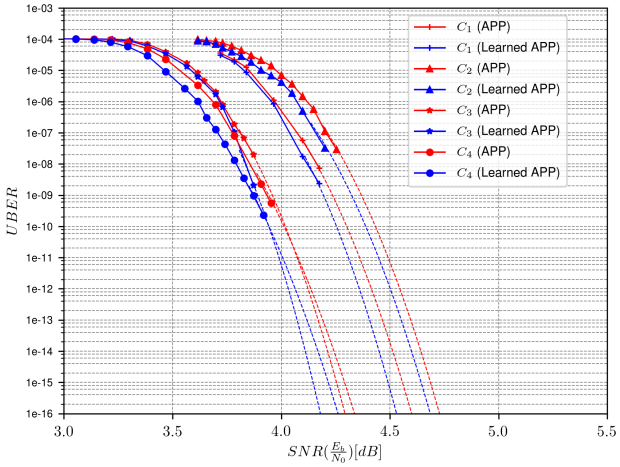


Figure 5.16: Floating-point APP Vs. Learned APP for C_1, C_2, C_3, and C_4

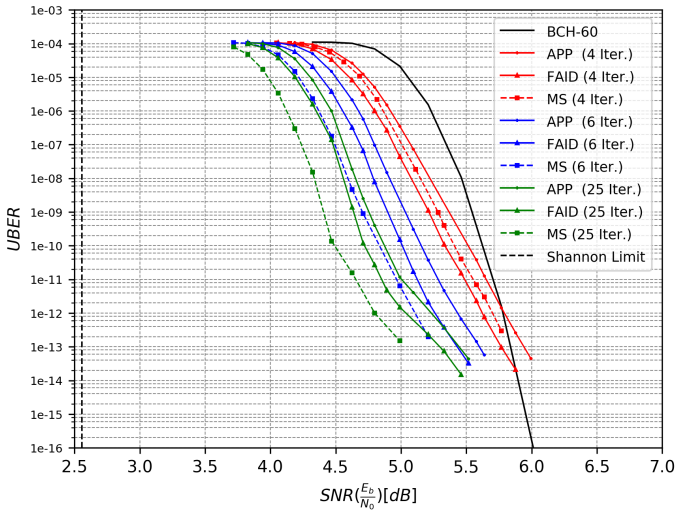


Figure 5.17: Learned 4-bit FAID Vs. APP Vs. MS in 7-level sensing SLC for C_1

low RBER experiments. The results for all candidate codes show that with 4 decoding iterations, the floating-point learned APP improves the noise threshold by 0.1 dB to 0.2 dB at target UBER=10^{-13}.

4-Bit FAID Vs. APP

The learned weights are used to construct a 4-bit FAID based on Equation 5.20. Figure 5.17 compares the code performance of the 4-bit FAID with a 4-bit APP and a 4-bit MS algorithm for C_1 for 7-level sensing for a SLC flash memory. The figure also shows the theoretical code performance of a BCH code with 1 KB code that is able to correct up to 60 errors (BCH-60). Although our learning process was performed for 4 iterations, we extended the maximum iterations to 6 and 25 iterations by setting a constant non-learned weight (similar to APP) for later decoding iterations. Figure 5.17 shows that even with learned weights for only 4 iterations, our FAID preserves the noise threshold advantage over APP for higher maximum decoding iterations as well. In addition, the results show that our FAID with 6 iterations is as good as the APP with 25 iterations in the extremely-low UBER region.

Figure 5.18 compares the code performance of our FAID to the conventional APP for all 4 candidate codes with 25 maximum decoding iterations. The results show that our FAID improves the noise threshold on all codes by 0.1-0.2 dB.

Impact on Decoding Iteration

Figure 5.19 plots the cumulative distribution function (CDF) for the required the number of iterations in 4-bit FAID and APP algorithms with 7-level sensing SLC for

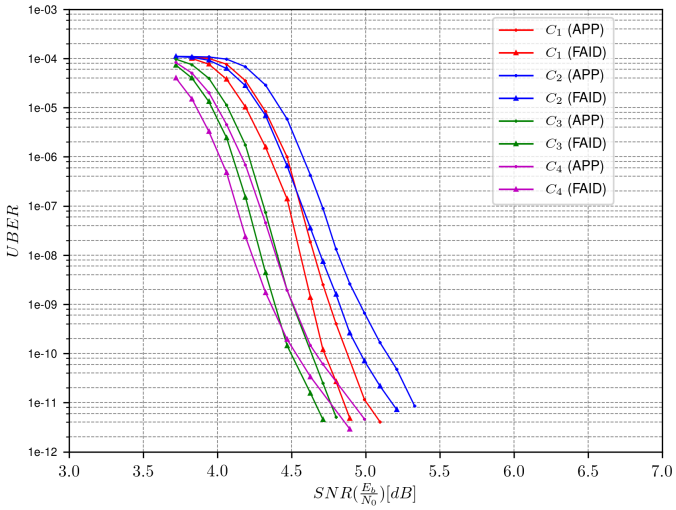


Figure 5.18: 4-bit FAID Vs. APP in 7-level sensing SLC for with 25 maximum iterations

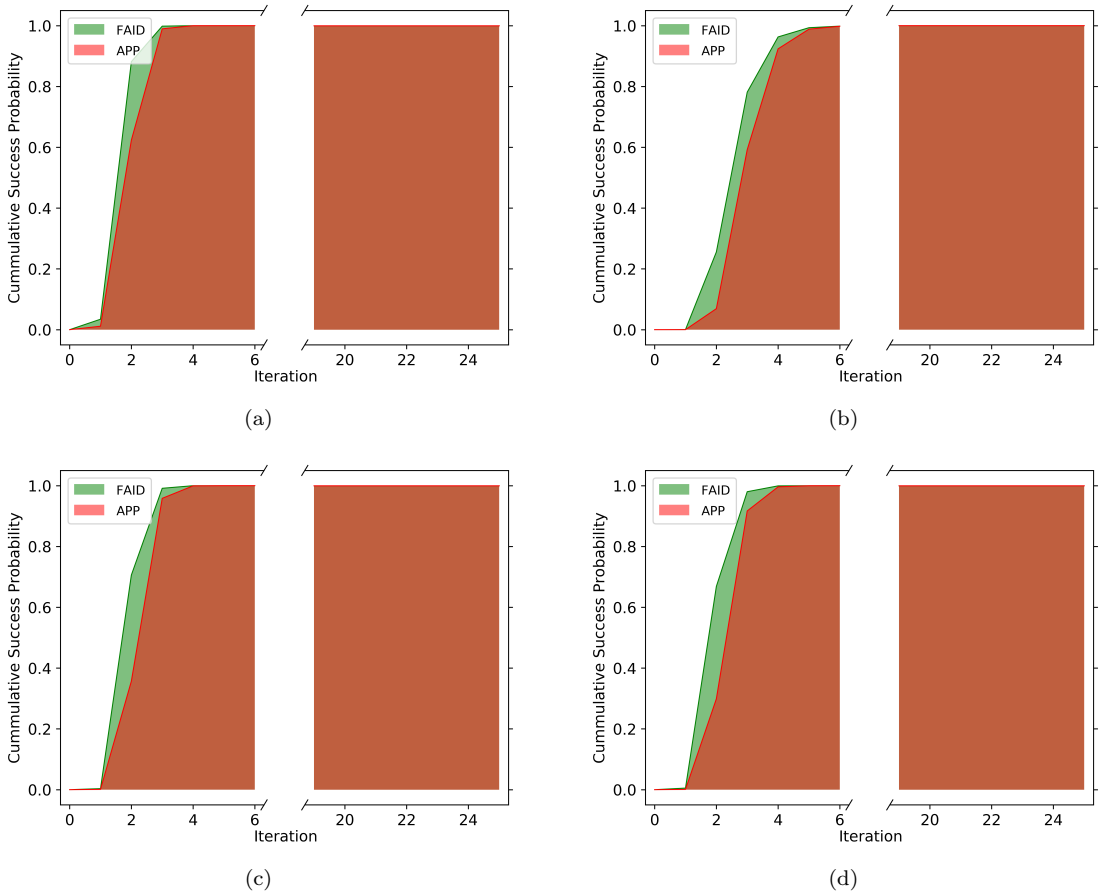


Figure 5.19: Decoding iterations for 4-Bit FAID and 4-bit APP for 7-level sensing SLC for (a) C_1, (b) C_2, (c) C_3, and (d) C_4

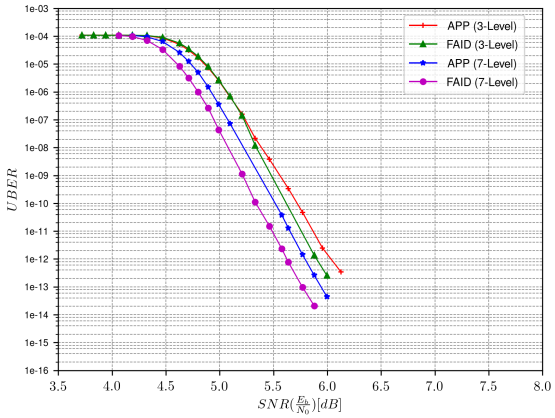


Figure 5.20: 3-level Vs. 7-level sensing for C_1

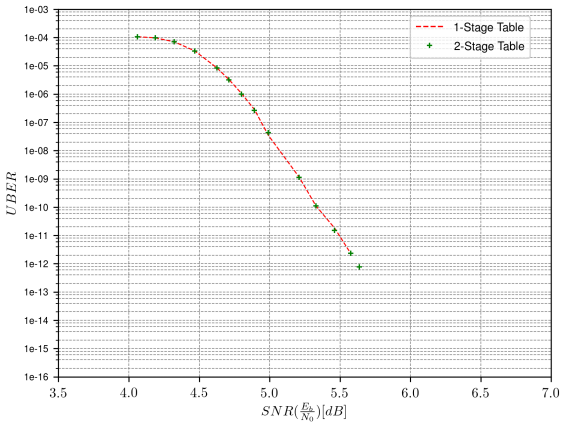


Figure 5.21: 2-stage Vs. 1-stage table FAID

4 candidate codes. The charts show that our FAID has $20\% - 35\%$ higher chance of successfully decoding the codeword with one iteration less than APP. Moreover, FAID has about $4\% - 13\%$ lower average number of decoding iterations which results in $4\% - 13\%$ higher throughput than the APP decoder.

