Prime-Sized Multilevel Flash Memory with Non-Binary LDPC

by

Mohammed Al Ai Baky

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

April 2018

Author .....
Department of Electrical Engineering and Computer Science
April 10, 2018

Certified by .....
Dr. James Fitzpatrick
Engineering Fellow at Western Digital Corporation
Thesis Supervisor
April 10, 2018

Certified by .....
Yury Polyanskiy
Associate Professor
Thesis Supervisor
April 10, 2018

Accepted by .....
Christopher Terman
Chairman, Masters of Engineering Thesis Committee

Prime-Sized Multilevel Flash Memory with Non-Binary LDPC

by

Mohammed Al Ai Baky

Submitted to the Department of Electrical Engineering and Computer Science
on April 10, 2018, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

Flash memory companies are increasing the number of bits per cell to obtain higher information capacity per cell, starting from 1 bit/cell and going to 4 bits/cell recently. This scaling is enabled by the advancements in flash semiconductor technology, specifically the Bit Cost Scalable (BiCS) technology. However, capacity per cell scaling comes with performance, reliability, and endurance challenges. The industry has only used integer number of bits per cell, which makes the tradeoff between the capacity and the other system features less flexible than using fractional bits. This project explores programming 13 levels of charge (3.7 bits) into a QLC flash cell that normally carries 16 levels of charge (4 bits). We evaluate the 13-ary scheme against the 16-ary one and we show that the 13-ary has the same reliability at a lower SNR as the 16-ary, or the 13-ary has higher reliability than the 16-ary at the same SNR. We design binary and non-binary Quasi-Cyclic LDPC codes and implement Belief Propagation decoders for them.

Thesis Supervisor: Dr. James Fitzpatrick
Title: Engineering Fellow at Western Digital Corporation

Thesis Supervisor: Yury Polyanskiy
Title: Associate Professor

Acknowledgments

I'd like to extend my deep appreciation to everyone that helped me with this work. First, I'd like to thank Western Digital Corporation and the MIT VI-A program for offering the opportunity of conducting this work. I'm deeply grateful to Seishi Takamura, of Nippon Telegraph and Telephone Corporation, for his help with fast implementations of Non-Binary LDPC decoders. I'm also grateful to Dariush Divsalar, of NASA JPL, for his LDPC code design suggestions. I am indebted to Idan Alrod who made his computational resources available to me. Special thanks to Ahmed Hareedy, from the Laboratory for Robust Information Systems at UCLA, for his insights on the cutting edge research in the LDPC codes area.

I am equally indebted to the experts at Western Digital, specifically Rick Galbraith for openly sharing all his research work with me. I also appreciate Majid Nemati's remarks on LDPC for flash memory. Thank you to these people at MIT and Western Digital: Bruce Kaufman, Dudy Avraham, Henry Yip, John Jackson, Niranjay Ravindran, Jonas Goode, Manish Madhukar, Mostafa El Gamal, Nima Mokhlesi, Ravi Kumar, Steven Aronson, Angela Liu, Nancy Semanko, Tomas Palacios, and Kathleen Sullivan.

Finally, I'd like to express my sincere gratitude to my advisors: Jim Fitzpatrick and Yury Polyanskiy for connecting me to world class experts in this research area, and helping me finish this project in a tight schedule.

Contents

List of Figures

Chapter 1

Flash Memory Systems Introduction

1.1 Flash Memory Physics and Technologies

Flash memory was invented by Fujio Masuoka of Toshiba in the early 1980s. Intel and Toshiba started to commercialize the new technology in the late 1980s. Flash memory penetration in consumer and enterprise products has been increasing since then. All memory cards used in digital camera and mobile phones are flash-based storage devices, and the same is true for USB flash drives. In addition, *Solid-state Drives (SSDs)* are flash-based storage devices similar to *Hard-disk Drives (HDDs)*, but have a better performance than HDDs. In the late 2000s, SSDs started to replace HDDs in personal laptops for their desired features [3]. SSDs are also used in enterprise storage, such as data centers [4].

Historically, the basic unit, the *cell*, of flash memory consists of the floating-gate transistor that stores electrical charge (shown in Figure 1-1). The information is encoded in the amount (level) of this charge. In most of the flash products, these transistors are connected in the NAND configuration that resembles the NAND gate architecture (shown in Figure 1-2). These cells are laid in two-dimensional configuration and packaged into integrated chips. In fact, the term *NAND* has become interchangeable with *flash*, and we use it interchangeably in this monograph.

The floating-gate of the transistor is isolated and surrounded by an insulator, so that it traps the charge. A high voltage is applied to pass a charge across the

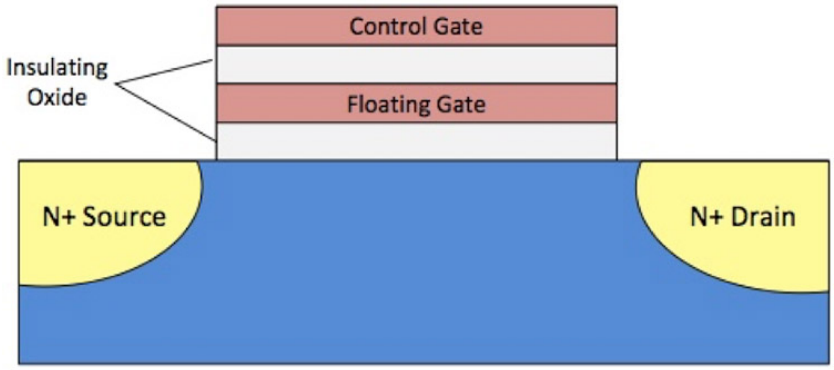


Figure 1-1: Floating-gate transistor.

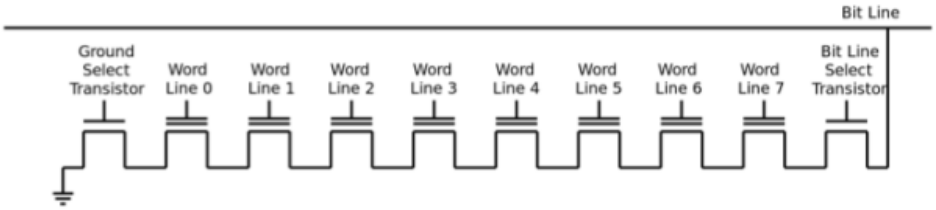


Figure 1-2: NAND flash architecture.

insulator into the gate, in a process called *Programming*. The floating-gate transistor is a noisy medium, resulting in a difference between the amount of charge programmed (written) to the transistor and the amount charge sensed (read). In programming, the exact charge passing into the gate is not deterministic, as the crossing through insulator is a complex statistical process [5].

The medium imposes challenges on the flash technology. There is a reliability issue preventing the stored charge levels from having deterministic values, instead they are approximated by a Gaussian distributions (shown in Figure 1-3). Endurance is another problem in which the gate insulator degrades gradually due to the high programming voltage applied across it. Endurance is measured in the number of *Program/Erase (P/E)* cycles the flash can sustain meeting a certain reliability condition. These challenges are tackled with signal processing and coding. We focus on the latter in this work, refer to Chapter 2 and Section 3.2. In addition, flash silicon fabrication processes can be improved to mitigate these challenges, as

in the technology in Section 1.1.1.

At the beginning of the flash technologies, all the products used the *Single-level Cell (SLC)*, in which a cell carries a single bit only represented by two levels. To increase the capacity of the cell, *MLC* consumer flash products with 4 levels followed in the late 90s (Figure 1-3), and were deployed in enterprise business around a decade after[6]. Adding more charge levels is very difficult because it decreases the reliability, endurance, and performance of the cell and the cell array. For the performance, the cells need to be programmed more slowly and precisely to result in narrower charge distributions in the same voltage space. The read performance also goes down as there are more charge levels to be read. The reliability, inversely proportional to the *Bit Error Rate (BER)*, decreases as the number of levels increases because the overlap between the distributions increases. For a similar reason, the endurance in terms of program/erase (P/E) cycles decreases with higher number of levels as well.

1.1.1 Bit Cost Scalable (BiCS) Technology

A further increase in the capacity density ($bits/mm^2$) has been developed to reduce the bit cost and meet the market demands for flash storage. Traditionally, the number of $bits/mm^2$ has increased through reductions in the feature size, but as the number of electrons in a cell has become very small, new technologies were necessary to scale up the capacity density of the NAND.

In 2007, Toshiba announced the BiCS technology, in which the memory cell arrays are fabricated in three dimensions (3D) with 64 and 96 layers of NAND cells, to scale up the capacity density [7], as shown in Figure 1-4. BiCS replaces the floating gate in the basic memory unit with a charge trapping layer [8]. For this reason and the fact that the spaces between the cells in BiCS are wider, the inter-cell coupling is lower in BiCS than in 2D NAND, and the inter-bitline coupling is significantly lower than inter-wordline coupling in BiCS. The coupling reduction results in increased reliability and endurance for BiCS, enabling BiCS cells to carry

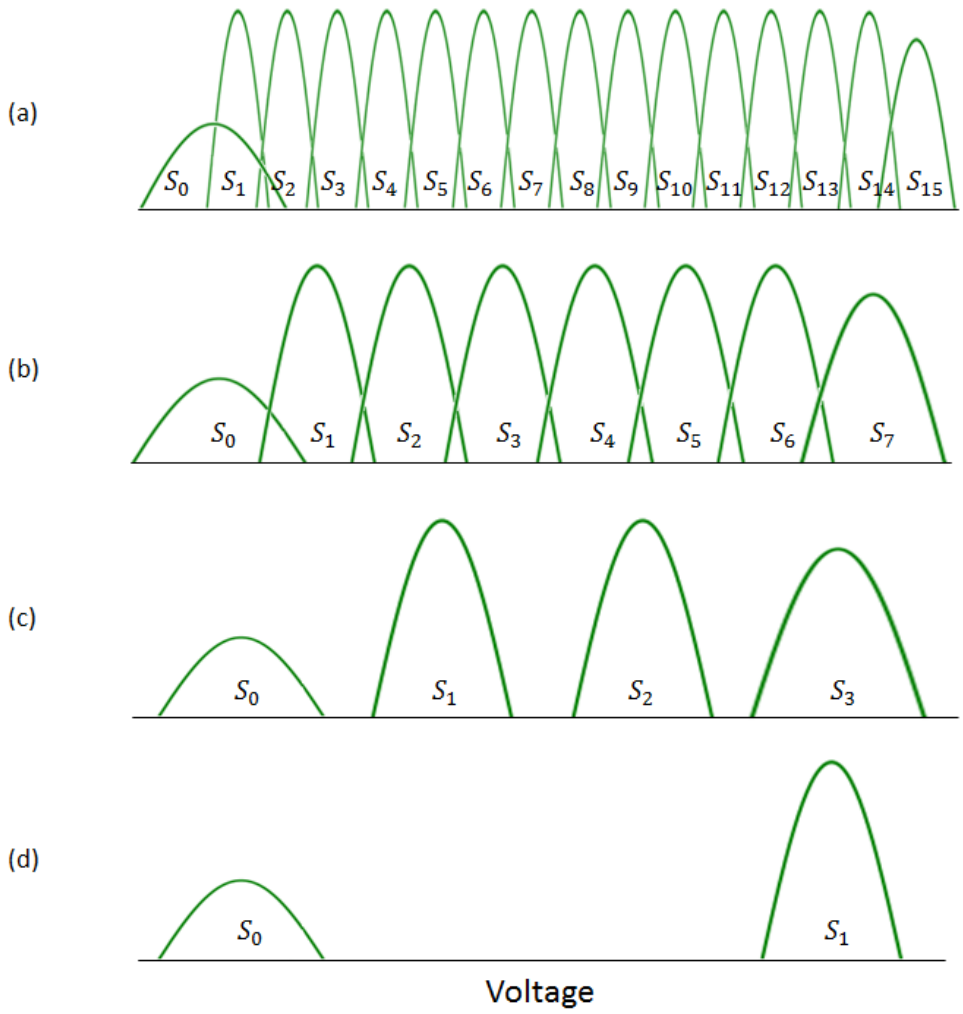


Figure 1-3: Simplified model of charge distributions in flash memory. (a) 16 levels of charge in QLC. (b) 8 levels of charge in TLC. (c) 4 levels of charge in MLC. (d) 2 levels of charge in SLC. Note that each distribution is Gaussian plotted on a log scale. Note S_0 and S_{15} have higher variance than the other distributions. Note also the x-axis is voltage, called the *Threshold Voltage* (V_t), and it is proportional to the stored charge.

3 bits in *TLC* cells and 4 bits in *QLC* cells. Therefore, BiCS improves the $bits/mm^2$ by carrying more information per cell and placing more cells per unit area.

Since the flash channel always has some raw BER, an *Error Correction Code (ECC)* layer needs to be added on top of the NAND to maintain the integrity of the data stored in flash memory (Section 2).

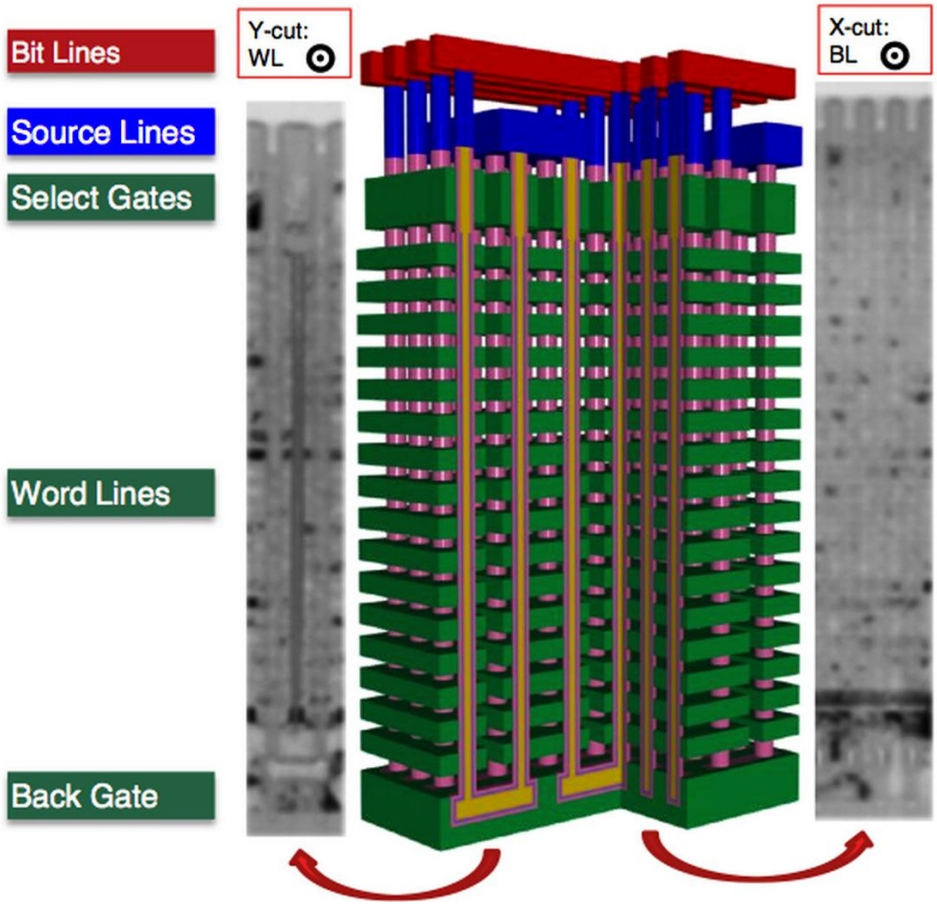


Figure 1-4: Bit Cost Scalable (BiCS) memory [1].

