The Report committee for Varsha Regulapati

Certifies that this is the approved version of the following report

Error Correction Codes in NAND Flash Memory

APPROVED BY SUPERVISING COMMITTEE:

Supervisor: ______________________________

Nur Touba

____

Earl E. Swartzlander, Jr.

Error Correction Codes in

NAND Flash Memory

by

Varsha Regulapati, B.E.

Report

Presented to the Faculty of the Graduate School

of the University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December, 2015

Acknowledgements

I would like to express my sincere gratitude to Prof. Nur Touba for supervising this work and for providing constant guidance and support. It has been a very pleasant and inspiring experience working with him on this report as well as attending his lectures. I would also like to thank Prof. Earl E. Swartzlander, Jr. for agreeing to review this report and providing insightful inputs and feedback.

I am also grateful to Mr. Karthikeyan Ramamurthi and Mr. Zion Kwok from Intel for graciously helping me during the course of my work on this report.

I am forever indebted to my loving family and friends for their invaluable support and understanding.

Error Correction Codes in NAND Flash Memory

by

Varsha Regulapati, MSE

The University of Texas at Austin, 2015

SUPERVISOR: Nur Touba

Error Correction Codes (ECC) are used in NAND Flash memories to detect and correct bit-errors. With shrinking technology nodes and increased memory complexity, bit error rates continue to grow. With mainstream usage of MLC/TLC devices where 2 or 3 bits of data are stored in each Floating-Gate transistor, this issue has become even more critical, and to address this, strong ECC schemes are being implemented. ECC is a good way to recover the wrong value from the remaining good bits, and robust error correction codes ensure data integrity.

This work discusses the operation of Floating-Gate transistors and NAND Flash memory. Various causes of bit-errors in these memories such as Read Disturb, repeated Program/Erase cycles and Program Disturb are presented. Analysis of various ECC schemes such as Hamming Codes, Bose, Chaudhuri, and Hocquenghem (BCH) codes and Reed-Solomon codes and their implementation in NAND Flash memory is examined. The encoding-decoding algorithms of these codes, as well as their performance and suitability for different types of Flash technology are discussed. Special emphasis is given to the discussion on Low Density Parity Codes (LDPC), which is increasingly being used as an ECC mechanism in today's NAND Flash devices.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Flash Memory: NAND and NOR Flash

NAND Flash is a non-volatile memory technology, which is primarily used for data storage purposes. It is a type of Flash memory, which is electrically re-programmable and erasable. NAND Flash devices are made up of Floating Gate transistors, which can retain data even when the power has been switched off.

Flash memory was first introduced in NOR Flash devices. NOR devices are made up of Floating Gate transistors, with the cells connected in parallel i.e. source nodes of the cells are connected together. Later, NAND Flash technology was introduced, which had the transistors connected in series, with adjacent cells sharing the source and drain regions. The main advantage of such a configuration is that, it eliminates metal contacts between the shared source and drain regions. Because of this, a NAND cell is 60% smaller than a NOR cell, leading to higher densities and lower costs for NAND technology.

1.2 Advantages of NAND Flash memory

NAND flash memories have the following advantages:

- Non-Volatile
  NAND Flash retains data even after power has been switched off.
- Significantly lower costs and Higher density
  The main advantage of NAND Flash memory is that is possesses a high density because of the small cell size. This in turn results in low costs.
- Fast Write (Program) and Erase operation
  It is faster than NOR Flash during memory write and erase. However it is slower than DRAM.

Because of the significant cost and density benefits, NAND Flash has overtaken NOR as the predominant non-volatile memory. However, NAND Flash has some distinct issues like slower performance when compared to DRAM memories, endurance and reliability problems. Various device level and circuit techniques (including ECC) are employed to rectify these issues.

Despite these issues, it is especially preferred for data storage in consumer and enterprise applications because of its higher density, lower cost-per-bit and fast programming & erasing speeds. Due to this, NAND Flash usage has been steadily increasing over the years, and it is has overtaken NOR Flash as the predominant non-volatile memory. It is also currently the second most widely used memory technology after DRAM.

1.3 Presence of NAND Flash in today's semiconductor industry

In 2014, the global semiconductor industry was reported to have sales of $335 billion [1], with Memory contributing to $79.2 billion [1]. Meanwhile, NAND Flash revenues grew from around $15 billion [2] in 2009 to more than $27 billion [2] in 2014. Thus NAND Flash memory, apart from demonstrating a huge growth, also constitutes a major chunk of the Memory market, while effectively contributing to the global semiconductor sales as well.

NAND Flash demand has also seen a huge rise because of its application in the following four market segments:

- Smartphones
- Solid-State Drives
- Tablets
- Data centers

Due to the faster write and erase speeds, NAND Flash is suitable for applications where data is continually written and erased like in smartphones, mp3 players and digital cameras. With increased smartphone and tablet usage, the demand for NAND Flash has also been steadily growing.

Additionally NAND Flash in the form of Solid State Drives (SSDs) are increasingly replacing hard disks. Even though SSDs are more expensive, they have faster speeds, smaller sizes and lower power consumption when compared to traditional hard-disks. For this reason, SSDs are being adopted for usage in both consumer and enterprise markets.

The technology industry has also seen a huge boom in big-data, cloud applications and social media. All these segments utilize datacenters, which require efficient and fast storage mechanisms. This need is being fulfilled by the usage of NAND Flash in the form of SSDs.

Currently some of the major manufacturers in the NAND Flash memory are:

- Samsung
- Toshiba
- SanDisk
- Micron
- SK Hynix
- Intel

Due to the high demand as well as the competition within major players, NAND Flash memory has witnessed aggressive scaling to achieve higher densities. Apart from aggressive scaling, most of the top manufacturers have started producing 3D-NAND devices, which provide very high densities.

1.4 Error Correction in NAND Flash memory

Since the Floating gate transistors in NAND Flash share adjacent source and drain regions, they can be packed in lesser space. However the closer proximity leads to increased cell-to-cell interference and coupling, which leads to corruption of data stored in the cells. Thus NAND Flash has an inherent problem of bit-errors, which is overcome by using Error Correction Codes (ECC). With scaling technology, the number of bit-errors becomes higher and so stronger ECC schemes are required for error correction.

Error Correction in NAND Flash is now a critical issue with the widespread usage of MLC and TLC NAND devices (where 2-bits and 3-bits are stored on every transistor). With robust ECC schemes, these devices can provide reliable data while providing increased bit density.

During initial days of NAND Flash, the error correction requirement was a single bit correction for 512B of data. Now, the error correction requirements have increased because of scaling and usage of MLC/TLC devices:

- SLC: 1 bit to 12 bits (depends on the technology node)
- MLC: 4 bits to 40 bits (depends on the technology node)
- For TLC devices, 60+ bits of correction are required.

Various ECC schemes are used for error detection and correction in NAND memories. For single bit error correction, Hamming Codes are used. For multi-bit error correction in MLC and TLC devices, Bose, Chaudhuri, and Hocquenghem (BCH) codes are used. Low Density Parity Codes (LDPC) are also being increasingly used for error correction in NAND Flash memories. A particularly specific feature of LDPC is specially suitable for NAND Flash error detection – LDPC codes can decode soft-bit data in addition to hard-bit data, and have very high performance rates when both hard-bit and soft-bit information is available. NAND chips have read features to extract soft-bit information, and thus LDPC is suitable for error correction in NAND Flash.

This work focuses on various factors, which impact bit-errors and techniques used to correct these errors in NAND memories. Chapter 2 of this work discusses the operation of NAND Flash memory, while Chapter 3 discusses various ECC schemes and their associated encoding-decoding algorithms. In Chapter 4, details about implementation of Hamming, BCH and LDPC codes in NAND Flash memories are given. Factors impacting bit-error rates and error correction in NAND memories are also given. Chapter 5 concludes this work on various Error Correction schemes implemented in NAND Flash memories.

Chapter 2: NAND Flash Memory

2.1 Introduction

NAND Flash memories are made up of Floating Gate transistors. A Floating Gate transistor is similar to a conventional MOSFET, except that it has an extra gate called the Floating gate (FG). This gate is electrically isolated and thus any charge inside the FG remains trapped. Since the FG is electrically isolated, the charge inside the floating gate can be altered by either Fowler-Nordheim tunneling [3] or hot-carrier injection mechanism.

A Floating Gate transistor ("cell") in the original state, with no electrons trapped inside the FG, can be considered to be in an "*Erased State"*. When a high positive voltage is applied to the gate, electrons from the channel tunnel through the oxide and reach the FG, where they remain trapped. This in turn leads to an increase in the amount of gate voltage required to turn on the device, leading to an increase in the Threshold voltage of the device. This new state of the device is called as the "*Programmed State*".

The Threshold voltage distribution of Floating gate transistors can be modelled as a Gaussian distribution. Thus by applying a high positive gate voltage, the distribution shifts to the right. When the device is in the "*Erased State"*, it is considered to store the binary bit "1". When the device is in the "*Programmed State*", it stores the bit "0". Thus by altering the gate voltage on a Floating gate transistor, data can be stored in the device.

Program operation [4] refers to the operation of applying high positive gate voltage, whereby causing electrons to be trapped inside the FG. Thus the device goes to the programmed state and is considered to store the bit "1". Erase operation refers to the operation of applying a high voltage to the substrate to pull back the electrons from the FG into the substrate. Thus the device comes back to the erased state and is considered to store the bit "0".

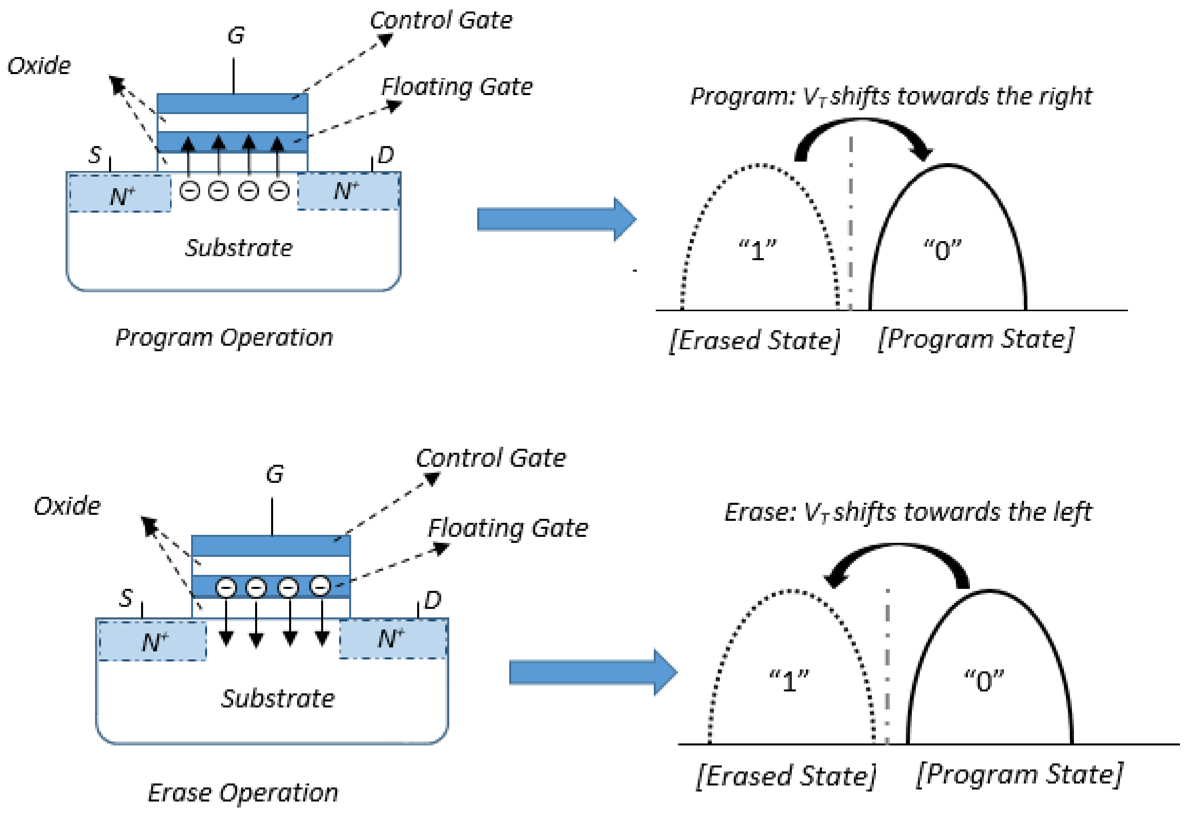


Fig 1: Programming and Erasing a Floating Gate Transistor

2.2 Single-Level-Cell and Multi-Level-Cell

A NAND Flash memory cell with only 2 states is known as a Single-Level-Cell (SLC). The granularity of the applied gate voltage can be increased in such a way that there will be more than 2 states. If the resultant threshold voltage distribution has 4 states then such a device can store 2 bits per cell [5]. This device is known as a Multi-Level-Cell (MLC). Similarly a device with 8 voltage distributions can store 3 bits per cell and is known as a Triple-Level-Cell (TLC).

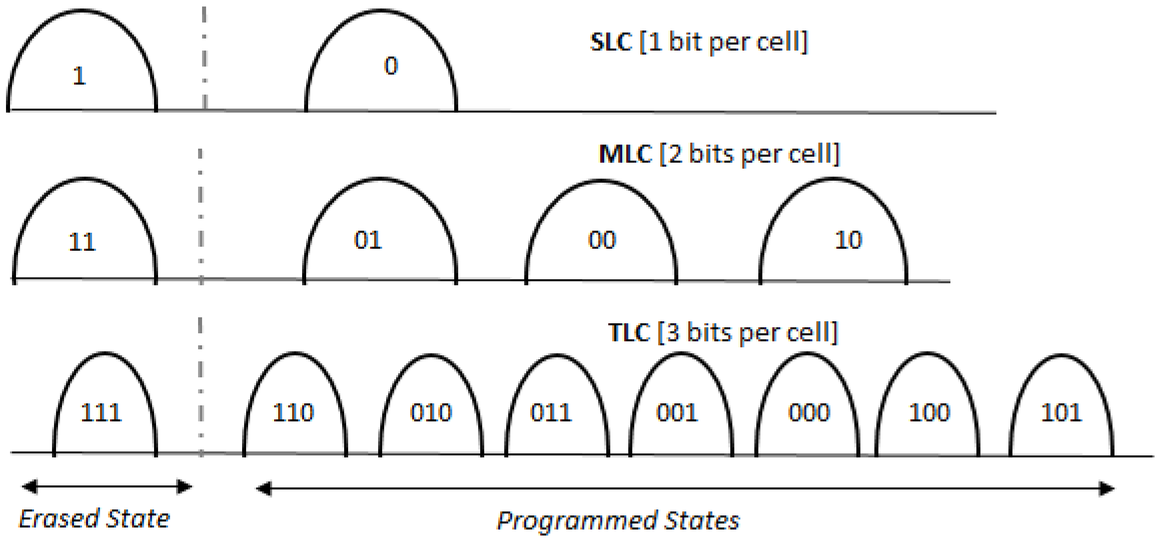


Fig 2: Threshold Voltage distribution of SLC, MLC and TLC cells

2.3 Memory Array Organization

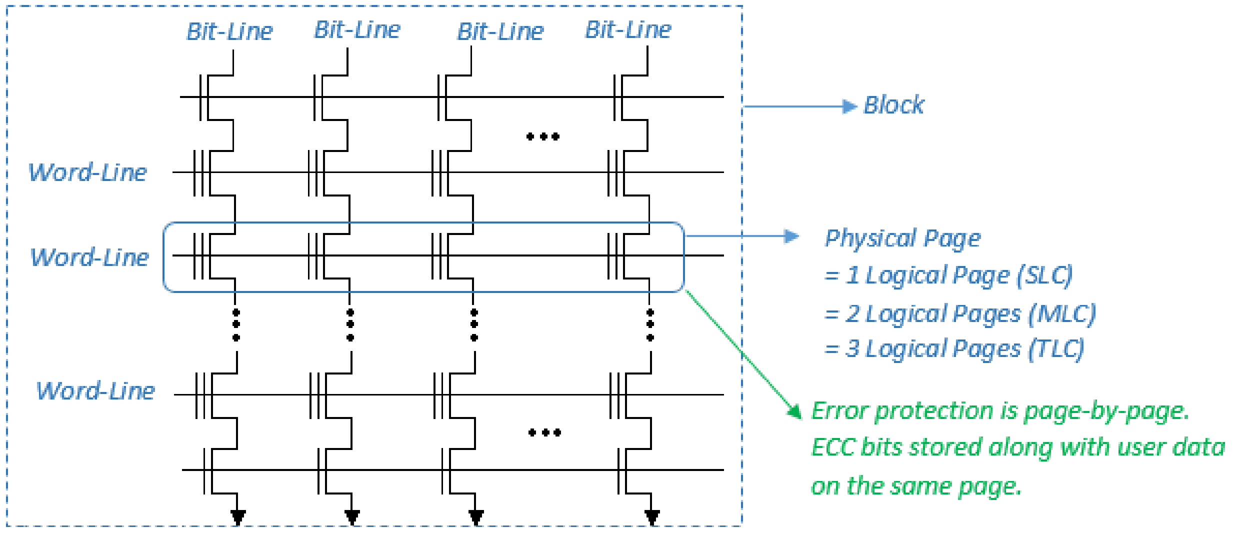


Fig 3: Organization of blocks and pages in the memory

The NAND Flash array is organized into blocks, which is the smallest unit that can be erased. Typical block sizes are 128KB, 256KB and 512KB. Each block consists of a number of pages. Read and Program operations are done one page at a time. Page sizes vary from 512B to 2KB. Error correction is given for the entire page and the ECC bits are stored along with the user data in the same page. The Bit-Lines (BL) are connected to the ends of the vertical strings of cells while the Word-Lines (WL) run in a horizontal direction and are connected to the gates of the cells. All cells connected to the same WL are considered to be in 1 physical page. In an SLC device, since one bit is stored per cell, 1 physical page is equivalent to 1 logical page. In an MLC device, 1 physical page is equivalent to 2 logical pages, and for a TLC device it is 3 logical pages.