3-Level Vs. 7-level Sensing

Figure 5.20 compares the code performance of C_1 for 3-level and 7-level sensing SLC flash memory with 4 decoding iterations. The plot shows that FAID's noise threshold advantage increases for higher sensing levels. Moreover, a 3-level sensing FAID almost halves the noise threshold gap between 3-level and 7-level sensing APP.

1-Stage Vs. 2-Stage Table FAID

As discussed in Section 5.5.2, our 2-stage lookup table approach has significantly lower hardware overhead compared to a 1-stage lookup table approach. Moreover, it is more scalable for codes with higher variable node degree, d_v. However, since it involves a 2-stage quantization, its output might slightly deviate from the 1-stage lookup table. Therefore, it is necessary to ensure that the 2-stage FAID does not have significant code performance degradation. Figure 5.21 compares the code performances of a 1-stage and a 2-stage lookup table FAID. The plot shows that the 2-stage lookup table FAID shows no performance degradation compared to the 1-stage lookup table.

5.7.4 Hardware-Code Performance

Table 5.4 compares the hardware-code performance of our proposed 4-bit FAID with the 4-bit APP decoder for C_1. The results show that compared to the APP decoder, our 2-stage lookup table FAID improves the noise threshold by 0.16 dB with same

Decoder	**APP**	**FAID (2-stage)**	**FAID (1-stage)**
LUT Count	89612	109189	156176
Register	103464	100443	96539
Throughput ($\frac{GBps}{Iteration}$)	1.86	1.73	1.45
Efficiency ($\frac{MBps/KLUT}{Iteration}$)	21.25	16.26	9.54
Average Iterations	2.38	2.09	
Overall Throughput ($GBps$)	0.81	0.82	0.66
Noise Thr. @10^{-13}	4.00E-3 (5.46 dB)	4.60E-3 (5.3 dB)	

Table 5.4: Hardware-code performance for C_1

overall throughput and at the cost of about 20% more hardware resources. However, since our FAID improves the average number of iterations by almost 12%, it is fair to say that the 0.16 dB noise threshold improvement is achieved at the cost of about 12% lower hardware efficiency.

The results also show that our 2-stage table FAID reduces LUT utilization by 30% compared to a 1-stage lookup table approach for C_1 with $d_v = 6$. Moreover, the smaller tables of the 2-stage FAID achieves 20% higher throughput. With these improvements the 2-stage table FAID is 77% more efficient than the 1-stage table FAID with no noticeable code performance degradation based on Figure 5.21.

Table 5.5 reports the hardware-code performance improvement of FAID compared to APP for the 4 candidate codes. At the cost of about 20% area overhead, our FAID achieves up to 15% higher throughput by reducing the average decoding iterations by $12\% - 15\%$ and improves the noise threshold by $10\% - 17\%$. All in all, it could be concluded that compared to APP, our FAID has $10\% - 17\%$ better noise threshold at the cost of $5\% - 17\%$ less efficient hardware.

	C_1	C_2	C_3	C_4
Area	-21.85%	-22.43%	-21.58%	-20.91%
Average Iterations	12.24%	12.18%	14.18%	15.77%
Throughput	6.27%	0.85%	9.52%	15.55%
Efficiency	-12.79%	-17.63%	-9.92%	-4.43%
Noise Threshold @10^{-13}	15% (0.16 dB)	17% (0.16 dB)	17.65% (0.19 dB)	9.30% (0.1 dB)

Table 5.5: FAID Hardware-code performance improvement over APP

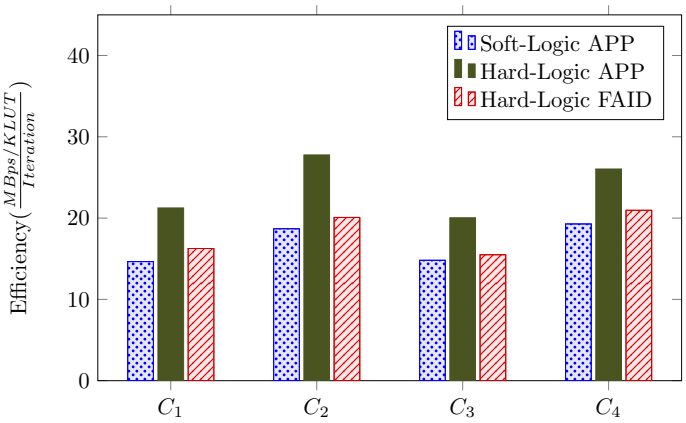


Figure 5.22: The impact of leveraging FPGAs hard-logic on efficiency

5.7.5 FPGA Hard-Mux Impact

In order to study the impact of leveraging hard-logic in our RRF-based decoder (discussed in Section 4.3.3) *i.e.*, leveraging FPGA's hard multiplexers to implement the RRF, we also implemented an APP-based decoder with similar architectural features but entirely on FPGA's soft-logic. The soft-logic decoder is inspired by the prior decoders discussed in [18] and [21]. Figure 5.22 plots the efficiency for the soft-logic APP, hard-logic APP, and hard-logic FAID decoders for all 4 candidate codes. For a fair comparison, we swept the parallelization factor for all decoders and reported the most efficient instance for them. The results show that the leveraging FPGA's hard-logic results in a $35\% - 45\%$ more efficient hardware. Moreover, despite being less efficient than hard-logic APP, our hard-logic FAID implementation is $4\% - 11\%$ more efficient than soft-logic APP decoders.

5.8 Summary

The inherent limitations of error correction in flash memories results in a finite decoder design space. We proposed an end-to-end solution to improve the correction capability of conventional decoders while respecting the hardware efficiency. In particular, we used machine learning techniques to find the best decoding algorithm for a given code. The learned decoder is then used to design a finite alphabet decoder. In order to support higher variable node degrees, we suggest a 2-stage lookup table FAID which significantly reduces the hardware cost without any code performance degradation. The majority of prior efforts on FPGA implementation of FAID focus on small block codes (≈ 1000 bits) with variable node degree 3 [55, 56, 57]. In general, these codes have simpler Tanner graphs which may result in a more efficient decoder.

However, their code performance makes them impractical to be used in flash memories where high-rate strong codes are necessary. To the best of our knowledge, this is the first academic research on exploring machine learning techniques and finite alphabet decoders on large block codes. Our experimental results show that compared to the APP decoder, our 4-bit FAID improves the noise threshold by $10\% - 17\%$ at the cost of $5\% - 17\%$ less efficient hardware. Moreover, our scalable 2-stage table FAID is 77% more efficient than the 1-stage table FAID with no code performance degradation.

Chapter 6

A Reconfigurable Architecture for Erasure Coding

Data replication has been one of the most popular approaches to deliver reliability in data centres due to its simplicity and low reconstruction cost. However, it suffers from high storage overhead. In contrast, erasure coding has emerged as an important alternative due to its low storage overhead.

Example 6.1 *Assuming the cost of raw enterprise storage is* $\$0.1/GB$*, a peta-byte data center storage would cost* $\$100k$ *for the main storage, and another extra* $\$100k$ *for each replica. Consequently, a triplicated storage would carry* $\$200k$ *extra capital cost. On the other hand, an erasure coded solution could deliver the same reliability with only* 30% *extra storage, leading to only* $\$30k$ *extra capital cost.*

The main drawback of erasure coding is the computational overhead of encoding and decoding algorithms. Therefore, there have been significant interests in the research and commercial communities to accelerate erasure coding [58] [59] [60]. However, such efforts have primarily been investigated on general-purpose CPUs. On the other hand, hardware acceleration has only been widely reported on its cousins in the context of error correcting code ECC [61] [62], whose computation requirements are not exactly the same.

In this chapter, we propose an FPGA implementation of erasure coding for data storage applications. In particular, we make several contributions: First, with probabilistic analysis, we show that more than 90% of failure cases are single block failures, and we could set up separate performance targets, and therefore separate resource allocations, for common and general cases. Second, we show that the common-case computation can be significantly reduced by pre-computation, and further reduced by exploiting the inherent structure of the decoding matrix. As a result, we are able

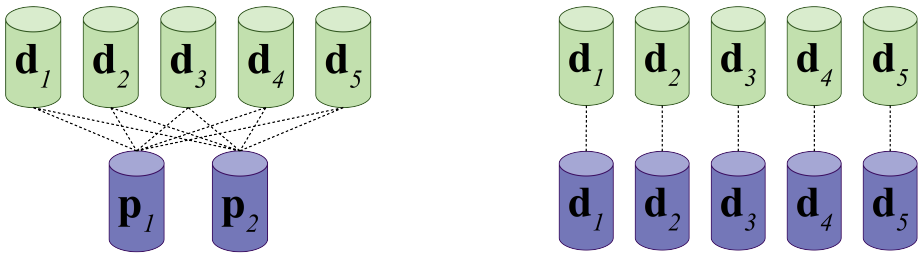


Figure 6.1: Example 6.2: (a) Erasure coding Vs. (b) Replication

to report an efficient, quantitative result on the FPGA acceleration of erasure coding. We show that even for a pessimistic value for the disk failure probability, our design outperforms other existing designs. As an added bonus, our parametrized design facilitates the design space exploration for erasure code designers.

We first explain the basic concepts of erasure encoding and decoding algorithms, as well as the reliability metric. Then, we point out the major related works in this area. Finally, we describe the details of our proposed architecture for the erasure encoder and decoder and evaluate them.

6.1 Erasure Code

A Reed-Solomon (RS) erasure code is denoted by RS(n,k). It divides data into k blocks and utilizes them to generate $m = n - k$ parity blocks using RS coding. In the case of block failure, any k blocks (either parity or data block) can be used to reconstruct the failed blocks.

