ERROR CORRECTION METHODS FOR LATENCY-CONSTRAINED FLASH MEMORY SYSTEMS

by

PRIYANKA ANKOLEKAR

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2008

ACKNOWLEDGEMENTS

Foremost, I would like to thank my mentors and supervisors, Stephan Rosner at Spansion Inc. and Dr. Jonathan Bredow at UT-Arlington, without whom I would not have done a thesis. Dr. Bredow not only supervised my thesis, he agreed to do so from Texas while I was in California for the entire duration of my work. I would like to thank him for being patient, having faith in me and providing me with vital guidance all the time. Stephan Rosner shared a lot of his expertise and insight with me refusing to accept anything less than the best I could do. His enthusiasm and inspiration was always there when I needed it.

I wish to thank Roger Isaac and Qamrul Hasan of Spansion for patiently answering all my questions and for the endless discussions which helped me out of many a tight-spot I encountered while working on this thesis. There are many others at Spansion whose assistance I deeply value – George Minassian, Venkat Natarajan, Mark Randolph and Darlene Hamilton. I would like to specially thank Filomena Mendonsa for brightening up the atmosphere at work.

I would like to express my appreciation to Dr. A. Davis and Dr. S. Gibbs for helping me to take this thesis to consummation by being a part of my thesis committee.

I am tempted to individually thank all of my friends, but the list will be too long and from fear of leaving someone out, I will simply say *thank you all very much*. However, Hari deserves a special thank you for helping me in every way and making my stay in California so easy.

I cannot finish without saying how grateful I am to Vinayak for making me laugh when I was stressed out and for the incredible amount of patience he has had with me. Lastly and most importantly, I would like to thank my parents whose faith in me sometimes borders on craziness and without which I would not be studying for a Master's degree. To them I dedicate this thesis.

July 11, 2008.

ABSTRACT

ERROR CORRECTION METHODS FOR LATENCY-CONSTRAINED
FLASH MEMORY SYSTEMS

Priyanka Ankolekar, M.S.

The University of Texas at Arlington, 2008

Supervising Professor: Jonathan Bredow

Maintaining the reliability of data stored in Flash devices and reading it correctly has become a challenge as the demand for higher density is forcing aggressive shrinking of Flash architectures. For all Flash systems, especially latency-constrained NOR Flash, an on-chip error correction code (ECC) is the only viable and robust solution to this problem.

This thesis investigates and optimizes low-latency error correction schemes for on-chip implementation in NOR systems using existing error correction methods as a starting point. As the first step towards doing this, a mathematical relation has been derived to compute the bit error rate (BER) of a memory array using technology-specific voltage distribution curves. The required error correction capacity is calculated using the BER of the memory array. Current on-chip error correction (ECC) schemes in NOR Flash consist of a single error correcting Hamming code. However, for emerging Flash devices single bit error correction does not suffice to maintain data reliability. This problem has been addressed by analyzing and optimizing existing ECC schemes for low latency and minimal hardware and parity overhead while achieving at

least 2-bit error correction. One of the proposed algorithms is a dual bit Hamming code which uses the Hamming code for 2-bit error detection and correction. Another optimized scheme, called Hierarchical BCH, makes effective use of the fast and simple Hamming code to correct frequently occurring single bit errors and the multi error correction BCH code to correct higher order errors in the rare case when the Hamming code detects a 2-bit error. This scheme gives an average latency of around 4ns while improving the array BER from 10^{-7} to 10^{-15}. Thus all these methods have been quantitatively proven to be applicable in latency-constrained eXecute-in-Place (XiP) NOR Flash systems.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

1.1. Data Reliability in Flash Memory

Flash technology is the fastest growing semiconductor business because the embedded devices market and especially mobile devices require a substantial amount of fast, non-volatile, solid-state storage having high densities. Flash memory is indispensable in battery-powered applications like cell phones, cars, printers, networking equipment, set-top boxes, high-definition TVs, games and other consumer electronics. The increasing complexity of and demand for these products along with an enormous price pressure forces aggressive shrinking of device geometries as well as increasing storage capacities per area by storing multiple bits in each memory cell through multi-level cell (MLC) architectures for Flash memories.

These advances in increasing stored information per unit area by storing multiple bits as different charge levels in a memory cell result in a significant technical challenge in storing and detecting bits. Increased density leads to an increase in the Bit Error Rate (BER) of memory devices. This BER is affected by common disturb mechanisms such as silicon defects, cross-coupling, charge loss (or gain) over time. Bit disturb mechanisms increasingly affect data reliability and need to be compensated for with new methods as currently used solutions are not adequate anymore. For system stability it is mandatory to maintain some maximum BER.

There are two main approaches to achieve a suitable BER in a Flash memory array. One approach is making designs adaptable to technology errors. However, an extensive debug phase extends the overall design cycle of the product affecting the cost adversely and resulting in a loss in market opportunity. This method requires research on a per product basis and therefore is impractical for cost reasons.

The second approach is to correct bit errors in real-time using error correction codes (ECCs). Error correction methods reconstruct lost information by adding redundancy to the stored information. They can be implemented in a controller outside the memory device or on-chip with the memory array itself. The controller external to the memory chip can allow an area-efficient implementation of the algorithm. But controllers designed for NOR Flash memory devices do not have the infrastructure for supporting an ECC implementation making this a non-viable option.

This makes on-chip ECC the only solution. Only simple ECC algorithms can be implemented in the memory device because typical memory technologies do not easily support the efficient integration of large scale digital circuits. Ideally less complex ECC algorithms provide low latency access to the memory array and thus do not interfere with the software model of the memory system. The advantage of applying ECC on the memory device is that, differently from controller-based ECC, ECC-requirements of the memory, like low latency and low hardware and parity overhead, can be matched with the complexity of the applied ECC algorithm. The disadvantage is the limitation to low-complexity (and typically low-latency) ECC algorithms.

Advancements in Flash technology demand an improvement in and optimization of the methods used to protect and correct stored bits. This thesis addresses this problem in low latency NOR Flash memories in the following manner:

The causes for failures in Flash memory have been studied and related technology data has been used to extract error probabilities for a given memory array. These probabilities help determine the bit error rate for the memory array. The computed bit error rate (BER) has been used to quantify the exact requirements for an on-chip ECC implementation in NOR Flash. These requirements are low latency, required error correction capacity, a low gate count (in case of a hardware implementation) and a parity overhead that matches the ‘spare’ area available in the array (Spare area in a given memory array is a fixed area consisting of a certain

fixed number of bits that are not available for use by the consumer. This area can be used to store ECC redundancy computed internally in the memory device. As long as the ECC redundancy is less than or equal to the number of bits in the spare area, it does not add any additional bit overhead). These requirements are used as the basis for comparison of relevant published error detection/ correction (EDC/ECC) algorithms. This study helps to identify shortcomings in existing algorithms with regards an on-chip implementation in low-latency Flash systems. Existing algorithms are optimized in the light of these shortcomings to adhere to NOR Flash requirements and constraints. It is found that there is a trade-off between the error correction capacity of the algorithm, hardware complexity, latency and data overhead.

Giving primary consideration to low latency and high error correction capacity followed by a low hardware complexity and data overhead, a system-level implementation is proposed for each recommended solution. Every implementation is evaluated in terms of latency, bandwidth impact, die size and scalability across generations. This helps to arrive at optimum solutions for ECC algorithms that can be applied to improve data reliability in low-latency NOR Flash memories as device geometries become smaller.

CHAPTER 2

FLASH MEMORY AND FAILURE MECHANISMS

2.1. Flash Memory Architectures

Flash memory is nonvolatile (NVM) memory that can be electrically erased and programmed.

2.1.1 Conventional Flash Technology

Information is stored in a Flash memory in an array of memory cells consisting of floating gate transistors. A floating gate transistor is a MOSFET having two gates – a control gate (CG) and a floating gate (FG). The floating gate is completely surrounded by the dielectric layer. Hence charge trapped on it remains unchanged for extended periods of time. This charge alters the threshold voltage (V_t) [5] of the transistor. To read data, a voltage is applied to the control gate. The presence (logical '0') or absence (logical '1') of current through the channel helps detect the stored bit. Figure 2.1 shows a floating gate transistor.

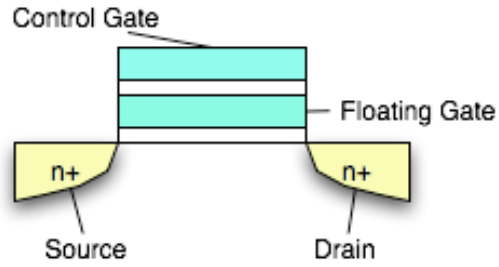

Figure 2.1 The Conventional Memory Cell – A Floating Gate Transistor [12]

2.1.2 Flash Architectures

A single memory cell can store one or more bits of data. A cell which stores a single data bit is called a *single level cell (SLC)* while one which stores more than one bit is a *multilevel cell (MLC)*. MLCs store multiple bits per cell by storing varying amounts of charge for each bit pattern. Therefore, in MLCs, the amount of current flow is sensed, rather than the mere

presence or absence of it as in SLCs, in order to determine the level of stored charged. Figure 2.2 shows the threshold voltage distributions in (a) single level cells and (b) multilevel cells.

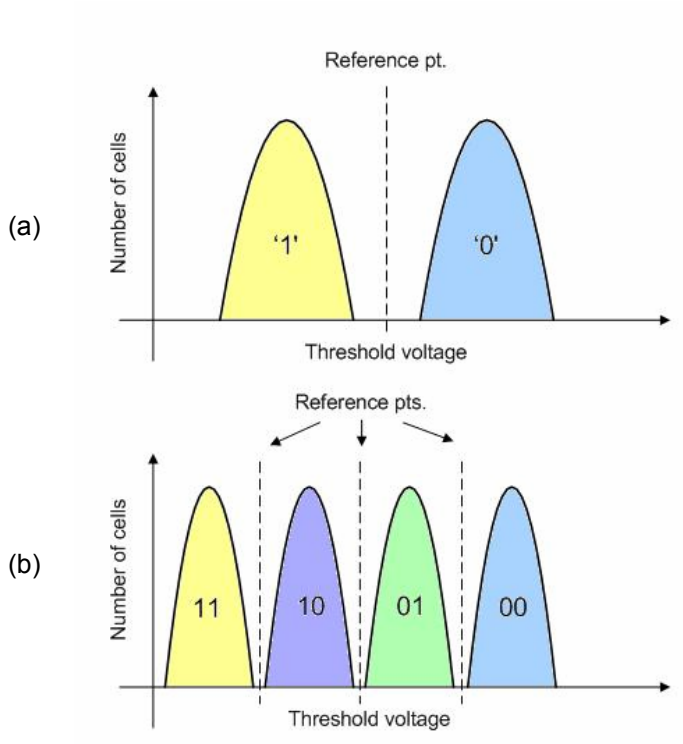

Figure 2.2 Transistor threshold voltage distributions of cells in

(a) an SLC array (b) a 2 bits/cell MLC array

2.1.3 Spansion MirrorBit® Flash Technology

Spansion's MirrorBit® Flash technology physically stores two independent bits on a single cell. This makes it an SLC technology that can store multiple bits per cell. In a MirrorBit cell (Figure 2.3) data is stored as charge trapped in a thin insulating oxide-nitride-oxide (ONO) layer over the junction edges of MOSFET transistors (Note – In a floating gate transistor charge is trapped on a conducting gate terminal). The cell is programmed by injecting channel hot electrons (CHE) into the ONO layer and is erased by band-to-band-generated tunnel-assisted hot hole injection (HHI). The stored charge is sensed by reversing the role of the source and the

drain relative to programming conditions and reading the cell current [2]. Since charge can be stored on both sides of the transistor two-bit operation per cell is attained [3, 4].

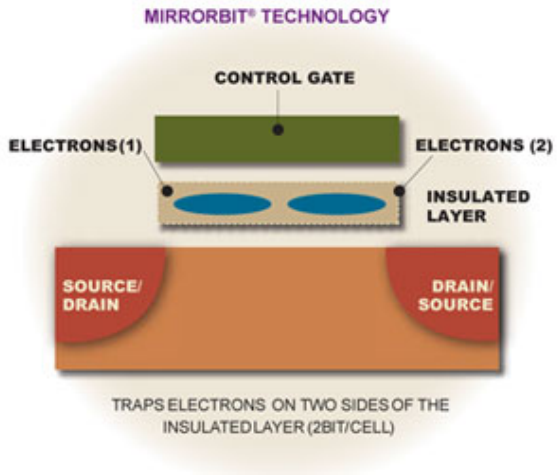

Figure 2.3 A MirrorBit cell

MirrorBit architecture involves storing two bits (MirrorBit) or four bits (MirrorBit Quad) per cell. Thus a single MirrorBit cell can store at least 4 (2 bits/cell) up to 16 (4 bits/cell) levels of charge. This significantly improves the storage density. At smaller geometries, for example, 45nm or 32nm, the ONO layer can store a mere few hundred electrons making it difficult to read the amount of stored charge. Therefore read errors are inherent to Flash technology.

2.1.4 Types of Flash memory: NAND and NOR

Flash memories are classified as – NAND and NOR Flash. These two types differ in the manner in which individual memory cells are connected [13]. Table 2.1 compares NAND and NOR Flash operation.

Table 2.1 Comparing NAND and NOR Flash memory

Parameter	NOR	NAND
Density	1Mbit – 1Gbit	64Mbit – 16 Gbit (or higher)
Read initial access	80ns	20,000ns
Program	2 Mbytes/s	10 Mbytes/s
Erase	2 Mbytes/s	Very high
Access method	Random	Sequential

2.1.5 NOR XiP Execution Model

Random read accesses in NOR result in a very low initial read access time (~80ns) in comparison with sequential access NAND (~20,000ns). Therefore NOR is used for code storage and execution while NAND is used for data storage purposes. NOR is used in eXecute-in-Place (XiP) execution models (Figure 2.4) for code storage and execution. In such a model, the processor executes code directly from memory without copying it into RAM and then executing it. This makes for faster execution of codes while minimizing the RAM requirement of the system resulting in an overall reduction in cost. The XiP Execution Model combines high performance read with relatively inexpensive storage.

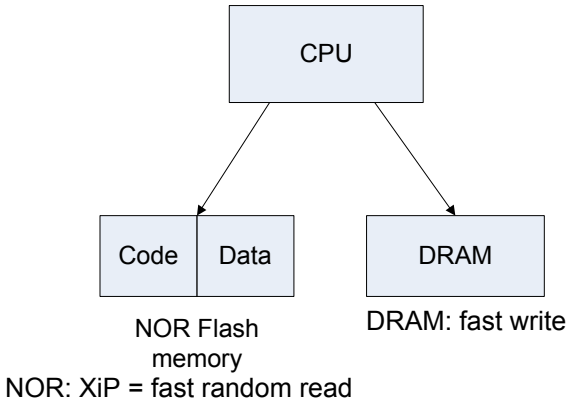

Figure 2.4 NOR XiP Execution model

2.2 Flash Memory Failure Mechanisms

2.2.1 Narrowing of the threshold voltage window

The gate dielectric layer has point defects in which charges get trapped. These trapped charges may migrate and redistribute between traps due to thermal activation. This redistribution may change the threshold voltage (V_t) of the cell ultimately leading to a read error [1]. During an erase cycle, holes are injected into the ONO layer. In the subsequent programming step some holes stay and accumulate from cycle to cycle [1]. This degrades the V_t of the cell. The difference between the threshold voltages of a stored 0 and a 1 is called the *V_t-window*. It is also known as the *Complementary Bit Disturb (CBD)* window because if this window reduces so that the threshold voltages of 1 and 0 overlap, a 1 may be read as a 0 and vice versa.