2.4 Program Operation

Program is done on a page-by-page basis. To program one page of data, the WL of the selected page is raised to a high voltage. All the other WLs are raised to an intermediate voltage so as to make them conducting.

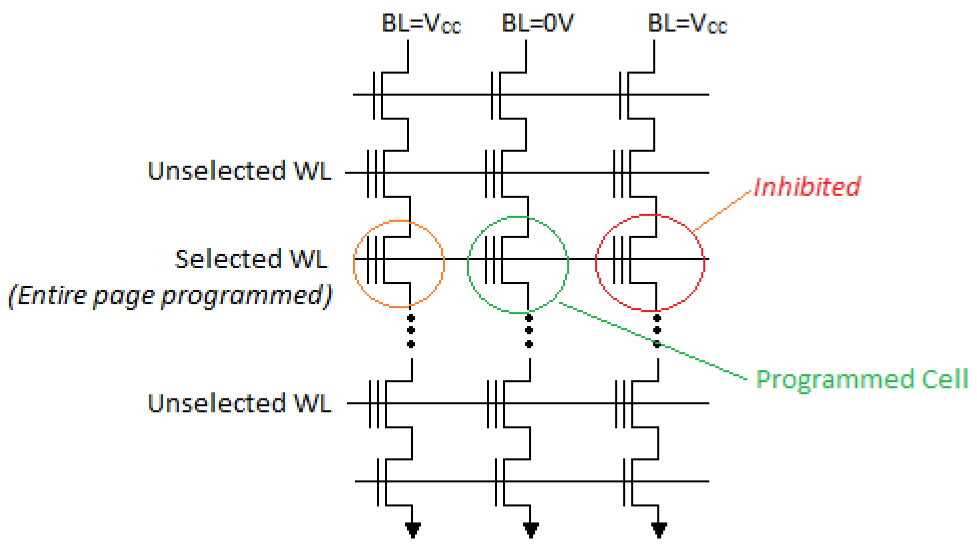


Fig 4: Program Operation

To program a cell, the BL is kept at 0V. Thus the channel is at 0V, the gate is at a high voltage, and so the cell gets programmed and the data ‘0’ is stored into it. To store a

‘1’ into a cell (which is the cell’s original state since Erased state corresponds to a logic 1), the programming needs to be inhibited [6]. This is done by raising the BL to a high voltage. Thus both the channel and gate are at a high voltage and so the cell is inhibited.

2.5 Read Operation

Read operation is done page-by-page. Apart from the cells in the selected page, all the other cells in the block are made to conduct. Then a Read voltage (V_R) is applied to the selected WL. This read voltage is exactly in between the Erased and Programmed state. If the cell is in an erased state then V_T is less than V_R and so the cell conducts. If the cell is in a programmed state, then V_T is higher than V_R and the cell doesn’t conduct. Based on the cell conduction, the BL connected to the string drops low or stays at the same voltage. This is sensed by a sense amplifier, which then outputs the read-out data.

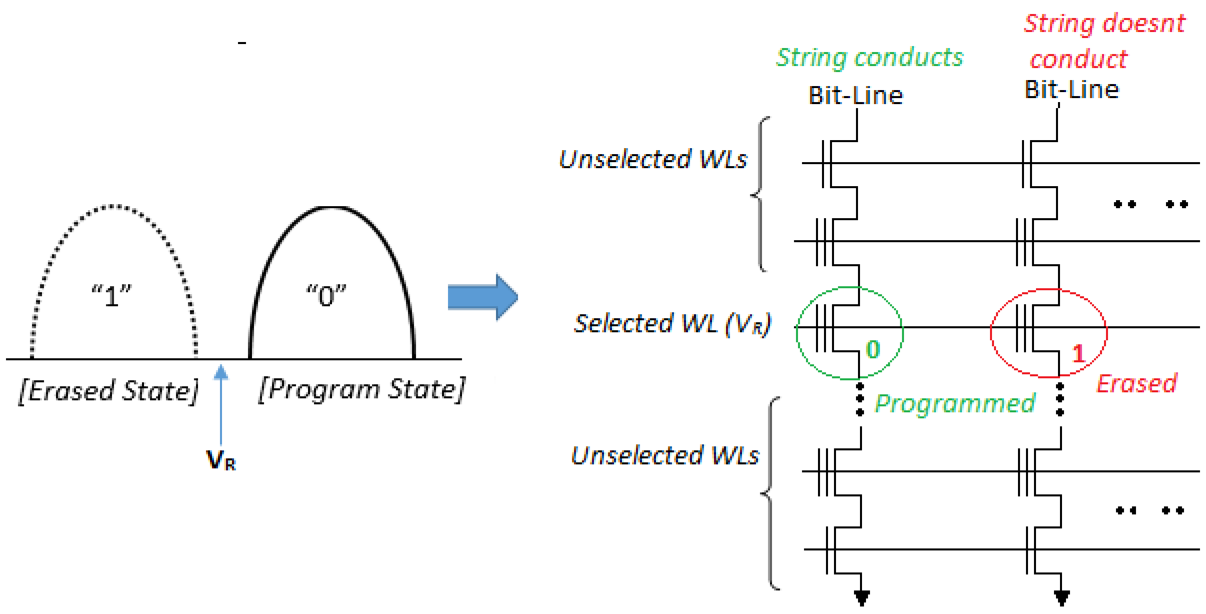


Fig 5: Read Operation

2.6 Erase Operation

Erase is done at a block level. Thus the substrate is biased to a high voltage while the gate voltage is kept low, thus forcing the electrons to move from the Floating gate and back into the substrate, thus erasing the charges stored in the FG.

Chapter 3: Error Correcting Codes

3.1 Introduction

Error Correction techniques are used to correct bit-errors in memories. Error correction adds redundancy in the form of check-bits, to detect and correct errors. For an m-bit message, if k-bits of redundancy are added, there is a codeword, which is m+k (=n) bits wide. Out of the total 2^Npossible combinations, some would be valid codewords (i.e. having no errors) while the rest would be non-codewords (i.e. having an error). "*Error detection*" thus happens when a non-codeword is detected. For "*Error Correction"* to happen, it needs to be determined, which of the codeword bits is the error. As an example consider a 3-bit wide code, which has only 2 valid codewords: 000 and 101. If the data after the memory read-out if 100, then an error has been detected. The error cannot be corrected since the codeword written into the memory could have been 000 or 101.

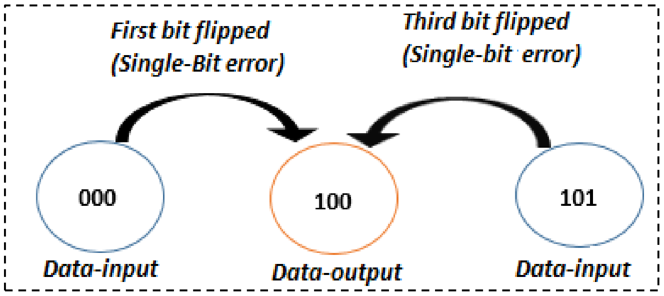


Fig 6: Error Detection Example

However if the data after the memory read-out is 010, then Error correction is possible if the number of errors that occurred is known. If it was a single-bit error, then the codeword written into the memory was 000. If it was a multi-bit error occurred, then the actual codeword was 101 and so error correction can be accordingly done.

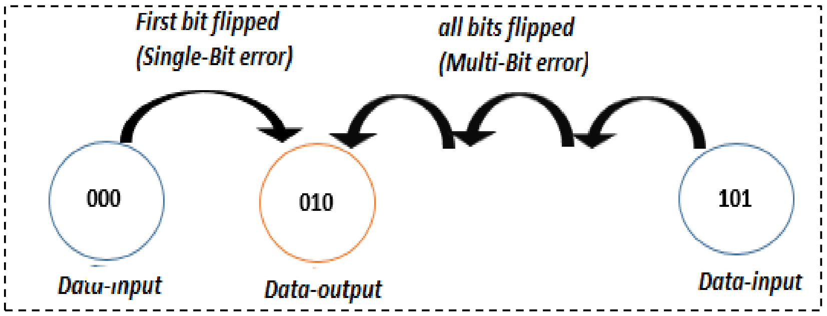


Fig 7: Error Correction Example

The Generator Matrix (G-Matrix) is used to generate codewords. For a linear separable code, the Generator Matrix can be expressed as

- G = [I: P], where I is identity and P is parity matrix.

The Parity-Check Matrix (H-Matrix) is used to detect and correct errors. The rows of the H-Matrix correspond to parity check equations and the columns correspond to the codeword bits. The parity check matrix can be expressed as

- H = [Q:I] where Q = Transpose(P)

Below diagram shows the parity-check matrix and the corresponding parity-check equations.

$$\mathbf{H} = \begin{array}{cccccc} c0 & c1 & c2 & c3 & c4 & c5 \\ \end{array}$$

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \begin{array}{l} \longrightarrow C0 \oplus C2 \oplus C4 = 0 \\ \longrightarrow C0 \oplus C1 \oplus C5 = 0 \\ \longrightarrow C1 \oplus C3 \oplus C5 = 0 \end{array}$$

Fig 8: Parity Check Matrix and Equations

If C is the code-word and H is the parity-check matrix then,

- Syndrome = CH^T is calculated

If syndrome is all 0’s, all the parity check equations have been satisfied and there are no errors in the codeword. If syndrome is not 0, then the parity check equations have not been satisfied, and an error has occurred.

An ECC encoder is used to generate the codeword bits by adding parity bits to the user data. This data is then written into the memory. Once the data has been read-out, an ECC decoder is used to check and detect errors. The decoder makes use of the user bits, parity bits and the parity-check matrix (H-Matrix) to detect errors.

3.2 Parity and Hamming Codes

Parity code is the simplest error detection code. An extra bit is added to the message bits such that the parity of the codeword bits is even (or odd). Once the message has been received, the parity of the codeword bits can be calculated to check if it is even (or odd). In case the parity has changed, it means that one of the bits has flipped. Though the implementation of a parity code is simple, it can only detect a single error. If two bits flip, then the parity would be unchanged and so the errors would go undetected.

Hamming codes can correct single-bit errors and detect double-bit errors (SEC-DED). A single-error correcting Hamming code (SEC) has an H-Matrix with no repeated columns and no all-zeroes column. A single-error correcting and double-error detecting Hamming code (SEC-DED) has a similar H-Matrix but with additional parity bits. Reed-Solomon and BCH codes are used to correct multi-bit errors.

3.3 Bose, Chaudhuri, and Hocquenghem (BCH) codes

Bose, Chaudhuri, and Hocquenghem (BCH) codes are cyclic error correcting codes, capable of correcting multiple-bit errors in NAND Flash memories [7] [8]. Cyclic codes have the special feature that the cyclic shift of a valid codeword results in another valid codeword. Generation of BCH codes involves two major steps:

Step1: Constructing a generator polynomial G(x)

Step2: Encoding the data.

The generator polynomial is used to encode a k-bit message into an n-bit BCH codeword.

3.3.1 Construction of a BCH generator polynomial:

For an integer $m \geq 3$ and $t < (2^{m-1})$ a BCH code can be defined as having

- Block length of n bits ($n = 2^m - 1$)
- Capable of correcting "t" error bits.
- The number of check bits is $(n\text{-}k) \leq mt$
- Minimum distance is $\geq (2t + 1)$

To construct this BCH code, a generator polynomial G(x) is required.

G(x) is defined as L.C.M $[\alpha, \alpha^3, \alpha^5 \ldots \alpha^{2t-1}]$ where α is a polynomial in the Galois Field (2^m). For example, for a 3-bit error correcting BCH code, with a block length of 15:

- $t = 3$, $n = 15$, $m = 4$
- Consider a polynomial in the GF (2^4)

  $\alpha = 1 + x + x^4 \rightarrow \alpha^3 = 1 + x + x^2 + x^3 + x^4 \rightarrow \alpha^5 = 1 + x + x^2$
- Thus, $G(x) = \text{L.C.M} [\alpha, \alpha^3, \alpha^5] = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$

This generator polynomial can be used to generate a BCH code, which has the following features: Block length (n) = 15; Number of Check Bits (n-k) = 10; Number of message bits (k) = 5

3.3.2 BCH Encoding

Messages can be encoded into codewords using the Generator polynomial. If the number of message bits is (k), and the degree of the Generator polynomial is (n-k), then the resultant codeword will be n bits wide. The encoding can be done in one of the following ways:

- Multiplicative encoding (Galois multiplication)
- Separable encoding (Galois division)

Multiplicative encoding

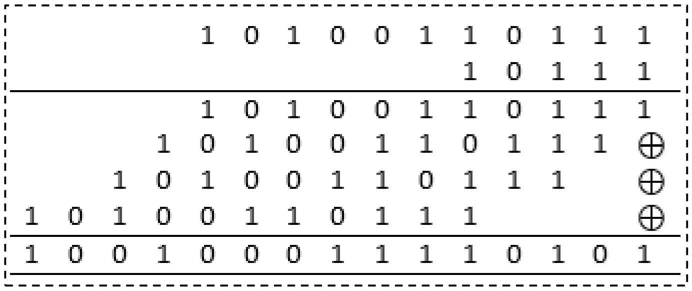
To encode the message m(x) into a codeword n(x) using a generator polynomial G(x), Galois multiplication is performed between m(x) and G(x). Galois multiplication is similar to conventional binary multiplication, except that an XOR operation is done on the bits instead of addition. Thus at every step of the multiplication, the bits are shifted left, followed by an XOR operation.

As an example,

Message m (x) = $x^4 + x^2 + x + 1$ = [1 0 1 1 1]

G(x) = $x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$ = [1 0 1 0 0 1 1 0 1 1 1]

Performing Galois (modulo-2) multiplication on them, yields the encoded codeword, which is 15 bits wide: [1 0 0 1 0 0 0 1 1 1 1 0 1 0 1].



```
        1 0 1 0 0 1 1 0 1 1 1
                    1 0 1 1 1
-----------------------------
        1 0 1 0 0 1 1 0 1 1 1
      1 0 1 0 0 1 1 0 1 1 1   ⊕
    1 0 1 0 0 1 1 0 1 1 1     ⊕
1 0 1 0 0 1 1 0 1 1 1         ⊕
-----------------------------
1 0 0 1 0 0 0 1 1 1 1 0 1 0 1
-----------------------------
```

Fig 9: Multiplicative BCH encoding

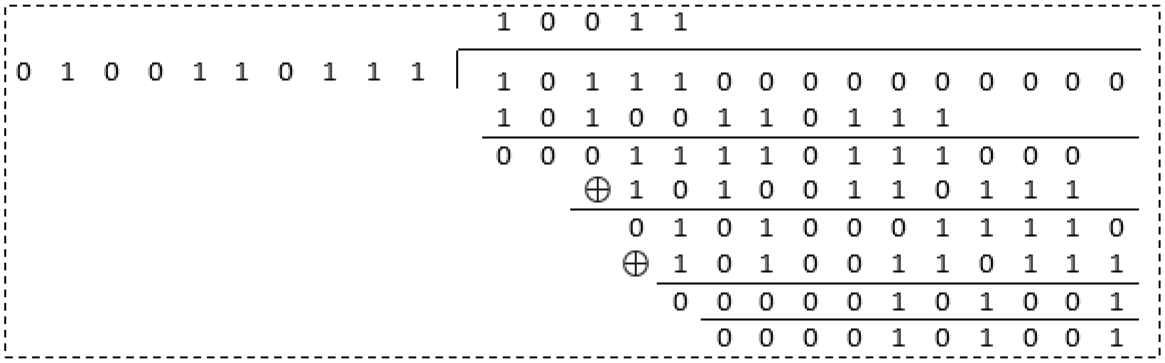
Separable encoding

Codewords obtained by multiplicative encoding are not separable in that, the message bits cannot be directly obtained from the codeword. Thus, once the codeword has been received, it is not possible to separate out the message bits from the check bits. To get separable codewords, Galois division can be performed instead of Galois multiplication.