Compared to replication, erasure coding provides higher reliability with significantly lower storage overhead:

Example 6.2 *Figure 6.1a illustrates a RS(*7,5*) erasure coded storage while Figure 6.1b depicts a replicated storage. In the erasure coded storage,* p_1 *and* p_2 *store parities generated from all data blocks. If any* 2 *out of the* 7 *blocks, fail, the surviving data and parity blocks can reconstruct them. Therefore, the system can tolerate any two block failure. On the other hand, in replication, each replica of a disk does not contain any data from other disks. Therefore, it would not be possible to reconstruct a block if both instances of that block fail. Moreover, the erasure coded storage has* 40% *while the replicated storage has* 100% *storage overhead.*

Although erasure coding can be more reliable with lower storage overhead, it has not yet been widely replaced the replication due to its encoding and decoding latency.

6.1.1 Encoding

In RS erasure encoding, the data blocks need to be multiplied by a matrix of encoding coefficients to generate the parity blocks:

$$\mathbf{G}_{m\times k}\times\mathbf{D}_{k\times B} = \begin{bmatrix}\mathbf{g}_1\\ \vdots\\ \mathbf{g}_m\end{bmatrix}\times\begin{bmatrix}\mathbf{d}_1\\ \mathbf{d}_2\\ \vdots\\ \mathbf{d}_k\end{bmatrix} = \begin{bmatrix}g_{1,1} & g_{1,2} & \cdots & g_{1,k}\\ \vdots & \vdots & \ddots & \vdots\\ g_{m,1} & g_{m,2} & \cdots & g_{m,k}\end{bmatrix}\times\begin{bmatrix}d_{1,1} & d_{1,2} & \cdots & d_{1,B}\\ d_{2,1} & d_{2,2} & \cdots & d_{2,B}\\ \vdots & \vdots & \ddots & \vdots\\ d_{k,1} & d_{k,2} & \cdots & d_{k,B}\end{bmatrix} = \begin{bmatrix}\mathbf{p}_1\\ \mathbf{p}_2\\ \vdots\\ \mathbf{p}_k\end{bmatrix} = \mathbf{P}_{m\times B} \tag{6.1}$$

where m is the number of parity blocks, k is the number of data blocks, and B is the block size. $\mathbf{G}$ is the encoding coefficient matrix, $\mathbf{D}$ is the data blocks matrix with k rows where each row is a data block, $\mathbf{d}$, and $\mathbf{P}$ is the parity matrix with m rows where each row is a parity block,$\mathbf{p}$.

Based on Equation 6.1, the erasure encoding is a matrix multiplication, where each row of the encoding coefficients matrix ($\mathbf{G}$) contributes in generation of the same row in the parity matrix ($\mathbf{P}$). In order to prevent any overflow, all computations are performed over the Galois field (GF) (2^w) in which w is referred to as the word size. The encoding coefficients matrix, $\mathbf{G}$, should be full rank, *i.e.*, all the rows have to be linearly independent. A commonly used matrix is the Vandermonde [63] matrix that is proved to be full rank under the following conditions [64]:

$$\mathbf{V} = \begin{bmatrix}1 & \alpha_1 & \alpha_1^2 & \cdots & \alpha_1^k\\ 1 & \alpha_2 & \alpha_2^2 & \cdots & \alpha_2^k\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 1 & \alpha_m & \alpha_m^2 & \cdots & \alpha_m^k\end{bmatrix}, \quad \forall 0 < i \leq m \quad \alpha_i \neq 0, \quad \forall i \neq j, \quad \alpha_i \neq \alpha_j \tag{6.2}$$

6.1.2 Decoding

RS decoding involves two steps; syndrome detection to identify the presence and location of error, and reconstruction to restore the erroneous bits/blocks. In erasure coding, the erased blocks, are identified through separate mechanisms. For instance, if erasure coded blocks are stored in separate disks, the unhealthy disk can easily be identified. Therefore, RS erasure decoding refers only to the reconstruction phase of the RS decoding.

In decoding an RS(n,k) erasure code, there should be at least k available blocks to recover the failed blocks. In that case, a reconstruction matrix, $\mathbf{R}$, should be formed by replacing the failed data blocks by the parity blocks in $\mathbf{D}$. Hence, in case $\mathbf{d}_i$ is

failed and $\mathbf{p}_j$ is available for reconstruction, we have:

$$\mathbf{R} = \begin{bmatrix} \mathbf{d}_1 \\ \vdots \\ \mathbf{d}_{i-1} \\ \mathbf{p}_j \\ \mathbf{d}_{i+1} \\ \vdots \\ \mathbf{d}_k \end{bmatrix} \tag{6.3}$$

Then, a matrix $\mathbf{F}$ is formed from the *Identity* matrix of size k. The rows of the *Identity* matrix corresponding to the missing data blocks are replaced by the rows of $\mathbf{G}$ that corresponds to the replacing parity blocks. Finally, the reconstruction is done as follows:

$$\mathbf{F}^{-1}{}_{k\times k} \times \mathbf{R}_k = \mathbf{D}_k \tag{6.4}$$

We refer to $\mathbf{F}^{-1}$ and its elements as the decoding matrix and the decoding coefficients respectively.

Example 6.3 *Equation 6.5 shows the encoding for RS(7,5) erasure code.*

$$\begin{bmatrix} g_{1,1} & g_{1,2} & g_{1,3} & g_{1,4} & g_{1,5} \\ g_{2,1} & g_{2,2} & g_{2,3} & g_{2,4} & g_{2,5} \end{bmatrix} \times \begin{bmatrix} \mathbf{d}_1 \\ \mathbf{d}_2 \\ \mathbf{d}_3 \\ \mathbf{d}_4 \\ \mathbf{d}_5 \end{bmatrix} = \begin{bmatrix} \mathbf{p}_1 \\ \mathbf{p}_2 \end{bmatrix} \tag{6.5}$$

Equation 6.6 is the decoding operation for the case that $\mathbf{d}_2$*, and* $\mathbf{d}_4$ *are failed* $\mathbf{p}_1$ *and* $\mathbf{p}_2$ *replace them respectively.*

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ & & \mathbf{g}_1 & & \\ 0 & 0 & 1 & 0 & 0 \\ & & \mathbf{g}_2 & & \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \times \begin{bmatrix} \mathbf{d}_1 \\ \mathbf{p}_1 \\ \mathbf{d}_3 \\ \mathbf{p}_2 \\ \mathbf{d}_5 \end{bmatrix} = \begin{bmatrix} \mathbf{d}_1 \\ \mathbf{d}_2 \\ \mathbf{d}_3 \\ \mathbf{d}_4 \\ \mathbf{d}_5 \end{bmatrix} \tag{6.6}$$

6.2 Reliability Metric

An RS(n, k) erasure code can tolerate up to $m = n - k$ failed blocks. Therefore, assuming the block failures to be independent, the probability of system failure is equal to the probability that more than m blocks fail simultaneously:

$$P(fail) = \sum_{i=m+1}^{n} \binom{n}{i} q^i (1-q)^{n-i} \tag{6.7}$$

where q is the block failure probability. Consequently, reliability is calculated as follows:

$$\mathcal{R} = 1 - P(fail) = 1 - \sum_{i=m+1}^{n} \binom{n}{i} q^i (1-q)^{n-i} = \sum_{i=0}^{m} \binom{n}{i} q^i (1-q)^{n-i} \tag{6.8}$$

A block failure probability, q, is equal to the probability that the disk is unavailable:

$$q = 1 - Availability \tag{6.9}$$

Therefore:

$$q = 1 - \frac{MTTF}{MTTF + MTTR} = \frac{MTTR}{MTTF + MTTR} \approx \frac{MTTR}{MTTF} \qquad (MTTR \ll MTTF) \tag{6.10}$$

where $MTTR$ is the mean time to repair, and $MTTF$ is the mean time to failure.

6.3 Prior Works

The concept of reliable data storage emerged in late 80s when Patterson [65] introduced RAID. He introduced six RAID levels and later extended it to seven in 1993 [66]. Reliability received more attention in the context of data centers such as those that powered Internet services of Facebook and Google. For example, the Hadoop distributed file system (HDFS) [67], uses triplication scheme to protect the data.

Facebook introduced an RS(14, 10) code augmented with extra parities to reduce network overhead of erasure coding [68]. In the case of a failure, 10 blocks are moved (over the network) to a single location to reconstruct the failed blocks. To reduce this overhead, an extra parity is calculated for each 5 data blocks so that the failures within those blocks can be recovered by only moving 5 blocks to a single location. Similarly, Microsoft proposed locally-repairable code (LRC) for the Azure storage system [69] that targets network overhead reduction.

The efficient implementation of the RS erasure coding algorithm is mostly addressed in the software community. A forward error correction (FEC) library [70] was introduced by Rizzo in 1998 and achieved the best efficiency of its time: a throughput of 100Mb/s. As a further enhancement of the FEC library, zFEC is made open source and found usage in many distributed storage systems such as SheepDog [71]. In particular, zFEC implements GF multiplications using pre-computed lookup tables, and therefore is limited to small GF. Jerasure [72], employed by CEPH [60], is the state-of-the-art erasure coding package that provides a variety of functionality and GF sizes

Table 6.1: The main ideas and achievements of the related works in erasure coding

Name	Year	Throughput	Comments
FEC [70]	1998	$\sim 100\ Mb/s$	First implementation. Putting erasure coding in data storage perspective.
zFEC [76]	2008	$\sim 1\ GB/s$	Improved from FEC. Using multiplication table.
Jerasure [72]	2007	$\sim 1.8\ GB/s$	Single-threaded, GF size: 8, 16, 32.
Jerasure-SIMD [58]	2014	$\sim 10\ GB/s$	Utilizing SIMD shuffling instructions and small multiplication tables to perform multiple multiplication in a single instruction.

up to 32. Its enhanced version, referred to as Jerasure-SIMD [58], accomplishes its performance by using the GF-complete package [73], which leverages the shuffling instruction of modern single instruction multiple data (SIMD) instruction set. This is similar to the ISA-L [59] library provided by Intel.