1.2 Channel Model and Channel Detector

The information stored in flash is encoded in analog voltage levels, called *Cell Voltages*, proportional to the charge carried by the flash cells. The observed cell voltage levels are shown in Figure 1-5. The flash cell introduces noise approximated as Additive White Gaussian Noise (AWGN).

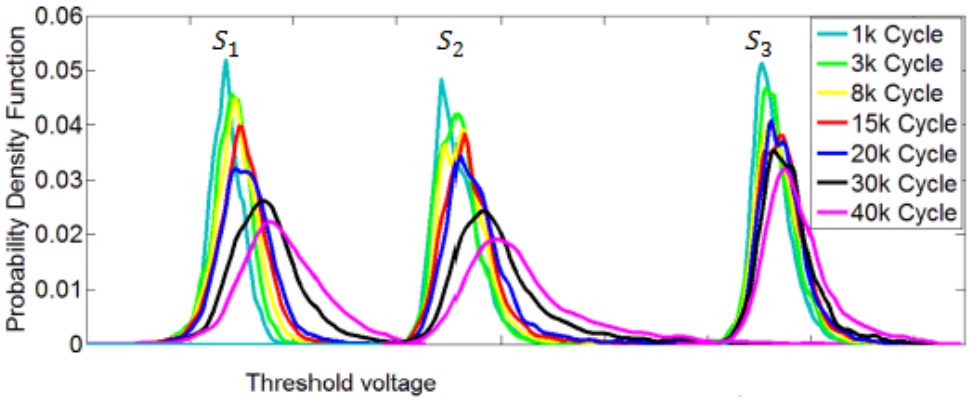


Figure 1-5: The distributions of NAND voltage levels collected from real hardware [2]. The figure shows the distributions after different P/E cycle points. Note the variance of a distribution increases with the number of P/E cycles. Note only three levels are shown in this figure.

The signal constellation used with the flash channel is Pulse-Amplitude Modulation (PAM). The AWGN noise associated with different symbols from the signal constellation has different variance, with the first and last symbols having the most noticeable difference (Figure 1-3). In QLC, every 4-bit string of user data in encoded into one of the 16-ary symbols stored in the flash channel. *Gray Code* is used for this encoding to minimize raw BER of the flash, as detailed in 1.2.1.

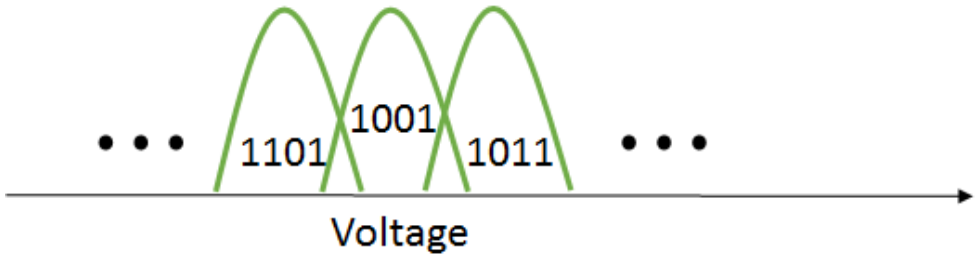


Figure 1-6: This figure shows three symbols of QLC flash. The Gray encoding guarantees one bit flip between the adjacent symbols to minimize the BER when a symbol is misread.

The means of the symbol distributions are not static but they move due to different effects during the lifetime of the NAND. One effect is the *Program Disturb (PD)*, in which programming cells will disturb the already programmed neighboring cells. The program disturb increases the variance of the levels and move

them to the right in the voltage space, as shown in Figure 1-7. Another effect is called *Data Retention (DR)*, which is a time effect where the variance of the levels increases and the means move towards some point near the zero voltage. This means the levels to the right of the point moves to the left and vice versa, as shown in Figure 1-8. The characterization of these effects depends on the NAND silicon and the fabrication process.

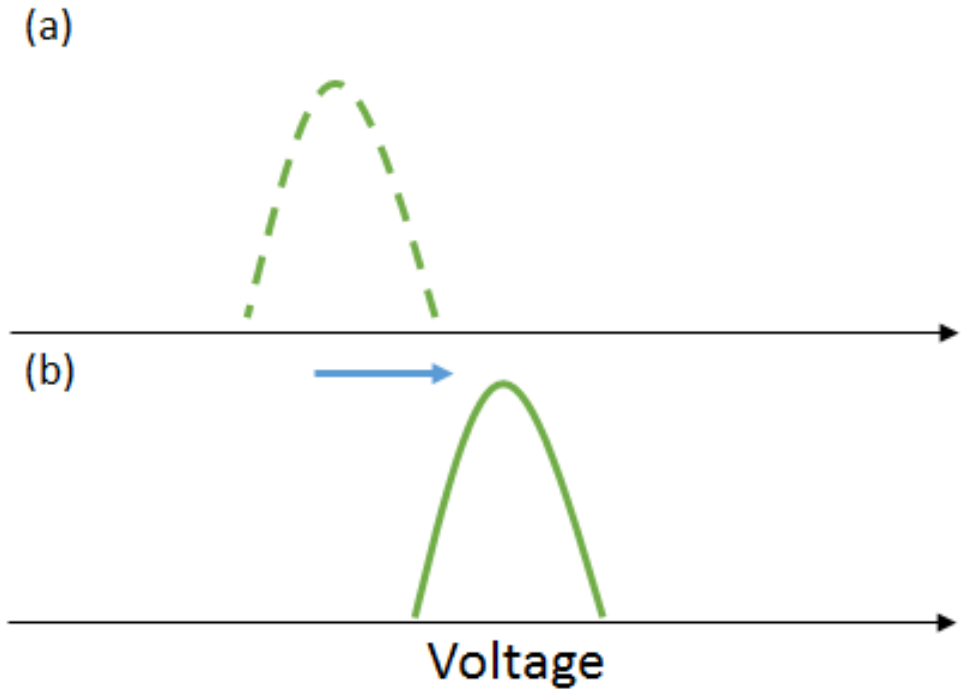


Figure 1-7: Program Disturb. (a) Before Program Disturb. (b) After Program Disturb. Program Disturb increases the voltage of the neighboring programmed cells.

1.2.1 Channel Detector

The information is read from the flash in discrete voltage levels, *Read Thresholds*. We set a number of these threshold voltages at the channel detector, and the detector returns the information if the flash cell voltage level is above or below these thresholds, as shown in Figure 1-9. This means the flash channel does not only depend on the flash physical characteristics, but also the place of the read thresholds. The positions of the thresholds are optimized to maximize the channel capacity

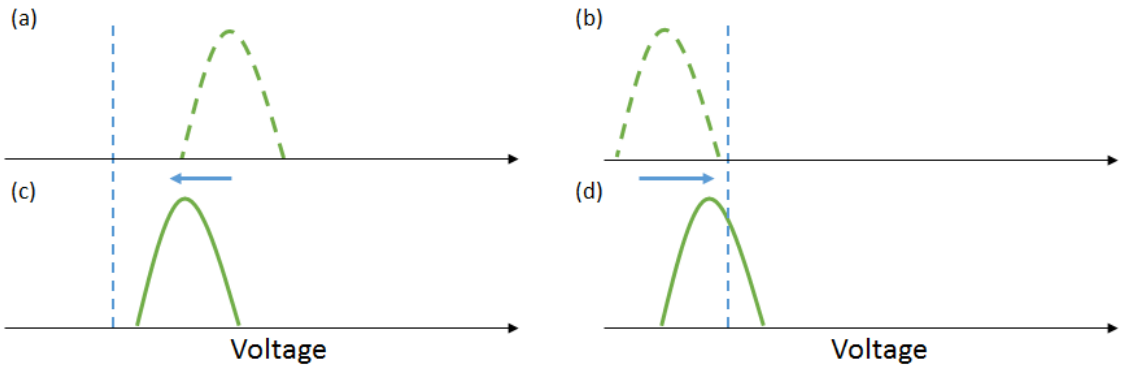


Figure 1-8: Data Retention. (a,b) Before Data Retention. (c,d) After Data Retention. Note the dashed line represents the point in the voltage space that the distribution move towards with data retention.

(section 3.3.2). When a certain symbol is written to the cell, but misread as the adjacent or second adjacent symbol, the number of user data bit flips is minimized by the Gray encoding, as shown in Figure 1-9.

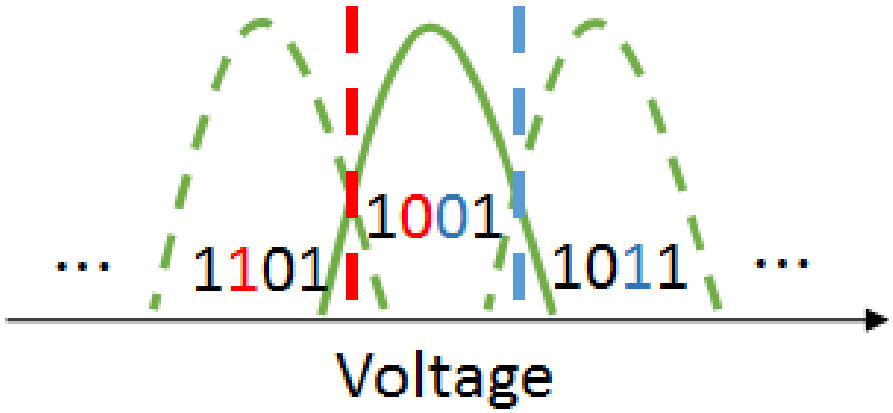


Figure 1-9: The channel detector decides the symbol transmitted is 1001 with high probability if the cell voltage is detected between the red and blue thresholds. If the cell is detected in the symbols to the right or left from the middle one, but 1001 was actually transmitted, then there will be a single bit flip only due to the Gray coding.

The Gaussian behavior of the flash memory is characterized by writing random data symbols and observing the analog voltage levels of the cells. The voltages are collected in a histogram that will converge to the channel probability distribution when the sample size is large and random (Glivenko-Cantelli lemma [9]). The probability model $P(y_i \text{ received} | x_i \text{ transmitted})$ is constructed based on this data

and the set read thresholds. $P(x_i \text{ transmitted}|y_i \text{ received})$ is also computed from the former model using Bayes' rule. The latter probability is the output of the channel detector and the input to the LDPC decoder 2.2.1. To clarify, observing the cell voltages is different from reading the cell with the read thresholds. The former is a reading mode over a continuous range of voltage and the latter results in discrete values depending on the read thresholds.

The channel detector of the cell passes a q-vector $(P(x_i = 0), P(x_i = 1), ..., P(x_i = q-1) \text{ transmitted}|y_i \text{ received})$, where q is the size of the symbol alphabet. To get more information from the channel, another read of the cell is issued but with slightly different threshold. This increases the resolution of detection, and allows the detector to give lower probabilities to the points detected close to the threshold between two distributions (Figure 3-11), as explained in section 3.3.5.

Chapter 2

Low-Density Parity-Check (LDPC) Codes

2.1 Error Correction Codes and Linear Block Codes

Information is encoded in bits and transmitted over a channel to a receiver. The problem is some bits could be modified by the channel. To maintain the integrity of the information, the data sent should have the information bits plus some redundant bits computed from the information bits. The purpose is that the redundancy helps recovering the bits modified by the channel.

The scheme of encoding these redundant bits is called *Error Correction Codes (ECC)*. There are different categories of ECC, and the one we are concerned with in this work is *Linear Block Codes*. They are defined over Galois Fields (finite fields) which are closed under addition and multiplication. $GF(q)$ denotes a Galois Field of order q, which is the size of its elements set. A Galois field exists if and only if it has an order that is prime number $q = p$, or a power of a prime number $q = p^n$, where $n \in \mathbf{Z}^+$

In this chapter, we explain binary ($GF(2)$) ECC concepts first, including LDPC, in Sections (2.1 - 2.3), then we introduce non-binary ($GF(q)$) LDPC in the last section (Section 2.4).

In binary linear block codes, a group of size k bits of information, called *information bits*, is encoded into a block, hence the name, of size n bits of data. This block is called a *codeword*. The extra $m = n - k$ bits are called *parity bits*. Each parity bit is computed by XORing, i.e. addition over a binary field, a number of information bits. The coderate (r) is defined as $(r = \frac{k}{n})$.

For example, consider a codeword of length $n = 7$, and $k = 4$, and let p_i denotes the i-th parity bit. In this example, there are 3 parity bits, and let the example code constrain them this way:

$$p_0 = x_0 + x_1 + x_3$$

$$p_1 = x_0 + x_2 + x_3$$

$$p_2 = x_0 + x_1 + x_2$$

Note the addition is over *GF(2)*. So we take 4 information bits and encode them into a codeword of 7 bits. If we start with 1101, then 1101100 is the codeword that satisfies the code in our example.

The ECC is called *Systematic Code* when the codeword consists of information bits and appended to them are the parity bits. The ECC codes do not have to be systematic, and the parity bits could be placed non-contiguously anywhere in the codeword. For implementation simplicity, systematic codes are the most popular in practical systems, including flash storage systems [10].

A codeword that belongs to a code must satisfy all the parity check equations of that code. We write the parity equations in a form where one hand side is zero and all the other non-zero terms are on the other side. This form is more suitable for linear algebra and matrix operations. In this form, the parity check equations of our example will be:

$$p_0 + x_0 + x_1 + x_3 = 0$$

$$p_1 + x_0 + x_2 + x_3 = 0$$

$$p_2 + x_0 + x_1 + x_2 = 0$$

The magnitude on the left-hand-side of these equations is called, the *syndrome*, and the parity-check is satisfied if the syndrome equals to zero. We represent a codeword by a row vector x of size $1 \times n$. The linear block code is defined by a *Parity Check Matrix (H)* of size $m \times n$ where each row represents a parity check equation. $H_{ij} = 1$ if the j-th bit in the codeword is present in the i-th parity check equation, and $H_{ij} = 0$ otherwise. The coderate of this matrix is $r = \frac{n-m}{n}$. The code has $(k = n - m)$ degrees of freedom, so it has 2^{nr} codewords belong to it. The H-matrix of our example is:

$$H = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Let $\mathbf{C}$ be a code with $H_{\mathbf{C}}$, and Let x be a codeword. x is a valid codeword iff:

$$H_{\mathbf{C}} x^T = 0 \tag{2.1}$$

Assume $x \in \mathbf{C}$ and $y \in \mathbf{C}$. Then:

$$\begin{matrix} \mathbb{H}_{\mathbf{C}} x^T = 0 \\ H_{\mathbf{C}} y^T = 0 \end{matrix} \qquad \Rightarrow \qquad H_{\mathbf{C}}(x^T + y^T) = 0$$

Therefore, $x^T + y^T$ is a valid codeword, and the all-zero codeword $\mathbf{0}$ is also a valid codeword. This means every linear combination of valid codewords in a code is also a valid codeword in that code, the reason why these codes are called linear.