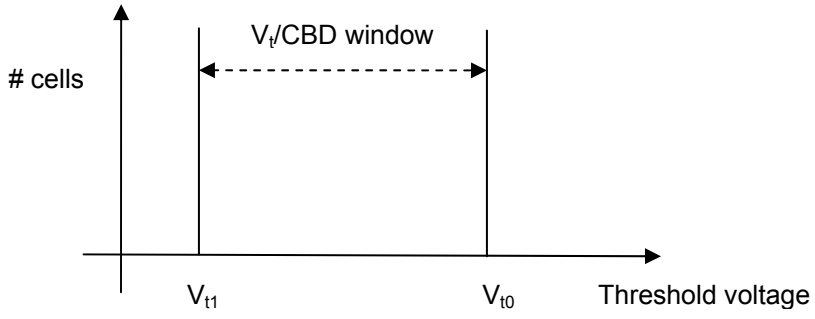

Figure 2.5 V_t/Complementary Bit Disturb window – ideal V_t distributions

2.2.2 Widening of charge distributions

Figure 2.5 illustrates the ideal distributions of the V_t of the cells having a 1 and a 0 respectively. In reality, all the cells storing a 1 (or a 0) do not have the same V_t, their V_t values deviate from the ideal (Figure 2.2). This results in a V_t distribution as shown in Figure 2.6.

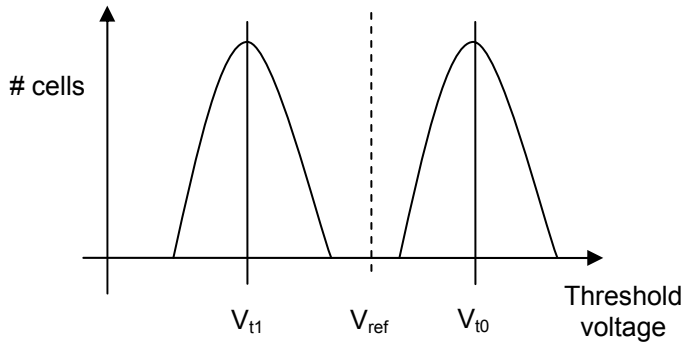

Figure 2.6 Actual V_t distributions of cells storing a '1' and a '0' respectively

As the number of programming cycles increases, trapped charges alter the V_t – distributions shown in Figure 2.6. The distributions widen. This widening of distributions along with the narrowing of the V_t window results in an overlap of the '1' and '0' voltage distribution curves (Figure 2.7). If a programmed or erased bit lies in the overlap region, there is a read error.

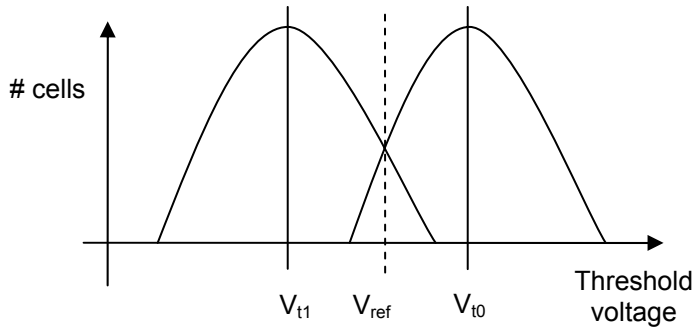

Figure 2.7 Overlap of V_t distributions of cells containing '1' and '0'

2.2.3 Single bit charge loss/gain

Single bit charge gain (SBCG) typically occurs after some program-erase cycling. It results in a very small number of cells having a large threshold voltage shift from the normal distribution. It has been attributed to localized defects in the tunnel oxide [6-8]. Such bits are called 'tail' bits because they lie at the ends of the voltage distribution curves. This effect is

commonly associated with the transient charging and discharging of cycling induced traps in the tunnel oxide [9], [10].

There are primarily three failure mechanisms in MLC Flash memory devices causing random bit errors which affect the reliability of stored data particularly as device geometries become smaller (for example, 45nm or 32nm).

2.3 Standard Approaches against Flash Failures

Data reliability issues in Flash memory are currently being addressed in three ways: by using dynamic reference voltage tracking, compensating technology errors in design cycles and using error correction codes.

2.3.1 Dynamic Reference Voltage Tracking

Bit failures caused by narrowing and shifting of the CBD window can be mitigated to a certain extent by using a *dynamic reference tracking* scheme. In this method, the reference (V_{ref}) of the voltage distributions is made dynamic so that it varies in accordance with the shift in the distribution curves. The dynamic reference is computed by taking an average of three voltages – V_t of a cell containing a '1', a '0' and a *read reference* cell. The *read reference cell* is set to a predetermined V_t value at wafer sort that distinguishes erased bits from programmed bits. The result is a *reference curve* over a period of time instead of a fixed reference line or point. So Figure 2.7 is redrawn as Figure 2.8.

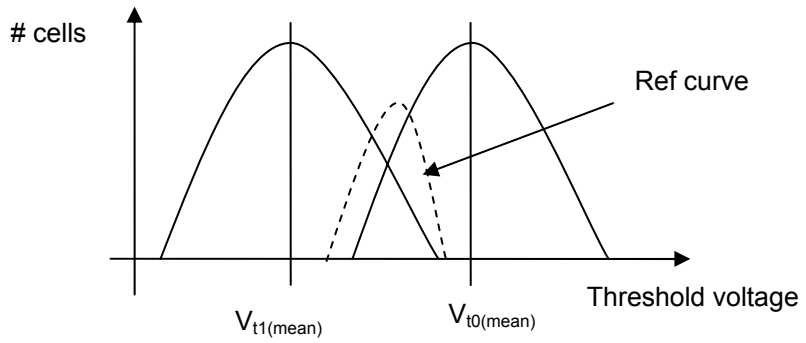

Figure 2.8 Dynamic reference curve

2.3.2 Compensating Technology Errors in Design Cycles

This method involves in-depth, hence prolonged, research on a per product basis to compensate for failure mechanisms for individual designs. Each design is analyzed for possible failure mechanisms and solutions are proposed to eliminate or minimize their effect. This results in increased design cycles which has an unfavorable impact on the cost of and the time taken to market the product.

2.3.3 Using Error Correction Codes (ECCs)

An ECC algorithm can be applied on the controller of the Flash system or on the Flash memory chip itself.

An error correction algorithm can be implemented on a controller which is part of the system architecture but external to the memory chip. The ECC algorithm reconstructs lost information by adding redundancy to the stored data.

This technique allows large scale integration which supports an area effective implementation of ECC algorithms. It is easy to implement robust and more sophisticated ECC algorithms, but at the cost of a significant increase in read latencies. Therefore, it works well for storage-optimized devices where longer read latencies are acceptable. There are techniques to

reduce the read latencies for a controller-based ECC, but for XiP or code-optimized Flash memories the infrastructure for implementing an ECC algorithm on the controller does not exist.

This makes an on-chip ECC the only choice. The major advantage of an on-chip ECC is the algorithm hardware can be scaled down along with the device. On the flip side, complex ECC algorithms cannot be implemented easily as they require large digital circuits. Besides, since the algorithm is on-chip, read latencies have to be kept very small. Therefore the algorithms should be simple. Thus the ECC requirements of the memory can be matched with the complexity of the applied ECC algorithm. Therefore, on-chip ECC is the preferred implementation for latency constrained XiP NOR Flash. Software algorithms can also be implemented on-chip due to the presence of an on-chip 8051-like microcontroller.

On-chip ECC in NOR Flash memory should be constrained to achieve a low latency – typically 10ns, a low implementation complexity which translates to a low gate count (< 5000 NAND gates) in case of hardware and minimal RAM footprint in case of software and most importantly, a target bit error rate ~10^{-15} for the memory array.

2.4 Defining Bit Error Rate as a Measure for Failures

The raw BER of the array should be known accurately in order to decide the error correction capacity of a given ECC algorithm needed to achieve the required target BER (~10^{-15}). The BER prior to using ECC can be computed using the technology data which is plotted as voltage distribution curves for the memory array (Section 2.1). The aggregate error distribution function is defined using the '1' and '0' threshold voltage distribution as a function of the number of programming cycles. The reference voltage varies dynamically as the voltage distribution for the array changes. Based on this distribution and the consequent variation of the reference (V_{ref}), a formula is derived to compute the BER for a given Flash device. Six cases have been considered.

2.4.1 Case 1 – The General Case

In general the 1 and 0 distribution curves overlap. The reference voltage is in the middle of the overlap region (Figure 2.9).

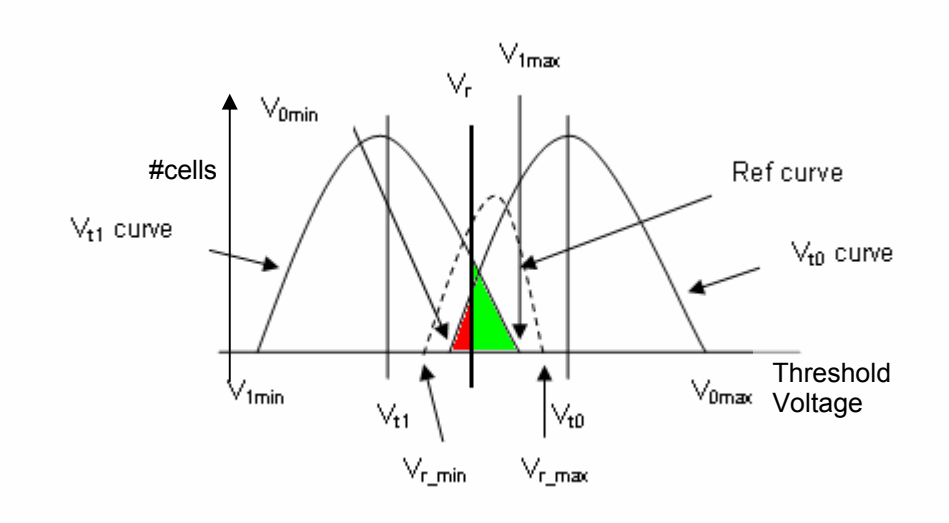

Figure 2.9 General case for calculation of raw array BER using a dynamic reference voltage

The region in green consists of all the erased (1) bits which are read as a 0 while the region in red represents the programmed (0) bits which are read as a 1. At any given time, a read error occurs in a cell if and only if:

i. (program a '0') AND (programmed cell lies in the red region) or

ii. (erase/write a '1') AND (erased cell lies in the green region)

Consider (i):

P('0' read as a '1')

= P(program '0' , $V_{t0} < V_{ref}$)

= {Area (red region)} / {Area (V_{t0} curve) distribution}

$$= \{ \int_{V_{0\min}}^{V_r} P(V_{t0}) dV_{t0} \} / \{ \int_{V_{0\min}}^{V_{0\max}} P(V_{t0}) dV_{t0} \} \qquad (2\text{-}1)$$

Taking into account statistical variability in the decision threshold (reference) voltage V_{ref}, this becomes,

$$P(\text{'0' read as a '1'}) = \{ \int_{V_{r\,min}}^{V_{r\,max}} P(\text{program'0'} \mid V_{t0} < V_{ref}) P(V_{ref}) dV_{ref} \} \quad (2\text{-}2)$$

Similarly, for (ii)

P('1' read as a '0')

= P(erase/write '1' , $V_{t1} > V_{ref}$)

= {Area (green region)}/{Area (V_{t1} curve)}

$$= \{ \int_{V_r}^{V_{1max}} P(V_{t1}) dV_{t1} \} / \{ \int_{V_{1min}}^{V_{1max}} P(V_{t1}) dV_{t1} \} \quad (2\text{-}3)$$

Taking into account statistical variability in the decision (reference) threshold voltage (V_{ref}), this becomes,

$$P(\text{'1' read as a '0'}) = \{ \int_{V_{r\,min}}^{V_{r\,max}} P(\text{program'1'} \mid V_{t1} > V_{ref}) P(V_{ref}) dV_{ref} \} \quad (2\text{-}4)$$

2.4.2 Case 2 – Voltage Distributions at Start of Life

Figure 2.10 is a representation of the distribution at the start of life of the device. The 1 and 0 threshold voltage distributions are very tight and do not overlap each other. The resulting reference voltage (V_{ref}) also lies approximately at the center of the CBD window. Therefore there are no read errors possible in this case.

$$BER = 0 \quad (2\text{-}5)$$

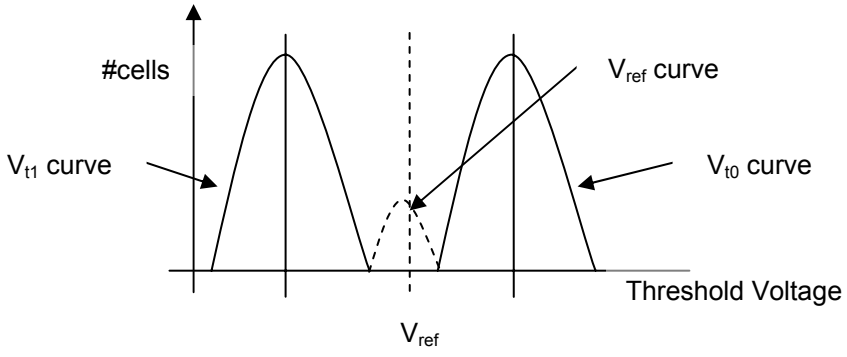

Figure 2.10. Threshold voltage distributions across the memory array at the start of life

2.4.3 Case 3 – Non-overlapping distributions: '1' read as '0'

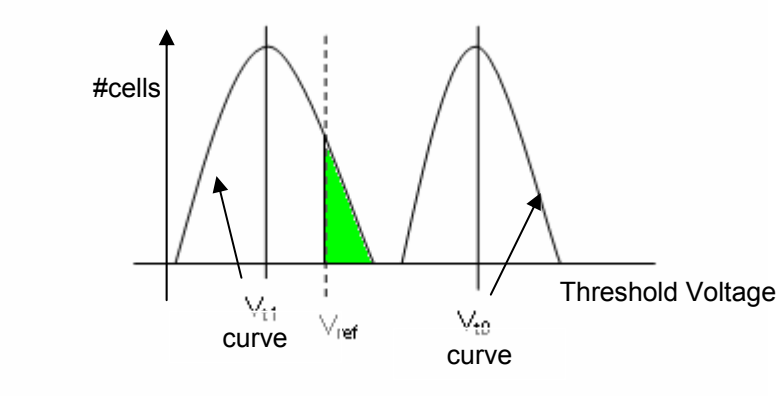

Figure 2.11. Read error in non overlapping distributions ('1' read as a '0')

This (Figure 2.11) is a special case of Case 1. Although the threshold voltage distributions for '0' and '1' do not overlap each other, the computed V_{ref} lies within the '1'-distribution. The cells that lie towards the right of V_{ref} in the shaded region are incorrectly read as a 0. The probability of a cell containing '1' being read as a '0' is:

P('1' read as '0')

= P(erase/write '1' , $V_{t1} > V_{ref}$)

= {Area (green region)}/{Area (V_{t1} curve)}

$$= \{ \int_{V_r}^{V_{1max}} P(V_{t1})dV_{t1} \} / \{ \int_{V_{1min}}^{V_{1max}} P(V_{t1})dV_{t1} / \} \quad (2\text{-}6)$$

Accounting for the statistical variability in the decision threshold voltage (V_{ref}),

$$P(\text{'1' read as a '0'}) = \{ \int_{V_{r\,min}}^{V_{r\,max}} P(\text{program'1'} \mid V_{t1} > V_{ref}) P(V_{ref}) dV_{ref} \} \quad (2\text{-}7)$$

2.4.4 Case 4 - Non-overlapping distributions: '0' read as '1'

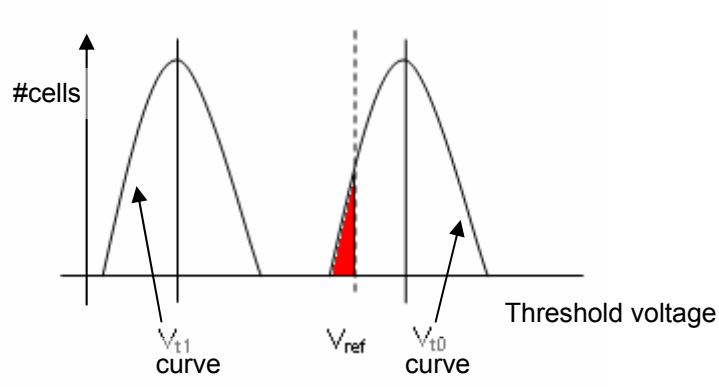

Figure 2.12. Read error in non overlapping distributions ('0' read as a '1')

This is similar to Case 3. Here the reference lies within the V_{t0} distribution. All the cells lying to the left of V_{ref} in the red region are read as '1' instead of a '0'. The probability of this read error is

P('0' read as '1')

= P(program '0' , $V_{t0} < V_{ref}$)

= {Area (red region)} / {Area (V_{t0} curve)}

$$= \{ \int_{V_{0\min}}^{V_r} P(V_{t0})dV_{t0} \} / \{ \int_{V_{0\min}}^{V_{0\max}} P(V_{t0})dV_{t0} / \} \quad (2\text{-}8)$$

However, since the decision threshold voltage (V_{ref}) varies over the lifetime of the device,

$$P(\text{'0' read as '1'}) = \{ \int_{V_{r\min}}^{V_{r\max}} P(\text{program'0'}|\ V_{t0} < V_{ref})P(V_{ref})dV_{ref} \} \quad (2\text{-}9)$$

2.4.5 Case 5 – Overlapping distributions: '1' read as '0'

This is similar to Case 3. The difference is the overlap between the threshold voltage distributions for cells containing '1's and '0's respectively. A read error occurs if a stored '1' is read as a '0', i.e. the memory cell lies in the green region of the distribution.