There are limited hardware implementations of erasure coding. Mellanox introduced a commercial ASIC design of erasure coding on its network adapters to offload the encoding/decoding tasks to the hardware [74]. However, little detail is known for such a commercial ASIC design. Ruan has implemented an erasure codec on FPGA using VIVADO high-level synthesis (HLS) [75]. Independently, Ruan also adopted one of our reported techniques, namely, pre-computation. However, the results show that our combined techniques significantly out-perform that design.

Table 6.1 summarizes the major performance milestones accomplished by the above efforts. This work complements these efforts by providing an alternative implementation path using FPGAs, which requires different considerations. For example, GF multiplication can be directly implemented using a combinational circuit rather than lookup tables using memory. And in general, we could have more freedom in selecting micro-architectures than being limited by the SIMD instruction set. Our goal is therefore to find an efficient FPGA implementation such that the host CPUs are released for the compute, rather than the storage function.

6.4 Design Decisions

This section drives three major design decisions for the proposed architecture. It is shown through analysis that single block failures are significantly more probable than multiple block failures. Then, the concept of pre-computation and its efficiency is described. Finally, the inherent structure of the decoding matrix is leveraged to reduce the time complexity of computations.

6.4.1 A Case For Common-Case

In order to identify the potentials to accelerate erasure decoding, we analyze the probability of failures with different numbers of failed blocks. We enumerate all possible failure cases and calculate their probabilities assuming that block failures are independent. Our calculations do not need to enumerate the cases where only parity blocks fail. The reason is that if all data blocks are available and some parity blocks fail, there would be no need to decode and the failed parity blocks can be constructed by encoding the data blocks (Equation 6.1). Moreover, if some parity blocks and some data blocks fail simultaneously, the failed data blocks are first reconstructed using the remaining available parity blocks. Then, data blocks can be used to generate the failed parity blocks.

The goal is to calculate the probability of *recoverable* failures. A failure case with α failed data blocks is *recoverable* if at least α parity blocks are available. We use the following notations:

- $P(R)$: The probability of having a failure that is recoverable (recoverable failure).
- $P_\alpha(RF)$: The probability of having a recoverable failure with exactly α failed data blocks.
- $P_\alpha(FD)$: The probability of having exactly α failed data blocks.
- $P_\alpha(AP)$: The probability of having exactly α available parity blocks.

Let q be the probability of a block to fail. We have:

$$P_\alpha(FD) = \binom{k}{\alpha} \cdot q^\alpha \cdot (1-q)^{k-\alpha} \tag{6.11}$$

and:

$$P_\alpha(AP) = \binom{m}{\alpha} \cdot q^{m-\alpha} \cdot (1-q)^\alpha \tag{6.12}$$

Consequently, the probability of a recoverable failure with α failures is the probability of having α failed data blocks multiplied by the probability of having at least α available parity blocks. Therefore:

$$P_\alpha(RF) = P_\alpha(FD) \times \sum_{i=\alpha}^{m} P_i(AP) \tag{6.13}$$

Example 6.4 *For an RS(14, 10) code, the probability of a recoverable failure with 3 failed data blocks is:*

$$P_3(RF) = P_3(FD) \times [P_3(AP) + P_4(AP)]$$

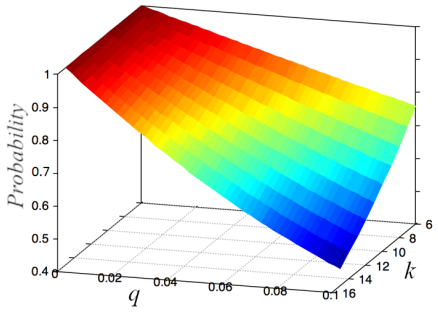


Figure 6.2: Probability of a recoverable failure with single failed block for $m = 4$ codes

It is the probability of having 3 failed data blocks multiplied by the probability of having 3 or 4 available parity blocks.

Since the system can tolerate up to m failures, the total probability for having a recoverable failure is:

$$P(R) = \sum_{i=1}^{m} P_i(RF) \tag{6.14}$$

Figure 6.2 illustrates the probability of having a recoverable failure with single failed data block. It illustrates the probability for different values of k and q with $m = 4$. This graph confirms that even for a pessimistic value of $q = 0.1$, at least 45% of recoverable failures are single block failures.

In 2017, Zhang *et al.* performed an analysis of reliability in erasure-coded data centers [77]. The study showed that about 99.5% of the independent failures and even about 99.3% of the correlated failures are related to single block failures. Based on our probabilistic analysis and Zhang's analytical study, we conclude that single block failures have strongly higher probability compared to other failure cases. Therefore, we refer to the single block failures as the *common-case* failures. Since the common-case failures strongly dominate other failures, it would be reasonable to allocate more FPGA resources to the common-case erasure coding while reducing the performance target for other failures (referred to as general-case failures). For instance, if we have a processing unit that can process the common-case at $5GB/s$ and a processing unit that can process the general-case at $1GB/s$, assuming 95% of failures would be common-case failures, the overall throughput would be $4.8GB/s$.

Design Decision 6.1 *Since a strong majority of the failures are common-case failures, more FPGA resources should be allocated for the common-case erasure decoding, while reducing performance target for the general-case.*

6.4.2 A Case for Coefficient Pre-Computation

Although the Vandermonde matrix is always invertible, it suffers an $O(n^3)$ time complexity with the standard Gaussian elimination algorithm. As discussed in Section 6.1.2, $\mathbf{F}$ is an identity matrix with some rows replaced by the rows of the encoding coefficients matrix ($\mathbf{G}$) rows. Similar to $\mathbf{F}$, $\mathbf{F}^{-1}$ is a $k \times k$ matrix in which k is the number of data blocks. It is possible to avoid matrix inversion by enumerating all possible failure cases and pre-computing their corresponding decoding matrices.

The overhead for pre-computing the inverted matrices would be the number of failure cases multiplied by the size of each inverted matrix. For an RS(n, k) code operating over GF(2^w), the overhead for pre-computing the decoding matrices for failures with α failed data blocks is:

$$\mathrm{OV}_{\alpha}(\mathrm{k},\mathrm{m}) = \binom{k}{\alpha} \times \binom{m}{\alpha} \times k \times k \times \frac{w}{8}(Bytes) \tag{6.15}$$

The first two terms count the number of possible cases while the last three terms count the required storage for each inverted matrix. As mentioned before, w is the word size for the GF. Consequently, the overhead for pre-computing the inverted matrices for all recoverable failures is:

$$\mathrm{OV}(\mathrm{k},\mathrm{m}) = \sum_{\alpha=1}^{m} \mathrm{OV}_{\alpha}(\mathrm{k},\mathrm{m}) \tag{6.16}$$

Example 6.5 *An RS(14,10) code operating over GF(2^8) can tolerate up to 4 failed data blocks. Table 6.2 shows the overhead for pre-computing the inverted matrices for this code.*

Example 6.5 shows that pre-computing the decoding matrices of a typical erasure code does not impose a significant overhead. Moreover, common-case failures *i.e.*, single data block failures, contribute to a small fraction of the total overhead. Consequently, it would be reasonable to pre-compute the decoding matrices for the common-case.

Table 6.2: Example 6.5:Pre-computation overhead for RS(14,10) code with $w = 8$

# Data Block Failures (α)	$\mathrm{OV}_{\alpha}(\mathrm{k},\mathrm{m})$
1 Block ($\alpha = 1$)	$4KB$
2 Blocks ($\alpha = 2$)	$27KB$
3 Blocks ($\alpha = 3$)	$48KB$
4 Blocks ($\alpha = 4$)	$21KB$
Total ($\mathrm{OV}(\mathrm{k},\mathrm{m})$)	**100KB**

Design Decision 6.2 *Since there are a limited number of possible coefficient matrices for the common-case failures, they can be inverted in advance. Therefore, with limited storage overhead, the matrix inversion computation of common-case erasure coding can be completely avoided at runtime.*

6.4.3 A Case for Reduced Computations

The inherent structure of the decoding allows reducing its storage overhead as well as avoiding the unnecessary computations.

Definition 6.1 *Row i of a square matrix, $\mathbf{A}$, is called identity Row if:*

$$a_{i,j} = \begin{cases} 1 & i = j \\ 0 & otherwise \end{cases}$$

Theorem 6.1 *Let $\mathbb{S}(.)$ denote the set of identity rows of an invertible matrix. We have:*

$$\mathbb{S}(\mathbf{A}) = \mathbb{S}(\mathbf{A}^{-1}) \tag{6.17}$$

In other words:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots \\ 0 & 1 & 0 & 0 & \cdots \\ a_{i,0} & a_{i,1} & a_{i,2} & a_{i,3} & \cdots \\ 0 & 0 & 0 & 1 & \cdots \\ a_{j,0} & a_{j,1} & a_{j,2} & a_{j,3} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots \\ 0 & 1 & 0 & 0 & \cdots \\ a'_{i,0} & a'_{i,1} & a'_{i,2} & a'_{i,3} & \cdots \\ 0 & 0 & 0 & 1 & \cdots \\ a'_{j,0} & a'_{j,1} & a'_{j,2} & a'_{j,3} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Proof: *Let row i of matrix $\mathbf{A}$ be an identity row:*

$$\mathbf{A} = \begin{matrix} \\ i \end{matrix} \begin{matrix} \overset{\quad\quad\quad\quad i}{} \\ \end{matrix} \begin{bmatrix} | & | & | & | & | \\ 0 & \dots & 1 & 0 & \dots \\ | & | & | & | & | \end{bmatrix}$$

Multiplying A by any matrix does not change its i^{th} row. Let $d_{i,j}$ be the elements of matrix D. Then:

$$\mathbf{A} \times \mathbf{D} = \mathbf{D'} = \begin{bmatrix} | & | & | & | \\ d_{i,0} & d_{i,1} & d_{i,2} & \dots \\ | & | & | & | \end{bmatrix}$$