The main benefit of Linear Block Codes is their efficient implementation in practical systems, as they take less memory to store than other codes [11]. A general code with length n and coderate r takes $n2^{nr}$ bits of memory. However, with the linear structure, the code could be defined by a matrix H taking nm bits only. We use a special and widely popular type of Linear Block Codes in this work.

These are called *Low-Density Parity-Check (LDPC)* codes. There are only few 1's in each row and each column of the code parity check matrix (H). In other words, the matrix is sparse or low-density. The LDPC codes reduces the decoding complexity [12], and performs better with the belief propagation decoding algorithm as explained in Section 2.2.1. LDPC codes approaches the channel capacity asymptotically [12]. Refer to Section 3.3.2 for the channel capacity.

2.1.1 Minimum Distance

The Hamming distance $D(x, y)$ between two codewords x and y is the number of bits with different values between x and y. The important quantity is the *Minimum Distance* of a certain code (d), which is the lowest Hamming distance between two codewords over the entire range of codewords of that code. The larger the minimum distance, the more bits in a codeword could be flipped in transmission and corrected by the code at the receiver, as the transmitted codeword will converge to the closest valid codeword. If the minimum distance is small, the received codeword could decode to a different codeword from the one transmitted. We define the weight $w(x)$ of a codeword x as the number of $1's$ in x. The minimal weight of a code is the weight of the codeword of lowest non-zero weight. The minimum distance of a linear code is the minimal weight of the code. To see this, let x and y be two codewords in a linear code $\mathbf{C}$ and let the Hamming distance $d(x, y)$ between them be the minimum distance of the code. Then:

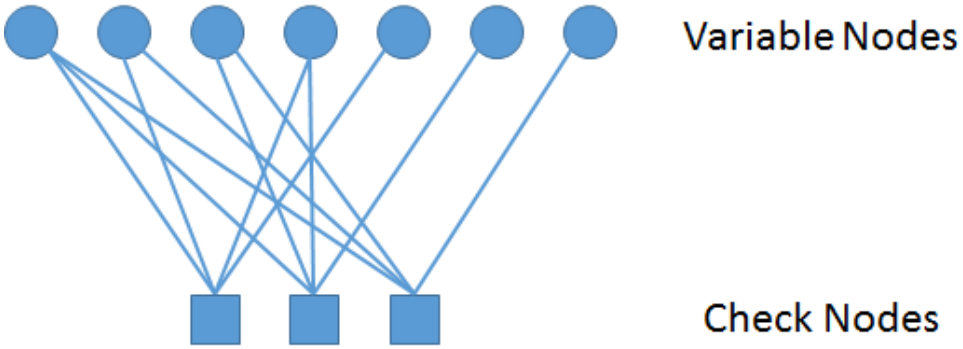
$$d(x, y) = w(x - y) \qquad \text{(definition of Hamming distance)}$$

But $(x - y) \in \mathbf{C}$, since $\mathbf{C}$ is linear, and $w(x - y) = d(x - y, 0)$. Therefore:

$$d(x, y) = d(x - y, 0)$$

2.1.2 Tanner Graph Representation

A linear block code with H_C of size $m \times n$ is represented by a bipartite graph, called the *Tanner Graph*, with n *Variable Nodes* and m *Check Nodes*. Each variable node corresponds to a single bit in the code, and each check node corresponds to a parity check constraint and is connected to the variable nodes of the that parity check. Therefore, H_C is the *Adjacency Matrix* of the graph. The graph representation is useful to study linear block codes and their properties under belief propagation decoding, as we will see in Section 2.2.1. The Tanner graph of our example code in Section 2.1 is shown in Figure 2-1(b).



(a)

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(b)

Figure 2-1: (a) The Tanner graph of our example code. (b) The parity check matrix of our example code

The degree of a node is defined as the number of edges connected to the node. In a certain code, if all check nodes have the same degree, and all variable nodes have the same degree, then the code is called *Regular*. Otherwise, it is called *Irregular*. For a regular code, we denote by d_c its check degree, also called *row weight*,

and by d_v its variable degree, also called *column weight*. Irregular LDPC codes have higher error correction power than regular ones. However, this difference becomes insignificant in high coderate codes. In this work, we use a regular LDPC with coderate $r = 0.9$.

2.2 Decoding LDPC code

We explain the decoding problem on data transmitted over a *Binary Symmetric Channel (BSC)*. This makes the decoding problem simpler to explain than using other channels, and the solution generalizes to other channel models. We denote by $BSC(p)$ a binary symmetric channel of parameter $p \leq 0.5$. This parameter is the bit flip probability of transmission across the channel. Figure 2-2 shows a diagram of the BSC channel.

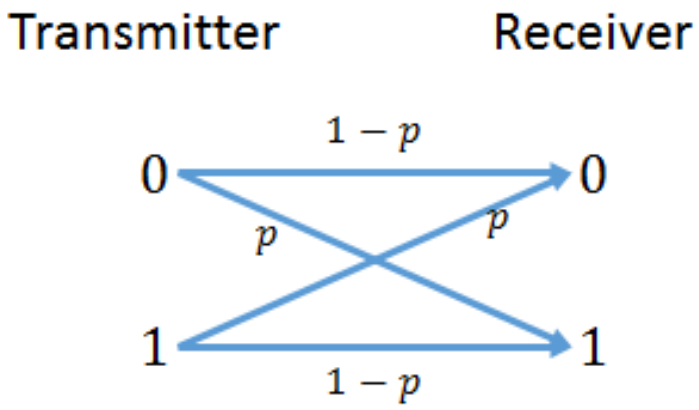


Figure 2-2: BSC(p) Binary Symmetric Channel with parameter (p).

Let us go back to the decoding problem, let the codeword x of block length n be transmitted over a $BSC(p)$, and let y be the received codeword. The decoding question is: what is x given y is observed? The natural answer is the most likely x' given y and the channel model. Mathematically speaking, this is the codeword $x' = x^{MAP}$ that maximizes the *Maximum A Posteriori (MAP)* distribution of all codewords $x' \in \mathbf{C}$.

$$
\begin{aligned}
x^{MAP} &= \underset{x' \in \mathbf{C}}{\operatorname{argmax}} \mathbf{P}(x'|y) \\
&= \underset{x' \in \mathbf{C}}{\operatorname{argmax}} \frac{\mathbf{P}(y|x')\mathbf{P}(x')}{\mathbf{P}(y)}
\end{aligned}
\tag{2.2}
$$

And:

$$
\mathbf{P}(y|x') = p^{d(y,x')}(1-p)^{n-d(y,x')} \tag{2.3}
$$

We know that $\mathbf{P}(x')$ is a constant when the codeword transmitted is randomly selected (i.e. uniform distribution), and $P(y)$ is a constant for a certain y. Therefore, the decoding problem reduces to selecting $x^{MAP} = x'$ that maximizes $\mathbf{P}(y|x')$, which is the one closest in distance to y. Note this computation requires iterating over all the codewords $\in \mathbf{C}$, which is slow and complex to implement in a practical system. In Section 2.2.1, we explain belief propagation decoding algorithms that are sub-optimal to MAP decoding, but have lower complexity, making them practical for implementation.

Note that $x^{MAP} \neq x$ if y is closer in distance to another codeword $\in \mathbb{C}$. In this case, this is called *Undetected Error*. In practical systems, the codeword contains *Cyclic Redundancy Check (CRC)* which is a group of bits computed as a hash function of the rest of the codeword. After the codeword is decoded, the hash function is computed for the decoded codeword to verify if it the transmitted one or not.

2.2.1 Belief Propagation and the Sum-Product Algorithm

Belief propagation reduces the complexity of MAP computation over a high-dimensional space by performing local computations at the check and variable nodes. Each node computes probability messages and exchanges them with the neighboring nodes. These messages are used to compute the bits of the decoded codeword at the variable nodes [13].

The probability message considered in this work, and most widely used in research and practice, is *Log-Likelihood Ratio (LLR)*, defined as:

$$LLR(x) = \ln \frac{P(x=0)}{P(x=1)}$$

Where $x \in \{0,1\}$ is a random variable. ln denote the natural logarithm.

Before describing the steps of the belief propagation algorithm. We introduce some notations. Let q_i be the belief of variable node v_i, q_{ij} be the message from variable node v_i to check node c_j and r_{ij} be the message from check node c_i to variable node v_j. A superscript symbol t on the message quantities, q_{ij}^t, r_{ij}^t, and q_i^t, denotes the message at the t-th iteration. Let V_i denotes the set of indices of check neighbors to variable node v_i, and C_i the set of indices of variable neighbors to check node c_i. Let also ch_i denotes the i-th channel node that carries the received bit $LLR(x_i|y_i)$ (Figure 2-3 and 2-4).

Consider an $m \times n$ code $\mathbf{C}$. Let the n-sized data string x be transmitted over some channel, and received as y. The m-sized syndrome vector s of x is computed based on the code and given as an input to the belief propagation algorithm. The flow of the belief propagation algorithm to decode y is as follows:

1- **Variable Node Message Initialization:** Each variable node v_i initializes its outgoing messages q_{ij}^0 to its neighboring checks c_j's as:

$$q_{ij}^0 = LLR(x_i|y_i), \qquad \forall i \in \{1,...,n\}, j \in V_i \tag{2.4}$$

This is the channel message transmitted from ch_i to v_i (Figure 2-3). Note that $P(x_i|y_i)$, thus $LLR(x_i|y_i)$, is based on the channel model.

2- **Variable Node Message Computation:** Each variable node v_i computes its outgoing messages q_{ij} to its neighboring checks c_j's as:

$$q_{ij}^t = LLR(x_i|y_i) + \sum_{j' \in V_i / \{j\}} r_{j'i}^{t-1}, \qquad \forall i \in \{1,...,n\}, j \in V_i \tag{2.5}$$

In other words, the variable node message depends on the messages it receives from the channel and the neighboring check nodes except the one it is transmitting

to (Figure 2-3).

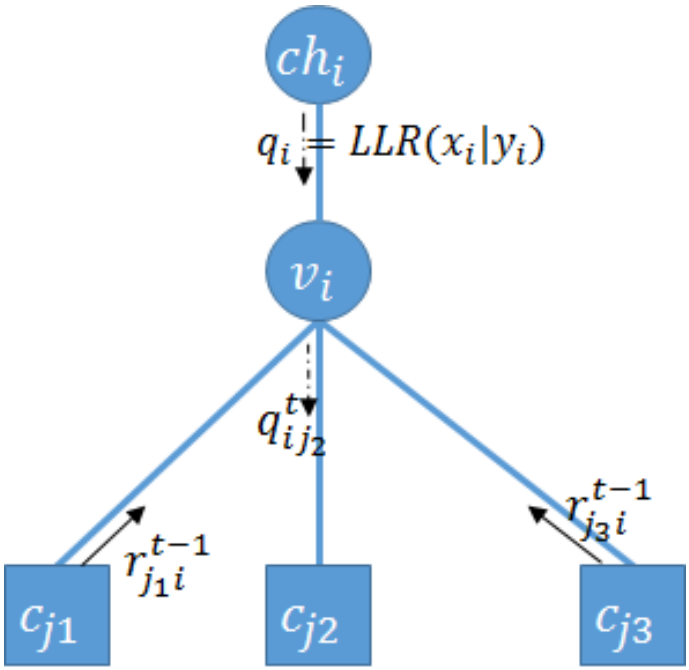


Figure 2-3: Variable node processing. The message $q_{ij_2}^t$ is the variable node v_i message to check node c_{j_2} at iteration t. $q_{ij_2}^t$ depends on the messages from the channel and from the neighboring check nodes to v_i excluding the check node transmitted to c_{j_2} at the previous iteration $t-1$. Note $V_i = \{j_1, j_2, j_3\}$.

3- **Check Node Message Computation:** Each check node c_i computes its outgoing messages r_{ij} to its neighboring variable nodes v_j's (Figure 2-4) as (see [14] for derivation):

$$r_{ij}^t = 2s_i \prod_{j' \in C_i / \{j\}} \tanh\left(\frac{1}{2} q_{j'i}^{t-1}\right), \qquad \forall i \in \{1, ..., m\}, j \in C_i \tag{2.6}$$

where:

$$s_i = \begin{cases} 1 & \text{if } c_i \text{ syndrome is 0} \\ -1 & \text{if } c_i \text{ syndrome is 1} \end{cases}$$

This is the check node message biased depending on the syndrome. Note that passing the syndrome vector is only possoible for code simulation. In a practi-

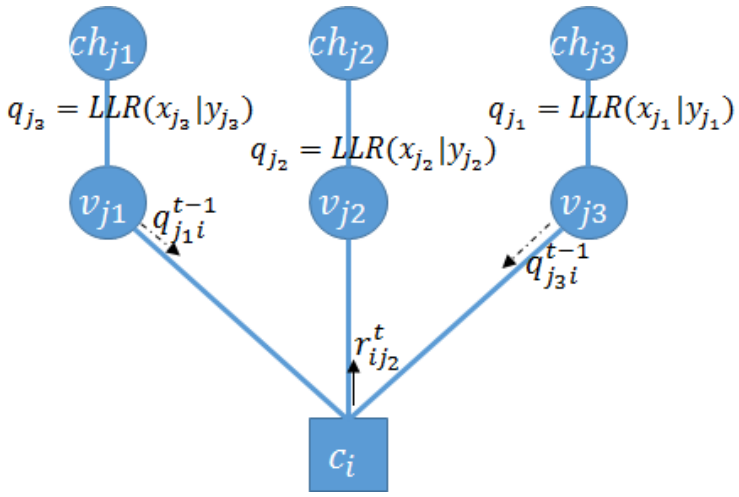


Figure 2-4: Check node processing. The message $r_{ij_2}^t$ is the check node c_i message to variable node v_{j_2} at iteration t. $r_{ij_2}^t$ depends on the messages from the neighboring variable nodes to c_i excluding the variable node transmitted to v_{j_2} at the previous iteration $t-1$. Note $C_i = \{j_1, j_2, j_3\}$.

cal system, this vector cannot be reliably transmitted over the channel. Instead the data string transmitted is constrained by the code into a codeword such that the syndrome vector is the all-zero vector. At the decoder, all the check nodes are biased to the zero syndrome. In simulation, we compute the syndrome of a data string given the H-matrix rather than generate codewords. The latter requires finding the generating matrix of H with matrix Gaussian elimination.

4- **Variable Node Belief Computation and Bit Decision:** Each variable node v_i computes its belief q_i as:

$$q_i = LLR(x_i|y_i) + \sum_{j' \in V_i} r_{j'i}, \qquad \forall i \in \{1, ..., n\}, j \in V_i \tag{2.7}$$

$$x_i^{'} = \begin{cases} 0 & \text{if } q_i > 0 \\ 1 & \text{if } q_i < 0 \end{cases}$$

Where $x_i^{'}$ is the i-th decoded bit.