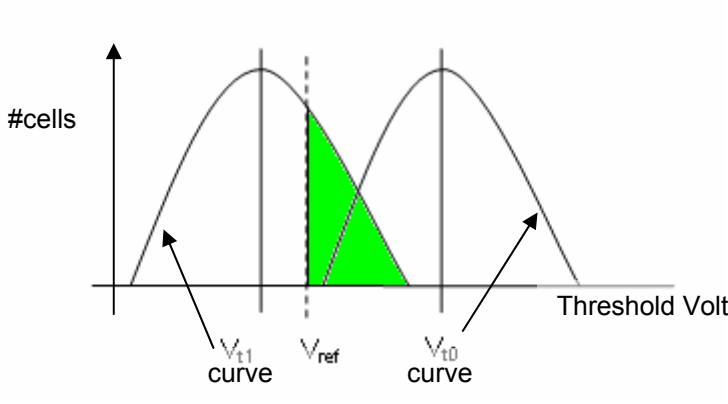

Figure 2.13. Read error in overlapping V_t distributions (‘1’ read as a ‘0’)

The probability of error is given by:

P(‘1’ read as a ‘0’) = P(erase/write ‘1’ , $V_{t1} > V_{ref}$)

= {Area (green region)}/{Area (V_{t1} curve)}

$$= \{ \int_{V_r}^{V_{1max}} P(V_{t1})dV_{t1} \} / \{ \int_{V_{1min}}^{V_{1max}} P(V_{t1})dV_{t1} / \} \quad (2\text{-}10)$$

Considering the statistical variability of V_{ref},

$$P(\text{‘1’ read as a ‘0’}) = \int_{V_{r\,min}}^{V_{r\,max}} P(\text{program'1'} \mid V_{t1} > V_{ref}) P(V_{ref}) dV_{ref} \} \quad (2\text{-}11)$$

2.4.6 Case 6 – Overlapping distributions: ‘0’ read as ‘1’

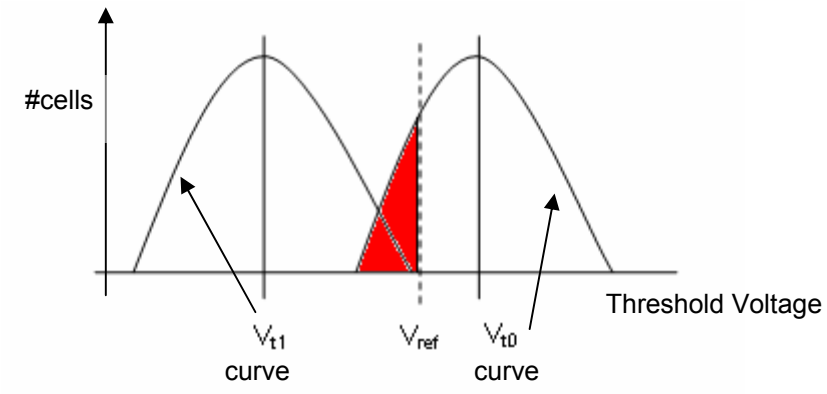

Figure 2.14. Read error in overlapping distributions (‘0’ read as a ‘1’)

This case is similar to Case 4. The probability of a '0' read as a '1' is:

P('0' read as a '1')

= P(program '0' , $V_{t0} < V_{ref}$)

= {Area (red region)} / {Area (V_{t0} curve)}

$$= \{ \int_{V_{0\min}}^{V_r} P(V_{t0})dV_{t0} \} / \{ \int_{V_{0\min}}^{V_{0\max}} P(V_{t0})dV_{t0} / \} \quad (2\text{-}12)$$

Accounting for a variable decision threshold voltage, V_{ref},

$$P(\text{'0' read as a '1'}) = \int_{V_{r\min}}^{V_{r\max}} P(\text{program'0'}|\ V_{t0} < V_{ref})P(V_{ref})dV_{ref} \} \quad (2\text{-}13)$$

Thus a mathematical relation has been developed to compute the current bit error rate using threshold voltage distributions in an array for all possible read error conditions that can occur in Flash memory.

XiP type NOR uses MLC architectures at smaller geometries. These are highly prone to bit disturbs. The bit error rate for a given memory array can be computed mathematically using technology data. Error correction codes are one of the important methods used to maintain data reliability by keeping the BER below 10^{-15}. The low latency requirements of XiP NOR make a low-complexity on-chip ECC the preferred choice. Hamming codes and binary BCH codes are well suited for implementing as on-chip ECC for NOR Flash memory.

Hamming codes are the simplest error correcting block codes. They provide single bit error ECC and 2-bit error detection (EDC) (Section 3.1). These codes and their implementation as an on-chip ECC in Flash has been discussed in [31, 32]. [31] shows a Hamming decoder using asynchronous techniques. Asynchronous pulse generators are used to design a controllable clock for the decoder which is independent of the system clock. The transistors in the pulse generator have to be properly tuned. This may be an unnecessary effort for small

power savings. An area efficient implementation has been proposed in [32] for an on-chip Hamming decoder in NAND Flash. A few hundred gates are required to implement a Hamming decoder as will be proved in a later section. Therefore an area overhead is not a severe problem.

Hamming codes are suitable for a lower order BER (~10^{-12}). They fail to ensure data reliability as BER increases to around 10^{-7} for multi level cell architectures at smaller geometries. This necessitates the use of multi bit error correction. Convolutional codes effectively correct multiple bit errors. However a hardware implementation of a Viterbi encoder/decoder requires 20-30K gates which is very high for NOR Flash. [33] discusses a possible implementation of convolutional codes as on-chip ECC in MLC NOR. It is shown that the BER can be taken from 10^{-2} to 10^{-11} which is not the target bit error rate that is expected to maintain reliability of the Flash array. The performance of this code can be said to be comparable to a BCH algorithm discussed in detail later. Binary BCH codes can be designed to detect and correct multi bit errors. However there is a tradeoff between implementation complexity, latency and error correction capability of the code. The previous chapter discusses two BCH decoding algorithms of which Massey's step-by-step decoding algorithm [20] is expected to be effective in NOR Flash. The possibility of using BCH codes for error control in Flash memories has been mentioned in [36]. However it does not elaborate the possible implementation methods that may be used.

CHAPTER 3

ANALYSIS OF ERROR CORRECTION CODES

On-chip error correction codes effectively compensate for deteriorating bit error rates which are generally of the order of 10^{-7} to 10^{-11} in Flash memory. Error correction codes improve the reliability of data by adding carefully designed redundant data over time (convolutional codes) or space (block codes). Convolutional codes operate on data streams where each bit is processed together with its succeeding and preceding bits. The results improve for higher encoder rates. The encoders and decoders used for convolutional codes (e.g. the Viterbi decoder and the Trellis encoder) are hardware-intensive circuits easily consisting of 20k-50k gates. This makes convolutional codes a good candidate only when hardware overhead is not a stringent constraint. On the other hand, block code algorithms can be designed to work within the latency and hardware constraints of NOR devices making them the codes of choice for on-chip ECC in Flash memory.

3.1. Hamming Codes

Block codes transform large blocks of data into code words. They only use current input data to compute redundancy for given data block as opposed to their convolutional counterparts which use past and future data too. The computed redundancy creates an extended decision ‘space’ around each information/data block. If the data word that is read out from memory lies in the decision space that ‘belongs’ to a certain information block, it is decoded as that information block.

Hamming codes illustrate the creation of redundant space around data blocks to detect and/or correct errors by simple mapping. Simple mechanics and the ease of implementation make these codes a popular choice for communication and data storage systems. However, they are ineffective if the number of random errors is large or if the errors are bursty.

3.1.1. The Mechanics

The (7, 4) (= (*n*, *k*)) Hamming code is the simplest single-error detection and correction, double-error detection code. Here 4 is the information block length (*k*) and 7 is the length of the output code word (*n*). The 3 additional bits (*n*-*k*) in the output code word are the redundant *parity* bits. In the example (Figure 3.1), d_1, d_2, d_3 and d_4 are the data bits to be encoded. p_1, p_2 and p_3 are the parity bits.

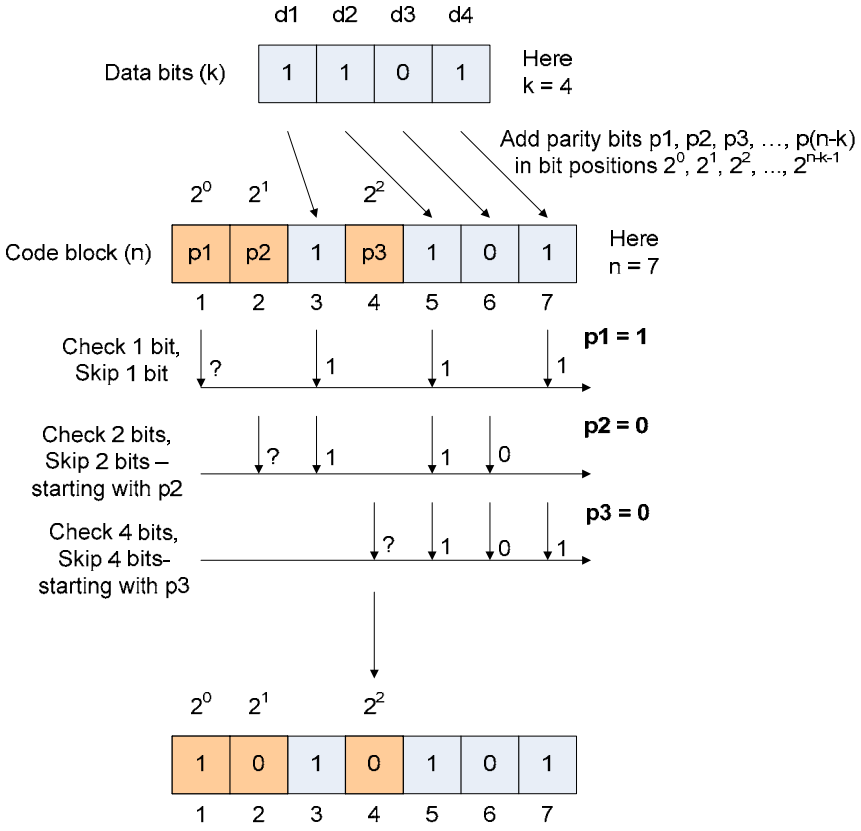

Figure 3.1. The mechanics of encoding of a (7, 4) Hamming code

Suppose [d_1 d_2 d_3 d_4] = [1 1 0 1]. The parity bits (p_1, p_2, p_3, …, p_{n-k}) are inserted in positions which are powers of 2 (2^0, 2^1, 2^2, …, 2^{n-k-1}) because each parity bit represents an even parity check on information bits which are in bit positions whose binary representations have a '1' in the same place as the bit position (binary representation of a power of 2) of the parity. This

has the advantage of making the check positions independent of each other. For this example, the parity bits are computed in the following manner:

For p_1: Check alternate bits to see if even parity condition is satisfied.

$$p_1 = d_1 \oplus d_2 \oplus d_4 = 1 \tag{3-1}$$

For p_2: Check alternate sets of 2 bits each for even parity, starting with p_2

$$p_2 = d_1 \oplus d_2 \oplus d_3 = 0 \tag{3-2}$$

For p_3: Check alternate sets of 4 bits each (starting with p_3) for even parity,

$$p_3 = d_2 \oplus d_3 \oplus d_4 = 0 \tag{3-3}$$

In general, for p_{n-k} alternate sets of 2^{n-k-1} bits each are checked to satisfy even parity. In this example, 1101 is encoded and stored as 1001101 where the underlined bits are parity.

The relation between parity bits and the data bits is represented in matrix form as,

$$\begin{bmatrix} p_1 & p_2 & p_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix} \tag{3-4}$$

To detect and correct a single bit error: Suppose the retrieved code word is 1001111.

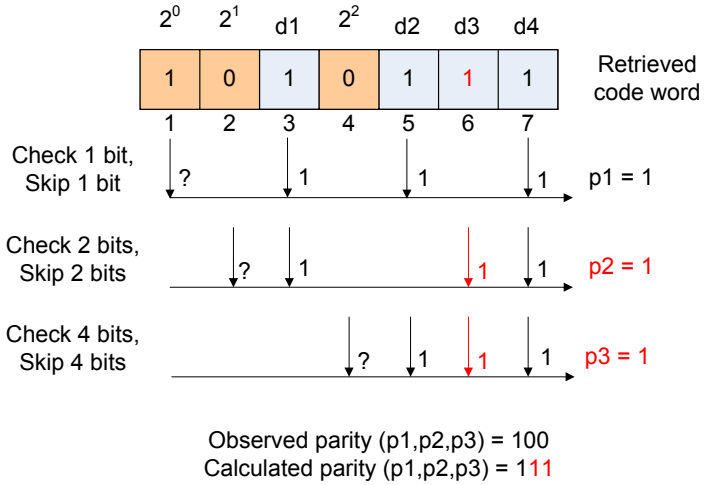

Figure 3.2. Single bit error detection and correction using Hamming codes

Assuming that at most a single bit may be disturbed, the erroneous bit is identified in the following manner (Figure 3.2):

If the observed parity (p_1, p_2, p_3) = 100 and the calculated parity = 111, then the parity bits in conflict are p_2, p_3. The bit that is exclusive to p_2 and p_3 is d_3. Hence d_3 is the disturbed bit.

Now suppose the retrieved code word is 1100101 (2-bit error). In this case, the observed parity (p_1, p_2, p_3) = 110 and the calculated parity = 011 showing that (p_1, p_3) are in conflict. Hence d_2 is incorrectly predicted to be in error.

In Hamming codes with distance 3 a double bit error is indistinguishable from a single bit error in a different code. In order to detect two bit errors and detect and correct a single bit error simultaneously, an additional parity bit is included in the code word. This increases the distance of the code to 4. This additional parity bit is calculated by adding (modulo-2) all the other bits in the code vector. Thus the presence of two bit errors can be detected but it cannot be corrected while a single bit error is detected and corrected.

3.1.2. Encoding

As k increases (e.g. a (31, 26) code), it is impractical to use the above method.

The "generation" of a code word is achieved by a *generator matrix*, **G**. The rows of a generator matrix generate all the code words for a particular code. **G** is a (k x n) matrix for a (n, k) code where n is the length of the code word and k is the data block length. In the k-dimensional vector space of all the binary n-tuples, it is possible to find k linearly independent code words such that every code word **v** is a linear combination of these k code words. If **u** is the information vector then,

$$\mathbf{v} = \mathbf{u}.\mathbf{G} \tag{3-5}$$

The generator matrix for the (7, 4) code is

$$\mathbf{G} = \left[\begin{array}{ccc:cccc} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right] \tag{3-6}$$

Equation (3-6) shows the generator matrix in *systematic form*. The code word obtained after multiplying data with this matrix will have message bits easily distinguishable from the parity bits. Such a form of the encoded code work is called the *systematic form*. The generator matrix in systematic form is

$$\mathbf{G} = [\mathbf{P}\ \ \mathbf{I_k}] \tag{3-7}$$

Where

P = k x (n-k) matrix which generates the parity bits.