To encode a message m(x) that is k bits wide, using a generator polynomial G(x) that is (n-k) bits wide, append the message with (n-k) number of zeroes. Then perform Galois division with the appended message as the Dividend and G(x) as the Divisor. The remainder is then concatenated with the message to obtain the encoded codeword.

In the following example,

- Message m (x) = $x^4 + x^2 + x + 1$ = [1 0 1 1 1] has been appended with 10 zeroes.
- G(x) = $x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$ = [1 0 1 0 0 1 1 0 1 1 1] is the divisor



```
                                 1  0  0  1  1
                               ______________________________________________
0  1  0  0  1  1  0  1  1  1  |  1  0  1  1  1  0  0  0  0  0  0  0  0  0  0
                                 1  0  1  0  0  1  1  0  1  1  1
                                 _________________________________________
                                 0  0  0  1  1  1  1  0  1  1  1  0  0  0
                                      ⊕  1  0  1  0  0  1  1  0  1  1  1
                                      ____________________________________
                                         0  1  0  1  0  0  0  1  1  1  1  0
                                          ⊕  1  0  1  0  0  1  1  0  1  1  1
                                             _________________________________
                                             0  0  0  0  0  1  0  1  0  0  1
                                                _______________________________
                                                0  0  0  0  1  0  1  0  0  1
```

Fig 10: Separable BCH encoding

Concatenating the message and the remainder the codeword is obtained as

Codeword = [1 0 1 1 1 0 0 0 0 1 0 1 0 0 0 1]

Since the final codeword is obtained by concatenating the message and the remainder, the final codeword is separable.

3.3.3 BCH Decoding

To decode the codeword, Galois division is performed on the received codeword using α, which is the original polynomial in GF (2^m), which was used to construct G(x). The received codeword r(x) is divided (Galois division) using each of α, $α^2$, $α^3$ …. α $^{(2t-1)}$. The remainders obtained are called as Syndromes. A zero syndrome indicates no error, while a non-zero Syndrome indicates that an error has occurred.

- For the above obtained separable codeword, [1 0 1 1 1 0 0 0 0 1 0 1 0 0 0 1] the generator polynomial used was G(x), which was constructed using α = 1 + x + x^4 = [10011].
- Thus if the received codeword does not have any errors,
  Then r(x) = [1 0 1 1 1 0 0 0 0 1 0 1 0 0 0 1]
  Performing Galois division using α = [10011] as the divisor, the Syndrome was 0 and so it confirms that no error has occurred. Similarly when Galois division is performed using multiples of α (x) also, the Syndrome is 0.
- If however the received codeword had one of the bits flipped, then the Syndrome will be non-zero. If the received codeword is [1 0 1 1 1 0 **1** 0 0 1 0 1 0 0 0 1] (7th bit flipped),

then performing Galois division using α = [10011] as the divisor, yields a non-zero Syndrome as shown below. Similarly when Galois division is performed using multiples of α (x) also, the Syndrome is non-zero, confirming that the received codeword has errors.

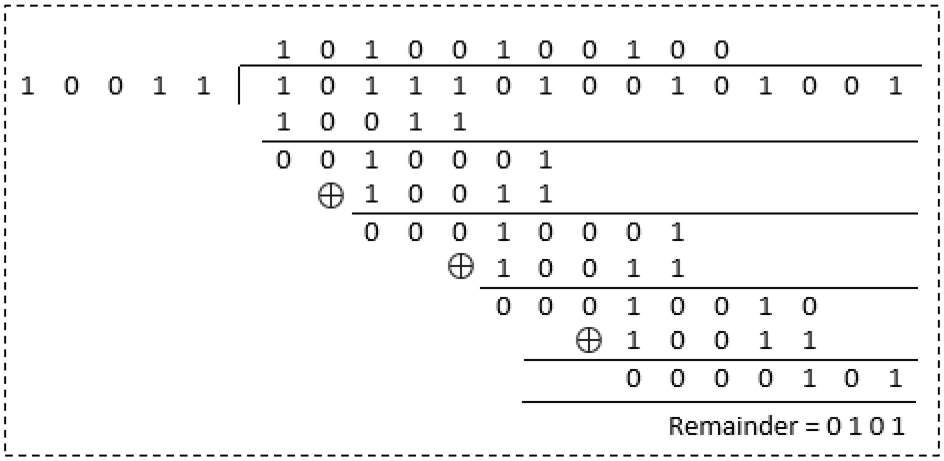


Fig 11: BCH Syndrome calculation

To correct errors, information is needed about the error locations. To find out the error locations, the non-zero syndromes are expressed as equations and an error location polynomial is constructed from these equations. Algorithms like the Berlekamp-Massey algorithm [9] [10] or Euclid's algorithm are used to find out the error location polynomial. Then a Chien search is done to find out the roots of this polynomial. Once the error locations are known, the errors can be corrected by flipping the bits.

3.4 Reed-Solomon Codes

Reed-Solomon Codes [11] are linear block codes, which are a special class of BCH codes.

- BCH codes: $(n - k) \leq mt$
- Reed-Solomon codes: $(n – k) = 2t$

Thus, for a k-bit message, 2t check bits are added to generate a Reed-Solomon codeword of size n-bits. Such a code can correct t-bits of errors. The minimum distance for an RS code is $2t + 1 = (n - k +1)$ and because of this Reed-Solomon codes are optimal since the minimum distance (n - k + 1) is the maximum possible value for a linear code of block length n.

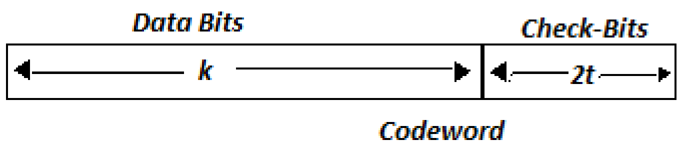


Fig 12: Reed-Solomon Codeword

Reed-Solomon codes can also be adapted to a non-binary version by using symbol based encoding. Each symbol is defined to be m-bits wide. For a message that is km-bits wide, the message is broken down into k symbols. Analogous to the binary version, 2t check symbols are added and encoded using the generator polynomial The result is a codeword that is n = k + 2t symbols wide (each symbol containing m-bits).

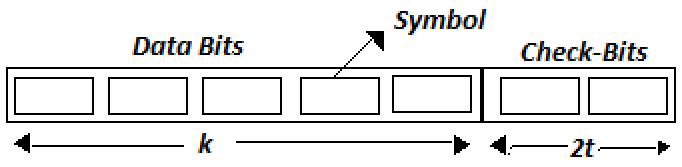


Fig 13: Reed-Solomon Codeword split into symbols

The advantage of symbol based encoding is that they can be used to correct burst errors. This is because if a symbol contains an error, than the entire symbol is corrected. So for burst errors, this is an efficient correction scheme.

3.4.1 R/S Encoding

The construction of the generator polynomial for a RS code is different from the case of BCH codes.

- G(x) is defined as $(x + \alpha)\ (x + \alpha^2) \ldots.\ (x + \alpha^{2t})$

Similar to BCH codes, the generator polynomial can be used to generate codewords by encoding message bits using Galois multiplication or division.

3.4.2 R/S Decoding

The decoding process for Reed-Solomon codes is similar to BCH codes. Syndromes are calculated using α. A non-zero Syndrome indicates an error, in which case the Berlekamp-Massey algorithm is used to find out the error location polynomial. For Reed-Solomon codes, where symbol based encoding is used, each symbol contains m-bits. Thus in addition to knowing the error location, information is also needed about the error values (since simple bit flipping will not work since each symbol is m-bits wide). Thus in cases of non-binary BCH and Reed Solomon codes advanced algorithms like Forney algorithm are used to find out the error values. Upon getting the error locations and error values, the errors can be accordingly corrected.

3.5 Low Density Parity Codes (LDPC)

3.5.1 Introduction

Low density parity codes are a class of linear block error correcting codes. The distinguishing feature of these codes is that they have sparse H-Matrices. For large block lengths, they have a very high performance rate, which is close to the Shannon Capacity and for this reason, they are being increasingly used for error correction purposes. Low Density Parity Codes were first proposed by Robert G. Gallagher [12] in his doctoral defense at MIT. Although LDPC codes were first proposed by Prof. Gallagher in 1960, they weren’t used for a long time because of the decoding complexity involved. With today’s advanced computational resources, it is possible to implement LDPC codes for error correction purposes.

The special property about LDPC codes is that the H-Matrix is not dense i.e. the number of 1’s in the matrix is very less. Typically less than 1% of the matrix consists of 1’s while the rest are 0’s.

LDPC codes can be categorized into regular LDPC codes and irregular LDPC codes. Regular LDPC codes have constant weight rows and constant weight columns i.e. they have the same number of 1’s in each row (and each column). Irregular LDPC codes relaxes the constant weight condition, and in doing so the rows and columns can have varying number of 1's. Irregular LDPC codes are known to perform better than regular LDPC codes because the number of 1's can be skewed in such a manner to provide optimal distribution across the codeword bits. Usually, long and irregular LDPC codes are known to provide a higher degree of protection.

LDPC codes are especially suitable for usage in NAND Flash memories, because of their ability to process both hard-bit (binary) and soft-bit (probabilities of bits) information. In case, the LDPC decoder cannot correct all the errors using the hard-bit information, it tries to correct the errors using soft-bit information. Thus, it works well when both soft-bit and hard-bit information are available. NAND Flash memories can be accessed to get both hard-bit and soft-bit data, and this information can be passed to the LDPC decoding circuit to detect and correct errors .With the added benefit of being able to process soft-information as well, LDPC allows the correction of more errors for the same Data-to-Parity bits ratio when compared to other ECC schemes. Thus, with the availability of both hard-bit and soft-bit data (as is the case in NAND Flash memories), the performance of LDPC codes is quite close to the theoretical limits i.e. the Shannon Capacity Limit.

3.5.2 LDPC H-Matrix

The Parity Check matrix (the H-Matrix) consists of parity check equations and is used to check if the received codeword has any errors or not. The special feature of LDPC codes is that the corresponding Parity-Check matrix (H-Matrix) has very few number of 1's and hence is sparse or "less-dense". This is advantageous while detecting errors for the following reasons-

- Since the rows of the H-Matrix correspond to the parity check equations and the columns correspond to codeword bits, a low number of 1's in the H-matrix means that the parity check equations would be dependent on few codeword bits only. Thus if a parity check equation fails, it is relatively easier to figure out which of the codeword bits had the error because of the very few number of 1's in the check equation. Thus, the number of iterations required to flip all the erroneous codeword bits is less when compared to other ECC schemes.

- In codes having dense H-Matrices, each parity check equation is dependent on a large number of codeword bits. So if the equation fails, it is difficult to find out which codeword bit caused the failure. In case of a sparse H-Matrix, if a parity check equation fails, it is relatively easier to figure out which of the bits caused the failure.
- Because of the sparseness of the H-Matrix, if a codeword bit is present in multiple failing parity check equations, it greatly increases the probability that this particular codeword bit is erroneous. LDPC decoding algorithms rely on this reasoning to detect and correct errors.

$$\begin{array}{cl}
\begin{pmatrix}
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
&
\begin{array}{l}
\xrightarrow{P2} \textit{P2 Fails} \rightarrow \textit{Either C4 or C8 is erroneous} \\
\\
\\
\\
\xrightarrow{P7} \textit{P2 and P7 Fails} \rightarrow \textit{High probability C8 is erroneous}
\end{array}
\end{array}$$

Fig 14: LDPC H-Matrix

3.5.3 Tanner Graph

A Linear block code can be described in one of the following ways:

- Parity-Check Matrix/ Generator Matrix
- Graphical representation ("Tanner Graph")

LDPC codes are often represented in the form of Tanner Graphs [13]. Tanner Graph is a bipartite graph, which has two types of nodes:

- *Bit Nodes* – These represent the codeword bits
- *Check Nodes* – These represent the parity check equations.

A line is used to connect a *bit node* and a *check node* only if that particular codeword bit appears in the corresponding check equation.

Tanner Graphs are heavily used in LDPC decoding algorithms. The way these algorithms work is that, messages are iteratively exchanged between the bit nodes and the check nodes, along the edges of the graph. Messages are initially sent from the bit nodes to the check nodes broadcasting the values of the bit nodes. Each check node, upon receiving information from all its connected bit nodes, uses its check equation to estimate new values for each of the bit nodes, and broadcasts this information back to the bit nodes. Each of the bit nodes, then updates its own value if required, and sends the information back to the check nodes. This process is continued iteratively, wherein the messages are then sent back and forth between the bit nodes and check nodes until all the parity check equations have been satisfied.

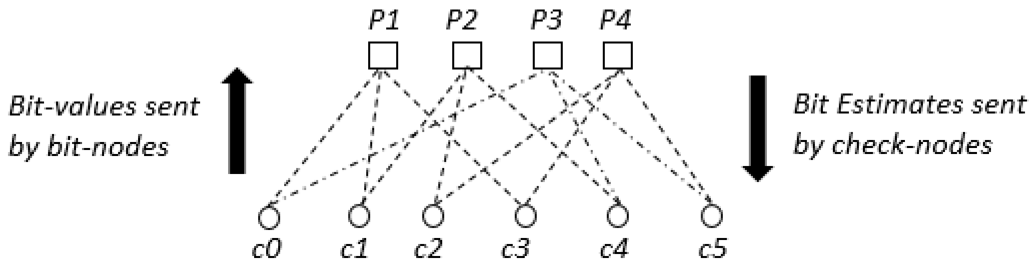

P1
P2
P3
P4
Bit-values sent by bit-nodes
Bit Estimates sent by check-nodes
c0
c1
c2
c3
c4
c5

Fig 15: Tanner Graph

3.5.4 Decoding Algorithms

LDPC decoding algorithms are used to detect and correct errors in received codeword bits. They can be broadly classified into two main categories:

- Hard Decision Decoding Scheme
    - Uses hard bit (binary) information (‘1’s and ‘0’s) to detect and correct errors.
    - To correct an erroneous bit-node, the algorithm flips the bit.
- Soft Decision Decoding Scheme
    - Uses soft bit information (probability of the bit being ‘1’ or ‘0’) to decode.
    - Here the bit-nodes contain soft-bit data, which is usually the probability of the bit being ‘1’ or ‘0’.
    - Most commonly, these probabilities are represented in the form of LLR’s.
    - This is suitable for NAND Flash memories since soft-bit information can be obtained during a NAND Flash read.

These algorithms are iterative in nature, since the messages are passed around between the bit nodes and check nodes iteratively till the process is completed. Also, as discussed above, since the information is passed from the bit-nodes to the check-nodes in a Tanner Graph, these algorithms are also known as *message-passing* algorithms.