Note that the exchanged messages are all in the log-domain described above.

Note also we decide the bit is zero when its associated belief is positive and vice versa. This has to do with the way we defined the LLR. Using the LLR domain results in simpler implementation that uses adders instead of multipliers if the belief propagation was done in the probability domain. In addition, digital circuit implementation uses fixed-point arithmetic where decoding in the LLR domain results in better error correction power [15].

After step 4 is finished, the decoded codeword is usually checked if it is valid $Hx^{'T} = 0$ in which case the algorithm terminates. Otherwise, another iteration through steps 2-4 is performed. A maximum number of iterations is specified, after which the decoding is stopped and failure to decode y is declared. Otherwise, if no maximum number of iterations is specified, the algorithm could run forever. The belief propagation algorithm described above is called the *Sum-Product Algorithm (SPA)* [13], and it is the algorithm we use in our experiment 3. Variants of this algorithm, such as min-sum, are used in research and industry. These variants explore different tradeoffs, ranging between error correction performance, complexity, and speed. In fact, the min-sum algorithm is the one most commonly implemented in flash storage systems [16].

For the belief propagation algorithm described above to be equivalent to MAP-decoding, it requires in every node computation, the neighboring messages events are independent. After few decoding iterations, this is no longer the case, since the LDPC graph always contains cycles [12]. The events at different nodes will be correlated due to circulating messages between nodes via cycles and throughout the decoding iterations. Therefore, the longer the shortest cycle, called the *girth*, of a code is, the better it performs with belief propagation algorithms. Due to their sparsity, deeper trees can be extended from the nodes of LDPC codes compared to denser linear codes, where a tree is a graph structure with no cycles. This makes LDPC perform better with belief propagation than the other linear codes [12].

2.3 Quasi-Cyclic LDPC Codes QC-LDPC

Quasi-Cyclic codes have a structure that enables decoding parallelism in digital implementations. The parity check matrix H of a quasi-cyclic code consists of smaller submatrices, called *Circulants*, as shown in Figure 2-5. A circulant could be the all-zero matrix $\mathbf{0}$ or a cyclically permuted identity matrix [17]. A cyclically permuted identity matrix I_k of size $Z \times Z$ is an identity matrix, but with every row shifted to the right by k. In other words, if a_{ij} is an entry of I_k, then:

$$a_{ij} = 1 \quad iff \quad j \equiv (i+k) \mod Z, \quad for \quad 0 \leq i,j \leq Z-1$$

This is an example for I_2 of size 7×7:

$$I_2 = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

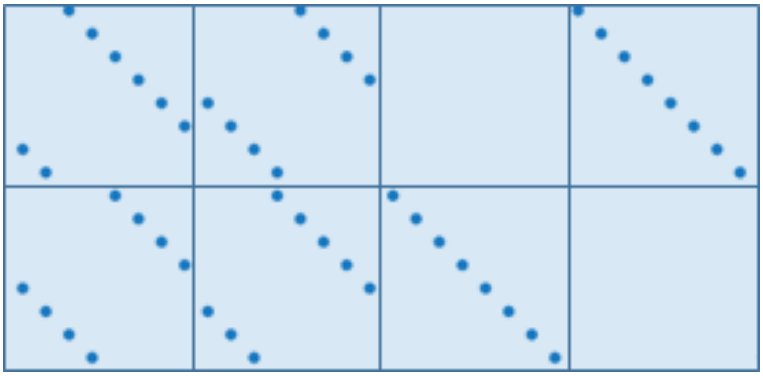


Figure 2-5: Quasi-cyclic matrix of size 16×32 with circulant size 8. Note the all-zero circulants and cyclically permuted identity matrices.

Note the identity matrix is a circulant matrix with zero-shift $I = I_0$. For a quasi-cyclic regular (d_v,d_c)-code with circulant size Z, the parity check matrix H is of size

$m \times n$, where $m = d_v Z$ and $n = d_c Z$,

$$H = \begin{pmatrix} \mathbf{P_{0,0}} & \mathbf{P_{0,1}} & \cdots & \mathbf{P_{0,d_c-1}} \\ \mathbf{P_{1,0}} & \mathbf{P_{1,1}} & \cdots & \mathbf{P_{1,d_c-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{P_{d_v-1,0}} & \mathbf{P_{d_v-1,1}} & \cdots & \mathbf{P_{d_v-1,d_c-1}} \end{pmatrix}$$

Where $\mathbf{P_{i,j}} = I_l$ is a $Z \times Z$ matrix, $i \in \{0, 1, ..., d_v - 1\}$, $j \in \{0, 1, ..., d_c - 1\}$, and $l \in \{0, 1, ..., Z - 1\}$, or $\mathbf{P_{i,j}} = \mathbf{0}$ (all-zero matrix). An example of 16×32 quasi-cyclic matrix of circulant size 8 is shown in Figure 2-5.

Quasi-cyclic codes take less memory to store by storing each circulant permutation only. The messages computed at a certain step of belief propagation (Section 2.2.1) are stored in memory, then fetched in the next step that depends on the messages from the previous one. If the messages are stored in a single memory block, then reading and writing messages will be a bottleneck because it happens sequentially due to the address-based architecture of the memory. To enable parallelism, QC-codes are used with multiple memory blocks corresponding to circulants. Each node fetches the messages from multiple neighbors at the same time, as they are stored in multiple blocks [18].

2.3.1 Quasi-Cyclic Code Construction

To construct QC-codes, we start from a small Tanner graph, called a *Protograph* [17]. We lift this protograph into the desired QC-LDPC graph. The process of lifting includes copying this protograph Z times (Figure 2-6), where Z is the *Lifting Factor*, which is also the circulant size of the constructed code. Copying the protograph means copying the nodes and the edges. The copies of a single edge is called an *Edge Group*. Next, we permute the edges in each edge group, resulting in the cyclically permuted identity matrices introduced in Section 2.3. Figure 2-6 illustrates the lifting process.

As mentioned in Section 2.2.1, our goal of LDPC code design is to maximize

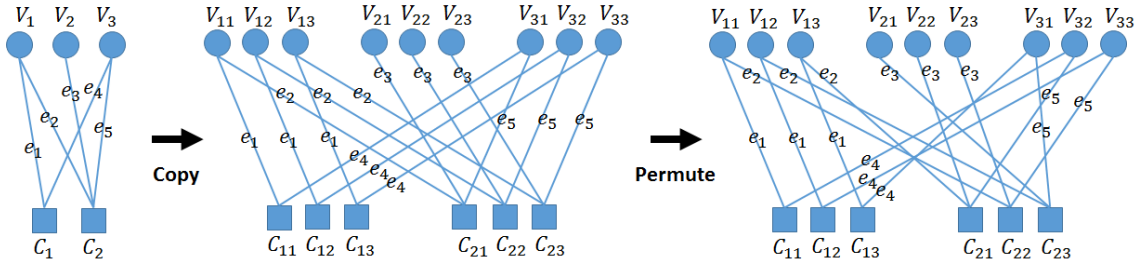


Figure 2-6: Protograph lifting. Starting from the protograph on the left, which is copied, then the edges are permuted. This graph has Z = 3 and 6×9 H-matrix.

the girth of the code. We define the *girth* as the length of the shortest cycle in the graph. A $2l$-cycle in a graph could be associated with a sequence of circulants and their permutation matrices, as:

$$\mathbf{P}_{\mathbf{i}_0,\mathbf{j}_0}, \mathbf{P}_{\mathbf{i}_0,\mathbf{j}_1}, \mathbf{P}_{\mathbf{i}_1,\mathbf{j}_1}, ..., \mathbf{P}_{\mathbf{i}_{l-1},\mathbf{j}_{l-1}}, \mathbf{P}_{\mathbf{i}_{l-1},\mathbf{j}_0} \tag{2.8}$$

For $1 \leq k \leq l-1$, $i_k \neq i_{k-1}$ and $j_k \neq j_{k-1}$. Also, $i_{l-1} \neq i_0$ and $j_{l-1} \neq j_0$. Other than these conditions, the permutation matrices in the sequence could be repeated more than once, as a cycle could traverse a circulant more than once. Let us use $\phi(\mathbf{P}_{\mathbf{i},\mathbf{j}})$ to denote the cyclic shift to the left associated with $\mathbf{P}_{\mathbf{i},\mathbf{j}}$.

A necessary and sufficient condition for the existence of a $2l$-cycle [19] [20] [21] is:

$$\sum_{k=0}^{l-1} (\phi(\mathbf{P}_{\mathbf{i_k},\mathbf{j_k}}) - \phi(\mathbf{P}_{\mathbf{i_{k+1}},\mathbf{j_k}})) \equiv 0 \mod Z \tag{2.9}$$

Note $\mathbf{P}_{\mathbf{i_l},\mathbf{j_l}} = \mathbf{P}_{\mathbf{i_0},\mathbf{j_0}}$. Therefore, for an $m \times n$ graph with girth $\geq 2(l+1)$, and circulant size Z, we need:

$$\sum_{k=0}^{l-1} (\phi(\mathbf{P}_{\mathbf{i_k},\mathbf{j_k}}) - \phi(\mathbf{P}_{\mathbf{i_{k+1}},\mathbf{j_k}})) \not\equiv 0 \mod Z, \quad \forall \quad 0 \leq i_k \leq \frac{m}{Z} - 1, \quad 0 \leq j_k \leq \frac{n}{Z} - 1 \tag{2.10}$$

The higher Z is, the easier it is to satisfy 2.10 for a given girth. Note it is very computationally-intensive to iterate through all the circulants to make sure is 2.10

satisfied and find the minimum circulant size to achieve a certain girth. In the next section, we introduce a practical method for constructing QC-codes with high girth.

2.3.2 Circulant Progressive Edge Growth (CPEG)

As we saw in the previous section, it is computationally hard to choose the circulant permutations to maximize the girth. Instead, we use another method based on a greedy algorithm, called *Circulant Progressive Edge Growth (CPEG)*. The method is sub-optimal, but it is computationally practical [22].

Before introducing the algorithm, we introduce some notation. For an LDPC code with matrix H, let $N_{v_i}^l$ denotes the set of all the check nodes in the tree rooted at variable node v_i and extended to depth l. Its complementary set $\overline{N_{v_i}^l}$ is the set of all the check nodes in the graph except $N_{v_i}^l$ (Figure 2-7).

The input to the CPEG is the number of variable nodes t and check nodes r, a degree profile (d_v, d_c) and lifting factor Z. You can think of this input as an $r \times t$ protograph with degree profile (d_v, d_c), but without specified edges. The output is a graph $m \times n$, where $m = Zr$ and $n = Zt$. CPEG chooses the edges in the lifted $m \times n$ graph with the goal of large girth. The algorithm is as follows [23]:

1: **for** $i = 0 : t - 1$ **do**
2: **for** $j = 0 : d_{v_i} - 1$ **do**
3: **if** $j == 0$ **then**
4: $E_{iZ}^0 : (c_k, v_{iZ})$, where c_k is a randomly selected check node from the lowest degree check nodes in the current state of the graph.
5: **for** l = 1:Z-1 **do**
6: $E_{iZ+l}^l : (c_{Z(k/Z)+mod(k+l,Z)}, v_{iZ+l})$
7: **end for**
8: **else**
9: Extend a tree from v_{iZ} up to some depth L such that:

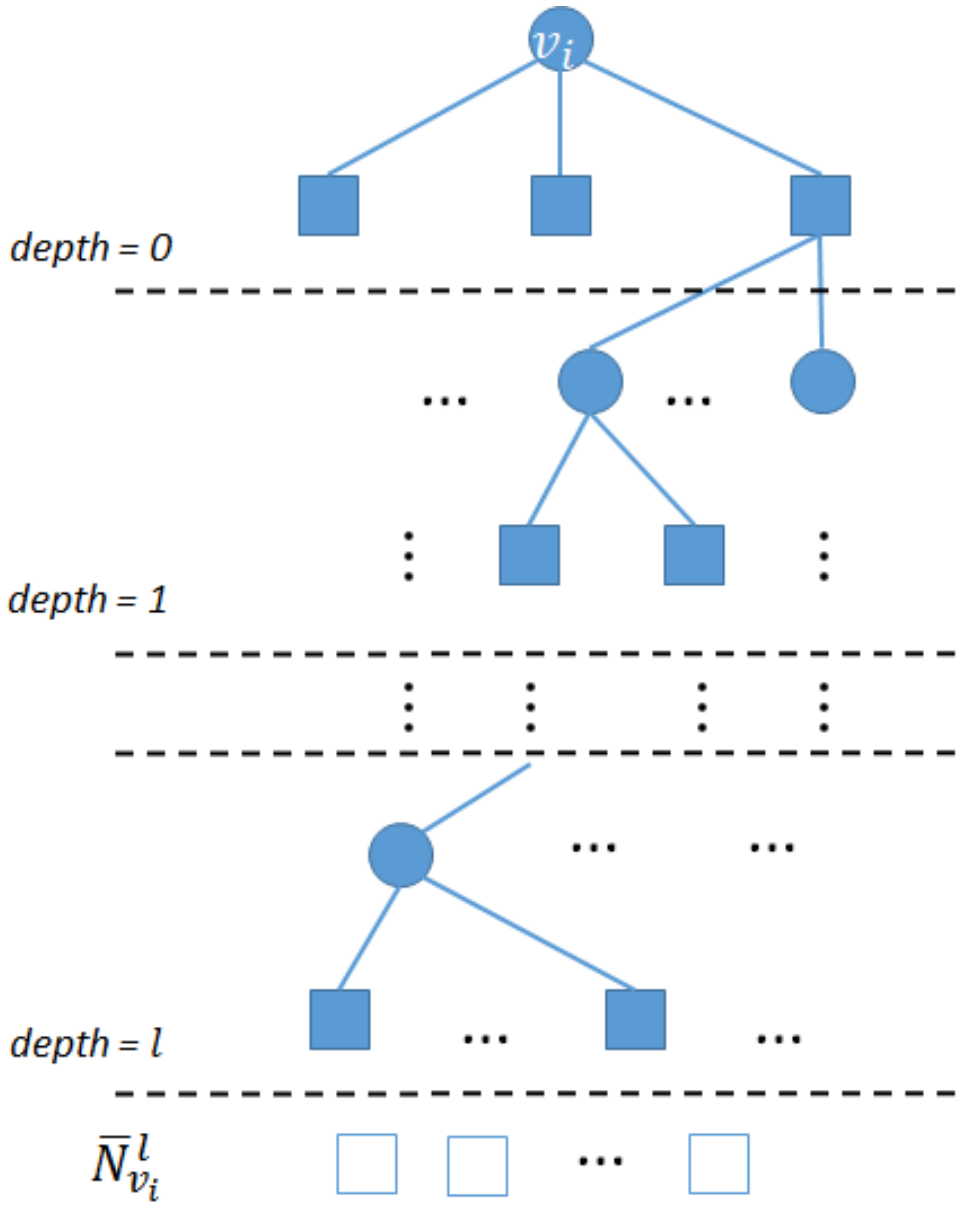


Figure 2-7: This figure shows the tree expanded from v_i to depth l. The unshaded squares represent the check nodes in the LDPC graph that are not within the l-deep tree extended from v_i.

either: $\overline{N_{v_{iZ}}^{L}} \neq \phi$, but $N_{v_{iZ}}^{L+1} = \phi$

or: the cardinality of $N_{v_{iZ}}^{L}$ stops increasing and is smaller than m the cardinality of the set of all check nodes.