$\mathbf{I_k}$ = k x k identity matrix, i.e. a matrix having elements, $b_{ij} = 1$ for all i = j else $b_{ij} = 0$.

Another useful matrix is called the *parity check matrix* (**H**). **H** is a (n-k) x n matrix such that $\mathbf{G.H^T} = \mathbf{0}$ i.e. **G** and **H** are orthogonal. Alternately, an n-tuple **v** is a code word in the code generated by **G** if and only if $\mathbf{v.H^T} = \mathbf{0}$. For **G** as written in Equation (3-7), the parity check matrix **H** is

$$\mathbf{H} = [\mathbf{I_{n-k}}\ \ \mathbf{P^T}] \tag{3-8}$$

Where

$\mathbf{P^T}$ = transpose of matrix **P** in Equation (3-7)

$\mathbf{I_{n-k}}$ = (n-k) x (n-k) identity matrix.

H is called the *parity check matrix* because each row of **H** represents an even parity group with '1's in the positions of the bits that comprise the group.

3.1.3. Decoding

For a (n, k) linear code, if **v** = (v_0, v_1, v_2, …, v_{n-1}) is the stored vector and **r** = (r_0, r_1, r_2, …, r_{n-1}) is the vector read out from memory (received vector), then **r** may differ from **v** due to noise.

$$\mathbf{r} = \mathbf{v} + \mathbf{e} \tag{3-9}$$

Where **e** is an n-tuple called the *error vector* or *error pattern*.

When **r** is read out, the host system should first determine if **r** contains any errors. If so, an error correction algorithm must be executed. Errors are detected by computing the *syndrome*, **s**, which is the vector product of the output vector and the transpose of the parity-check matrix.

$$\begin{aligned} \mathbf{s} &= \mathbf{r}.\mathbf{H}^T \\ &= (\mathbf{v} + \mathbf{e}).\mathbf{H}^T \\ &= \mathbf{e}.\mathbf{H}^T \end{aligned} \tag{3-10}$$

If there are no errors or undetectable errors, $\mathbf{s} = \mathbf{0}$, else $\mathbf{s} \neq \mathbf{0}$. Undetectable error patterns are those which transform **v** into another valid code. Since there are $2^k - 1$ nonzero code words, there are $2^k - 1$ undetectable error patterns.

If **H** is expressed in the systematic form, then Equation (3-10) yields a linear relationship between the syndrome and the error digits.

$$\begin{aligned} s_0 &= e_0 + e_{n-k}p_{00} + e_{n-k+1}p_{10} + \ldots + e_{n-1}p_{k-1,0} \\ s_1 &= e_1 + e_{n-k}p_{01} + e_{n-k+1}p_{11} + \ldots + e_{n-1}p_{k-1,1} \\ &\ldots \\ &\ldots \\ s_{n-k-1} &= e_{n-k-1} + e_{n-k}p_{0,n-k-1} + e_{n-k+1}p_{1,n-k-1} + \ldots + e_{n-1}p_{k-1,n-k-1} \end{aligned} \tag{3-11}$$

Any error correction scheme is a method to solve the (n-k) linear equations of Equation (3-11) for the error digits. Once **e** is found, the vector **r** + **e** is taken as the actual stored code word [15]. Solving these equations is not easy since they have 2^k solutions. One of the most popular decoding schemes for linear block codes like Hamming codes, is the *Standard Array Method* which is also known as the *syndrome decoding* or *table lookup decoding method*.

3.1.3.1. Standard Array Method

For an (n, k) linear block code, there are 2^k valid code vectors, $\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_2}^k$. The received vector **r** may be any one of 2^n possible n-tuples. Any decoding scheme is a rule to partition the 2^n possible received vectors into 2^k disjoint subsets D_1, D_2, …, $D_2{}^k$ such that the

code vector $\mathbf{v_i}$ is contained in the subset D_i for $1 \le i \le 2^k$ [15]. Thus a data 'space' is generated around each valid code word. If **r** is located in the subset D_i, then **r** is decoded into $\mathbf{v_i}$. This decoding is correct only if the actual stored vector was indeed $\mathbf{v_i}$.

$\mathbf{v_1}$	$\mathbf{v_2}$	…	$\mathbf{v_i}$	…	$\mathbf{v_2^k}$
$\mathbf{v_1+e_2}$	$\mathbf{e_2+v_2}$	…	$\mathbf{e_2+v_i}$	…	$\mathbf{e_2+v_2^k}$
$\mathbf{v_1+e_3}$	$\mathbf{e_3+v_2}$	…	$\mathbf{e_3+v_i}$	…	$\mathbf{e_3+v_2^k}$
			.		
			.		
			.		
$\mathbf{v_1+e_i}$	$\mathbf{e_i+v_2}$	…	$\mathbf{e_i+v_i}$	…	$\mathbf{e_i+v_2^k}$
			.		
			.		
			.		
$\mathbf{v_1+e_2^{n-k}}$	$\mathbf{e_2^{n-k}+v_2}$	…	$\mathbf{e_2^{n-k}+v_i}$	…	$\mathbf{e_2^{n-k}+v_2^k}$

Figure 3.3. Standard array for an (n, k) linear code

$\mathbf{v_1}$, $\mathbf{v_2}$, …, $\mathbf{v_2^k}$ are the 2^k valid code vectors of the (*n*, *k*) linear block code. Generally, $\mathbf{v_1} = \mathbf{0}$. $\mathbf{e_2}$, $\mathbf{e_3}$, …, $\mathbf{e_2^{n-k}}$ are distinct *n*-tuples from the remaining $2^n - 2^k$ n-tuples. Thus, all the *n*-tuples are used in the array. Each subset D_i for $1 \le i \le 2^{n-k}$ is the i^{th} column in this array (Figure 3.3). The 2^{n-k} rows are known as the *cosets* and the n-tuples $\mathbf{e_2}$, $\mathbf{e_3}$, …, $\mathbf{e_2^{n-k}}$ are the *coset leaders* for the corresponding rows. **r** is decoded correctly only if the error pattern is a coset leader. All the 2^k *n*-tuples of any given coset have the same syndrome. The syndromes for different cosets differ.

Summarizing the decoding process for a linear block code: The first step is to compute the syndrome $\mathbf{s} = \mathbf{r}.\mathbf{H}^T$. This helps to locate the corresponding coset leader, say $\mathbf{e_t}$, for this

syndrome. The deduced coset leader is used to decode **r** as $\mathbf{v} = \mathbf{r} + \mathbf{e}_t$. **u** is easily obtained from **v** since it is in the systematic form.

3.1.4. Error Detecting and Correcting Capabilities

An important property of a code is the code's minimum *distance*, also known as the minimum *Hamming distance*. *Hamming distance* is the number of positions in which two code words differ. The minimum Hamming distance, d_{min}, is the least possible distance between a pair of code words for a given code. It determines the error-detecting/correcting capabilities of the code. The rows of generator matrix **G** define a basis for the code vectors. For Hamming codes, the basis (rows of **G**) satisfies even parity conditions. This characterizes d_{min} (= 3) for a Hamming code.

If a code has minimum distance d_{min}, no error patterns of $d_{min} - 1$ or fewer errors can change one code vector into another code vector. Therefore, the random error-detecting capability of a block code with minimum distance d_{min} is $d_{min} - 1$.

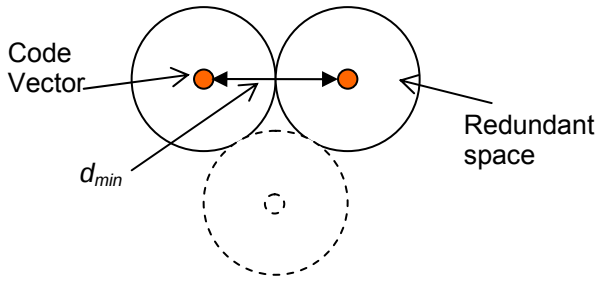

Figure 3.4. 2D representation of redundancy around each code vector and the concept of d_{min}

In order to be able to detect and correct an error correctly, the received vector must lie in the space surrounding the corresponding transmitted vector (Figure 3.4). If the received vector differs from the transmitted vector in *t* places (*t* errors); it can be corrected only if

$$t \leq \lfloor (d_{min} - 1)/2 \rfloor \qquad (3\text{-}12)$$

Where $\lfloor (d_{min} - 1)/2 \rfloor$ denotes the largest integer no greater than $(d_{min} - 1)/2$.

For a Hamming code d_{min} = 3. Hence it can detect up to 2 bit errors and can detect and correct a single bit error.

3.1.5. Cyclic Hamming Codes

Cyclic Hamming codes can be represented as a polynomial. For **v** = (v_0, v_1, …, v_{n-1}), the *code polynomial* is

$$\mathbf{v}(X) = v_0X^0 + v_1X^1 + v_2X^2 + \ldots + v_{n-1}X^{n-1} \quad (3\text{-}13)$$

The power of X denotes the position of the code vector component. For example, the vector (1 0 0 1 0 1 1) is represented as $1 + X^3 + X^5 + X^6$.

The encoding and decoding circuits of a cyclic code consist of shift registers with feedback connections. The power of X represents the number of serial shifts of the components through the circuit. Due to the cyclic structure of the code, the circuit may be designed to decode the first received bit and decode subsequent bits using the same circuitry. Thus a cyclic code saves hardware but increases the computational delay which is not favorable for NOR Flash. On the other hand, the linear decoding process explained in Section 3.1.3 can be made parallel making the circuit faster. Therefore linear Hamming codes are preferred over cyclic Hamming codes for implementation in NOR Flash devices.

Hamming codes are very simple to understand and implement. They are well-suited when the BER has to be improved from around 10^{-10} to 10^{-15}. They fail to detect and correct multi bit errors and hence are inefficient as raw BER increases to 10^{-6}.

3.2. Multi Bit Error Correction: BCH Codes

The Bose-Chaudhuri-Hocquenghem (BCH) codes are a generalized form of Hamming codes for multi bit error detection and correction. They include both binary and multilevel codes. Multi level BCH codes are suitable for correcting burst errors. Since the errors in a Flash device are random, it is convenient to use binary BCH codes for Flash memory.

Before evaluating BCH codes, a review of *finite field arithmetic* is required in order to understand and utilize multi bit error correcting codes. Binary arithmetic is a subset of finite field arithmetic. A *finite field* is a set of finite elements over which math operations like addition, subtraction, multiplication and division generate values which belong to the same set. More formally it can be said that addition, subtraction, multiplication and division are *closed* on the field. A finite field is also known as a *Galois field* (GF). A Galois field in which the elements can take q different values is referred to as GF(q) [16]. In GF(p), where p is a prime number, modulo-p arithmetic is used. An example of modulo-2 arithmetic is shown in Figure 3.5.

+	0	1
0	0	1
1	1	0

(a)

.	0	1
0	0	0
1	1	0

(b)

Figure 3.5. (a) Modulo-2 addition (b) Modulo-2 multiplication

In any prime size field, there is always at least one element whose powers constitute all the nonzero elements of the field [16]. This element is called the *primitive*. For example, in GF(5), the number 3 is primitive because:

$3^0 = 1$

$3^1 = 3$

$3^2 = 4$

$3^3 = 2$(all modulo-5)

The pattern repeats for higher powers of 3.

The finite field GF(p) can be extended to a field of p^m elements where m is any positive integer. The field thus formed, GF(p^m), is called an *extension field* of GF(p). For binary BCH codes, GF(2) and its extension field GF(2^m) are used.

A single bit error detection and correction code like a Hamming code differs from a multi bit error detection and correction code in the manner in which the data bits are utilized to generate redundant space. For a Hamming code there is only one set of linearly independent equations, hence it can detect and correct only single bit errors. On the other hand, if multiple sets of linearly independent equations are superimposed on the same data bits, a code that can correct multiple errors, each corresponding to a linearly independent set of equations is obtained.

3.2.1. The Mechanics

Code vectors in polynomial form can be represented graphically. For example, **u** = (1 1 1) ≡ **u**(X) = $1+X+X^2$ is represented as shown in Figure 3.6.

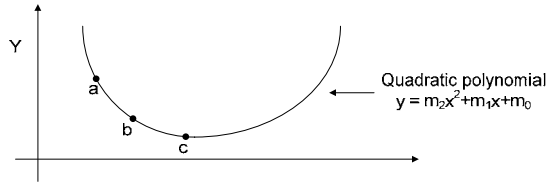

Figure 3.6. Graphical representation of **u** = (1 1 1)

For a polynomial of degree *n*, *n*+1 distinct points describe the polynomial completely. The idea behind BCH codes is to store more than *n*+1 points satisfying the polynomial. This is *oversampling* the polynomial. While reading out data, as long as any set of *n*+1 correct points is read out, the information polynomial **u**(X) can be rebuilt.

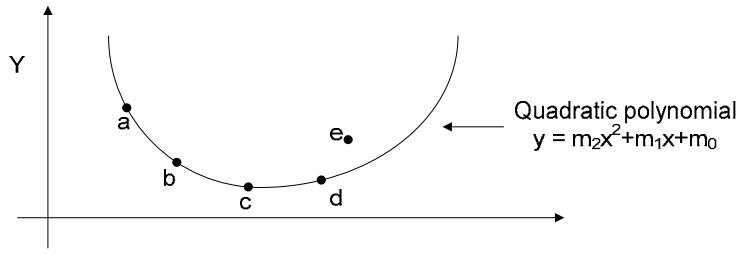

Figure 3.7. Oversampling polynomial **u**(X)

In the above example, **u**(X) is oversampled with 5 points, a, b, c, d and e (Figure 3.7). When these points are transmitted point e is disturbed. Polynomials are constructed using all permutations of n + 1 (= 3, here) points. The results are tabulated in Table 3.1.

Table 3.1. Reconstructing the oversampled polynomial u(X)

Points	Corresponding polynomial
a, b, c	**u**(X)
a, b, d	**u**(X)
a, b, e	$\mathbf{u}_1^*(X)$
a, c, d	**u**(X)
a, c, e	$\mathbf{u}_2^*(X)$
a, d, e	$\mathbf{u}_3^*(X)$
b, c, d	**u**(X)
b, c, e	$\mathbf{u}_4^*(X)$
b, d, e	$\mathbf{u}_5^*(X)$
c, d, e	$\mathbf{u}_6^*(X)$

u(X) occurs more frequently than any other polynomial ($\mathbf{u}_i^*(X)$ represents any polynomial other than the correct one). Hence it is assumed that **u**(X) is the information vector that was transmitted. This illustrates how oversampling a polynomial helps to recover the actual data at the receiving end.

3.2.2. Encoding

The encoding process for BCH codes can be explained in the following manner: The first step is to oversample **u**(X) which is done by multiplying it with **g**(X). The code polynomial can be written as **v**(X) = **u**(X).**g**(X) (Equation 3-5). **u** cannot be assumed to be oversampled since it represents user information which has to be protected. Therefore the only means of

introducing redundancy is via appropriate selection of **g**. The example in Section 3.2.1 showed that for every error to be detected there should be at least two redundant points or samples. Therefore for *t* errors to be detected **g** should introduce atleast 2*t* redundant samples. This implies **g**(X) should have at least 2*t* roots.

Consider formulating these statements mathematically, i.e. showing how the 2t roots of **g**(X) help introduce redundancy in **u**(X) by oversampling it.

If *n* denotes the length of the code word **v** and *k* is the length of the information vector **u** then *n* and *k* are related such that the number of redundant symbols is

$$n - k \leq mt \text{ [15]}$$

where *m* is a positive integer ($m \geq 3$) defined in terms of *n* as $n = 2^m - 1$ and *t* is the maximum number of errors that can be corrected ($t < 2^{m-1}$).

The generator polynomial **g**(X) for a binary BCH code is the *lowest degree polynomial* over GF(2) which has ($\alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2t}$) as its roots i.e. $g(\alpha^i) = 0$ for $1 \leq i \leq 2t$ [15]; where α is a primitive element in GF(2^m). *Lowest degree polynomial* implies **g**(X) has only ($\alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2t}$) as its roots and no roots other than these. Using a binary Galois field is a typical scenario since information is generally represented as a combination of 1's and 0's. An extended binary field GF(2^m) uses binary symbols of length *m*.