We have:

$$\mathbf{A}^{-1} \times \mathbf{D'} = \mathbf{D}$$

Since the i^{th} row of $\mathbf{D}$ *and* $\mathbf{D}'$ *are identical, the i^{th} row of* $\mathbf{A}^{-1}$ *has to be an identity row. Consequently:*

$$\mathbf{A}^{-1} = \begin{array}{c} \\ i \end{array} \begin{bmatrix} | & | & | & | & | \\ 0 & \dots & 1 & 0 & \dots \\ | & | & | & | & | \end{bmatrix} \begin{array}{c} \\ \\ i \end{array}$$

■

In Equation 6.4, the number of non-identity rows in $\mathbf{F}$ *i.e.*, the rows that contain $c_{i,j}$s, is equal to the number of failed data blocks. Since the rest of the rows of the matrix $\mathbf{F}^{-1}$ are identity rows, we can avoid storing them. Moreover, since the purpose of the decoding is just to generate the failed blocks, we can avoid multiplying $\mathbf{R}$ by the identity rows of $\mathbf{F}^{-1}$ since it results in generating non-failed data blocks. In other words, we can multiply $\mathbf{R}$ by only non-identity rows of $\mathbf{F}^{-1}$, which significantly reduces the storage overhead of the decoding matrices. Consequently, pre-computing the decoding coefficients for a failure with α failures requires storing α (rather than k) rows of the decoding matrix. This observation reduces the overhead for pre-computing the decoding coefficients ($\mathrm{OV}_\alpha(\mathrm{k},\mathrm{m})$) by a factor of $\frac{\alpha}{k}$. Therefore we have:

$$\mathrm{OV}_\alpha(k,m) = \binom{k}{\alpha} \times \binom{m}{\alpha} \times \alpha \times k \times \frac{w}{8}(Bytes) \tag{6.18}$$

Any α blocks out of k data blocks could fail and any α blocks of the m parity blocks can replace them (the first two terms in Equation 6.18). On the other hand, the decoding matrix of each recoverable failure with α failed data blocks has α not-identity rows. Since F is a square matrix of size k, pre-computation of the decoding matrix for each α data block failure requires storing $\alpha \times k$ words (the second two terms in Equation 6.18). The size of each coefficient *i.e.*, the word size, is equal to the code's GF size. Consequently, the total overhead of pre-computing decoding coefficients would be:

$$\mathrm{OV}(k,m) = \sum_{i=1}^{m} \mathrm{OV_i}(k,m) \tag{6.19}$$

Example 6.6 *An RS(*14, 10*) code operating over GF(*2^8*) requires* 28KB *to pre-compute and store all decoding matrices. This is while pre-computing decoding matrices for single data block failures requires* 400 *bytes.*

The numbers in Example 6.6 show that although single data block failures are the common-case, they have small overhead for pre-computation. Consequently, in this work, we only store the decoding coefficients for the single data block failures.

However, our proposed architecture for the common-case, can easily be accommodate pre-computations for multiple data block failures as well.

Design Decision 6.3 *Since the decoding matrix is dominated by the identity rows, the complexity of common-case erasure coding can be reduced from matrix-vector multiplication to vector-vector dot product.*

The advantage of pre-computation is its scalability since the number coefficients does not depend on the block size. Instead, it only depends on the number of data and parity blocks, and the Galois Field size.

6.5 Architecture

In this section, we describe our FPGA implementation of the erasure encoding and decoding algorithms (referred to as the erasure code unit). We detail on both, high-level architecture and the micro-architecture of our design.

As mentioned in Section 6.4.1, it is reasonable to implement a high-throughput erasure unit for the common-case while setting a lower performance target for the general-case. Pre-computing the decoding matrices makes the decoding operation identical to the encoding *i.e.*, multiplying the vector of coefficients by the input data. Since the common-case unit has higher throughput, it is designed to be able to perform encoding as well. Therefore, encoding requests are forwarded to the common-case erasure code unit.

Figure 6.3 depicts the high-level architecture of our design. The design has the following interface:

- *Input data*: The data that needs to be processed. It is stored in memory and the design can access it through a memory interface.
- $Dec/\overline{Enc}$: Determines whether the input data should be decoded or encoded.
- *Fail Idxs*: An array that contains the indices of the failed data blocks (utilized in decoding operation).

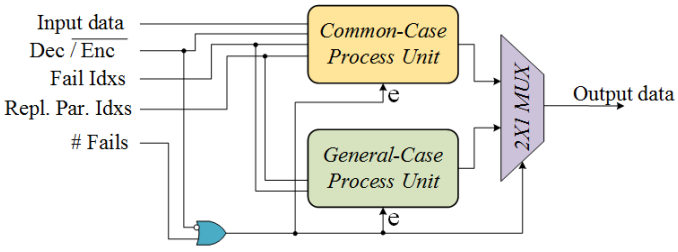


Figure 6.3: High-level diagram for the erasure code unit

- *Repl. Par. Idxs*: An array that contains the indices of the parity blocks that have replaced the data blocks in the input data (utilized in decoding operation).
- $\#Fails$: Determines the number of failed data blocks (utilized in decoding operation).
- *Output data*: The output data that is written to a memory through a memory interface.

At the input stage, the values of $Dec/\overline{Enc}$ and $\#Fails$ inputs determine which unit should be enabled. If the operation is encoding, or decoding with single block failure, then the common-case unit is activated. Otherwise, the general-case unit is activated to process the input data. At the output stage, a multiplexer decides the proper output from the activated unit.

6.5.1 Design for General-Case Decoder

Figure 6.4 illustrates the block diagram of the general-case unit. The matrix multiply unit is the block that performs the matrix multiplication. The other blocks are to provide the suitable coefficients for the operation.

The matrix inverse unit determines the failure case through *Failed Idxs* and *Repl. Par. Idxs* inputs. It then calculates the inverse matrix required for the decoding. Finally, the matrix multiply unit uses the computed decoding coefficients and the input data to reconstruct the failed data blocks.

6.5.2 Design for Common-Case Decoder and Encoder

We implemented a fully parametrized tool using C programming language that generates the register transfer level (RTL) Verilog for any erasure code unit. Our RTL generator has the following parameters:

- w: the GF size

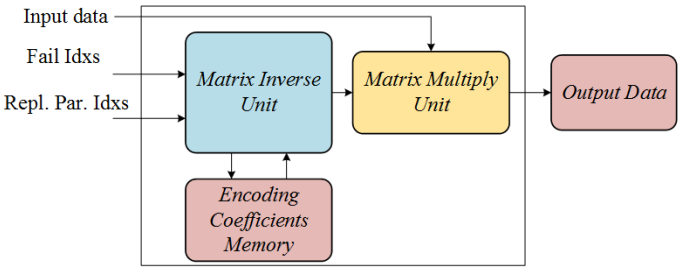


Figure 6.4: Block diagram for the general-case decoder

- *Poly*: the primitive polynomial for the GF
- k: the number of data blocks
- m: the number of parity blocks
- WPC: the number of words per cycle

This parametrized environment is suitable for erasure code design space exploration. At first, *Poly* and w are utilized to generate the GF multiplier. Then, this multiplier and the rest of the parameters are used to produce the erasure code unit.

We changed the order of operations in matrix multiplication to avoid storing the input data in the memory. In the conventional order of matrix multiplication, $A \times B$, the rows of A are multiplied by the columns of B. This computation order requires each element of B at different stages of the multiplication. Instead, each element of B, $b_{i,j}$, can be multiplied by column i of matrix A to partially generate column i of the result. We elaborate on this intuition by an example.

Example 6.7 *Consider the following multiplication:*

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \end{bmatrix}$$

$b_{1,1}$ is required to generate two terms: $a_{1,1} \times b_{1,1}$ and $a_{2,1} \times b_{1,1}$. The former is used in computing $c_{1,1}$ while the latter is used in computing $c_{2,1}$. It is possible to calculate these terms upon arrival of $b_{1,1}$ so that it is not required for the rest of multiplication process.

In the common-case design, the data is fed to the design in a streaming fashion and upon arrival of each word, all its contributions to the result are computed. Therefore, it is not required for further computations.

Figure 6.5 illustrates the block diagram for our common-case design. The input data is fed to the unit in a streaming fashion. The $Dec/\overline{Enc}$ input determines the operation. In the encoding case, all nodes are incorporated to generate all parities. There are m nodes *i.e.*, P_1 to P_m, each computing one of the parity blocks. As mentioned before, each parity block corresponds to one row in matrix E. Therefore, each node has a round shift register of size (k) that holds its corresponding coefficients.

In the decoding case, *Fail Pattern* determines the address of the decoding matrix. Then, the appropriate decoding coefficients are loaded into P_1's round shift register. Finally, the input data is streamed in to reconstruct the failed block. We store

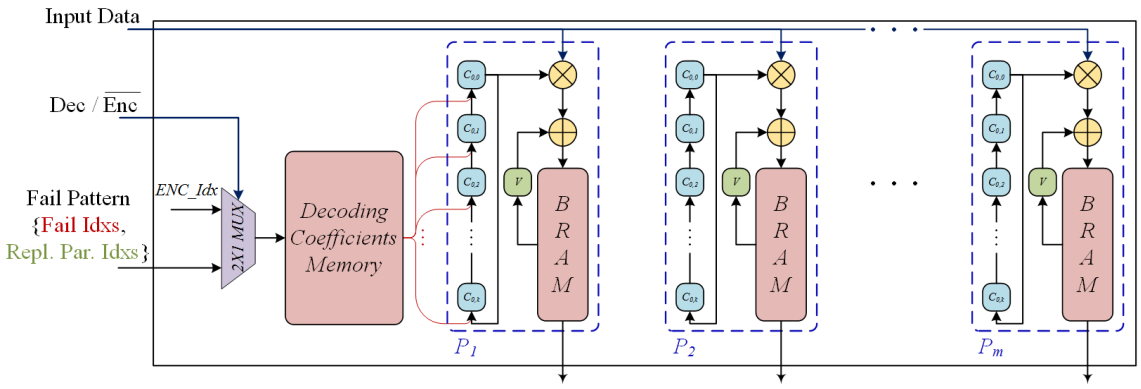


Figure 6.5: Block diagram for the common-case erasure code decoder and encoder

the encoding coefficients for the first parity block as the last entry of the decoding coefficients memory, so that they could be loaded back to the round shift register in the case of an encoding operation. Our RTL generator has a parameter WPC to set the number of words that are being processed in parallel. It has to be mentioned that the common-case design potentially supports pre-computation for multiple block failures. In that case, the only modification would be increasing the decoding coefficients memory and storing the pre-computed decoding matrices. In addition, there should be a logic to compute the decoding matrix's address based on the *Fail Pattern*.