Choose c_k as a randomly selected check node among the smallest degree nodes $\in \overline{N_{v_{iZ}}^{L}}$

10: **if** current degree of $c_k < d_{c_k}$ **then**

11: $E_{iZ}^{j} : (c_k, v_{iZ})$

12: **for** l = 1:Z-1 **do**

13: $E_{iZ+l}^{l} : (c_{Z(k/Z)+mod(k+l,Z)}, v_{iZ+l})$

14: **end for**

15: **else**

16: Delete $E_{iZ}^{0}, ..., E_{(i+1)Z-1}^{0}$ and go to step (4)

17: **end if**

18: **end if**

19: **end for**

20: **end for**

Where $E_{v_i}^{t} : (c_j, v_i)$ denotes an edge between c_j and v_i and this edge is the $t - th$ incident edge on v_i in the order of CPEG progress. Basically, the algorithm iterates over all the variables in the input protograph (line 1), then adds edges to each variable based on the input d_v. The lifted graph is considered when adding edges to these nodes with each variable node being the first in its circulant column. There are two cases in assigning these edges. The first case is when the edge is the first one (line 3-4) and assigned to a randomly selected check node from the lowest degree check nodes in the lifted graph. The second case deals with edges added after the first edge (line 9). After every edge added to the first variable node in a circulant, an edge is added between the variable nodes and check nodes in the rest of the circulant separately, and in a circular fashion (lines 5-7 and 12-14).

The variable degree profile of the resultant graph is guaranteed by the fact that the edges assignment in the algorithm is guided by this degree profile. In (line 10), the check nodes degree profile d_c constraint is checked to make sure it is satisfied.

Note this step is dropped in some variants of the CPEG algorithm where the check nodes degree profile is not constrained.

Note d_v and d_c are vectors of sizes t and r respectively, and d_{v_i} and d_{c_j} denotes the i-th variable node degree and the j-th check node degree respectively.

2.4 Non-Binary LDPC (NB-LDPC)

LDPC codes could be defined over a Galois field of any order, and the codeword transmitted consists of symbols over that field. What we have seen so far are codes over GF(2) only, or binary codes, where we call the symbols transmitted, bits, in this case. The LDPC code defined over $GF(q)$ where $q > 2$ is called, *Non-Binary LDPC Code (NB-LDPC)*. Let **C** be a NB-LDPC code over $GF(q)$, and codeword $x \in$ **C**. The parity check matrix $H_{m \times n}$ of **C** consists of entries in $GF(q)$. Each row is a parity check equation:

$$\sum_{\substack{j=0 \\ a_{ij} \neq 0}}^{n-1} a_{ij} x_j = 0, \quad \forall \quad i \in \{0, 1, ..., m-1\}$$

The parity check equation is a linear combination of codeword symbols weighted by the H-matrix entries. Note we do not write the weights in the binary parity check, as they are all 1's, which is the identity of the multiplication operation. An example of a small H-matrix over $GF(5)$ is:

$$H = \begin{pmatrix} 1 & 2 & 0 & 4 & 1 & 0 & 0 \\ 3 & 0 & 2 & 1 & 0 & 4 & 0 \\ 2 & 2 & 1 & 0 & 0 & 0 & 3 \end{pmatrix}$$

2.4.1 Belief Propagation with NB-LDPC

The concept of exchanging belief messages between graph nodes to reinforce or undermine certain bits (or symbols) of the codeword in binary decoding 2.2.1 is also the basis of the NB-LDPC belief propagation. However, there are differences

in the message content and the node equations to serve the purpose of multi-symbol decoding.

Let us consider decoding LDPC code over $GF(q)$. First, we present the general algorithm with the multiple vector convolution, then we present a trick of partial sums to implement this convolution. The straightforward convolution has a complexity of $O(q^{d_c})$ per check node, where d_c is the degree of that node. $d_c = 30$ The code we design for the experiment in Section 3.4. The partial sums technique reduces the complexity to $O(q^2)$.

In non-binary belief propagation, the messages exchanged are q-tuples of probability $(P(x = 0), P(x = 1), ..., P(x = q - 1))$. Note this tuple has one redundant entry, as the probabilities add up to 1, but we keep it this way to simplify the implementation, especially the convolution as we will see.

Let $x = (x_1, x_2, ..., x_n)$ be the data string transmitted and $y = (y_1, y_2, ..., y_n)$ is the received one, where x and y are over $GF(q)$. The syndrome vector s is given to the algorithm too. We use a similar notation for the messages as in 2.2.1, but with a modified superscript. For instance, $q_{ij}^{l,(a)}$ denotes the message from v_i to c_j holding the probability of symbol $a \in GF(q)$ at the l-th iteration. We describe the steps of the algorithm:

1- **Variable Node Message Initialization:** Each variable node v_i initializes its outgoing messages q_{ij}^0 to its neighboring checks c_j's as:

$$q_{ij}^0 = (P(x_i = 0), P(x_i = 1), ..., P(x_i = q - 1)|y_i), \qquad \forall i \in \{1, ..., n\}, j \in V_i \quad (2.11)$$

This is the channel message transmitted from ch_i to v_i (Figure 2-8). Note that $(P(x_i = 0), P(x_i = 1), ..., P(x_i = q - 1)|y_i)$ is based on the channel model.

2- **Variable Node Message Computation:** Each variable node v_i computes its outgoing messages q_{ij} to its neighboring checks c_j's as:

$$q_{ij}^{l,(a)} = P(x_i = a|y_i) \prod_{j' \in V_i/\{j\}} rp_{j'i}^{l-1,(a)}, \qquad \forall i \in \{1,...,n\}, j \in V_i \tag{2.12}$$

In other words, the variable node message depends on the messages it receives from the channel and the neighboring check nodes except the one it is transmitting to, as shown in Figure 2-8.

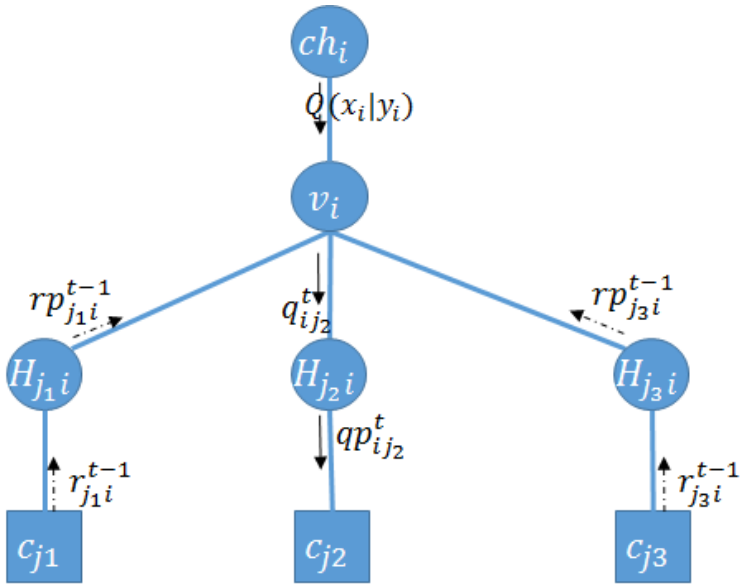


Figure 2-8: Variable node processing. The message $q_{ij_2}^t$ is the variable node v_i message to the permutation node $H_{j_2 i}$ at iteration t. The permutation node permutes the incoming message from the variable node and sends the resulting message $qp_{ij_2}^t$ to check node c_{j_2}. It depends on the messages from the channel and from the neighboring check nodes to v_i except the check node transmitted to c_{j_2} at the previous iteration $t-1$. Note $V_i = \{j_1, j_2, j_3\}$.

3- **H-matrix Multiplication (Permutation):** The multiplication over a finite field results in a permuted vector of the original one.

$$qp_{ij}^{l,(a)} = H_{ji} q_{ij}^{l,(a)}, \qquad \forall i \in \{1,...,n\}, j \in V_i \tag{2.13}$$

4- **Check Node Message Computation:** Each check node c_i computes its outgoing messages r_{ij} to its neighboring variable nodes v_j's (figure 2-9) as:

$$r_{ij}^{l,(a)} = \sum_{w \in cnf(a,s_i)} \prod_{\substack{j' \in C_i / \{j\} \\ a' \in w}} qp_{j'i}^{l-1,(a')}, \qquad \forall i \in \{1,...,m\}, j \in C_i \tag{2.14}$$

Where $cnf(a, s_i)$ is the set of all vectors w of size $d_{c_i} - 1$ such that $\sum_i w_i + a = s_i$. In other words, $cnf(a, s_i)$ is the configuration set of all the possible weighted symbol values of the neighboring nodes to c_i can take such that the parity check is satisfied with syndrome s_i and one of the neighboring nodes is fixed at symbol a.

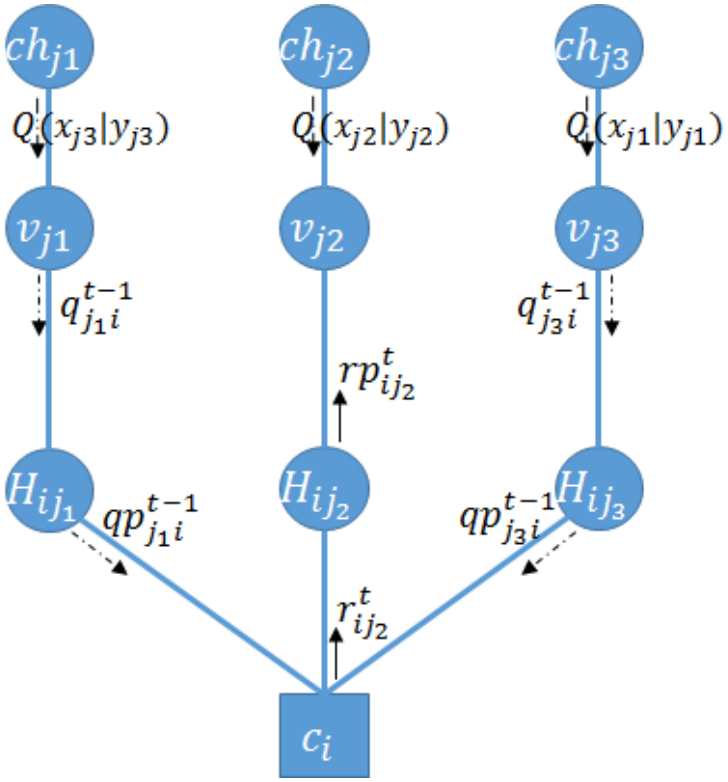


Figure 2-9: Check node processing. The message $r_{ij_2}^t$ is the check node c_i message to the permutation node H_{ij_2} at iteration t. The permutation node permutes the incoming message from the check node and sends the resulting message $rp_{ij_2}^t$ to variable node v_{j_2}. It depends on the messages from the neighboring variable nodes to c_i except the variable node transmitted to v_{j_2} at the previous iteration $t-1$. Note $C_i = \{j_1, j_2, j_3\}$.

Equation 2.14 is basically a $d_{c_i} - 1$-fold convolution of message vectors $q_{j'i}$, since it is a summation of products of the vectors components such that these components sum up to $z - a$ in each product term. The straightforward computation of $d_{c_i} - 1$-fold convolution is $O(q^{d_c - 1})$ and we need to compute d_c messages per

check node, therefore, the total complexity per check node is $O(q^{d_c})$. This computation repeatedly computes the same smaller sub-problems. Davey and MacKay [24] proposed a method of computing solutions for these repeated problems once and re-using that in the repeated instances of these problems. This reduces the complexity to $O(q^2)$. We use this method in our software decoder. We describe this method as:

Define $\sigma_{ik} = \sum_{j \leq k} q p_{ij}$, and $\rho_{ik} = \sum_{j \geq k} q p_{ij}$. Choose $k > j$ as two successive indices in C_i, then:

$$P(\sigma_{ik} = a) = \sum_{z,t:t+z=a} P(\sigma_{ij} = z) q p_{ik}^{t}$$

Similarly choosing $k < j$, we have:

$$P(\rho_{ik} = a) = \sum_{z,t:t+z=a} P(\rho_{ij} = z) q p_{ik}^{t}$$

Equation 2.14 becomes:

$$r_{ij}^{a} = P(\sigma_{i(j-1)} + \rho_{i(j+1)} = s_i - a) = \sum_{z+t=s_i-a} P(\sigma_{i(j-1)} = z) P(\rho_{i(j+1)} = t), \quad \forall i \in \{1, ..., m\}, j \in C_i \tag{2.15}$$

The complexity of computing $P(\sigma_{i(j-1)}$ is $O((j-1)q)$ and similarly for $\rho_{i(j+1)}$ assuming 1-indexing. Therefore, the complexity of computing r_{ij} is $O(nq^2)$.

5- **H-matrix inverse Multiplication (Permutation):**

$$rp_{ij}^{l,(a)} = H_{ji}^{-1} r_{ij}^{l,(a)}, \qquad \forall i \in \{1, ..., m\}, j \in C_i \tag{2.16}$$

6- **Variable Node Belief Computation and Bit Decision:** Each variable node v_i computes its belief q_i as:

$$q_i^{a} = P(x_i = a | y_i) \prod_{j' \in V_i} rp_{j'i}^{a}, \qquad \forall i \in \{1, 2, ..., n\}, j \in V_i \tag{2.17}$$

This belief is normalized to:

$$qn_i^a = \frac{q_i^a}{\sum_{a'=0}^{q-1} q_i^{a'}} \quad \forall i \in \{1, 2, ..., n\}, j \in V_i$$

Then the decoding decision is:

$$x_i^{'} = argmax_a(qn_i^a), \qquad \forall i \in \{1, 2, ..., n\}$$

Where $x_i^{'}$ is the i-th decoded bit, and $P(x_i = a|y_i)$ from the channel message.

Note that if the code is defined over $GF(q)$ such that $q = 2^p$, where p is prime, then the convolution could be replaced by a product by transforming the problem into the Fourier Transform domain over a finite field, also called *Number-theoretic Transform (NTT)*. This reduces the complexity to $O(q \log_2(q))$, and the domain conversion is done with Cooley-Tukey algorithm [25]. In the case of $GF(p)$, there are algorithms for NTT, such as Blueshtein's [26] and Rader's [27] algorithms. However, these are more complex to implement and could be more costly than the partial sums implementation for small field sizes, such as $GF(13)$, which we use in this project.