3.5.5 Hard-Decision Decoding: *Bit-Flipping Decoding Algorithm*

The Bit-Flipping decoding scheme is a hard-decision decoding scheme, which directly utilizes binary information of the codeword bits. Once the decoding circuitry receives the codeword bits, it passes them along to the check nodes.
Shown below is an H-Matrix containing 6 codeword bits and 4 parity check equations.
A codeword C [0 1 1 0 1 1] has been received for decoding. As can be seen, the Syndrome is non-zero and hence the codeword C contains errors.

$$[0\quad 1\quad 1\quad 0\quad 1\quad 1] \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} = [\ 1\quad 1\quad 0\quad 0\ ]$$

$$CH^T = Syndrome$$

Fig 16: LDPC Syndrome calculation

The steps for decoding the above codeword using bit-flipping decoding scheme are:

Step1:

Each bit-node sends its bit value to all its connected check nodes along the edges of the Tanner graph. The Tanner graph has 6 bit-nodes and 4 check-nodes. At the start of the decoding algorithm, the 6 codeword bits [0 1 1 0 1 1] are sent to the check-nodes P1, P2, P3, P4.

$$\mathbf{H} = \begin{array}{cccccc|c} c0 & c1 & c2 & c3 & c4 & c5 & \\ 1 & 1 & 0 & 1 & 0 & 0 & P1 \\ 0 & 1 & 1 & 0 & 1 & 0 & P2 \\ 1 & 0 & 0 & 0 & 1 & 1 & P3 \\ 0 & 0 & 1 & 1 & 0 & 1 & P4 \end{array}$$

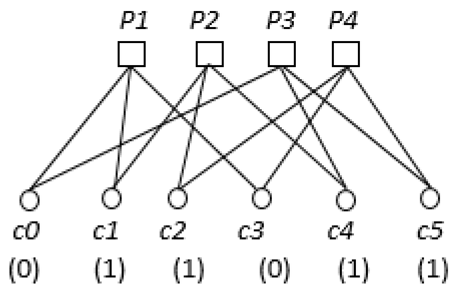


Fig 17: LDPC H-Matrix and corresponding Tanner Graph

Step2:

Each check node calculates an estimate for each of its connected bit-nodes. To calculate the estimate for a bit-node, the check-node makes the following assumptions

(a) The check equation is passing (i.e. modulo-2 sum of all the connected bit-nodes is 0)

(b) The estimates received from the other connected bit-nodes is correct.

In the following example, check-node P2 is connected to bit-nodes C1, C2, C4.

- It assumes its check equation is passing ➔ C1 ⊕ C2 ⊕ C4 = 0.
- To calculate an estimate for C1, it assumes values sent by C2 and C4 are correct. Thus since C2 =1, C4 =1, then it estimates C1 to be 0.
- Similarly, to estimate C2, it assumes C1 and C4 values are correct, and calculates C2 to be a 0, and sends it back.

This way, all the check-nodes send back estimates for each of their connected bit-nodes. Following diagrams show the estimates sent back by each of the check nodes.

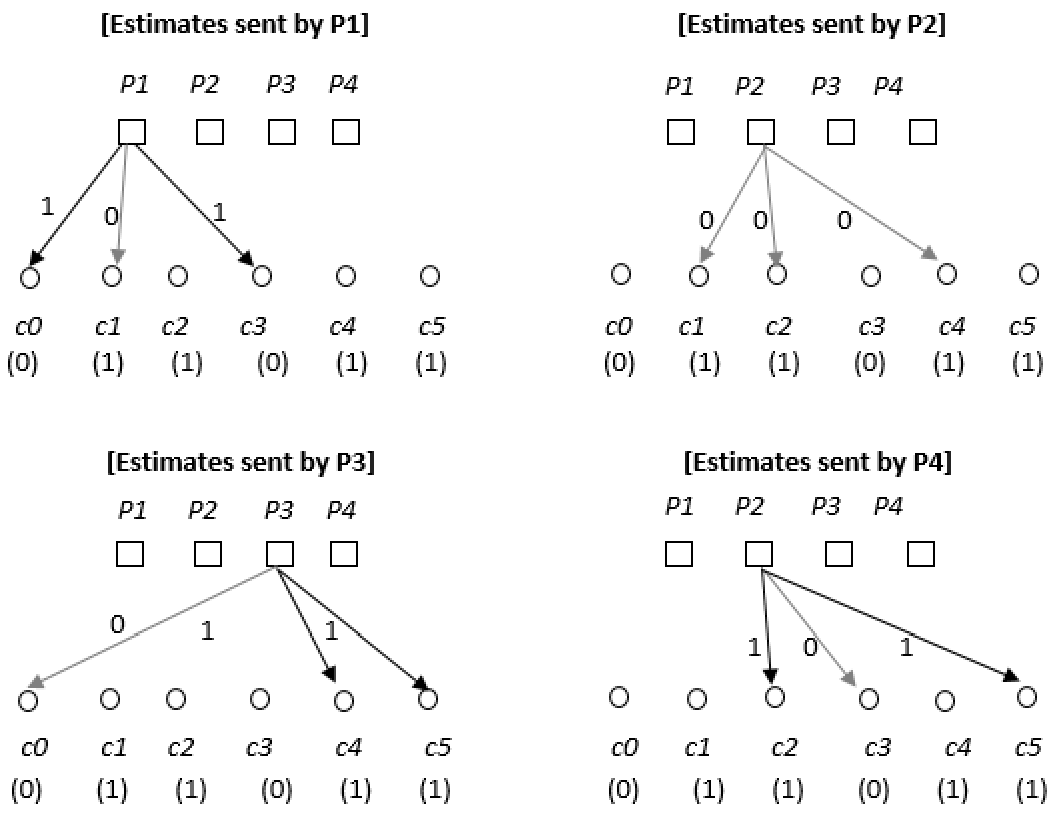


Fig 18: Check nodes calculating bit estimates

Step 3:

Each bit-node thus receives estimates about its value from each of its connected check-nodes. Then each bit-node does a majority vote (including the initial codeword bit). This is the new bit-node value. Thus bit C1, is flipped from 1 to a 0.

Initial Codeword Bits	**c0**	**c1**	**c2**	**c3**	**c4**	**c5**
	0	**1**	**1**	**0**	**1**	**1**
Estimate sent by P_1	1	0	-	1	-	-
Estimate sent by P_2	-	0	0	-	0	-
Estimate sent by P_3	0	-	-	-	1	1
Estimate sent by P_4	-	-	1	0	-	1
Majority Vote	**0**	**0**	**1**	**0**	**1**	**1**

Fig 19: Majority voting at bit-nodes

The algorithm works on the premise that a bit-code that is present in more failing check-equations is more likely to fail. Thus, it flips bits that are present in more failing check-equations. Considering the above example, the check equations at check nodes P1, P2, P3, P4 are calculated:

P1 Check Equation → $C0 \oplus C1 \oplus C3 = 0 \oplus 1 \oplus 0 = 1$ → Failing Equation (**X**)

P2 Check Equation → $C1 \oplus C2 \oplus C4 = 1 \oplus 1 \oplus 1 = 1$ → Failing Equation (**X**)

P3 Check Equation → $C0 \oplus C4 \oplus C5 = 0 \oplus 1 \oplus 1 = 0$ → Passing Equation (✓)

P4 Check Equation → $C2 \oplus C3 \oplus C5 = 1 \oplus 0 \oplus 1 = 0$ → Passing Equation (✓)

Codeword	C0	C1	C2	C3	C4	C5
Equation1	**X**	**X**	**X**	**X**	**X**	✓
Equation2	✓	**X**	✓	✓	✓	✓

Code-bit C1 is present in two failing equations, and so is more likely to fail, and so the bit is flipped. Originally C1 was 1, so the bit is now flipped to a 0.

The new codeword is now [0 0 1 0 1 1] and it can be verified from the equation HC^T that this codeword is valid and does not contain any errors.

$$[0\ \ 0\ \ 1\ \ 0\ \ 1\ \ 1] \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} = [\,0\ \ 0\ \ 0\ \ 0\,]$$

$CH^T = \text{Syndrome}$

Fig 20: LDPC syndrome calculation after Bit-Flipping decoding

- Originally:

  Codeword is [0 1 1 0 1 1] → HC^T is non-zero → contains error

- After Bit-Flipping decoding:

  Codeword is [0 0 1 0 1 1] → HC^T is zero → contains no error

3.5.6 Soft-Decision Decoding Algorithms

Soft-Decision Decoding Algorithms [14] are similar to the Hard-Decision decoding algorithms, except that they accept probabilities of codeword bits being ‘1’s and ‘0’s, rather than binary information. The advantage in processing probabilities rather than actual binary information is that, probabilities give more information about the bit. If the probability of a bit is close to 0.5, then it is more likely that the bit could be an error since it could be either a ‘1’ or ‘0’. On the other hand, if the probability is very close to 1 (or very close to 0) it increases the confidence that the bit is correct. Decoding schemes that use the probabilities of bits are also known as *belief-propagation* schemes since the belief is propagated that the bit is a ‘1’ or ‘0’, rather than transmitting the bit information itself.

Two common soft-decision decoding schemes are:

- Sum-Product Algorithm
- Min-Sum Algorithm

3.5.7 Soft-Decision Decoding Scheme: *Sum-Product Decoding Algorithm*

Sum-Product Decoding Algorithm is a soft-decoding scheme, which accepts probabilities of bits being ‘1’s or ‘0’s rather than hard-decisions. The algorithm is similar to the bit-flipping algorithm except that it is the probabilities that are being exchanged. The steps in this algorithm are as follows:

- *Step 1*

  Each bit-node receives information about its probability of being a ‘1’, and passes it along to the connected check-nodes.
- *Step 2*

  Similar to the bit-flipping algorithm, each check-node then calculates the probability of a check node being a ‘1’ assuming that (a) its check equation is passing (b) the probabilities of the other connected bit-nodes are correct.

  The estimated probability for a bit-node B_X is calculated as follows:

- Let the check node be connected to bit-nodes B_1, B_2, B_3 …. B_N and B_X, and P_i denote the probability of B_i being a 1.
- The parity check assumes its equation is passing → $B_1 \oplus B_2 \oplus B_3 \oplus \ldots B_N \oplus B_X = 0$
- For 2 nodes,

  $P[(B_1 \oplus B_2) = 0] = P_1P_2 + (1-P_1)(1-P_2) = 1 - P_1 - P_2 + 2P_1P_2$

  $P[(B_1 \oplus B_2) = 1] = P_1 + P_2 - 2P_1P_2 = P_{12}$
- For 3 nodes,

  $P[(B_1 \oplus B_2 \oplus B_3) = 0] = P[((B_1 \oplus B_2) \oplus B_3) = 0] = P_{12}P_3 + (1-P_{12})(1-P_3)$

  $= 1 - (P_1 + P_2 - 2P_1P_2) - P_3 + 2P_1P_3 + 2P_2P_3 - 4P_1P_2P_3$

  $= 1/2\,[1 + (1 - 2P_1)(1 - 2P_2)(1 - 2P_3)]$
- Similarly for n nodes,

  $P[(B_1 \oplus B_2 \oplus B_{3 \ldots} \oplus B_N) = 0] = 1/2\,[1 + \pi(1 - 2P_i)]$

  Given $P[(B_1 \oplus B_2 \oplus B_{3 \ldots} \oplus B_N \oplus B_X) = 0] = 1$ → $P[B_X = 1] = 1/2\,[1 - \pi(1 - 2P_i)]$
- Below are given two examples of how the probability estimates are calculated. In the first case, bit-node probabilities are [0.9 0.2 0.7] that correspond to the codeword bits [1 0 1]. These satisfy the parity equation and this is reflected in the estimated probabilities, which are sent back by the check node.

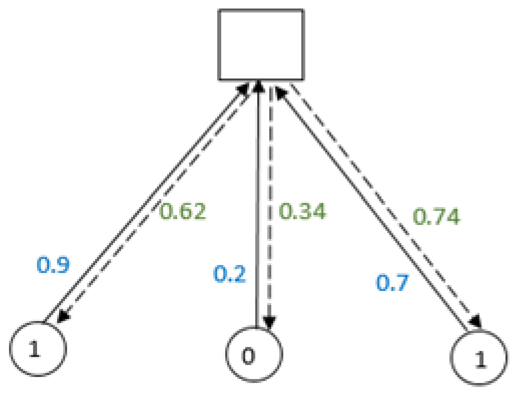


Fig 21: Calculation of probability estimate when parity equation is satisfied

- In the second case, probabilities are [0.9 0.2 0.3] that correspond to the codeword bits [1 0 0]. This set doesn't satisfy the equation, and thus the check-node tries to flip all the bits.

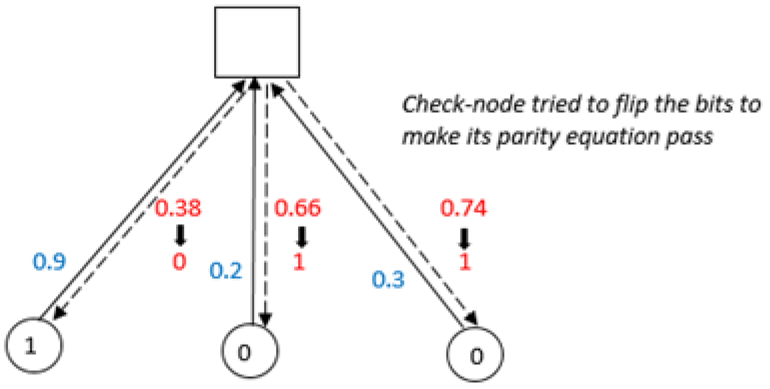

Check-node tried to flip the bits to make its parity equation pass
0.38
0
0.66
1
0.74
1
0.9
0.2
0.3
1
0
0

Fig 22: Calculation of probability estimate when parity equation is not satisfied

- *Step 4*

  Each bit-node re-calculates its own probability taking into account the estimated probabilities sent by each of the connected check-nodes.
- This way the process continues iteratively until all the check equations are satisfied or a maximum number of iterations has been reached. Then the probabilities of each of the bit-nodes are outputted.
- In the decoding algorithm, at both the check-nodes and bit-nodes probabilities are multiplied. Since multiplication is a complex process, an easy way to get around this is to express the probabilities as logarithmic ratios. This way, the product of probabilities can be achieved as the sum of logarithmic values. Thus decoding algorithms that operate on soft-bit information expressed in the form of logarithmic values are known as *Sum-Product algorithms*.
- The probability of a bit being '1' or a '0' can be expressed as a Log-Likelihood Ratio (LLR).

A Log Likelihood ratio can be expressed as:

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

Where x is the bit programmed into the cell, and y is the output after a memory-read

- A positive LLR indicates that the bit is more likely to be a 0, while a negative LLR indicates it is more likely to be a 1.
- The absolute value of the LLR gives the relative confidence in the bit being a '1' or a '0'. The higher the magnitude of the LLR, the more likely the bit is a '1' (if the LLR is negative) or a '0' (if the LLR is positive).
- In a Sum-Product Decoding schemes, LLRs are passed along the edges of the Tanner graph. The bit-nodes send the initial LLR values to the check-nodes. The check nodes then send back the estimated LLR value to the bit-nodes. At the bit-node, all the LLRs are added and the resultant LLR is calculated. For example, if the initial LLR of a bit is +0.72 and it receives LLR values of -0.81 and -0.63 from two check-nodes, the resultant LLR will be -0.72. In essence, this means that the bit has flipped from '0' to '1'.
- Sum-Product decoding schemes are specially suitable for NAND Flash memories, since soft-bit data read out of these memories is often expressed in the form of LLRs. Thus, this decoding scheme is increasingly being used for error correction in NAND Flash memory.

3.5.8 Soft-Decision Decoding Scheme: *Min-Sum Algorithm*

The Min-Sum algorithm [15] is similar to the Sum-Product Algorithm, but has reduced complexity. It simplifies the calculation of the LLR of the estimated probabilities at the check-nodes, by approximating product of terms with the minimum term. Thus at the check-nodes, the minimum is considered while at the bit-nodes, the sum of LLRs is taken and so it is called the *min-sum algorithm*. It simplifies complex calculations by trading-off with small performance loss. For this reason, this algorithm is being increasing implemented in NAND Flash memories.

3.5.9 Hard-decision sensing and Soft-decision sensing

LDPC codes can perform error correction using hard-bit and soft-bit data.

- Hard-bit information is the binary output obtained after a memory read operation.
- Soft Bit information tells the probability of a bit being a ‘1’ or a ‘0’ and is usually represented in the form of Log-Likelihood-Ratios (LLR).
- In NAND Flash memories, Soft-bit information is obtained by performing multiple sense operations at different read levels.

To read data out of a NAND cell, the signal is sensed at a level (V_R), which is in between two adjacent threshold voltage distributions. The Word-line voltage is placed at V_R level. If the V_T of the cell is less than V_R then the cell conducts, otherwise it does not. The Bit-Line charging/discharging is sensed to find out if the cell has conducted or not. Thus, the result of such a sensing gives the data that is stored in the NAND cell.

However due to various reasons such as Cell-to-Cell interference or over-programming, in some of the NAND cells, the V_T distributions might overlap. In such a case, a V_R sense might give incorrect data. In such a case, sensing at more than one level gives more accurate information about the data in the NAND cell. *Figure-23* shows such a case, where sensing is done at 3 levels. The signal is sensed at V_1, followed by V_2 and V3. This tells if the V_T of the NAND cell, belongs to region A, B, C or D. The information extracted out of such a multi-sensing scheme is called soft-bit information and is used for the LDPC soft-decision decoding. Soft-bit information is often represented in the form of Log-Likelihood Ratios (LLR) [16], which are a logarithmic representation of the probabilities. Mathematically, an LLR can be represented as:

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

Where x is the bit programmed into the cell, and y is the output after the 3 sense-read.