6.6 Evaluation

We used Vivado Design Suite 2016.2 to implement our erasure codec on a Virtex7 (xc7vx485tffg1761-1) FPGA. For our first set of experiments, we use the RS(14, 10) code which is same as Facebook's erasure code [68] and the block size is set to $1KB$. We also set the GF size to 8 *i.e.*, 1 byte per word.

While it is common to compare the merits of competing designs by their peak throughput, both state-of-the-art software and hardware designs can scale their performance with either more cores or more hardware resources. To make apple-to-apple comparisons, we focus on *efficiency metrics*: To examine resource efficiency, we compute the achievable *throughput per thousand LUTs* ($MBps/KLUT$) and compare our design with Ruan's design.

6.6.1 Erasure Codec Results

Table 6.3 reports the resource and power consumption results for our general-case and common-case designs. The table shows the results for two parallelism factors (WPC). WPC=16 is the case that achieves the peak throughput, while WPC=8 is

Table 6.3: Resource usage and performance results for our erasure code design

Resource	Common-Case Design WPC=8	Common-Case Design WPC=16	General-Case Design
LUT	3315	6575	7562
FF	5579	11107	4479
BRAM	16	32	11
$\mathbf{f_{max}}$**(MHz)**	312	303	125
Throughput (GBps)	2.5	4.85	0.63

the case where we achieved the best efficiency. Our general-case design has lower performance. However, since the common-case is strongly dominant, the overall throughput is mainly dictated by the performance of the common-case design. Since WPC=8 achieves best resource efficiency, it is used for rest of our evaluations.

6.6.2 The Impact of Disk Failure Probability

As mentioned before, we used our computations in Section 6.4.1. in order to calculate the overall throughput from our general-case and common-case designs. For that, we assumed the typical disk failure probability, $q = 0.01$. However, this parameter may vary based on the disks quality and age. We varied the disk failure probability (q) to investigate its impact on our performance metrics. Figure 6.6 illustrates the efficiency and the peak throughput for different values of q. The last bar shows the efficiency result for the Ruan's codec. Ruan's design suggests pre-computing the decoding coefficients for all failure cases. Therefore, there is no notion of common-case in his design. The results show that even for a very pessimistic value for the disk failure probability ($q = 0.1$), our design is 4.2× more efficient. This is due to fact that our architecture allocates more resources to the common-case.

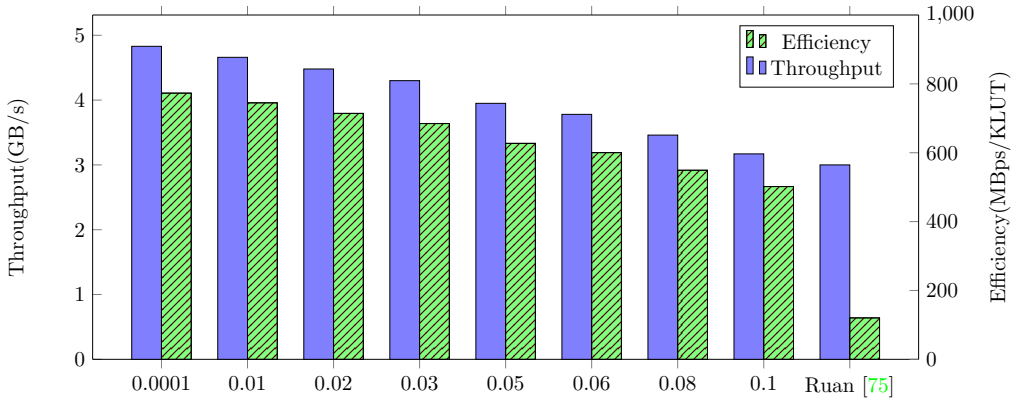


Figure 6.6: The impact of disk failure probability and comparison with the previous work

Table 6.4: Experimental results for design space exploration

Exp#	n	k	GF Size(w)	WPC	LUT	Register	BRAM	Throughput ($_{GBps}$)	Efficiency ($_{MBps/KLUT}$)	# nines of reliability
1	14	10	8	8	3315	5579	16	2.5	772	5
2	14	10	16	8	8733	12543	16	4.85	568	5
3	12	8	8	8	3369	5003	16	2.42	736	5
4	16	12	8	8	3539	5883	16	2.5	723	5
5	15	10	8	8	4545	7324	20	2.5	563	6
6	16	10	8	8	5457	9028	24	2.35	441	8
7	14	10	8	16	6575	11107	32	4.85	755	5

6.6.3 Design Space Exploration

The overall performance is highly dictated by the performance of our common-case design. Our common-case design is parameterized to ease the design space exploration. As our last set of experiments, we varied the parameters to study their impacts on overall system performance. In each set of experiments, one parameter is varied while others are fixed. Table 6.4 reports the results for our design space exploration experiments. The second column shows the parameter under investigation for each experiment. The last column shows the reliability level for each experimented code. Since one parameter is varied during each exploration, experiment **1** is shared across all our investigations. Our common-case unit has streaming input and the block size does not affect the overall performance. Therefore, the block size is fixed ($1KB$) for all our experiments. Table 6.4 leads to the following conclusions:

- **GF Size** (w): The results from experiments **1** and **2** show that the achieved throughput is proportional to the GF size. However, larger GF sizes lead to lower efficiency. Moreover, since the number of required input pins is $WPC \times w$, larger GF size leads to higher peak throughput. However, the efficiency is decreased due to increased computation complexity.

- **Number of Data Block** (k): The results from experiments **3**, **1**, and **4** show that the number of data blocks does not affect the peak throughput. This is due to the fact that the number of data blocks does not affect resource usage in our design. Instead, it determines the shift operation.

- **Number of Parity Blocks** (m): The results from experiments **1**, **5**, and **6** show that increasing the number of parity blocks does not affect the peak throughput. However, it reduces resource efficiency due to higher resource usage.

- **Word Per Cycle** (WPC): The results from experiments **1** and **7** show that increasing the WPC parameter would increase the resource usage. However,

this increase is proportional to the absolute throughput. Therefore, it does not affect the efficiency.

6.7 Summary

In this chapter, we demonstrated a quantitative study of erasure coding design on FPGAs. We argue, with probabilistic analysis, that an efficient implementation should allocate more resources for the common-case failure mode. With the efficiency metric established as throughput per thousand LUT, and with proposed techniques, we conclude that erasure codec can be implemented on modern FPGAs with $772 MBps/KLUT$ for an RS(14, 10) code, with 5 nines of reliability.

Chapter 7

Summary & Conclusion

The ever-increasing demand for data storage has led to the development and proliferation of cloud storage systems as well as flash memory based storage devices. For that, cloud storage service providers and flash memory manufactures often provide data reliability through some error correcting code (ECC).

This dissertation centers around the implementation of ECC in data storage. We target FPGA as a suitable custom computation platform featuring short design cycles to cope with the ever-changing ECC requirements and specifications in storage industry. Throughout this research, we aimed to design efficient FPGA micro-architectures for error correcting codes as well as identifying interesting trade offs between hardware and code performance.

7.1 Summary of Contributions

The majority of this research focus on the device-level ECC, quasi-cyclic LDPC (QC-LDPC) codes, a class of LDPC codes that are not only hardware-friendly, but also of good code performance. In a separate effort, we explored FPGA's potential for erasure coding as an alternative to replication as a system-level ECC. Our contributions in this research can be summarized as follows:

- **Foldable FPGA micro-architecture for QC-LDPC decoders**: We leverage FPGA's inherent architecture to design rotary register file (RRF), a foldable parallel architecture for strided circular access, seen in QC-LDPC decoding. RRF is then used to propose an FPGA micro-architecture for QC-LDPC decoders. Our decoder's support for foldable parallelism enables a spectrum of design instances form the most serial to the most parallel decoder. We used the proposed architecture to implement the noisy gradient decent bit flipping (NGDBF) decoder.

- **Learning-based finite alphabet QC-LDPC decoder for flash memories**: With practicality as one of the main objectives in this research, we explored the inherent limitations in flash memories that could impact the ECC performance in these devices. These limitations were then desirably exploited to define a finite LDPC decoder design space. Then, machine learning techniques were incorporated along with the finite alphabet decoding concept to propose an end-to-end solution for hardware-friendly finite alphabet iterative decoder (FAID) for flash memories. Finally, we used our proposed micro-architecture to implement our FAID.

- **Erasure coding design on FPGA**: By performing a quantitative study of erasure coding design on FPGA, we demonstrated, through probabilistic analysis, that an efficient implementation requires allocating more resources to the common-case, while reducing the performance target for less probable cases.

7.2 Conclusion

This research aims at harnessing FPGA's potential for error correcting codes in data storage. Based on the knowledge obtained throughout this pursuit, we address the initially raised questions as follows:

- **Is there an FPGA-optimized micro-architecture for LDPC?**

  By desirably exploiting FPGA's inherent physical architecture, it would be possible to design efficient LDPC micro-architectures. Moreover, the support for foldable parallelism is vital for design space exploration in this context. In fact, by deferring the design decision, foldable parallelism enables the end-user to strike a trade-off between resource utilization and throughput.

- **Is there interesting trade off between LDPC micro-architecture and code performance?**

  Code has been masterfully designed by scientists with sophisticated mathematical skills, and implementation concerns such as quantization and micro-architectures usually come as after-facts. This study represents a new trend where implementation constraints, such as finite precision, can come as first-class citizen, and important aspects of code, can be *learned*, after-facts.