In this chapter, we covered the LDPC concepts necessary for our experiment in Section 3.4. We started with binary LDPC codes and showed their graph representation in Section 2.1. We described the binary LDPC decoding using a belief propagation concept and the sum-product algorithm in Section 2.2. We explained that LDPC codes with larger girth perform better under belief propagation decoding. For this reason, we introduced the CPEG algorithm for designing quasi-cyclic codes with maximized girth in Section 2.3.2. Finally, we generalize LDPC codes to the non-binary field order, and provide an efficient implementation of the sum-product algorithm algorithm for non-binary decoding in Section 2.4.

In the next chapter, we describe our experiment on a model of a flash memory channel, in which we compare two flash storage schemes: one with binary LDPC and 16 levels of charge per cell (QLC); and the other with non-binary LDPC over $GF(13)$ and 13 levels of charge per cell. We use the flash concepts explained in

Chapter 1 to develop the channel model of the experiment in Chapter 3. We design binary and non-binary LDPC codes with CPEG and we implement an binary and non-binary decoders in C to simulate the two schemes.

Chapter 3

Experiment and Evaluation

3.1 Non-Binary Scheme

The goal of this work is to compare the storage of non-binary number of states in the flash cell against the traditional binary number. The comparison evaluates the two schemes in terms of error correction performance with the LDPC module. In this project we compare a 13-ary scheme against the 16-ary (QLC) scheme. For a fixed error correction power, the non-binary scheme operates at a lower SNR than the 16-ary scheme, where we define the SNR in section 3.3.3. We explain these results in detail in section 3.5 and discuss what it means for the non-binary scheme to support lower SNR in relation to the physical properties of the flash.

In practical systems, programming non-binary states in the flash requires a special architecture to interface between the binary world of user data and the non-binary flash memory. We propose two different architectures to achieve this, and we choose one of them and explain why it is the best to choose.

The first architecture is shown in Figure 3-1, where **the writing process is as follows:**

1. Encode the user binary data with binary LDPC into binary codewords.

2. Encode the binary data into 13-ary symbols with the modulation encoder.

3. Program the 13-ary non-binary symbols into the NAND.

The reading process is the reverse of writing:

1. The 13-ary symbols are read from the NAND.

2. The 13-ary symbols are decoded into binary data with the modulation decoder.

3. The binary is decoded with the LDPC decoder into the user binary data.

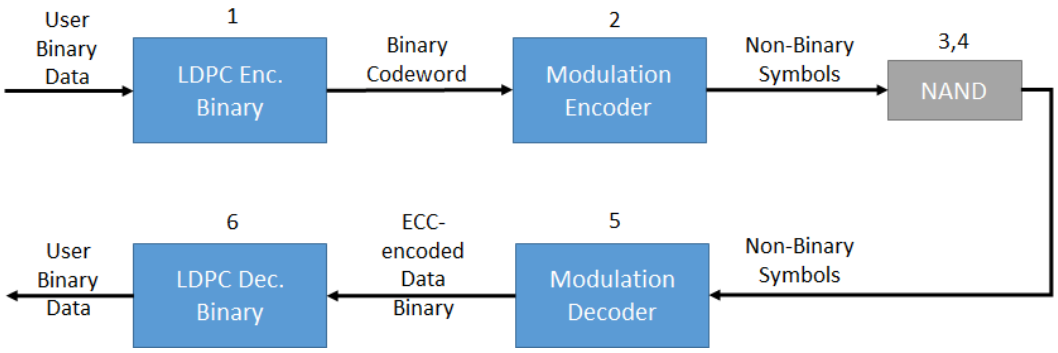


Figure 3-1: The first architecture. Non-Binary scheme with binary LDPC.

In the second architecture , shown in Figure 3-2, we flip the order of the modulation and LDPC. **The writing process becomes:**

1. Encode the user binary data into 13-ary symbols with the modulation encoder.

2. Encode the 13-ary symbols into 13-ary codewords with a non-binary LDPC.

3. Program the 13-ary non-binary symbols into the NAND.

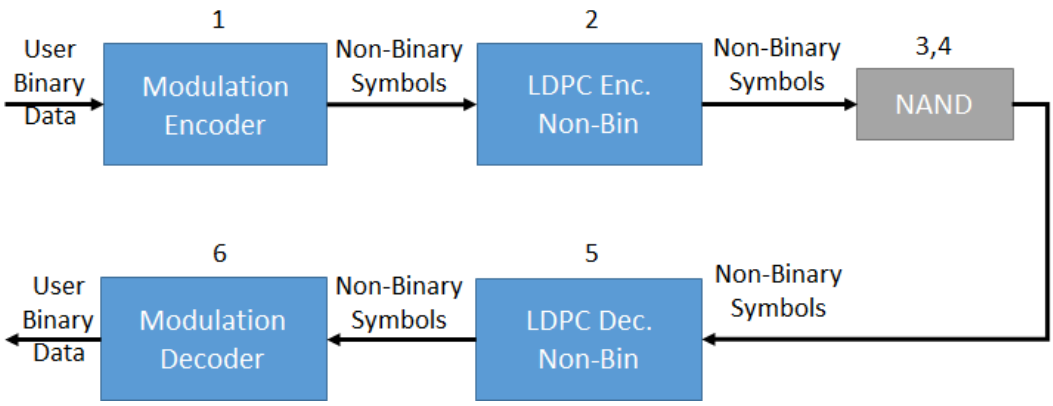


Figure 3-2: The second architecture. Non-Binary scheme with non-binary LDPC.

And the reading process is the reverse of writing.

The first architecture has the disadvantage of error propagation. A group of bits are modulated together into a smaller sized group of 13-ary symbols. This makes any symbol flip in the symbols read (step 4) propagate to a possibly longer string of binary bit flips, which could be the entire group of bits modulated together, after the symbols are decoded by the modulation decoder (step 5). Note the initial symbol flip is due to noise introduced by the NAND. The second architecture does not result in error propagation, since the modified symbols are corrected by the LDPC decoder before the modulation decoding. However, this architecture requires the non-binary LDPC decoder that is more complex to implement.

The error propagation in the first architecture can be mitigated using smart modulation code design and other techniques, such as interleaving [28] and Gray coding. On the other hand, the second architecture is more complex, but it is more powerful in terms of error correction, since it does not get exposed to the error propagation. This architecture is chosen for its reliability, especially in a flash system with 0.9 LDPC coderate and where a decoding failure is extremely expensive. The second architecture will be compared against a basic binary scheme with binary LDPC (Figure 3-3).

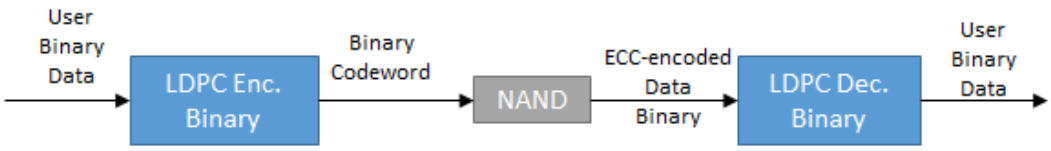


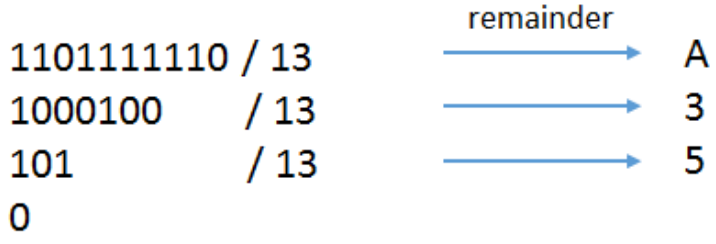
Figure 3-3: Basic binary scheme. It uses binary LDPC.

3.2 Modulation Codes and Programming 13-ary Symbols

Modulation coding is the mapping of data symbols between two domains with different constraints. In the context of this project, we map binary data symbols (bits) to 13-ary data symbols. In this case, the two constraints of the two domains are data symbols over $GF(2)$ and $GF(13)$, respectively. We call the mapping from binary to 13-ary, modulation encoding, and the opposite mapping, modulation decoding. Note that modulation does not have to be between domains of different field orders. For instance, there are modulations between two binary domains in magnetic recording, where the constraints are on the length of the strings of consecutive ones.

A basic way of mapping binary to 13-ary is to take the binary data string as one value and repeatedly divide it by the new base, that is 13, until the value becomes zero. The remainder of each division is a 13-ary symbol in the new data string with the first remainder being the least significant symbol. This method is called *Base Conversion*, and an example is shown here:

$$1101111110_2 \rightarrow 53A_{13}$$



When we map n bits to m 13-ary symbols, the coderate is $\frac{2^n}{13^m}$. The coderate ≤ 1 because the mapping is injective. In other words, the number of elements in the set of n-bit strings must not exceed the number of elements in the set of m-symbol strings, so there is no information loss in the modulation process. Note coderate $\neq 1$ if m and n are integers. Therefore, there are some m-symbol data strings that are not used in any modulation code. In fact, the modulation code could be optimized to use the noisier symbols less often, if some symbols are noisier than others. Indeed, this is the case in the flash memory channel (section 1.2) where different symbols have different noise variance. This optimized modulation code is implemented with a look up table in digital hardware, which maps n-bit data strings into m-symbol strings. In this work, we do not focus on designing and evaluating modulation codes, especially that all the 13-ary distributions have the same variance in our channel model.

Figure 3-4 shows the coderate of modulation at different values of m, the corresponding value of n in each case is maximized such that coderate ≤ 1. Note as m increases, the modulation encoder and decoder need more hardware resources to implement. On the other hand, low-valued $(m = 1, m = 2)$ have low coderate, therefore, low channel capacity for the system. In this work, we evaluate the second architecture with non-binary LDPC code (steps (2-5) in Figure 3-2). We suggest $m = 10, n = 37$ modulation code for its high code rate (0.997). $m = 3, n = 11$ is good for lower complexity and high coderate.

3.3 Channel Model

We model the flash memory channel as an AWGN channel, and we use PAM signal constellation as described in section 1.2.1. In this model, the total voltage space of the flash cell is normalized to 1, and all the symbols have the same variance.

We compare two signal constellations: One with 16 symbols; and the other with 13 symbols. The lowest mean (S_0 mean) in the 16-ary constellation is fixed at 0 and the highest (S_{15} mean) is fixed at 1, and the rest of the symbol means are distributed

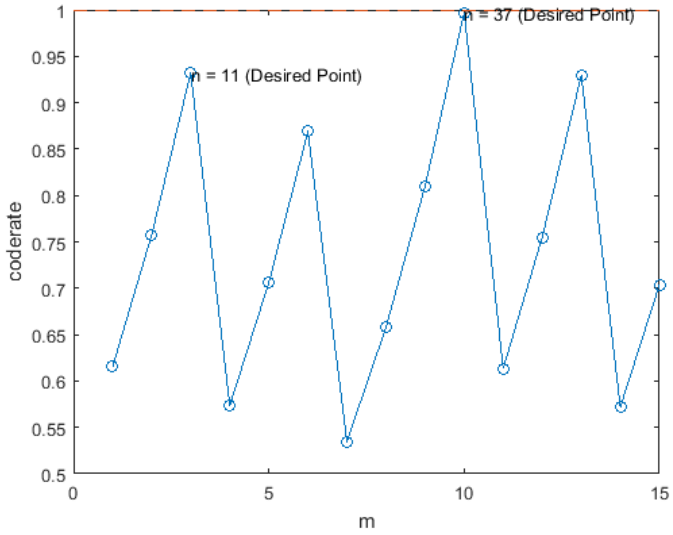


Figure 3-4: The coderate of modulation at different values of m, the corresponding value of n in each case is maximized such that coderate ≤ 1.

in between, as shown in Figure 3-5. The means of the $S_1, ..., S_{14}$ distributions are optimized to balance out the error of misreading a symbol as another symbol for all the 16 symbols. Note $S_1, ..., S_{14}$ will not simply be equally separated, since S_0 and S_{15} only overlap with one neighboring symbol instead of two in the case of $S_1, ..., S_{14}$.

The 13-ary constellation is similar to the 16-ary one, but with the first two symbols and the last one removed and the total voltage space normalized to $\frac{14-2}{15} = \frac{12}{15}$, as shown in Figure 3-6. The means of the symbols $S_3, ..., S_{13}$ are optimized to balance out the error as in the 16-ary case. This choice of the removed symbols is only significant in the more sophisticated model (section 3.3.4), where the symbol distributions have different variance. We follow the same choice here for consistency. Note we still refer to the symbols by the same label in both cases.

3.3.1 State Transition Matrix (STM)

Let X be the symbol transmitted through the flash channel. X is a discrete random variable $\in 0, 1, ..., q-1$, where q is the symbol alphabet size. U is a continuous random variable over the flash cell voltage space. This is the variable observed

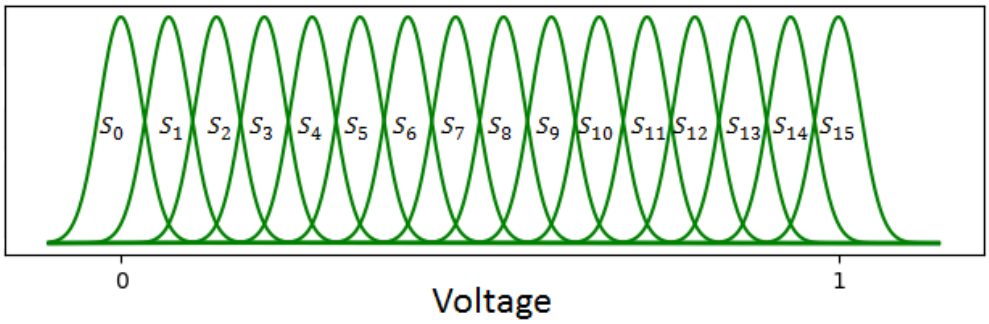


Figure 3-5: Flash channel model with 16-ary signal constellation. Note S_0 mean is fixed at 0 and S_{15} mean is fixed at 1. Note the symbol distributions are not equally separated, but the difference in separation is very small that it is hard to see on the figure.