- If y is in region C or D, it is more likely that x is a 0 → LLR is positive
  → The more positive the LLR, the more likely that the bit read out is a 0.
- If y is in region A or B, it is more likely that x is a 1 → LLR is negative
  → The more negative the LLR, the more likely that the bit read out is a 1.

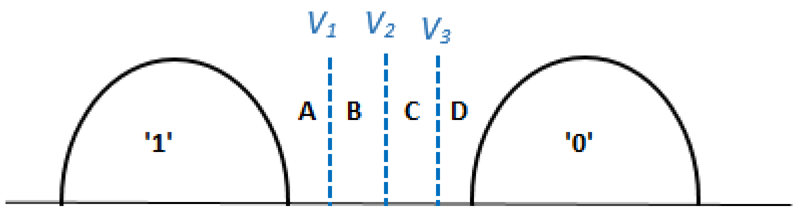


Fig 23: Sensing at multiple levels

3.5.10 LDPC: Hard-Decision and Soft-Decision Decoding schemes

- Extracting soft-bit data requires multiple senses → Added latency
- It also increases the latency during data transfer out of the flash because of the additional data (hard-bit and soft-bit data) that is transferred.

Thus, Soft-bit decoding schemes have a higher latency and so to counteract this, both LDPC Hard-Bit and Soft-Bit decoding schemes are used for error-correction.

Typically these are the steps employed are:

1. A single sense at V_R level is done to get the hard-bit data, which is transferred from the Flash to the ECC circuitry.
2. A soft-bit read (multiple-sense read) is done to get the soft-bit information.
3. An LDPC Hard-Bit decoding scheme (such as Bit-Flipping scheme) is used to detect and correct errors.
4. In case all the errors have not been corrected, then LDPC Soft-Bit decoding schemes are used to detect and correct errors.
5. If these schemes are successful then the Memory Read is successfully completed, else the Read Fails.

During initial lifetime of the chip, when the number of P/E cycles is low, Hard-Bit decoding schemes might be sufficient to correct all errors. So initially Hard-Bit Decoding schemes are used, and after a certain number of P/E cycles has been crossed Soft-Bit decoding schemes will be used.

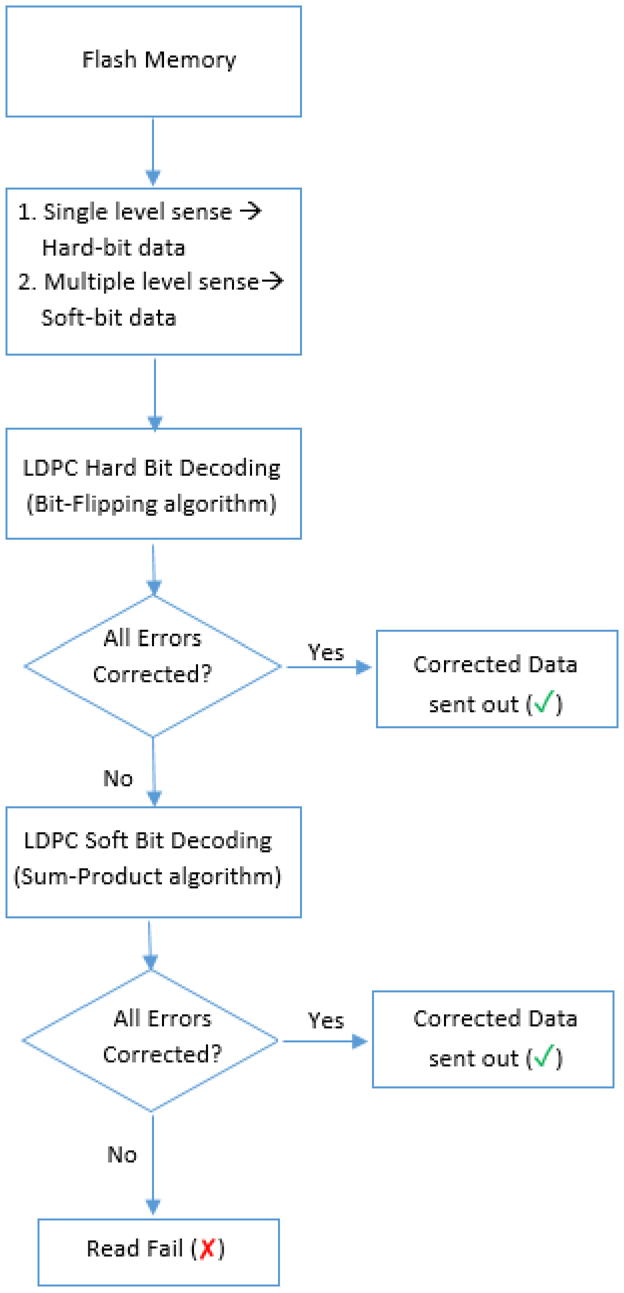

Flash Memory
1. Single level sense → Hard-bit data
2. Multiple level sense → Soft-bit data
LDPC Hard Bit Decoding (Bit-Flipping algorithm)
All Errors Corrected?
Yes
Corrected Data sent out (✓)
No
LDPC Soft Bit Decoding (Sum-Product algorithm)
All Errors Corrected?
Yes
Corrected Data sent out (✓)
No
Read Fail (✗)

Fig 24: LDPC Decoding Scheme Algorithm

Chapter 4: Error Correction in NAND Flash memory

4.1 Introduction

Error correction in NAND Flash memory chips is done using ECC encoders and decoders. Following diagram depicts the steps followed in implementing ECC in NAND Flash chips.

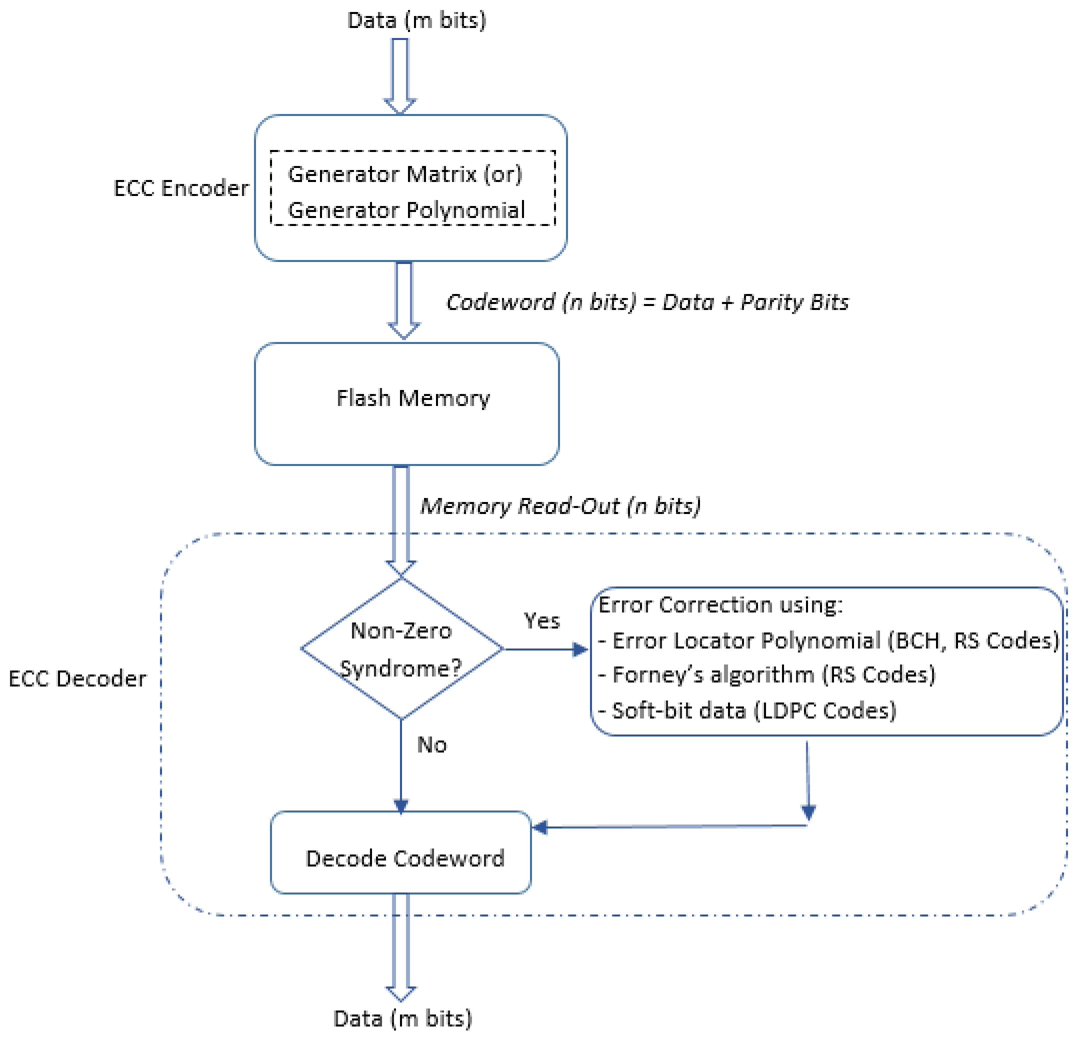


Fig 25: Steps followed in implementing ECC in NAND Flash chips

4.2 Bit Error Rates

Data read-out from memories often has bit errors. Bit error rate (BER) is the number of bit errors occurring per unit time. Raw Bit error rate (RBER) [17] is the bit error rate of the raw data i.e. the data obtained after a memory read-out and before any error correction techniques have been applied to it.

However not all failing bits cause a data loss, since most of them can be corrected with ECC. To quantify this, the bit error rate after ECC has been carried out is measured and this post-ECC bit error rate is called as Uncorrectable Bit Error Rate (UBER). It is called uncorrectable since these errors can no longer be corrected, as they are remnants after error correction has been carried out.

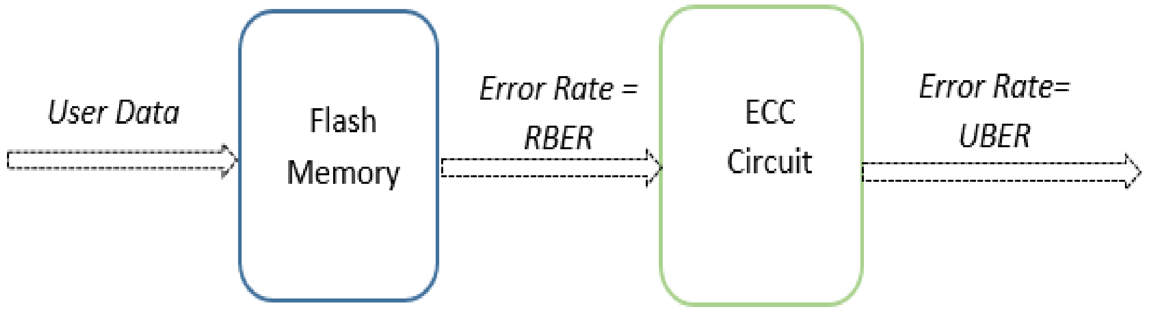


Fig 26: RBER and UBER in NAND Flash

The ratio of RBER to UBER gives the magnitude of improvement offered by ECC. RBER ranges around 10^{-4} to 10^{-2}. Ideally the UBER rate should be as close to 0 as possible. In reality, the UBER ranges between $10^{-16}$and 10^{-15}. RBER depends on the number of Program/Erase (P/E) cycles. It increases as the number of P/E cycles increases.

Increasing RBER requires stronger Error Correction Codes to achieve the same UBER level. Thus for the same number of user data bits, the number of check bits (parity bits) required increases as RBER increases. This in turn leads to a decrease in the Code-Rate (= Data bits / (Data + Parity bits)). A flowchart depicting this sequence is given below:

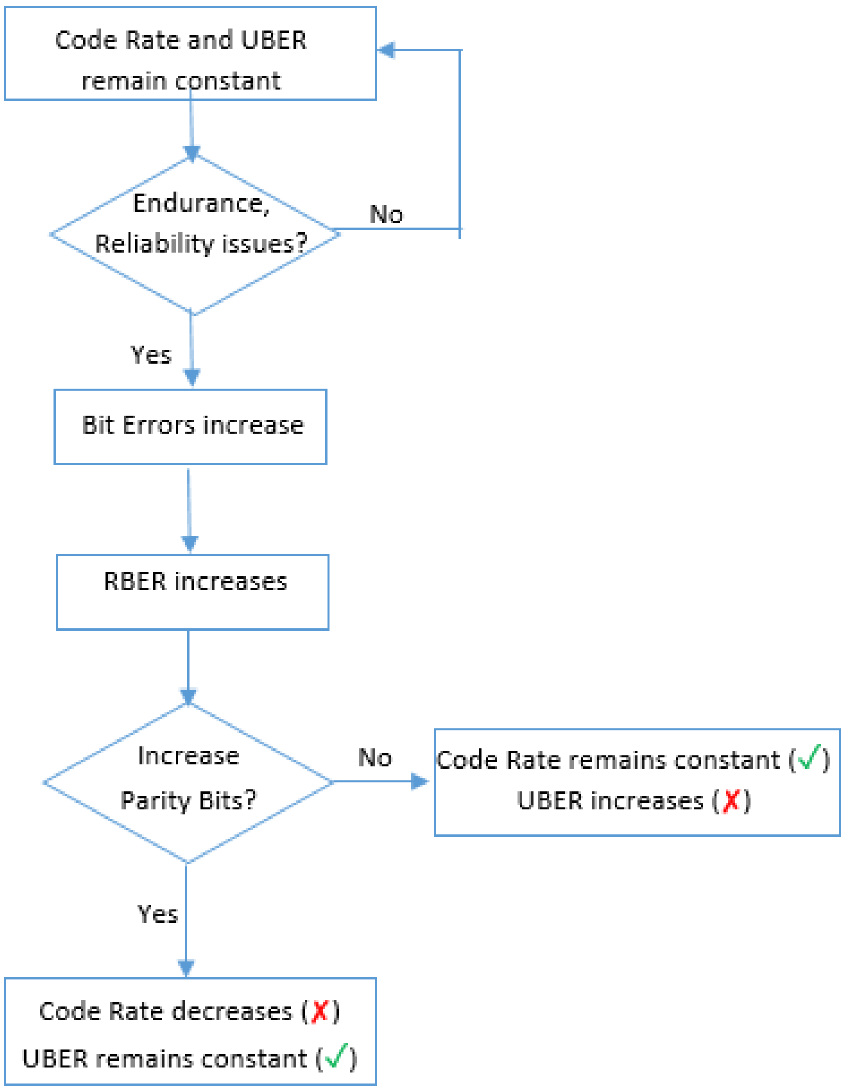

Code Rate and UBER remain constant
Endurance, Reliability issues?
No
Yes
Bit Errors increase
RBER increases
Increase Parity Bits?
No
Code Rate remains constant (✓)
UBER increases (✗)
Yes
Code Rate decreases (✗)
UBER remains constant (✓)

Fig 27: UBER and Code-Rate

4.3 Causes of Bit Errors in NAND Flash memories

Bit errors lead to lower endurance and reliability of NAND chips. There are various causes for bit errors to occur in NAND Flash memories:

4.3.1 Read Disturb

To read flash memory transistors, a reference voltage is applied on the selected flash memory transistor ("cell"). The other cells in the Bit-String are made to conduct by applying voltage on their gate terminal. A sense amplifier at the end of the Bit-String detects whether the selected cell is conducting or not, based on the current conduction through the string. The cell conduction indicates if the reference voltage is higher or lower than the threshold voltage of the cell. Since a non-zero gate voltage is applied to the other cells in the Bit-String, the Threshold Voltage of these cells might shift leading to a change in the state of these cells. This unintentional shift in V_T of neighboring cells during a memory read operation is called Read Disturb.