The 2*t* roots of **g**(X) indicate that for *t* bit errors to be detected and corrected at least 2*t* redundant samples are introduced. **g**(X) can be represented in the factored form. The generic way of doing this is

$$\mathbf{g}(X) = (X + \alpha).(X + \alpha^2).(X + \alpha^3)\ldots(X + \alpha^{2t}) \quad (3\text{-}14)$$

However, according to the definition of **g**(X), its coefficients lie in GF(2). Therefore, equation (3-14) is not the correct definition of **g**(X) in terms of its roots. Instead **g**(X) can be defined in terms of *minimal polynomials* of each of ($\alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2t}$). To include all roots that may have coefficients over GF(2^m), the following procedure is used:

The minimal polynomial $\Phi(X)$ of any element β in $GF(2^m)$ is defined as the polynomial of smallest degree over GF(2) having β as its root, i.e. $\Phi(\beta) = 0$. For example, the minimal polynomial of 0 in $GF(2^m)$ is X and that of 1 is X + 1.

Suppose the minimal polynomial of α^i for $1 \le i \le 2t$ is denoted by $\Phi_i(X)$. Therefore, the factored form of $\mathbf{g}(X)$ can be written as

$$\mathbf{g}(X) = \text{LCM}\{\Phi_1(X), \Phi_2(X), \ldots, \Phi_{2t}(X)\} \tag{3-15}$$

LCM: Least Common Multiple

Equation (3-15) is simplified using the concept of conjugates in a finite field, explained below:

If β is an element in $GF(2^m)$, then the element $(\beta)^{p\wedge 2}$, which also belongs to $GF(2^m)$ satisfies $[f(\beta)]^{p\wedge 2} = f(\beta^{p\wedge 2})$. β and $(\beta)^{p\wedge 2}$ are called conjugates. Conjugates are transparent to the order of equations.

$f(X)$: polynomial with binary coefficients.

Every even power of α in $GF(2^m)$ can be written in terms of an odd power and a power of 2. For example, $\alpha^{12} = (\alpha^3)^4 = (\alpha^3)^{2\wedge 2}$. Therefore every even power of α is a conjugate of some preceding odd power of α. In the above example, α^{12} is a conjugate of α^3.

Appendix A shows that a root and its conjugate have the same minimal polynomial. Therefore all roots contained in the even polynomials are contained in the odd polynomials as well. Therefore the even polynomials do not contribute to $\mathbf{g}(X)$ and can be eliminated. The expression for $\mathbf{g}(X)$ is reduced to

$$\mathbf{g}(X) = \text{LCM}\{\Phi_1(X), \Phi_3(X), \ldots, \Phi_{2t-1}(X)\} \tag{3-16}$$

This is the generator polynomial for a binary t-error-correcting BCH code of length $2^m - 1$.

For a single-error-correcting BCH code of length $2^m - 1$, the generator matrix is

$$\mathbf{g}(X) = \Phi_1(X) \tag{3-17}$$

Since α is a primitive element of $GF(2^m)$, $\Phi_1(X)$ is a primitive polynomial of degree m. Therefore, the single-error-correcting BCH code of length $2^m - 1$ is a Hamming code. [15]

Thus while a single-error-correcting Hamming code is defined by a single primitive polynomial, a *t*-error-correcting BCH code is defined by *t* primitive polynomials as explained in Equation (3-16).

Here is an illustrative example:

For a (15, 5) (= (*n*, *k*)) triple-error correcting code. The assumption $t = 3$ makes the example simpler to understand.

$n = 15 = 2^m - 1 \Rightarrow m = 4$.

Let α be a primitive element in GF(2^4). According to the definition; α, α^2, α^3, α^4, α^5 and α^6 are the roots of $\mathbf{g}(X)$ for the this code. The minimal polynomials for α, α^2 and α^4 are identical (Appendix A) and

$$\Phi_1(X) = \Phi_2(X) = \Phi_4(X) = 1 + X + X^4 \tag{3-18}$$

The minimal polynomials for α^3 and α^6 are the same,

$$\Phi_3(X) = \Phi_6(X) = 1 + X + X^2 + X^3 + X^4 \tag{3-19}$$

The minimal polynomial for α^5 is,

$$\Phi_5(X) = 1 + X + X^2 \tag{3-20}$$

Taking the LCM, the generator polynomial for the (15, 5) triple-error correcting BCH code is,

$$\mathbf{g}(X) = 1 + X + X^2 + X^4 + X^5 + X^8 + X^{10}$$

The parity-check matrix for BCH codes can be derived from the roots of $\mathbf{g}(X)$. Since the code polynomial, $\mathbf{v}(X) = \mathbf{u}(X).\mathbf{g}(X)$, $\mathbf{v}(X)$ has $\alpha, \alpha^2, \alpha^3, \ldots, \alpha^{2t}$ as its roots. Hence, for $1 \le i \le 2t$,

$$\mathbf{v}(\alpha^i) = v_0 + v_1\alpha^i + v_2\alpha^{2i} + \ldots + v_{n-1}\alpha^{(n-1)i} = 0 \tag{3-21}$$

This equation can be written as a product of two matrices where each element of the resulting vector equals Equation (3-21) for the corresponding i; $1 \le i \le 2t$.

$$(v_0, v_1, \ldots, v_{n-1}).\begin{bmatrix} 1 & 1 & 1 & . & \ldots & 1 \\ \alpha & \alpha^2 & \alpha^3 & & \ldots & \alpha^{2t} \\ \alpha^2 & \alpha^4 & \alpha^6 & & \ldots & \alpha^{4t} \\ \alpha^3 & \alpha^6 & \alpha^9 & & & \alpha^{6t} \\ . & . & & & & . \\ . & . & & & & . \\ \alpha^{n-1} & \alpha^{2(n-1)} & \alpha^{3(n-1)} & . & \ldots & \alpha^{2t(n-1)} \end{bmatrix} = 0 = \mathbf{v.M} \qquad (3\text{-}22)$$

Comparing this with $\mathbf{v.H^T} = 0$ (Equation 3-8) the parity-check matrix, $\mathbf{H} = \mathbf{M^T}$, for the BCH code:

$$\mathbf{H} = \begin{bmatrix} 1 & \alpha & \alpha^2 & \alpha^3 & \ldots & \alpha^{n-1} \\ 1 & \alpha^2 & \alpha^4 & \alpha^6 & \ldots & \alpha^{2(n-1)} \\ 1 & \alpha^3 & \alpha^6 & \alpha^9 & \ldots & \alpha^{3(n-1)} \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & . \\ 1 & \alpha^{2t} & \alpha^{4t} & \alpha^{6t} & \ldots & \alpha^{2t(n-1)} \end{bmatrix} \qquad (3\text{-}23)$$

3.2.3. Decoding

Decoding BCH codes involves detecting and correcting a larger number of errors as compared with single bit error detection and correction in Hamming codes.

There are two popular approaches, namely, the Standard Algebraic Decoding Method and Massey's Step-by-Step Decoding Algorithm [20]. There are several modifications made to the Step-by-Step algorithm [21- 23]. The basic Step-by-Step algorithm [20] has been analyzed here.

3.2.3.1. Standard Algebraic Decoding Method

This method closely emulates the mechanics described earlier (Section 3.2.1). The decoding process has two steps: Error detection and Error Correction.

To detect errors all probable n^{th} degree information polynomials ($\mathbf{u}(X)$) are constructed using all possible permutations of $n+1$ points. If the polynomials are not all equal, it implies at least one of the points is disturbed. Errors are corrected by identifying the most frequently occurring polynomial, from amongst those constructed in the previous step, and assuming it to

be the stored information **u**. The point(s) common to the remaining (less frequently occurring, hence assumed to be erroneous) polynomials is/are said to be the disturbed sample(s).

This process can be mathematically formulated in the following manner:

Suppose the code vector stored at a memory location is **v**(X). If **r**(X) is the vector that is read out and **e**(X) is the error pattern then

$$\mathbf{r}(X) = \mathbf{v}(X) + \mathbf{e}(X)$$

The syndrome **S** is constructed in order to determine the presence or absence of errors. If **S** is non-zero it indicates the presence of errors and vice versa. For a *t*-error-correcting BCH code, **S** is a 2*t*-tuple since **H** is a 2*t* x *n* matrix.

$$\mathbf{S} = (S_1, S_2, \ldots, S_{2t}) = \mathbf{r}.\mathbf{H}^T = \mathbf{e}.\mathbf{H}^T \tag{3-24}$$

where $S_i = \mathbf{r}(\alpha^i) = \mathbf{e}(\alpha^i)$ for $1 \le i \le 2t$.

Error Correction involves finding the exact locations of disturbed samples and correcting the samples by complementing them. The error-location algorithm for BCH codes is designed to detect locations of multiple-errors. Equation (3-24) shows that **S** depends only on the error pattern **e**. Suppose **e**(X) has ν errors at locations $X^{j1}, X^{j2}, \ldots, X^{j\nu}$.

$$\mathbf{e}(X) = X^{j1} + X^{j2} + \ldots + X^{j\nu} \tag{3-25}$$

where $0 \le j_1 < j_2 < \ldots < j_\nu \le n$.

Combining Equations (3-24) and (3-25);

$$S_1 = \alpha^{j1} + \alpha^{j2} + \ldots + \alpha^{j\nu} \tag{3-26}$$

$$S_2 = (\alpha^{j1})^2 + (\alpha^{j2})^2 + \ldots + (\alpha^{j\nu})^2$$

$$S_3 = (\alpha^{j1})^3 + (\alpha^{j2})^3 + \ldots + (\alpha^{j\nu})^3$$

.

.

.

$$S_{2t} = (\alpha^{j1})^{2t} + (\alpha^{j2})^{2t} + \ldots + (\alpha^{j\nu})^{2t}$$

where α^{j1}, α^{j2}, …, $\alpha^{j\nu}$ are unknown. Any method for solving these equations is a decoding algorithm for the BCH codes. [15]

Let $\beta_p = \alpha^{jp}$ for $1 \le p \le \nu$. These are the *error location numbers*. Equation (3-26) is rewritten as

$$S_1 = \beta_1 + \beta_2 + \ldots + \beta_\nu \tag{3-27}$$

$$S_2 = \beta_1^2 + \beta_2^2 + \ldots + \beta_\nu^2$$

$$S_3 = \beta_1^3 + \beta_2^3 + \ldots + \beta_\nu^3$$

.

.

.

$$S_{2t} = \beta_1^{2t} + \beta_2^{2t} + \ldots + \beta_\nu^{2t}$$

The error locator polynomial is defined [15] as:

$$\sigma(X) = (1 + \beta_1 X)(1 + \beta_2 X) \ldots (1 + \beta_\nu X) \tag{3-28}$$

$$= \sigma_0 + \sigma_1 X + \sigma_2 X^2 + \ldots + \sigma_\nu X^\nu$$

The roots of this polynomial specify the locations of the errors. S_1, S_2, …, S_{2t} can be written in terms of the roots of $\sigma(X)$.

For the (15, 5) BCH code example (Section 3.2.2) suppose the received systematic code polynomial is,

$$\mathbf{r}(X) = X^3 + X^5 + X^{12}$$

The syndrome components are remainders when $\mathbf{r}(X)$ is divided by $\Phi_1(X)$, $\Phi_3(X)$, $\Phi_5(X)$ (Equation 3-18, 3-19 and 3-20) successively,

$$\mathbf{b}_1(X) = 1, \tag{3-29}$$

$$\mathbf{b}_3(X) = 1 + X^2 + X^3$$

$$\mathbf{b}_5(X) = X^2$$

Using the power and polynomial representations of the elements in GF(2^4) and substituting α, α^2 and α^4 into $\mathbf{b}_1(X)$,

$$S_1 = S_2 = S_4 = 1$$

Substituting α^3 and α^6 into $\mathbf{b}_3(X)$,

$$S_3 = 1 + \alpha^6 + \alpha^9 = \alpha^{10},$$

$$S_6 = 1 + \alpha^{12} + \alpha^{18} = \alpha^5$$

Substituting α^5 into $\mathbf{b}_5(X)$,

$$S_5 = \alpha^{10}$$

The error locator polynomial is determined by an iterative procedure known as Berlekamp's iterative algorithm. The roots of this polynomial or the error location numbers are found using Peterson's substitution method [18] or Chien's search algorithm [19], the latter preferred because of available optimizations of its hardware implementation. The simplest way to implement Chien's search algorithm is using a look-up table. This is impractical for large block sizes. Another hardware circuit used is called the Chien searcher [21]. The complexity of the circuit increases in proportion to the block length. Hence it too is inefficient for long block codes.

Summarizing standard algebraic decoding method, the first step is to compute the syndrome, $\mathbf{S} = \mathbf{r}.\ \mathbf{H}^T$ in order to detect errors. If **S** is nonzero, which implies there are errors in the received word, then the error location polynomial is determined from the components of **S**. The roots of this polynomial give the error-location numbers. Once the error locations have been identified, correcting these errors is to complement the bit at that position.

The Step-by-Step decoding algorithm (Section 3.2.3.2) proposed by Massey is an efficient alternative to the standard algebraic method.

3.2.3.2. Massey's Step-by-Step Decoding Algorithm

The step-by-step decoding algorithm corrects erroneous bits by checking the impact of changing (complementing) each bit on the total number of errors in the code word. If changing a

bit reduces the number of errors, then the change is retained else the bit is changed back to its original value. This algorithm has been explained by Massey in [27].

It is easier to implement this algorithm since it avoids calculating the coefficients of an error locator polynomial and searching the roots [21]. It exploits the cyclic nature of BCH codes.

The decoding algorithm is explained below:

Errors are detected by computing the syndrome **S** (Equation 3-10). A nonzero value of the syndrome indicates the presence of errors.

For correcting the errors: The weight of the error pattern **e** indicates the number of errors in **r**. Each bit of **r** is successively complemented (one bit at a time) and the weight of the resulting error pattern is observed. If the weight reduces it implies the bit under consideration was erroneous and has been corrected by complementing it. On the other hand, if the weight increases, it is assumed that the bit is not disturbed and its value should remain unchanged. Since this method involves changing the received symbols one at a time and testing the resultant weight of error pattern, it is called the *step-by-step algorithm.*

To formulate this algorithm mathematically, consider the received vector is **r**(X) which is equal to **v**(X) + **e**(X) where **v** is the transmitted (stored) code word and **e** is the error pattern.

For a *t*-error correcting BCH code (Equation 3-24)

$$\mathbf{S} = (S_1, S_2, \ldots, S_{2t}) = \mathbf{r}.\mathbf{H}^T = \mathbf{e}.\mathbf{H}^T$$

where $S_i = \mathbf{r}(\alpha^i) = \mathbf{e}(\alpha^i)$ for $1 \leq i \leq 2t$.

A non zero value of **S** indicates the presence of errors in **r**.

The total number of non zero elements in **e**, i.e. the Hamming weight of **e**, gives the total number of errors in the received code word. This helps the decoder determine when it should stop complementing the code word bit-by-bit.

Massey defines a procedure [20] to determine the weight of **e**. The total number of errors can be estimated by determining the singularity or non singularity of matrix, $\mathbf{L}_j$. (Equation 3-30)

The matrix $\mathbf{L}_j$ [20] is defined for any binary BCH code having code length n and any j such that $1 \leq j \leq n\text{-}1$, $\mathbf{L}_j$ is the j x j matrix

$$\mathbf{L}_j = \begin{bmatrix} S_1 & 1 & 0 & 0 & \dots & 0 \\ S_3 & S_2 & S_1 & 1 & \dots & 0 \\ & & \vdots & & & \\ S_{2j-1} & S_{2j-2} & S_{2j-3} & S_{2j-4} & \dots & S_j \end{bmatrix} \tag{3-30}$$

$\mathbf{L}_j$ is singular if the weight of $\mathbf{e}$ is j -1 or less, and is non-singular if the weight of $\mathbf{e}$ is j or j+1 [20].