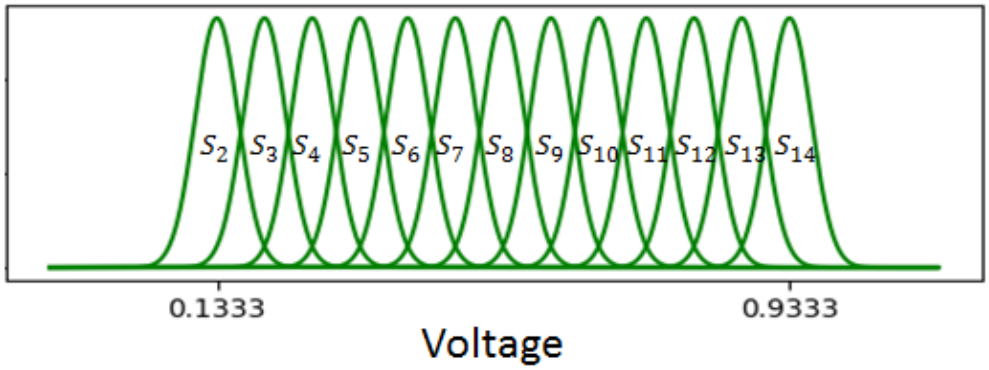


Figure 3-6: Flash channel model with 13-ary signal constellation. Note S_2 mean is fixed at $\frac{2}{15} = 0.1333$ and S_{14} mean is fixed at $\frac{14}{15} = 0.9333$. Note the symbol distributions are not equally separated, but the difference in separation is very small that it is hard to see on the figure.

before the channel detector. The channel detector takes U as input and computes Y based on the read thresholds $V_{t_1}, ..., V_{t_{q-1}}$ as an output (section 1.2.1). Y is the received symbol, which is a discrete random variable $\in 0, 1, ..., q-1$.

$$P(U = u|X = i) = \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{(u-\mu_i)^2}{2\sigma_i^2}} \tag{3.1}$$

$$Y = \begin{cases} 0 & \text{if } u < V_{t_1} \\ i & \text{if } Vt_i < u < V_{t_{i+1}} \\ q-1 & \text{if } u > V_{t_{q-1}} \end{cases} \tag{3.2}$$

$$P(Y=j|X=i) = \begin{cases} \Phi_{U|X=i}(V_{t_1}) & \text{if } j=0 \\ \Phi_{U|X=i}(V_{t_{j+1}}) - \Phi_{U|X=i}(V_{t_j}) & \text{if } 0<j<q-1 \\ 1-\Phi_{U|X=i}(V_{t_{q-1}}) & \text{if } j=q-1 \end{cases} \tag{3.3}$$

Where $\Phi_X(x)$ is the CDF of the probability distribution of X at x. The *State Transition Matrix (STM)* is a $t \times r$ matrix, where r is the size of received symbols alphabet and t is the size of transmitted symbols alphabet. Note the two alphabets could be different and have different sizes, as in section 3.3.5. Each element a_{ij} in this matrix is $a_{ij} = P(Y=j|X=i)$. In our experiment, this matrix simulates the transition from the transmitted codewored to the received data. It is also used to assign $LLR(X|Y) = \ln \frac{P(X=0|Y)}{P(X=1|Y)}$ values, after computing P(X = i | Y = j) from the STM using Bayes' rule:

$$P(X=i|Y=j) = \frac{P(Y=j|X=i)P(X=i)}{P(Y=j)} \tag{3.4}$$

Where $P(Y=j) = \sum_{i=0}^{q-1} P(Y=j|X=i)P(X=i)$

3.3.2 Channel Capacity

Let X and Y be two discrete random variables representing the transmitted and received symbols over a channel, respectively. The channel capacity C is then defined as:

$$C = \sup_{P(x)} I(X;Y) = \sup_{P(x)} \sum_{y\in Y} \sum_{x\in X} P(x,y) log(\frac{P(x,y)}{P(x)P(y)}) \tag{3.5}$$

Where $I(X;Y)$ is the mutual information of two random variables X and Y. The definition states that the channel capacity is the mutual information between the transmitted and received data maximized over the distribution of the transmitted data. Note that C depends on the read thresholds through its dependence

on $P(x,y)$ and $P(y)$. The read thresholds are optimized to maximize the channel capacity.

We also define the *Effective Channel Capacity* C_{eff}, as the mutual information of X and Y for a certain distribution $P(X)$:

$$C_{eff} = I_{P(X)}(X;Y) \tag{3.6}$$

In our experiment, we assume a uniform distribution $P(X = i) = \frac{1}{q}$. Note that when a symbol i has a relatively high transition probability to a symbol $j, j \neq i$, we can transmit it less often, i.e. have a non-uniform $P(x)$, through modulation encoding (section 3.2). This is beyond the scope of this thesis and we will stick with the uniform $P(x)$. Note the channel capacity computations of AWGN channels must carried out on a computer to evaluate cumulative distribution function of the Gaussian random variables.

3.3.3 Signal-to-Noise Ratio (SNR) Definition

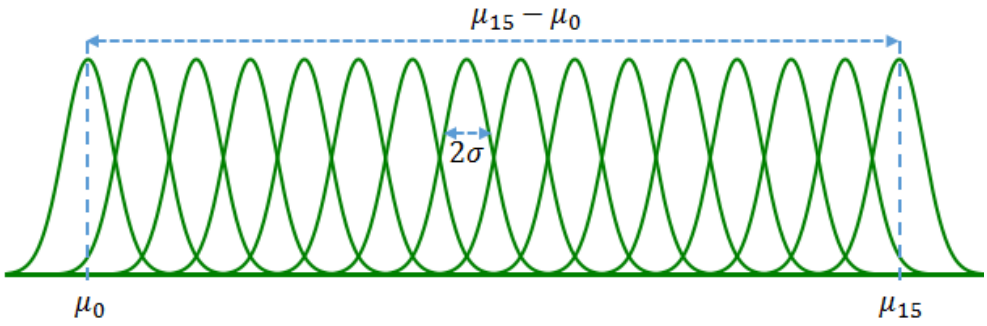
$$SNR = 20 \log_{10} \frac{(\mu_r - \mu_l)}{\sigma} \tag{3.7}$$

Where μ_l and μ_r are the lowest and highest means of the signal constellation, respectively, which you could think of as the power of the signal in this context. σ is the standard deviation of the noise in the channel. The SNR is a logarithmic with units in decibel (dB).

Note the SNR captures the characteristics of AWGN channel. Therefore, the model has the same characteristics as the physical channel although the voltage space (signal power) and noise variance are different. In fact, this is what we have in our model, as we normalized the voltage space to 1.

3.3.4 A More Sophisticated Channel Model

We present a closer model to the behavior of the flash channel, which is similar to the basic model, but with different variance for the symbols. S_0 having variance



$$SNR = 20\ log_{10}(\frac{\mu_{15} - \mu_0}{\sigma})$$

Figure 3-7: Signal-to-Noise Ration (SNR) calculation.

$\sigma^2_{S_0}$, S_{15} having $\sigma^2_{S_{15}}$, and all the other symbols having σ^2, such that $\sigma^2_{S_0} > \sigma^2_{S_{15}} > \sigma^2$, as shown in Figure 3-8. Note the distributions means are optimized to maximize the effective channel capacity. This results in relatively higher separation between the high variance distributions and the rest of the distributions. We evaluate both models in section 3.5.

Figure 3-8: Flash channel model with 16-ary signal constellation. Note S_0 and S_{15} have higher variance. The separation between these two symbols and the other symbols is relatively high to balance out the raw error rate and maximize the channel capacity. The dots represent the rest of the 16 symbols with variance σ. Note the labels on the figure are twice the variance.

3.3.5 Soft Information

We read with $q-1$ read thresholds when we have a symbol alphabet of size q. The information we get with these thresholds are called *Hard Information*. When

we read again with a different set of $q-1$ thresholds, each received symbol region will be divided into regions giving different beliefs of the symbol transmitted as shown in Figure 3-10 and Figure 3-11. We call this belief information obtained with multiple reads, *Soft Information*.

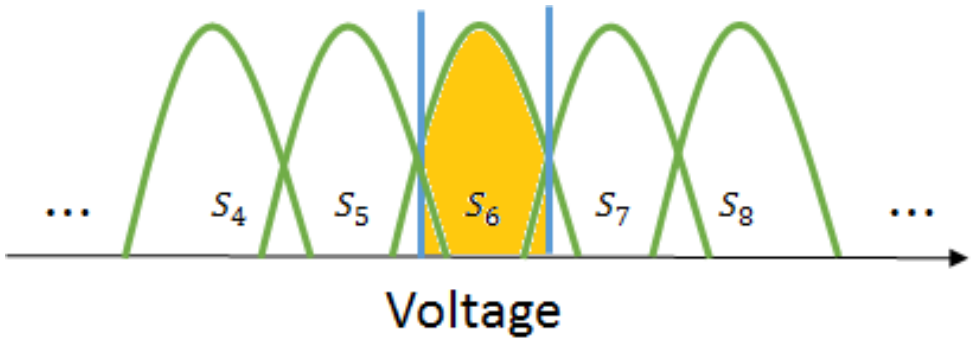


Figure 3-9: Single read. The cell is detected in S_6 region. The channel detector gives high belief the symbol transmitted is S_6. Note the belief is non-zero in the other symbols as their distributions overlap in the detection area. The detector gives low beliefs in S_5 and S_7, and much lower beliefs in the rest.

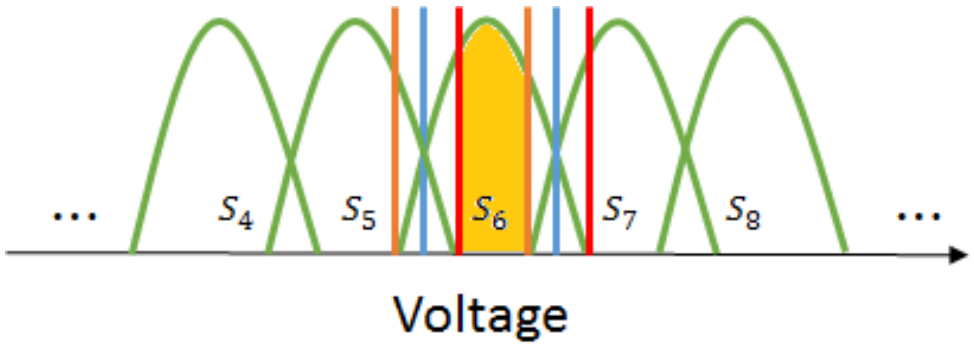


Figure 3-10: Three reads. The cell is detected in the wide region of S_6. The channel detector gives high belief the symbol transmitted is S_6, and lower beliefs in the rest.

As with the original set of read thresholds, the goal of choosing the other sets associated with the multiple reads is maximizing the effective channel capacity. The support set of the received symbol random variable Y increases with the number of reads n_{reads}. It becomes $n_{reads}q-2$ where each received symbol corresponds to a region between two adjacent read thresholds. In reference to section 3.3.1, the STM that captures the flash channel with multiple reads has higher column

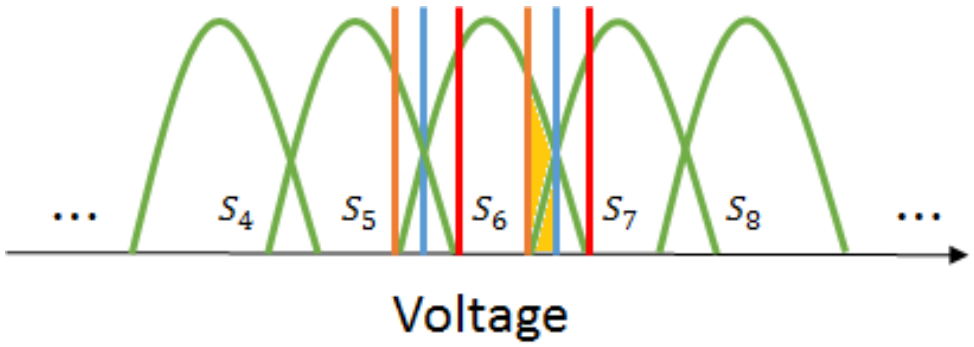


Figure 3-11: Three reads. The cell is detected in a narrow S_6 region close to S_7 region. The channel detector gives comparable beliefs in the symbol transmitted being S_6 and S_7, and lower beliefs in the rest.

dimension than row dimension. Therefore, we get higher resolution beliefs of the symbols transmitted through the channel, which makes the LDPC decoder decode more codewords and converge faster, i.e. with fewer iterations.

The transmitted symbols are associated with the same type of distribution (Gaussian), and the distributions have approximately the same variance. Therefore, the read thresholds associated with multiple reads are only related by a shift factor δ. In other words:

$$V_{t_i}^{j} = V_{t_i}^{k} + \delta_{j,k}$$

Where Vt_i^j denotes the Vt_i of the j-th read. This makes the numerical optimization to maximize the effective channel capacity easier than the case with different offset for every single read threshold.

3.4 Experiment

We compare two schemes: a 13-ary flash system with non-binary LDPC against a 16-ary system with binary LDPC. We use the channel distributions $(S_2 - S_{14})$ in the 13-ary case. We also compare both channel models, simplified and sophisticated, in the 16-ary case. In the sophisticated channel, we choose $\sigma_{S_0} = 1.5\sigma$ and $\sigma_{S_{15}} = 1.2\sigma$, where σ is the standard deviation of the rest of the symbols. Note there is

only one model in the 13-ary case as we do not use S_0 and S_{15}.

We do not focus on the modulation code design in this work, but we suggest using a code with $n = 37$ bits mapped to $m = 10$ 13-ary symbols, for its very high coderate (0.997) with 3.7 bits/symbol. In this case, the non-binary LDPC codeword size has to be a multiple of 10. The binary LDPC codeword size $n = 16000$, that is 2K bytes, a common codeword size in the flash storage systems. The non-binary codeword size is $n = 4320$ symbols. This is the size of the binary codeword divided by the 3.7 bits per symbol use and rounded to the nearest multiple of 10. This is to guarantee a fair comparison between the two schemes by having roughly the same size codeword from the user binary world perspective, i.e. number of bits/codeword. Note that the codeword size is an important characteristic considered with the LDPC code performance, since the longer the codeword, the better the code performs [12].

Both the binary and non-binary LDPC codes are quasi-cyclic codes designed with the CPEG algorithm described in section 2.3.2, starting from a 4×40 protograph with coderate $r = 0.9$ and column weight = 3. This protograph is lifted with CPEG using lift factor $Z = 400$ to make the binary code, and lift factor $Z = 108$ to make the non-binary one. Note we do not specify the edge connections of the protograph given as an input to CPEG (Section 2.3.2). The coefficients in the NB-LDPC H-matrix are assigned randomly over $GF(13)$.

Another fairness measure taken in comparing the two schemes is to assign the received bits LLRs based on the 16-ary symbol channel model in the binary scheme. This means not all the received 1's will be given the same LLR, and the same is true for 0's, as illustrated in Figure 3-12. This gives the binary decoder more accurate channel beliefs of the bits similar to the channel beliefs supplied to the non-binary decoder, which are naturally based on the 13-ary channel model.