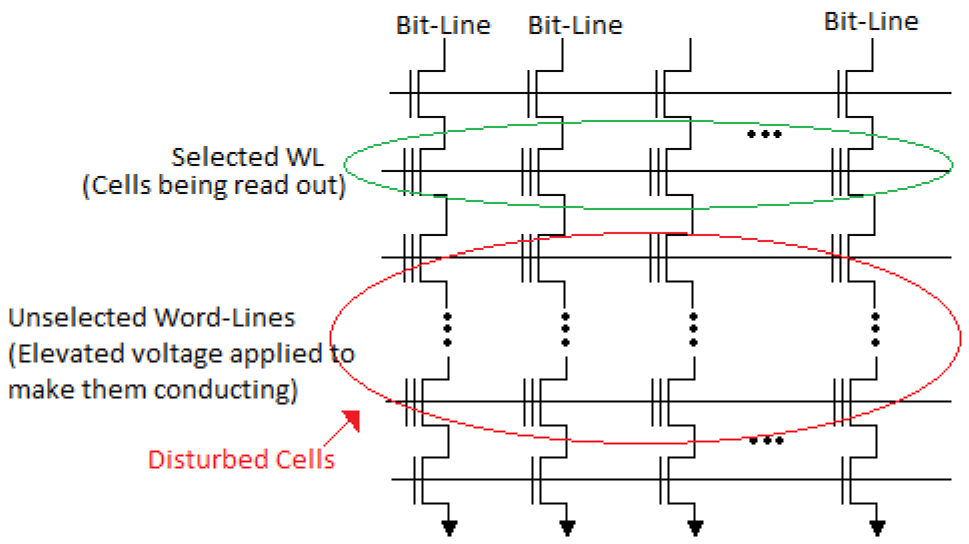


Fig 28: Read Disturb

4.3.2 Program Disturb

While programming, a high voltage is applied on the selected Word-Line. Due to parasitic coupling, charge might collect on cells of neighboring Word-Lines, thus causing unwanted storage of charge on unselected cells. This in turn might cause a V_T shift in unselected cells.

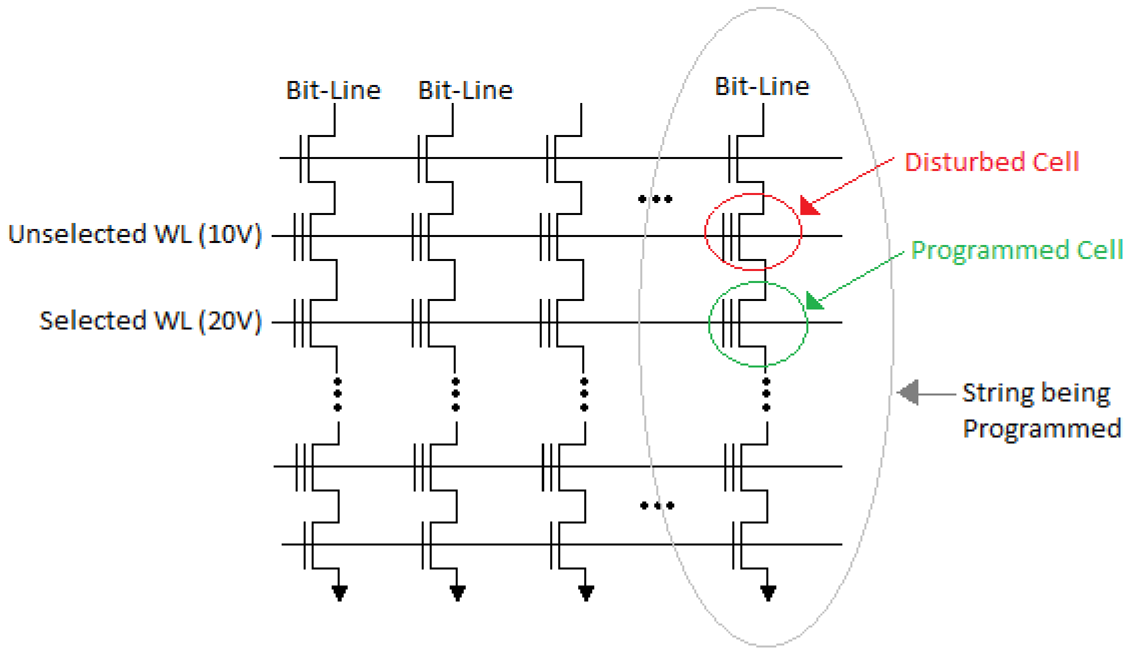


Fig 29: Program Disturb

4.3.3 Failed Erase Operation

Due to process variations or interstitial charges trapped within the oxide, a cell might not be completely reset after an erase operation. This will lead to residual charge in the transistor, and the V_T of the cell might not be in the erased-state distribution, which will lead to bit-errors.

4.3.4 Program/Erase (P/E) Cycles

Repeated Program/Erase cycles leads to oxide stress thereby causing charge to be trapped in the oxide. This causes a permanent change in the FG transistor characteristics, which cannot be recovered by erasing the FG transistor. This affects the flash endurance, and blocks that are affected by this are marked as ‘BAD’ and retired. In place of Bad Blocks, Spare Blocks will then be used.

4.3.5 Retention issues

Over time, due to retention issues [18], the charge in a Floating Gate transistor might leak, eventually causing a shift in its V_T distribution. Because of above reasons, the threshold voltage of a Floating Gate transistor will shift unintentionally, leading to an overlap of adjacent V_T distributions. This in turn will cause incorrect sensing to happen during a memory read, which will result in bit errors.

Following diagram shows the impact of Read Disturb, Program Disturb, High P/E cycles, Erase Failure and Retention Issues on the Threshold Voltage distribution. All these cause a shift in V_T leading to bit-error failures.

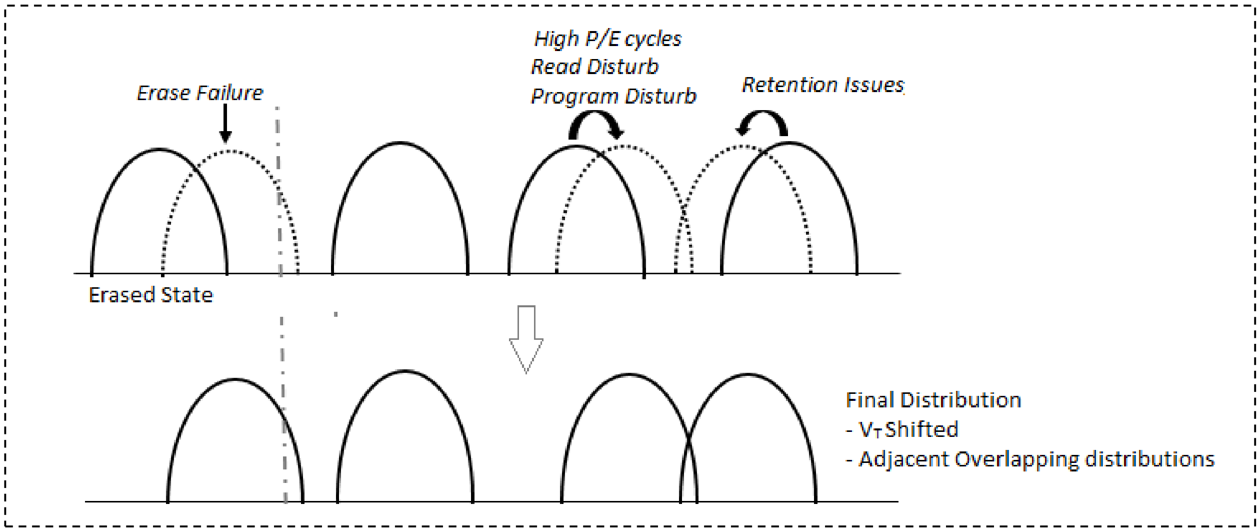


Fig 30: Threshold Voltage Distribution shift due to Disturb and Failure issues

4.4 Factors impacting ECC requirements

There are three major factors impacting the bit-error rates and in turn the error correction requirements in NAND Flash memories.

4.4.1 Scaling

With shrinking technology nodes, as the cell size decreases, fewer electrons are trapped inside the Floating Gate. This in turn increases the uncertainty of programming to a specific threshold voltage level, thereby increases the number of bit errors.

4.4.2 Program/Erase [P/E] Cycles

NAND Flash memories are limited by the number of P/E cycles that they can endure, before the memory device fails. The more the number of P/E cycles, the more the cell degradation leading to an increase in bit-errors. This issue becomes worse in MLC devices. MLC cells can endure a much less number of P/E cycles than SLC cells. Typically, P/E cycles are 100,000 times for SLC and around 10,000 times for MLC memories.

4.4.3 Multi-Level-Cell Technology

As the number of bits-per-cell increases, the number of V_T voltage windows increases. This decreases the gap between adjacent V_T distributions. Thus there is a higher probability of a bit accidentally ending up in the wrong V_T window, which will lead to a bit error. Thus for the higher bits-per-cell schemes, there is a higher probability of getting bit errors. Because of this MLC memories have more number of bit errors and so stronger ECC schemes are required for MLC. For TLC (Triple-Level-Cell) technology, the number of bit-errors is even higher because of the 8-state voltage distribution. This can be counteracted by improving the programming algorithms to end up with tighter V_T distributions or strong ECC schemes can be used to correct the resultant bit errors.

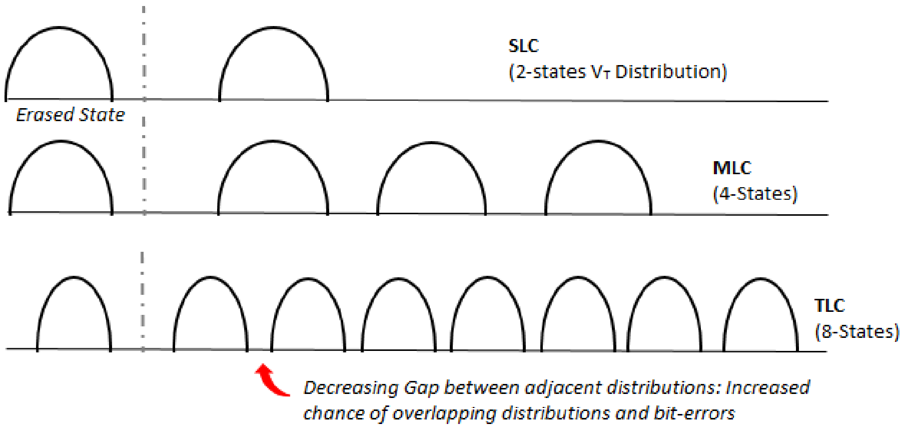

SLC
(2-states V_T Distribution)
Erased State
MLC
(4-States)
TLC
(8-States)
Decreasing Gap between adjacent distributions: Increased chance of overlapping distributions and bit-errors

Fig 31: Threshold Voltage windows in SLC, MLC, TLC cells

Below graph shows the number of ECC bits required for SLC and MLC devices. Initially SLC devices needed single-bit error correction for every page of data (around 512B then), while MLC devices needed 4-bits error correction. As technology scaling increased, the ECC requirements have increased to around 12 bits correction for SLC devices and around 40 bits correction for MLC. TLC devices need even higher number of ECC bits and so LDPC codes are being used to overcome this challenge.

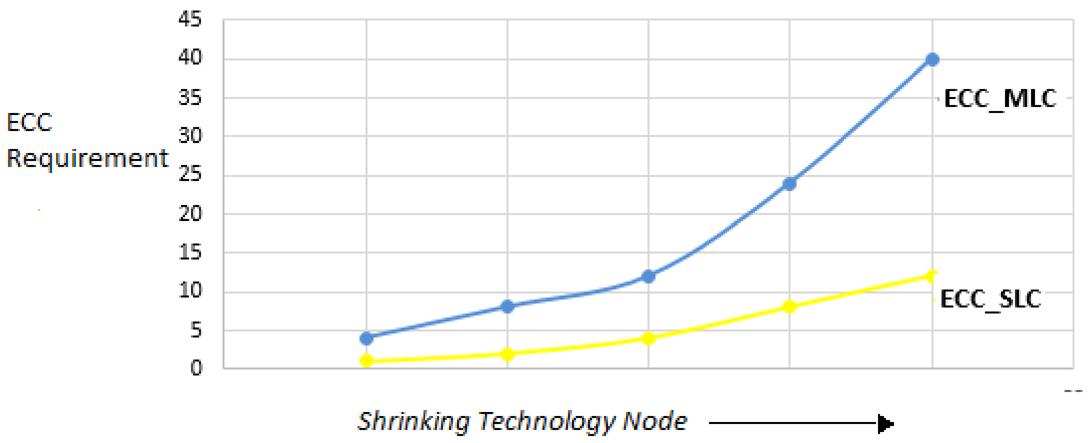


Fig 32: ECC requirement for SLC and MLC devices

The impact of all the above factors can be explain using the following graph:

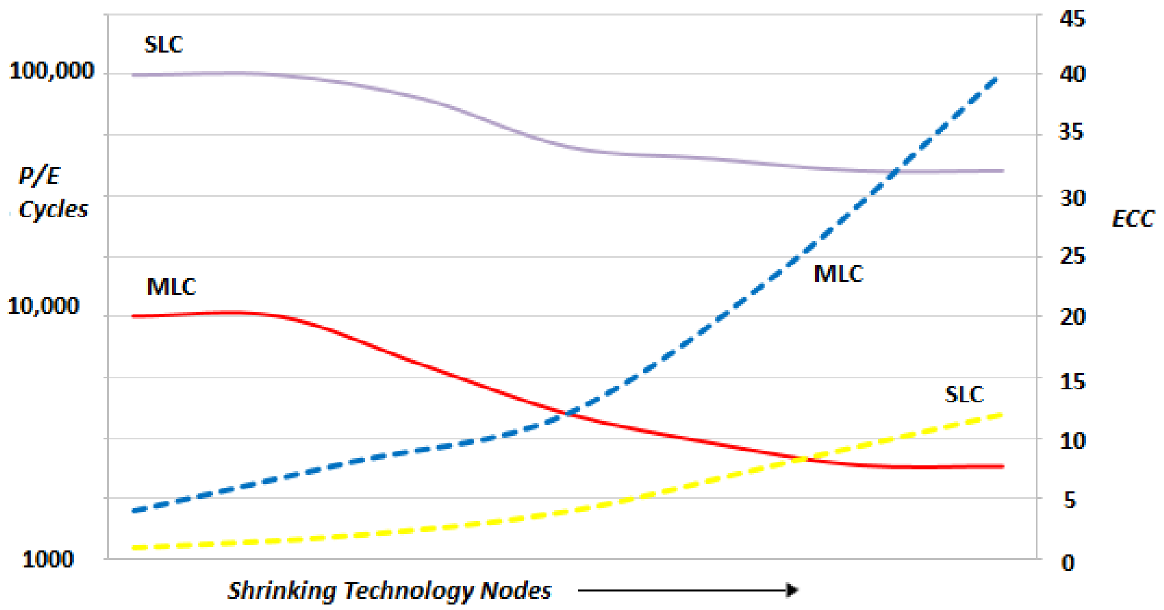


Fig 33: Impact of P/E cycles, Scaling and MLC on ECC requirements

For SLC devices, the number of P/E cycles the device can withstand before it fails is in the range of 100,000 cycles. However with scaling, this number drastically falls. Similarly for MLC, the P/E cycle count is 10,000, which also drops dramatically with scaling.

On the other hand, ECC requirements increase with scaling. SLC devices initially required single-bit error correction but that number has gone up to 12 bits correction per page for the latest technology nodes. Similar the MLC ECC requirements started off at around 4, and are currently upwards of 40 bits.

4.5 ECC Schemes used for error correction in NAND Flash memories

4.5.1 Hamming Codes

Error correction in NAND Flash memories is done by adding ECC bits to every page of user data, such that the added ECC bits will be sufficient to correct 1 or multiple bits of errors in a page. During early days of Flash technology, when only SLC devices were used, the ECC requirement was to correct 1 bit in a single page of data. Since the page size was typically 512Bytes, Hamming codes were used to provide single-bit correction for 512Bytes of User Data. For Hamming codes, for a block length of 2^N-bits, the number of parity bits required is n bits. Thus for 512Bytes of data, the number of parity bits required is 12 bits. Similarly data is given below for varying page sizes.