Using this property the number of errors can be defined in terms of det($\mathbf{L}_1$), det($\mathbf{L}_2$), …, det($\mathbf{L}_t$). For example, det($\mathbf{L}_4$) = 0 implies that the number of errors is three or less. The exact number of errors can be determined in terms of the relations among det($\mathbf{L}_1$), det($\mathbf{L}_2$), .., det($\mathbf{L}_t$). For example, if det($\mathbf{L}_1$) ≠ 0, det($\mathbf{L}_2$) ≠ 0, and det($\mathbf{L}_p$) = 0 for p = 3, 4, …, t, then two errors have occurred [20]. The only thing that is of consequence here is whether det($\mathbf{L}_p$) ($1 \leq p \leq t$) is equal to zero or not.

The process of complementing bits and checking the weight of the resulting error pattern against the weight of the actual error pattern can be done by decoders in parallel. This reduces the computational time at a nominal hardware overhead. This makes this algorithm one of the prime choices for use in low latency NOR Flash.

Thus, broadly, there are two error correction codes to choose from – Hamming and BCH. Selecting an appropriate one depends on the current BER of the array and the required BER (~ 10^{-15} for NOR). The error correction capacity of the algorithm is a function of these parameters.

3.3. Computing Required Error Correction Capacity

The current BER of a memory array can be computed from the voltage distribution curves (Section 2.4).

The current BER for a given memory array is denoted by $\mathbf{P_e}$ which denotes the probability of one exclusive bit to be in error in the array. Using this, the probability that a single bit is in error in a *k*-bit block is given by

$$1 - (1\text{-}P_e)^k = \text{current BER of the } k\text{-bit block} \tag{3-31}$$

After using ECC, the size of the encoded block (parity + data) = *n* (say) and the maximum no. of errors that can be corrected = *t*

Therefore the probability of an error in a *n*-bit block after applying ECC is given by

P(error in a n-bit block after ECC) = P_{ECC}

$$= 1 - P(\text{all possible errors that the ECC can correct})$$

$$= 1 - \{P(0 \text{ error}) + P(1b \text{ error}) + P(2b \text{ error}) + \ldots + P(t\text{-bit errors})\}$$

$$= 1 - \{ {}^nC_0.P_e^{\,0}.(1\text{-}P_e)^n + {}^nC_1.P_e^{\,1}.(1\text{-}P_e)^{n-1} + \ldots + {}^nC_t.P_e^{\,t}.(1\text{-}P_e)^{n-t}\}$$

$$\mathbf{= 1 - \sum_{i=0}^{t} {}^nC_i.P_e^{\,i}.(1\text{-}P_e)^{n-I}} \tag{3-32}$$

If $\mathbf{P_{new}}$ is the BER of the array after using ECC, then P_{new} is the **target BER of the array**. P_{new} can be computed by solving the following relation:

$$\boxed{\mathbf{1 - \sum_{i=0}^{t} {}^nC_i.P_e^{\,i}.(1\text{-}P_e)^{n-i} = 1 - (1 - P_{new})^k}} \tag{3-33}$$

Thus, the required error correction capacity of an error correction code is a function of the current bit error rate and the target bit error rate. [31] also uses a similar relation to compute the improved bit error rate after using a single bit correcting Hamming code on a 512-bit NOR block.

There are two main choices for an error correction code for low latency Flash memory. [36] also lists these as possible methods to improve Flash device reliability. For lower bit error rates of the order of 10^{-12} a single error correcting Hamming code is a viable choice because its error correction capacity suffices at low BER levels such as these and the Hamming decoding algorithm involves very simple math which may be implemented using XOR gates. For higher

BER in high density devices today (~10^{-7}), multiple error correction is required. The BCH code is a good option. Although it is mathematically more involved than the Hamming algorithm, there are relatively simpler BCH decoding algorithms which have simple implementations [21-23]. Massey's step-by-step decoding algorithm is one example. The required error correction capacity of the code can be computed as a function of the current array bit error rate and the expected or target bit error rate. These algorithms are used as a starting point for optimizing methods to ensure reliability of data in a Flash device.

CHAPTER 4

IMPLEMENTATION AND RESULTS

Presently NOR Flash memory have an on-chip Hamming code implemented in hardware. These codes suffice when bit error rates are of the order of 10^{-12}. However for technology nodes 45nm and 32nm the raw BER is in the range 10^{-7} to 10^{-11}. This necessitates multi bit error correction in the memory array. From Equation 3-33, it can be shown that 2 bit error correction on a 256-bit or smaller block of data helps achieve the required target BER of 10^{-15}. The Hamming and BCH algorithms studied earlier serve as a good starting point. These algorithms have been combined and modified to obtain architectural schemes which have latencies < 10ns and a high error correction capacity making them suitable for NOR Flash.

4.1. Error Correction Architectures for NOR Flash

Tradeoffs between latency, error correction capability and complexity differ for Hamming codes and BCH codes. Considering that higher order errors are less likely, combinations of these two codes are investigated further to take advantage of simple fast algorithms for more common single bit errors and using slower stronger algorithms for less likely higher-order errors in order to minimize the performance impact on XiP NOR.

The proposed optimized architectures are summarized in Table 4.1.

The decoding algorithms for each of these architectures are discussed briefly here. A mathematical explanation is covered in Chapter 3.

4.1.1. Single Bit Hamming Code

Hamming codes (Section 3.1) can detect and correct single bit errors and only detect two bit errors. These are the simplest block codes and have been the primary choice for

an ECC in Flash memory until now. However they are inefficient in improving data reliability in high BER MLC Flash devices.

Figure 4.1 illustrates the decoding process for a single bit ECC Hamming code. Section 3.1.2 explains the encoding. A part of the decoder circuit is used for encoding.

Table 4.1. Summary of optimized architectures for latency-constrained Flash systems

Code	Key Concept
Single bit Hamming code	▪ 1 bit ECC. ▪ Adding even parity to certain data groups.
Dual bit Hamming code	▪ 2 bit ECC ▪ Limiting error possibilities to data bits only by duplicating Hamming parity.
BCH code	▪ ≤ 2 bit ECC ▪ Oversampling data to add redundancy.
Hierarchical BCH	▪ ≤ 2 bit ECC ▪ Performance improvement by using BCH to correct multi bit errors and Hamming

The Decoding Algorithm (Section 3.1.3) for a single bit ECC Hamming code is explained below:

The primary step is to determine if an error has occurred or not. A non-zero output from the syndrome computation block indicates a single bit error in either the data or its parity. (This block helps determine parity bits when used for encoding). If there is no error, then the decoding process comes to an end. However, if the syndrome is not zero, the error pattern corresponding to the syndrome pattern is determined. This is bit pattern is added (modulo-2) to the input bits to obtain the corrected output vector.

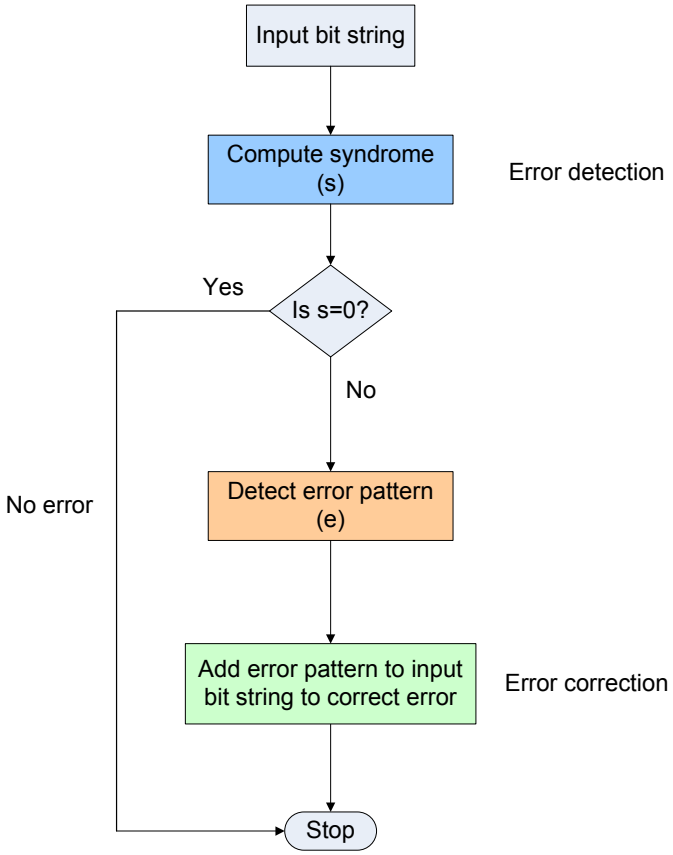

Figure 4.1. Single bit Hamming decoding algorithm

It is assumed that there is at most a single bit error in the input. The syndrome may be nonzero for higher number of errors as well and may correspond to an incorrect error pattern. In this case, an input bit may be erroneously ‘corrected’.

Figure 4.2 shows the block diagram of a Hamming decoder. It has been color coded with respect to the flow diagram (Figure 4.1) to depict the functionality of the blocks.

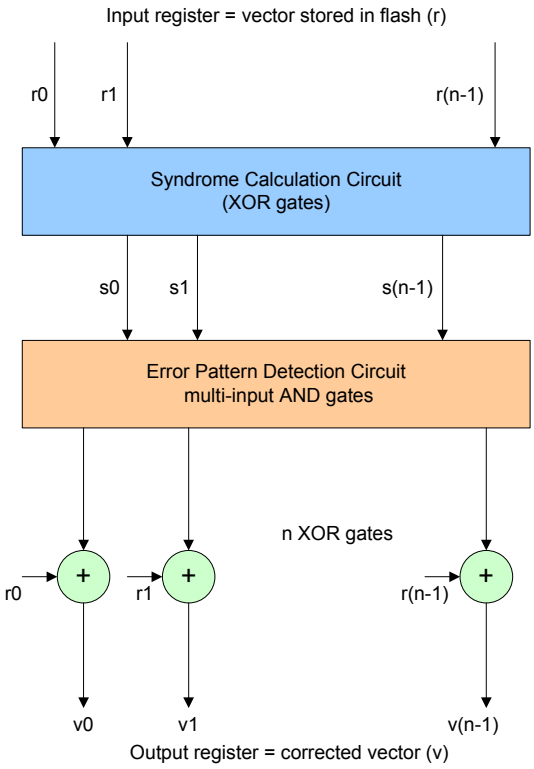

Figure 4.2. Hamming decoder block diagram

A Hamming code is the simplest to understand and the easiest to implement single bit correction code. However, it does not suffice for bit error rates as high as 10^{-6}. Complex codes can take care of higher number of errors; however, it will be highly beneficial if Hamming codes can be modified to provide more than single bit error correction. This way the simple implementation of Hamming codes can be exploited to provide stronger protection.

4.1.2. Dual Bit Hamming Code

A Hamming code has a minimum Hamming distance of 3. This allows only single bit error detection and correction. If the distance is increased to 4, double bit error detection only is possible. However, it can be shown that for a very small block size (4 or 7 data bits, i.e (8, 4) or

(12, 7) codes) 2 bit errors can be detected and corrected. (The additional parity bit is for 2-bit error detection).

The encoding process is the same as for a Hamming code. The decoding process is shown in Figure 4.3.

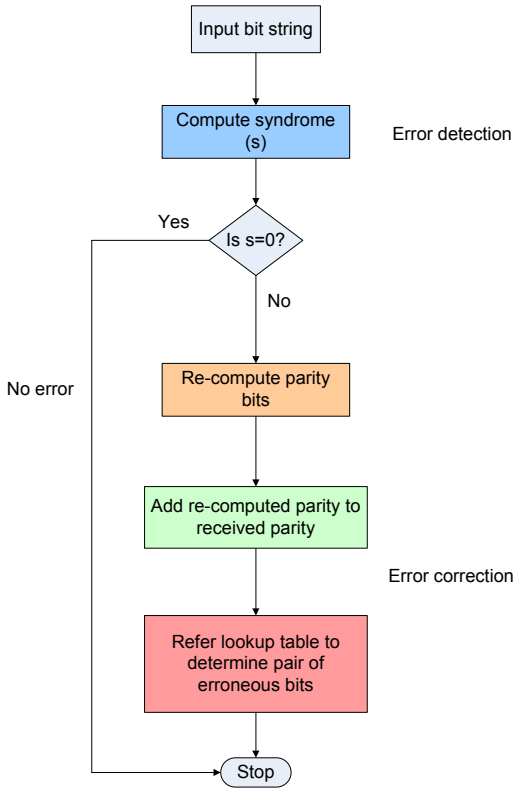

Figure 4.3. Dual Bit Hamming Code Flow Diagram

The Decoding Algorithm for 2-bit error correction using Hamming codes:

Before starting the decoding process, the presence of errors should be detected. The syndrome computation block does this. The number of errors detected is also important because the correction process differs for single and double bit errors. The error correction process for correcting a single bit error has been described in Section 3.1.3. For a double bit error, parity bits are recomputed using the disturbed (received) data. The recomputed parity is

added (modulo-2) to the received parity. The bit string obtained from this operation is compared against a lookup table (Table 4.2) to determine the double bit error pattern.

This algorithm works correctly only under the assumption that errors occur in data bits alone, not in parity bits. For protecting parity bits, two Hamming codes should be used in tandem. But the resultant overhead is unacceptable.

Applicability to small block sizes only

This scheme is applicable only to (7, 4) and (11, 7) Hamming codes.

Consider a (7, 4) code. The number of data bits is 4 and the number of parity bits is 3. Therefore the total number of possible 3-bit patterns (on adding received and recomputed parity) is 2^3 or 8 (including the all-zero pattern). For a 4-bit data block, the number of data bit pairs is ${}^4C_2$ which is equal to 6. It has been observed that a one-to-one correspondence exists between erroneous data pairs and 3-bit patterns obtained by adding received and recomputed parity (Table 4.2). (The all-zero pattern indicates no error!). This correspondence is independent of the values of the data bits themselves.

Say, data = (d_1 d_2 d_3 d_4) (MSB)

parity = (p_1 p_2 p_3) (MSB)

A similar relation can be found for a (11, 7) code. For larger data block sizes, the number of data bits is far larger than the number of parity bit combinations. Hence there cannot be an injective relation between the two. Therefore this scheme fails.

This error control scheme has an overhead of nearly 30-50%. Therefore it can be applied only in very small but sensitive areas of the memory device, for example, the Flash File System (FFS). The FFS is a small percentage of the entire memory. Therefore, although the absolute overhead is very high, it is still a very small number with respect to the storage capacity of the entire chip.

Table 4.2. Lookup table for erroneous data bit pairs and corresponding 3-bit pattern for (7, 4) Hamming code

Erroneous data bits pair	Sum of parity
(d_1, d_2)	011
(d_1, d_3)	101
(d_1, d_4)	001
(d_2, d_3)	110
(d_2, d_4)	010
(d_3, d_4)	100

Thus, Hamming codes for very small block sizes can provide 2-bit error correction. The implementation is very simple. However, due to the extremely high overhead it finds limited applicability. This calls for a need to have algorithms which will be effective for multi-bit error correction in the entire array. The tradeoff would be obviously, higher complexity and higher latency.

4.1.3. BCH Code

Hamming codes find limited applicability as BER increases. Stronger algorithms are required to maintain data reliability. BCH codes are a good choice because they can be designed for any level of error correction.

The encoding process is explained in detail in (Section 3.2.2). Encoding takes place when data is written into memory, and need not be done during the write cycle. As a result latency is not a tight constraint for the encoder. So the BCH encoder may work satisfactorily

even if its design is not highly optimized. On the other hand, decoding always occurs during the read cycle. As mentioned earlier, there is a latency constraint of 10ns on the decoder. The decoder architecture described in this section has been optimized for latency (and area, in case of a hardware only implementation).

It was found (Section 3.2.3) that the step-by-step decoding algorithm is one of the most optimum ones for application in XiP NOR. (Section 3.2.3.2) provides a complete mathematical analysis of the algorithm. It has been explained graphically in

The Decoding Algorithm (two-bit error correction) using BCH codes

As for the previous algorithms, the first step is to check for errors. This is done using a syndrome computation block. An all-zero syndrome indicates there is no error, or an undetectable error. If the syndrome is non-zero, determinants L_1 and L_2 are computed.