7.3 Future Work

This research's mandate was mainly to pursue interesting trade offs between hardware efficiency and code performance in the context of ECC for data storage. We intend to expand our methodology to other codes and we expect to find more interesting trade offs.

In this work, we used linear congruential random number generators for our software simulation, and LFSR-based random generators for our experiments in Chapter 5. However, statistically better random generators such as the permuted congruential generator (PCG) [78] can be used to improve the confidence and accuracy of our experiments.

Our proposed learning method involves two stages; full-precision training, and designing the quantization function based on the training phase outcome. However, the quantization process could be part of the learning process through quantized training. A critical issue for training with low-precision activation functions is that their gradients could disappear which makes the backward propagation challenging. However, there are some recent works aiming to mitigate this issue [79, 80]. However, there is no discussion on hardware implementation or study on large-block, high-rate codes that are suitable for flash memories. We believe quantized training for FAID could reveal more interesting trade offs between hardware efficiency and code performance in this context.

Our erasure coding design takes advantage of pre-computing the decoding coefficients. These pre-computed coefficients are then stored in a lookup table to accelerate the decoding process. However, using cyclic codes, it would be possible to leverage the cyclicity of the decoding matrix to reduce the lookup table size [81].

Bibliography

[1] Petter Bae Brandtzg. "Big Data, for better or worse: 90% of worlds data generated over last two years". In: *SCIENCE DAILY* (2013, Accessed on March 2016). URL: https://www.sciencedaily.com/releases/2013/05/130522085217.htm.

[2] David Reinsel, John Gantz, and Johnl Rydning. "Data Age 2025: The Evolution of Data to Life-Critical Dont Focus on Big Data; Focus on Data Thats Big". In: *IDC, Seagate, April* (2017).

[3] *Amazon S3*. URL: https://aws.amazon.com/s3/.

[4] *OneDrive*. URL: https://onedrive.live.com/about/en-ca/.

[5] Robert Gallager. "Low-density parity-check codes". In: *IRE Transactions on information theory* 8.1 (1962), pp. 21–28.

[6] Jia Dong. "Estimation of bit error rate of any digital communication system". PhD thesis. Télécom Bretagne, Université de Bretagne Occidentale, 2013.

[7] JEDEC STANDARD. *Solid-State Drive (SSD) Requirements and Endurance Test Method, JESD218*. 2010. URL: https://www.jedec.org/sites/default/files/docs/JESD218A.pdf.

[8] Rino Micheloni et al. *3D Flash memories*. Springer, 2016.

[9] N. Mielke, T. Marquart, Ning Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. R. Nevill. "Bit error rate in NAND Flash memories". In: *2008 IEEE International Reliability Physics Symposium*. 2008, pp. 9–19. DOI: 10.1109/RELPHY.2008.4558857.

[10] Keith B. Oldham, Jan C. Myland, and Jerome Spanier. "The Error Function erf(x) and Its Complement erfc(x)". In: *An Atlas of Functions: with Equator, the Atlas Function Calculator*. New York, NY: Springer US, 2009, pp. 405–415. ISBN: 978-0-387-48807-3. DOI: 10.1007/978-0-387-48807-3_41. URL: http://dx.doi.org/10.1007/978-0-387-48807-3_41.

[11] Claude Elwood Shannon. "Communication in the presence of noise". In: *Proceedings of the IRE* 37.1 (1949), pp. 10–21.

[12] J. Wang, K. Vakilinia, T. Chen, T. Courtade, G. Dong, T. Zhang, H. Shankar, and R. Wesel. "Enhanced Precision Through Multiple Reads for LDPC Decoding in Flash Memories". In: *IEEE Journal on Selected Areas in Communications* 32.5 (2014), pp. 880–891. DOI: 10.1109/JSAC.2014.140508.

[13] Thomas M. Cover and Joy A. Thomas. *Elements of information theory*. Wiley, 2006.

[14] David JC MacKay and Radford M Neal. "Near Shannon limit performance of low density parity check codes". In: *Electronics letters* 32.18 (1996), p. 1645.

[15] 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". In: *Proceedings of the IEEE* 95.11 (2007), pp. 2142–2156. ISSN: 0018-9219. DOI: 10.1109/JPROC.2007.905132.

[16] C. Yang, Y. Emre, and C. Chakrabarti. "Product Code Schemes for Error Correction in MLC NAND Flash Memories". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20.12 (2012), pp. 2302–2314. DOI: 10.1109/TVLSI.2011.2174389.

[17] Jinghu Chen and M. P. C. Fossorier. "Decoding low-density parity check codes with normalized APP-based algorithm". In: *GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No.01CH37270)*. Vol. 2. 2001, 1026–1030 vol.2. DOI: 10.1109/GLOCOM.2001.965573.

[18] Jonghong Kim and Wonyong Sung. "Rate-0.96 LDPC decoding VLSI for soft-decision error correction of NAND flash memory". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.5 (2014), pp. 1004–1015.

[19] Gopalakrishnan Sundararajan, Chris Winstead, and Emmanuel Boutillon. "Noisy gradient descent bit-flip decoding for LDPC codes". In: *IEEE Transactions on Communications* 62.10 (2014), pp. 3385–3400.

[20] *LSI Delivers Industry's First 40nm Read Channel To Hard Disk Drive Manufacturers*. 2009. URL: https://www.networkcomputing.com/storage/lsi-delivers-industrys-first-40nm-read-channel-hard-disk-drive-manufacturers/1591961818.

[21] X. Chen, J. Kang, S. Lin, and V. Akella. "Memory System Optimization for FPGA-Based Implementation of Quasi-Cyclic LDPC Codes Decoders". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 58.1 (2011), pp. 98–111. ISSN: 1549-8328. DOI: 10.1109/TCSI.2010.2055250.

[22] S. A. A. Zaidi, M. Awais, C. Condo, M. Martina, and G. Masera. "FPGA accelerator of Quasi cyclic EG-LDPC codes decoder for NAND flash memories". In: *2013 Conference on Design and Architectures for Signal and Image Processing*. 2013, pp. 190–195.

[23] A. Amaricai and O. Boncalo. "Cost effective FPGA implementation for hard decision LDPC decoders". In: *2016 24th Telecommunications Forum (TELFOR)*. 2016, pp. 1–4. DOI: 10.1109/TELFOR.2016.7818769.

[24] S. Nimara, O. Boncalo, A. Amaricai, and M. Popa. "FPGA architecture of multi-codeword LDPC decoder with efficient BRAM utilization". In: *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. 2016, pp. 1–4. DOI: 10.1109/DDECS.2016.7482452.

[25] Y. Liu, C. Zhang, P. Song, and H. Jiang. "A high-performance FPGA-based LDPC decoder for solid-state drives". In: *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2017, pp. 1232–1235. DOI: 10.1109/MWSCAS.2017.8053152.

[26] O. Boncalo and A. Amaricai. "Ultra High Throughput Unrolled Layered Architecture for QC-LDPC Decoders". In: *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2017, pp. 225–230. DOI: 10.1109/ISVLSI.2017.47.

[27] M. Milicevic and P. G. Gulak. "A Multi-Gb/s Frame-Interleaved LDPC Decoder With Path-Unrolled Message Passing in 28-nm CMOS". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.10 (2018), pp. 1908–1921. DOI: 10.1109/TVLSI.2018.2838591.

[28] L. Shannon, V. Cojocaru, C. N. Dao, and P. H. W. Leong. "Technology Scaling in FPGAs: Trends in Applications and Architectures". In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 2015, pp. 1–8. DOI: 10.1109/FCCM.2015.11.

[29] Kai Zhao, Wenzhe Zhao, Hongbin Sun, Tong Zhang, Xiaodong Zhang, and Nanning Zheng. "LDPC-in-SSD : making advanced error correction codes work effectively in solid state drives." In: *FAST*. Vol. 13. 2013, pp. 244–256.

[30] S. Korkotsides and T. A. Antonakopoulos. "Architecture of a NVM-based storage system using adaptive LDPC codes". In: *2016 5th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. 2016, pp. 1–4. DOI: 10.1109/MOCAST.2016.7495149.

[31] SNIA Standard. *Solid State Storage (SSS) Performance Test Specification (PTS)*. 2020. URL: https://www.snia.org/sites/default/files/technical_work/PTS/SSS_PTS_2.0.2.pdf.

[32] *UltraScale Architecture Configurable Logic Block*. 2017 Accessed on Feb 2021. URL: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf.

[33] *Intel Stratix 10 Logic Array Blocks and Adaptive Logic Modules User Guide*. 2020 Accessed on Feb 2021. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-lab.pdf.

[34] W Snyder, P Wasson, and D Galbi. *Verilator-Convert Verilog code to C++/SystemC*. 2012.

[35] T. J. Richardson and R. L. Urbanke. "The capacity of low-density parity-check codes under message-passing decoding". In: *IEEE Transactions on Information Theory* 47.2 (2001), pp. 599–618. DOI: 10.1109/18.910577.

[36] Jinghu Chen, A. Dholakia, E. Eleftheriou, M. P. C. Fossorier, and Xiao-Yu Hu. "Reduced-complexity decoding of LDPC codes". In: *IEEE Transactions on Communications* 53.8 (2005), pp. 1288–1299. DOI: 10.1109/TCOMM.2005.852852.

[37] S. K. Planjery, D. Declercq, L. Danjean, and B. Vasic. "Finite Alphabet Iterative DecodersPart I: Decoding Beyond Belief Propagation on the Binary Symmetric Channel". In: *IEEE Transactions on Communications* 61.10 (2013), pp. 4033–4045. DOI: 10.1109/TCOMM.2013.090513.120443.

[38] T. T. Nguyen-Ly, K. Le, V. Savin, D. Declercq, F. Ghaffari, and O. Boncalo. "Non-surjective finite alphabet iterative decoders". In: *2016 IEEE International Conference on Communications (ICC)*. 2016, pp. 1–6. DOI: 10.1109/ICC.2016.7511111.

[39] F. Ghaffari, K. Le, and D. Declercq. "The Probabilistic Finite Alphabet Iterative Decoder for Low-Density Parity-Check Codes". In: *2019 17th IEEE International New Circuits and Systems Conference (NEWCAS)*. 2019, pp. 1–4. DOI: 10.1109/NEWCAS44328.2019.8961256.