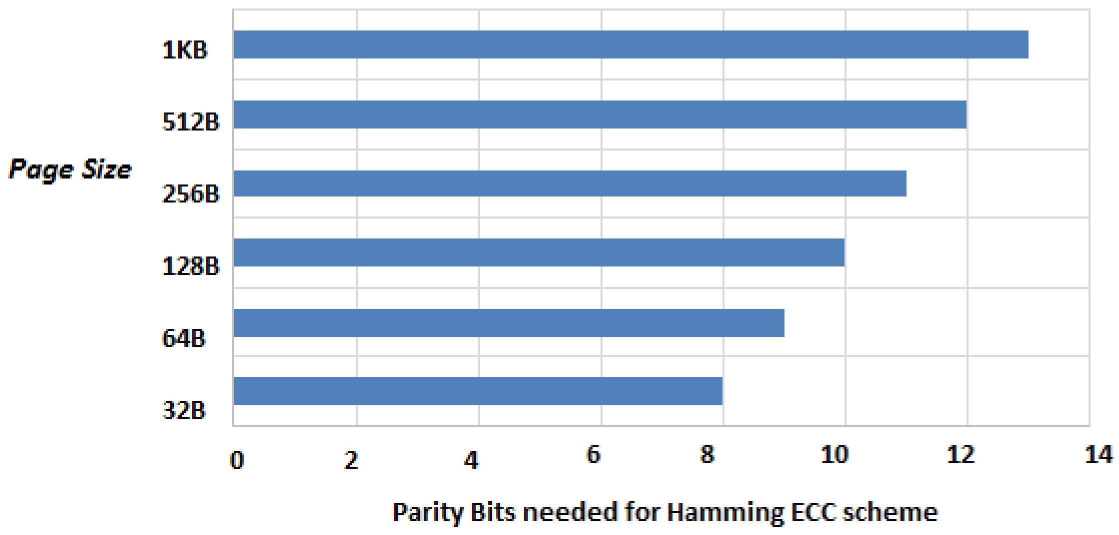


Fig 34: Page size vs Parity bits for Hamming ECC

On every page of a NAND array, in addition to User data and ECC bits, some spare bits are stored for wear-leveling and other purposes. Typically, 64 spare bytes are allotted for 2048 data bytes [19]. For a 512Byte User Data block, 16 spare bytes are allocated which results in a total size of 528 bytes. The ECC bytes are included within the spare bytes.

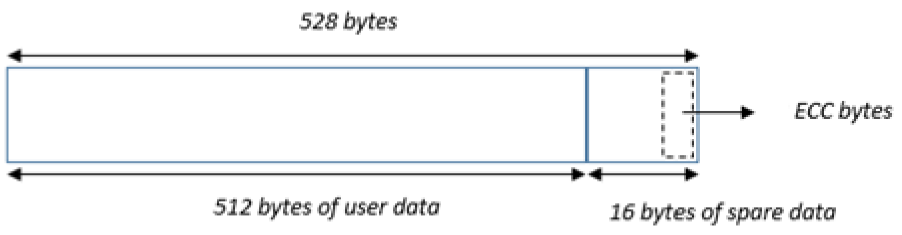


Fig 35: Arrangement of ECC bytes for a 528B block

In today's SLC devices, the page size is longer than 512 bytes. In this case the data is broken up into blocks of 512bytes, and an SEC Hamming code is used for every single 512byte block. Below figure demonstrates such a scheme for a 2KB Page size.

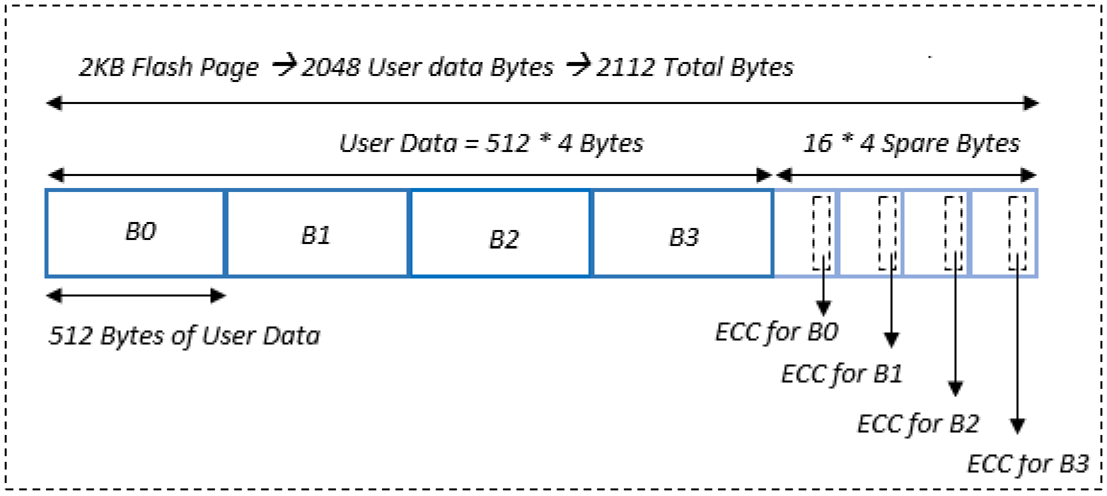


Fig 36: Arrangement of ECC bytes for 2KB Page

4.5.2 BCH and Reed-Solomon Codes

With scaling technology and multiple bits-per-cell technology (MLC), the likelihood of multi bit errors in a single page increases. To correct multiple errors in a single page:

- As shown above, the page can be split into smaller blocks of data, and then apply Hamming code for each block (or)
- Use multiple-bit error correction schemes like Reed-Solomon and BCH codes.

As discussed before, Reed-Solomon (R/S) codes split a block of data into symbols and then do error correction. For a 512B page size, a Reed-Solomon code will break up the data into 512 symbols, with each symbol being 8-bits wide. If an error is detected in a symbol, then the entire symbol is corrected. Another example of a Reed-Solomon code is RS (255, 223) with a symbol length of 8-bits. For a user-data block of 223 bytes (k), this code adds 32 parity bytes to result in a codeword of size 255 bytes (n). Thus for this code the number of errors that can be corrected is t = [(n-k)/2] = 16.

R/S codes are efficient for correcting burst errors since a single symbol correction will correct multiple clustered bit errors. However the errors encountered in NAND Flash are more often randomly distributed, individual bit errors rather than clumped, burst errors. For such a type of error distribution, BCH codes are more efficient and for this reason BCH codes are more suitable for error correction in NAND Flash memories.

- For a code length 'n' (such that n= 2^m -1) and message length of 'k' bits, with error correction capability 't' (i.e. 't' errors can be corrected), for BCH codes (n-k) ≤ m*t. Since (n - k) is the number of parity bits, the maximum number of parity bits is m*t.
- Below are the results of simulations, which have been run with BCH codes of different block lengths. As the block length increases, m increases and thus the number of parity bits required also increases.

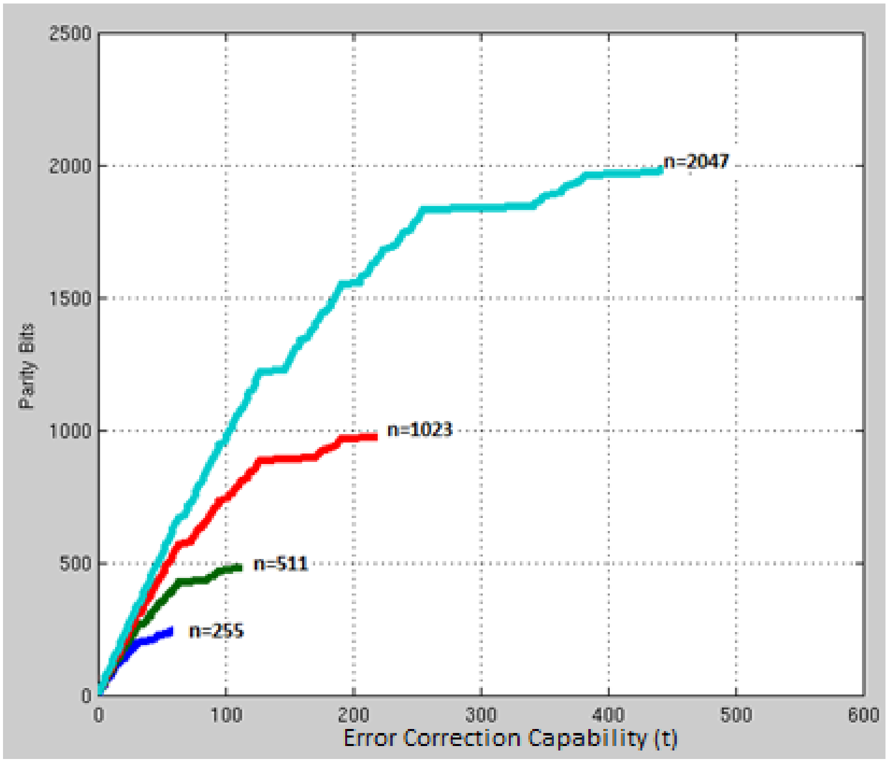


Fig 37: Parity bits vs Error Correction Capability for BCH code

- For small number of parity bits, (n - k) = m*t. This can be seen in below simulation results, where there is a linear relation between parity bits and ‘t’.

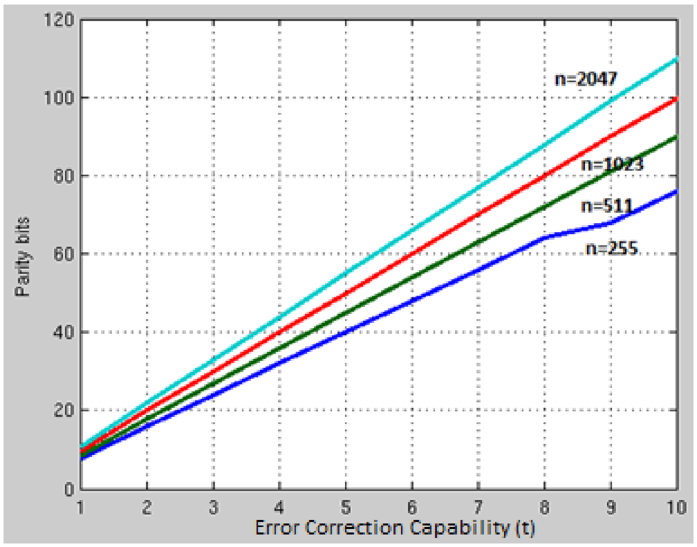


Fig 38: Parity bits vs Error Correction Capability for small ‘t’

- For different BCH block lengths, the code-rate performance has also been analyzed as shown in the graph below. For the same error correction capability, increasing the block length leads to an increase in code-rate. Thus, longer block lengths have a better performance than shorter block lengths.

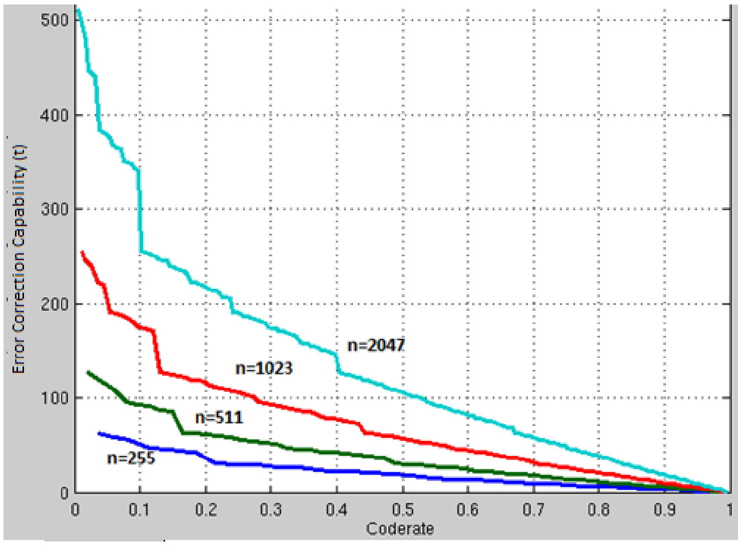


Fig 39: Error Correction Capability vs Code-rate for BCH code

- This increase in error correction strength with increasing code lengths can be explained as follows: Let an n-block code correct 5 errors, and a 2n-block code correct 10 errors. For the case of 2n-data bits having to correct 10 errors, the 2n-block code will pass. In the case of using two n-block codes, if the errors are distributed unevenly (as shown in the diagram), then the ECC will fail. Thus, it is better to having longer code lengths as they offer higher protection capabilities.

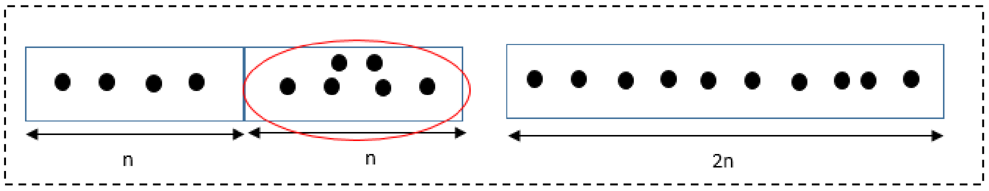


Fig 40: Error Correction for different block lengths

- For a Reed-Solomon code, (n-k) = 2*s*t where 's' is the symbol length. Thus the number of parity bits required (n - k) can be directly calculated.

- For m = 13, s =9 (i.e. for the R/S code, the symbol width is 9 bits), the number of parity bits required for BCH and R/S codes is calculated using the above formulas.

Table 1: Parity bits for R/S and BCH codes

t	Reed Solomon Parity bits	BCH Parity bits
1	18	13
2	36	26
3	54	39
4	72	52
5	90	65
6	108	78
7	126	91
8	144	104
9	162	117
10	180	130

- From below graph it can be seen that the number of parity bits required for BCH is less than that required for Reed-Solomon, for the same error correction capability.

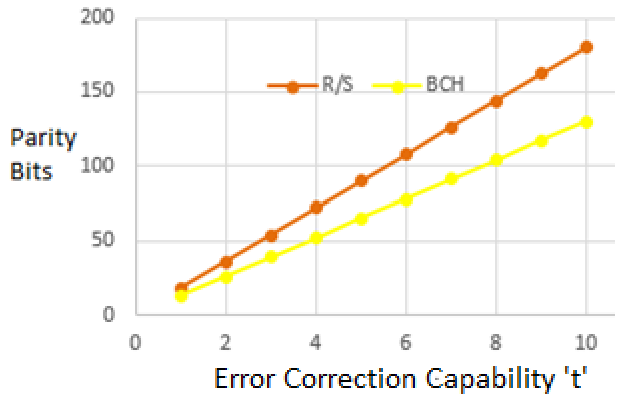


Fig 41: Parity Bits vs Error Correction Capability

- Following is the graph comparing the code-rates for BCH and R/S codes. For the same Error Correction Capability 't', BCH has a higher code-rate than R/S codes, which is an advantage of using BCH codes.

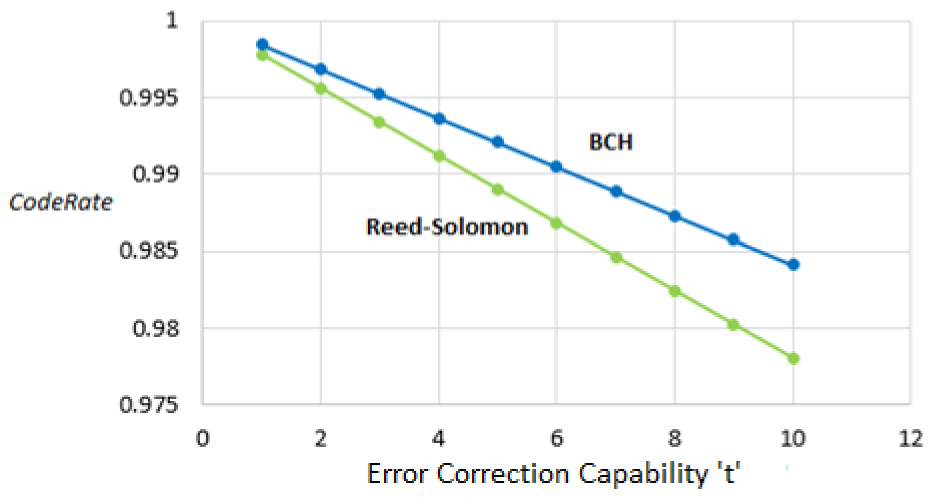


Fig 42: Code-rate vs Error Correction Capability for BCH and R/S codes

- To analyze the performance of BCH codes, simulations have been run using BCH codes on an AWGN channel using BPSK modulation. Parameters of the BCH codes used in the simulation are as follows:

Table 2: Simulation parameters for performance analysis of BCH codes

Code-Rate	Codeword Length	Message Length	Number of errors that can be corrected
0.87	255	223	4
0.86	511	439	8
0.84	1023	863	16

- The simulation results for the Bit Error rate against the Signal-to-Noise ratio are:

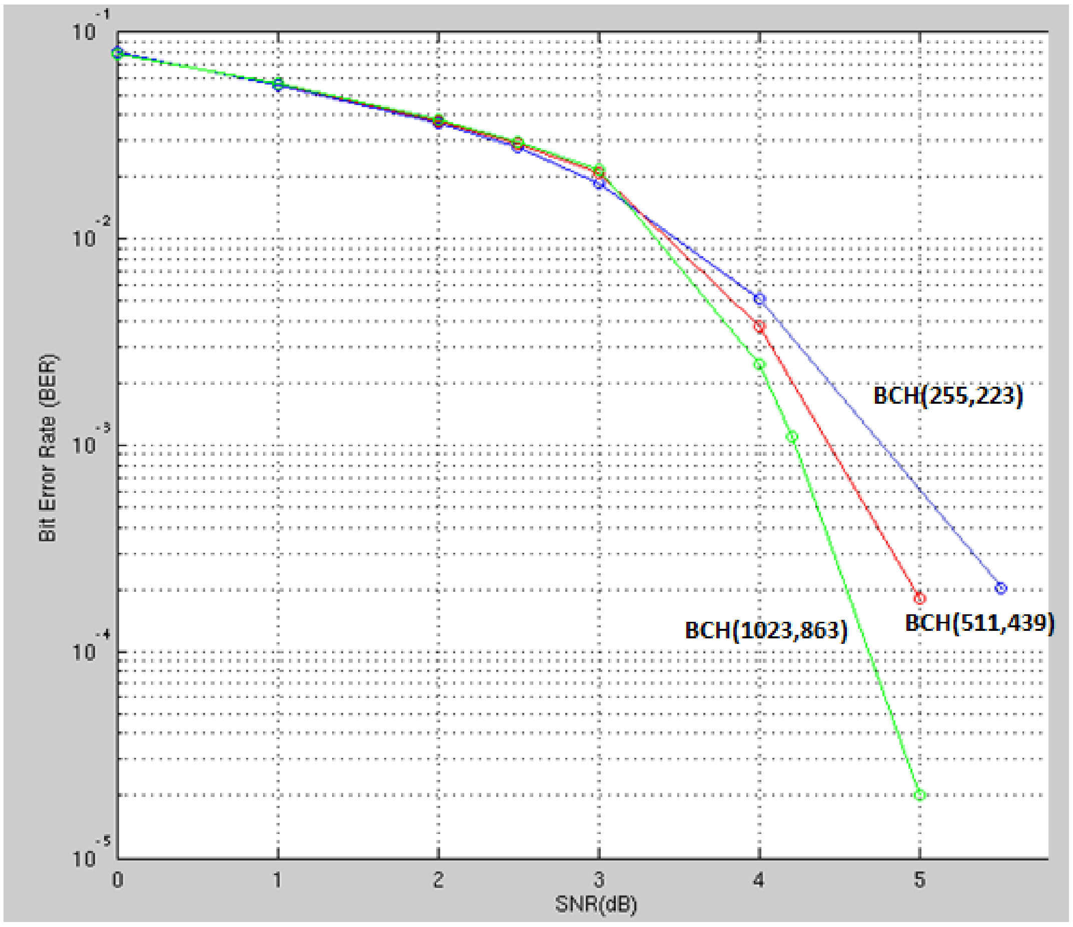


Fig 43: BER vs SNR for BCH Code

- The results show that as the code-rate increases, the performance of the BCH codes deteriorates.
- Furthermore it can be seen that in terms of error correction capabilities a BCH (511, 439) code is equivalent to two BCH (255, 223) codes (Both can correct 8 errors). However BCH (511, 439) offers a better performance than BCH (255, 223) for the same error correction capability.

4.5.3 LDPC codes

LDPC codes are very suitable for error correction in NAND Flash memories because of their ability to decode hard-bit and soft-bit data and offer high performance rates for large block lengths. Simulations have been run on an AWGN channel using QPSK modulation to analyze performance of LDPC codes. Following are the dimensions of the codes used for the analysis.

Table 3: Simulation parameters for performance analysis of LDPC codes

Code-Rate	Dimensions of H-Matrix
1/2	32400 x 64800
2/3	21600 x 64800
4/5	12960 x 64800
9/10	6480 x 64800

- Following is a view of the H-Matrix (32400 x 64800) for a ½ LDPC code.

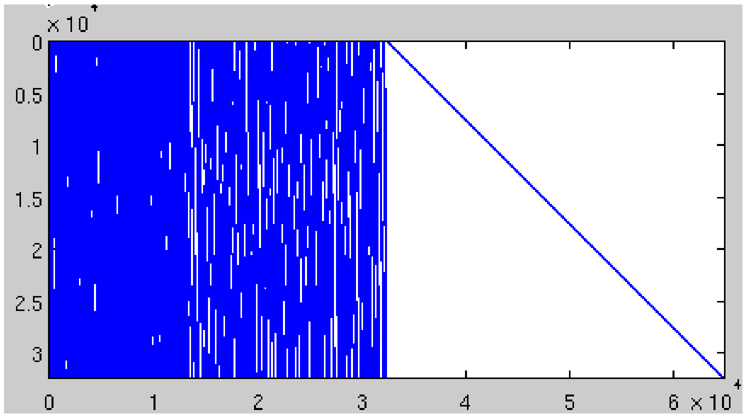


Fig 44: H-Matrix for a ½ LDPC Code

- The simulation results show that as the code-rate increases, the performance of the LDPC code deteriorates.

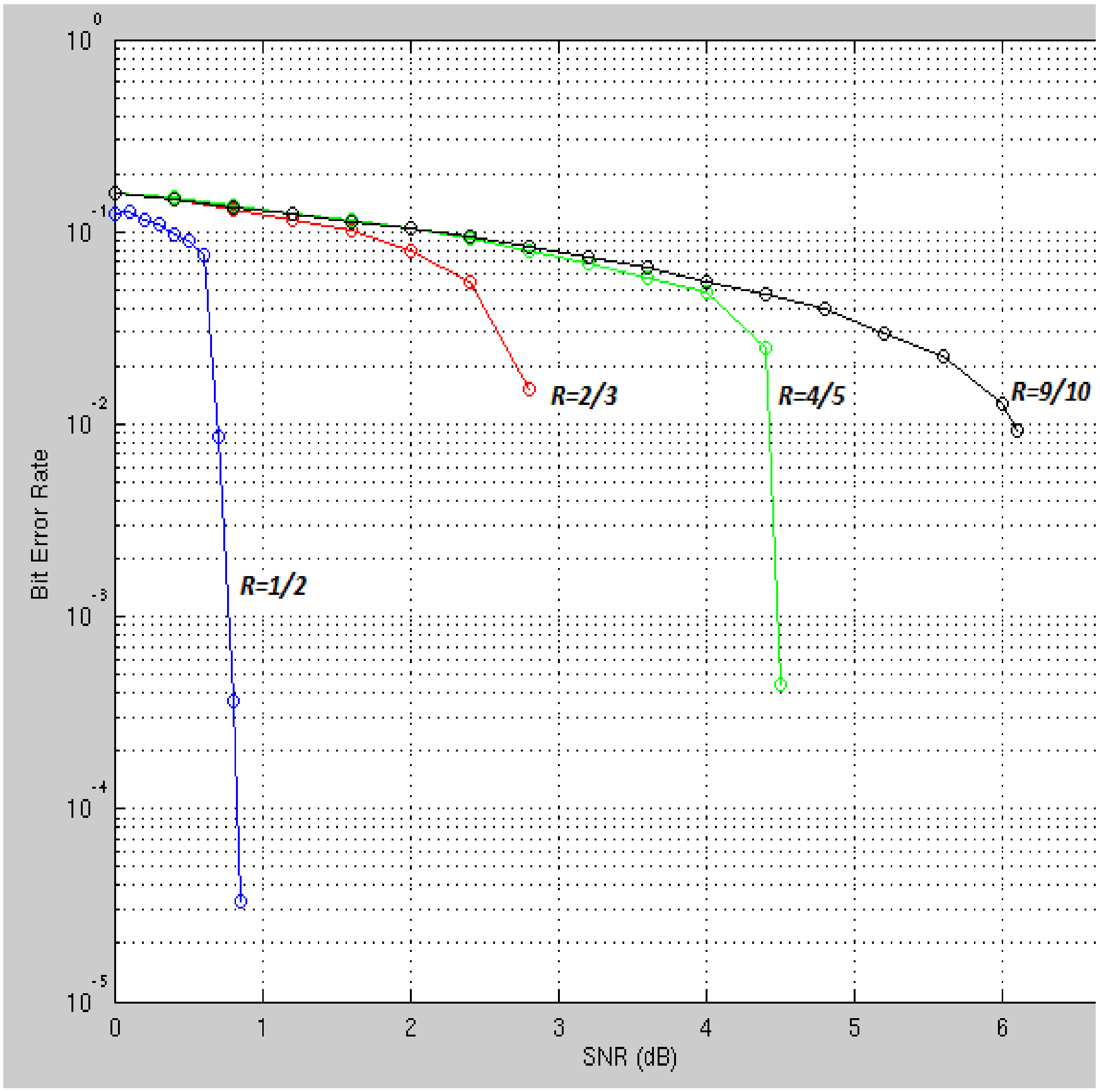


Fig 45: Bit Error Rate vs SNR for LDPC Codes

4.6 Comparison of ECC schemes

Following table gives a comparison of Hamming, BCH, Reed-Solomon and LDPC codes.

Table 4: Comparison of ECC Schemes

	Hamming	Reed-Solomon	BCH	LDPC
Error Correction	Single-bit	Multi-Bit	Multi-Bit	Multi-Bit
Device Type	SLC	MLC	MLC/TLC	MLC/TLC
Error types	Scattered errors	Burst Errors	Scattered errors	Scattered
Soft Bit Decoding	No	No	No	Yes

Circuit Considerations

	Hamming	Reed-Solomon	BCH	LDPC
ECC Area	Small	Large	Large	Large
Power Consumed	Less	High	High	High
Software Implementation	Feasible	Difficult	Difficult	Difficult

Performance

	Hamming	Reed-Solomon	BCH	LDPC
Algorithm Complexity	Simple	Complex	Complex	Complex
Error Correction Capability	Limited	High	High	High
Performance	Medium	Medium	High	Very high

Each of the schemes can be used for different NAND Flash devices depending on the requirement:

- Hamming Codes:
    - Suitable for error correction in SLC Devices only because of its limited error correction capabilities
- Reed-Solomon:
    - Suitable for correction of burst errors. However BCH preferred over R/S for error correction in NAND Flash since bit-errors in NAND Flash are mostly individual, randomly scattered errors.
- BCH:
    - Suitable for multi-bit error correction in MLC/TLC Devices because of its strong error correction capabilities.
- LDPC:
    - Increasingly being used for error correction in MLC and TLC devices
    - Better error correction than BCH for the same code rate
    - Both Hard-Bit and Soft-Bit information from NAND Flash is used while decoding
    - Near-Shannon capacity performance rates for large block lengths

Chapter 5: Conclusion

Since NAND Flash memories are not intrinsically free from bit errors, Error Correction Codes (ECC) are used to correct these bit errors. Thus ECC is used to improve the raw storage reliability**.** With increasing bit-error rates because of scaling technology and usage of Multi-Level-Cell devices, it has become imperative to use robust Error Correction Codes in NAND Flash memories to provide data protection and extend lifetime of the device.

In NAND Flash memory, data is stored by altering the threshold voltage of Floating Gate transistors. Bit errors in NAND Flash mainly result from unintended V_T shifts of neighboring transistors because of high voltages applied on the selected transistor. In addition, Read Disturb, Failed Erase operations, retention and leakage issues also cause bits to flip.

Bit-error rates are impacted by scaling nodes, repeated Program/Erase (P/E) cycles and multi-level-cell technology. In MLC/TLC cells, due to the multiple threshold voltage levels, there is a higher uncertainty while programming data into the cell and while reading data out of the cell. Due to this the number of bit errors and the ECC requirement is high in multi-level-cell devices.

Error correction codes are employed to detect and correct these bit errors. An ECC encoder adds check/parity bits to the user data bytes, and this information is stored in the memory. After memory-read out, an ECC decoder checks the data using the check bits. In case there was an error, the check bits can be used to reconstruct the original data.

Hamming codes are used in SLC devices to correct single bit errors in a page of user data. For large page sizes, the page is split into chunks and a Hamming code is used for each chunk of data. For multi-bit errors Bose, Chaudhuri, and Hocquenghem (BCH) codes are preferred over Reed-Solomon codes because of their capability to correct randomly scattered errors.

Of-late, LDPC codes are also being increasingly used for error correction in MLC and TLC devices. LDPC codes differentiate themselves from other ECC schemes in that they can be used to decode hard-bit (binary) information as well as soft-bit information. With the additional soft-bit data from the memory, LDPC codes provide much stronger correction. Thus LDPC codes are especially beneficial for NAND Flash since soft-bit information is available and provide near-Shannon capacity performance for large block lengths.

Bibliography

[1] Dan Rosso, "Global Semiconductor Industry posts Record Sales in 2014," *Semiconductor Industry Association.* Available: http://www.semiconductors.org/news/2015/02/02/global_sales_report_2014/global_semiconductor_industry_posts_record_sales_in_2014/

[2] Jim Handy (Objective Analysis), "Flash Technology: Annual Update," *Flash Memory Summit 2015*. Available: http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2015/20150811_SU1_Handy.pdf

[3] R. H. Fowler and L. Nordheim, "Electron Emission in Intense Electric Fields," *Proceedings of the Royal Society*, Part A, vol. 119(781), pp. 173-181, 1928.

[4] B. Ricco, et al.: "Nonvolatile multilevel memories for digital applications," *Proceedings of the IEEE*, vol. 86, pp. 2399–2423, 1998.

[5] K. Takeuchi, T. Tanaka, and T. Tanzawa, "A multipage cell architecture for high-speed programming multilevel NAND Flash memories," *IEEE Journal of Solid-State Circuits*, vol. 33, pp. 1228–1238, 1998.

[6] K. Suh, B. Suh, Y. Lim, J. Kim, Y. Choi, Y. Koh, S. Lee, S. Kwon, B. Choi, J. Yum, J. Choi, J. Kim, and H. Lim, "A 3.3 V 32 Mb NAND Flash Memory with Incremental Step Pulse Programming Scheme," *IEEE Journal of Solid-State Circuits*, vol. 30, pp. 1149-1156, 1995.

[7] R. Micheloni, R. Ravasio, A. Marelli, et al, "A 4Gb 2b/cell NAND Flash Memory with Embedded 5b BCH ECC for 36MB/s System Read Throughput," *IEEE International Solid-State Circuits Conference*, pp. 497-506, Feb. 2006

[8] S. gun Cho, D. Kim, J. Choi, and J. Ha, "Block-wise concatenated bch codes for NAND flash memories," *IEEE Transactions on Communications*, vol. 62, pp. 1164–1177, 2014.

[9] E. R. Berlekamp, *Algebraic Coding Theory*, New York: McGraw-Hill, 1968. (revised ed.—Laguna Hills, CA: Aegean Park, 1984).

[10] H. O. Burton, "Inversionless decoding of binary BCH codes," *IEEE Transactions on Information Theory*, vol. IT-17, pp. 464–466, 1971.

[11] B. Chen, X. Zhang, and Z. Wang, "Error correction for multi-level NAND flash memory using Reed-Solomon codes," *Proceedings of the 2008 IEEE Workshop on Signal Processing Systems (SiPS2008)*, Washington, DC, U.S.A., pp 94-99, 2008.

[12] R. Gallager, "Low density parity-check codes," *IRE Transactions on Information Theory*, vol. IT-8, pp. 21-28, 1962.

[13] R. M. Tanner, "A recursive approach to low complexity codes," *IEEE Transactions on Information Theory*, vol. 27, pp. 533-547, 1981.

[14] G. Dong, N. Xie, and T. Zhang, "On the use of soft-decision error correction codes in NAND flash memory," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, pp. 429–439, 2011.

[15] J. Zhao, F. Zarkeshvari, and A.H. Banihashemi, "On implementation of min-sum algorithm and its modifications for decoding low-density Parity-check (LDPC) codes," *IEEE Transactions on Communications*, vol. 53, pp. 549-554, 2005.

[16] R. Motwani, Z. Kwok, and S. Nelson, "Low density parity check (LDPC) codes and the need for stronger ECC," in *Flash Memory Summit, 2011*. Available:

http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2011/20110810_T2A_Nelson.pdf

[17] N. Mielke, et al, "Bit Error Rate in NAND Flash Memories", *IEEE International Reliability Physics Symposium*, pp. 9-19, 2008.

[18] Y. Cai, et al., "Error Patterns in MLC NAND Flash Memory: Measurement, Characterization and Analysis," *Design, Automation & Test in Europe Conference & Exhibition*, pp. 521-526, 2012.

[19] Z. Wang, M. Karpovsky, and A. Joshi, “Reliable MLC NAND flash memories based on nonlinear t-error-correcting codes,” *IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 41-50, 2010.