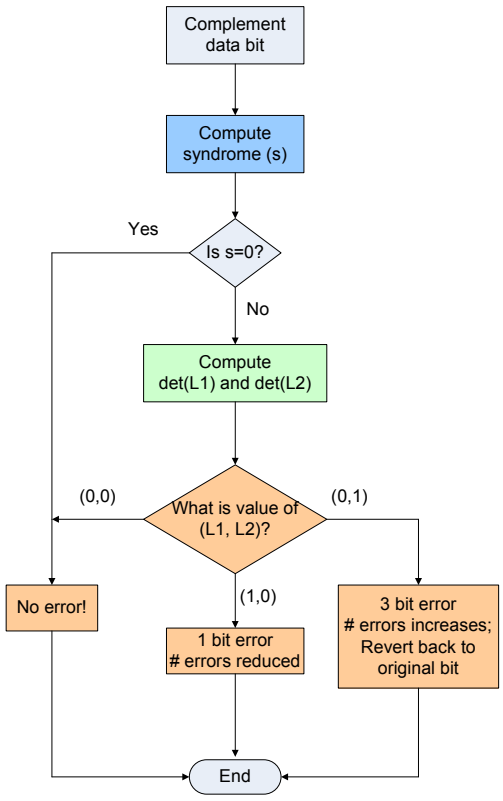

Figure 4.4. Step-by-step BCH decoding algorithm for 2-bit error correction

An assumption is made that at most two bits may be disturbed. The singularity/non singularity of each determinant helps determine the initial number of erroneous bits. Each data bit complemented at a time and the total number of errors are checked again. If the number of errors decreases, then the complemented bit was in error and has been corrected by complementing it. On the other hand, if the total number of errors increases, it implies an error was introduced upon complementing the data bit. Hence it is reverted back to its original value. This step is executed until the number of errors is zero or until all data bits have been checked. This process can be done in parallel using one decoding engine for each data bit. (Correcting parity bits is not required). As a result the entire decoding process executes in one cycle.

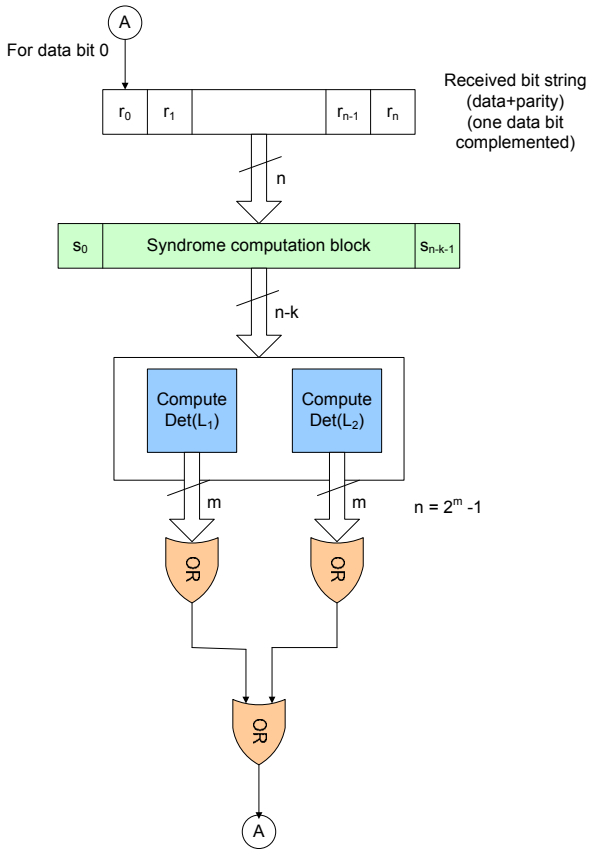

Figure 4.5. Block Diagram of Massey's step-by-step BCH Decoding Algorithm

This algorithm can be generalized for a *t*-error correcting code depending upon the required error correction capability for a given block size to maintain a BER of 10^{-15}. As the number of bits to be corrected and data block size increases the algorithm becomes more complex, especially the determinant computation block.

The likelihood of higher order errors although nonzero is a small number. Therefore, the strengths of Hamming code and BCH code can be combined if there is a scheme which executes the simple and fast Hamming code for lower order errors and the more complex BCH codes for less likely higher order errors. Such a scheme is elaborated in the next section and its feasibility in terms of overhead and maximum latency (area for hardware implementation) is studied.

4.1.4. Hierarchical BCH

The Hamming code is simple and fast but has minimal error correction capacity while the BCH code is more complex but can be designed to achieve any level of error correction. The hierarchical ECC scheme explained here combines the strengths of both these codes.

The Hamming code is applied on smaller blocks of data. All these small data blocks are together protected by a single BCH code. The BCH code is executed only if any of the Hamming decoders detects a 2-bit error in the data. In the example shown in Figure 4.6, 5 bits of Hamming parity protect a 16 bit data block. Six such data blocks are together protected by a single (128, 96) 4-bit error-correcting BCH code. The BCH decoder executes only if any of the six Hamming codes detect a 2-bit error.

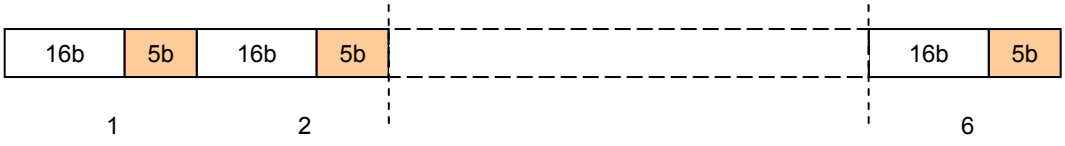

Figure 4.6. Example of Hierarchical BCH code

The error correction capacity of the BCH code is computed using the formula given in (Section 3.3).

The Hierarchical decoding algorithm

Hamming decoders work on small blocks of data, typically 16 or 32b each, correcting possible single bit errors and checking for 2b errors. If a 2b error is detected by any Hamming decoder, the BCH decoder is set into action. This decoder is initialized if a 2b error is detected is any of the smaller Hamming data blocks. Thus the more frequently lower order errors are resolved by a simple and fast Hamming code while the complex and timing intensive BCH code corrects only the less frequent higher order errors. Figure 4.7 explains the operation graphically. The Hamming-BCH hierarchical design combines the strengths of both codes at the cost of additional overhead in terms of data and hardware and higher latency. A sample implementation (Figure 4.8) will help check if these are within acceptable limits.

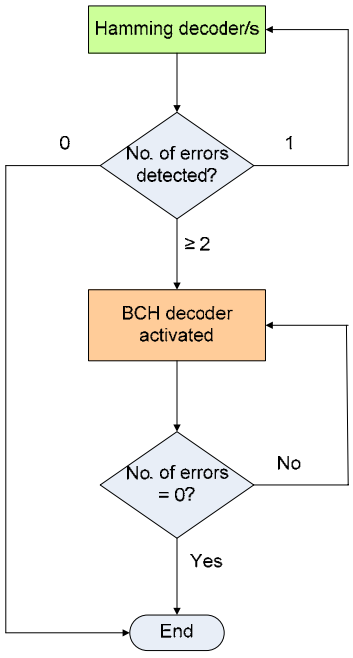

Figure 4.7. Flow Diagram for the Hierarchical BCH Decoding Scheme

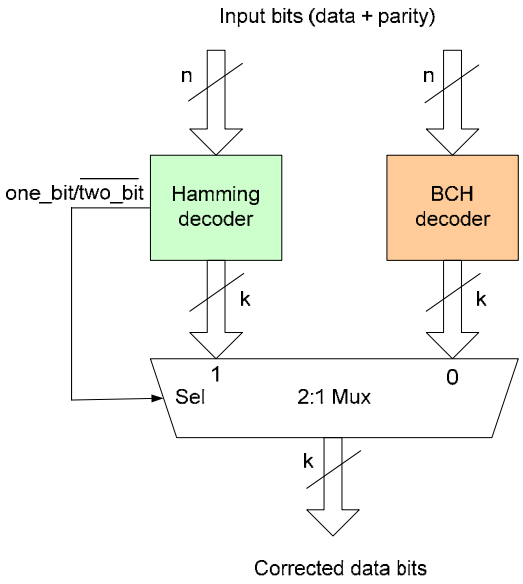

Figure 4.8. Block Diagram of the Hierarchical BCH Decoding Scheme

4.2. Analyzing and Comparing Implementations

The architectures elaborated in the previous section have to be verified for possible applicability in the NOR Flash device. This implies studying the impact of each implementation on latency, RAM footprint (software) and gate count or area (hardware). There are three possible implementation choices (Table 4.3) –

- Software
- Hardware
- Mixed (hardware + software)

The architectures are verified for small block sizes: (7, 4) Hamming code, (15, 7) BCH code and a combination of (11, 7) Hamming code and (15, 7) BCH code for the hierarchical process. Small block sizes make it easy to validate the output. Larger block sizes will alter the parameters only to a tolerable extent.

Table 4.3. Possible Implementation Choices for ECC Architectures

	Single bit Hamming (7, 4)	Dual Bit Hamming (7, 4)	BCH code (15, 7)	Hierarchical BCH HC: (11,7) BCH: (15,7)
Software				
Hardware				
Mixed				

4.2.1. Software Implementation

A software implementation is possible inside the memory chip due to the presence of an on-chip 8051-like microcontroller. This microcontroller operates at a clock frequency of 40MHz.

A C code was written for each algorithm. These codes verify the functionality of the algorithm besides exploring their applicability. Codes were written for (7, 4) Hamming code, (15, 7) BCH code and (11, 7) Hamming and a (15, 7) BCH code for the Hierarchical BCH scheme. An assumption was made that each C instruction takes 1 controller clock cycle to execute. This is a good first order estimate because the approximate cycles per instruction (CPI) for a clock speed of 40MHz is around 1.5 [41]. This means a single instruction takes around 1.5 clock cycles to execute. Besides, there may be branches within a code.

The latency estimates for each of the architectures are shown in Table 4.4.

An assembly code would make the execution at least 10-20x times faster. However, the on-chip 8051-like controller does not necessarily execute one instruction in a single clock cycle, it may take more. Therefore the latencies shown in

Table suggest a good ballpark figure. Since these numbers completely rule out a pure software solution the architectures were not investigated for RAM usage.

Table 4.4. Latencies for Software Implementation of ECC Architectures

	Single bit Hamming (7, 4)	Dual Bit Hamming (7, 4)	BCH code (15, 7)	Hierarchical BCH HC: (11,7) BCH: (15,7)
Software	~40 clocks=1μs	~40 clocks=1μs	~400 clocks = 10μs	~450 clocks = 11.25μs
Hardware				
Mixed				

4.2.2. Hardware Implementation

Each ECC hardware implementation has been studied and optimized for latency and silicon area (or gate count).

There are several possible hardware implementation schemes available for each algorithm [21-23]. The appropriate one was chosen through an elimination process based on rough estimates of latency and gate count deduced from the basic block diagram of the architecture. The calculations were done by hand based on valid approximations (for example, a single register/flipflop stage may be approximated to take up one clock cycle = 10ns). The numbers deduced were expected to be within at least a 20-30% margin of the actual ones. Modifications were made to the chosen architectures to keep the latency within 10ns. The estimates for the chosen architectures are shown in Table 4.5. Table 4.6 shows the estimated latency and gate count for a 256-bit data block.

The architectures singled out based on hand analysis were finally verified by writing a Verilog code for each of these and synthesizing it using Synopsys Design Compiler. The latency and gate count obtained after synthesis are tabulated in . The numbers shown prove the veracity of the estimates.

Table 4.5. Estimated Latency and Gate Count for Hardware Implementation

	Single bit Hamming (7, 4)	Dual Bit Hamming (7, 4)	BCH code (15, 7)	Hierarchical BCH HC: (11,7) BCH: (15,7)
Software	~40 clock cycles = 1μs	~40 clock cycles = 1μs	~400 clock cycles = 10μs	~450 clock cycles = 11.25μs
Hardware	Lat. ~10ns Gates~150	Lat. ~10ns Gates~180	Lat. ~10ns Gates~360	Lat. ~20ns Gates~510
Mixed				

Table 4.6. Estimated Latency and Gate Count for 256b Generic Codes

	Single bit Hamming (265, 256)	Dual Bit Hamming	BCH code (255, 239)	Hierarchical BCH ($n = 2^m - 1$) HC (38, 32) BCH (255, 239)
Software				
Hardware	Lat. ~10ns Gates~1200	N/A	Lat. ~10ns Gates~2500	Lat. ~20ns Gates~4000
Mixed	-	-	-	-

It is clear that a hardware implementation of ECC algorithms will help to improve data reliability without making hefty demands of the bandwidth and silicon area. For the small data blocks considered for synthesis, the overhead in terms of parity bits per data bit is very large. However, the ratio decreases significantly as the data block size is increased keeping the latency and gate count within acceptable limits.

Figure 4.9 shows the gate-level circuit representation for a (7, 4) Hamming code and Figure 4.10 shows the circuit for a (15, 7) BCH code after synthesizing the respective Verilog codes.

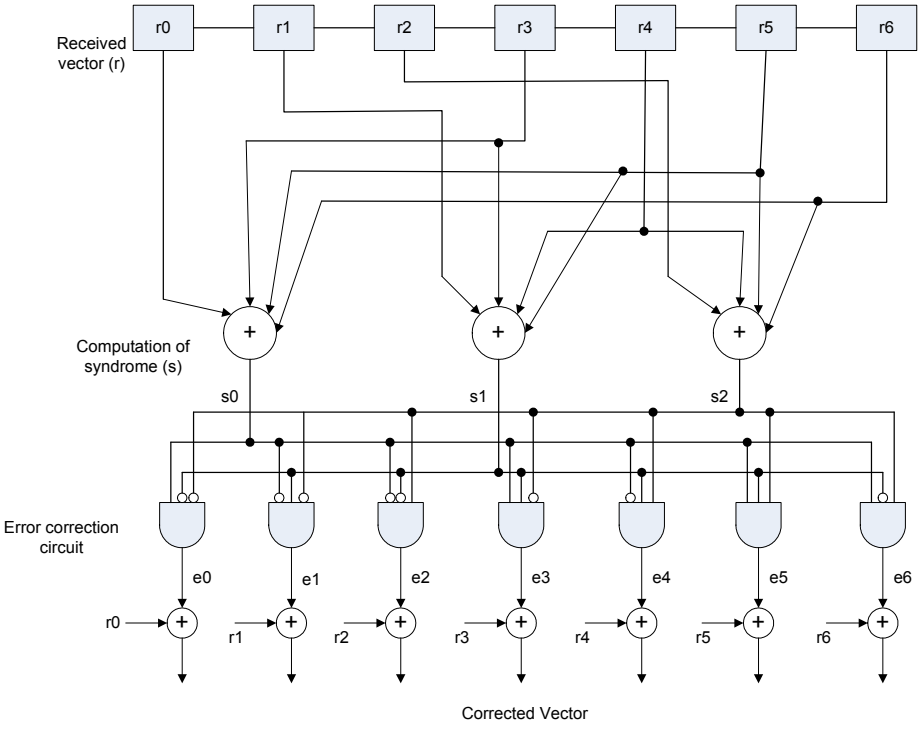

Figure 4.9. Gate Level Circuit for a (7, 4) Hamming Code

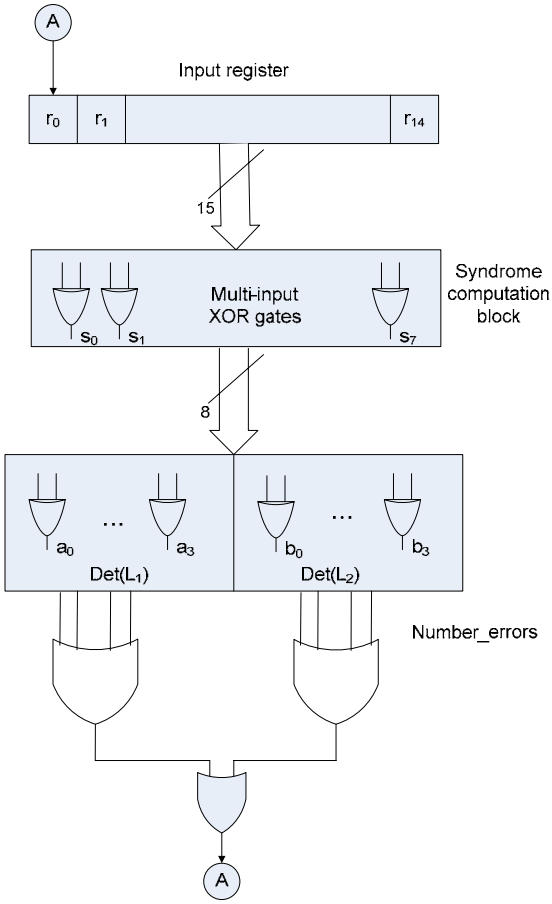

Figure 4.10. Gate Level Circuit for a (15, 7) BCH Code.