The LDPC decoder maximum number of iterations is set to 20, and 10^5 decoding trials are simulated per SNR point. In each trial, data is generated randomly, syndrome is computed for the data, and noise is generated according to the channel model and added to the data. The decoder and data generation are

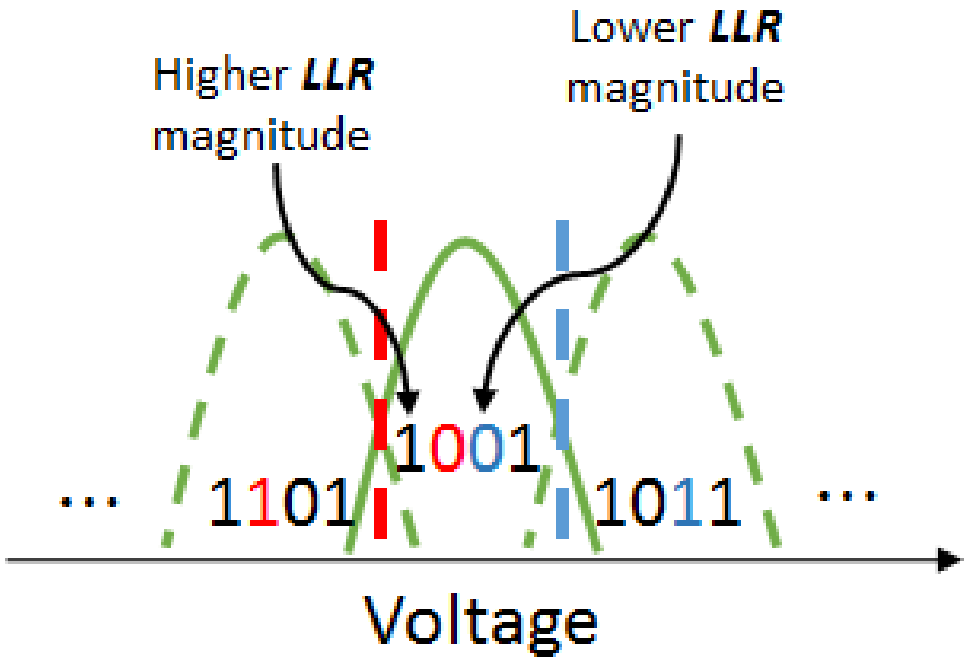


Figure 3-12: Symbol-based bit LLR assignment. Bits that change according to the Gray code if the adjacent symbol is transmitted are given lower confidence than the other bits

implemented in C. The simulation was run on a computation farm of multiple computers of multi-core processors, with MATLAB Distributed Computing Server tools used for task scheduling.

3.5 Results

First, we compare the 13-ary scheme with NB-LDPC against the 16-ary scheme with Binary LDPC and the simple channel model (Section 3.3). The bit LLR assignment in the 16-ary scheme is done symbol-based as explained in section 3.4. The results are shown in Figure 3-13 below. Here we introduce the definition of Decoding Failure Rate, which is the number of times the decoder fails to decode a codeword per total number of decoding trials. The *Decoding Failure Rate* only is only meaningful when it is specified for a certain SNR point in the AWGN channel case, or some channel characteristic measure in general.

There is approximately 2.3 dB gain in hard information decoding of the 13-

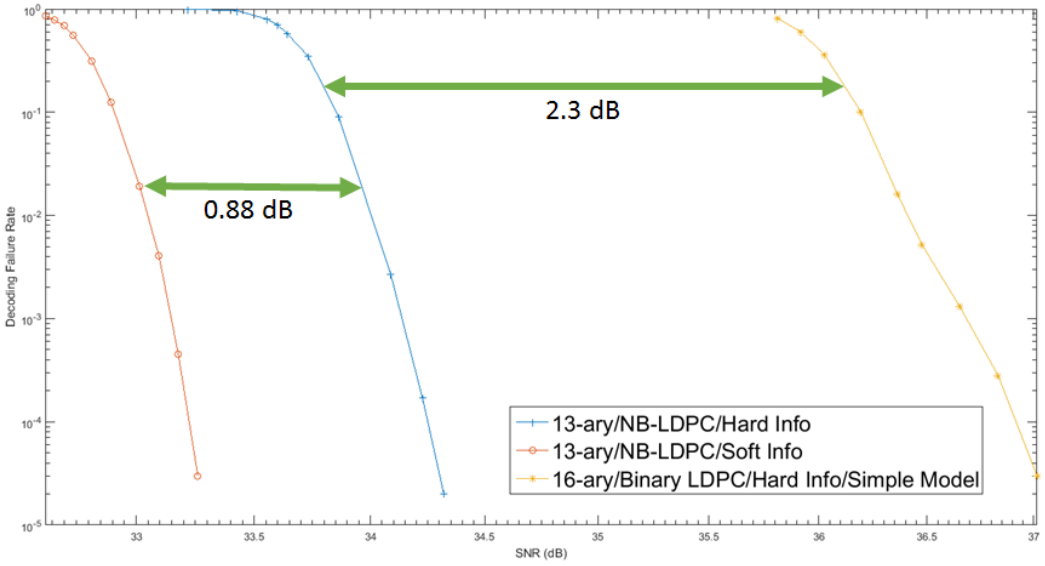


Figure 3-13: Decoding Failure Rate results of the 13-ary and 16-ary schemes. Note the soft information decoding in the 13-ary scheme is done with three reads.

ary scheme over the 16-ary one. Note there is a 1.94 dB decrement in the SNR of the 13-ary scheme, since it has lower signal amplitude 0.8 in the cell normalized voltage space. This decrement is computed as $20(\log_{10}\frac{0.8}{\sigma} - \log_{10}\frac{1}{\sigma}) = -1.94dB$. The extra 0.36 dB is the gain due to the higher error correction power of the NB-LDPC over the binary LDPC. To better understand this, if the 13-ary distributions are expanded in the voltage space so the signal has the same amplitude of 1 as in the 16-ary case, the SNR of the 13-ary signal will increase by 1.94 dB. However, since the NB-LDPC is better by 2.3 dB, it is providing an extra .36 dB benefit. There is approximately 0.88 dB gain in soft information decoding with three reads over the hard information decoding in the 13-ary scheme. Note all the read thresholds are optimized to maximize the effective channel capacity.

The decoding at a lower SNR in the 13-ary scheme with the same decoding failure rate means the system tolerates higher noise than in the 16-ary case. For a fixed SNR, if the 13-ary scheme distributions are expanded over the same amplitude as in the 16-ary scheme, the variance of the distribution needs to increase to keep the SNR constant. This is interpreted as the 13-ary scheme operating at higher P/E cycles or longer data retention causing increased noise variance (section 1.2).

The results clearly demonstrates the benefit of NB-LDPC. Practically, this comes at the cost of implementing a NB-LDPC decoder, which takes more silicon area for the same throughput in terms of decoding iterations per clock cycle. Refer to Section 2.4.1 for see the complexity of decoding NB-LDPC. We suggest that the choice of using NB-LDPC in a flash product is driven by the cost and benefit. Although the benefit is always decoding at some lower SNR, this means different things depending on the flash technology and the flash product specification. For instance, higher endurance is needed in enterprise flash products than in consumer products. The flash technology also matters such that the flash channel noise can already be so low that the benefit from NB-LDPC is not so high, and vice versa.

Other use cases of the 13-ary scheme includes increasing the yield of the fabrication process. Some QLC flash wafer may not qualify due to low reliability, i.e. high BER, but this reliability can sufficiently increase to pass the qualification if the wafer is used with the 13-ary scheme. In addition, a mix of 13-ary and 16-ary flash could be used in a single SSD, or 16-ary flash can be turned into 13-ary after being exposed to a certain number of P/E cycles to restore reliability. The complexity of implementing these different use cases is outside the scope of this work.

Note The binary and non-binary codes are constructed the same way using the same CPEG algorithm. This is a measure of fairness in the experiment to compare the benefit of decoding over higher orders specifically, and not other factors in the LDPC code structures.

Now we add another result of hard information decoding of the 16-ary scheme with the more sophisticated model introduced in section 3.3.4. The result is shown in Figure 3-14 below.

The noise standard deviation of the sophisticated model σ is calculated as:

$$\sigma = \frac{1.5\sigma_{S_0} + 14\sigma + 1.2\sigma_{S_{15}}}{16}$$

The average standard deviation of all the symbols. The two curves of the simple and sophisticated channel models in the 16-ary scheme are almost the same. We

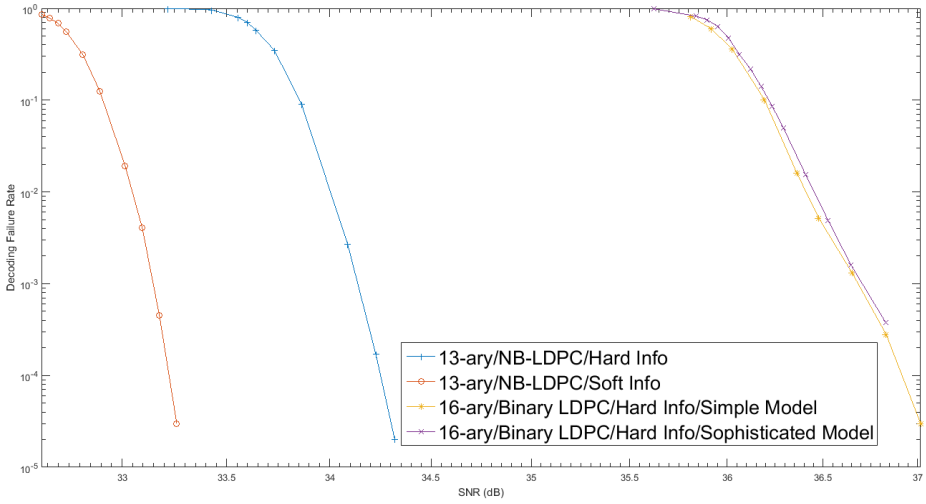


Figure 3-14: Decoding Failure Rate results of the 13-ary and 16-ary schemes. Note the results from the simple and sophisticated channel models in the 16-ary scheme are almost the same. We believe the slight discrepancy comes from defining the noise of a channel that adds Gaussian noise with different variance to different symbols.

believe the slight discrepancy is due to the definition of the noise standard deviation of the sophisticated model. The equivalent noise of a multi-variance channel like this could be different from the average above although the average gives a very close result.

3.6 Conclusion

We compared two schemes of storing information in QLC flash memory: One stores 4 bits/cell occupying the full QLC capacity with 16 levels of charge; and the other stores around 3.7004 bits/cell using 13 charge levels per cell. The non-binary scheme also has a modulation overhead, for which we presented a modulation code that makes this overhead very small, storing 3.7 bit/cell with a 0.0004 bits/cell loss only.

The coderate of the binary LDPC in the 16-ary scheme and that of the NB-LDPC in the 13-ary scheme were equalized to compare the error correction power of the

two LDPC codes. In addition, the modulation coderate of the 13-ary scheme is designed to be very close to 1 (0.997), so that the entire 13-ary scheme has a very close coderate to the 16-ary scheme, and the two schemes can be compared using the same codeword size.

Although the 13-ary scheme results in lower cell capacity, it increases the cell endurance and reliability. In this work, the 13-ary scheme was designed by removing the first two S_0 and S_1 and the last S_{15} symbols from the 16-ary scheme 3.3. We assume we can move the means of the 13-ary distributions freely in the voltage space. Therefore, we can expand the 13-ary distributions over the entire space to boost the SNR of the flash cell. Different other choices of symbols removal can be made, and these choices can be better than ours, especially if the system hardware restricts the movement of the distributions or changes the noise characteristics of the distributions depending on their means.

Bibliography

[1] Anton Shilov. Sandisk, toshiba begin to purchase equipment to make bics 3d nand, 2015.

[2] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Threshold voltage distribution in mlc nand flash memory: Characterization, analysis, and modeling. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 1285–1290. IEEE, 2013.

[3] Simon Aughton. Dell gets flash with ssd option for laptops, 2007.

[4] Sun Microsystems. Solaris zfs enables hybrid storage pools-shatters economic and performance barriers, 2008.

[5] Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, 2003.

[6] Zsolt Kerekes. Are mlc ssds ever safe in enterprise apps?, 2008.

[7] H Tanaka, M Kido, K Yahashi, M Oomura, R Katsumata, M Kito, Y Fukuzumi, M Sato, Y Nagata, Y Matsuoka, et al. Bit cost scalable technology with punch and plug process for ultra high density flash memory. In *VLSI Technology, 2007 IEEE Symposium on*, pages 14–15. IEEE, 2007.

[8] Ya-Chin King, Tsu-Jae King, and Chenming Hu. Charge-trap memory device fabricated by oxidation of si/sub 1-x/ge/sub x. *IEEE Transactions on Electron Devices*, 48(4):696–700, 2001.

[9] Hao Yu. A glivenko-cantelli lemma and weak convergence for empirical processes of associated sequences. *Probability theory and related fields*, 95(3):357–370, 1993.

[10] Shu Lin and Daniel J Costello. *Error control coding*. Pearson Education India, 2004.

[11] Tom Richardson and Ruediger Urbanke. *Modern coding theory*. Cambridge university press, 2008.

[12] Robert Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

[13] Achilleas Anastasopoulos. A comparison between the sum-product and the min-sum iterative detection algorithms based on density evolution. In *Global Telecommunications Conference, 2001. GLOBECOM'01. IEEE*, volume 2, pages 1021–1025. IEEE, 2001.

[14] Joachim Hagenauer, Elke Offer, and Lutz Papke. Iterative decoding of binary block and convolutional codes. *IEEE Transactions on information theory*, 42(2):429–445, 1996.

[15] Henk Wymeersch, Heidi Steendam, and Marc Moeneclaey. Log-domain decoding of ldpc codes over gf (q). In *Communications, 2004 IEEE International Conference on*, volume 2, pages 772–776. IEEE, 2004.

[16] Guiqiang Dong, Ningde Xie, and Tong Zhang. On the use of soft-decision error-correction codes in nand flash memory. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 58(2):429–439, 2011.

[17] Jeremy Thorpe. Low-density parity-check (ldpc) codes constructed from protographs. *IPN progress report*, 42(154):42–154, 2003.

[18] Zhongfeng Wang and Zhiqiang Cui. Low-complexity high-speed decoder design for quasi-cyclic ldpc codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(1):104–114, 2007.

[19] Madiagne Diouf, David Declercq, Marc Fossorier, Samuel Ouya, and Bane Vasić. Improved peg construction of large girth qc-ldpc codes. In *Turbo Codes and Iterative Information Processing (ISTC), 2016 9th International Symposium on*, pages 146–150. IEEE, 2016.

[20] Marc PC Fossorier. Quasicyclic low-density parity-check codes from circulant permutation matrices. *IEEE Transactions on Information Theory*, 50(8):1788–1793, 2004.

[21] Yige Wang, Jonathan S Yedidia, and Stark C Draper. Construction of high-girth qc-ldpc codes. In *Turbo Codes and Related Topics, 2008 5th International Symposium on*, pages 180–185. IEEE, 2008.

[22] Xiao-Yu Hu, Evangelos Eleftheriou, and Dieter-Michael Arnold. Regular and irregular progressive edge-growth tanner graphs. *IEEE Transactions on Information Theory*, 51(1):386–398, 2005.

[23] Zongwang Li and BVK Vijaya Kumar. A class of good quasi-cyclic low-density parity check codes based on progressive edge growth graph. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, volume 2, pages 1990–1994. IEEE, 2004.

[24] Matthew C Davey and David MacKay. Low-density parity check codes over gf (q). *IEEE Communications Letters*, 2(6):165–167, 1998.

[25] David Declercq and Marc Fossorier. Decoding algorithms for nonbinary ldpc codes over gf (q). *IEEE Transactions on Communications*, 55(4):633–643, 2007.

[26] Leo Bluestein. A linear filtering approach to the computation of discrete fourier transform. *IEEE Transactions on Audio and Electroacoustics*, 18(4):451–455, 1970.

[27] Charles M Rader. Discrete fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968.

[28] Giuseppe Caire, Giorgio Taricco, and Ezio Biglieri. Bit-interleaved coded modulation. *IEEE transactions on information theory*, 44(3):927–946, 1998.