[40] B. Vasic, X. Xiao, and S. Lin. "Learning to Decode LDPC Codes with Finite-Alphabet Message Passing". In: *2018 Information Theory and Applications Workshop (ITA)*. 2018, pp. 1–9. DOI: 10.1109/ITA.2018.8503199.

[41] E. Nachmani, Y. Be'ery, and D. Burshtein. "Learning to decode linear codes using deep learning". In: *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. 2016, pp. 341–346. DOI: 10.1109/ALLERTON.2016.7852251.

[42] L. Lugosch and W. J. Gross. "Neural offset min-sum decoding". In: *2017 IEEE International Symposium on Information Theory (ISIT)*. 2017, pp. 1361–1365. DOI: 10.1109/ISIT.2017.8006751.

[43] X. Wu, M. Jiang, and C. Zhao. "Decoding Optimization for 5G LDPC Codes by Machine Learning". In: *IEEE Access* 6 (2018), pp. 50179–50186. DOI: 10.1109/ACCESS.2018.2869374.

[44] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[45] Chris Loken, Daniel Gruner, Leslie Groer, Richard Peltier, Neil Bunn, Michael Craig, Teresa Henriques, Jillian Dempsey, Ching-Hsing Yu, Joseph Chen, L Jonathan Dursi, Jason Chong, Scott Northrup, Jaime Pinto, Neil Knecht, and Ramses Van Zon. "SciNet: Lessons Learned from Building a Power-efficient Top-20 System and Data Centre". In: *Journal of Physics: Conference Series* 256 (2010), p. 012026. DOI: 10.1088/1742-6596/256/1/012026. URL: https://doi.org/10.1088/1742-6596/256/1/012026.

[46] Marcelo Ponce, Ramses van Zon, Scott Northrup, Daniel Gruner, Joseph Chen, Fatih Ertinaz, Alexey Fedoseev, Leslie Groer, Fei Mao, Bruno C. Mundim, Mike Nolta, Jaime Pinto, Marco Saldarriaga, Vladimir Slavnic, Erik Spence, Ching-Hsing Yu, and W. Richard Peltier. "Deploying a Top-100 Supercomputer for Large Parallel Workloads: The Niagara Supercomputer". In: *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*. PEARC '19. New York, NY, USA: Association for Computing Machinery, 2019. ISBN: 9781450372275. DOI: 10.1145/3332186.3332195. URL: https://doi.org/10.1145/3332186.3332195.

[47] *pseudo-random numbers using the linear congruential algorithm*. URL: https://linux.die.net/man/3/lrand48.

[48] Ramses van Zon. *Implementation of a linear congruential generator with log(n) skipping*. 2016. URL: https://gitrepos.scinet.utoronto.ca/public/?a=summary&p=serdy.

[49] Pierre L'Ecuyer. "Tables of Maximally Equidistributed Combined LFSR Generators". In: *Math. Comput.* 68.225 (Jan. 1999), 261269. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-99-01039-X. URL: https://doi.org/10.1090/S0025-5718-99-01039-X.

[50] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw. "Parallel random numbers: As easy as 1, 2, 3". In: *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–12. DOI: 10.1145/2063384.2063405.

[51] JunKyu Lee, G. D. Peterson, R. J. Harrison, and R. J. Hinde. "Hardware accelerated Scalable Parallel Random Number Generators for Monte Carlo methods". In: *2008 51st Midwest Symposium on Circuits and Systems*. 2008, pp. 177–180. DOI: 10.1109/MWSCAS.2008.4616765.

[52] S Y Jun, P Canal, J Apostolakis, A Gheata, and L Moneta. "Vectorization of random number generation and reproducibility of concurrent particle transport simulation". In: *Journal of Physics: Conference Series* 1525 (2020), p. 012054. DOI: 10.1088/1742-6596/1525/1/012054. URL: https://doi.org/10.1088/1742-6596/1525/1/012054.

[53] Q. Diao, J. Li, S. Lin, and I. F. Blake. "New Classes of Partial Geometries and Their Associated LDPC Codes". In: *IEEE Transactions on Information Theory* 62.6 (2016), pp. 2947–2965. DOI: 10.1109/TIT.2015.2508455.

[54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alch'e-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[55] O. Boncalo, A. Amaricai, and S. Nimara. "Memory-Centric Flooded LDPC Decoder Architecture Using Non-surjective Finite Alphabet Iterative Decoding". In: *2018 21st Euromicro Conference on Digital System Design (DSD)*. 2018, pp. 104–109. DOI: 10.1109/DSD.2018.00031.

[56] O. Boncalo, V. Savin, and A. Amaricai. "Unrolled layered architectures for non-surjective finite alphabet iterative decoders". In: *2017 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*. 2017, pp. 1–5. DOI: 10.1109/NORCHIP.2017.8124944.

[57] T. T. Nguyen-Ly, V. Savin, X. Popon, and D. Declercq. "High Throughput FPGA Implementation for regular Non-Surjective Finite Alphabet Iterative Decoders". In: *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*. 2017, pp. 961–966. DOI: 10.1109/ICCW.2017.7962783.

[58] James S Plank and Kevin M Greenan. *Jerasure: A library in C facilitating erasure coding for storage applications–version 2.0*. Tech. rep. Technical Report UT-EECS-14-721, University of Tennessee, 2014.

[59] *Intel Intelligent Storage Acceleration Library*. URL: https://software.intel.com/en-us/storage/ISA-L.

[60] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. "Ceph: A scalable, high-performance distributed file system". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 307–320.

[61] Anindya Sundar Das, Satyajit Das, and Jaydeb Bhaumik. "Design of RS(255, 251) Encoder and Decoder in FPGA". In: *International Journal of Soft Computing and Engineering (IJSCE)* 2.6 (2013), pp. 2231–2307.

[62] Mustapha Elharoussi, Asmaa Hamyani, and Mostafa Belkasmi. "VHDL Design and FPGA Implementation of a Parallel Reed-Solomon (15, K, D) Encoder/Decoder". In: *International Journal of Advanced Computer Science and Applications (IJACSA)* 4.1 (2013).

[63] Luigi Rizzo. "Effective erasure codes for reliable computer communication protocols". In: *ACM SIGCOMM computer communication review* 27.2 (1997), pp. 24–36.

[64] N. Macon and A. Spitzbart. "Inverses of Vandermonde Matrices". In: *The American Mathematical Monthly* 65.2 (1958), pp. 95–100. ISSN: 00029890, 19300972. URL: http://www.jstor.org/stable/2308881.

[65] David A Patterson, Garth Gibson, and Randy H Katz. *A case for redundant arrays of inexpensive disks (RAID)*. Vol. 17. 3. ACM, 1988.

[66] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. "RAID: High-Performance, Reliable Secondary Storage". In: *ACM Comput. Surv.* 26.2 (June 1994), 145185. ISSN: 0360-0300. DOI: 10.1145/176979.176981. URL: https://doi.org/10.1145/176979.176981.

[67] Dhruba Borthakur. "HDFS architecture guide". In: *HADOOP APACHE PROJECT* (2008).

[68] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. "Xoring elephants: Novel erasure codes for big data". In: *Proceedings of the VLDB Endowment*. Vol. 6. 5. VLDB Endowment. 2013, pp. 325–336.

[69] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. "Erasure Coding in Windows Azure Storage." In: *Usenix annual technical conference*. Boston, MA. 2012, pp. 15–26.

[70] Luigi Rizzo. *On the feasibility of software FEC*. Tech. rep. 1–16. Univ. di Pisa, Italy, 1997. URL: http://www.iet.unipi.it/~luigi/softfec.ps.

[71] Kazutaka Morita. "Sheepdog: Distributed storage system for qemu/kvm". In: *LCA 2010 DS&R miniconf* (2010).

[72] *Jerasure1.2: A Library in C/C++ Facilitating Erasure Coding for Storage Applications version 1.2*. URL: http://web.eecs.utk.edu/~plank/plank/papers/CS-08-627.html.

[73] James S Plank, Kevin M Greenan, and Ethan L Miller. "Screaming fast Galois field arithmetic using intel SIMD instructions." In: *FAST*. 2013, pp. 299–306.

[74] *Understanding Erasure Coding Offload*. Accessed 2017-01-10. URL: https://community.mellanox.com/docs/DOC-2414.

[75] *Reed-Solomon Erasure Codec Design Using Vivado High-Level Synthesis*. Accessed 2017-01-10. URL: https://www.xilinx.com/support/documentation/application_notes/xapp1273-reed-solomon-erasure.pdf.

[76] Z. Wilcox-O'Hearn. *Zfec 1.4.24 Open source code distribution*. URL: https://pypi.python.org/pypi/zfec/1.4.24.

[77] M. Zhang, S. Han, and P. P. C. Lee. "A Simulation Analysis of Reliability in Erasure-Coded Data Centers". In: *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. 2017, pp. 144–153. DOI: 10.1109/SRDS.2017.19.

[78] M. E. ONeill. "PCG: A family of simple fast space-efficient statistically good algorithms for random number generation". In: *ACM Transactions on Mathematical Software* (2014).

[79] X. Xiao, B. Vasic, R. Tandon, and S. Lin. "Designing Finite Alphabet Iterative Decoders of LDPC Codes Via Recurrent Quantized Neural Networks". In: *IEEE Transactions on Communications* 68.7 (2020), pp. 3963–3974. DOI: 10.1109/TCOMM.2020.2985678.

[80] X. Xiao, B. Vasic, R. Tandon, and S. Lin. "Finite Alphabet Iterative Decoding of LDPC Codes with Coarsely Quantized Neural Networks". In: *2019 IEEE Global Communications Conference (GLOBECOM)*. 2019, pp. 1–6. DOI: 10.1109/GLOBECOM38437.2019.9013364.

[81] Szymon Czynszak. "Decoding algorithms of Reed-Solomon code". MA thesis. Blekinge Institute of Technology, 2011.