Table 4.7. Latency and Gate Count for Synthesized Hardware Designs

	Single bit Hamming (7, 4)	Dual Bit Hamming (7, 4)	BCH code (15, 7)	Hierarchical BCH HC: (11, 7) BCH: (15, 7)
Software	~40 clock cycles = 1μs	~40 clock cycles = 1μs	~400 clock cycles = 10μs	~450 clock cycles = 11.25μs
Hardware	Lat. ~3ns Gates~70	Lat. ~4ns Gates~110	Lat. ~7ns Gates~320	Lat. ~8ns Gates~450
Mixed	N/A	N/A	N/A	N/A

4.2.3. Mixed Implementation

A mixed implementation typically employs hardware for timing sensitive tasks and uses software for the variable blocks in the design. In this case, a pure hardware analysis satisfies both latency and area constraints making it the best choice for an on-chip ECC in NOR Flash. This makes a mixed implementation unnecessary.

It has been shown that a hardware on-chip error correction code satisfies latency and die area constraints for single as well as multi bit error correction using standard ECC algorithms and optimized schemes using the standard algorithms.

CHAPTER 5

SUMMARY AND CONCLUSIONS

The aim of this thesis is to develop error correction methods for latency-constrained Flash systems.

As a first step towards this goal, it was necessary to extract error probabilities of a NOR Flash array from technology-specific threshold voltage data. This was not done earlier simply because it was not important to precisely determine the error correction capacity that is required to achieve a certain target bit error rate. Earlier Flash arrays had a raw BER on the order of 10^{-12} for which a single bit Hamming proves to be sufficient. The BER derived from the proposed mathematical relations was a good starting point to determine correction requirements for the memory array. The next step was to review existing error correction algorithms with respect to NOR Flash requirements, namely, low latency (< 10ns) and low hardware gate count (< 5000 NAND gates). Applying these constraints to existing algorithms and their architectural possibilities brought several shortcomings to light. For example, Hamming codes were easy to implement but could not help with higher order bit error rates. On the other hand, BCH codes had excellent correction capacities but very complex implementations. This lead to efforts to optimize existing architectures and develop new schemes concentrating on the strengths of Hamming and BCH codes.

One such optimization is the dual bit Hamming code. This code gives 2-bit error correction using the simple Hamming algorithm for block sizes less that 1 byte in length. The small block sizes are suitable for applying error correction in Flash File Systems which have a read granularity of 1 byte. Flash File System is a very small but important block on the Flash chip. Therefore it is important to maintain its reliability. This cannot be done using the same block codes which are used in the memory array because of the large block sizes that are typically used in the array (256 bits).

It has been shown that the BCH code is very useful for correcting multiple bit errors with a latency of around 10 ns which is suitable for XiP application. The hardware complexity of this implementation is around 3500 NAND gates which is well within the margin set for NOR Flash. It is important to note that lower order errors (1 bit errors) occur at least 10^{12} times more frequently than higher order errors (2 bit errors). Therefore the BCH code is mostly correcting single bit errors with rare exceptions. The Hierarchical BCH code overcomes this problem by having the Hamming code perform error correction on the lower order errors (single bit ECC) and the BCH code being executed only once out of approximately 10^{12} times when there is a 2 bit error. This results in an average latency equal to the latency of the Hamming code (3 – 4ns) along with multiple error correction capacity. This algorithm is expected to be very useful in Flash systems like LPDDR2 (32nm) which has a read access time of 50ns. For such a low read access time even a latency of 10ns proves to be a huge penalty making the Hierarchical scheme an implementation of choice. The applicability of these optimizations was proved in simulations and via implementations in both hardware and software. The software implementation, although not applicable proved the functionality of the algorithms while the hardware implementation gave a gate count well within the 5000 NAND gates constraint.

The proposed and analyzed algorithms take error correction in NOR Flash a step ahead from single bit correction to 2 bit correction at a minimal latency (< 10ns) and hardware overhead (< 5000 NAND gates). The software and hardware simulations proved that the proposed solutions provide at least 2x improvement in protecting data in NOR Flash arrays. Array bit error rates of the order of 10^{-7} for XiP Flash can be brought down to 10^{-15} using 2-bit error correction.

Low latency, low complexity error correction architectures make it possible to have reliable high storage density Flash systems at smaller geometries.

APPENDIX A

HOW TO COMPUTE MINIMAL POLYNOMIALS

For any element β in the field GF(2^m), the corresponding *minimal polynomial* Φ(X) is the polynomial of smallest degree over GF(2) such that Φ(β) = 0. Φ(X) is unique. For example, the minimal polynomial of 0 is X and that of 1 is X + 1.

Before learning to compute minimal polynomials it is essential to learn the following theorem.

Theorem: If *f*(X) is a polynomial having coefficients in GF(2) and β is an element in the extension field GF(2^m) such that β is a root of *f*(X), then for any $j \geq 0$, $(\beta)^{2^j}$ is also a root of *f*(X).

$$[f(\beta)]^{2^j} = f(\beta^{2^j})$$

The element β^{2^j} is called a *conjugate* of β.

The general equation for the minimal polynomial Φ(X) of β in GF(2^m) is given as;

$$\Phi(X) = \prod_{i=0}^{e-1} \{X + (\beta)^{\wedge}(2^{\wedge}i)\}$$

Where *e* is the smallest integer such that $\beta^{2^e} = \beta$.

Example:

Consider GF(2^4). Let $\beta = \alpha^1$. The conjugates of β are

$$\beta^2 = \alpha^2,\ \beta^{2^2} = \alpha^4, \qquad \beta^{2^3} = \alpha^8 \text{ and } e = 4 \text{ (since } \beta^{2^4} = \alpha^{16} = \alpha)$$

Substituting in the general equation, the minimal polynomial for $\beta = \alpha^1$ is computed as;

$\Phi(X) = (X + \beta)(X + \beta^2)(X + \beta^4)(X + \beta^8)$

$= (X + \alpha)(X + \alpha^2)(X + \alpha^4)(X + \alpha^8)$

$= X^4 + X^3(\alpha^8 + \alpha^4 + \alpha^2 + \alpha) + X^2(\alpha^{12} + \alpha^{10} + \alpha^9 + \alpha^6 + \alpha^5 + \alpha^3) + X(\alpha^{14} + \alpha^{13} + \alpha^{11} + \alpha^7) + \alpha^{15}$

$= X^4 + X + 1$ (Table below)

Representation for the elements of $GF(2^4)$ generated by $(1 + X + X^4)$

Power representation	Polynomial representation	4-Tuple representation
0	0	0000
1	1	1000
α	α	0100
α^2	α^2	0010
α^3	α^3	0001
α^4	$1 + \alpha$	1100
α^5	$\alpha + \alpha^2$	0110
α^6	$\alpha^2 + \alpha^3$	0011
α^7	$1+ \alpha + \alpha^3$	1101
α^8	$1+ \alpha^2$	1010
α^9	$\alpha + \alpha^3$	0101
α^{10}	$1 + \alpha + \alpha^2$	1110
α^{11}	$\alpha + \alpha^2 + \alpha^3$	0111
α^{12}	$1+ \alpha + \alpha^2 + \alpha^3$	1111
α^{13}	$1 + \alpha^2 + \alpha^3$	1011
α^{14}	$1 + \alpha^3$	1001

REFERENCES

1. *MirrorBit Technology – The Future of Flash memory is here today*,

Available at <http://www.spansion.com/flash_memory_technology/mirrorbit.html>

2. M. Janai, B. Eitan, A. Shappir, E. Lusky, I. Bloom, and G. Cohen, *Data Retention Reliability Model of NROM Nonvolatile Memory Products*, IEEE Transactions on Device and Materials Reliability, Vol.4, No.3, Sept. 2004, pp. 404-415.

3. B. Eitan, P. Pavan, I. Bloom, E. Aloni, A. Frommer, and D. Finzi, *Can NROM, a 2-bit trapping storage cell, give a real challenge to floating gate cells?*, Proc. SSDM, 1999, Tokyo, Japan, Sept. 1999, pp. 522-524.

4. B. Eitan, P. Pavan, I. Bloom, E. Aloni, A. Frommer, and D. Finzi, *NROM: A Novel Localized Trapping, 2-bit non-volatile memory*, IEEE Electron Device Lett., Vol. 2, No. 11, Nov. 2000.

5. *Threshold Voltage*, Wikipedia, Nov. 2007,

Available at: < http://en.wikipedia.org/wiki/Threshold_voltage>

6. T. C. Ong *et al.*, *Erratic erase in ETOX flash memory array*, VLSI Technology Symp., 1993, p. 83.

7. S. Yamada *et al.*, *Non-uniform current flow through thin oxide after Fowler-Nordheim current stress*, Proc. Int. Reliability Physics Symp., 1996, pp.108.

8. S. Satoh, G. Hemink, K. Hatakeyama, and S. Aritome, *Stress-induced leakage current of tunnel oxide derived from flash memory read-disturb characteristics*, IEEE Trans. Electron Devices, Vol. 45, pp. 482-486, Feb. 1998.

9. S. Shuto et al., *Read disturb degradation mechanism for source erase flash memories*, IEEE Symp. VLSI Technology Dig. Tech. Papers, 1996, pp.242.

10. M. Kato *et al.*, *Read-disturb degradation mechanism due to electron trapping in the tunnel oxide for low-voltage flash memories*, IEDM Tech. Dig., 1994, p. 45.

11. Atwood, G. *et al*., *Future Direction and Challenges for Etox Flash memory scaling*, IEEE Trans. Device and Material Reliability, vol. 4, no.3, Sept 2004, pp. 301-305.

12. *Floating Gate Transistor*, Wikipedia, May 2008.

Available at <http://en.wikipedia.org/wiki/Floating-gate_transistor>

13. *Flash memory*, Wikipedia, June 2008.

Available at < http://en.wikipedia.org/wiki/Flash_memory>

14. *Bayesian Inference*, Wikipedia, June 2008.

Available at < http://en.wikipedia.org/wiki/Bayesian_analysis>

15. Lin, S. and Costello, D., Jr., *Error Control Coding: Fundamentals and Applications*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1983.

16. Sweeney, P., *Error Control Coding: From Theory to Practice*, John Wiley & Sons, Ltd., England, 2005.

17. Meggitt, J., *Error Correcting Codes and their implementation for data transmission systems*, IEEE Trans., IT-7, vol. 4, pp. 234-244, October 1961.

18. Peterson, W.W., *Encoding and Error-Correction Procedures for the Bose-Chaudhuri codes*, IRE Trans. Inf. Theory, IT-6, pp.459-470, September 1960.

19. Chien, R.T., *Cyclic decoding procedures for Bose-Chaudhuri-Hocquenghem codes*, IEEE Trans., IT-10, pp. 357-363, 1964.

20. Massey, J.L., *Step-by-Step Decoding of the Bose-Chaudhuri-Hocquenghem Codes*, IEEE Trans., IT-11, pp. 580-585, October 1965.

21. Wei, S. and Wei C., *High-speed hardware decoder for double-error-correcting binary BCH codes*, IEE Proc., Vol. 136, pp. 227-231, June 1989.

22. Wei, S. and Wei, C., *A High-Speed Real-Time Binary BCH Decoder*, IEEE Trans. On Circuits and Systems for Video Technology, Vol. 3, no. 2, April 1993.

23. Che, C., Su S. and Wei, S., *New Step-by-Step Decoding for Binary BCH Codes*, The Ninth Int. Conf. on Comm. Sys., pp. 456-460, September 2004.

24. Szwaja, Z., *On step-by-step decoding of BCH binary code*, IEEE Trans. Info. Theory, IT-13, pp. 350-351, 1967.

25. *Cyclic Redundancy Check*, Wikipedia, 6 Dec. 2007,

Available at: < http://en.wikipedia.org/wiki/Cyclic_redundancy_check>

26. Wozencraft, J.M. and Reiffen, B., *Sequential Decoding*, MIT Press, Cambridge, Mass., 1961.

27. Massey, J.L., *Threshold Decoding*, MIT Press, Cambridge, Mass., 1963.

28. Dowla, F., ed., *Handbook of RF and Wireless Technologies*, Elsevier, 2004.

Available at: <books.google.com>, pp. 377.

29. Shannon, C., *Communication in the presence of noise*, Reprinted in the Proc. of the IEEE, Vol. 86, No. 2, Feb. 1998.

30. Mackay, D., *Information Theory, Inference, and Learning Algorithms*, Cambridge University Press, 2003.

31. Ou, E. and Yang W., *Fast Error Correcting Circuits for Fault Tolerant Memory*, Intl. Workshop on Mem. Tech., Des., and Testing (MTDT), pp. 8-12, 2004.

32. T. Tanzawa, T. Tanka, K. Takeuchi, R. Shirota, A. Aritome, H. Watanabe, G. Hemink, K. Shimizu, S. Sato, Y. Takeuchi and K. Ohuchi, *A Compact On-Chip ECC for Low Cost Flash Memories*, IEEE J. Solid-State Circuits, vol. 32, pp. 662-669, May 1997.

33. F. Sun, S. Devarajan, K. Rose and T. Zhang, *Multilevel Flash Memory On-chip Error Correction based on Trellis Coded Modulation*, IEEE Symp. On Circuits and Systems, pp. 1443-1446, May 2006.

34. S. Gregori, O. Khouri, R. Micheloni and G. Torelli, *An Error Control Code Scheme for Multilevel Flash Memories*, IEEE Int. Workshop on Memory Tech., Des. & Test., pp. 45-49, 2001.

35. C. Winstead, *Analog Soft Decoding for Multi-level Memories*, Proc. of the 35th Int. Symp. on Multiple-Valued Logic, pp. 132-137, 2005.

36. S. Gregori, A. Cabrini, O. Khouri and G. Torelli, *On-Chip Error Correcting Techniques for New-Generation Flash Memories*, IEEE Proc., vol. 91, pp. 602 – 616, April 2003.

37. S. Gregori, P. Ferrari, R. Micheloni and G. Torelli, *Construction of Polyvalent Error Control Codes for Multilevel Memories*, 7th IEEE Int. Conf. on Electronics, Circuits and Systems, vol. 2, pp. 751 – 754, 2000.

38. H. Chang, C. Lin, T. Hsiao, J. Wu and T. Wang, Multi-level Memory Systems using Error Control Codes, Int. Symposium on Circuits and Systems, vol. 2, pp. II-393 – 396, May 2004.

39. C. L. Chen and M. Y. Hsiao, *Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review*, IBM J. Res. Develop., vol. 28, pp. 124 – 134, March 1984.

40. B. Benjauthrit, L. Coady and M. Trcka, *An Overview of Error Control Codes for Data Storage*, Intl. NonVolatile Memory Technology Conf., pp. 120 – 126, 1996.

41. *Cycles Per Instruction*, Wikipedia, June 2008.

Available at <http://en.wikipedia.org/wiki/Cycles_Per_Instruction>

BIOGRAPHICAL INFORMATION

Priyanka is currently pursuing a Master's degree in Electrical Engineering at the University of Texas at Arlington. Her research interests include hardware architecture and digital circuit design. She has been an intern in the Product Development and Systems Engineering group at Spansion Inc. since May 2007. She has also worked on projects at the Tata Institute of Fundamental Research and the Bhabha Atomic Research Centre in India. She enjoys reading autobiographies and biographies, and when she is not reading she likes to trek, hike, run or walk.